An extendible MIPS–I processor kernel in VHDL for hardware/software co-design

Michael Gschwind, Dietmar Maurer

{mike,dm}@vlsivie.tuwien.ac.at

Technische Universität Wien
Treitlstraße 1–182–2
A–1040 Wien
AUSTRIA

Abstract

This paper discusses the design of a MIPS-I processor kernel using VHDL. The control structure of this processor is distributed, with a small controller in each pipeline stage controlling sequencing of operations and communication with adjacent pipeline stages. Instruction flow management is performed using asynchronous communication signals. Due to its high-level description and distributed control structure, the kernel can easily be extended. Thus, instruction set extension hardware/software co-evaluation can be performed efficiently using rapid prototyping.

1. Introduction

To optimize processor performance to a particular application, one approach of hardware/software co-design is instruction set extension. To improve performance on any particular problem, many instruction sets have been proposed. While most proposals thoroughly evaluate the impact of proposed instruction set extensions on the software, the impact on hardware design is rarely evaluated. In this work, we present a processor kernel to support *hardware/software co-evaluation* of instruction set extensions.

Depending on the proposed instruction set extensions, an implementation may either be too large to be implemented economically, or the extensions may slow down the processor so that any performance effects of an optimized ISA may be lost. To evaluate hardware effects of instruction set extensions, we have designed an extendible RISC processor architecture. By implementing proposed instruction set extensions, hardware aspects of the proposed extensions can be evaluated.

Adapting an instruction set to a particular problem is a difficult task, as many unknown problems have to be explored. To achieve overall optimization of program performance, the execution time equation has to be optimized [4]

$$\textit{execution time} = \frac{\textit{instructions}}{\textit{program}} * \frac{\textit{cycles}}{\textit{instruction}} * \frac{\textit{seconds}}{\textit{cycle}}$$

Due to the many factors involved in performance optimization, suggested optimization solutions often minimize only the number of instructions necessary to solve a problem, or at best the number of cycles.

However, many of the suggested special purpose instructions are complex. As a result, it may not be possible to clock an extended processor at the same frequency as the original design. This is often neglected by studies, as the processor characteristics can be difficult to predict, and as a full implementation of a processor is often out of scope. Thus, studies most often use software simulators such as SPIM [10] to predict performance. While these instruction set emulators can be used to test software, generate traces and gather statistics, they do not allow to predict the effects of the extended instruction set architecture on the processor design itself.

To investigate hardware/software co-evaluation of various instruction set extensions, we have decided to implement an extensible MIPS-I architecture kernel [6]. This kernel gives us the possibility to study the effects of extended instruction set architectures on processor speed and implementation area.

For the processor to be useful for these purposes, we identified the following requirements:

high-level description The format of the processor description should be easy to understand and modify.

modular To add new instructions, only the relevant parts should have to be modified. A monolithic design would make experiments difficult.

extendible All data structures and interfaces should be designed such that new fields can be added with ease.

synthesizable The processor description should be synthesizable to derive actual hardware implementations.

This work is organized as follows: we discuss the implementation of the MIPS-I processor in section 2. We describe test and validation of the processor description in section 3, and report synthesis results in section 4. Various applications for the presented model are given in section 5, and we discuss related work in section 6. We draw our conclusions in section 7.

2. Implementation

2.1. Design Approach

To make the design easy to modify and adaptable to various purposes, we decided to use a high-level description. While a low-level physical or standard-cell based layout may have higher performance, this design would be harder to modify. Added instructions would require either total redesign, or would achieve sub-optimal results by being forced to fit into a design not optimized for the new functionality.

Thus, we decided to describe the processor in a hardware description language and use synthesis tools generate the actual layout. For architecture studies, a high-level description approach is advantageous, as it is easier to modify and adapt to accommodate new instructions to be studied. Also, the effects of various implementation decisions, such as whether to use a ripple-carry adder or a carry look-ahead architecture can be studied by postponing these implementation decisions to the synthesis step.

The base MIPS processor is designed using VHDL as description language and the Synopsys Design Compiler [16] as synthesis tool [11]. To ensure portability and simplify maintenance of the code, we have developed the VHDL processor description following the ESA VHDL modeling guidelines [15].

2.2. Architecture

An easily modifiable description is necessary, but not sufficient. To experiment with different extensions, the structure of the base design has to provide mechanisms to add instructions easily, and keep modification to a minimum and as localized as possible. The overall processor framework should remain unmodified, and should be capable of integrating additional instructions. Not only additional ALU instructions should be handled, but all types of instructions,

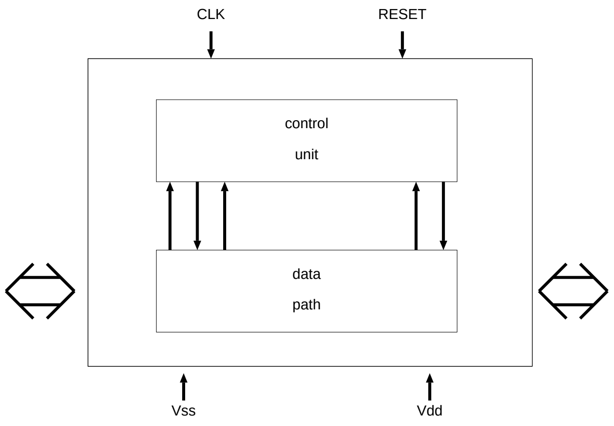

Figure 1. Control unit in a conventional processor design: the data path and the control unit are separate blocks. A single, monolithic control unit controls the entire data path.

such as new control flow mechanisms or novel memory access methods.

In many traditional processors, the control unit is centralized and controls all CPU functions. However, esp. for pipelined architectures, this control unit is one of the most complex parts of the design: the state machine has to take care of introducing pipeline stalls, squashing bubbles, handle exceptions, etc.

Even in “normal” architectures, where the instruction set is fixed before implementation starts, the control unit is one of the most difficult parts of the design. When the architecture is designed to be extendible and the instruction set is not fully defined, designing such a state machine is not feasible. As new instructions are added, they may use an unpredictable number of cycles or have some special synchronization demands that cannot be implemented easily.

In our design, we have decided to break up the unstructured control unit in small, manageable units. Instead of a centralized control unit, which aims to control the complete data path (see figure 1), the control unit is integrated with the pipelined data path. Thus, each pipeline stage is controlled by its own, simple control unit. Overall flow control of the processor is implemented by cooperation of the control units in each stage (see figure 2). This control model is based on communicating state machines using asynchronous control.

Each pipeline stage contains a process modeled in VHDL that stores the current state and determines what actions are to be taken. Each pipeline stage can be in one of three basic states:

WORK The stage executes an instruction.

STALL The stage has computed a result, but cannot pass it to the following stage.

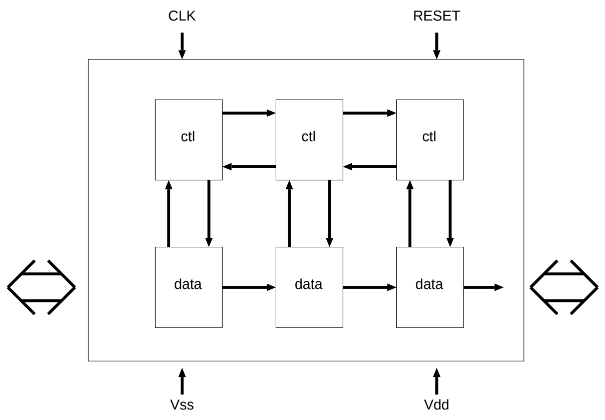

Figure 2. Decentralized control unit: the control unit has been partitioned to reflect the different data path segments. The control units communicate using asynchronous handshake signals to implement the global control functions such as flow control.

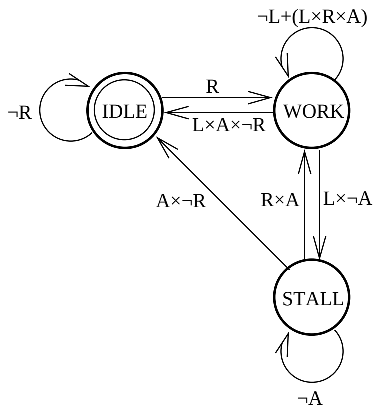

Figure 3. FSM transition diagram for pipeline synchronization. The following conditions are evaluated: last cycle of computation (L), next pipeline stage can accept data (A), previous pipeline stage can supply data (R).

IDLE The stage is idle, i.e., it does not compute any instruction, nor is it stalled.

This basic model is extended to cover the operational requirements of each stage. For example, in the memory access stage MEM, the WORK state is subdivided into states for cache access, main memory access, processor bus busy, cache update and fixup.

The processor pipeline is controlled by cooperation between the state machines present in each pipeline stage. Each pipeline stage is connected to its immediate neighbors, and indicates whether it is able to supply or accept a new instruction. Communication with adjacent pipeline stages is performed using two asynchronous signals, `ready` and `accept`. When a stage has finished processing, it asserts the `ready` signal to indicate that data is available to the next pipeline stage. The next pipeline stage will then indicate whether it is ready to accept these data by using the `accept` signal. The state transitions of a single pipeline stage depending on the asynchronous control signals are shown in figure 3.

Using this approach, the centralized control unit present in most CPUs can be replaced by cooperating state machines (see figure 4). These state machines are easier to debug (since they have fewer states) and extend. Since all data necessary to determine what actions are to be taken next are available in the pipeline stage (current operation status, and synchronization signals from neighboring stages), computing the next state is simple.

This asynchronous, self-arbitrating approach has been proven to be an efficient and elegant solution for data flow problems, but has to be adapted for implementing CPUs [12], [9].

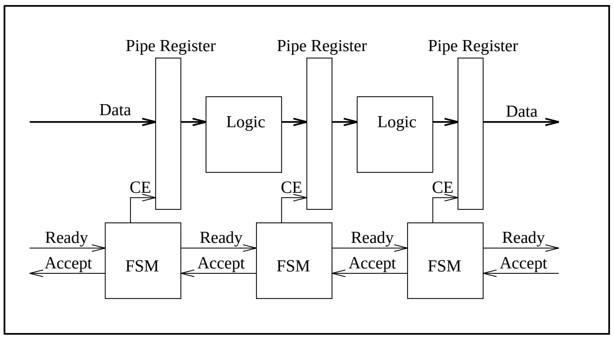

Figure 4. Pipeline synchronization using cooperating state machines.

A drawback of this approach is that synchronization between adjacent state machines has a price. Communication is only possible at the end of a cycle, when the result of the current operation is known. Only then can the next state of the FSM be computed from local information and the state of neighboring pipeline stages. Since information has to propagate from one end of the pipeline to the other at the very end of a cycle, the synchronization mechanism is potentially a critical path.

In a CPU, data flows not only in one direction, information is also passed to earlier pipeline stages. Processor features such as branches, delay slots, forwarding and esp. exception handling require additional coordination. Thus, additional communication and synchronization is needed between non-adjacent pipeline stages.

To implement exception handling, we have implemented an exception controller which collects exception information from all pipeline stages. This controller is responsible for annulling the appropriate pipeline stages and start fetching the exception handler.

3. Test and Validation

To verify our design, we have used traditional verification methods based on test benches. The test bench implements a MIPS processor environment, complete with interrupt controller, read/write buffer, cache, and RAM. We have used this test bench to simulate the processor model both at the behavioral and gate levels. In addition to this test vector based simulation, we have added a high-level graphical verification phase.

The high-level tool RISCview is an X Window System application which displays the internal state of the model. This is performed by parsing traces generated by a VHDL simulator to extract information about the CPU state. The state displayed includes: active machine instructions, the pipeline, the state of each pipeline stage, the register file, and whatever else may be necessary to verify functionality.

This approach has the advantage of being able to trace the operation of the CPU and identify any problems by scrolling through a program and look at the corresponding CPU state. This method does not replace other verification techniques, but reduces the amount of ASCII trace data which has to be scanned to identify problems in the design. The tool can be used with both the original VHDL description and the post-synthesis net list, as the only data used are traces generated by the VHDL simulator.

Most notably, the following functionality can be verified directly using a graphical display of traces:

- synchronization/communication between pipeline stages
- state machines
- basic computation and data flow
- exception handling mechanism

Figure 5 shows a snapshot of the graphical validation tool: the top button bars allow single stepping, tracing, and scrolling through program execution. The main window displays the pipeline state graphically: each pipeline stage is associated with its current FSM state, the instruction executed in the pipeline, and the instruction address. A summary of the register file status and the program code are also given. Active instructions are high-lighted in the program code window.

High-level verification is mainly used to verify functional correctness. Low-level problems (such as timing violations) cannot be verified directly (although many problems can be uncovered because of their impact on program correctness).

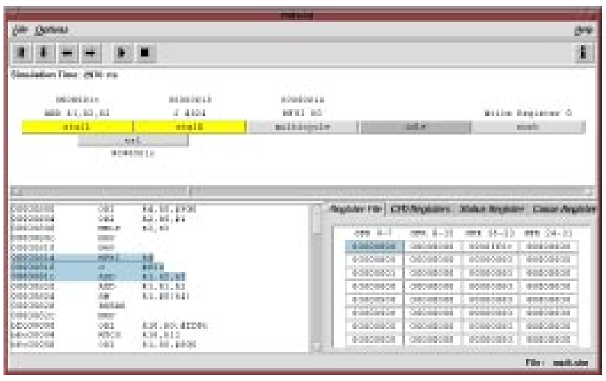

Figure 5. Snapshot of the graphical validation tool: The pipeline displayed executes an instruction in the write-back stage, and the memory stage is idle because the execute stage operates on a multicycle operation. This operation stalls the instruction decode and address translation phases, while the instruction fetch is allowed to complete (the stage is in state CAL, i.e., cache access load).

To ensure correctness of low-level implementation details, interfaces, and timing, we also use traditional simulation techniques with test vectors for ensuring compliance with specifications.

4. Synthesis and Complexity of Design

We have synthesized the VHDL description of the MIPS-I kernel using the Synopsys Design Compiler, and the resulting circuit operates at 10 MHz (i.e., 10 native MIPS) using the LSI 10K library (a single poly, double metal, 1.5μ process). Synthesis is performed bottom-up, using constraints to ensure the optimization of global paths. This is necessary since synchronization between the different FSMs (which is propagated through the entire pipeline at the end of each clock cycle) is one of the critical paths in the design.

By using a bottom-up compilation strategy, all modules can be synthesized in parallel, on different workstations. This speeds up the building of a processor considerably. The simpler top-down strategy which consists of translating the entire design at once required considerable resources: synthesis time was about 5 days on a Sun 10/41, using up to 300 MB of swap space. Also, most synthesis tools can only efficiently synthesize designs with several thousand gates. Our current design exceeds the recommended module size by an order of magnitude!

Table 1 gives the number of source code lines and the gate count for each module of the synthesized design. The register file was not synthesized using Synopsys, as regular memory structures cannot be efficiently generated using logic synthesis. Instead, module generators provided by the

Module	Lines	Gates	Description
AT	394	1008	AT pipeline stage (PC, TLB access)
IF	608	1177	instruction fetch unit
ID	885	1840	instruction decode unit
EXE	726	4393	execution unit (ALU)
MD	411	3414	integer multiply/divide unit
MEM	958	3503	data memory access
WB	149	386	register file writeback
RF	150	N/A	register file
CP0	688	3505	coprocessor 0 (exception handling, TLB)
CCON	229	437	cache controller
R3000	1024	N/A	top level entity (glue logic between modules, declaration of data types, etc.)
Total	6222	19712	MIPS-I kernel

Table 1. Overview of the MIPS-I model: this table gives the number of code lines per module, and the gate count for the synthesized design. (This design was compiled using Synopsys (v3.3a) and targeted at the LSI 10K library.)

ASIC vendor should be employed.[1] Also, the CP0 coprocessor gate count does not include the register file necessary to implement the TLB.

As expected, the critical path of the design is the synchronization of the pipeline stages. This path begins when the result of a memory operation is known at the end of a cycle in the MEM stage, and has to propagate to through the EXE and ID stages to the program counter in the AT stage.

Other tight paths are the branch decisions in the EXE stage, the processing of the cache miss signal in the MEM FSM and the exception handling mechanism which has to collect data from all pipeline stages, and issue the appropriate discard signals to squash pipeline operations.

To optimize the communication paths between state machines, state machine extraction can be used to generate a single centralized state machine from the state machines present in each function block. Since all state machines operate with the same clock, the decentralized communicating states machines can also be viewed as a single state machine where the state vectors of all state machines are concatenated to form a single state register.

The concatenated state machine can then be optimized to use efficient state encoding and minimize the logic to implement state transitions. Thus, the multiple layers of combinational logic distributed across several modules can be optimized to just a few layers. The resulting monolithic state machine is similar to the centralized control units in other designs, but in the source level description, this machine is broken down into several cooperating modules.

[1]Using Synopsys synthesis, the register file would translate to about 14000 gates!

To optimize the control logic, we have tried to automatically generate a single state machine from the state machines found in each pipeline stage and driven by the same clock. This optimization would allow reduction of the communication overhead by global state assignment and optimization, while maintaining the advantages of a modular structure at the source level. However, due to limitations of the Synopsys state machine optimization tools, this has not been possible.

For comparison, the original 2μ MIPS R2000 was designed using custom design techniques and then shrunk to 1.5μ as MIPS R3000. Both the original MIPS parts and our design have been targeted at a single poly, double metal process. The MIPS R2000 part used about 110k transistors for the entire design, with about 30k used for the TLB and register files, for a total of about 80k for the data path. The MIPS R3000 achieved an operating frequency of 16.7 MHz.

5. Model Usage

The presented model can be used for two purposes:

- Evaluation of proposed instruction set extensions.
- Derivation of a prototype implementation of an extended application-specific MIPS processor.

Our MIPS processor model can be used to evaluate how well proposed instruction set extensions integrate in an existing macro-architecture. To allow for new functionality present in extended instructions, the available resources can be parameterized. This includes adding additional function blocks, extending existing function blocks (such as adding additional functionality to the ALU) and introducing new interfaces between blocks.

Our evaluation approach is to extend the base MIPS processor to implement proposed instruction set extensions. Then, both the base processor and the extended model are synthesized to produce an ASIC implementation (using the same compilation parameters). From the resulting ASIC net lists, we extract information such as area and timing.

By comparing the synthesis results of the original MIPS processor with the extended model, we can derive data such as implementation area and maximum cycle time for a processor. However, this data is only valid for a given macro-architecture, as a major change in the architecture may improve the efficiency of a proposed instruction (e.g. by adding an additional pipeline stage to account for the latency of some instructions).

Once evaluation has been completed, a full model of the extended processor is available in synthesizable VHDL. This can be used to generate a final implementation to be used in a computer system.

The proposed model can also be used to derive FPGA prototypes from the synthesizable VHDL description. This allows to run extensive tests of newly implemented instruction set extensions with early prototypes. To facilitate this use, we have investigated the use of high-level optimization techniques for FPGA targets to minimize FPGA resource usage [2].

We have started to evaluate instruction set extensions targeted at improving Prolog program performance [3], which have been developed to support the high-performance VIP Prolog interpreter designed at our university [8].

6. Related Work

Many architecture evaluation and codesign projects have used VHDL at a purely behavioral, non-synthesizable level (e.g., in TOSCA [1]). Usage of VHDL in these models is similar to writing high-level programming language (such as C) models [13].

Software models have a long history to derive software performance data and instruction set usage [5]. While high-level simulators such as SPIM [10] can be used to generate traces to study software effects of proposed instruction sets on software, they do not include enough detail for low-level analysis. By performing a simulation of the internal process structure, low-level simulators modeling the architecture can be used to collect many low-level data such as memory band-width, processor stalls etc. This cycle-by-cycle model deals with the allocation of operations to pipeline stages, but ignores effects on processor speed and implementation size.

The PEAS-I system generates instruction sets from a set of benchmark programs and generates CPU cores described in the SFL hardware description language [14].

Processor implementation has been done mostly to perform validation using special emulators such as the Quickturn RPM system. Koe at al. [7] report that processor emulation on FPGA systems is limited mostly by off-chip interconnect resources, which mirrors our experience.

7. Conclusion

Up to now, hardware/software codesign has been concerned mostly with partitioning of problems where the problem is partitioned between software on a well-known and existing CPU and peripheral support chips. In such environments, hardware/software co-simulation is mainly concerned with validating the interface [13].

We have shown how to apply hardware/software codesign strategies to processor instruction set evaluation. In these environments, the hardware/software interface consists of the instruction set proper, and a full evaluation must include a detailed model of the microprocessor.

We have introduced a synthesizable MIPS-I kernel written in VHDL. This kernel has been designed to be extendible in order to support hardware/software co-evaluation of instruction sets.

References

[1] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto. The role of VHDL within the TOSCA hardware/software codesign framework. In *Proceedings EURO-DAC '94 with EURO-VHDL '94*, Grenoble, France, September 1994. ACM.

[2] M. Gschwind and V. Salapura. A VHDL methodology for FPGA efficiency. In *Proceedings of the European Design and Test Conference ED&TC '96*, Paris, France, March 1996.

[3] M. K. Gschwind. *Hardware/Software Co-Evaluation of Instruction Sets.* PhD thesis, Technische Universität Wien, Vienna, Austria, 1996. (to be published).

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture–A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[5] J. C. Huck and M. J. Flynn. *Analyzing Computer Architectures.* IEEE Computer Society Press, Washington, DC, 1989.

[6] G. Kane. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[7] W.-Y. Koe, H. Nayak, N. Zaidi, and A. Barkatullah. Pre-silicon validation of Pentium CPU. In *Hot Chips V – Symposium Record*, Palo Alto, CA, August 1993. TC on Microprocessors and Microcomputers of the IEEE Computer Society.

[8] A. Krall. The Vienna Abstract Machine. *The Journal of Logic Programming*, 1996. (to appear).

[9] B. Lang. Self Arbitrating Elements for Modelling Systolic Dataflow in Field Programmable Gate Arrays. TU Hamburg-Harburg, 1994.

[10] J. R. Larus. SPIM S20: A MIPS R2000 simulator. Technical Report 966, University of Wisconsin-Madison, Madison, WI, September 1990.

[11] D. Maurer. Synthese eines MIPS-I Prozessorkernes in VHDL [Synthesis of a MIPS-I processor kernel using VHDL]. Master's thesis, Technische Universität Wien, Vienna, Austria, 1995.

[12] T. H. Meng and S. Malik. *Asynchronous Circuit Design for VLSI Signal Processing.* Kluwer Academic Publishers, Boston, MA, 1994.

[13] J. A. Rowson. Hardware/software co-simulation. In *Proc. of the 31st Design Automation Conference (DAC '94)*, San Diego, CA, 1994. ACM. (Tutorial).

[14] J. Sato, A. Y. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai. PEAS-I: A hardware/software codesign system for ASIP development. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, E77-A(3):483–491, March 1994.

[15] P. Sinander. VHDL modelling guidelines. Technical Report ASIC/001, Issue 1, European Space Research and Technology Centre, European Space Agency (ESA), Noordwijk, The Netherlands, September 1994.

[16] Synopsys. *Design Compiler Family Reference.* Synopsys, Inc., Mountain View, CA, April 1995. (Version 3.3a).