LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Test and Validation for Monsoon Processing Elements

Computation Structures Group Memo 339
January 1991

Michael J. Beckerle
Motorola Cambridge Research Center

Gregory M. Papadopoulos
Massachusetts Institute of Technology

To appear in *Proceedings of the 1991 IEEE International Conference on Computer Design*, Cambridge, MA, October 1991.
Also published as Motorola Technical Report MCRC-TR 16

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.
Subcontractor: Motorola, Inc.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Test and Validation for Monsoon Processing Elements

Michael J. Beckerle
Motorola Cambridge Research Center

Gregory M. Papadopoulos
Massachusetts Institute of Technology

Abstract

The Monsoon multiprocessor presents a number of test and validation challenges. The overall strategy used is described including the scan-path discipline, and use of gate-level and automatically generated instruction-set-level simulators.

1 Introduction

The Monsoon multiprocessor presents a broad set of design, test, and validation challenges. The Monsoon processing element is a new, and radically different, processor architecture[2]. It incorporates a deep pipeline in which up to eight independent or interdependent threads are simultaneously interleaved. Also, in order to facilitate experimentation, its instruction set is "soft decoded" by a number of downloadable decoding tables; so, the validation techniques must accommodate continuous change.

We have developed an aggressive, integrated hardware/software approach for the test and validation of Monsoon processing elements. The strategy comprises three main elements: (1) scan-paths of internal processor state, (2) gate-level simulator of the entire processing element for hardware timing verification, and (3) a novel table-driven instruction-level interpreter. In this paper, we detail each of these elements, and show how they contribute to design, test, validation, and debug.

Much of the validation strategy can be understood from Figure 1, which details the interactions between the hardware and software components of the system from the standpoint of testing and validation. A new instruction set is defined using a database-like tool which describes how a instruction is to be decoded during the various Monsoon pipeline stages. The tool outputs a file of the contents of the decode tables. The file is downloaded (using the scan path) by the host processor during Monsoon bootstrap. The same file is imported by the gate-level simulator during startup *and* is used as input to a program which *generates* the instruction-level interpreter, MINT. In this way, we maintain coherence across all platforms – hardware, gate-level simulator, and interpreter.

The validation strategy leveraged this coherence in two ways. First, it facilitated the validation process by allowing creation of a series of instruction subsets. Second, it permitted a rapid way of determining whether an error was a hardware design flaw, a manufacturing defect, or a software bug.

The three instruction subsets are called IS_0, IS_1, and IS_2. IS_0 is a very small subset, consisting of only about 50 opcodes; it was sufficient to serve as a target for the compiler since many small programs could be compiled and run; however, not all program constructs could be expressed. IS_1 is somewhat larger, having about 300 opcodes, and IS_2 is the full instruction set, having more than 1000 opcodes. IS_1 is sufficient to express any program construct, but the increased flexibility of the full IS_2 instruction set allows the compiler to produce better code.

The IS_0 subset is small enough that small programs using it could be run on the gate-level simulator. In addition, compiled code for small programs could be run on the IS_0 version of the MINT interpreter. Finally, IS_0 was used during initial testing of the first Monsoon processor board.

The IS_1 subset served as the focus of our most systematic validation effort, ISV or *Instruction Set Verification*. A tool for rapidly composing test vectors for instruction sets was built and used to create a comprehensive test suite for each instruction in IS_1. From these test vectors, small host programs were generated which lead a Monsoon processor through the execution of a single instruction and then verify the results. Since our interpreter, MINT, and the Monsoon hardware both implement an identical server protocol, these test programs could be run transparently on either the interpreter or the hardware.

2 MINT: The Monsoon Instruction-Set Interpreter

The MINT interpreter was developed to provide an efficient way to run Monsoon programs without hard-

CH3040-3/91/0000/0160$01.00 © 1991 IEEE

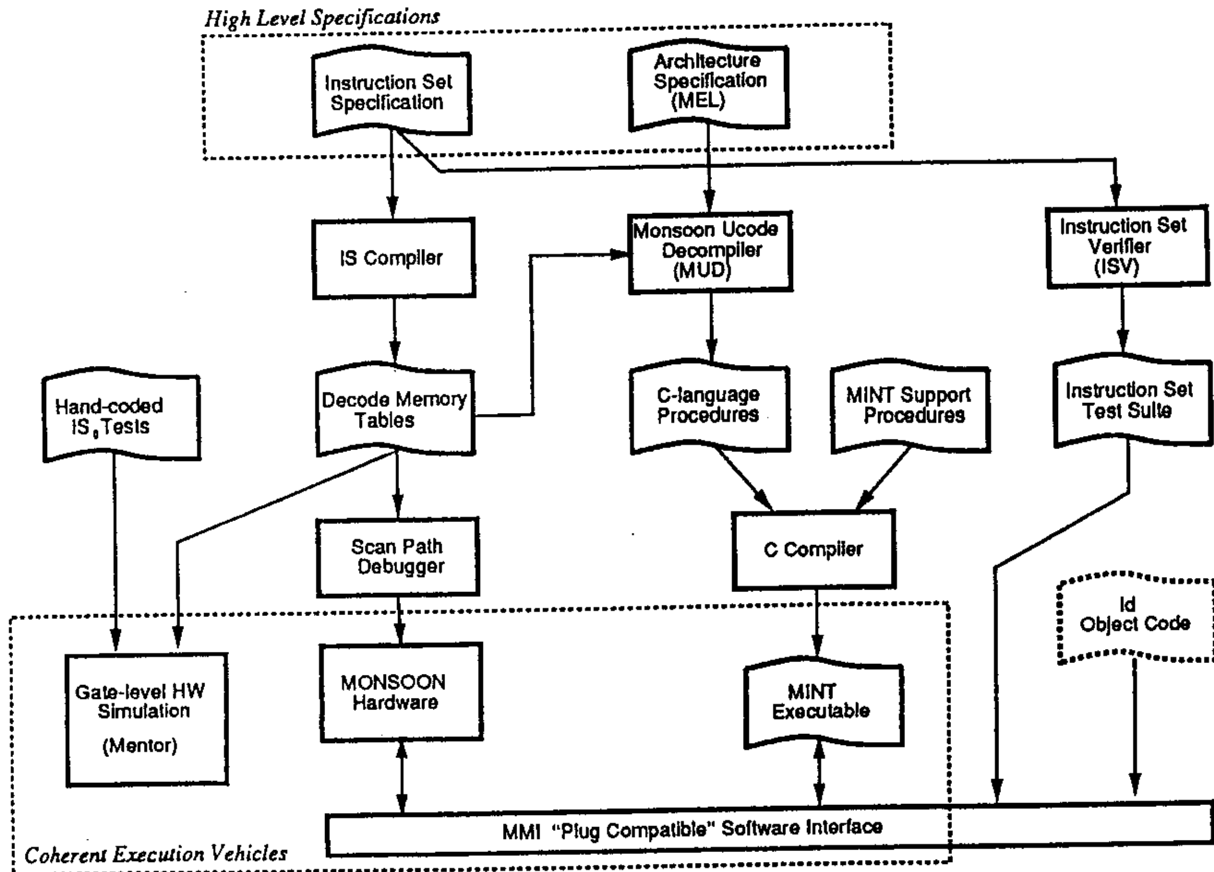

Figure 1: Organization of the Testing and Validation Process.

ware. This alone would have been helpful in the validation effort by providing a vehicle for software development before the hardware was completed. However, because of the way MINT was implemented an additional benefit was gained, which is that the debug memory contents that define the instruction set could be extensively debugged without need for the hardware.

To provide adequate speed, MINT could not simulate the hardware accurately. In particular, it had to avoid the overhead of interpreting the decode memory tables in software. Instead, corresponding to each Monsoon instruction a single procedure was needed which directly implemented its functionality without simulating the activities of the hardware. These could have been hand-coded from the instruction set specification; however, this immediately raised the issue of keeping the interpreter consistent with the changing decode memory as the instruction set specification evolved. To avoid this problem we used a unique approach: a macro-language description of the Monsoon architecture is fed into a program called MUD, the *Monsoon Ucode Decompiler*. Given the decode memory contents, MUD generates the procedures corresponding to each instruction thereby guaranteeing that the interpreter is consistent with the instruction set as defined by the decode memory[1].

The macro language used to describe the Monsoon architecture is called MEL, *Monsoon Emulator Language*. The MUD program takes this architecture description and the decode memory contents and produces the instruction interpreting procedures in an intermediate form. These are then optimized using standard compiler optimization techniques. Finally, C-language procedures are emitted. These C procedures form the core of the MINT interpreter and are linked with other components which simulate the remaining parts of the processor and which implement the Monsoon server network interface.

As a server program MINT is indistinguishable from the actual Monsoon hardware. This guarantees that the client programs, such as the ISV validation tests, will run unchanged on MINT and the hardware.

3 Scan Path Organization and Design Principles

Monsoon employs a full scan design technique for controllability and observability of most every bit of internal processor state. The fundamental concept of a full scan design is to provide a serial access path through all state elements (latches and flip-flops) such that the state can be set or observed in an amount of time directly proportional to the number of state elements. Full scan is quite useful for low-level diagnosis and debugging—a processor can be placed into any state and/or the entire current state of the processor can be analyzed.

In the Monsoon implementation, over 6000 bits of internal state are accessible through one of fifteen circular scan paths (also called *scan rings*). The scan paths can be manipulated via a UNIX device driver interface by a VME-based frontend processor. We use the scan facility to initialize the processor decode memories, to download machine code into instruction memory, to initialize the processor pipeline, and for low-level debugging.

The reader should be aware of important differences between the full scan technique employed by Monsoon and the recently popularized boundary scan approach. In contrast to full scan, boundary scan is concerned primarily with the integrity of *interconnections* among chips (their boundaries) in order to supplant traditional in-circuit testing; access to internal state is still ad hoc.

3.1 Scan Protocol and Clocking

Scan design is costly in time, gate count, and pins. A scannable flip-flop typically requires a 50% larger setup/hold time window when compared to a flip-flop without scan. Furthermore, total system gate count (or area) is typically increased by 4-20% over non-scan designs. Finally, a scan protocol will add from two to five pins to each package in the system, depending on whether the scan path has its own clock and/or dedicated serial outputs.

Monsoon is a board-level design comprising discrete CMOS parts, programmable logic devices (PLD's and PAL's), and a core of medium density 1.5 micron CMOS gate arrays (eight arrays, 10,000 gates each). Because PLD's are relatively register-poor, we do not employ a shadow flip-flop for each state and instead thread the scan path directly through each state element. An obvious drawback of this approach is that level sensitive controls, like resets or memory write strobes, are not readily scannable because the other bits that move through the state during scanning may cause a spurious state change. PLD's also require that the system clock and scan clock be the same clock line. As a result, we use a five wire protocol which unifies system clocking, clock qualification, and scan:

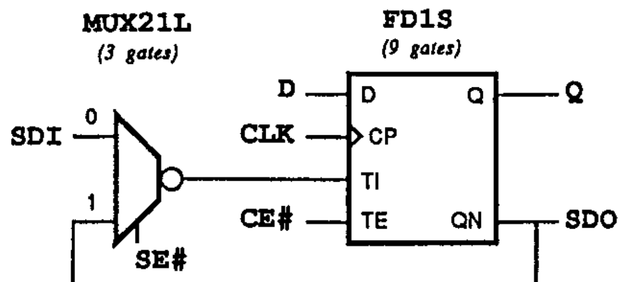

Figure 2: Gate Array Scan Cell with Clock Qualifier

Name	I/O	Description
CLK	*I*	Free running system/scan clock
CE#	*I*	Clock enable, active low
SE#	*I*	Scan enable, active low
SDI	*I*	Scan data in
SDO	*O*	Scan data out

To save pins, SDO is almost always aliased onto another (non three-state) output and SE# determines whether the output is scan data or the normal output. The CLK is a free-running system clock which is qualified by the clock enable CE# and the scan enable SE# as follows:

CE#	SE#	Action
H	H	Hold. Flip-flops recirculate state
L	H	Clock. Flip-flops clock D to Q
H	L	Scan. Flip-flops clock SDI to Q

Figure 2 illustrates the implementation of our scan protocol for the LSI Logic 10K series gate arrays. The FD1S is a vendor-supplied scannable flip-flop (9 gates), and the MUX21L is an inverting 2-input multiplexer (3 gates). Observe, the scan flip-flop is actually used to implement the clock hold function, while the external multiplexer implements the scan path using the complemented output of the flip-flop.

This organization has three advantages. First, the setup/hold time is minimized: $t_s = 1.3, t_h = 0.4$ nanoseconds (versus $t_s = 0.8, t_h = 0.3$ nanoseconds for the non-scannable flip-flip). Second, the CE# qualifier can come very late in the cycle permitting a clock to

be inhibited as close as 2 nanoseconds before the next rising edge. We use CE# to implement single stepping and to insert processor wait states for multicycle operations such as floating point divide. Finally, because the scan path is derived from the complemented output, the scan path interconnect does not add capacitive delay to the normal flip-flop output.

3.2 Scan Path Debugger

The scan and clock control logic of each processor is mapped into a set of VME-accessible registers which are manipulated by a UNIX device driver. The *Scan Path Debugger (SPD)* is an X-Window application layered on top of the device driver to allow a user (or another program) direct access to the processor state. Through a graphical interface, the processor pipeline can be inspected and manipulated, and a programmable number of clocks can be issued to single-step the processor. After each step, SPD scans out the processor state into a trace buffer which can then be viewed much like the output of a logic analyzer.

SPD can also be used in non-interactive mode by supplying command scripts. For example, SPD is used to bootstrap the processor by "backloading" the contents of the decode memories with an instruction set definition. A number of automated low-level hardware tests are also encoded as non-interactive SPD scripts.

4 Summary and Conclusions

Overall, we believe the techniques described above made the testing and validation task for Monsoon feasible and in fact greatly reduced the time that would have been required to bring up this initial prototype of a highly complex parallel processor system. Despite this success, we feel there are clear areas for improvement.

Our techniques maintained consistency in the use of the decode memory and its implied instruction set definition; nevertheless, there are several areas where even more could have been done to avoid maintaining multiple equivalent sources. First, the ISV system used to generate instruction set tests could have been more closely wed to the actual specification process for the instructions. This would have helped reduced the problem of knowing whether the tests in fact were testing for the proper functionality.

Furthermore, we view as a goal the ability to generate both hardware and an instruction-set interpreter like MINT off a common source language. The redundancy between the actual hardware specification, which was a schematic diagram, and the MEL macro-description of the machine architecture used to generate MINT was initially a source of frequent problems.

In the area of scan design, it has become clear than we would like boundary scan ***in addition to*** full scan in order to allow better isolation of manufacturing defects. Also, we have relied too heavily on scan for some operations (for example, the frontend loading code into instruction memory), where dedicated parallel access paths would improve performance significantly.

Acknowledgements

Andrea Carnevali helped design and implement the ISV system. Tom Kish implemented the scan-path debugger, and Rob Robinson implemented the lowest level scan ring diagnostics, drivers, and access libraries. Ken Traub is responsible for the client/server design philosophy, the implementation of which was done by Peter DeWolf. Jack Costanza and Ralph Tiberio handled the Mentor simulation.

We would like to acknowledge the contributions of the entire Monsoon team in the testing and validation of Monsoon. This includes the members of the MIT Computation Structures Group, the Motorola Cambridge Research Center, and the Monsoon Advanced Technology Laboratory at the Motorola Computer Group in Tempe, AZ.

References

[1] Derek T. Chiou. A Reverse Compiler: A Monsoon Dataflow Microcode to Common Lisp Compiler. Master's thesis, Massachusetts Institute of Technology, Cambridge MA, June 1989.

[2] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91. IEEE, 1990.