MDP Design Tools and Methods

Richard A. Lethin
William J. Dally
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139

Abstract

The Message Driven Processor (MDP) was produced by an academic-industry research collaboration. A conservative and pragmatic design strategy was adopted for logic and layout design to minimize effort and reduce risk. The effort demonstrated that a complex VLSI computer can be built successfully in an academic context. The methodology and tools used are described here.

1 Schedule and Staffing

The MDP is a 1.1M transistor processor and node for the J-machine[1], produced as a research effort in parallel computing by a collaboration between the Concurrent VLSI Architecture group at MIT, and the Platform Architecture group of Intel Corporation in Santa Clara, California. MIT designed the architecture and logic, and supported the layout and manufacturing performed at Intel Corporation.

The MDP architecture specification[7] developed during 1986-88 from preliminary studies[3] and a desire to build a fine-grained parallel computer node with efficient mechanisms to support several programming models. In fall 1988, a group consisting of four MIT graduate students and one faculty member began logic design. At Intel, two full-time engineers and later one layout designer were assigned to the project. A full-time MIT staff member joined and began assisting with testing in the summer of 1989. Floorplanning also started that summer, with three of the graduate students working on-site at Intel. Layout continued through the fall of 1990. Full-chip layout checking and verification began in fall of 1990 and continued through early 1991. Tapeout occurred in March, 1991. First silicon was received in June 1991, and was soon running small programs. Chip testing and program development began and several logic bugs were detected. These were corrected in early 1992; B-step masks were produced in March, 1992, and B-step silicon was received in May, 1992. A 128-node machine has been running since October 1991, and a 1024 node machine should be running by the end of summer 1992.

[1]See the companion articles in these proceedings for a description of the MDP architecture.

2 Methodology

We adopted a pragmatic and conservative design methodology for the MDP. We accepted costs in performance and area in order to reduce the likelihood of bugs.

The primary methodological constraint was that although the chip could have been full-custom, we limited ourselves to designing with gate-level standard cells (NAND, LATCH, etc). Individual transistors were not allowed. We used an Intel standard cell library with a large selection of functions and drive strengths. The most important benefit associated with standard cells was that we could simulate the design at the gate-level, which is 15 times faster than switch level simulation. This allowed us to run more testing cycles.

Using standard cells rather than transistor-level cells cost us approximately a factor of four in area (1/2 the density and 2× the devices) in the AAU and RALU. However, this did not have a huge effect on the total chip area, because a large proportion of the chip is random logic and dense RAM cells. The auto-placed and routed random logic had to be standard cell, and the RAM is a very efficient full-custom design. Standard cells also cost us a factor of two in clock cycle time; this was considered an acceptable cost for completing the project with the resources available in the academic context.

The MDP uses a nonoverlapping two-phase clock-

1063-6404/92 $3.00 © 1992 IEEE

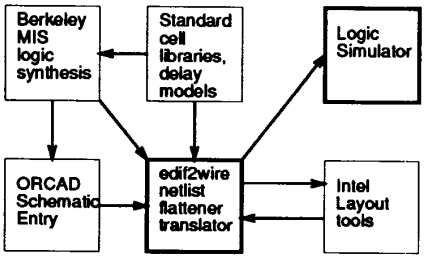

Figure 1: Information flow between CAD tools. Bold squares indicate homebrew tools.

ing scheme. We prohibited the use of edge-triggered registers by removing them from the library, to enforce conformity in clocking style: only latches were permitted. We did not have tools to enforce timing characteristics of inter-module signals, so we relied on timing verification checks and simulation of the integrated RTL and logic to detect timing bugs.

3 Schematic and Layout Design Tools

Intel engineers use a set of proprietary CAD tools that are supported internally. These tools were available for the MDP project only on-site at Intel, and thus only for the design tasks performed there, primarily layout and layout verification. For work on logic design and verification at MIT, we assembled a system of non-Intel tools, including our own netlist translation tools and simulators.

The flow of schematic information between tools is shown in Figure 1. Schematic entry was performed using OrCAD[8]. Initially, schematics were very unstructured, with little partitioning between control and datapath. When we began layout, we soon discovered the obvious: the hierarchical structure of the schematics must be isomorphic to the hierarchical structure of the layout. We spent much effort restructuring the schematics into strict control and datapath bitslice partitions.

In early phases of the design, some modules were written in the BDS language and synthesized using the Berkeley MIS program[1]. We extended MIS with a postpass to select logic gates with the appropriate drive based on output load. This was useful for early prototyping, but except for a few control circuits in the RALU and in the AAU, we found that the synthesized logic was too slow on the important critical paths; we ended up designing and tuning most control logic by hand.

The OrCAD hierarchical netlister was too inflexible to produce netlists for Intel tools (Intel had non-standard netlist formats and the OrCAD hierarchical naming and traversal was insufficiently powerful), so we netlisted the schematics individually, and used our own netlist translation program, EDIF2WIRE, to do selective flattening and combining of netlists. Later, we extended EDIF2WIRE to accept and produce netlists in several formats: EDIF, our simulation format, and the formats produced and used at Intel. Interestingly, most of the effort in writing EDIF2WIRE was not implementing sophisticated algorithms, but rather getting right details like power connections and signal naming in the diverse formats required by the tools. EDIF2WIRE also includes facilities to estimate gate delays based on fanout for logic simulation, to adjust the capacitance of major buses, and to check capacitance of the larger nets. The program was extensively tuned for speed because it was in the critical path for firing up a simulation after making small changes to schematics.

Layout was produced at Intel from the schematics through a combination of strategies. The RAM, pads, and clock generator were full-custom, borrowed from other chips. Large arrays like register files were hand-placed and routed to a standard datapath pitch by a staff of two mostly full-time layout engineers. A few large functional unit blocks with complex routing but regular structure (the shifter and find-first-bit circuit) were hand-placed and automatically routed, but the majority of random control logic blocks were both automatically placed and automatically routed, often under remote direction from MIT. Global routing of logic blocks was done with a powerful Intel proprietary hierarchical placement and routing tool. Design rule checking software looked for process rule violations. Layout verification was performed at Intel using hierarchical netlist checkers. Completing the layout and getting it through the netlist verification before tapeout took several months.

4 Simulators

We wrote our own MDP simulators, shown in Figure 2, offering a range of accuracy and speed. To verify the design, each simulator was checked against another, and eventually the logic simulation was validated with the hardware.

The instruction level simulator (17,000 lines of C++) runs MDP binaries at 2000 instructions/sec[2].

[2]All timing measurements in this paper are for the program

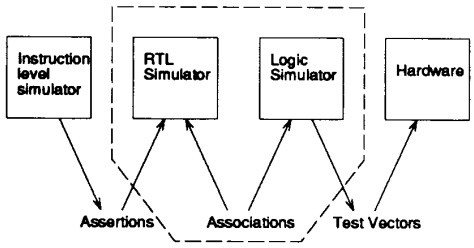

Figure 2: Simulation environment. The RTL and logic simulator are linked into a single program.

This simulator was used for architectural verification and for debugging the compilers and run-time kernel. It includes extensive debugging and architectural hazard-detection features. But the simulation results diverge slightly from the actual hardware: the microarchitecture is not modeled, and messages are delivered instantaneously. This limits the instruction level simulator's ability to be used as a "gold standard" for checking the RTL.

The RTL model of the MDP was used to design and debug the microarchitecture. The RTL runs programs at 50 cycles/sec. Rather than use a special-purpose hardware description language and simulator such as Verilog, we chose to write our RTL model in C (26,000 lines of code). This choice was made primarily for immediate convenience: we had a C compiler handy. Logic blocks in the RTL are stylized C subroutines. Wires are represented by variables, combinational logic is represented by C statements, and latches are represented by conditional statements that test the clock variables. A small kernel calls the major logic block subroutines in succession; the modules communicate with each other through shared C structures. The RTL is *not* event driven. Special care was taken to allow events to flow correctly between modules.

We first debugged the RTL by comparing its results against assertions generated by running test programs on the instruction simulator. These assertions specify the sequence of values in registers as the program runs. Most of the test suite eventually grew to run in this mode. For testing cases where the instruction simulator model was idealized, later test programs were more sophisticated and self checking. They could be run stand-alone (without assertions) to verify the correctness of the RTL, and they could also run on the hardware.

running on a SPARCstation IPC (roughly 18 SPECmarks) with 32 MB of main memory.

In the end, writing the RTL in C was probably more difficult than writing it in a behavioral language, though we did have the advantages of more programming flexibility, of having all source code, of easy portability to new platforms, and of the C compilation, tuning, and debugging environment. Most interestingly, though, our use of C for the RTL led us to an unusual and powerful strategy for testing the logic design.

The use of C for the RTL meant we were unable to ***substitute*** logic modules for RTL modules to test them. Instead, we chose to **piggyback** logic modules onto the complete RTL: the logic "rode on the back" of the RTL. The important distinction between substitution and piggybacking is that in piggybacking, both the RTL and the logic for the module being tested run simultaneously. The complete RTL is intact and supplies values for otherwise undriven logic inputs, and values for checking logic outputs.

The advantage of piggybacking is in the nature of verification test programs needed. Substitution relies on running tests that are self-checking and can detect potential bugs introduced by the substituted logic model in the simulation. With piggybacking, the test simply has to exercise the chip in a manner that will propagate divergence to the particular logic module's IO boundary. Most of the tests in our suite happened to be of the later type, simple exercises. Randomly generated code sequences are also excellent exercises of the later form.

When we initially tried to implement piggybacking by extracting test vectors from the RTL for a commercial logic simulator, we ran into two problems. First, we found ourselves facing unmanageably huge test vector files. Second, we found that our commercial simulator would often crash on larger circuits.

In response, we wrote our own 9-state, multiple-delay simulator[6], linked it to the RTL model and ran them in lock step, checking coherence of the two models using **associations**. With associations, the generation and transfer of the test vector information (the time/value to force and check) takes place at logic simulation time. The information is transferred through main memory rather than to and from disk. Running the RTL simultaneously adds little overhead to simulation time because it is so much more efficient than the logic simulation.

Associations are created when simulation begins by a function call that specifies correspondence between nets in the RTL and nets in the logic design, and a rich set of options controlling testing semantics (polarity, qualification, timing). Associations are evaluated

at the beginning and end of each clock phase, four times per major clock cycle. **Force associations** extract logic values from the RTL and drive them onto undriven or bidirectional pins, while **check associations** extract values from the RTL and logic, comparing them to ensure that the net behaved the same in both models. Associations are computationally cheap, encouraging wide testing both at the periphery and internally in logic modules (1700 check associations were used when running the fully-integrated CPU logic).

By writing our own simulator, we were able to tune performance as the size of the design grew. The RTL with a gate-level model of the whole CPU (33,000 gates) runs at 1.4 instructions/sec and 57,000 gate evaluations per second. The logic simulation uses 10 Mbytes of memory for data structures to represent CPU schematics (using $\approx$ 300 bytes/gate, which includes the ASCII names of the hierarchical nets).

Another unique feature in the logic simulator was **transition modeling**. The design uses a two-phase, symmetric, nonoverlapping clocking scheme. We developed a "quick and dirty" timing analyzer to find long paths in the combinational logic. However, it could not trace signals through latches; indeed, this represents an interesting research problem[5]. There were various places where the timing methodology was "extended" (violated) in the interest of performance; we needed some way of gaining confidence that there were no hazards or long paths lurking undiscovered in the design. To address this, we added transition modeling to the logic simulator, which simulated the potential skew in the output by making every transition from 0 to 1 or from 1 to 0 go through the X state. (This is similar to the hazard checking of [4].) It uncovered many bugs: particularly long paths and improperly qualified clocks. This feature is so fundamentally useful that it should be a required feature in logic simulators, even though it increases simulation time by 50%.

We added a switch simulation algorithm[2] to the logic simulator to allow us to simulate gates and switches together. This was useful because although the CPU was designed using a standard cell library of gates and latches, the clock generator and pads were borrowed from other chips, and had transistors as primitives. In addition, we were able to simulate extracted layout for large sections of the design using the switch level simulator. This served to "close the loop" - assuring that the layout design at Intel and the methodology used to generate it (checking our local standard cell models, netlist translators, network transfers, etc.) had no gross bugs. The switch simulator ran at 1/15 the speed of the gate simulator; at 6000 vicinity-solves/sec.

Test Directory	Lines of Code	Number of programs	Number of Cycles
branch	661	3	1941
call	267	2	1983
diag	553	6	15271
faults	5269	23	11576
ifetch	544	6	1609
memory	1293	10	15475
ops	24775	27	120308
priority	1630	9	2519
registers	3369	7	5345
send	7135	22	24101
xlate	5760	8	23710
TOTAL	51256	123	223838

Figure 3: The MDP test suite.

The simulator was recently modified to produce test vectors from the simulation of the CPU running a manually-written test program. The test vectors allow us to compare the simulation and the hardware on a cycle-by-cycle basis, and are used for wafer sorting.

5 Test Suite and Bugs

To test the MDP logic, a suite of programs was written by the logic designers. The number of cycles dedicated to the different areas of the chip are shown in Figure 3[3]. Our current suite runs for 220,000 cycles, taking 2 days to complete. The testing in the "ops" (ALU operation tests) section was particularly heavy due to tests which supplied random operands to the different functional units. Random operand testing was used to try to shake out bugs from the ALU; this accounts for 1/3 of the total cycles in the test suite. For the B-step, we also subjected the design to random network traffic, random interrupts, and repeated run/halt and single stepping events. Random tests seem to have been very successful: those sections of the design tested with random tests are relatively bug free. Other sections of the design that relied on hand-crafted tests for single features seem to have had more bugs.

[3]These figures include additional tests written after bugs had been detected in A-step hardware, 5% of the total lines, and 10% of the total cycles. Also, the tables do not show that most of the suite can be run in special modes which randomly pepper the chip with asynchronous events, diagnostic commands, and messages.

An on-line bug log was established early in the design effort. When a bug was detected, an entry was placed in the bug log and a "puck[4]" was passed to the responsible designer. Designers with the puck had two days to fix the bug or pass the puck to the person really responsible. In the course of the design effort, 191 bugs were detected, documented, and fixed in the instruction simulator, RTL, and logic.

In the months after the A-step chips arrived, we discovered 21 more bugs; nearly all were "second-order" bugs, in other words, involving the interaction of different modules. There were very few "first-order" bugs. Currently, no electrical, timing or hazard bugs have been detected. Luckily, all A-step hardware bugs had software workarounds which allowed thorough testing for preparing the B-step.

There seem to be five main classes of hardware bugs, related to: the serial diagnostic port, the network, external memory, changing mode bits, and interference of faults. The diagnostic port features to single-step, start, and halt the CPU were found to interfere with instruction execution, and with faults. The two network priorities, P0 and P1 interfered with each other. Several operations did not work when they accessed external memory rather than on-chip memory. Different modules used different pipelining of mode bits, so when mode bits changed, there was often a cycle where two modules were in inconsistent modes (eg., one module had faults enabled, another had them disabled). Finally, asynchronous faults such as external interrupts were found to interfere with synchronous faults.

Most of the bugs could have and should have been detected with a better test suite. But most of the bugs should have been avoided. Without a clear, universally understood design philosophy for certain intermodule interactions, we got bitten. With these philosophies clearly articulated, these bugs would more likely have been avoided by design and detected by test.

6 Closing

A buy versus build tradeoff permeates all tool decisions in large scale systems efforts. If tools are purchased, methodology must be adapted to the tools. However, if the methodology is nonstandard, it is the tools that must be adapted. Our methodology required Intel/MIT collaboration without the off-site use of Intel CAD tools. Consequently, it proved easiest to write many of our own tools. This gave us the flexibility to implement special innovative features by raw, immediate utility. We recommend these functions to other designers, particularly, piggybacking, associations, and transition modeling.

We spent significant effort developing our test suite. Alas, one can never have enough tests. "Second order" interactions did not receive enough coverage, and the result was bugs. However, given the complexity of the chip and the small size of the group that produced it, our methodology was relatively successful.

[4]The three Canadians on the design team made this ice hockey metaphor especially popular.

Acknowledgements

This work was funded in part by DARPA contracts N00014-88K-0738 and N00014-91-J-1698. Stuart Fiske, John Keen, Mike Noakes, Peter Nuth are the other designers of the MDP. Greg Fyler shepherded the designers and the MDP through Intel. Roy Davison led layout efforts. Richard Lethin is supported by a fellowship from the John and Fannie Hertz Foundation.

References

[1] R. K. Brayton, R. R. A. Sandiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Assisted Design*, CAD-6(6):1062–1081, November 1987.

[2] R. E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160–177, 1984. MOSSIM.

[3] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, 1987.

[4] Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal*, 1965. Use of ternary logic.

[5] A. Ishii. *Timing Verification of Level-Clocked Circuits*. PhD thesis, Massachusetts Institute of Technology, 1991.

[6] R. A. Lethin. A simulator for the message-driven processor. Master's thesis, MIT, Feb. 1991.

[7] M. Noakes. MDP programmer's manual. Concurrent VLSI Architecture Memo 40, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1991.

[8] OrCAD Systems Corporation, Hillsboro, Oregon. *ORCAD SDT III Schematic Design Tools*, 1987.