ICS

TECHNICAL REPORT

AMRM Prototype Board Design and Implementation *

Prashant Arora Dan Nicolaescu Rajesh Satapathy
Alexander Veidenbaum
Department of Information and Computer Science
444 Computer Science, Building 302
University of California Irvine
Irvine, CA 92697–3425
{dann,arora,ancuta,alexv}@ics.uci.edu
Technical Report #00–37

Information and Computer Science
University of California, Irvine

December 1999

AMRM Prototype Board Design and Implementation *

Prashant Arora Dan Nicolaescu Rajesh Satapathy
Alexander Veidenbaum
Department of Information and Computer Science
444 Computer Science, Building 302
University of California Irvine
Irvine, CA 92697–3425
{dann,arora,ancuta,alexv}@ics.uci.edu
Technical Report #00–37

Dept. of Information and Computer Science
Univ. of California at Irvine

December 1999

*This work was supported in part by the DARPA ITO under Grant DABT63-98-C-0045.

Contents

List of Figures

1 Introduction

This paper describes the design and implementation of an adaptive memory hierarchy in a board-level prototype using off-the-shelf FPGA and memory parts. While using these COTS parts, the AMRM board develops the capability to architect and adaptively configure the CPU-memory hierarchy to suit application needs. Performance gain in latency and available bandwidth can be achieved by using application-adaptive architectural mechanisms, hardware-assisted blocking, prefetching and dynamic cache structures that optimize movement and placement of application data through the memory hierarchy. For a description of the AMRM approach to architectural adaptation the reader is referred to [1] [2] [3].

The AMRM prototype board was designed to serve two purposes. It allows rapid prototyping of a variety of memory hierarchy architectures and adaptive caching mechanisms. Applications running on a host processor can be instrumented to use the board's memory hierarchy. Processor independence is provided through use of the PCI bus interface. The board is also designed to support a chip-level prototype on a daughter card. In presence of this daughter card, the board is designed to serve as a platform with on-board memory acting as main memory.

The design and usage of this board is enabled by key EDA tools. The paper also discusses the key challenges and opportunities for EDA tools to make design and efficient architectural exploration possible.

2 System Design Overview

The function of the AMRM prototype board is to emulate a reconfigurable memory hierarchy. Applications running on a host processor can be instrumented to use the memory hierarchy on the AMRM board. The board has a PCI interface that allows it to be driven by an execution-driven reference stream or an address trace from the host processor. The PCI bus interface allows the board hardware to be used for emulating the memory hierarchy of several popular processor architectures. This allows proposed changes to the memory hierarchy to be evaluated for several processors.

The memory controllers on the board are implemented inside an FPGA. This allows an in-system hardware reconfiguration of the AMRM memory hierarchy. The board implementation necessarily hard-wires certain parameters of the memory hierarchy. This includes the board's clock. In order to perform detailed and accurate simulation of the memory hierarchy at any clock speed, a hardware "virtual clock" has been implemented as part of the performance monitoring hardware. Performance monitoring hardware primarily includes various event counters which are readable from the host processor. The virtual clock emulates a target system's clock; the clock rate is determined by the target system's memory hierarchy design and technology parameters. The units in the design can be "programmed" through the use of configurable counters to take a certain number of virtual clock cycles for an operation. E.g. an L1 hit may be assigned n virtual clocks, a miss fetch may be programmed to take m clocks etc. The L1 cache controller handshakes with the virtual clock generation unit to increment the virtual time counter appropriately in each case. A unit can take multiple physical clocks to perform a certain access and increment virtual time only once. E.g. the tag and data stores of L1 cache can be a single RAM while virtual time may reflect a design with two separate RAM's. The virtual clock generation unit also increments virtual time appropriately when multiple units are operating in parallel.

The board includes performance-monitoring hardware in the form of event counters for read/write hits and misses, number of write-backs (for write-back caches) and simulation time in terms of virtual clocks. These counters can be read or reset using a command interface. Other registers that can be accessed are status and configuration registers. Status registers contain information about the internal state of the design, for example error codes for debugging purposes. Configuration registers can be used to configure the memory hierarchy; the various choices include cache size, line size, write policy etc. The delays in terms of virtual clocks for cache hit, miss fetch, write-through and write-back and miss fetch can also be configured.

As mentioned earlier, a processor in the system can access the board via memory mapped commands. Each command consists of a set of four words that specify the operation (e.g. memory read/write, register read/write), the address of the location to access and data in case of a write. Operations are available for cached/uncached access to the memory on the board. Commands are also provided for accessing the performance counters on the board and writing the configuration registers to "program" the memory hierarchy. For read commands, a read response is generated and data is written into host memory.

The prototype board currently implements reconfigurability of the memory hierarchy and can be extended to implement a range of adaptive hardware mechanisms like prefetching, stream buffers etc. Figure 1 shows the main components used in the implementation of the board. It contains:

- A PCI 9080 chip that provides the board's interface to the PCI bus [4].
- An Altera Flex 10K FPGA that contains the core of the design. It has controllers for the SRAM and DRAM and an L1 cache controller.
- Clock generation and distribution logic.
- An SRAM bank and a footprint for another bank for a total of 1MB. The SRAM is used as the multiplexed tag and data store of the L1 cache.
- Two DIMM banks used as main memory, for a total of 512MB.
- A mezzanine connector to support a mezzanine card. This card can be used for a variety of purposes including providing a slot for a chip-level prototype.

Monitor Program

For the purpose of testing the AMRM board, and running programs using the board we have written a GUI program that enables easy access to the internal structure of the board. Figure 2 shows some menus from this monitor program.

We have envisioned that initial testing of the board will be done in several steps, so the program was written to address the needs of all those steps. The initial step is testing communication with the board. The board contains a set of thirteen internal registers. We have an interface to read values from or write to those registers.

The next step is to test direct accesses to the DRAM on the board and do uncached accesses to the SRAM. We have an interface for reading and writing a range of values from those memories. When the uncached accesses have been sufficiently tested, we can test the cached access to the memory. Using the configuration registers we can configure the cache size, associativity and line size and perform experiments.

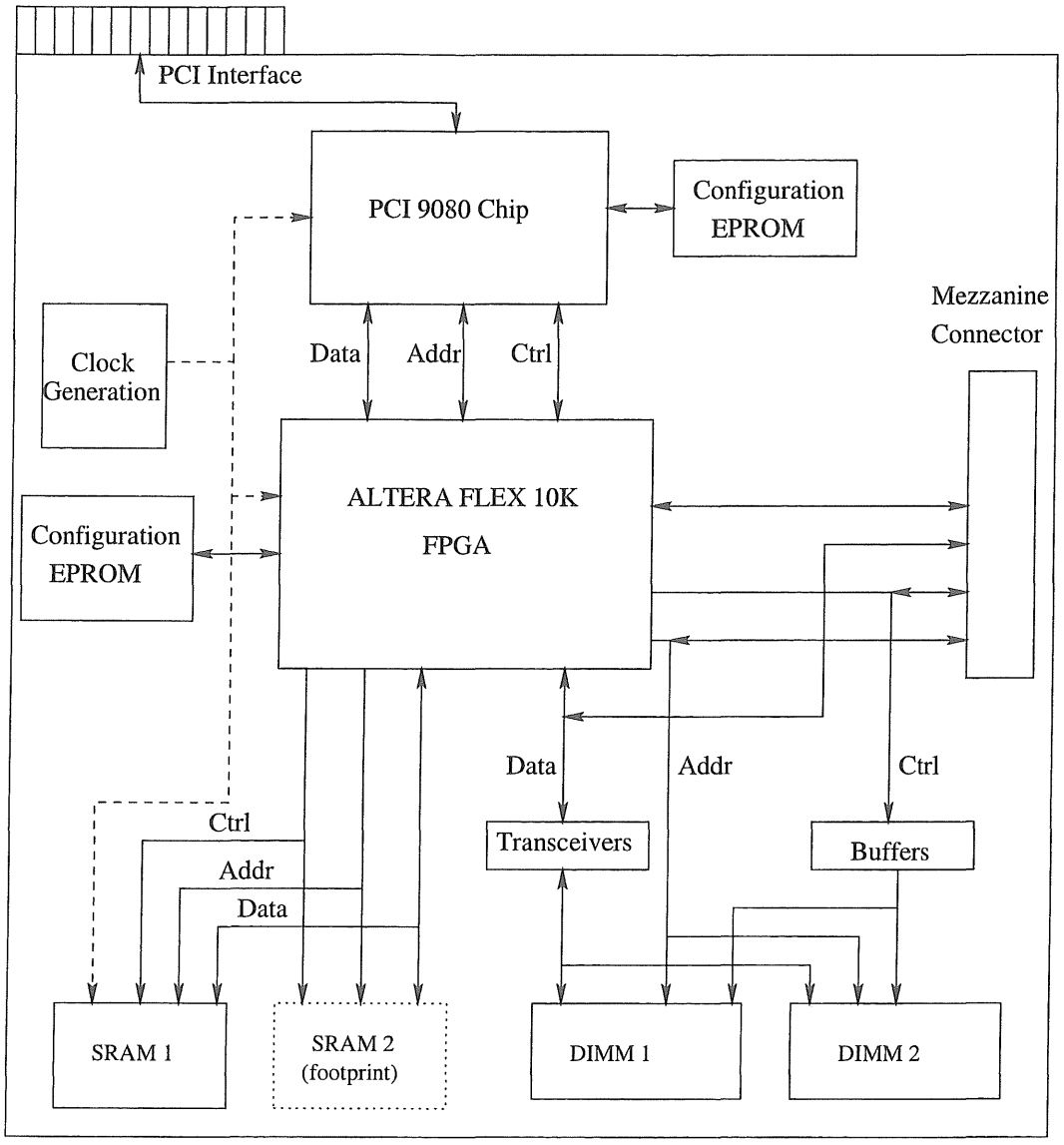

Figure 1: Block Diagram of AMRM Prototype Board

3 Implementation Methodology

3.1 FPGA Design and Implementation

The AMRM board features a re-programmable FPGA part that allows the system architect to download statically or at runtime memory controllers specific to a memory architecture. Figure 3 shows a block diagram view of the current FPGA design. The design is partitioned into the following modules:

- Sram control: Interface to the synchronous SRAMs on the board. The interface allows burst reads and writes of length 1, 2 and 4 on the SRAMs.
- Dram control: Interface to the DIMM modules on the board. The DIMMs are asynchronous; the synchronous state machine for the controller is thus written for a specific

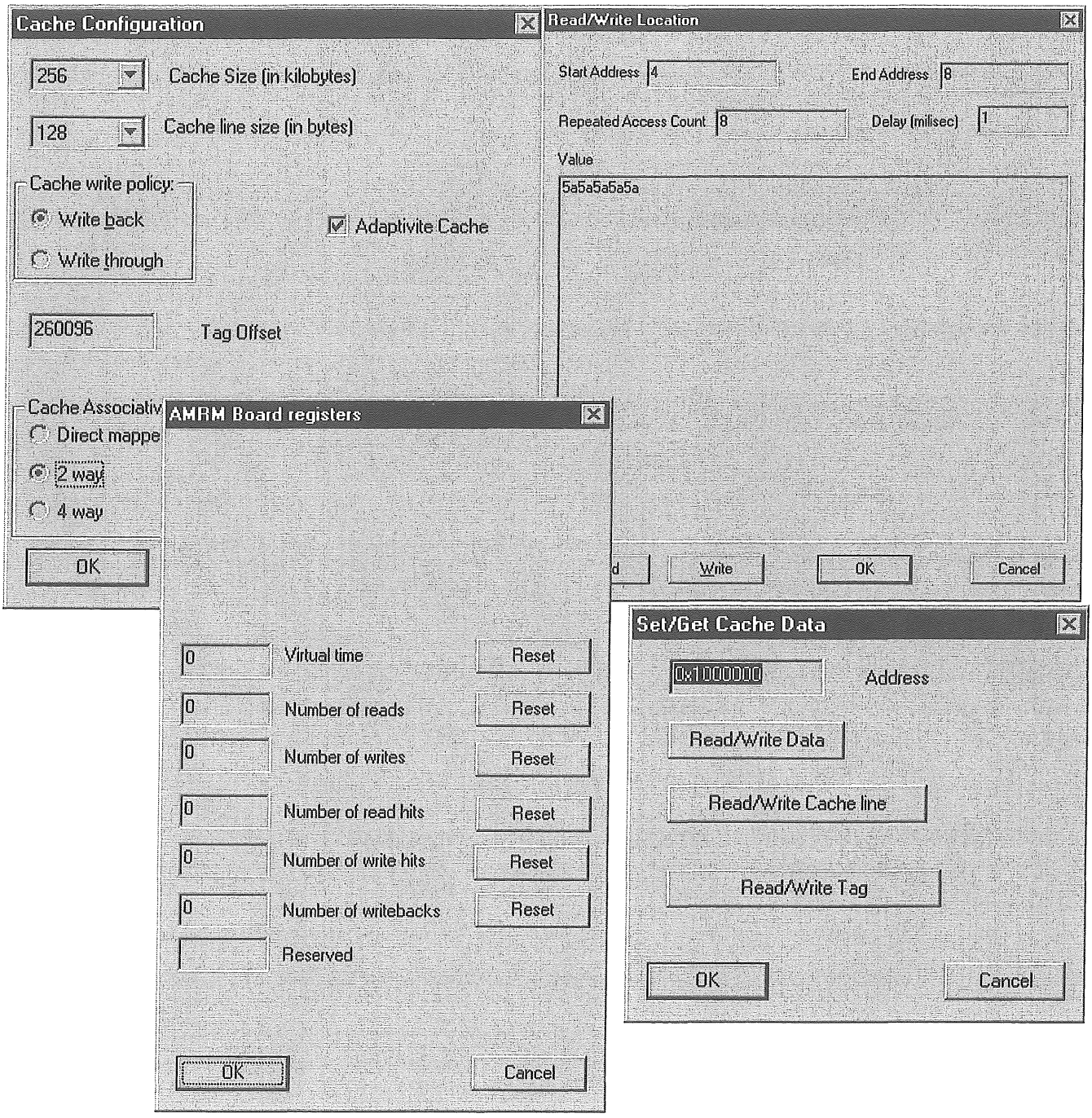

Figure 2: Monitor Program Dialog Boxes

clock frequency. The interface allows single word reads and writes on the DIMMs. The interface implements the initialization condition for the DIMM that involves generating eight refresh cycles after reset. One refresh cycle is generated every $15ns$ to meet refresh requirements (all rows should be refreshed every $64ms$).

- PLX Interface: This portion of the FPGA design interfaces with the PLX chip local bus. It contains input and output register files for getting command words and generating read responses.

- Command Processor: Reads commands from the PLX interface and decodes them to find the operation type (e.g. cached read/write, uncached sram read/write, register read/write etc.). It invokes the correct module to perform the operation. In case of reads, a response command is generated and passed back to the PLX interface module.
- L1 control: Implements the L1 cache controller. The cache controller design is configurable over a range of cache sizes ($8KB, 16, 32, ..., 512$) and cache-line sizes ($8B, 16, 32, ..., 512$). The write mechanism can be chosen to be write-through or write-back. The operating configuration is controlled by the host by writing the configuration registers inside the FPGA. Different applications or different sections of an application can use different values of the cache parameters. Note that the cache needs to be flushed before changing any of these parameters within an application.
- AMRM Register File: Contains control, status and performance monitoring registers that can be read and written/reset by the host through the command interface for the board, as described earlier.
- Virtual Clock Unit: Implements the virtual clock functionality as described earlier.

The FPGA design was carried out by splitting the design into these modules. Each of the modules was then written as one or more state machines with datapath in VHDL. The design sequence was chosen so as to fix the external interface of the FPGA as early as possible so that the board could be manufactured in parallel with FPGA design.

The design has two different clocks; the interface to the PLX chip inside the FPGA runs at half the clock frequency of the rest of the design.

3.1.1 Design Issues and Challenges

The FPGA defines the degree of architectural adaptivity possible with the AMRM board. Consequently the primary design challenge is to fit the best synthesized implementation that also satisfies the target clock period. The target clock frequency for most of the design was 66MHz ($15.15ns$) and 33MHz for the PCI specific portion.

Static timing analysis was carried out after each synthesis run and the critical paths were modified to improve timing. For example, asynchronous signals going across modules were changed to latched (with the associated timing changes) if they were in the critical path. Other design changes that improved timing were removing multiplexers on busses, duplicating signals with large fanout and splitting large state machines into smaller ones.

Signals with very large fan-outs appeared frequently in the critical paths, for instance, tri-state control signals for busses. These signals were generated at one flip-flop and were routed to a lot of inputs, sometimes far away from the generation point introducing large delays in the absence of global routes. A solution to this problem is to make copies of the high fanout signals in the design itself giving the P&R engine more flexibility in placing the control signals. Splitting large state machines into smaller state machines also lead to timing improvement by simplifying the decoding logic at the cost of making the design more complex.

The set of beneficial design changes was arrived at iteratively. Changes did not always improve timing as expected. Further, the process was time-consuming as the whole design had to be re-synthesized to evaluate any alterations. There was lack of tool support for modification of a

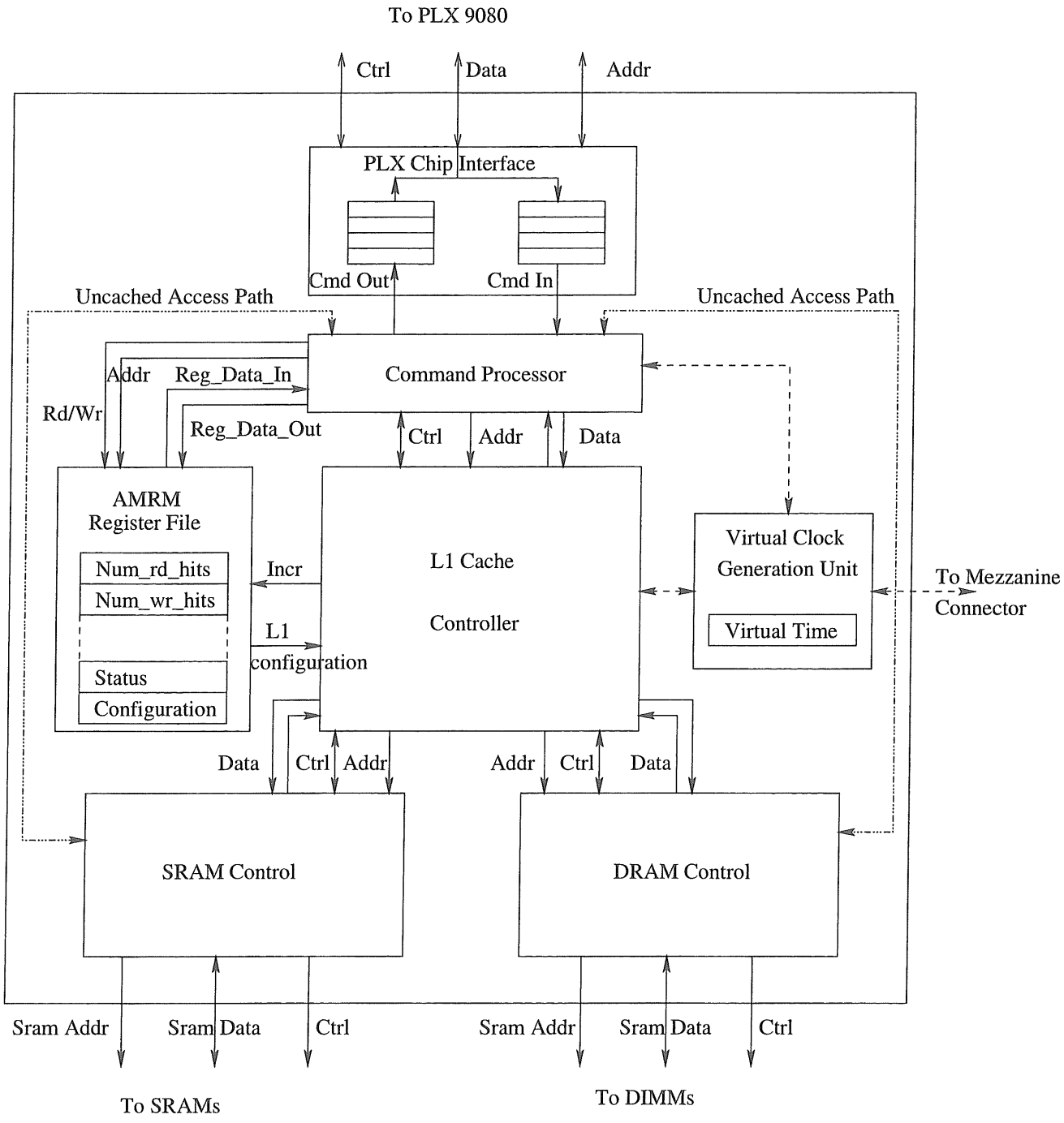

Figure 3: Block Diagram of FPGA Design

part of the design or accurate timing estimation for a particular design change. Improvement in timing was thus achieved very slowly. Other sources of timing improvements were changing the FPGA component and manual placement of the datapath. These are discussed in the section on synthesis.

3.2 Verification Methodology

Verification of the custom designed part of the FPGA inside the AMRM board was done through simulation using Synopsys VSS. Simulation was done bottom-up starting with sub-modules. Gate-level and timing simulation were also performed [5]. Figure 4 shows the verification flow.

3.2.1 Modeling the Environment

The FPGA design contains controllers for the memories on the board, the L1 cache controller and interface to PCI controller on board (PLX 9080). To perform functional simulation of the FPGA design, we needed models for the memories and the PLX chip. Models for the memories were generated using the memory modeling tools from Denali [8]. DIMM models were extended to include refresh and initialization requirements peculiar to our DIMMs. A simple functional model for the PLX chip's interface to the FPGA was written.

Availability of memory models saved us the effort of having to write these models on our own. A lot of timing errors in the controller state machines were immediately flagged by the models including some subtle errors during the latter stages of design. For example, when the clock period was increased from $15ns$ to $20ns$, the number of clocks being counted to time DIMM refreshes (asynchronous) was not reduced. The refresh checks that were added to the DIMM model flagged this error.

An important realization was the need to use models developed out of context as far as possible so that they are not influenced by the design at hand. For example, in our case, the dram refresh check was added to the model externally. The check initially counted clock cycles (not actual time) and verified that refresh was being done every certain number of clocks. This check thus assumed a certain period for the clock. The problem was later corrected but would obviously have let the previously mentioned bug slip by.

3.2.2 Structuring verification

Though the sub-modules were designed in a particular order with the aim of fixing the external interface of the FPGA as early as possible, it also helped structure the verification. Initially, the memory controllers were verified using the memory models as part of the test-bench. The high reliability of the models ensured that any bugs surfacing during simulation were due to errors in the controllers themselves. Next a pseudo cache controller was written that was later converted into the L1 cache controller. It was integrated with the memory controllers and the three units were simulated together; verification mostly helped find errors in the cache controller since the memory controllers were stable. Thus the tested modules were used as part of the test-bench for the remaining modules.

3.2.3 Test-bench automation

The regularity of memory design allowed us to automate our test-bench to a high degree. For the SRAM controller state machine, we needed to verify operation for burst writes and reads of length 1, 2 and 4. This was done by writing random data at a memory location and then reading it back and comparing with the value written. This was done for different random addresses

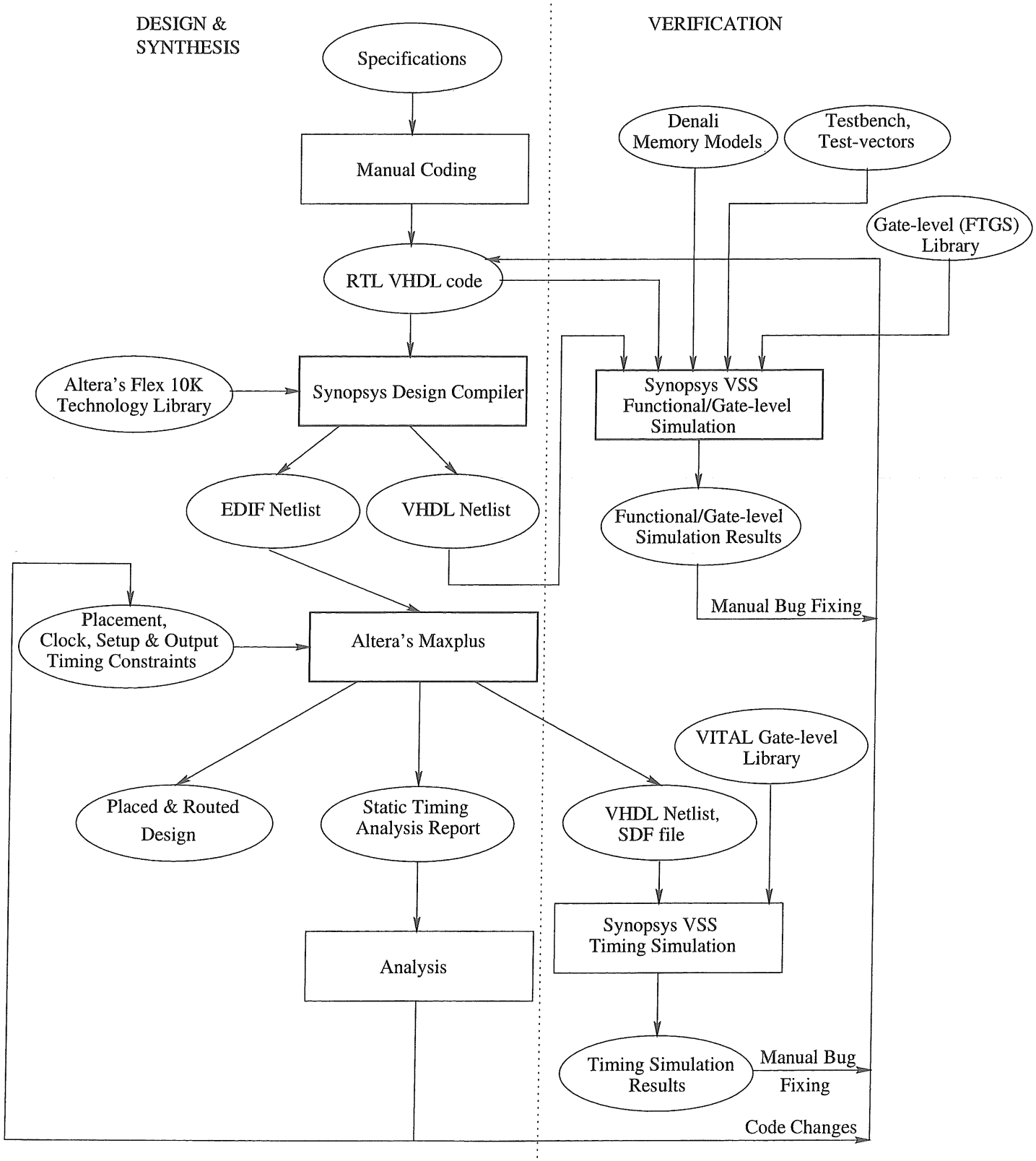

Figure 4: Tools Used and Methodology for the FPGA Design on the AMRM Prototype Board

and data. Different possible combinations and inter-leavings of these accesses were simulated. Output messages from the memory models were also monitored to check for errors.

For dram control, some other cases were tested for apart from simple reads and writes. These included the refresh mechanism; simulation was run as long as at least one refresh cycle was generated so that the tests in the DIMM were exercised. The checks for initialization added to the DIMM model were also exercised.

In order to cleanly separate test-vectors from the test-bench, a command language was defined. For example, there was a command each for resetting the design, sram read, write, dram read, write etc. The test-bench read the command file and 'executed' the commands on the design. This allowed us to change the test-vectors without recompiling the test-bench and maintain different sets of vectors.

The cache controller was verified in isolation initially to verify correctness of caching mechanisms. Some simple tests were written for the different possible scenarios (e.g. read hit/miss, write hit/miss, write-through, write-back) and the waveforms in these cases were manually verified to check correct operation. On integration with the other modules, much larger random test sets were added for all possible combinations of line size, cache size and write policy. Testing for correctness of memory accesses was automated by writing values into memory and reading them back. The addresses to write were generated so as to ensure the cases mentioned earlier got exercised.

Manual verification of caching mechanisms was not very convenient especially in the face of the quick verification iterations needed when the design was changed during synthesis runs. A better approach, on reflection, would have been to write an executable specification of the cache and compare the results produced off-line from this model with the VHDL simulation results.

3.2.4 Board Schematics Verification

Another important step in the verification effort was verification of the board netlist. The board schematics indicating connections between components, were drawn using Mentor Graphics' schematic capture tools [9]. The primary verification strategy was visual inspection. A VHDL dump of a portion of the schematics was generated. This had entities for the components connected through signals as in the schematics, with 'black-box' architectures. We substituted our simulation models for these components and carried out simulation.

Since the purpose of the simulation was to test connectivity, dummy models were written for some of the components. E.g. clock generation was done inside the crystal component, reset was generated inside the reset button etc. The memory models were extended with assertions at VCC and GND ports. Resistors on the board were also modeled with different architectures for series, pull-up and pull-down resistors.

We were able to discover a few potentially fatal errors in the interconnections, during our board-level simulation runs. Interestingly, most of the errors were found while setting up the simulation e.g. while doing the VHDL port maps, as we looked at the schematics from a different perspective. Some bugs were also discovered during simulation.

3.2.5 Verification Issues

Regularity of cache/memory designs presents the opportunity of developing a standard suite of tests for exercising most of the possible error scenarios. It seems feasible to define a set of test vectors to exorcise most bugs in the memory hierarchy. In our case, this methodology could not be formalized due to lack of time but this is certainly a possibility. This will prove enormously useful to memory hierarchy designers.

The final test-bench had eighteen commands of the form described earlier. In addition there were models of the environment including PLX, memory models and transceivers. Also included was code for automatically checking the results. Thus the test-bench grew to be quite complex. As a result, correcting errors in the test-bench itself consumed a significant portion of the verification time. Thus it is important for verification to proceed in tandem with design to ensure there is enough time to do a good job.

Using the right test vectors to detect most of the possible bugs is important. But an equally important thing is tracing the error to its source after it is known that a bug exists. This step, we estimate, consumed more than 90% of the manual effort spent on verification. The basic method was to run a set of test-vectors; if the test-bench detected an error, the waveforms were studied to find the source of the problem. This process was very cumbersome especially at the top-level of the design where there were hundreds of signals to sift through. The use of debugging aids was not found to be effective in reducing this time substantially.

3.3 Synthesis Methodology

Figure 4 shows the overall synthesis flow. The VHDL code for the design was synthesized to gates using Synopsys Design Compiler (DC). The necessary target library and Designware library for DC synthesis were provided by Altera. Gate level EDIF netlist generated from DC was passed on to Altera's MaxPlus for performing place and route [7].

We followed a top-down compile flow for synthesis in DC. In this flow, the designer specifies constraints for the top level design, sets synthesis directives for sub-designs and performs synthesis on the entire design hierarchy in one step. The class of constraints to DC can be divided into "Design Rule Constraints (DRC)" and "Optimization Constraints". In our scripts, we specified driving strength of the input pins and load capacitance at the output pins as DRC. For optimization constraints, we specified clocks, input delays, output delays and the false paths in our design.

The gate level EDIF netlist generated by DC and a constraints file was input to Maxplus for performing place and route, clock tree and reset tree synthesis. We experimented with different options of synthesis available from Maxplus to get the optimum settings in the constraints file for our design. FLEX10K devices have two dedicated nets for clock tree routing and four global nets [6]. These nets can be routed to all parts of the die with minimum skew. We used the dedicated clock nets for the two clocks in our design and one of the four global nets for the asynchronous reset.

Metric	Value
Total Area Estimated	2571.5 units (area of one LUT = 1 unit)
Critical Path Length (w.r.t 20ns clock)	58.71ns (slack = $-38.71ns$)
Critical Path Length (w.r.t 40ns clock)	26.02ns
Total number of LUTs in design	2365
Total number of FFs in design	1644
Total number of nets in design	4629

Table 1: DC synthesis results

Metric	Value
Maximum Clock Period (w.r.t. 20ns clk)	21.8ns
Maximum Clock Period (w.r.t. 40ns clk)	24.0ns
Number of Logic Cells used	3311 (approx 40K gates)
FPGA utilization	66%

Table 2: Maxplus synthesis results

3.3.1 Synthesis Results and Issues

The two clocks in our design were constrained at 66Mhz (15.15ns) and 33Mhz. These were later relaxed to 50MHz (20ns) and 25 MHz respectively. We performed a detailed reporting on the gate-level netlist. Flex10K FPGAs use different routing structures for local, row and column interconnects. When the design is not placed and routed, there is no information on the routing and hence the delays reported by DC here are highly unrealistic. However, we could get the nature of critical paths (e.g. fanout count, levels of logic) in our design by analyzing the timing reports. We used this information to specify additional constraints to DC and to alter the design for better results. The quality of results report from DC appears in table 1. The results are highly pessimistic.

The timing constraints to Maxplus are clock frequency, input setup time and clock to output delay. We iterated through different P&R runs on Maxplus to get the optimum settings for the optimization constraints. FSMs were designated as "cliques" to encourage Maxplus to place them as groups thus reducing internal routing delays for the modules.

To reduce the FPGA utilization and provide room for future growth to the design, we also tried mapping the design to a bigger FPGA. This device was unconventional compared to the other FLEX10K devices. In this device the row interconnects were slower than column interconnects. However, the Maxplus P&R algorithm, because of it's inherent assumptions about FLEX10K devices, placed highly interconnected signals along a row. So we did not get significant timing improvements in the larger FPGA. However when we migrated to a smaller FPGA for which the P & R algorithm is tailored, we got an jump in achieved clock frequency.

Table 2 shows the synthesis statistics after Maxplus place and route.

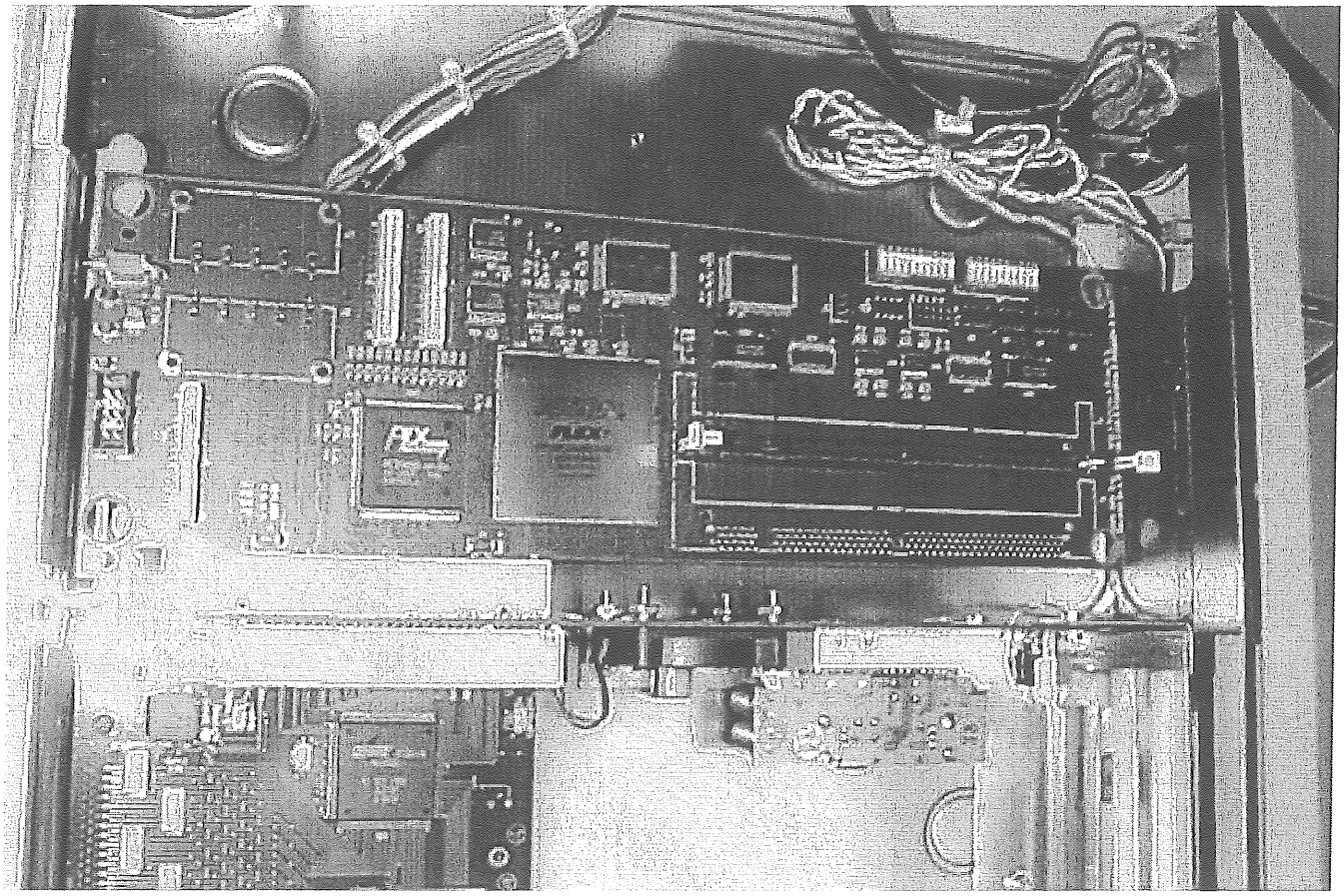

Figure 5: Photograph of AMRM Prototype Board (top half)

3.3.2 Issues with synthesis and place and route

We were unable to realize one-hot encoding for our designs using DC. The recommended methodology by Synopsys requires the designer to code the FSM such that the only registers in the FSM are the state registers. Thus all of the FSM outputs have to be non registered outputs. This type of FSM design is cumbersome when there is a large amount of datapath.

Since the quality of results reported by DC were unrealistic, we had to take the design through Maxplus place and route for each DC synthesis run. This was time consuming since one place and route run could be as long as 9-10 hours. This issue can be overcome to a good extent with the use of accurate wireload models for synthesis. Since FPGAs use non uniform routing, predicting an accurate wireload model prior to a place and route run is impossible. Thus it is essential that the designer generate a custom wireload model after an initial iteration on place and route. This custom wireload model should be used for next synthesis iterations. However, with Maxplus there is no backward path from post layout netlist to DC synthesis. Tighter coupling is needed between the synthesis and place and route tools to make design iterations faster.

Maxplus provides very little support for incremental design change once the design pin-out is fixed. In our case we discovered a small design bug after we got the PCB. To fix the bug we had to recode the VHDL to introduce an extra flip flop into the design. Since we already had the PCB designed, we had to use the same pin assignment as the PCB. Our design clock went up from 18ns to 21ns with one FF addition because of the additional pin constraints. Such results

from the P & R tool are unacceptable when trying to redesign the FPGA for implementing a different memory hierarchy.

4 Status and Future Work

The AMRM prototype board has been manufactured and is currently undergoing testing. The board implements the configurable memory hierarchy described in this paper. In its current form, the board can be used to perform efficient memory simulations for a variety of L1 cache configurations. Currently application adaptivity is implemented in software. After the current phase of testing we can start using the board for our experiments. Figure 5 shows a photograph of the board.

5 Conclusions

We have presented the design of a board-level prototype of an adaptive and reconfigurable memory hierarchy. The board can be used as a platform for experimenting with hardware implementations of adaptive caching mechanisms. The board is processor independent and can thus be used for experimenting with a variety of processor architectures.

The design, verification and synthesis methodology used has also been described. Several key EDA tools were used in the design. Our experiences using these tools and possible opportunities for improvement have been documented. The board designed using this methodology has already been manufactured and is available for experimentation. The next step is to test the board and run experiments on it.

References

[1] Andrew Chien and Rajesh K. Gupta, Morph: A System Architecture for Robust High Performance Using Customization, *Proc. Symp. on Frontiers of Massively Parallel Computing, Frontiers '96*, Oct 1996.

[2] Rajesh K. Gupta and Andrew Chien, Architectural Adaptation in MORPH, *Proceedings of SPIE Workshop on Configurable Computing*, Oct 1998.

[3] Alexander V. Veidenbaum et. al., Adapting Cache Line Size to Application Behavior, *1999 ACM International Conference on Supercomputing*, June 1999.

[4] *PCI 9080 Data Book, Version 1.05*, PLX Technology, Sept 1998.

[5] *VSS User Guide*, Chapter 16-18, Synopsys Online Documentation v1998.02 2.1.0, Synopsys Inc., 1998.

[6] *Flex 10K Embedded Programmable Logic Family Data Sheet*, Altera Corporation, Oct 1998.

[7] *MaxPlus II Getting Started*, Altera Corporation, Sep 1997.

[8] *Memory Modeler's User's Guide*, Denali Software, Inc., Jan 1997.

[9] *Design Architect User's Manual*, Software release C.2, Software version 8.6_2, Mentor Graphics, Aug 1998.