Smart Memories Polymorphic Chip Multiprocessor

Ofer Shacham, Zain Asgar, Han Chen, Amin Firoozshahian, Rehan Hameed,
Christos Kozyrakis, Wajahat Qadeer, Stephen Richardson, Alex Solomatnikov,
Don Stark, Megan Wachs, Mark Horowitz
VLSI Research Group, Stanford University

ABSTRACT

The Stanford Smart Memories polymorphic chip-multiprocessor architecture was conceived as a unified multipurpose hardware architecture base, capable of supporting a variety of programming models and per-application optimizations [17]. Backing the architectural claims, our team of PhD students set out to implement this challenging design in silicon, targeting $90nm$ technology. Now, with $55M$ transistors covering $61mm^2$, this is one of the most complex chips ever fabricated in academia.

Keywords

Stanford Smart Memories, Chip Multiprocessor, Memory System, Tensilica, Reconfigurable Design, Parallel Programming Model, Architectural/Design Exploration

1. MOTIVATION FOR THE PROJECT

Academia and industry as one are aggressively moving towards Chip Multiprocessors (CMP) as the main processing unit in current and future compute platforms. Yet the debate regarding the "right" programing model(s) for these machines, and hence how the memory system should be implemented, is far from being settled. While some advocate streams [10, 14, 19] due to their high compute density per Watt, streams have a more limited application space. Others prefer Thread Level Speculation [11, 21], recently generalized and formulated into the Transactional Memory model [15, 12], for its broader application domain and ease of use from the programmer's perspective. Meanwhile, the most prevalent general purpose platforms are still variations of cache coherent multi-thread models [6, 18, 13].

Stanford Smart Memories is a research project that aimed to build a single hardware platform that could support all these programming models by creating a flexible execution and memory system. We wanted to show that, from a hardware perspective, the similarities between the aforementioned programming models are greater than the differences. To achieve this goal, initially we planned to create a configurable data-path for the processor core, and add programmability to the memory system [17]. This conceptual design leveraged the fact that all memory models rely on physical memory storage in proximity to the processing unit, and on controllers that orchestrate data movements among these repositories and to and from the main memory. In addition, most memory systems need some "state" to be associated with the data (e.g., valid bit in caches or speculatively read/write bits in Transactional Memory), and therefore our memory system added meta-data bits to our local storage arrays. The differences between memory models were created by defining the meaning of the meta-data bits, and the protocols for the actions that need to be taken when specific conditions occur.[1] Having made these observations we decided to implement this polymorphic chip multiprocessor—the Smart Memories CMP. As we will see, some aspects of the architecture have significantly changed along the way, in order to allow a small design team to implement a complex ASIC design.

As the design process began, we quickly learned that while technology scaling had enabled the integration of an immense number of transistors on one die, leading to great CMP designs in industry, the complexity associated with efficiently utilizing these transistors typically renders CMP designs unrealizable in academia. In industry, a typical chip is the product of tens to hundreds of engineers leveraging years of accumulated methodologies. In contrast, academic teams are much smaller and typically attempt to tackle uncharted architectural frontiers. Moreover, if a new processor architecture is implemented, the design team must create not only the new hardware, but also the new software toolchain, which greatly increases the difficulty of the task. This paper provides an overview of the chip, but mainly focuses on the approach we took to allow a small academic team to complete this design. We strongly believe that although chip design is currently very expensive, the design and implementation of complex architectures has large value in academia, since it makes it very hard to sweep details under the rug, and we will show some of the changes in the architecture of the machine that arose from physical implementation considerations.

The Smart Memories (SM) eight-core CMP presented here is a fully implemented system: from an architectural simulator written in C++, through complete RTL/gate simulation environment, to the silicon chip currently being fabricated in $90nm$ technology at STMicroelectronics [3]. Four SM chips

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

[1] A complete analysis is out of this report's scope, but it can be found in this earlier paper [17].

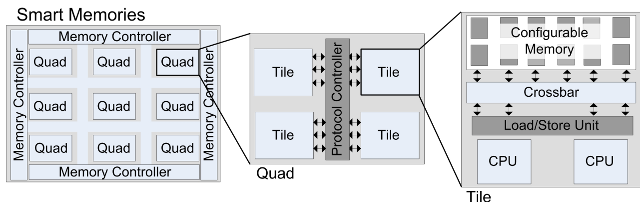

Figure 1: Smart Memories Architecture

will be integrated together in a system instrumented with additional FPGAs to provide full 32-core functionality. The software layer includes a C/C++ compiler[2] and runtime environments for the stream, transactional coherence and consistency (TCC) [12] and multi-thread execution models.

Throughout this paper we describe the different engineering efforts that facilitated production of the Smart Memories CMP. These include all aspects of the modern silicon process, from high-level architecture in Section 2.1 through the microarchitecture design in Section 2.2 and its associated verification collateral in Section 2.3. Section 2.4 then describes the physical implementation, including both methodology and final silicon statistics, and Section 2.5 describes the post-silicon testing methodology and infrastructure.

2. ENGINEERING EFFORTS

In mid-2005, the SM project's emphasis shifted from high level architectural study towards actual silicon implementation. In October of 2008, GDSII was streamed out to the ST fabrication plant. With an architecture, design, verification, implementation and software team totaling only 4 to 8 PhD students, many manpower and resource "design choices" had to be made throughout that period, and we emphasize these choices as we describe the Smart Memories engineering efforts.

2.1 Architecture

From the start we knew that we needed to construct a modular, reconfigurable architecture if we were to deal with VLSI wire and complexity issues. In our architecture work we initially planned to create a flexible data-path for the processor that would support different execution modes (light-weight threads, SIMD, etc.) [17]. However after completing much of the initial Verilog for this system, we realized that we were never going to have enough resources to complete the software support needed for this processor, and we needed to take a different approach. As a result we decided that we should focus our effort on the truly new part of the project, the programmable memory system, and try to leverage as much as possible existing solutions for the rest. Figure 1 shows the block diagram of the final architecture. Rather than building our own flexible processors, each of our *Tiles* now contained two Tensilica processors [9], thus providing us with most of the software tools needed for this project, while still allowing us to customize the processor. The Tiles also include several modular reconfigurable memory blocks called *Mats*, and a crossbar connecting them to the processors. Four Tiles are then connected to a shared local programmable *Protocol Controller* (PC), forming a *Quad*. The Quads are connected to each other and to main-memory controllers using an interconnection network. To reduce complexity, I/Os, and area of this first implementation, the *Network Switch* and *Memory Controller* are mapped to a host FPGA (Section 2.5).

Leveraging Tensilica processors, SM could now focus on the programmable memory system. This consists of the three major blocks highlighted in Figure 1: the processors' memory interface or *Load Store Unit* (LSU), the reconfigurable Mats and the PC. Since Tensilica enables the creation of new instructions and its processors can generate specialized memory system accesses, the LSU serves as the interface adapter between the Tensilica cores and our memory system. It performs Virtual to Physical address mapping, can generate multicast requests for set associative lookups, and can even generate parallel accesses to different mats to support some operations.

The second reconfigurable block is the array of Mats. Figure 2A shows a block diagram of a single Mat: each Mat in the Tile is an array of data words and associated meta-data bits. It is these meta-data bits that makes the memory system flexible: meta-data bits store the status of each data word and their state is considered in every memory access—each access to this word can be either completed or discarded based on the status of these bits. For example, when Mats are configured to form caches, these bits are used to store the cache line state, and an access is discarded if the status indicates that the cache line is invalid. Meta-data bits are dual ported and can be both read and updated atomically with each access to a data word (the meta-data's update function is set by the Mat internal configuration). In addition, a built-in comparator and a set of pointers allow the Mat to be used as tag storage (for cache) or even as a FIFO. Mats are connected to each other through an *Inter-Mat Communication Network* (IMCN) that communicates control information when the Mats are accessed as a group.

The final part of our reconfigurable memory system is the Protocol Controller. This reconfigurable engine executes sequences of basic operations, composed based on the memory model, to service the requests that are sent to it. These requests include moving data (e.g., DMAs to local memories, or cache spills and refills), updating memory state (e.g., cache coherence operations or full/empty bits for synchronization), or both (e.g., committing a transaction). To perform these tasks, it must track outstanding requests, and enforce correct order of operations when needed. The PC is connected to a network interface port and can send and receive requests to/from other Quads or Memory Controllers.

Mapping a programming model to the Smart Memories architecture requires the configuration of the LSU, the Mats, the Tile interconnect and the PC. For example, when implementing a shared-memory model, memory Mats are aggregated (using IMCN) to form the caches (Figure 2B), and the Tile crossbar is configure to route the processors' access information appropriately. The meta-data bits in tag Mats are configured to serve as line state bits (e.g. Modified, Shared and Exclusive), enabling the PC to act as a cache coherence engine, which refills the caches and enforces coherence.

The Mats introduce an interesting tradeoff in risk and performance. In 2004 we completed the implementation of our Mat [16], to prove that the overhead of this structure need not be large. While we had working silicon early, we decided it was not worth the risk and design cost to use custom circuits and cells in our ASIC. Instead we choose an implementation that uses only standard cells, even though

[2]Tensilica compiler instrumented with special Smart Memories TIE instruction [9].

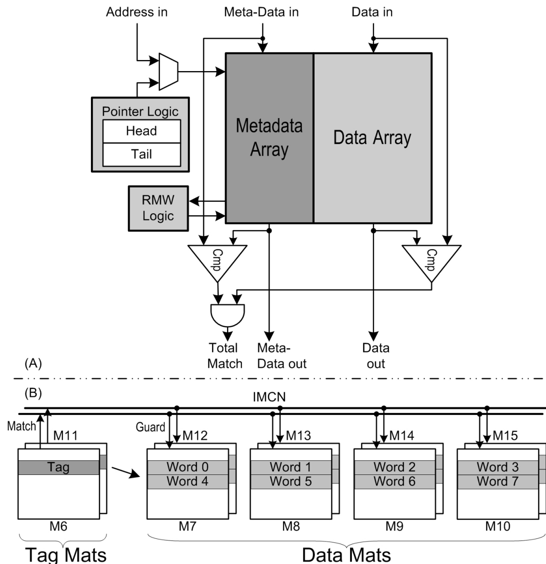

Figure 2: Configurable local memory. (A) Block diagram of the Memory Mat. (B) Example Mat organization for a 2-way cache.

the hardware cost of the programmable memory blocks is quite high in this approach.

2.2 Micro-Architecture Highlights

While we would like to think about the micro-architecture of CMPs as a simple aggregation of processor IP cores, in reality there are always adjustments to make, such that the processor is better suited for the memory system. Tensilica's Xtensa Processor Generator allowed us to do these modifications easily. The base Xtensa architecture is a 32-bit RISC instruction set architecture (ISA) with 24-bit instructions and a windowed general-purpose register file. We used pre-defined options such as floating-point co-processor (FPU) and integer multiplier and divider, but we also defined custom instruction set extensions using the Tensilica Instruction Extension language (TIE) [9]. The TIE compiler generates a customized processor, taking care of the low-level implementation details such as pipeline interlocks, operand and bypass logic, and instruction encoding. In addition, Tensilica's Processor Generator produces customized software tools: C compiler, linker, debugger, and application libraries, significantly reducing design time and effort.

For the micro-architecture of the flexible memory system controller, we decided to take a "RISC" like approach: instead of providing complex pre-defined operations, we provided a small number of basic operations and implemented complex data and state manipulations by executing a set of these basic operations. Unfortunately we could not literally use a RISC processor, since such a solution would not match the latency, throughput and power constraints of the system.

Figure 3 illustrates the internal organization of the Protocol Controller. The execution core consists of three major units: the Tracking and Serialization (T-Unit) serves as the entry point to the execution core of the controller. It stores and retrieves information of outstanding memory requests using either the *Miss Status Holding Registers* (MSHR) (for memory operations that must enforce ordering), or the *Uncached Status Holding Registers* (USHR) (for non-ordered operations). The State Update (S-Unit) performs read, writes and manipulation of the state information associated with data blocks (e.g., cache tags and cache line state). It therefore has a dedicated port to the Tiles' memory Mats. Finally, the Data Movement Engine (D-Unit) provides necessary functions for reading and writing data blocks from the Mats into an internal storage structure (*Line Buffer*). The controller is also equipped with independent DMA channels, which are programmable request generator engines.

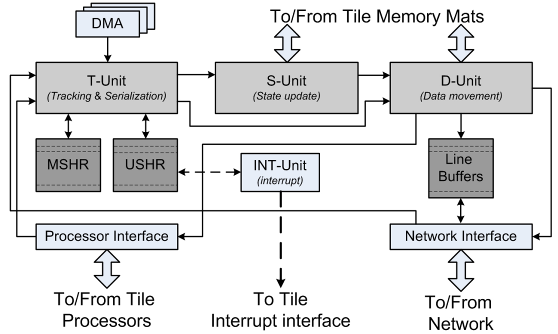

Figure 3: Protocol Controller micro-architecture

This approach works well because across many different memory models, the functions of all protocol controllers remain very similar: at their core all protocol engines track and move data. One can recognize such similarity at two levels: at the high level, many protocol actions that implement a memory model have the same conceptual functionality. At a lower level, the hardware operations that are combined to form the protocol actions are the same and can be categorized into five different classes: Data/State read and write, communication, ordering, tracking, and state information interpretation. The abstractions and instructions used by this controller are described in more detail in [8].

2.3 Logic Verification

Seven separate verification environments accompanied the design to enable verification even when the design phase was far from completed. At the lowest level, we started with an environment to stress-test the Memory Mat RTL, by writing a C++ reference model and instantiating both inside a simple Verilog testbench. We used random inputs and checked the RTL outputs against the C++ model.

We created the final six verification environments for testing at the Tile level (two processors and local caches), the Quad level (eight processors, protocol controller, complete memory system), and the Four-Quad level (32 processors, network switch, and up to four memory controllers). These environments were written mostly in OpenVera [22]. In cases where key RTL components were still missing (such as the Tile level environment that lacked the protocol controller's RTL), we used the architectural C++ simulator to provide any missing functionality. At each level of complexity we created a pair of environments: one "shim" environment without processor RTL, and one including processor RTL. In the shim environments, we used Vera to generate random memory accesses at the processor interfaces. In the testbenches with processor RTL, we simulated compiled applications, including Splash-2 [24] kernels. Our final regression suite included tests at the Memory Mat, Single-Quad, and Four-Quad level, both with and without processor RTL.

As RTL verification is becoming the biggest bottleneck

in today's chip design efforts, and given our limited manpower, we designed our testing environments with a focus on reusability. Each environment level was carefully chosen to have a well-defined interface, both from the logical and from the physical perspective. In this way, we were able to leverage portions of Tensilica's processor testbench in the Tile environment (instantiated once per processor), portions of the Tile testbench were used in turn at the Quad level, and the Quad testbench was used in the Four-Quad level. A second advantage of this approach was for gate level simulation. Since the verification environment was in sync with the architectural hierarchy, very few modifications were needed once synthesized/placed-and-routed/timing-back-annotated netlists were available.

A major challenge we had when verifying the system was enabling random testing. We wanted to create a reference model for the memory system that would check that all memory transactions were conforming with the configured memory protocol. However, since the protocols are not deterministic, and different configurations of the chip could result in drastically different correct outcomes for a series of transactions, we decided to build a new kind of reference model. Unlike conventional reference models, which predict a single correct output at check time, our model relaxed this constraint and allowed a set of possible outcomes, as long as the design obeyed the protocol. This "Relaxed Scoreboard" [20] enabled us to run strenuous random vectors on the design to cause arbitrations, back-to-back misses, unusual timings of operations, etc. We used this tool in the Single-Quad and Four-Quad environments, with and without processor RTL, and found many design errors even when self-checking diagnostic tests were passing.

In parallel with our software simulations, we made use of a BEE2 Board [7] to prototype our design on FPGA. Because the fully reconfigurable chip was too large, even when split up across multiple FPGAs on the board, we hard-coded configurations before FPGA synthesis. We also replaced the processors with simpler versions and only put one per tile instead of two. We then let the synthesis tool strip away irrelevant logic. In this way we could fit one tile per FPGA, with the protocol controller and memory controller on a separate FPGA. Following this methodology, we were able to run more benchmarks on the FPGA much faster, and with much larger data sets, than was feasible in simulation.

2.4 Physical Implementation

Conceptually, physical implementation comes after the logic design and verification phase has been completed. However, in reality these two efforts are interlocked, since the physical implementation process provides feedback regarding the feasibility of the logic design. Often, changes in architecture and logic design have to be made, especially when the design team does not have a lot of experience. As a result, we started physical design more than a year before tape-out, and well before the logic design was complete.

Early in the process, we established that the critical timing path is between the Mats and the processor. Specifically, whenever the processor loads a value from a Mat, it expects the result value within two clock cycles. If the value is not available (e.g., cache-miss) the processor must be stalled. This means gating the processor clock at least half a cycle before it reaches the positive edge, due to the clock tree propagation delay. In order to improve the physical implementability without adding more pipeline stages, thus compromising performance, the entire processor core was moved to the negative clock edge. This gave the clock stall logic an entire clock cycle rather than a half, and significantly improved timing closure. We were able to do this because Tensilica has good margin in their address generation pipeline stage.

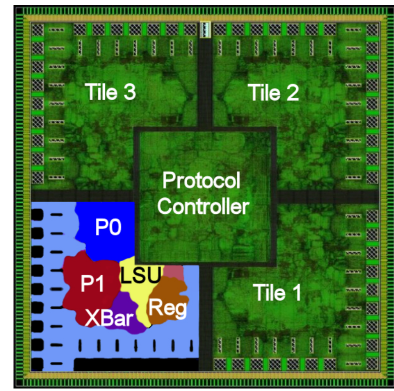

Figure 4: Smart Memories Die Plot and Floorplan

A similar trade off exists on the silicon boundaries, between the relatively slow I/Os and the much faster on-chip logic. Therefore, SM contains an adjustable I/O timing mechanism: Four fixed ratios of $1, \frac{1}{2}, \frac{1}{4}$ and $\frac{1}{8}$ of the core clock frequency are allowed for the IO rate control and skid buffers accommodate for the clock domain crossings. The I/O rate control logic allows the design to run quickly even if there are any unforeseen issues with the off-chip interface.

With feedback being so critical, in order to enable a small team to manage and implement a relatively complex design, we adopted a top-down design-flow methodology. Logical hierarchies were preserved, and physical partitions were established to keep sub-blocks within reasonable complexity. This allowed the sub-blocks to be implemented and optimized in parallel, thus reducing tool run-time and improving design feedback turn-around time. While tools work well when one knows how to use them, new teams frequently run into tool problems. We were not different, but were very fortunate that one team member also works at Synopsys™.

Along the road, the design changed many times to make the physical design better. Some of the more interesting changes are described next, and really have to do with creating "memory" like structures. For example, since we were using only standard cells, the meta-data bits associated with the Mats were implemented by flops. Initially these registers had a controlled clock instead of an enable signal to control write accesses. However, since there are about 48K of these in each Tile, the clock tree synthesis tool had difficulties when generating clocks and balancing skews. A simple fix was to do fine grain clock gating insertion during synthesis, while using enables for smaller groups. This made it consistent with the rest of the design which had integrated clock gating (ICG) cells used to control 8-128 flip flops.

Most of the synchronous access memory structures were implemented using block RAMs from STMicroelectronics (as expected). However, the design also uses a number of asynchronous memories, such as Ternary Content Addressable Memories (TCAM) and static configuration registers. Unfortunately, our vendor-generated memories were synchronous and the design of the TCAM and configura-

Table 1: Physical properties summary

Attribute	Value
Die size Core size	7.8mm x 7.8mm 7.15mm x 7.15mm
Transistor count	55$million$
Layout density	67% (excluding block RAMs)
Operating voltage	1.0V (core); 1.8V (I/O)
Clock speed	181MHz (core); $22-181MHz$ (IO)
Total Power	*1.348Watt* Coherence / *1.350Watt* TCC / *1.406Watt* Stream
Tile cells breakdown PC cells breakdown	2% HVT; 88% SVT; 10% LVT 2% HVT; 90% SVT; 8% LVT
Scan chains	8 PC pos-edge chains * 7671 flops 3 Tile pos-edge chains * 7923 flops 3 Tile neg-edge chains * 9992 flops 32 chains total per Quad
Package	384 pin custom plastic BGA
IO Pins	389 Total (70 core Vdd, 71 core Gnd, 22 IO Vdd, 22 IO Gnd)

tion memories required an asynchronous lookup, to avoid unnecessarily stalling the system. We decided that non-scanned flip-flop arrays best suit our needs: Since these were mostly configuration memories, the flops-based solution enabled the machine to power-up to a usable state (using default set/reset values). The clock load from these flip-flops is not substantial since they are clock gated during runtime.

Throughout the design process we also realized the importance of RTL coding style on the physical quality. We were able to improve the performance of several structures by modifying either their RTL code or their implementation, while not changing functionality. We also had several RTL vs. netlist equivalence checking failures due to RTL coding issues. These generally involved unsafe constructs such as `casex`, `casez` and synthesis pragmas. In order to fix these problems we completely banned the use of `casex` and added assertions to the `casez` statements to make sure none of the inputs were ever invalid.

Our design contains four identical Tiles making it an obvious decision to implement once and replicate. Since we had several hundred interface wires between the Tile and PC we wanted to efficiently allocate and align the pins across the boundary to minimize the global routing channel requirements. A rectangular structure would have yielded a non-optimal floorplan, therefore we chose a rectilinear structure as shown in Figure 4. Unfortunately, several tool problems arose during pin assignment with non-unique rectilinear blocks, and with limited manpower, manual assignment was not an option. To overcome this obstacle, we uniquified the Tiles for floor-planning and pin assignment. After floor-planning was complete one of the tiles was copied into place of the other tiles creating a non-unique, but fairly optimal, floor-plan.

As with most physical implementation flows, several different tools from multiple vendors were used. We used Synopsys [4] Design Compiler for synthesis, and JupiterXT, IC Compiler and Astro for physical implementation. Synopsys Formality was used for formal verification of the RTL vs. the netlist and netlist vs. netlist during the physical flow. Mentor Graphics [2] Calibre was used for physical validation after the design flow was completed.

Table 1 summarizes some of the final statistics such as area, power consumption, operating frequency and more.

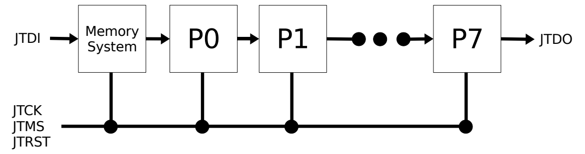

Figure 5: Smart Memories JTAG TAPs

2.5 Design for Testability

Post-silicon validation or *Bring-Up* is key in any chip design, and takes on a special meaning when resources are limited. The principal *Design for Testability* (DFT) goal we set for SM was that the exact state of the machine can be read at any point. Therefore, we used the Synopsys tool flow to add a *ScanTest* mode, in which all flops in the design are serially connected in dozens of chains (as noted in Table 1). Similarly, to enable testing of I/O and connectivity, all input and output pads[3] can be chained into one *Boundary Scan Register* (BSR). To avoid some of the design complexity associated with these structures, our system can scan the exact state out, but can not regain functional mode afterwords (unless reset). The "detection but no recovery" methodology reduces pressure on designers to avoid memory structure corruption when in scan mode.

While scan chains are an efficient tool for validating physical implementation issues, they are cumbersome when it comes to extraction of logic state. Therefore, the entire physical address space of the machine is designed to also be accessible through a JTAG test access port (TAP) controller [5], supporting both read and write operations. We leveraged the eight Tensilica processors' JTAG TAPs by chaining the *JTDI* and *JTDO* signals, and distributing the *JTMS*, *JTCK* and *JTRST* signals in parallel [23],practically creating a 9-wide JTAG composite controller (Figure 5). This method enables the use of any subset of JTAG TAPs in parallel, as well as the use of any subset of the Tensilica *On-Chip-Debuggers* (OCDs) [23], while keeping the I/O pin count minimal. In addition, for key debug signals (e.g., *Tile_req*, *PC_ack*, *mat_select*, etc.), a 12-bit bus, which can select any of 384 unique signals, was added. These signals' timing constraints were relaxed since they are used only in debug mode where clock frequency is reduced.

Special care and thought were given to the power-on/reset sequence as this can be Achilles' heel of any design, and all the more so in a reconfigurable design. Two signals, *Reset* and *DisableBootProc*, control Smart Memories' configuration and boot sequence. A positive edge on *Reset* places Smart Memories in its default configuration—caches disabled, instructions fetched directly from off-chip, etc.—and starts running one of the eight on-chip processors, known as the *boot processor.* The boot processor can then provide further configuration of the chip, say for a TCC or streaming memory model, and is able to turn on and off any and all of the other seven processors. Note that once the chip is running, any of the eight processors can control and configure itself and/or any of the other seven processors on the chip.

The *DisableBootProc* signal allows further control of the boot sequence. With *DisableBootProc* in its normal OFF position, the boot processor is enabled and boot proceeds as outlined above. With *DisableBootProc* ON, however, posedge *Reset* still configures the chip, but the boot processor remains off and nothing runs. In this state, for testing

[3]With the exception of JTAG and Clk pads.

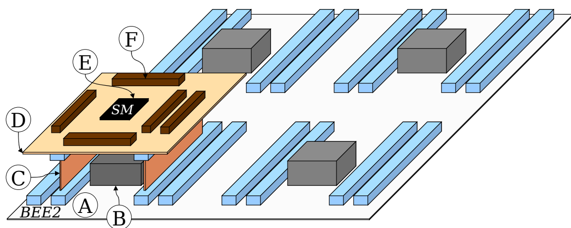

Figure 6: Smart Memories Testing System A) BEE2 Board B) Control FPGA C) Custom double-ended DIMM cards replace cables D) Custom daughter card E) Smart Memories Test Chip F) Five 40-pin headers compatible with logic analyzer

purposes, the chip can be further reconfigured via JTAG. Then, when *DisableBootProc* is again turned off, execution continues as usual.

One problem with Power-on/Reset sequences is that they generally have a long cycle count, thus a long simulation time. However, they still have to be thoroughly verified. Worse still, SM's memory system can be configured as one of dozens of cached, streaming and TCC configurations. Therefore, each of our RTL simulations could run in one of three modes. Configuration could be (a) forced by simulator into internal registers (fastest); (b) handled by executing boot processor instructions; (c) inserted by JTAG (slowest). Gate level simulations were also performed using methods (b) and (c), further boosting verification confidence levels.

With silicon in hand (expected Jan '09), we plan a bring-up on a specially designed, yet conceptually simple, system. As shown in Figure 6, this is a cableless system, connecting each SM chip to an FPGA on a BEE2 board [7]. A control FPGA on the BEE2 board uses its extra DIMM slots to interface with our custom daughter-card through double-ended DIMM cards. The DIMM cards act as 50Ω transmission lines. This design enables a quick bringup of the chip and alleviates concerns about signal integrity. Both the double-ended DIMM cards and daughter-card were designed using the Cadence Allegro toolchain [1].

3. CONCLUSIONS

Smart Memories is one of the largest and most complex processor designs to be completely designed and implemented at a university. To bring the concept from dream to reality has required discipline and hard work on the part of a dedicated team of students. Key to success has been an ongoing emphasis on test and verification, effectively captured by the motto "Test early and test often." The design process was made tractable by judicious decisions along the way, such as the choice to use customized Tensilica processor cores instead of fully-custom data paths, or the similar choice to use standard-cell rather than custom memory mats. At the end of a long road, the chip has been taped out and is in the process of fabrication. We look forward soon, finally, to test working silicon in the lab.

4. ACKNOWLEDGMENTS

This research was supported by DARPA grant F29601-03-2-0117, Advanced Micro Devices Inc, and Stanford Graduate Fellowships. Ofer Shacham is partly supported by the Sands Family Foundation. We would like to acknowledge and thank Tensilica™ for providing the processing units for Smart Memories, and STMicroelectronics™ for fabricating the chip. Finally, we would like to acknowledge Francois Labonte, Ken Mai, Ron Ho and Kyle Kelley for their contribution to the realization of the Smart Memories chip.

5. REFERENCES

[1] Cadence allegro PCB toolchain. http://www.cadence.com/products/pcb.
[2] Mentor graphics. http://www.mentor.com/.
[3] STMicroelectronics. http://www.st.com/index.htm.
[4] Synopsys. http://www.synopsys.com/.
[5] IEEE standard test access port and boundary-scan architecture. *IEEE Std 1149.1-2001*, pages i–200, 2001.
[6] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, Dec 1996.
[7] C. Chang, J. Wawrzynek, and R. W. Brodersen. Bee2: A high-end reconfigurable computing system. *IEEE Design & Test*, 22:114–125, 2005.
[8] A. Firoozshahian. *Smart Memories: A Reconfigurable Memory System Architecture*. PhD thesis, Stanford University, 2009.
[9] R. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20(2):60–70, Mar/Apr 2000.
[10] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8. ACM, 2006.
[11] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE MICRO*, pages 71–84, 2000.
[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, and K. P. Manohar. Transactional memory coherence and consistency. In *International Symposium on Computer Architecture (ISCA '04)*, page 102, 2004.
[13] M. Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31(8):28–34, Aug 1998.
[14] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The imagine stream processor. *Computer Design, International Conference on*, 0:282, 2002.
[15] J. R. Larus and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 1(1):1–226, 2006.
[16] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. Horowitz. Architecture and circuit techniques for a 1.1-GHz 16-kb reconfigurable memory in 0.18u CMOS. *Solid-State Circuits, IEEE Journal of*, 40(1):261–275, 2005.
[17] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 161–171, 2000.
[18] P. E. McKenney. Memory ordering in modern microprocessors, part I and II. *Linux J.*, 2005(136/7), 2005.
[19] J. Owens. Streaming architectures and technology trends. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9, New York, NY, USA, 2005. ACM.
[20] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz. Verification of chip multiprocessor memory systems using a relaxed scoreboard. *Microarchitecture, 2008. The 41st Annual IEEE/ACM International Symposium on*, 2008.
[21] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, 2005.
[22] Synopsys, http://www.open-vera.com/. *OpenVera*.
[23] Tensilica. *Tensilica On-Chip Debugging Guide*. 2007.
[24] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, pages 24–36, Jun 1995.