

A Mechanized Refinement Framework for Analysis of Custom Memories

Sandip Ray
University of Texas at Austin
sandip@cs.utexas.edu

Jayanta Bhadra
Freescale Semiconductor Inc.
jayanta.bhadra@freescale.com

Abstract—We present a framework for formal verification of embedded custom memories. Memory verification is complicated by the difficulty in abstracting design parameters induced by the inherently analog nature of transistor-level designs. We develop behavioral formal models that specify a memory as a system of interacting state machines, and relate such models with an abstract read/write view of the memory via refinements. The operating constraints on the individual state machines can be validated by readily available data from analog simulations. The framework handles both static RAM (SRAM) and flash memories, and we show initial results demonstrating its applicability.

I. INTRODUCTION

This paper describes an approach to verify embedded custom memories. Memory verification entails showing that a transistor implementation conforms to the high-level view of a state machine that stores and retrieves data at addressed locations. Memories are complex analog artifacts, optimized for performance, area, power, etc., and account for about half the real estate and more than 50% of the transistor count of a microprocessor. This makes their verification a critical component of the overall design validation. However, given the size and complexity of a custom memory core, it is impossible to validate the entire core by analog simulation. Thus, a key challenge is to derive an effective abstraction which can be formally compared against the high-level specification.

The common approach to abstract a traditional SRAM is to extract a *switch-level* model [1] that represents the memory netlist as a set of *nodes* connected by transistor *switches*. Each node has state 0, 1, or X; each switch has state “open”, “closed”, or “indeterminate”; state transitions are specified by *switch equations*. These models capture many aspects of transistor circuits, namely bidirectionality, signal strengths, etc. The common analyzers for constructing such models are the ANAMOS [2] and its variants; they partition a netlist into *channel connected subcomponents* (CCSs) and analyze each component separately to construct the switch equations.

However, in spite of their sophistication, switch-level analyzers ignore many analog effects. For instance, the *strength assignment* procedure in ANAMOS produces a significant mismatch with detailed analog simulations for netlists containing transistors of closely matching but different strengths [3]. While these discrepancies can be ameliorated by designing more and more accurate analyzers [4], such an approach does not solve the fundamental problem of effectively representing inherently analog behaviors with equations in a discrete

algebra. The problem is exacerbated with the advent of flash memories that contain both regular and *Floating Gate* (FG) transistors; FG transistors “break” the view of netlists as a collection of switches, making switch-level analysis untenable.

In this paper, we present a new approach to abstracting memory implementations. Instead of extracting a switch-level model by *structural analysis* of a transistor netlist implementing a memory core, we formalize its *behavior* as a system of interacting state machines. The viability of the method is based on the observation that in an industrial design flow, custom memories are designed *not* as an ad-hoc transistor network but by interconnecting small, cohesive units such as *bitcells*, *sense amplifiers*, etc., with well-understood electrical properties in their range of operation. For instance, in Motorola’s design flow, the units are carefully architected to operate over a limited sequence of *certified* stimulus patterns, each of which is validated by extensive analog SPICE simulation across various process corners and operating conditions [3]. Thus, it is natural to formalize the *behavior* of each unit as a state machine using guarded transitions that encode its operating constraints. This enables us to reduce a memory implementation to a formal behavioral model specifying an interacting composition of state machine components, with each component representing the behavioral model for a pre-computed unit. Note that by focusing on the *behavior* of the units, our approach is agnostic to the nature of the transistors (standard, FG, etc.) used in the implementation, making it applicable to both SRAM and flash designs. Finally, we show how to prove a refinement theorem relating such compositions with high-level memory specifications using an assume-guarantee technique.

Our approach is mechanized in the ACL2 theorem prover [5]. We show how the use of the expressive language of a general-purpose theorem prover enables effective compositional reasoning about the interacting state machines.

II. MODELING CUSTOM SRAM

We illustrate our approach with the bitcell implementation shown in Fig. 1. From an *electrical perspective*, reading from the bitcell can be explained by the following operations.

- Initially, the precharge (*pch*) signal turns 1 in order to precharge the bitline (*bl*) and bitline-bar (*blb*) to 1s.
- Next, the wordline (*wl*) turns 1 indicating that the decoded address matches that of the bitcell. Thus the data stored in the bitcell (say 0) and its complement (1) are gated to

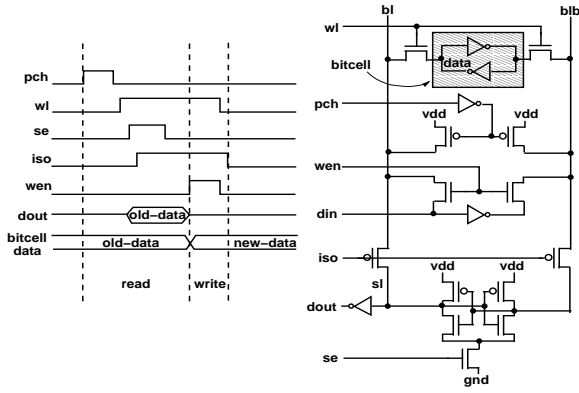


Fig. 1. A Standard SRAM Bitcell and Associated Timing Diagram

bl and blb respectively. Due to the large load on bl , the bitcell cannot pull it down from 1 to 0; bl is pulled down to an indeterminate value $(1 - \delta)$ (where $0 < \delta < 1$), while blb retains a 1. Furthermore, since the isolate (iso) signal is 0, the sense-line (sl) is pulled down to $(1 - \delta)$.

- Finally, the sense (se) signal, followed by iso , turn 1 resulting in (i) an electrical disconnection between bl and sl , and (ii) converting the bottom 5-transistor pack into a latch that pulls the value of sl from $(1 - \delta)$ to a solid 0. The value is then inverted and is obtained at $dout$.

Note that reading a bitcell is a complex analog problem — how to drive a *large* load (a bitline) with a bitcell that is quite *small* in strength. Furthermore, the circuit functionality depends on the constraints on the relative times within which successive signals change value. For example, wl must be 1 for a pre-determined bounded interval in order for the bitcell value to drive bl from 1 to $(1 - \delta)$. Nevertheless, the *behavior* of the signal pattern can be viewed as discrete state transitions: if the data stored in the bitcell is 0, then under appropriate conditions bl transits from state 1, through $(1 - \delta)$ and so on until finally the circuit produces 1 at $dout$.

Our approach works on the bitcell design as follows. We identify the design as a composition of state machines that correspond to the following two components:

- 1) the *bitcell* (from the shaded region together with the structure of the pch and iso), with the wl , pch , and iso signals as input and bl and blb as output, and
- 2) the *sense amplifier* (from the 5-transistor pack together with the transistors gated by iso), with bl and blb as inputs and $dout$ as the output.

Each state machine is pre-computed into a “library”. To make the library *generic*, the state machines are parameterized to work over a range of operating constraints. For instance, to model the time delay between pch and iso signals, the bitcell component contains parameters n_0 , n_1 , and n_2 (among others), with constraints that on a *read*, (i) pch is 1 and iso is 0 for at least n_0 units, (ii) both pch and iso are 0 for at least n_1 units thereafter and wl becomes 1 in this interval, and (iii) iso is 1 for at least n_2 units subsequently. For the reader familiar with ACL2, the parameters are modeled using encapsulation

(with constraints specifying operating conditions) and can be functionally instantiated [6] for concrete models.

We have used our library to model the transistor implementation of a simple but complete SRAM core consisting of an *array* of memory words, each word composed of a *row* of the bitcells in Fig. 1 (the sense amplifier being shared along a *bitcell column*), together with a decoder implementation. This example demonstrates scalability of the approach. A strength of the approach is compositionality: the extracted formal model of the memory core is merely a hierarchical composition of those of the individual bitcells, with intermediate state machines specifying the behavior of the glue logic.

III. MODELING FLASH MEMORY

Although we can abstract custom SRAMs, our principal goal is to develop a framework for handling flash memories. The additional complexity in flash arises from FG transistors, which have, in addition to the conventional *drain* (D), *gate* (G) and *source* (S) terminals, a *floating gate* (F) — a polysilicon layer inserted in the oxide between the gate and the substrate that is physically disconnected from both S and D. A detailed treatment of flash memories is provided by Cappelletti *et al* [7]. The key electrical effect is the capacitive coupling between G, F, and the substrate. The capacitance is exploited to design a bitcell with a *single* FG transistor as follows. Controlling the stored charge in the capacitive coupling allows dynamic regulation of the threshold voltage V_{th} (the minimum voltage to turn on the device); a low threshold voltage (V_{th}^L) represents logic 1 and high threshold voltage (V_{th}^H) represents logic 0. Additionally, some flash designs make use of *multiple* V_{th} levels to store 2 or 3 bits in one FG transistor; we do not consider multi-level flash in this paper.

Unfortunately, the capacitive coupling mentioned above breaks the simple view of a transistor as an on/off switch, as taken by ANAMOS-like analyzers, and makes it infeasible to extract precise switch-level abstractions. Consequently, current industry practice on flash validation amounts to (i) simulating the high-level model along with the encompassing SoC, and (ii) simulating individual FG bitcells through SPICE simulations. In particular, *no* formal correspondence is guaranteed between the transistor netlist and high-level specification.

Before describing our behavioral models, we discuss the electrical effects of flash operations. Below we summarize the three main operations of an FG bitcell: *read*, *program* (writing 0), and *erase* (writing 1). The operations involve both the bitcell and the surrounding control logic.

- **Read:** For the selected bitcell, one applies a voltage v ($V_{th}^L < v < V_{th}^H$) at G which is driven by the selected wordline, while keeping other wordlines at ground. If the cell has logic 0, the transistor does not turn on and no current flows to the associated sense amplifier; otherwise the bitcell turns on and current is detected, reading a 1.
- **Program:** The so-called *Channel Hot-Electron Injection* procedure is performed to inject negative charge into the FG, raising its V_{th} to V_{th}^H . Then there is a verification phase to ensure that V_{th} has been appreciably raised;

this is done by “reading” the cell with a gate voltage $v (> V_{th}^H)$. A result of 0 for the read indicates successful programming; otherwise programming is iterated until it succeeds or a specified number of attempts have been made, signalling failure in the latter case.

- **Erase:** Erasing is performed for an entire memory sector rather than one bitcell, and is based on removal of stored charge by a procedure called *Fowler-Nordheim tunneling*. The operation involves (i) *raising* the V_{th} s of the bitcells in the sector to V_{th}^H by programming, (ii) charge removal to lower all the V_{th} s to V_{th}^L , and finally, (iii) normalization, which employs *soft programming* to increase the V_{th} of the cells that have fallen below V_{th}^L .

The description underlines the complexity of the analog operations in a flash memory, and points to the difficulty of designing switch-level analyzers. Other factors to account for in abstracting flash memories include (i) multiple voltage levels, (ii) charge injection and removal, and (iii) complex sense amplifier activity to compare various current values. However, the *behaviors* of the individual components are still tractable (albeit more complex than SRAMs). For instance, the response of the state machine for the FG bitcell component to the electron injection phase of a *program* sequence is formalized as a non-deterministic transition raising the V_{th} by a bounded constant. Our library contains behavioral state machine models for the different components of flash memory, such as bitcell, sense amplifier, etc. Note that a few of the generic components are reused from the SRAM library.

We used behavioral abstractions to formalize a standard implementation of a NOR flash core:¹ bitcells are arranged in a two-dimensional array with a *row decoder* and a *column decoder*; a *read* of a bitcell causes a row to be activated by the row decoder, while the column decoder causes the appropriate column to be connected to the sense amplifier resulting in the loading of the data from the addressed bitcell to the output buffer. The extracted model is a composition of the behavioral models corresponding to the bitcells, the (row and column) decoders, the sense amplifiers, and the output buffers.

IV. SPECIFICATION AND VERIFICATION

We relate the executions of the memory core with a high-level specification. The specifications are abstract state machines representing the core’s interface to an SoC during functional verification. The SRAM specification supports *read*, *write*, and *reset* operations; the flash specification supports *read*, *program*, and *erase*, together with *core enable* that controls operations on the entire core, and *write protect* that regulates programming bitcells in the core.

We prove that the implementation is a *simulation refinement* [8], [9] of the specification up to stuttering, with respect to a *refinement map*. A refinement map enables us to appropriately view implementation states as specification states [10], and in our case, maps the bitcell states in the memory core

to an association list that models the core at the specification level. We require the notion of correspondence to be stutter-insensitive to account for the timing mismatch between the implementation and specification models.

We now discuss the proof obligations. Let *rep* be a refinement map. We then define predicates *inv* and *commit*, and a function *pick* such that (i) *inv* is an implementation invariant and (ii) the following formulas are provable:

1. $\forall s, i : inv(s) \wedge \neg commit(s, i) \Rightarrow rep(impl(s, i)) = rep(s)$
2. $\forall s, i : inv(s) \wedge commit(s, i) \Rightarrow rep(impl(s, i)) = spec(rep(s), pick(s, i))$

Here *impl* and *spec* are the (non-deterministic) state transition functions of the implementation and specification respectively; *commit* governs for an implementation transition if the specification transits or stutters; *pick* provides the specification stimulus in case of a matching transition. The formulas above thus state that for each transition of the implementation, the specification either has a matching transition or stutters.² These proof rules, of course, can be used to compare two systems modeled at different abstraction levels; they have been adapted from Manolios’ rules for stuttering simulations [11] with the restriction that stuttering is one-sided. The restriction is justified since one step of the specification corresponds to several steps of the implementation, but not vice versa.

Using the ACL2 theorem prover we have verified the SRAM and flash models of the preceding sections. Note that each implementation is a complex composition of a large number of state machines, which normally poses a challenge to formal verification. However, the problem is ameliorated in our case by a synergy of several factors. First, the implementation correctness is independent of the *size* of the core: we replace the core size with a symbolic constant, and use symbolic rewriting of the transition relations (an area of strength of theorem provers, in particular ACL2) rather than detailed reachability analysis. A second, subtle reason arises from the nature of the models and proof obligations. The expensive verification step involves the definition and proof of the appropriate invariant *inv*. However, this step is substantially automated by using the constraints attached to the component state machines. Since the implementation is merely an interactive, hierarchical composition, the *assumed* input constraints associated with a component *C* must be implied by the invariants (*guarantees*) associated with the state machines for their environmental components. Furthermore, in a theorem prover we can define invariants with *generic*, expressive predicates. Since ACL2 supports full first order logic, we define a predicate to express (by quantification) that each state *s* is reached by transitions in which the input sequences satisfy the associated constraints. Invariance proof for this predicate reduces to the above assume-guarantee reasoning.

²We also prove that stuttering is finite, by exhibiting a well-founded ranking function that decreases along stuttering steps; this proof is trivial since timing constraints upper-bound the completion of the state transition sequence by a natural number, namely the number of delay units in system-clock cycle.

¹Flash memories have two common organizations: NOR and NAND. A NOR flash is used as a nonvolatile memory with fast random access. A NAND flash can be used as a disc storage. We do not consider NAND flash.

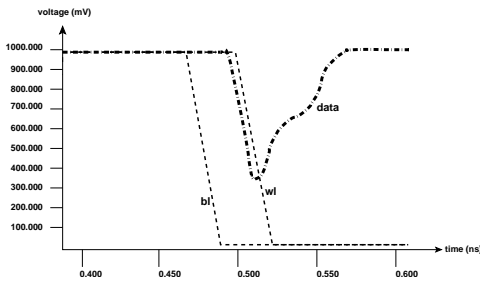


Fig. 2. A SPICE Simulation trace showing a failed write of 0 due to insufficient Setup Time

Finally, we note that one key feature of our framework is the direct behavioral correspondence between the components used for analog simulation and the formal models in our library. This facilitates corroboration of the models with readily available simulation data. Furthermore, this correspondence together with our assume-guarantee approach, can potentially identify corner cases missed in analog simulation. In one illustrative experiment, we inserted a bug in the SRAM library. During a write, the constraints on the state machines responsible for generating signals *bl* and *wl* (Fig. 1) did not guarantee that *bl* has been 0 for a sufficient time (*setup time*), before *wl* becomes 0. The bug was promptly discovered while attempting to prove the assumptions on the bitcell, and the scenario specified by the failure corresponds directly to the actual SPICE simulation pattern for the bitcell (Fig. 2).

V. RELATED WORK AND CONCLUSION

Formalization of transistor circuits has chiefly focused on developing switch-level analyzers such as SLS [12], MOSSIM-II [1], and ANAMOS [2]. Switch-level models have found extensive applications in academia and industry [2], [13]. In addition, there has been work on equivalence verification and conservative reachability analysis of small analog circuits [14], [15], [16], [17]. Finally, the PROSYD project (<http://www.prosyd.org>) aims to provide an assertion-based run-time monitoring tool supporting STL or PSL properties in analog circuits. This tool has been applied on simulation traces from a flash memory [18].

Our framework has been inspired by recent efforts of Bhadra *et al* [3] on behaviorally formalizing transistor implementations of custom memories. They show how to abstract SRAM designs using parameterized regular expressions, and compare those abstractions with a high-level memory specification using STE. However, a limitation of that work is the difficulty to correspond the abstract models with analog simulations; our approach overcomes this by carefully constructing our library of state machine models to formalize behaviors of design units that have direct correspondence with SPICE simulations.

Our key insight is that although custom memories consist of complex analog components, the behavioral characteristics of the components are well-understood, at least within the limited range of operating conditions. By focusing on the behavior rather than the structure of components, we circumvent the

complex problem of abstracting analog operations with a discrete algebra, and formalize memory implementations as interactive compositions of relatively simple state machines. To our knowledge, ours is the first platform that permits formal analysis of *both* SRAM and flash memories. Note, however, that our approach can only be applied to memory designs constructed by interconnection of well-defined components; in particular, we *cannot* abstract an arbitrary transistor netlist.

In future work, we plan to explore if the approach scales to industrial memory designs. We also plan to extend our library of models to handle multi-level flash designs.

Acknowledgements

Sandip Ray is funded in part by DARPA and NSF under Grant No. CNS-0429591. We thank our colleagues at Freescale Semiconductor Inc. for patiently answering our numerous questions on the electrical behavior of custom memories.

REFERENCES

- [1] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers*, vol. C-33, no. 2, pp. 160–177, Feb. 1984.
- [2] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proceedings of 24th Design Automation Conference*. ACM/IEEE, 1987, pp. 9–16.
- [3] J. Bhadra, A. K. Martin, and J. A. Abraham, "A Formal Framework for Verification of Embedded Custom Memories of the Motorola MPC7450 Microprocessor," *Formal Methods in Systems Design*, vol. 27, no. 1-2, pp. 67–112, 2005.
- [4] P. Agrawal, "Automatic Modeling of Switch-Level Networks Using Partial Orders," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 7, pp. 696–707, July 1990.
- [5] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
- [6] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J. S. Moore, "Functional Instantiation in First Order Logic," in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed. Academic Press, 1991, pp. 7–26.
- [7] P. Cappalletti, C. Golla, P. Olivo, and E. Zanon, Eds., *Flash Memories*. Kluwer Academic Publishers, 1999.
- [8] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1990.
- [9] D. Park, "Concurrency and Automata on Infinite Sequences," in *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, ser. LNCS, vol. 104. Springer-Verlag, 1981, pp. 167–183.
- [10] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.
- [11] P. Manolios, "Mechanical Verification of Reactive Systems," Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin, 2001.
- [12] Z. Barzilai, D. K. Beece, L. M. Hiusman, V. S. Iyengar, and G. M. Silberman, "SLS – a Fast Switch Level Simulator for Verification and Fault Coverage Analysis," in *Proceedings of 23rd Design Automation Conference*, 1986, pp. 164–170.
- [13] N. Krishnamurthy, A. K. Martin, M. S. Abadir, and J. A. Abraham, "Validating PowerPC™ Microprocessor Custom Memories," *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 61–76, 2000.
- [14] L. Hedrich and E. Barke, "A formal approach to nonlinear analog circuit verification," in *ICCAD*, 1995, pp. 123–127.
- [15] A. Salem, "Semi-formal verification of VHDL-AMS descriptions," in *Intl. Symp. on Circuits and Systems*, 2002, pp. 123–127.
- [16] A. Ghosh and R. Vemuri, "Formal Verification of Synthesized Analog Designs," in *Intl. Conf. on Computer Design*, 1999, pp. 40–45.
- [17] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda, "Verification of Analog and Mixed-signal Circuits Using Timed hybrid Petri Nets," in *Automated Technology for Verification and Analysis*, ser. LNCS, no. 3299, 2004, pp. 426–440.
- [18] D. Nickovic, O. Maler, A. Fedeli, P. Daglio, and D. Lena, "Analog Case Study, PROSYD Deliverable D3.4/2," Jan. 2007.