Abstracting and Verifying Flash 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 flash cores. Flash memories cannot be verified by traditional switch-level abstractions, due to capacitive coupling induced by the presence of floating gates. We discuss a new approach to abstracting transistor networks that is agnostic to the type of transistor used in the implementation. We show how to use this abstraction to model flash memory designs. The abstractions are used for functional verification of memory cores, and can be validated through analog simulation. We have used the approach in the verification of representative NOR and a NAND flash memory cores.**

I. Introduction

This paper is about a framework for formal functional verification of flash memory arrays. Verification of a memory array entails checking that its implementation as a network of transistors implements the high-level view of a state machine storing and retrieving data at addressed locations. Memory arrays account for more than 50% of the real estate and transistor count of a modern microprocessor. Furthermore, custom memories are complex analog artifacts with subtle and intricate behavior and aggressively optimized to satisfy performance, area, and power metrics. These two factors contribute to making memory verification a crucial component of the functional verification of a modern microprocessor or SoC design. 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.

For traditional SRAM arrays, this abstraction has been provided by a *switch-level model* of the transistor network. A switch-level model is a graph of nodes connected by transistor viewed as switches. A node has state 0, 1, or $\mathtt{X}$; a switch has state "open", "closed", or "indeterminate"; state transitions are specified by switch equations. Modern switch-level analyzers such as ANAMOS and its variants [1], [2], [3], [4] operate by partitioning a transistor network into a collection of channel-connected components, and analyzing these components to construct the requisite switch equations.

However, switch-level models cannot be used for flash memories, which contain both CMOS and *Floating Gate* (FG) transistors, where the capacitive coupling between the Floating Gate, Substrate, and Gate breaks the abstraction of a transistor as a switch. Consequently, there is a "verification gap" in current industrial practice for SoC designs containing flash components: a high-level (C/C++) description is used to model the *interface* of the flash with surrounding SoC blocks, but the underlying transistor network is not guaranteed to implement the description.

This paper bridges this gap with a new approach to abstracting transistor networks. The approach, termed *behavioral abstraction*, focuses on formalizing the behavior of analog components of the design rather than extracting switch-level models through structural analysis. This makes it agnostic to the type of transistors used in the implementation of the network. Furthermore, a key feature of the models is the direct correspondence between them and components used for analog simulation, which facilitates corroboration of models with readily available simulation data. We discuss the efficacy of the approach in the functional verification of representative NAND and NOR flash configurations.

The remainder of the paper is organized as follows. In Section II, we discuss the functional verification tool flow and explain the verification gap for flash memory arrays alluded to above. In Section III, we discuss the behavioral models and the rationale behind such abstractions. We cover some aspects of the verification in Section IV. We conclude in Section V.

II. Functional Verification of Memory Arrays

Functional verification entails the use of simulation and analysis tools to determine and expose design bugs. Since a transistor network implementing a memory is a complex analog artifact, functional verification of the array should ideally involve analog SPICE simulations. Unfortunately, SPICE simulations cannot be carried out at the level of an entire memory array. Instead, the functional verification of the array is broken down into the following two verification tasks that seek to answer the following two questions:

1) Does a *single bitcell* and its associated logic block operate according to the bitcell specification?
2) Does the *entire memory array* operate correctly inside a larger SoC design?

The first question is answered by extensive SPICE simulation (Fig. 1). Indeed, the bitcells are architected to operate over a limited sequence of *certified* stimulus patterns, each of which is validated by extensive analog SPICE simulation [5]. SPICE simulations are extensive and detailed, and cover various process corners and operating conditions. Any pattern that is not simulated with SPICE is assumed to be illegal. Unfortunately, such simulations are obviously too expensive to be carried out on designs beyond the scale of single bitcells.

The second question is answered typically by RTL or high-level simulations and analysis techniques that check for SoC-

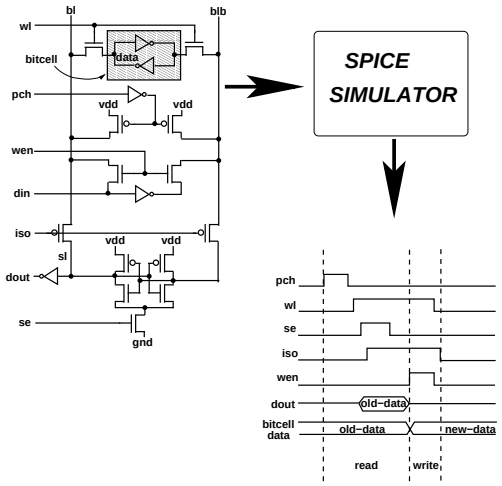

Fig. 1. Functional Verification Flow at Bitcell Level

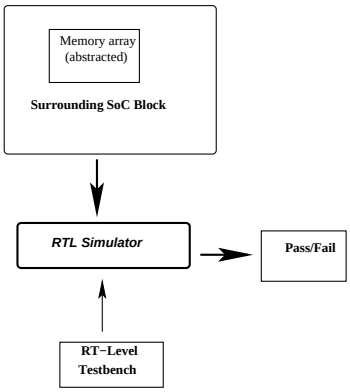

Fig. 2. Functional Verification Flow at SoC Level

level properties (Fig. 2). At this level, the memories in the design are typically abstracted to a C/C++ or RTL model that represents the interface of the memory core to the surrounding SoC block; the internals of the memory core are not exposed to the verification.

However, how do we know that the *network of bitcells* indeed implements the high-level interface used in the RTL or high-level verification? For SRAM memories this question has been answered by the switch-level abstraction of the network. However, there are two problems with switch-level models. First, it is difficult to correspond switch-level models to the SPICE models used in analog simulation; as a result, when an error is detected in switch-level analysis it is difficult to determine if the error is a real design bug or a consequence of switch-level abstraction. The second, more relevant problem from the point of view of this paper is that switch-level abstractions are broken by floating gate transistors used in flash memories. Since this problem is crucial to the arguments of this paper, we explain this problem in somewhat more detail below. For a detailed treatment of flash operations, the reader is referred to the book by Cappelletti *et al.* [6].

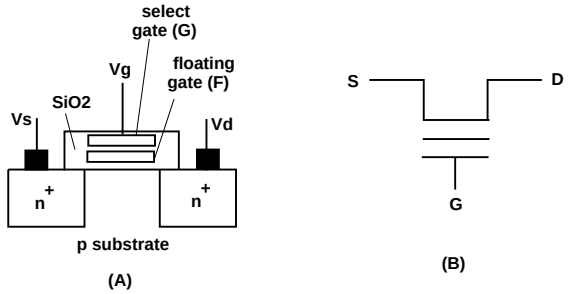

Fig. 3. (A) Structure of an FG Transistor. The polysilicon layer between the Gate G and Substrate provides the capacitive coupling. (B) Schematic representation of an FG transistor in a larger design.

FG transistors (Fig. 3) 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. 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.[1]

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.

III. Behavioral Models

How do we circumvent the above problem? Our solution entails a new approach for abstracting memory designs. Instead of extracting switch-level models by structural analysis, we model the *behavior* of the network. The viability is based on the observation that a custom memory is designed by interconnecting cohesive, logical units such as bitcells, sense amplifiers, etc. As explained in the preceding section, these units are architected to operate over a limited sequence of certified stimulus patterns, each validated by extensive analog simulation across process corners and operating conditions. It therefore makes sense to model the behavior of each unit *under operating condition* is formalized as a parameterized state machine, using guarded transitions to encode its operating constraints. The behavior of a complete memory core is then modeled as an interacting state machine composition.

[1]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.

To understand how the behavioral models work, consider the operations on a flash bitcell. Note that 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. Nevertheless, the *behaviors* of the individual components are still tractable (albeit perhaps more complex than SRAMs). Nevertheless, the *behavior* of the signal pattern corresponding to a flash operation can be viewed as discrete state transitions. 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.

We have developed a library of such behavioral models corresponding to flash components. 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 precharge (*pch*) and isolate (*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 the wordline *wl* becomes 1 in this interval, and (iii) *iso* is 1 for at least n_2 units subsequently.

Given such models for each constituent unit, how do we model the behavior of the entire memory array? The array consists of interconnection of these units according to specified configuration. For example, Fig. 4 shows the NOR flash configuration, and Fig. 5 shows a fragment of

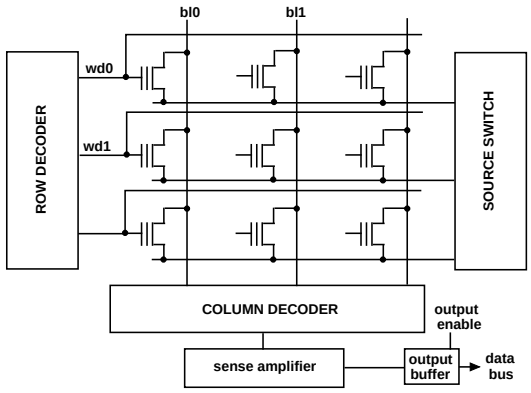

Fig. 4. Implementation of a NOR Flash Configuration with FG transistor.

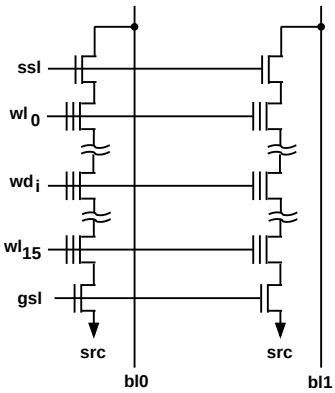

Fig. 5. Fragment of Implementation of a NAND Flash Configuration.

the NAND configuration. The configuration is produced by interconnecting individual analog units whose behaviors are formalized in our library. The behavior of the interconnection is therefore naturally modeled as an interactive composition of the individual state machines as follows. Suppose the interconnection specifies that the component $\mathcal{C}$ receives a sequence of stimuli from components $\mathcal{E}_1, \ldots \mathcal{E}_k$. Then the behavioral model of the unit is the interactive composition in which the output behaviors of the models corresponding to $\mathcal{E}_1, \ldots \mathcal{E}_k$ are composed with the input behavior of the model of $\mathcal{C}$.

IV. Verification Method

Given the above behavioral models, the verification task is to relate its execution with those of the high-level specification. The specifications are abstract state machines representing the core's interface to an SoC design. The 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. Indeed, the specifications correspond *exactly* with the C/C++ abstraction of the memory array used for provided for the memory for functional verification of the rest of the SoC design as discussed in Section II.

The final piece of the verification framework is the notion of correspondence used to relate the implementation and

the specification. The notion we use is based on *simulation refinement* [7], [8] 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 [9], 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. $\forall s, i : \mathit{inv}(s) \wedge \neg \mathit{commit}(s, i) \Rightarrow$
   $\mathit{rep}(\mathit{impl}(s, i)) = \mathit{rep}(s)$
2. $\forall s, i : \mathit{inv}(s) \wedge \mathit{commit}(s, i) \Rightarrow$
   $\mathit{rep}(\mathit{impl}(s, i)) = \mathit{spec}(\mathit{rep}(s), \mathit{pick}(s, i))$

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. These proof rules can be used to compare two systems modeled at different abstraction levels; they have been adapted from Manolios' rules for stuttering simulations [10] 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.

We have used the approach to verify parameterized models of both NOR and NAND flash configurations. Note that the models in the implementation consist of a complex composition of a large number of state machines. To ameliorate verification complexity, we make use of two techniques, namely parameterization and assume-guarantee reasoning. Note from above that the individual behavioral models are parameterized. Parameterization is done with respect to several dimensions, including (1) relative timing, (2) transistor threshold voltages, and (3) array size. This allows us to guarantee correctness of a *range* of concrete implementations in one verification run. Parameterization also facilitates focus on the design factors that are relevant to functional correctness while abstracting other details. Furthermore, the complexity of this parameterized verification problem can be effectively discharged by assume-guarantee reasoning as follows. 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, using 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.

V. Related Work and Conclusion

We are aware of *no* related effort on formal functional verification of flash memories. Formalization of transistor circuits has chiefly focused on developing switch-level analyzers such as SLS [4], MOSSIM-II [3], and ANAMOS [1]. Switch-level models have found extensive applications in academia and industry [1], [11]. In addition, there has been work on equivalence verification and conservative reachability analysis of small analog circuits [12], [13], [14], [15]. 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 [16].

We have presented a framework for formal functional verification of flash memory arrays. To our knowledge, our work provides the first platform for formal functional verification of flash designs. Since individual memory units are modeled as state machines, traditional simulation and verification tool flows can be easily adapted to handle these models. Furthermore, the specification is extracted from the flash interface with its surrounding SoC blocks; functional verification of digital components can be hierarchically composed with flash models for full SoC verification. Finally, a key feature of our framework is the direct correspondence between components used for analog simulation and behavioral models for individual units. This facilitates corroboration of models with readily available simulation data. This correspondence makes it viable to use learning techniques automate extraction of the behavioral models from simulation patterns as follows. Traces from SPICE simulation can be used to learn the parameters of the state machines for each unit through iterative refinement and the iterations can be seeded by the operating constraints used in the SPICE simulation.

The approach also opens the door for application of machine learning techniques to automatically construct behavioral abstractions. Note that the behavioral abstractions for individual units are small state machines, but their construction is delicate, involving careful characterization of different design parameters. There has been some work on applying learning techniques for estimation of trace machine models [17], [18]. We are exploring the possibility of applying such techniques for learning behavioral abstractions. The viability of applying learning techniques in our framework is based upon the fact that behavioral abstractions are state machines with close connection to models used for analog simulation; thus traces from analog simulation can be used to learn such models through iterative refinement and the iterations can be "seeded" by the operating constraints used in the analog simulation.

Acknowledgements

Sandip Ray is funded in part by the Defense Advanced Research Projects Agency and the National Science Foundation under Grant No. CNS-0429591, and by the Semiconductor Research Consortium under Grant No. 08-TJ-1849. We thank our colleagues at Freescale Semiconductor Inc. for answering our questions on the electrical behavior of flash memories.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Z. Barzilai, D. K. Beece, L. M. Hiusman, V. S. Iyegar, 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.

[5] 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.

[6] P. Cappalletti, C. Golla, P. Olivo, and E. Zanoni, Eds., *Flash Memories*. Kluwer Academic Publishers, 1999.

[7] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1990.

[8] 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.

[9] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.

[10] P. Manolios, "Mechanical Verification of Reactive Systems," Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin, 2001.

[11] N. Krishnamurthy, A. K. Martin, M. S. Abadir, and J. A. Abraham, "Validating PowerPC^{TM} Microprocessor Custom Memories," *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 61–76, 2000.

[12] L. Hedrich and E. Barke, "A formal approach to nonlinear analog circuit verification," in *ICCAD*, 1995, pp. 123–127.

[13] A. Salem, "Semi-formal verification of VHDL-AMS descriptions," in *Intl. Symp. on Circuits and Systems*, 2002, pp. 123–127.

[14] A. Ghosh and R. Vemuri, "Formal Verification of Synthesized Analog Designs," in *Intl. Conf. on Computer Design*, 1999, pp. 40–45.

[15] 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.

[16] D. Nickovic, O. Maler, A. Fedeli, P. Daglio, and D. Lena, "Analog Case Study, PROSYD Deliverable D3.4/2," Jan. 2007.

[17] A. Gupta and E. M. Clarke, "Reconsidering CEGAR: Learning Good Abstractions without Refinement," in *Proceedings of 23rd Inernational Conference on Computer Design (ICCD 2005)*. IEEE Computer Society, 2005, pp. 591–598.

[18] C. H. Wen, L. Wang, and J. Bhadra, "An Incremental Learning Framework for Estimating Signal Controllability in Unit-level Verification," in *Inernational Conference on Computer-Aided Design (ICCAD 2007)*, G. G. E. Geilen, Ed. IEEE Computer Society, 2007, pp. 250–257.