High Density STT-MRAM compiler design, validation and characterization methodology in 28nm FDSOI technology

Piyush Jain*, Akshay Kumar*, Nicolaas Van Winkelhoff*, Didier Gayraud†, Surya Gupta*, Abdelali El Amraoui†, Giorgio Palma†, Alexandra Gourio†, Laurent Vachez†, Luc Palau†, Jean-Christophe Buy†, Cyrille Dray†

*ARM Embedded Technologies Pvt. Ltd
Noida, India

†ARM France
Sophia-Antipolis, France

***Abstract*—Spin Transfer Torque Magneto-resistive Random-Access Memory (STT-MRAM) is emerging as a promising substitute for flash memories due to scaling challenges for flash in process nodes beyond 28nm. STT-MRAM's high endurance, fast speed and low power makes it suitable for wide variety of applications. An embedded MRAM (eMRAM) compiler is highly desirable to enable SoC designers to use eMRAM instances in their designs in a flexible manner. However, the development of an eMRAM compiler has added challenges of handling multi-fold higher density and maintaining analog circuits accuracy, on top of the challenges associated with conventional SRAM memory compilers. In this paper, we present a successful design methodology for a high density 128Mb eMRAM compiler in a 28nm fully depleted SOI (FDSOI) process. This compiler enables optimized eMRAM instance generation with varying capacity ranges, word-widths, and optional features like repair and error correction. eMRAM compiler design is achieved by evolving various architecture design, validations and characterization methods. A hierarchical and modular characterization methodology is presented to enable high accuracy characterization and industry-standard EDA view generation from the eMRAM compiler.**

Keywords— STT-MRAM, eMRAM, memory compiler, characterization, architecture

I. Introduction

Due to scaling challenges for flash memories beyond 28nm process nodes, new NVM solutions like STT-MRAM are emerging as promising substitute candidates . Various foundries are now reporting their readiness towards MRAM solutions [1, 2] which meet wide variety of specifications for commercial memory. STT-MRAM provides unique advantage of NVM properties with high endurance, fast speed and low power, which make it an attractive replacement for SRAM, giving a significant density advantage. An embedded MRAM is suitable for a wide variety of applications like embedded NVM in MCUs, SoCs, and last level cache memory in high end CPUs. These applications require memory capacities ranging from few Megabits to Giga bits. Flexible options for repair and error correction (ECC) are also essential, linked to memory capacity range and progressive maturity of technology [3]. This drives the need for designing an eMRAM memory compiler.

There are multiple challenges involved in designing an MRAM memory compiler for a capacity as large as 128Mb. Unlike smaller scale SRAM, it is not practical to simulate MRAM instances as large as 128Mb in order to do design validation and characterization. This is further aggravated due to presence of significant analog circuits [3] in MRAM designs requiring very high simulation accuracy. Even with fast spice simulators, simulation run time, memory footprint and accuracy levels would be far beyond practically acceptable limits. For example, a 128 Mb instance can require as much as 10-15X the memory footprint and 7-10X the runtime to simulate with an acceptable accuracy. At early stage of technology maturity, design methodologies need to be robust enough to adapt to the technology advances and enable a robust design without heavily compromising PPA. Further, design should be flexible to support sufficient tuning knobs and desired feature set. There has been a need for an eMRAM compiler, but it has not been addressed till now due to all these challenges.

The work presented in this paper describes design methodology for a high density eMRAM compiler and addresses the challenge of handling heavy simulation requirements without compromising on quality of the design validations, characterization. Rest of the paper is organized into five sections. Section II describes how memory compiler design requirements are different from a single macro design. Section III describes the eMRAM macro design techniques considering the compiler needs. Section IV describes the design validation and characterization methodologies. Section V describes the results and conclusions.

II. Requirements of commercial memory IP compiler vs single instance

A commercial memory IP compiler requirements are very different from a single macro design. It must support large number of memory configurations varying in terms of word depth, word width, capacity, aspect ratio and multiple optional features like redundancy, ECC, power gating, design for test (DFT). It should support generation of high-quality EDA views to enable smooth and reliable integration of memory instance into SoC. A careful architecture design and a tiling engine

(software code to stitch together basic building blocks to create a full instance) are required so that all possible instance configurations, including optional features can be created out of basic building blocks. In a memory compiler design self-time pulse (STP) circuit is required to achieve optimal performance across full range of compiler, through self-tracking of timing pulse w.r.t. instance size and PVT tracking [4]. For design, verification and characterization for a memory compiler, simulating every possible instance configuration will require time and compute resources beyond practical limits. To address this challenge, memory compiler is designed such that verification and characterization for limited instances at the corner of design range of compiler is enough, without loss of accuracy. Interpolation techniques are used to create the timing/power libs for middle instances. Optional features must be integrated in such a way that incremental cost of verification and characterization is not exponentially increased. These requirements make the memory compiler design different from a single instance design.

III. MRAM MACRO DESIGN CONSIDERING EMRAM COMPILER NEEDS

Core design of an MRAM macro consists of a core array which is built around 1T-1MTJ bitcells. The read operation is single-ended and based on sensing bitcell currents and comparing them to a carefully-designed reference. The write operation is performed by providing a very precise voltage to the terminals of the bitcell for a certain write time, to flip the bit. This precision is generated through a voltage regulator.

Design of a high density eMRAM compiler starts with a careful architecture choice, which can ensure that design validation and characterization can be performed at a smaller scale and overall development effort is not excessively high. A modular and hierarchical architecture is designed which enables design validations and characterization at sub-hierarchy level smaller modules, These modules are considered as validated black boxes for full hierarchical validation.

An MRAM sub-macro is one of the key modules which is a fully functional entity at its sub-hierarchy level. This sub-macro (figure 1(a)), is created with elementary blocks like core array, WL decoder, IO and control logic. All the necessary analog circuits like charge pump, voltage regulator and current references are also integrated inside this sub-macro to make it a complete entity. Each sub-macro is designed with an SRAM-like memory compiler approach so that its configuration can be chosen from 1Mb up to 8Mb capacity. A base sub-macro up to 8Mb capacity provides a good optimization among area, performance and development complexity for the complete MRAM compiler. In next upper hierarchy, up to 4 sub-macros are stitched together to form a slice, supporting max 32Mb capacity. In next hierarchy, up to 4 slices are stitched together using slice multiplexer module, supporting up to 128Mb capacity as shown in figure 1(b).

In STT-MRAM technology, write operation is much slower than read, hence making the read and write speed asymmetric. To handle slow and asymmetric read/write, we integrated a self-timing pulse (STP) scheme based on PVT compensated delay elements and partial tracking of instance size. PVT compensation in STP path is achieved by driving delay cells with a constant current source as shown in Figure 2. It helps to avoid large variation of self-time delay across PVTs and enables performance and dynamic power optimization. Write STP is longer and allowed to spread over two or more clock cycles, so that read and write can be supported at same external clock frequency, as shown in Figure 3. STP scheme is integrated inside each sub-macro which allows it to work in isolation to other modules and timing paths do not have cross-module dependencies.

Internal self-time pulse is used to latch all input signals at rising edge of clock. This achieves the objective of maintaining SRAM-like interface i.e. everything driven at rising edge of clock and no dependency on falling edge. This would help the seamless integration of eMRAM macro into the SoC using standard Static timing analysis (STA) procedures as used for SRAM. Dedicated trimming knobs are also added at sub-macro level to handle intra-die variability as used for SRAM.

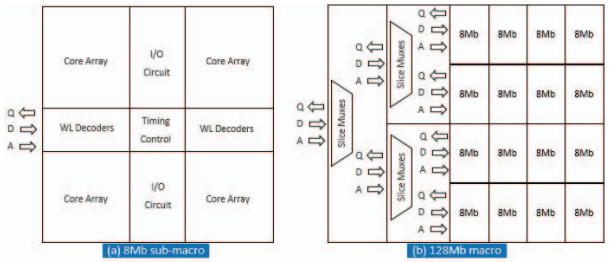

Figure 1: (a) Block Diagram of an 8Mb MRAM sub-macro, which works as an individual macro entity as well as sub-macro. (b) Block diagram of a 128Mb MRAM macro, composed of multiple sub-macros

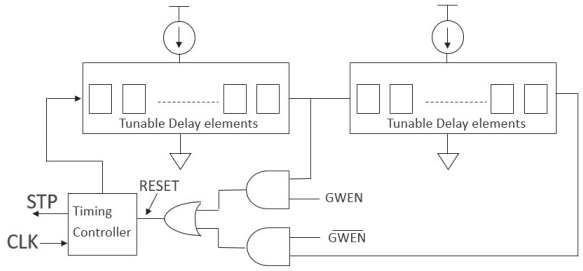

Figure 2 Self-time pulse (STP) generation scheme, designed with PVT compensated delay elements, supporting asymmetric read and write timing requirements

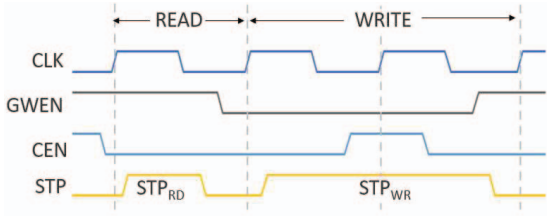

Figure 3 Timing diagrams of Read and Write Self-time pulse (STP), in which long write is distributed over two external clock pulses

IV. MRAM COMPILER VALIDATION AND CHARACTERIZATION METHODOLOGIES

Among the design validations, circuit checks, functional and race margins are checked at every module level. Margin

simulation of an 8Mb sub-macro with fully extracted tile cells and maintaining analog accuracy takes >8 days and ~300Gb memory on a multi-CPU machine with industry standard fast spice simulator. This is unaffordable owing to the iterative nature of simulation process in compiler development. To make these simulations lighter, donut approach is implemented through tiling engine, in which limited rows and columns at the boundary of core array are retained as post-layout and while rest of the core array is replaced with simplified RC models (Figure 4). With these optimizations, simulation run time could be reduced by a factor of >4x and memory footprint reduces by >7x, for a sub-macro level simulation.

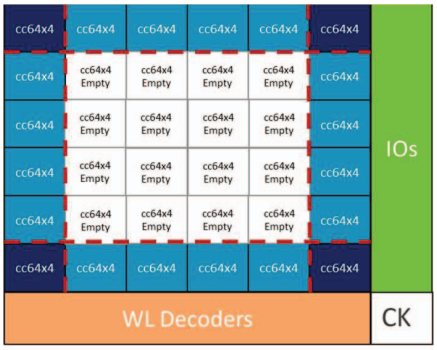

Figure 4 MRAM core array donut diagram. All devices and parasitic elements from bitcells inside donut are replaced with lumped RC model. Corner bitcells are accessed during simulations

Formal equivalence verification like Synopsys ESPCV must be performed up to highest hierarchy level to ensure that fully stitched instance is functional. For this, various black-boxing techniques were applied to make the simulations lighter. Bitcell and each analog block was replaced by an equivalent model.

Characterization of this high-density compiler cannot be done with standard SRAM techniques, in which instances at the corners of the design range are fully simulated and middle instances are either simulated or interpolated from corner instances. In the case of MRAM, boundary instances are multi-fold bigger, so it is not possible to simulate these. So, a hierarchical bottom-up characterization approach has been evolved, in which characterization is done for every unique module at the lower hierarchy levels using standard SRAM characterization techniques and then an exhaustive set of equations are written to combine the module-level characterized data to create the deliverable views for top-level instances.

For timing characterization, the MRAM sub-macro and slice-mux modules are characterized for all the conventional timing arcs as well as certain intermediary arcs, like the propagation delay of inputs and outputs through each block, and the slopes of critical signals at all interfaces. It is important to do this characterization for a range of slope and load combinations, particularly for all setup and hold timing arcs; in these arcs, any simplifying assumptions of delay arithmetic [5], cannot be applied, due to their high sensitivity to slope and load variations. So, wherever necessary, propagation delays are characterized for 7x7 slope/load tables, which is an industry standard. To calculate the timing values for bigger instances, equations are written for each timing arc, which use standard SRAM methodology based module-level characterized data as its input.

At this stage, all arcs are interpolated at their respective slope and load to achieve good accuracy. An example of equation based timing arc is shown below:

Access Time of 128 Mb instance = (Propagation Delay of CLK through Slice-mux) + (Propagation Delay of CLK through 3 MRAM sub-macros (banks) in a slice) + (CLK -> Q delay inside an MRAM sub-macro placed at the 4th bank) + (Propagation Delay of Q through 3 MRAM sub-macros (banks) in a slice) + (Propagation Delay of Q through Slice-mux)

The access time of a large macro is obtained by adding up delays at the sub-block levels, *interpolated* at the slope and load seen at each sub-block interface. For example, consider the Figure 5, shown below.

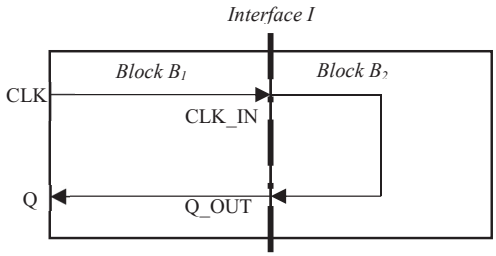

Figure 5 Block diagram representing the timing path interaction at module interface boundary

B_1 and B_2 are two modules which are characterized by running simulations. The desired outcome is the access time, i.e. delay from *CLK* to *Q*, as a function of *CLK* slope and load on *Q*, in a 7x7 table. This is developed in a hierarchical manner, by characterizing the following arcs on sub-blocks B_1 and B_2:

1. *T1* = Delay from *CLK* to *CLK_IN* through B_1
2. *T2* = Delay from *CLK_IN* to *Q_OUT* on B_2.
3. *T3* = Delay from *Q_OUT* to *Q* through B_1.

All these delays are 7x7 tables, characterized for a representative set of 7 input slopes, and 7 output loads. Because we are adding these delays together, if we simply add the matrices together, it will not be enough, because these signals will have a deterministic slope once they reach the signal interface I_1. Thus, the following arcs are also characterized:

1. *S1* = Slope of *CLK_IN* at I_1; *S2* = Slope of *Q_OUT* at I_1
2. *C1* = Input capacitance of *CLK_IN* as seen from I_1; *C2* = Input capacitance of *Q_OUT* as seen travelling from I_1 into B_1

Notice here these slopes are not 7x7 tables, but 1x7 tables. The architecture is done in a way that the input *CLK* slope will not affect the slope of *CLK_IN*, i.e. at the boundary of the sub-block B_1, and so on. The capacitances are scalar numbers. These additional arcs allow us to interpolate the previously obtained 7x7 tables and perform a very accurate computation of the access time. Thus, the following sequence of operations is done:

1. The slope of *CLK_IN* at I_1 is found by interpolating *S1* at *C1*. This yields a scalar number *s1*.

2. The slope of *Q_OUT* at *I1* is found by interpolating *S2* at *C2*. This yields a scalar number *s2*.

In subsequent sections representation $A_{B,C}$ is used for a table *A* is interpolated at its indices *B* and *C*. Armed with these arcs, the final access time of the instance is calculated as:

$$Tacc = T1_{C1} + T2_{s1,C2} + T3_{s2}$$

Similar approach can be applied to clock path and data path delay characterization across modules, which are used to compute the setup and hold time. For a multi-path synchronous architecture like MRAM, setup and hold equations must be written for all the points of convergence of data and clock, and the maximum of each of these paths is considered. Figure 6 shows the path that clock and data take through a multi-bank, multi-slice MRAM instance, and the points of convergence of data and clock. Equations are written to calculate the setup time at each of these points, labelled 1, 2, 3 and 4 in Figure 6.

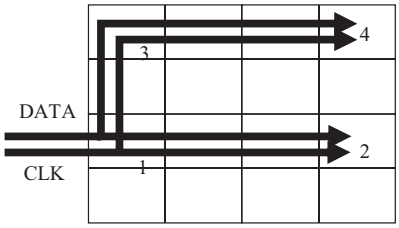

Figure 6 Block diagram of a big MRAM macro representing timing path termination points with varying delays

The point '1' corresponds to the shortest distance whereas the point '4' corresponds to the farthest distance that *DATA* and *CLK* have to travel inside the instance before they converge. Points '2' and '3' lie in the middle, and depending on the metal track configuration and surroundings, any one of the four locations may become the worst for setup and/or for hold time.

Depending on the size of the instance in terms of the number of banks and slices, the appropriate equations are chosen. Characterizing the whole compiler requires such equations to be written for 11 instance sub-types, 350 arcs of timing and dynamic power, and around 4 scenarios for every arc. This leads to nearly 15,000 individual equations being written!

To validate these equations, simulations at top level hierarchy need to be performed. For this, an STA-like approach was adopted, in which propagation delays up to the boundary of a characterized module are extracted from simulations of top-level instance. These propagation delays on both signal and clock path were added to the setup/hold timings of the base sub-macro, and were compared to the setup/hold time value that was the output of the equations. To enable these simulations, extensive black-boxing schemes were used to make the netlist lighter and the sub-macro was kept inactive as measurements are done at its boundary only. This approach could be considered reliable as each sub-macro itself was a fully characterized complete entity. A smaller set of instances and PVTs is sufficient to validate this methodology.

Dynamic power characterization was also done with a similar hierarchical approach. The power arcs were characterized at a lower hierarchy, and equations were used to build the data for the higher levels of hierarchy.

Leakage characterization could be done with the standard SRAM flow, which characterizes the leakage of every tile, and generates the instance leakage by adding up the leakage of every tile which is instantiated in the instance.

V. Results and Conclusion

We obtain very clean trends of characterized timing and power values across full compiler range. In figure 7, normalized access time degrades with increase in instance capacity, due to sub-macro size increase up to 8Mb and then clock and output Q path delay increase. Normalized read dynamic power shows increasing trend w.r.t. instance capacity, mainly due to growing contribution from long output bus toggling. Normalized write dynamic power trend remains almost flat as sub-macro level dynamic power component is the most dominant.

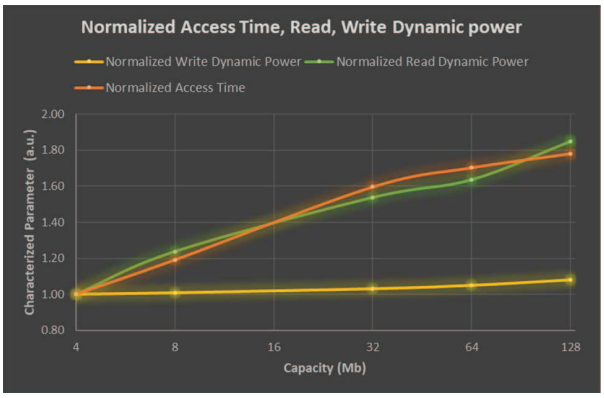

Figure 7 Normalized Access time, dynamic read and write power for various instance capacity points within eMRAM compiler range

In summary, architecture design, verification and characterization methodologies are evolved for a high density eMRAM compiler, supporting all industry standard EDA views for instance capacity up to 128Mb. Characterization accuracy within acceptable tolerance range are achieved with significant simulation run time and memory footprint reduction. Presented methodology is scalable to capacities beyond 128Mb.

References

[1] Yong Kyu Lee, et al, "Embedded STT-MRAM in 28-nm FDSOI Logic Process for Industrial MCU/IoT Application", IEEE Symposium on VLSI Technology, T17-1 pp. 181 (2018)

[2] K. Lee et al., "22-nm FD-SOI Embedded MRAM Technology for Low-Power Automotive-Grade-1 MCU Applications", IEEE Symposium on VLSI Technology, T17-2, pp. 183 (2018)

[3] Artur Antonyan, Suksoo Pyo, Hyuntaek Jung, Taejoong Song, "Embedded MRAM Macro for eFlash Replacement", IEEE International Symposium on Circuits and Systems (ISCAS), 2018

[4] Z. Wu, Z. Gao, and X. He, "Development of a deep submicrometer embedded SRAM compiler", Proceeding of IEEE International Conference on Electronics, Circuits and Systems, vol. 2, pp. 707–710, Dec. 2003.

[5] Yen-Yu Chen, Shi-Yu Huang and Yi-Chung Chang, "Rapid and Accurate Timing Modeling for SRAM Compiler", IEEE International Workshop on Memory Technology, Design and Testing, 2009