A Novel Design Methodology for Synthesizing Application-Specific Logic-in-Memory Blocks

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

H. Ekin Sumbul

B.S., Electronics Engineering, Sabanci University

Carnegie Mellon University
Pittsburgh, PA

July, 2015

Acknowledgments

First and foremost, I am grateful to my advisor, Professor Larry Pileggi, for the invaluable guidance and support he has given me throughout my years at Carnegie Mellon University (CMU). My doctoral work would not be possible without his mentorship and expertise. I would also like to thank my dissertation committee for their time and contributions to my work. I am especially grateful for the enthusiasm and advice of Professor Franz Franchetti (CMU), and thankful for all the feedback that Dr. Bruce M. Fleischer (IBM) and Dr. Ram K. Krishnamurthy (Intel) have given me.

I would like to thank my fellow group members Kaushik Vaidyanathan and Qiuling (Jolin) Zhu for their more than significant contribution to this work; David Bromberg, Renzhi Liu, and Fazle Sadi for their invaluable help; and Vehbi Calayir, Vanessa Chen, Bishnu Prasad Das, Tom Jackson, Shaolong Liu, Dan Morris, and Jinglin (Kiki) Xu for their endless support. I feel very lucky and privileged to have had the opportunity to work with them.

I am extremely grateful for the endless emotional and technical support of my friends at CMU. I would like to individually thank Berkin Akin, Onur Albayrak, Cagla (Beril) Cakir, Sila Gulgec, Melis Hazar, Meric Isgenc, Tugce Ozturk, Mert Terzi, Soner Yaldiz, Lale Yaldiz, and Sercan Yildiz for their close friendship. I really don’t think I could have made it this far without them, and will always cherish their friendship.

I would also like to express my gratitude to fellow Electrical and Computer Engineering (ECE) Department members. I’d like to thank Professor Ken Mai for all his technical and practical help, as well as his group members N. Etkin Can Akkaya, Mudit Bhargava, Burak Erbagci, and Mark

McCartney for their support. Special thanks to Da-Cheng Juan, Huan-Kai (Pumbaa) Peng, and Gillian Tay for all the good memories.

I cannot thank enough ECE administrators Judy Bandola, Michelle Echko, Samantha Goldstein, Adam Palko, and Shelley Phelps for all their time, help, and for also putting up with all my questions.

Most importantly, I would like to thank my family and close friends from Turkey for their never-ending support and love. I would not be able to go through this journey without their constant support as my anchor in life. I dedicate this work to my mom, my dad, and my sister.

I would like to also mention that this work was supported in part by the Intelligence Advanced Research Program Agency and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-12- C-2008. The work was also sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement No. HR0011-13-2-0007. I am thankful for their generous funding and support to our work.

Abstract

As the fraction of integrated circuit area dedicated to embedded memory continues to increase, the energy spent for transporting data on-chip becomes increasingly larger than the energy needed to perform computation, thereby creating system-level challenges for data-intensive application-domains. One effective approach for this challenge is to use *Logic-in-Memory* (LiM) blocks, whereby custom application-specific logic is embedded within the memory block to significantly improve the system's energy, performance, and area efficiency. Recent studies on technology scaling below the 20nm node demonstrate that extremely restrictive patterning enables the automated synthesis of LiM systems by the use of compatible logic and memory patterns. While in-memory processing architectures have been proposed and successfully built for various applications, this dissertation aims to exploit the extremely restrictive patterning to create an end-to-end design methodology for automated synthesis of application-specific LiM blocks.

The LiM synthesis methodology eliminates all the full-custom design effort that is inherently needed to build a LiM block, thereby enabling the co-design of algorithms and hardware at an affordable design cost that would be otherwise impractical. Silicon results for two data-intensive applications demonstrate that systems based on synthesized LiM designs can provide dramatic improvements of one to two orders of magnitude of energy and performance efficiency. This methodology further enables rapid design-space exploration for the overall LiM based system by making customization efficient and robust with no extra design-cost. This dissertation attempts to formulate, implement, and validate a novel design methodology that provides automated synthesis of application-specific LiM blocks.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Embedded memory occupies more than 50% of the overall chip area in a modern SoC (System on Chip) design [1] [2], and the area trends indicate that it will continue to occupy more chip real-estate as technology scales [3] [4] [5] [6]. From a performance stand-point, due to the ever-growing disparity between the speed of processor and off-chip memory, known as the memory-wall [7], embedding more memory on chip has a positive impact on the system performance by alleviating long data accesses latencies [1]. From an energy standpoint, as the IC technology is getting close to hitting the power limitations of a die [8] [9] (power-wall), accelerating a task by embedding more memory as opposed to building a higher-performance processor is the preferred choice for power efficiency since memory has a lower activity factor than logic [2]. From a modularity stand-point, IC technology already shifted towards multi-core systems to better exploit localized computation and parallelism [10] [11] [8], whereby simpler and lower performing but modular processing blocks with higher memory bandwidths can meet the ever tightening power budgets while still delivering good performance for today's data-intensive applications.

Large monolithic memories with mostly-square aspect ratios are the most area efficient choice as the periphery and lithographic area penalties are minimized [12] [13]. However such large compiled memories cause system-level issues, especially for data-intensive applications, as most of the performance and energy are spent on transferring on-chip data over long distances rather than performing useful computation. An effective solution to address this challenge is to co-optimize the algorithm, architecture, and hardware, wherein finely tailored small memory arrays integrated with application-level knowledge localize the computation and minimize the bits traveled per *mm* of distance. Such application-specific customization, however, incurs high design cost. Alternatively, compiling granular small arrays of embedded memory blocks following a conventional ASIC approach is very area inefficient. This suggests the need for a design methodology that effectively addresses the challenges of memory-intensive applications while maintaining an affordable design cost.

An enabling technology for creating such a methodology comes from scaling trends of sub-20nm technology nodes. Restrictive patterning in deeply scaled nodes constrains designers to map memory and logic to a small set of well-characterized layout patterns [14]. While memory compilers have been traditionally used to assemble hard IP layout slices of bitcells and periphery, restrictive patterning makes the patterning the only critical "hard IP". Therefore, restrictive patterning, while conventionally an impediment, does provide an opportunity to place memory cells and random logic cells in very close proximity without lithographic hotspots or area spacing penalties. Leveraging this technology constraint as a "feature," the opportunity for a *Logic-in-Memory* (LiM) methodology was proposed in [15], wherein specialized computation logic and embedded memory could be tightly integrated for localized computation and energy savings (Figure 1.1). Importantly, since the memory and logic cells are lithographically compatible, any application specific customization could be reliably synthesized into the embedded memory block with this approach.

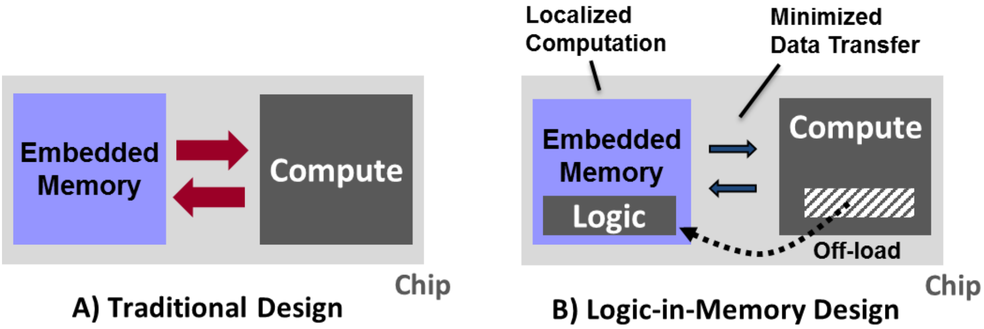
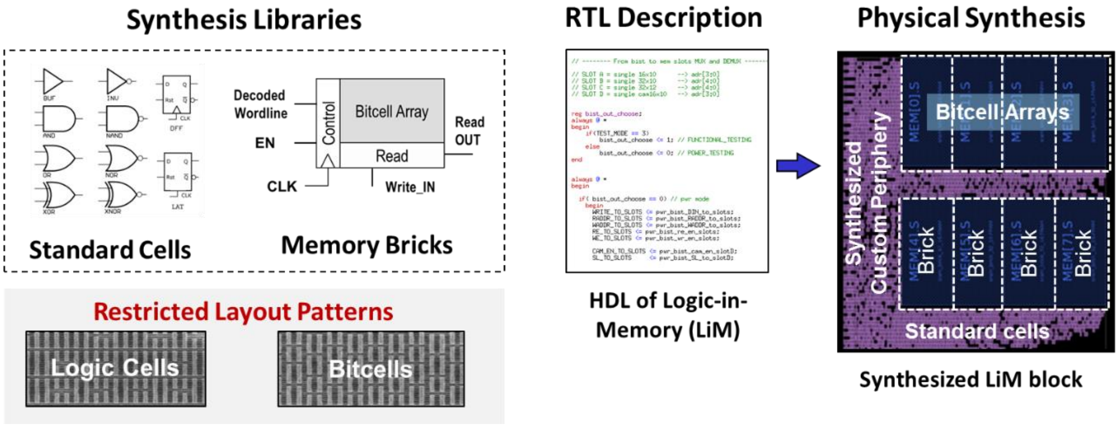


Figure 1.1: Embedded memory abstraction in traditional vs. Logic-in-Memory (LiM) design approaches

The energy and performance benefits of localizing computation for data-intensive applications are well known [10] [11] [16]. There are various examples of “processor-in-memory” designs wherein processing units are placed in memory abstraction, or more recently, near data computing in 3D and 2.5D stacks to provide more bandwidth with less energy [17]. Based on high-level models and RTL level simulations, it has also been shown that various data intensive applications highly benefit from LiM blocks [18] [19] [20]. However, there remains the need for a physical synthesis and design flow (rather than compilation) that can create LiM block designs with cost and robustness comparable to that of traditional compiled memory blocks, and that is compatible with a full chip physical synthesis.

1.2 Thesis Contribution

This dissertation attempts to formulize, implement, and validate a circuit design methodology to enable robust and affordable physical synthesis of LiM blocks. The efficiency of synthesizing a LiM block comes from combining the benefits of full-custom and ASIC design approaches, whereby efficient customization benefits are achieved through fine-grained integration of logic and memory while an affordable design cost is maintained through synthesis.



```
// -------- From bist to mem slots MUX and DEMUX -------

// SLOT A = single 16x10       --> adr[3:0]
// SLOT B = single 32x10       --> adr[4:0]
// SLOT C = single 32x12       --> adr[4:0]
// SLOT D = single cam16x10    --> adr[3:0]

reg bist_out_choose;
always @ *
begin
    if(TEST_MODE == 3)
        bist_out_choose <= 1; // FUNCTIONAL_TESTING
    else
        bist_out_choose <= 0; // POWER_TESTING
end

always @ *
begin

  if( bist_out_choose == 0) // pwr mode
    begin
      WRITE_TO_SLOTS <= pwr_bist_DIN_to_slots;
      RADDR_TO_SLOTS <= pwr_bist_RADDR_to_slots;
      WADDR_TO_SLOTS <= pwr_bist_WADDR_to_slots;
      RE_TO_SLOTS <= pwr_bist_re_en_slots;
      WE_TO_SLOTS <= pwr_bist_wr_en_slots;

      CAM_EN_TO_SLOTS <= pwr_bist_cam_en_slotD;
      SL_TO_SLOTS     <= pwr_bist_SL_to_slotD;
```

Figure 1.2: LiM synthesis flow driven by standard cell and memory brick libraries

At the core of the LiM synthesis flow, the memory arrays are comprised of "*memory bricks*" that represent the lowest level of physical abstraction, analogous to a standard cell. The non-storage cells, namely, the application-specific logic and memory peripherals, are comprised of standard logic cells that are lithography-pattern compatible with the memory bricks. Standard cell and memory brick libraries drive the physical synthesis flow together so that there is no requirement to modify the existing physical synthesis tools (Figure 1.2). Netlist, layout, and synthesis file generation of memory bricks is automated in a compiler-like fashion with a well-defined circuit formulation to eliminate any full-custom design cost. The automated brick generation further enables the use of any unconventional bitcell array sizes (non-power of two row and column sizes).

Both memory bricks and logic cells are at the same physical abstraction level for synthesis, therefore, the conventional black box memory becomes a “white box” since the barrier between memory and logic disappears. Such an approach enables memory arrays to be distributed in a fine-grained manner, thereby reducing signal travel distances and allowing the inside and outside of any memory block to be optimized across its boundary for performance, energy, and area. At the system-level, synthesis further enables rapid design-space exploration for the overall system by making customization efficient and robust. Using such a design methodology opens up opportunities to co-design algorithm and hardware, which far outweighs any loss of performance due to using automated synthesis rather than manually customized design.

The efficacy of our LiM synthesis methodology is validated in silicon for two data-intensive applications: a Synthetic Aperture Radar (SAR) image re-formatting system and a low-power accelerator for Generalized Sparse Matrix Sparse Matrix Multiplication (SpGEMM). Silicon measurement results for both designs are compared with baseline design implementations using the same bitcells that follow a conventional ASIC approach and with comparable areas. Results show that the LiM based SAR system improves energy efficiency by 37%, whereas LiM based SpGEMM design provides superior benefits up to 250x speed-up and 310x better energy efficiency, coming from the co-design of algorithm and hardware. Rapid system-level design-space exploration enabled by the LiM synthesis flow is also demonstrated on a smart stream buffer implementation for Sparse-Matrix Vector Multiplication (SpMV) application.

1.3 Outline

The remainder of the thesis is organized as follows:

Chapter 2 provides background and related work for the thesis. In-memory processing systems and their benefits, and their conventional implementation methods are first laid out. Restrictive layout patterns that enable LiM synthesis, and existing work on LiM based systems are discussed subsequently.

Chapter 3 highlights the circuit-level details of memory bricks and how to utilize them in the LiM synthesis flow. Building a memory brick for three different memory types are detailed for single-port 6-transistor (6T) bitcell, 1-read 1-write (1R1W) 8T bitcell, and content-addressable memory (CAM) based memory bricks.

Chapter 4 presents the automation tool for memory brick generation in three parts; circuit compiler, layout generator, and timing and energy estimator. An overview of the finalized LiM synthesis flow is followed by silicon verification of the methodology on a test-chip comprised of various LiM based synthesized SRAM blocks.

Chapter 5 provides silicon demonstration of LiM synthesis methodology on two data-intensive applications, namely SAR image reformatting and low-power SpGEMM accelerator core. Background, algorithm, implementation details, and silicon results are detailed for both applications, followed by an analysis of algorithm and hardware co-design.

Chapter 6 further demonstrates the rapid design-space exploration capability enabled by the LiM synthesis methodology on SpMV application as a case study. Background on SpMV and its LiM based implementation details, tool flow for generating the design-space, and system optimization by exploring the design-space are highlighted.

Chapter 7 discusses the applicability of the LiM synthesis methodology for future scaling trends and points out to possible future work directions. In the conclusion of this dissertation, the overall impact and the contributions of this work are laid out.

2 Background

In this chapter, related work on in-memory processing systems, and conventional digital IC design methods to implement these systems are highlighted first. Then the restrictive patterning trend that enables this Logic-in-Memory (LiM) synthesis paradigm is discussed. Existing work on LiM based implementations on several data-intensive application domains are also highlighted.

2.1 Application-Specific Smart-Memories

Custom smart-memories have been proposed and implemented for a varying set of applications to achieve the localized computation that leverages in-memory bandwidth for performance and saves energy by minimizing wasted data traffic [10] [16]. A smart-memory is a specialized memory block that is customized to perform extra functionalities, such that the output of a smart-memory is not just stored data, but a function of the input and the stored data. There are numerous examples of in-memory processing architectures where the computation and memory are integrated on the same chip. A brief overview of the related work and implementation methods of smart-memories are given in this section.

i) Related Work

Data-intensive architecture (DIVA) [21] [22] is a processing-in-memory (PIM) system where multiple smart-memory co-processors are put in a traditional processor to accelerate several instructions by locally executing them within the smart-memory. Intelligent RAM [16] integrates logic and embedded DRAM on a single chip for low power and high performance processing based on MIPS instruction sets. VIRAM [23] [24] is a vector processing type architecture that broadens the MIPS instruction set of IRAM targeted for multimedia applications. It utilizes the high bandwidth provided by the on-chip DRAM to perform vector processing and exploit the fine-grained data parallelism. “Active Pages” [25] is a computation model that consists of a page of data and a function set to operate on the data. By partitioning applications in between a conventional processor and a reconfigurable PIM based DRAM system, data-intensive computations are off-loaded to the PIM based DRAM while the processor can still run at its peak performance.

Computational RAM (CRAM) [26] [27] is another in-memory processing architecture wherein simple computational logic is pitch-matched and placed under memory columns, such that memory works either as a traditional memory or configured as a single instruction, multiple data (SIMD) smart-memory. Using this technique, parallel work is localized and in-memory bandwidth is used efficiently for various applications such as computer graphics, image processing, etc. Another processing-in-memory work that targets SIMD architectures is TERASYS [28], whereby high performance is achieved through an array of multiple PIMs. The PIM array is integrated with a high-performance computer, or a “host”, which is used as a SIMD processor or a as a conventional memory based on the needs of the application.

Recent developments in 3D and 2.5D IC stacks and platforms, such as the hybrid memory cube (HMC) [29], provide very high bandwidth by efficient utilization of through silicon vias (TSV). Near memory computation examples leverage this high bandwidth in between the stacked memories and the processor as another degree of localization [30]. For instance, "smart-3D" [31] architecture demonstrates how to maximize the useful data traffic in between stacked DRAM caches and the processor to minimize overall execution time.

There are also various examples that embed functionality into the SRAM to perform specific tasks for the application. An SRAM can be implemented by choosing and tailoring its bitcells and peripheries carefully for the needs of the target application, or the SRAM peripheral circuity can be further customized to perform extra functionalities. For instance, a parallel access memory (or a rectangular access memory) is demonstrated in [32] for power efficient motion estimation in high-definition video processing. Parallel-access memory is customized to access a window of *MxN* pixels of the image in a single cycle (both *M, N* are integers). This functionality can be achieved by using *M.N* number of parallel accessible banks in a conventional compiled memory. This approach, however, does not exploit address pattern commonality and leads to energy and area issues if the access window or the image size are large. A smart-memory that exploits the address pattern commonality is implemented in [32]. By using a customized and shared row decoder that simultaneously activates *M* rows to access multiple pixels and a column decoder to choose in between *N* columns, single cycle rectangular window access is achieved for less power and area. A simplified architecture of smart parallel-access memory is shown in Figure 2.1.

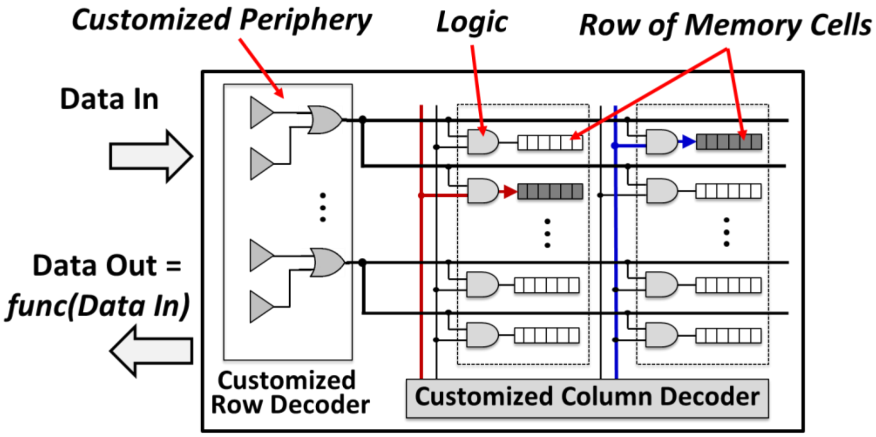


Figure 2.1. Application-specific smart-memory design example: Parallel-access memory

There are also reconfigurable smart-memory examples to meet the demands of multiple applications. "Smart-Memories" [33] is a modular reconfigurable architecture of PIM arrays, whereby processing core and embedded memory are tightly coupled to offer efficient, low power, and high performance solutions for a varying type of applications. These smart-memory arrays are reconfigured to match the needs of the chosen application by configuring their on-chip memory operation, interconnect network, and computational elements accordingly.

ii) Conventional methods to implement smart-memories

A smart-memory design can be implemented by using conventional digital IC design methods of ASIC synthesis or full-custom design approaches (Figure 2.2). For a given application that performs logical function(s) on stored data, the traditional ASIC approach is implementing these functions with synthesized logic block(s) and transferring the data from/to embedded memory. To reduce power and data movement cost, the logic can be synthesized to reside near the compiled memory. The same functionality can be integrated directly into the memory abstraction with the full-custom design approach to save further energy and gain performance. A semi-custom design approach also exist wherein non-crucial logic block(s) are synthesized, but crucial blocks of the system (such as the customized memory) are still built with full-custom design.

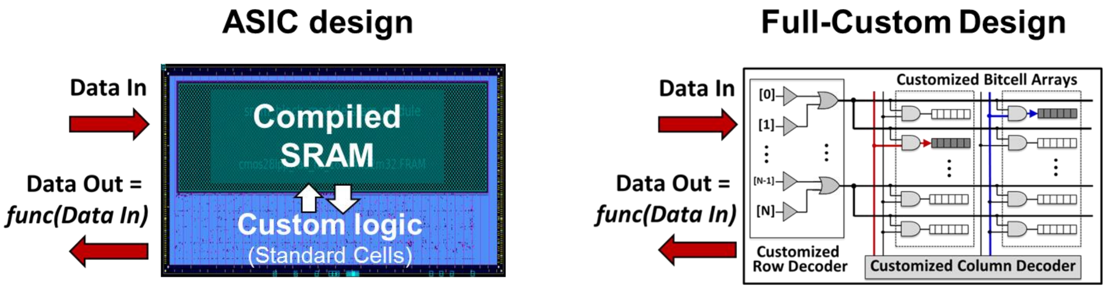


Figure 2.2. Conventional methods for implementing a smart-memory

An ASIC synthesis approach provides an affordable design-cost. In-memory customization benefits, however, are extremely limited since the compiled memory instances are provided as “black-box” or hard-IP instances from the vendor. Other than configuring the memory size, banking, partitioning, or input/output bus-widths, extra functionality cannot be integrated into the black-box memory. Full-custom design approach, on the other hand, provides efficient customization benefits as extra functionalities can be finely embedded into the memory by designing it from the transistor level. Full-custom design of smart-memories with such capabilities, however, incurs a high design cost. Transistor-level design involves intricate hand calculations, manual schematic and layout implementation, and SPICE level simulations. Any design iteration, design-space exploration, or designing alternative architectures for the same application adds on to the non-recurring design-cost. Customization benefits and non-recurring design-cost can assumed to be the same for semi-custom design approach.

The trade-off between efficient in-memory customization and design-cost suggests the need for a novel design methodology for implementing smart-memories whereby in-memory customization benefits and an affordable design-cost can be achieved together.

2.2 Restrictive Patterning Enablement

As CMOS scales below 22nm technology node with 193nm immersion lithography, design rules become increasingly complex and grow in number [14]. Morris et al. showed in [15] that a pattern construct based layout design approach can provide an efficient and affordable interface between design and manufacturing by limiting the number of patterns used to construct a block design. With such an approach, both logic and memory bitcells can be mapped on to a small set of pre-characterized layout patterns, or “pattern constructs”, thereby making them lithographically compatible. Enabled by this notion, the opportunity for efficient Logic-in-Memory patterning has been introduced where computation logic can be put inside the memory abstract without yield, area, or manufacturing penalties [34] [35]

To demonstrate this patterning influence, Vaidyanathan et al. placed area efficient and manufacturable standard cells next to embedded bitcell arrays on a 14nm IBM test chip in [36] to analyze the impact of neighborhood of random logic on the printability of bitcell arrays. Three test cases in Figure 2.3 show Scanning Electron Microscope (SEM) images for Metal-1 layers. As a reference for optimum printability, bitcell printability in the neighborhood of other bitcells is shown in Figure 2.3.A. It is observed that design rule compliant standard cells with conventional “non-regular” layout style hurt the printability of bitcells (Figure 2.3.B), whereas extremely regular standard cells next to bitcells do not impact bitcell printability (Figure 2.3.C). These results validate that restrictive patterning, which is already a necessity for deeply scaled technologies, enables tightly integrating logic and embedded memory cells without requiring extra spacing for lithographic compatibility. Design methodology for implementing the layouts of both memory and logic cells based on restricted pattern constructs is explored and demonstrated in the Ph.D. dissertation of K. Vaidyanathan in [35].

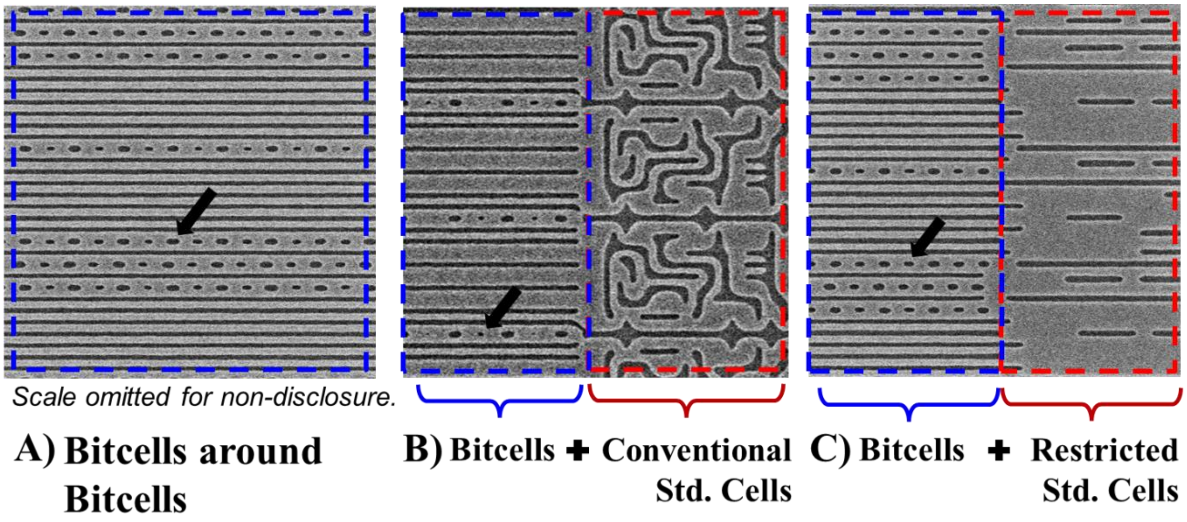

Scale omitted for non-disclosure.
A) Bitcells around Bitcells
B) Bitcells + Conventional Std. Cells
C) Bitcells + Restricted Std. Cells

Figure 2.3. Metal-1 SEMS from a 14nm IBM test-chip showing bitcell printability when random logic is put next to bitcell arrays. SEM image reprinted with permission from [36].

2.3 Application-Specific Logic-in-Memory Designs

If memory cells and standard cells can be lithographically compatible, memory periphery (customized or non-custom) can be then synthesized using compatible logic cells without compromising manufacturing robustness, and integrated with the bitcell arrays as shown in Figure 2.4. Synthesizing smart-memories would provide the best of both ASIC synthesis and full-custom design approaches. Fine-grained integration of logic and memory enables efficient customization benefits while synthesis maintains an affordable design-cost.

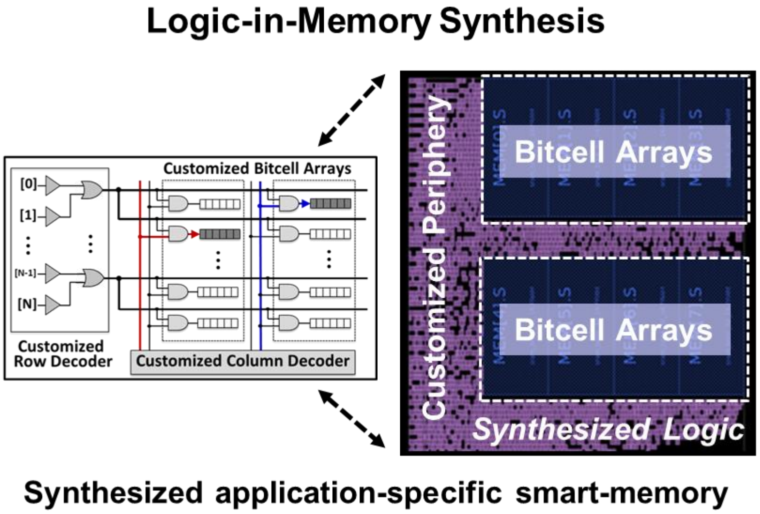

Logic-in-Memory Synthesis
Customized Bitcell Arrays
[0]
[1]
[N-1]
[N]
Customized Row Decoder
Customized Column Decoder
Customized Periphery
Bitcell Arrays
Bitcell Arrays
Synthesized Logic
Synthesized application-specific smart-memory

Figure 2.4 A synthesized application-specific smart-memory.

We have worked collaboratively with others who are working on algorithm-level exploration and conceptual system demonstrations. Application-specific LiM designs, algorithms, and their targeted applications are analyzed in great detail in the Ph.D. dissertation of Q. Zhu in [18]. Basics of a LiM synthesis framework are conceptualized in [20].

Based on the same parallel access memory proposed in [32], a smart interpolation memory is proposed in [37] [38] to accelerate the bottleneck of polar to rectangular grid conversion in Synthetic Aperture Radar (SAR) application in an energy efficient way. The proposed interpolation memory is a LiM based seed table that uses a parallel access memory as a smaller seed table and interpolates the required data on the fly as if it is readily stored. Experimental simulation results in [20] and [39] show that a LiM based architecture have great potential for improving the performance of SAR application. Further experimental simulations in [40] show that same LiM based architecture can be ported to a 3D IC stack as a low-power accelerator layer for SAR application to exploit the high bandwidth of TSVs. The interpolation memory is also proposed to be used as a performance accelerator for computing twiddle factors in fast Fourier transform (FFT) operation in [41] [42], targeted for image processing in computed tomography back-projection application.

Graph processing is another data-intensive application-domain that LiM based architectures are demonstrated to provide high gains in energy efficiency and performance. As graphs are structurally large and sparse, processing graphs effectively translates into processing sparse matrices. LiM based architectures are then ideal platforms for accelerating graph processing applications, as logic and memory are finely distributed. A smart content-addressable memory (CAM) based algorithm and architecture is proposed in [19] [43] to affordably accelerate sparse-matrix sparse matrix multiplication (SpGEMM) kernel. It is further demonstrated in [44] on silicon that synthesized LiM architecture for SpGEMM kernel provides orders of magnitude improvement in latency and energy efficiency.

Although the benefits of LiM designs have been successfully demonstrated, a formalized design methodology is still required to ensure affordable and efficient synthesis of LiM blocks. In this dissertation, we will conceptualize, implement, and verify a circuit design methodology for the automated synthesis of the application-specific LiM blocks.

3 LiM Synthesis and Memory Bricks

In this chapter, our LiM synthesis flow and its building block "memory bricks" are discussed in detail. Automation of the synthesis flow is discussed next in Chapter 4. By leveraging the restrictive patterning for deeply scaled technologies, we developed a framework to fully synthesize logic and memory using the pattern constructs as the only necessary hard-IP. By directly synthesizing application-specific functionality into the LiM block design in a fine-grained manner, smart-memories would not be custom designed at great cost, or compiled from hard IP slices at the expense of significant loss of performance and area. Moreover, by fully automating the synthesis approach, rapid design-space exploration and co-optimization of hardware and algorithm would be available for system level exploration.

3.1 Overview of LiM Synthesis Flow

A high-level overview of our LiM synthesis flow is shown in Figure 3.1. Custom periphery and logic are mapped to compatible standard cells, and bitcell arrays are mapped to "memory bricks." Then using a conventional physical synthesis flow, smart-memory blocks are described in RTL, and physically synthesized using the gate-level netlist. Commercial synthesis tools, such as Synopsys Design Compiler (DC) for logic synthesis, and Synopsys IC Compiler (ICC) or Cadence Encounter for physical synthesis are used to implement the designs. In traditional ASIC design, compiled memory blocks are “black boxes” with very limited customization allowed. In LiM synthesis flow, however, since all logic and memory arrays are represented at the same abstraction level, any memory block becomes a highly customizable “white box”. Libraries of memory bricks and standard cells are used for the LiM synthesis flow. Bricks are integrated by Verilog modules at the RTL, by library files at the gate netlist (.lib that includes timing, power, and area), and as macro blocks at physical synthesis. No modification to the existing physical synthesis tools is required by defining and utilizing the bricks as conventional macro cells.

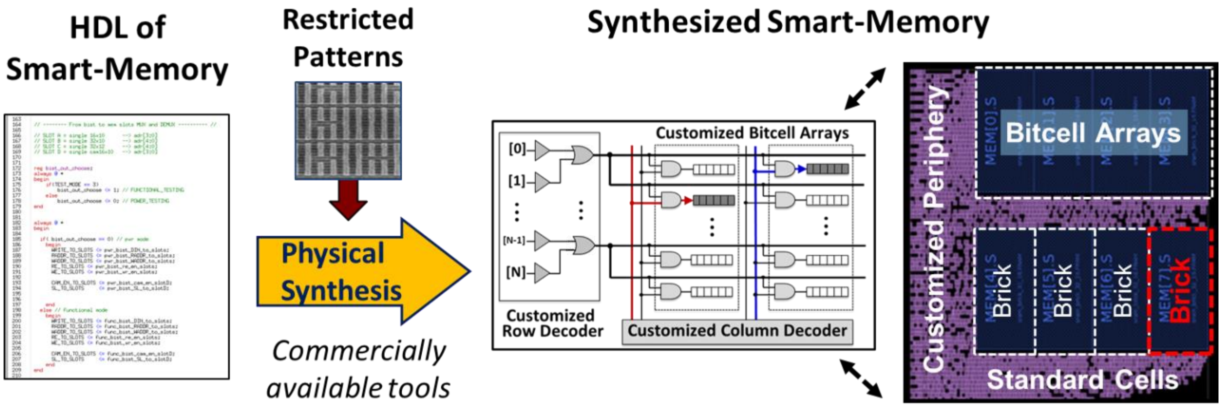


Figure 3.1. LiM synthesis flow: Smart-Memory is synthesized from its RTL description.

3.2 Memory Bricks

Memory bricks are the main building blocks of our LiM synthesis flow for building smart-memories. Similar to a standard cell that describes a Boolean logic function, a memory brick is an abstraction for the storage function. As shown in Figure 3.2, a brick is a bitcell array with simplified local periphery that enables communication to standard cells in a rail-to-rail manner, but it is not a fully functional memory slice. This simplified structure allows for integration with custom/non-custom periphery, as well as other bricks. They are designed to be stackable so that any banking configuration, memory structure (e.g. different hierarchies and/or partitioning), or memory size that is envisioned in RTL can be implemented. Wordlines (WL) and read/write operations are clocked so that the brick behaves like a sequential cell in the netlist. Decoders, write drivers, and input/output latching are not included inside the brick so that they can be synthesized with the logic to allow for smart-memory customization at the RTL level. Any type of bitcell, such as 6T, 8T, CAM (content addressable), embedded DRAM, or multi-ported bitcells can be utilized to form a brick.

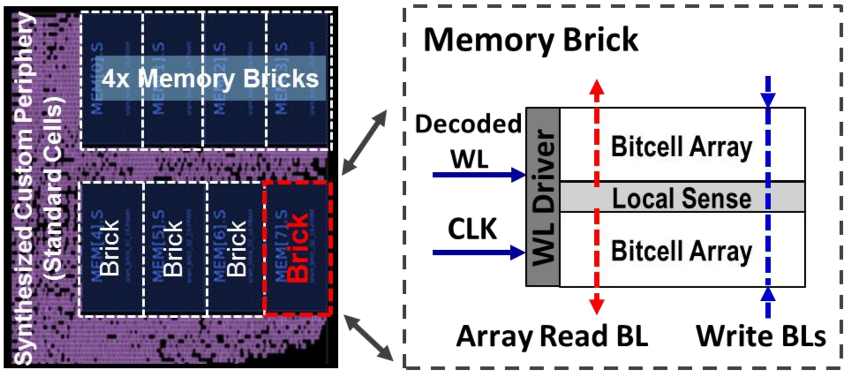


Figure 3.2. Memory brick as a bitcell array with simplified local periphery.

3.2.1 Bitcells Used in Memory Bricks

To construct a LiM synthesis flow, 6T, 8T, and CAM (content addressable) based memory bricks are implemented. Circuit topology and layout for these bitcell types are shown in Figure 3.3. Layouts of the bitcells are built in a commercial 65nm technology node. Scale and technology information is omitted for non-disclosure.

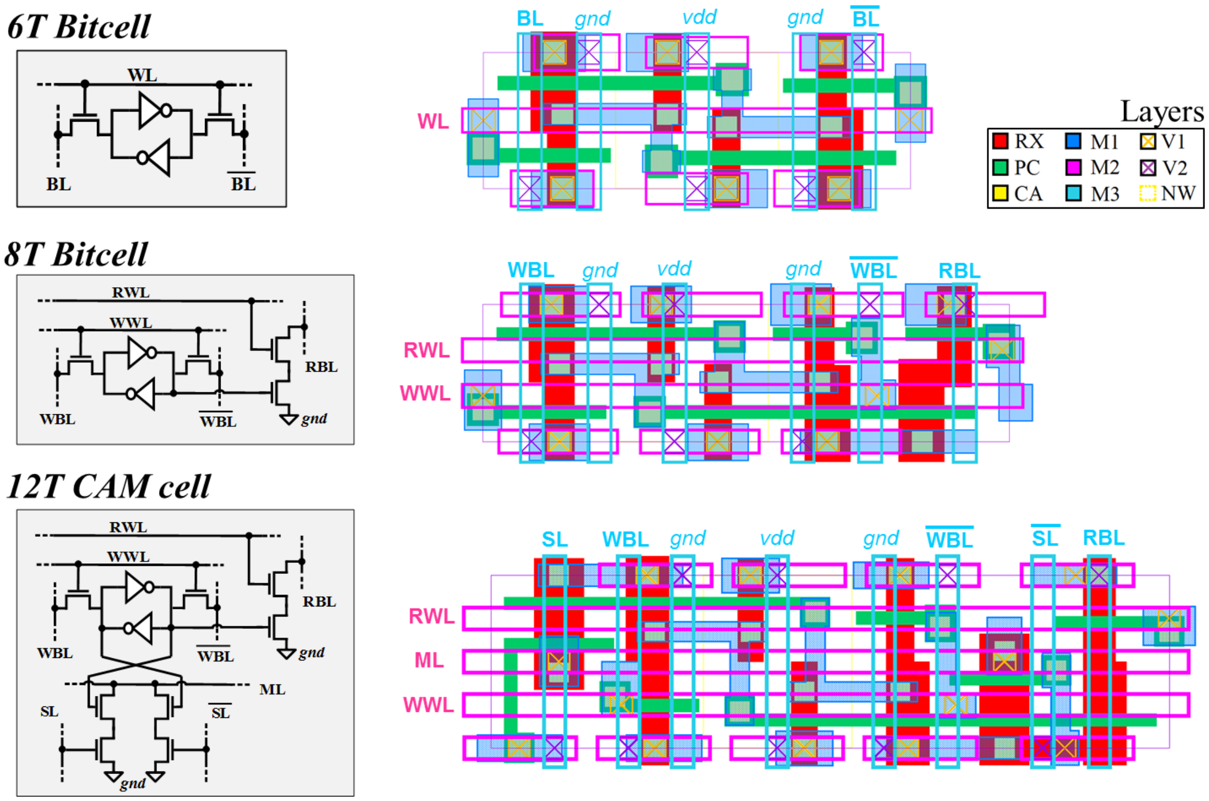


Figure 3.3. Bitcell schematics and their corresponding layouts for 6T, 8T, and 12T CAM bitcells. Scale omitted for non-disclosure.

6T bitcell is designed as a conventional single port SRAM cell with a back-to-back inverter to hold the data. NFET pass gates are activated by a word-line (WL) to form access paths to differential bitlines (BL). 6T bitcell layout is implemented as “modern type layout”, with long WL and short BLs for faster BL driving [45]. Although WLs are longer, access delay can be still optimized by adequate sizing of the WL drivers. For good readability and writability, NFET access transistors are sized to be weaker than the pull-down NFETs and to be stronger than the pull-up PFETs of the inverter-pair [46]. Read and write operations are done by driving the WL to enable an access path from BLs to the storage nodes. For a read operation, differential BLs are precharged to Vdd and left floating. As the pull-down NFET of the back-to-back inverter is sized to be stronger than the pass transistor, the storage node that holds logic-0 slowly drives the floating BL to ground [46]. For a write operation, BLs are driven differentially to logic-1 and logic-0. As the pass transistor is sized to be stronger than the pull-up PFET of the back-to-back inverter, it overpowers the storage node that is holding logic-1 and writes logic-0, flipping the values of the storage nodes in the process [46].

The 8T bitcell is designed as 1R1W capable SRAM cell by adding an extra read-stack to a 6T bitcell [47]. Stacked NFETs are used to decouple the read and write operation in the 8T bitcell [47]. Read-stack is activated by a separate read WL (RWL) to drive a single ended read BL (RBL). Write operation is performed the same way as the 6T bitcell by activating the write WL (WWL) to open a path to write BLs (WBL). As read and write are decoupled, 8T bitcell provides more robustness in static read and write noise margins compared to 6T bitcell [47]. 8T bitcell layout is built on top of the 6T bitcell by adding a horizontal RWL wire, a vertical RBL wire, and the two NFETs of the read-stack as shown in Figure 3.3.

Content addressable memory (CAM) is a memory type that can work as look-up table (or a hash-table) in addition to conventional random-access capability [48]. When an input key (or select data) is provided to the CAM, a single cycle search (or matching) operation is performed against the stored data. If there is a match, the matching address (or match line) corresponding to the select data is activated. For the CAM brick, we have built a NOR-type CAM cell [48] by adding extra CAM NOR structure to the existing 8T bitcell as shown in Figure 3.3. We have chosen the 8T bitcell as the basis of the CAM cell for single ended read and robustness in static read and write noise margins. NOR type structure is chosen for rail to rail read on match line [48], which results in a 12T CAM cell. Read and write operations are performed in the same way as the 8T bitcell. NOR NFET stacks are added to drive the match line (ML), and are gate connected to select line (SL) pair and storage nodes (Figure 3.3). ML is precharged to logic-1 before the matching operation. When the select data and stored data match, the ML is left floating, otherwise it is driven to logic-0. This way, ML is left at logic-1 only if all the stored bits in a row are exactly matching to the select data. Layout of the CAM cell is built on top of the 8T bitcell layout as shown in Figure 3.3 by adding a horizontal ML wire, vertical SL pair, and the NOR type NFET stacks.

Dimensions and areas of the bitcells are compared in Table 3.1. Note that the numbers are normalized for non-disclosure. All bitcells are designed with regular design rules of a commercial 65nm technology node (without the “pushed design rules” for SRAMs).

	Area	*Height*	*Length*	*X/Y ratio*
6T	1.00	1.00	3.15	3.15
8T	1.33	1.04	3.84	3.69
CAM	2.42	1.39	5.48	3.94

Table 3.1. Area comparison of the bitcells. Area is normalized with respect to 6T bitcell area. Height and Length are normalized with respect to 6T bitcell height. X/Y ratio is Length/Height for aspect ratio.

Logic peripheries in a memory are laid-out by matching the pitch of the bitcells for compact area, or by “pitch-matching”. Layout design, however, is a technology dependent process. Bitcells and their local peripheries can be laid out differently under various different technology constraints. It should be noted that although circuit characteristics and layout styles may vary from technology to technology, the methodology of building memory bricks and the benefits of using them as building blocks in synthesis remain the same.

3.2.2 8T Bitcell Based Memory Brick

Circuit structure of 8T bitcell based memory brick, or 8T brick, is shown in Figure 3.4, and its corresponding layout is shown in Figure 3.5. Memory architecture of the brick is chosen as a T-shaped memory topology, wherein the bitcell array is row-divided into two same size arrays that share a local read circuit. This architecture is chosen for its good performance as the length of RBLs is halved, and its compact area as read block is shared. The 8T brick structure is comprised of three local periphery blocks to control and access the bitcell arrays: WL driver, local read, and control block.

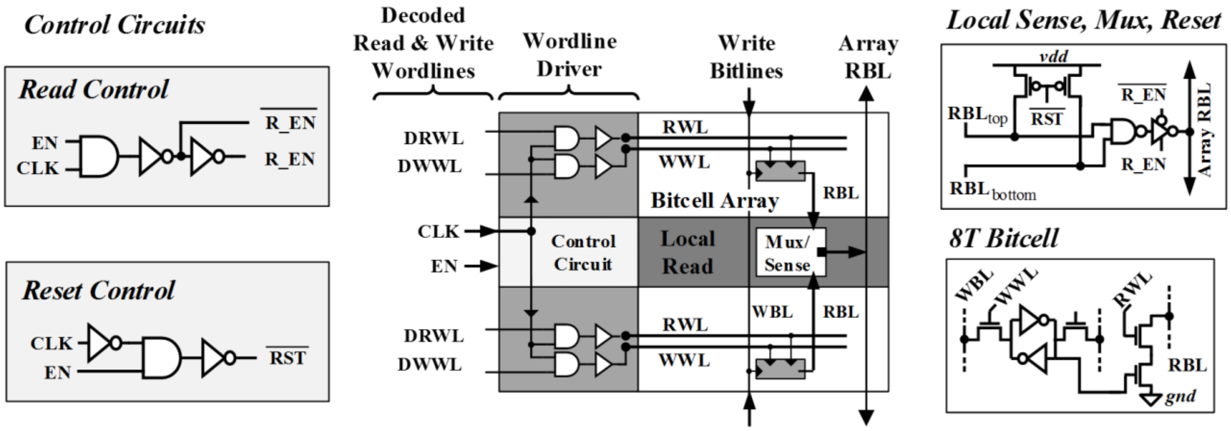


Figure 3.4. 8T bitcell based memory brick schematic

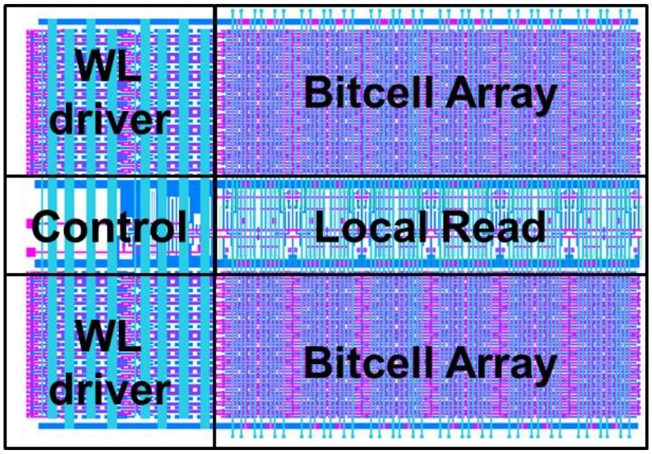


Figure 3.5. 8T bitcell based memory brick layout

As there is no decoding within the brick, WL drivers simply act as clocked buffers for the external row-decoder. They require decoded one-hot WL signal (DRWL and DWWL) as input. This structure allows peripheral operations to be handled by the synthesized logic. All WLs and read/write operations are clocked so that the memory brick behaves as a sequential block in the RTL netlist. Layout of the WL driver is pitch-matched to the bitcell row. As 8T bitcell requires both RWL and WWL, there are two separate WL drivers per row that are placed contiguously in the same row to keep the pitch-matching.

The 8T bitcell allows 1R1W access, and has a single ended RBL. Local sense is performed by a shared and skewed NAND gate that reads in two RBLs (of top and bottom arrays). As the critical edge for sensing is the falling edge, the shared NAND gate has a skew ratio of strong PFETs to weak NFETs for faster sensing. The NAND is followed by a tri-state inverter that drives global Array Read Bitlines (ARBL). ARBLs can be shared by multiple bricks, as they can be left floating by the tri-state drivers. RBL precharge is handled by PFETs which are also placed in the local sense cell. Layout of the local sense cell is pitch-matched to a single bitcell column.

A control block is also needed to produce read enable for the tri-state inverter and reset signal for the precharge PFETs. To stack multiple bricks, ARBLs have to be shared. Therefore, the brick that contains the read address actively drives the ARBLs while other bricks leave them floating. An external *“EN”* signal is used to generate a read-enable signal to activate the tri-states. Read enable generation is clocked so that the read operation is performed at clock-high. Every RBL has to be initially pre-charged, or “reset”, as the read operation in 8T bitcell is performed by pulling-down a precharged RBL. Reset signal to control RBL pre-charging is generated from EN and inverted clock, since the precharge is needed at clock-low before a read starts. Control block is fitted into the brick by matching its layout dimensions to WL driver and local sense cells.

Power and signal routing in layout is done so that bricks are stackable, thereby any type of BL is shared without any design rule violations. Our design choice for brick circuits is static logic, as it makes integration with the standard cells in the netlist easier. Write drivers, and input/output latching are also left for the outside synthesized periphery for any possible customizations.

Timing diagram of the memory brick is given in Figure 3.6. By making the brick behave as a sequential cell in the netlist, it is made compatible with other standard cells and registers in the netlist. As a design choice, read and write operations are performed at clock high, and the RBLs are precharged at clock low. For this particular configuration (rising clock edge triggered brick), the read-out data is captured by negative clock edge triggered registers in the netlist. For a clock-low active brick and positive edge triggered registers for data capturing, only requirement for the brick schematic is to invert the clock (CLK) signal for the WL driver and the control circuits.

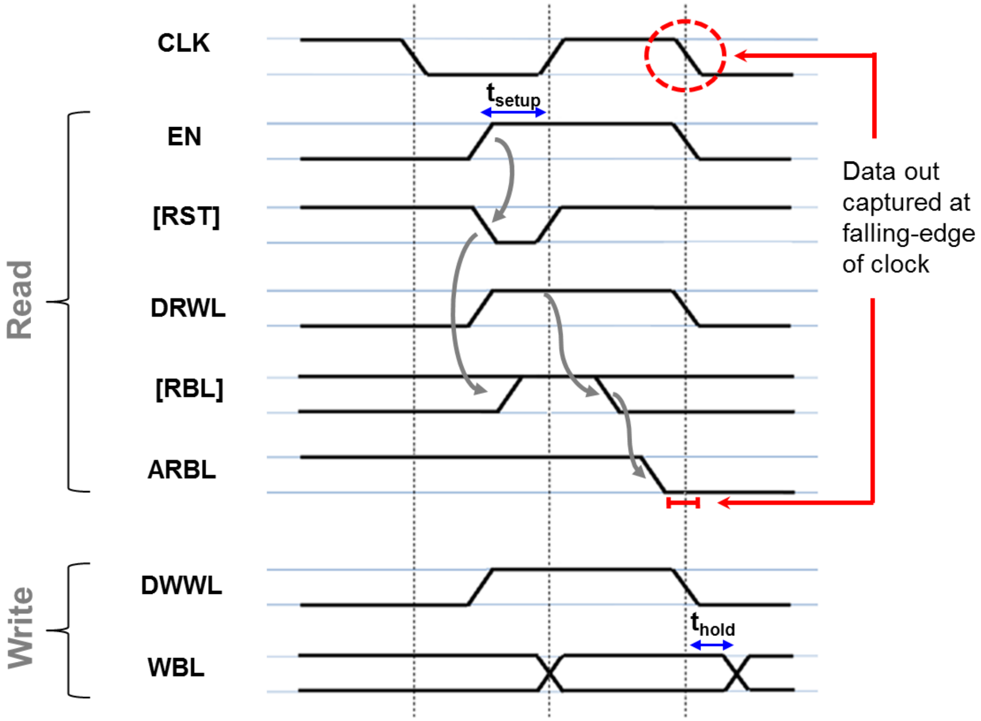


Figure 3.6. Timing diagram for 8T bitcell based memory brick

For the read operation, it is expected that one-hot DRWL and EN signal are already at logic-1 before CLK is received. When CLK goes high, chosen RWL is activated and the tri-state inverter for ARBL is enabled through EN going high. Activated bitcell row drives the RBLs, and shared local sense reads the data. Activated brick then drives the ARBL (EN at logic-1), while other bricks remain inactive (EN at logic-0). This operation expects the RBLs to be precharged before CLK goes high. For write operation, there is no need for activating the brick beforehand; however it is again expected for one-hot DWWL to be at logic-1. When CLK goes high, chosen WWL is activated and the WBLs are written into the bitcell row. As 8T bitcells are 1R1W, a read-write operation is handled the same way, however the read-out and written addresses cannot be the same address, as it would cause a logic failure. This structure makes the critical path of the brick read operation. If the brick is the slowest path in the overall netlist, then the minimum clock period for the netlist becomes:

$$T_{brick} = T_{WL\,driver} + T_{BL\,drive} + T_{ARBL\,drive} \qquad Eq.\ 3.1$$

$$T_{CLK\,min} = 2.\,(T_{brick} + t_{setup-flipflop}) \qquad Eq.\ 3.2$$

Read and write operations present several timing constraints on external signals. DRWL and DWWL are expected to be at logic-1 (or monotonously rising) when CLK goes high. Any switch on DWLs during CLK high will result in a read failure or corrupt data in write. Brick read enable signal EN has to arrive before CLK high to precharge the RBLs. When EN is received during CLK low, control block first generates a reset signal, and then reset PFETs precharge the RBLs. This deterministic delay enforces a setup time for EN signal with respect to CLK rising edge. Moreover, for a successful and error-free write, WBL pairs have to be held constant when WWL is going low to deactivate the bitcell row. As there is a deterministic delay from CLK going low to WWL driven back to logic-0, WBL pairs have to remain constant for a certain time after CLK falling edge. This dictates a hold time on WBL with respect to CLK falling. These constraints are embedded into a brick library file that is used by the synthesis tools. Brick library files are discussed in Section 3.3.

Design choices for the brick topology and its sub-blocks, timing of the operations, and decoupled read and write at array-level enable bricks to be stackable on the same brick type. Write and read operations are exclusively handled on WBL and ARBL respectively. For multiple bricks sharing WBLs and ARBLs in a stack, WBLs can be driven with static gates with no need for a periodic reset, and ARBLs are always driven by the enabled brick without a drive fight. RBLs are always local to a brick, and are precharged internally without affecting other bricks or the netlist.

3.2.3 6T Bitcell Based Memory Brick

The 6T bitcell based memory brick (6T brick) is implemented with the same T-type memory topology as 8T brick. 6T brick circuit topology and its corresponding layout are shown in Figure 3.7 and Figure 3.8 respectively. 6T bitcell is a single port cell where read and write operations are coupled and handled by the same WL and BLs. However, from a synthesis point of view, a brick has to behave as a sequential cell in the netlist, both compatible with other static standard cells and stackable on the same type of bricks. This challenge is overcome by decoupling the read and write operations at array-level (or global-level) as it is handled in the 8T brick, such that all external access operations to and from the 6T brick are static and rail-to-rail. This design choice, however, comes with an area and power overhead. All the extra circuitry to decouple the read and write operations require an intricate controlling mechanism.

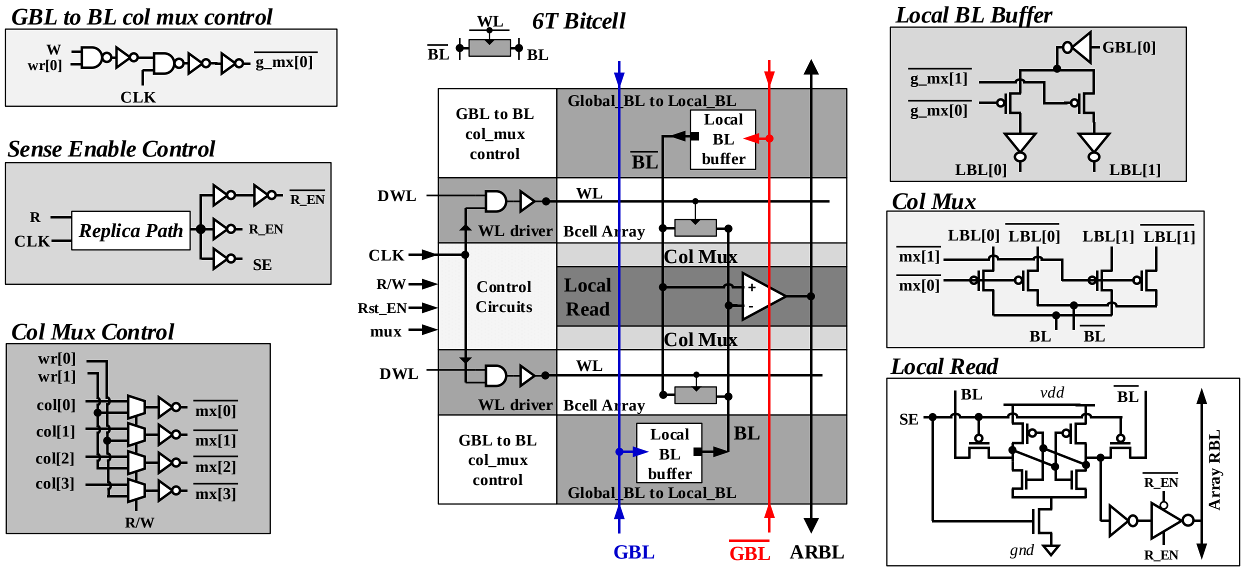


Figure 3.7. 6T bitcell based memory brick schematic. BL reset and Reset control circuits are identical to 8T bitcell based memory brick.

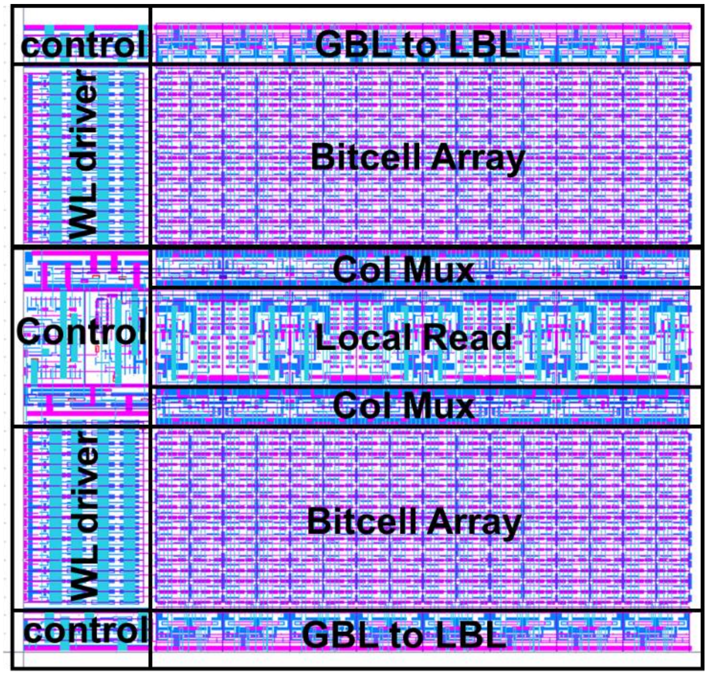


Figure 3.8. 6T bitcell based memory brick layout

Local Peripheries

WL driver schematic and working mechanism for 6T brick is exactly the same as the 8T brick; it is a clocked buffer for one-hot decoded WL (DWL), and its layout is pitch-matched to the 6T bitcell. As 6T bitcell is single port, however, only a single WL driver is needed for a bitcell row compared to two WL drivers of 8T brick for RWL and WWL.

Read operation is handled by a latch-type sense amplifier (SA) for fast performance [49]. Since the 6T bitcell has a weak driver to drive a large BL capacitance, a full-rail pull-down of BL can take a long time depending on the array size. Therefore, using a sense amplifier speeds up the read process by sensing a small voltage difference on the differential BL pair and producing a full-rail read output. As shown in Figure 3.7, latch-type SA is basically a bitcell that is controlled by a sense enable (SE) signal. At the beginning of a read operation, internal storage nodes of the SA are precharged to Vdd same as the BLs by holding SE signal at logic-0. When WL goes high, the bitcell starts driving one of the BLs, thereby creating a voltage difference at SA storage nodes. When SE signal goes high, it isolates the storage nodes and creates a virtual ground through the NFET it activates. Unequal voltages on the storage nodes creates a regenerative feedback, and resolves the voltage difference to a full rail-to-rail difference [49].

Two equally sized inverters follow the storage nodes of the SA for symmetric loading. One of these inverters then passes the full-rail read-out value to a tri-state inverter. As with the 8T brick, the tri-state inverter drives the ARBL and is controlled by a read enable signal. Local sense cell is comprised of the SA, two equal sized inverters (one of which is unused) and the tri-state inverter. Due to this large area requirement and to achieve a compact block, local sense is pitch-matched to two bitcell columns. This design choice, however, necessitates column muxing.

Since the local read block is pitch-matched to two bitcells, SA is shared by two bitcell columns. Two-to-one column muxing (col-mux 2:1) is implemented to pass the differential BLs of the chosen bitcell column to the SA. Muxes are built with only PFETs since the critical transmitted signal is Vdd or close to Vdd (Figure 3.7). Moreover, as the top and bottom bitcell arrays share the local sense in T-shaped memory topology, col-mux block is instantiated for both array sides. This means that, however, we need one more col-mux 2:1 hierarchy to choose from top and bottom arrays, which adds one more PFET delay to the already slow BL driving time delay. The extra top/bottom array muxing is pushed to the control side instead, such that the col-mux 2:1 blocks at top and bottom sides of the SA are activated exclusively. The circuit topology for the col-muxes are shown in more detail in Figure 3.9. Differential BLs have to be precharged before a read operation, similar to 8T brick. Reset PFETs are then implemented the same way as the 8T brick, but instantiated at both sides of col-muxes to precharge the local BLs. Col-muxing comes with the price of area penalty and energy spent on activating the muxes. Since two bitcells are now activated for a single bit during a read, there is also wasted energy coming from precharging un-used BLs.

As read and write are desired to be decoupled at the global-level, differential BLs have to be separated as global BLs (GBL) and local BLs (LBL) from a netlist perspective. GBL should be only used for writing data into the brick without the need for any reset, same as the WBLs of 8T brick. Read and write operations are decoupled this way, since read operation is already handled through ARBLs.

A GBL to LBL driver is implemented for the hierarchical separation of GBL and LBL pairs as shown in Figure 3.7 and Figure 3.9. GBLs are driven externally from the netlist to full-rails, whereas LBLs are internal to the 6T brick, and driven by the GBLs only for the write operation. As two bitcell columns share a single SA, GBL to LBL also needs a 2:1 muxing. Also by disabling the mux pass-gates during a read, GBL to LBL paths are disconnected at the array-level.

As the GBL to LBL buffer brings an area overhead, driving the differential LBL pairs are separated to top and bottom sides. As it can be seen in more detail in Figure 3.9, one side buffers the GBL to LBL (bottom side of the figure), while the other side buffers the GLB_bar to LBL_bar (top side of the figure). Two-to-one muxing inside the GBL to LBL buffers are built with the PFET-only structures instead of transmission gates (TG) for saving area. Design choice can be TG based muxes if Vdd versus Vth scaling becomes a problem. Control signal for GBL to LBL muxing, however, has to be paired accordingly as TG based mux require both select and inverted select for NFET-PFET pairs.

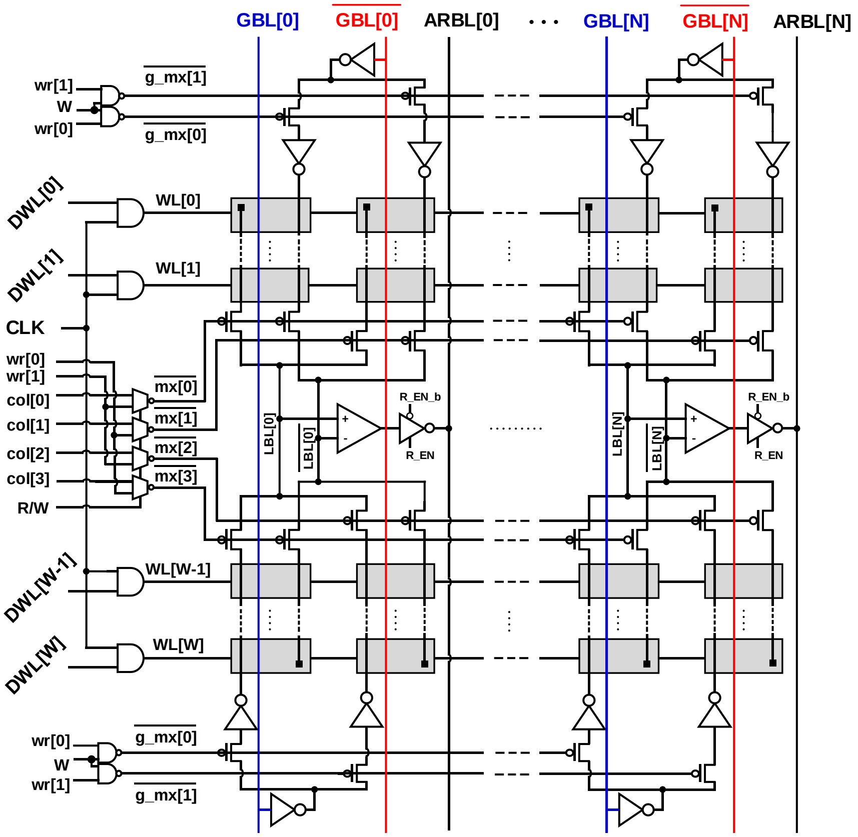

GBL[0]
$\overline{\text{GBL[0]}}$
ARBL[0]
· · ·
GBL[N]
$\overline{\text{GBL[N]}}$
ARBL[N]
wr[1]
W
wr[0]
$\overline{\text{g\_mx[1]}}$
$\overline{\text{g\_mx[0]}}$
DWL[0]
WL[0]
DWL[1]
WL[1]
CLK
wr[0]
wr[1]
col[0]
col[1]
col[2]
col[3]
R/W
$\overline{\text{mx[0]}}$
$\overline{\text{mx[1]}}$
$\overline{\text{mx[2]}}$
$\overline{\text{mx[3]}}$
LBL[0]
$\overline{\text{LBL[0]}}$
R_EN_b
R_EN
LBL[N]
$\overline{\text{LBL[N]}}$
R_EN_b
R_EN
DWL[W-1]
WL[W-1]
DWL[W]
WL[W]
wr[0]
W
wr[1]
$\overline{\text{g\_mx[0]}}$
$\overline{\text{g\_mx[1]}}$

Figure 3.9. Detailed circuit topology of 6T bitcell based memory brick under 2:1 column muxing configuration.

Control for the local peripheries

For read operation, col-mux is required to pass only a single LBL pair to the SA from a group of four possible LBL pairs; 2:1 for left/right bitcell columns and 2:1 for top/bottom bitcell arrays as shown in Figure 3.9. For a successful write, both GBL to LBL muxing and the column muxing have to work in coordination to pass GBL pair to the activated LBL pair (left/right) to form a single connection in between top and bottom bitcell arrays.

Control circuit for the col-mux signal re-direction mechanism is shown in Figure 3.9. Four separate signals are generated to control the col-muxes, namely *mx[0:3]*. During read, only a single MX control signal is activated to choose a single LBL pair from the group {top-left, top-right, bottom-left, and bottom-right} to be directed to SA inputs. During write, two MX control signals are activated together to connect the top and bottom LBLs but to choose either left or right column. MX signals are controlled by *col[0:3]* signals during read, and by *wr[0:1]* during write externally. These signals are distributed to four MX signals by using 2:1 muxes that are controlled by an external *R/W* signal. *R/W* is logic-0 for read and logic-1 for write. Generation of *col[0:3]*, *wr[0:1]*, and *R/W* are handled at the netlist for any possible memory configuration that the user may want. Distribution of these signals to the col-muxes are handled internally. At the RTL level, *col[0:3]* can be generated by a 2:4 decoder and *wr[0:1]* by a 1:2 decoder, depending on the distribution of the address space and the bits on bricks. As GLBL to LBL drive for write operation works in cooperation with col-mux blocks, it is also controlled by the same *wr[0:1]* signals activated exclusively at write. During a read or reset, connection of array-level GBL is cut with the local LBLs for decoupling the write from read at array-level.

As a design choice, control of the reset enable is left to the outside netlist. Therefore, brick read enable signal of 8T brick (EN) is replaced with a reset enable signal (RST_EN) in the reset control circuit that is shown in Figure 3.4 (control circuit identical to 8T brick). This design choice makes it possible to intelligently reset the LBLs such that a 1R1W-like access can be possible even for the 6T brick. This approach, however, requires array-level blocking of bricks and is discussed in more detail in Section 3.3.

SE signal generation for controlling the SA requires an accurate timing circuit. SE signal should arrive to the SA when bitcell drives the BL to a certain offset [50]. SA offset is chosen as 75-80% of Vdd empirically for a good enough sensing margin. To control the timing of SE signal arriving to SAs at the chosen offset, two alternatives circuits can be used. A simple and area efficient way is to build an inverter chain to match the worst-case delay of SE signal reaching the SA simultaneously the differential BL offset is achieved. This inverter chain delay, however, can drastically change under process and random variations [50]. Therefore a buffer time window has to be put to make sure SA offset is handled correctly. A replica path, on the other hand, mimics the critical path delay under any variation and matches the SA offset timing [50]. By replicating the read critical path with a WL and BL driving, the SA offset is matched under any variation. This option, however, comes with the price of extra area allocated to replica WL driver and replica bitcell column. SE signal is created using either of these options. Read enable signal for the tri-state inverter is generated by buffering the SE signal.

WL drive can be pulsed to turn-off the bitcells after the SAs are activated to reduce precharge energy. A pulsed WL period matches the timing of SA offset reaching to chosen 75% of Vdd. BLs can be then precharged starting from 75% of Vdd instead of a full-rail precharge. Pulsed WL driver is implemented by controlling the falling edge of the WL signal by inserting a series transmission gate (TG) followed by a pull-down NFET in between the AND and buffer stages of the WL driver. In this mechanism, rising edge of the WL pulse is still generated by the AND stage of the WL driver (generated by CLK and DWL high), whereas the falling edge is controlled by SE signal going high. When SE goes high, series TG is deactivated to disconnect the buffer stage from the AND stage, and the NFET is activated to pull down the input of the buffer stage, essentially pulling down the WL signal. Both the series TG and the pull-down NFET are controlled by the SE signal. An extra control block is needed for this mechanism to buffer the SE signal to WL drivers.

Two-to-one column muxing enforces two words to be stored on the same row for a brick. Under this rule, a given *Words x Bits* array is mapped to a memory brick bitcell array with *Rows = Words/2* and *Columns = Bits x2*. An example mapping under 2:1 col-muxing is shown in Figure 3.10 for an *8word x 4bits* array. The decoder that controls the DWL generation is also shown for this example mapping. Col-muxing is controlled by the least and most significant bits of the address (LSB and MSB), where LSB controls choosing left/right columns for a bit and MSB controls choosing top/bottom of the bitcell array. Other address mappings are also valid as long as *Rows x Columns* of the brick bitcell array is mapped to *(Words/2) x (Bitsx2)*.

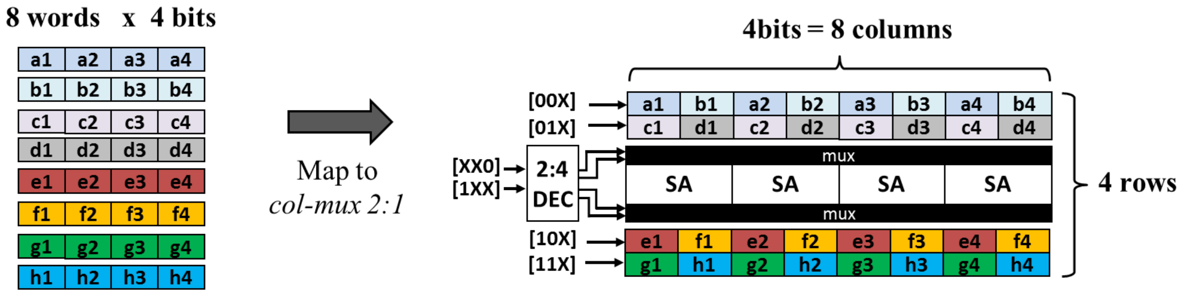


Figure 3.10. WordxBit mapped to RowsxCols in 6T based memory brick under column muxing configuration of 2:1

Design choices of the local periphery circuits and their controlling mechanisms enforce 6T brick to behave the same way as a 8T brick at array-level since the read and write operation are decoupled. As a result, the timing diagram and constraints for the 6T brick are identical to the timing provided in Figure 3.6. The minimum clock period that the brick enforces on the netlist is also the same as Eq. 3.2. Timing constraints on col-muxing controls (*col[0:3], wr[0:1]*) are identical to DWL pin with respect to clock rising edge. As DWL is used in both read and write operation, there is also a hold time on DWL with respect to clock falling edge at the end of a write operation. *Rst_EN* and *R/W* have the same setup time constraints with respect to clock rising edge with the *EN* signal of 8T brick.

As a result of the 6T brick topology, GBLs can be driven statically by standard cells for write operations, and ARBL ports drive other standard cells rail-to-rail on read operations. Multiple 6T bricks can be also stacked, as ARBL driving is handled by the brick that is activated for read while other bricks in the stack are deactivated.

3.2.4 Content Addressable Memory (CAM) Based Brick

To demonstrate the flexibility of the memory brick idea, we also built a content addressable memory (CAM) brick that behaves as a standalone sequential cell in the netlist. CAM requires extra circuitry to handle matching operations. Since the 6T brick already brings an area overhead due to read/write decoupling, CAM brick is built by expanding on the 8T brick. 8T brick is inherently capable of handling array-level operations with rail-to-rail signals at the netlist as read/write are decoupled at the bitcell level. CAM cell design is based on 8T bitcell (Section 3.2.1) so that there is no need for extra circuitry to make the CAM brick compatible with the netlist for read/write. The same T-shaped memory architecture and circuit topology of 8T brick is built by using CAM cell arrays. This way, CAM brick behaves identical to 8T brick in the netlist for read and write operations, and is capable of 1R1W random access. Extra circuitry for match operations are built around the T-shaped architecture as shown in Figure 3.11 with the corresponding layout in Figure 3.12.

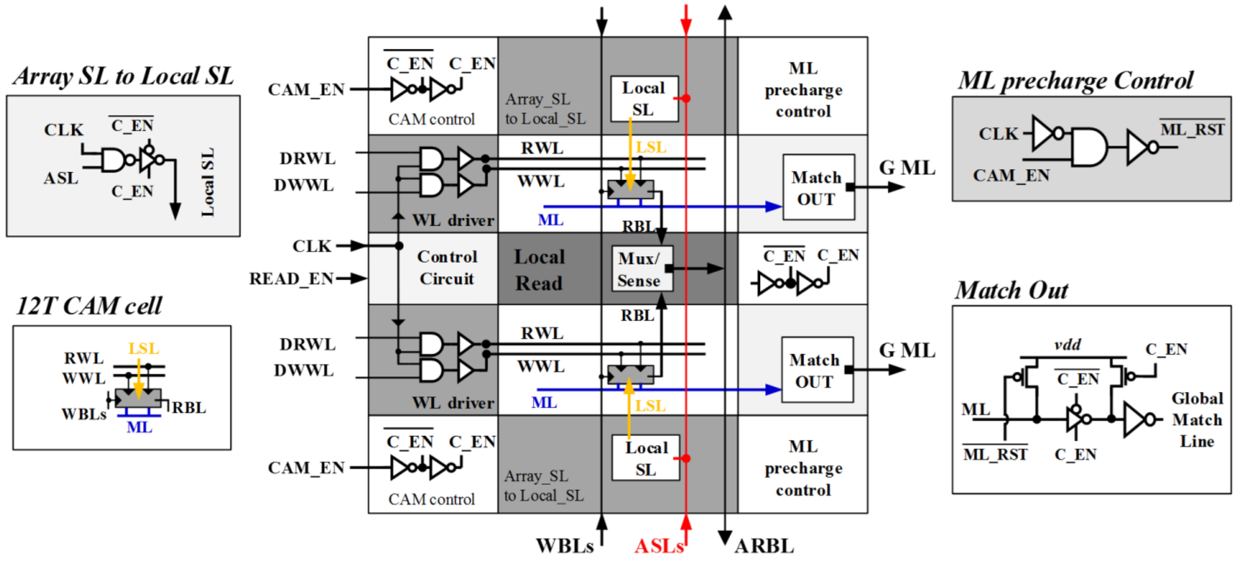


Figure 3.11. CAM brick schematic. Local read, read-enable control, and reset control are identical to 8T brick.

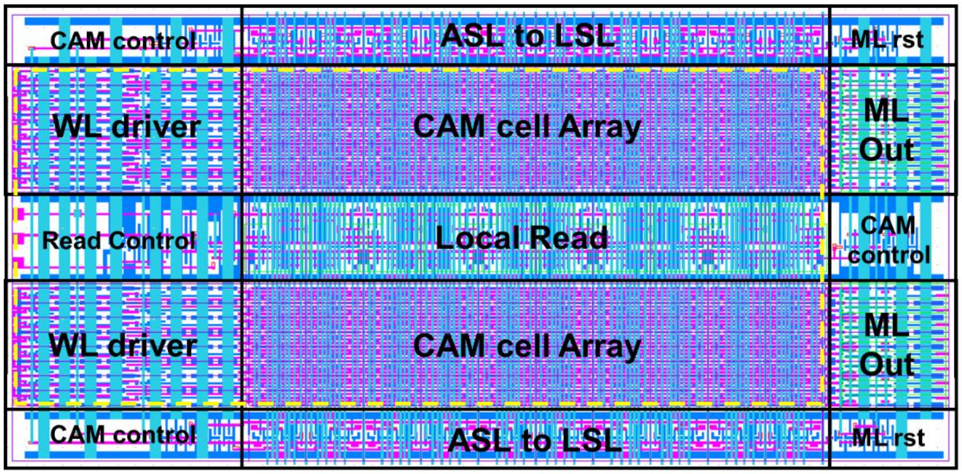


Figure 3.12. CAM brick layout based on the 8T brick architecture at its core (highlighted with yellow dotted lines).

Local periphery for read and write operations are implemented identical to 8T brick peripheries; namely, WL driver (driving RWL and WWL), local read, and read control. These blocks have the same circuit and layout styles, except for being pitch-matched to the larger CAM cell. As a result of this design choice, CAM brick circuits, layouts, external signals, control mechanisms, timing diagram, and timing constraints on the external pins are identical to that of 8T brick for read and write operations.

Single cycle matching operation in the CAM cell is discussed previously in Section 3.2.1. Match operation for the brick is done by passing the array select-lines (ASL) into the CAM cells to drive the local select-lines (LSL) differentially. Then the NOR-type NFET stacks within the CAM cells resolve the matching instantaneously without any row activation. A match-line (ML) of a row connects all the NOR-type NFET stacks, and is precharged to Vdd. Even a single-bit mismatch per row drives the ML to the ground. If all the LSLs and stored bits are matching for a row, then the ML is left floating at logic-1. All the row MLs are then passed to outside netlist through a tri-state buffer. Driving the global MLs (GML) are activated with a CAM enable signal.

Same with all other bricks types, match operation in the CAM brick has to be compatible with the standard cell based netlist, and the CAM bricks have to be stackable. Match operation, however, is fired by the select lines. This suggests the need of a global (or array level) vs. local SL mechanism whereby ASLs can be driven externally and statically without misfiring a local match operation. For this, an ASL to LSL driver is implemented for the differential SLs, as shown in Figure 3.11. ASL buffering is clocked and LSL is driven only if CAM enable signal is high. CAM enable (C_EN and inverted C_EN) are simply buffered from the external CAM enable (CAM_EN) signal as shown in Figure 3.11. To save area, driving LSL from ASL is separated into two for LSL and inverted LSL, and placed on top and bottom arrays as shown in Figure 3.11 and Figure 3.12. When there is no match operation, connection between ASL and LSL is cut by disabling the tri-state inverter driving the LSL through CAM_EN going low. This mechanism imposes a hold-time on ASLs with respect to clock falling edge, in an analogous way to WBL hold time.

Match operation expects the MLs to be precharged to logic-1. A reset PFET is placed into the ML to GML driver to precharge the MLs. Reset PFETs are controlled by ML precharge control circuit which activates the reset when a CAM_EN is received during CLK-low. When the match operation is done, ML to GML drivers drive the MLs that are at logic-0 or logic-1 to the outside world. GMLs are captured by negative edge flip flops with or without an encoder first. When matching operation is done, a weak PFET pull-up in the ML out cell is activated by C_EN going low to drive all GMLs to logic-0 to make sure that there are no floating signals at the netlist. A weak keeper mechanism can also replace this design choice. The overall CAM operation imposes both a setup time and a hold time on the CAM_EN signal. With respect to clock rising edge, there is a setup time on CAM_EN such that every MLs are precharged before the clock arrives. With respect to clock falling edge, a hold time is needed such that every negative edge triggered flip flops connected to GMLs capture their corresponding GML correctly before the outputs are all pulled down.

Depending on the array size, CAM brick critical path can be either read operation for a large number of words or the matching operation for a large number of bits. Both delay information is put in the library file for the synthesis. Critical path for the CAM brick is then:

$$T_{CAM\ brick} = \max\{\ (T_{WL\ driver} + T_{BL\ drive} + T_{ARBL\ drive}),\ (T_{LSL\ driver} + T_{ML\ drive} + T_{GML\ drive})\ \} \qquad Eq.\ 3.3$$

$$T_{CLK\ min} = 2.\left(T_{CAM\ brick} + t_{setup-flipflop}\right) \qquad Eq.\ 3.4$$

3.2.5 Analysis of Bricks

Critical path delay, energy per operation, and total area of the bricks are cross-analyzed in this section. Same bitcell array size of 16x10bits is used for all the bricks for a better comparison. Load for each brick is assumed to be stack of 8x bricks of the same type. Since 6T brick has different memory size (*WordxBits*) mapping to bitcell array size (*RowsxCols*) due to col-mux 2:1, two instances of 6T bricks are added to the comparison. Memory sizes of 16x10bits and 32x5bits are mapped to bitcell array sizes of 8x20bits and 16x10bits respectively for the compared 6T bricks. The following tables summarize SPICE simulations for post-layout bricks.

	Memory size	*Bitcell Array size*	*Number of Sense Cells*	*Read Critical Path*	*Total Area [um x um]*	*Length [um]*	*Height [um]*
8T brick	16x10b	16x10b	10	307ps	664.2	36.1	18.4
CAM brick	16x10b	16x10b	10	330ps	1254.2	51.4	24.4
6T brick	16x10b	8x20b	10	331ps	879.5	44.3	19.8
6T brick	32x5b	16x10b	5	362ps	636.7	26.1	24.4

Table 3.2. Critical path and area of 16x10bit bricks

Delay [ps]	*Self-loading*	*1xWL cap*	*2xWL caps*	*4xWL caps*	*5xWL caps*	*6xWL caps*	*8xWL caps*
Match critical path	388.8	402.7	413.2	430.3	438	445.8	461.1

Table 3.3. Match delay of 16x10bit CAM brick under different loads

Energy [fJ]	*8T based 16x10bit*	*CAM based 16x10bit*	*6T based 16x10bit*	*6T based 32x5bit*
Read opp	920	1091	524	382
Write opp	149	177	356	236
Match opp	-	2463	-	-

Table 3.4. Energy per operation (opp) of 16x10bit bricks

For the read delay, 8T brick is the fastest and 6T brick is the slowest as expected. CAM brick read delay is also relatively close to the 8T brick, as the CAM cells are built based on the 8T cells. Performance penalty on CAM read is paid due to extra capacitance coming from larger area. For the 6T brick with 8rows x 20bits, read delay is relatively close to 8T brick, as BL heights are approximately the half of 8T brick RBLs. However, when we look at the comparable bitcell array of 6T with 16rows x 10bits, 6T brick is slower. WL drivers and final tri-state inverters are sized optimally for minimum delay, so the delay coming from WL driving and final load driving is roughly the same for all the bricks. Sizing of the brick circuits is explained in detail in Chapter 4. CAM match delay with no loads on GML (self-loading) is still slower than CAM read delay with 8x stacked bricks. This does not necessarily mean, however, that critical path of CAM is always the match operation, as a CAM array with a large number of rows (or long RBLs) and short bit-length (or short ML) may result in CAM read delay being comparable or worse than the CAM match delay. Table 3.3 shows the CAM match operation delay with changing final load capacitances. Load capacitance unit is set as WL capacitance.

6T brick area with comparable bitcell array size of 16rows x 10cols has the smallest area, whereas CAM brick has the largest area as expected. But it should be noted that 8T brick area and 6T brick area with comparable bitcell array size are very close. This is mainly coming from the extra area overhead of the local periphery that is introduced in 6T brick for global read/write decoupling. Also, the 6T brick with 16words x 10bits (bitcell array size 8rows x 20cols) has a larger area compared to 8T brick, although 8T bitcell is larger than 6T bitcell. This is mainly due to the aspect ratio that col-mux 2:1 enforces. As the 6T brick with same memory size has two times the columns of 8T brick, length of the 6T brick becomes nearly 25% longer than the length of 8T brick. It also has nearly the same height with 8T brick due to extra area overhead at top and bottom arrays.

Energy-per-operation is compared for read and write operations for one full clock cycle, such that both operations include the energy dissipated on precharging BLs. When energy results are analyzed for read operation, 6T brick is the low-power choice as expected, and CAM brick is the least power efficient. In between two 6T bricks, the one with 32x5bit memory size consumes less energy as it is comprised of half the number of local sense cells, precharge circuits, and columns per activated row when compared to the one with 16x10bit memory size. During the write operation, local BLs are driven internally and global BLs are driven externally by the netlist for the 6T bricks. Write BLs are all global in 8T and CAM bricks, so they are all driven externally by the netlist. As a result, write energy for 6T bricks are higher than the 8T and CAM bricks. The energy for driving the write BLs for 8T and CAM bricks, however, will be still dissipated by the standard cells that will drive the WBLs. For 8T and CAM bricks, 8T brick has lower write energy as expected with its lower array capacitances. CAM match operation consumes high energy, as all the MLs are first precharged and then driven to the ground except for the matching row(s) due to the chosen NOR-type CAM cell structure.

Brick layouts are technology, PDK, and bitcell dependent. All layouts are implemented with conventional technology rules, and "pushed design rules" are not used. Preferred directions for M2 and M3 are horizontal and vertical respectively. All layouts are done such that signals are distributed in up to M3 only, with the exception of read-enable control signals going up to M4 (horizontal) in the 6T brick due to wire congestion. Both rail and grid type power distribution layout styles are used up to M3 wherever possible. Clock distribution in the bricks is implemented as wire-only with no internal buffering. Clock signal is only used at the initial stage of WL driver and control circuits, so keeping the first gates at minimum size for these blocks help with minimizing the loading on the clock network of the netlist.

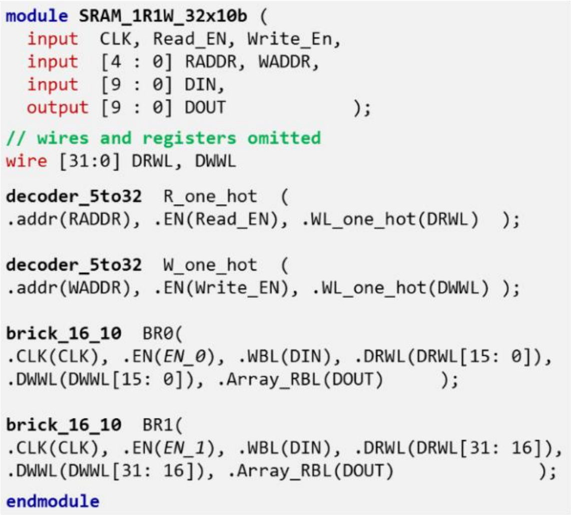
3.3 Memory Bricks in Synthesis

Enabled by their design, memory bricks are physically and functionally compatible with standard cells and other bricks. By further defining bricks in hardware-description languages (HDL) as a module, they are integrated with RTL designs and behavioral simulations. To use the brick modules in a netlist at physical synthesis, their circuit level timing, power, area details and pin capacitance information are exported into a library file, which is then used in the synthesis flows as a macro cell. Flexible brick stacking allows building any memory configurations at the RTL.

3.3.1 Verilog for Synthesized SRAM

A simplified HDL example on how the bricks are used in the netlist and RTL is given in this section. A simplified Verilog code that is given in Figure 3.13 implements a 32x10bit 1R1W SRAM using an 8T brick with 16x10bit array size (*brick_16_10*). Array size of 32x10bits is first created by instantiating two of 16x10bit bricks and stacking them by connecting their input

WBLs and output ARBLs. As bricks do not have a decoder, a 5 to 32 decoder generating 32 one-hot DWLs is built with standard cells (*decoder_5to32*). Since the SRAM is desired to be 1R1W, the same decoder is instantiated twice for handling read and write addresses separately. Enable signal for the bricks can be generated from the most significant bit of the address to activate one of the bricks while the other stays idle to preserve energy during a read operation. Depending on the brick size or the memory brick type (8T, CAM, 6T), the module that is instantiated is changed to the appropriate Verilog module. A Verilog pseudo-code of the used memory brick in this simplified example (1R1W 8T brick with 16x10bit size) is given in Appendix A.



```
module SRAM_1R1W_32x10b (
  input  CLK, Read_EN, Write_En,
  input  [4 : 0] RADDR, WADDR,
  input  [9 : 0] DIN,
  output [9 : 0] DOUT              );
// wires and registers omitted
wire [31:0] DRWL, DWWL
decoder_5to32  R_one_hot  (
.addr(RADDR), .EN(Read_EN), .WL_one_hot(DRWL)  );

decoder_5to32  W_one_hot  (
.addr(WADDR), .EN(Write_EN), .WL_one_hot(DWWL) );

brick_16_10  BR0(
.CLK(CLK), .EN(EN_0), .WBL(DIN), .DRWL(DRWL[15: 0]),
.DWWL(DWWL[15: 0]), .Array_RBL(DOUT)     );

brick_16_10  BR1(
.CLK(CLK), .EN(EN_1), .WBL(DIN), .DRWL(DRWL[31: 16]),
.DWWL(DWWL[31: 16]), .Array_RBL(DOUT)              );
endmodule
```

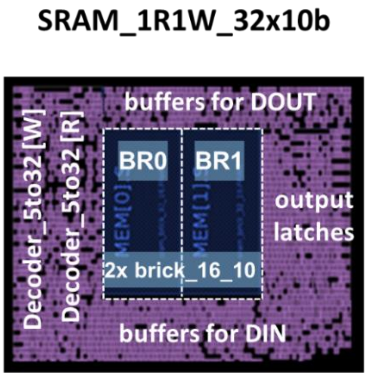


Figure 3.13. Simplified Verilog code describing a 32x10bit 1R1W SRAM using two stacked 16x10bit memory bricks.

3.3.2 Memory Bricks and Partitioning

Circuit and layout structure of memory bricks enable their flexible use within a design, making it possible to build any memory configuration and hierarchy at the RTL. As bricks are stackable and distributable, local and global peripheries defined at RTL can configure and utilize memory brick partitions to form different memory hierarchies. Partitioning example for building an

SRAM with three different configurations are shown in Figure 3.14. An SRAM with a given memory size can be built as a single partition by stacking N bricks, or with multiple partitions by stacking N/2 bricks or N/4 bricks. Partitions can share the decoder for the same memory size, or can have separate decoders to implement parallel accessible banks.

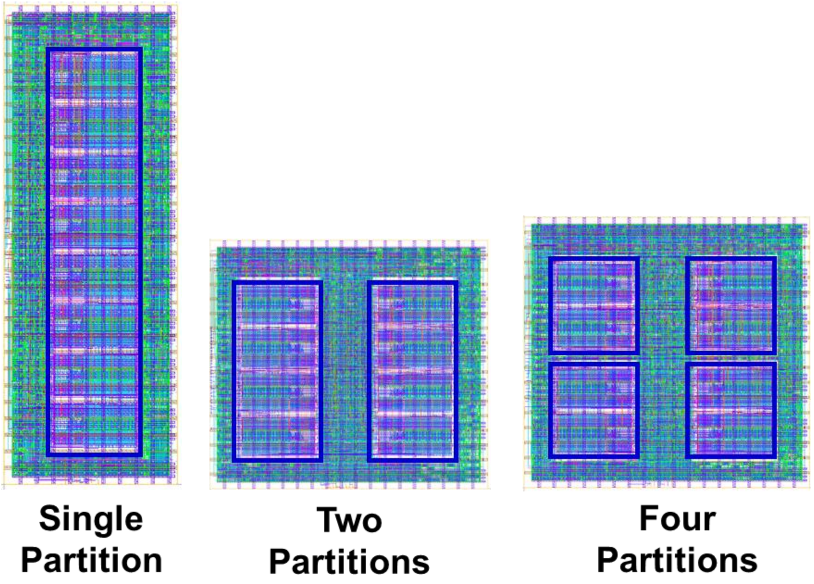


Figure 3.14. Partitioning examples for a synthesized SRAM

More complex hierarchies can be built by configuring the global accesses to WLs and BLs. By introducing different levels of hierarchies to row/column decoders and banking formations at the RTL level, various different memory configurations can be implemented; such as divided WL structures [51] [52], hierarchical word decoding [53], or hierarchical divided BLs [54] [55]. As bricks permit their activation and deactivation through their control pins (read enable, reset enable, or CAM enable), global memory structures can be controlled by generating the appropriate control signals from the available address space.

An interesting capability for the 6T brick is an "area-windowing" mechanism, as described in

[56], whereby the reset-enable and read-enable pins are used to control internal read, write, and precharge operations. A 3R2W application-specific register file is implemented in [56] by using four 1R1W banks with shared global BLs. By mapping read and write operations to available address spaces, or "area-windows", 3R2W capability is achieved without modifying the bitcells. The same concept can be used for the 6T bricks by carefully timing the read, write, and precharge operations. After a read operation that is performed at clock high, BLs in the 6T brick are precharged at clock low. Since reset-enable of a brick is controlled externally, however, we can leave the BLs discharged if we know that we will perform write immediately after the read. Then a write operation can be done at clock low instead, making the 6T brick behave as a 1R1W capable brick for that cycle. For the next cycle, however, the same brick should be blocked for the address space if a consecutive read operation is scheduled, and another brick in the partition / stack should be used. In the consecutive cycle, BLs of the blocked brick are then precharged by activating reset-enable signal, making the brick available at the beginning of the third cycle and so forth. With this control and address mapping at the RTL, a 6T brick can be used as a "modify after read" (or read-before-write) [57] memory where the decoded WL is kept the same to immediately modify the read-out address, or as a 1R1W memory by implementing two decoders that work at exclusively at clock high and clock low.

Different partitioning choices lead to different area, power, and delay numbers for the same SRAM size. The choice of memory brick array size further changes the overall performance. For instance the same single partition can be built by a single brick, or by multiple bricks stacked on top of each other. Although bitcell array size stays exactly the same at the RTL level, changing brick sizes impact the delay, energy, and area as. All of these partitioning and memory configurations span to a design-space, and the exploration of this LiM based design space is analyzed in detail in Chapter 7.

3.3.3 Library of Memory Bricks

A library file of a cell guides the synthesis tool regarding how the cell behaves under different conditions. It includes gate-level characteristics and technology dependent timing models of the cell under different input driver and output loading scenarios [58].

The timing of a cell is characterized with respect to varying output loads and input signal slews. These numbers are then tabulated in a characterization look-up table (LUT). When a cell is placed in the netlist, its timing behavior is modeled by performing interpolation on its characterization LUTs with respect to the input slew and output load the gate sees. Modeling involves cell delay and output slew, which in turn, becomes the input slew of the next gate(s) placed on the same net. Hold and setup times with respect to a signal edge are also modeled and used in the same way. Power is modeled for different input-output combinations for multiple operations based on a unit clock period. Then, depending on the netlist clock period and pin switching activity, power of a cell is calculated by appropriately scaling the power numbers reported in the library. Pin capacitances are also reported in the library to calculate loading of the previous gates in the path driving the cell. Area information is used in logic-synthesis to estimate the overall area of a synthesized netlist.

Memory brick library files for synthesis are created in Liberty format (.lib) [58], and they include the following information:

- Timing and capacitance indexes for the characterization LUTs
- Critical path delay

- Output rise and fall times
- Hold times with respect to a signal
- Setup times with respect to a signal
- Pin capacitances (extracted wire and total gate capacitance)
- Area
- Power for read, write, no-operation (no-op), leakage, match, mismatch, and any possible multi-port operations such as read & write, match & read, etc.

Typical-typical (TT), slow-slow (SS), and fast-fast (FF) corners (for both NFETs and PFETs) for each of these characterization points are needed in three separate library files. Library files in .lib format are human readable, and can be generated by a tool, or via a text editor by modifying an existing .lib file.

LUT that holds the timing models of the bricks are stored in a 7x7 matrix format, where rows and columns are indexed by the input index key. LUT columns denote output loads and rows denote input slew. The appropriate index key has to be provided in the brick library file. Typical loading scenario for a brick is other bricks in a partition plus the final gate load. Then a typical load capacitance index key for a brick partition with 8 stacked bricks is *{0x, 1x, 4x, 8x, 9x, 12x, 16x}* of total ARBL pin capacitance, where *0x* denotes no load case (self-loading) and *16x* denotes twice the intended load of *8x* bricks. Note that the typical load that the brick will see in the netlist is placed into the middle of the index. In the same way, a row index is also created for typical input/output slews in terms of “fan-out of four” (FO4) delays [59]. Rise and fall times of the output pin are also stored in the same way. Post-layout RC extracted (RCX) SPICE simulations are performed on the bricks by sweeping the output load and input slews to simulate the timing delays. SPICE simulations for generating all these numbers have to be repeated in TT, FF, and SS corners. Note that performing all these simulations are very time-consuming. Alternatively, an automated delay estimation to generate the timing characterization LUTs is discussed in Chapter 4.

Critical path delay is modeled as CLK received (CLK rising) to output delay. As the critical path is read operation for 8T and 6T bricks, delay is modeled as mid-to-mid delay of CLK to ARBL delay. For CAM brick, there is additional delay of match operation that is defined as mid-to-mid delay of ASL to GML. Brick delays and output rise/fall slews are modeled by sweeping possible output loads that the brick can see in a netlist. Since a brick is a large macro block with more than eight stages of gates on its read delay path, the effect of input CLK slew on the ARBL rise/fall time and CLK to ARBL delay is typically negligible. As discussed in the previous section, each brick has its own hold and setup time constraints on certain input signals with respect to CLK. Hold and setup time characterizations are performed by pushing the signal towards the intended CLK edge and checking if the operation fails. FF simulations for hold time and SS simulations for setup time are critical.

Area information is directly imported from the layout, and pin capacitances are calculated from RCX wire capacitances plus the total gate capacitance on the net. Power for brick operations are simulated in SPICE at a typical clock period. Power operations are defined by logical checks on input pin data, such that the tool can understand which power number to use for a given operation. For instance, in an 8T brick, if any of DRWL pins are at logic-1 and all the DWWLs are at logic-0, then the operation is read and read power should be used. On the other hand if none of the DRWL or DWWL is activated (all at logic-0), then the operation is a no-op.

3.3.4 Synthesized SRAM vs. Compiled SRAM

Using the Verilog modules described Figure 3.13, the partitioning examples given in Figure 3.14, and the brick library files described in the previous section, we synthesized several brick based 1R1W capable SRAMs. RTLs are created and verified in Mentor Modelsim, gate-level netlist is synthesized in Synopsys DC, and final layout (in gds format) is generated in Synopsys ICC. A commercial standard cell library of low-Vt gates (LVT) in 65nm technology is used. Chosen memory size is 128x10bits, and it is synthesized by using 32x10bit, 64x10bit, and 128x10bit bricks with possible partitioning configurations of single, two, and four partitions. Bricks are stacked whenever needed and six different SRAM configurations are synthesized. Used brick type is 8T bitcell based 1R1W capable brick.

To analyze the circuit-level penalty introduced by synthesis, two different versions of the same SRAM size are designed as a compiled memory alternative, namely SRAM_v1 and SRAM_v2. Bitcell arrays are distributed to four 32x10bit arrays that share local sense in SRAM_v1. Only one array is activated to read out the 10bit word. Bitcell arrays are distributed to four 64x5bit arrays that share the local sense in SRAM_v2. Two arrays are activated simultaneously to read out the 10bit word. SRAM is implemented by using the 8T bitcell, and pitch-matched sense and WL driver (for row-decoder) instances of the brick. Pre-decoder is designed by modifying the WL driver of the brick. In/out flip-flops for read and write addresses, and data-in/out are used from the same LVT standard cell library.

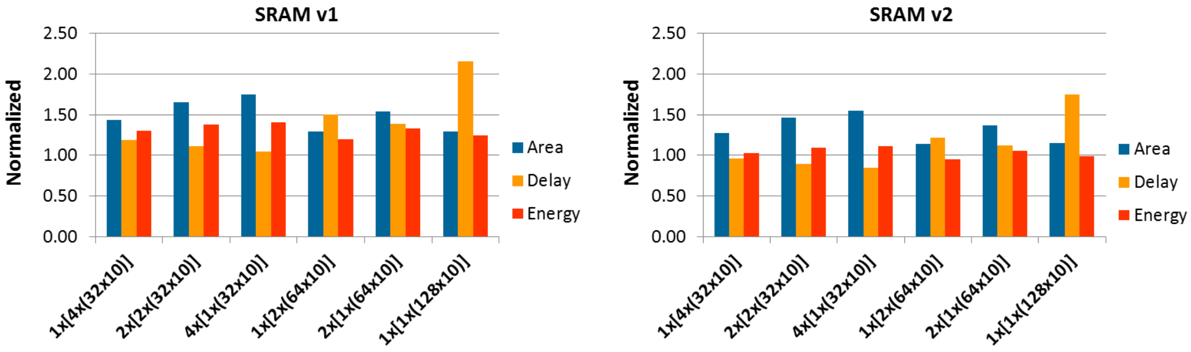


Figure 3.15. Synthesized LiM based SRAM comparisons to compiled-type SRAMs.

Comparison results for synthesized SRAM configurations and compiled type SRAM are shown in Figure 3.15 for area, frequency, and read energy. Results are aggregated from SPICE simulations and block-level estimations. All results are normalized for SRAM_v1 and SRAM_v2 numbers for an easier discussion. Results point out that the comparison of a compiled SRAM and synthesized SRAM based on LiM flow depend on many interacting factors, and a clear-cut apples-to-apples quantitative analysis is not feasible. Bitcell design, memory formation, architecture of the compiled SRAM, partition sizes and aspect ratios have a visible impact on the results. Furthermore, synthesis parameters such as flat vs. hierarchical synthesis, increased wire congestion for smaller area, brick sizes, partitions formed with bricks (stacking bricks), use of global or hierarchical peripheries vs. single monolithic array architecture have equally important impact on the comparison results. Note that this comparison is simply done on conventional SRAM without any application-specific customizations. Any smart-memory type customization would make the overall quantification of the comparison even harder.

When the results are compared, we can see that the area penalty coming from synthesized global peripheries ranges from 20-50% for matching bitcell array sizes. Overall delay results have a bigger margin, showing that synthesized SRAM can be faster or slower depending on how it is configured. As bricks have their own local sense cells, any stacked brick partition immediately becomes a hierarchical access with fast read due to shorter BLs. Energy comparison is the hardest to quantify as it heavily depends on the stored data and switching of address bits. Read energy, however, roughly matches the area trends.

A qualitative analysis of this comparison is more meaningful in this case. Design choice of including the WL driver into the brick results in a roughly 10% area penalty. WL driver is embedded into the row-decoder in a compiled SRAM, whereas decoder is re-instantiated in the netlist for the synthesized SRAM, resulting in an additional penalty of 10-20% depending on how big the area allocated to standard cells is. When more bricks are stacked, they each include their local sense and control circuits, analogous to local vs. global read in the hierarchy. This increases the speed, but comes with a price in area and energy. Critical path of the brick sets the clock-high period as given in Eq. 3.2, whereas the critical path of the compiled SRAM is pre-decoder, row decoder, bitcell read, and sense-out delay. This total delay is hidden under the whole clock cycle and not only at clock-high, resulting in a faster frequency. This does not, however, mean that more work is done with a compiled SRAM, as the other half of the clock cycle is used for any application-specific computation in the synthesized SRAM. Energy difference depends on total switched capacitance at the decoder side, as the bitcell and read energies are more or less the same for equal bitcell array sizes. As a result, it follows the area trends for equal bitcell array sizes. Changing the partitioning and brick sizes, however, has an immediate impact on the energy.

It is also meaningful to analyze the impact of using different brick and stacking on the memory performance. Results show that bricks with larger bitcell array sizes are more area efficient compared to the equal memory size built from stacking multiple bricks. Smaller bricks are, however, faster as they have shorter BLs. WL driver delay for a constant bit number per row roughly gives the same delay, as sizing, load capacitance, and gate stages are the same. For the same bitcell array size, energy burned for precharging BLs and read-out are roughly the same. More bits per row, however, results in more power burned for the control of the reset PFETs for the same array size. Single partitions result in area and energy efficiency, but this is traded off with a penalty in speed due to longer wire delays.

3.4 Design-cost of Implementing Bricks

Without a design methodology, design cost of building smart-memories is pushed back to brick generation. To obtain full customization benefits and reduced design cost, a complete LiM framework requires a well formulated end-to-end design methodology. For this purpose, we automated the memory brick generation.

If done manually, non-recurring custom design cost of an application-specific smart memory still exists since the memory bricks have to be generated both in circuit and layout level, and simulated accurately to get timing and power numbers to be used in physical synthesis. Memory bricks need to be compatible with any given application and novel algorithms; therefore any type of memory brick should be readily available. Custom designing bricks every time for the needs of a new application defeats the purpose of eliminating the design cost, and recreates a relatively high cost. In addition, this cost increases drastically if an effective design-space exploration is to be performed, since all memory types and sizes may be needed. For a true design-space exploration, LiM synthesis framework requires having a virtually infinite memory library including any combination of any word numbers, bit lengths, bank sizes, etc.

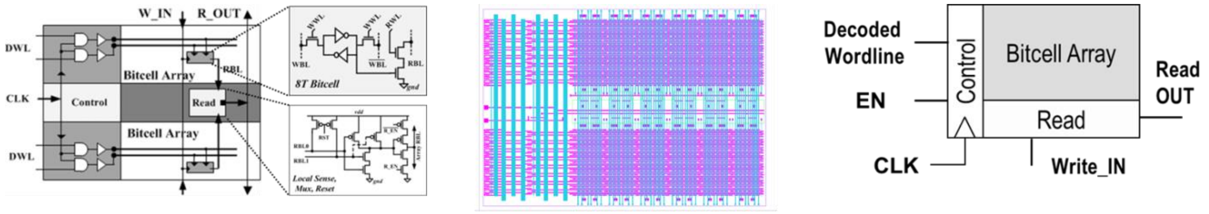
A formalized design methodology is implemented to ensure that LiM synthesis framework can be used as an automated synthesis flow that can facilitate co-design of algorithm and hardware, and design-space exploration. Automated memory brick generation methodology is discussed in the next chapter.

4 Dynamic Brick Library Generation

An automated memory brick generation flow is formulized and laid out in this chapter under three main parts; netlist generator, layout generator, and synthesis library estimator. The automation flow provides parameterized memory bricks for any memory type, thereby allowing creation of a dynamically sized, virtually infinite brick library for the synthesis. Bricks are sized optimally for minimum delay, specifically for the intended memory configuration that is provided by the user as an input. Accuracy of timing and energy estimations are also verified on a test-chip by comparing simulation results based on estimated brick library and silicon results for various synthesized SRAM blocks. Such flow further allows a rapid design-space exploration of the LiM block targeted for the application. Dynamic generation of the brick library at the backend of the LiM synthesis flow now enables the co-design of the algorithm and hardware.

4.1 Automated Brick Generation

Our automated brick generation tool consists of three main parts, namely netlist, layout, and library generations (Figure 4.1). To enable instantaneous generation of the necessary synthesis files for a given brick, brick netlists are first generated by a circuit compiler. The netlist is then passed to a layout generator for automated generation of its corresponding layout. Finally, the transistor sizing and layout area information are passed to performance estimation tool to generate the corresponding library files for synthesis.



Netlist Generator **Layout Generator** **Synthesis Library Generator** (Performance & Energy Estimation)

Figure 4.1. Automated brick generation comprised of netlist generator, layout generator, and performance estimator (synthesis library generator).

4.1.1 Memory Brick Netlist Generator

Circuit structure and working mechanisms for bricks are pre-defined for any memory type (Chapter 3). Taking the memory type, array size (words x bits), and number of bricks to be stacked in a bank as user input parameters, a netlist of a brick is automatically generated by translating the desired user configurations into a gate sizing problem. To optimally size the peripheral blocks within the brick, we have developed a formulized circuit design methodology based on logical effort calculations and RC delay estimations.

Logical Effort [59] [60] is a circuit methodology that is used for optimally sizing a stage of gates, wherein each gate is driving the consecutive gate. Fundamental expressions and equations for logical effort calculations are given in Eq. 4.1. Then going through logical effort calculations, we arrive at the following equation set:

$$LE = \frac{C_g}{C_{g_{inv}}} \quad ; \quad FO = \frac{C_{out}}{C_g} \quad ; \quad SE = LE.FO \quad ; \quad PE = \prod_{i=1}^{N} SE_i \quad with\ N = number\ of\ stages \qquad Eq.\ 4.1$$

$$For\ minimum\ delay \quad SE_{i\pm1} = SE_i = SE \qquad Eq.\ 4.2$$

$$PE = \prod_{i=1}^{N} SE_i \xRightarrow{SE\,=\,SE_i} PE = (SE)^N = \prod_{i=1}^{N} SE_i = \prod_{i=1}^{N} (LE_i.FO_i) \qquad Eq.\ 4.3$$

$$Since\ \ FO_i = \frac{C_{i+1}}{C_i} \qquad then \quad (SE)^N = \prod_{i=1}^{N} (LE_i.FO_i) = \left(\prod_{i=1}^{N} LE_i\right).\frac{C_{load}}{C_{in}} \qquad Eq.\ 4.4$$

$$\Rightarrow\ SE^N = \left(\prod_{i=1}^{N} LE_i\right).\frac{C_{load}}{C_{in}} \qquad Eq.\ 4.5$$

Given that the technology dependent β ratio is a known parameter, logical effort *LE* of any gate (LE_i) is known. So from Eq. 4.5, for a known number of stages *N* and gates types (LE_i), output load C_{load} of the final stage, and input capacitance C_{in} of the first stage, we can calculate the stage effort *SE* that gives the minimum delay for all the stages. Based on Eq. 4.5, we then conclude that every gate in a memory brick can be sized optimally for minimum delay considerations, given that the input and final load capacitances are set.

The formulation we use in Eq. 4.5 is applicable for any technology node or physical design kit (PDK). As sizing transistors, however, is technology dependent, several technology dependent parameters have to be initially characterized before going into circuit generation. These parameters are β ratio for equal PFET and NFET delays, FO4 delay (fanout of four delay), and gate capacitance (Cg) per gate width (Wg) for transistors with minimum gate length in form of fF/uM. We further characterized charging time of a given capacitance with respect to PFET gate size (Wp) by finding a fitting function in the form of $t_{ps} = \alpha\left(R_p\right)^K (C)^M$. These parameters are characterized in SPICE simulations using the chosen PDK.

The formulation driven in Eq. 4.1 to Eq. 4.5 dictates that we need to know the load capacitance of a given path. As all the local periphery in the memory bricks are designed to drive and control the operations of bitcell arrays, their load capacitances can be driven out from array sizes and bitcell capacitances. Any type of bitcell (6T, 8T, or CAM cells) are either custom designed and laid out, or acquired as a hard-IP. There is no modification or resizing done on bitcells. Then a look-up table (LUT) that stores capacitance values of bitcell arrays can be created to be used when local peripheries are sized. For this purpose, a single column of bitcells with 8x, 16x, and 32x rows and a single row of bitcells with 8x, 16x, and 32x bits are created. By performing post-layout RC extraction (RCX) on these instances, we created a LUT for all wordline (WL), bitline (BL), and array read-bitline (ARBL) capacitances, swept over 8 to 32 rows and columns. Two dummy columns for the single column instance and two dummy rows for the single row instance are also added to each side as wrappers to include coupling capacitances.

As a case sizing example, assume that the user requested an 8T bitcell based 1R1W capable memory brick. As the circuit topology set (refer to Chapter 3.2), there are three main local periphery circuits in 8T memory brick; wordline driver, local sense (comprised of read circuit and bitline precharge PFETs), and control block. The user inputs set the bitcell array size and the number of bricks stacked per bank. Assume that the desired array size is W words, each word L bits, and there will be S bricks stacked within a bank. By using the post-layout RCX extraction LUT of bitcell array capacitances, RWL and WWL capacitances are interpolated with respect to L, RBL capacitance is interpolated with respect to W, and ARBL capacitance is interpolated with respect to W and S. For minimizing the loading on the clock network and signal routing at system-level, all gates that receive clock and signal wires in the brick directly from a pin are set to minimum sizes. For the wordline driver, then the input gate capacitance C_{in} is known from minimum gate size $Wmin$, output load capacitance C_{load} is RWL capacitance (or WWL capacitance) and the total gate capacitance coming from the bitcell row, number of stages *N* is 4, and *LE* of each gate are known (*LE* of NAND and inverters under a known β ratio). Then from Eq. 4.5, *SE* for the wordline driver that gives minimum delay can be derived using these parameters. Going through logical effort calculations in the same manner for all stages as shown in Figure 4.2, memory brick peripherals are sized in an automated fashion based on the user inputs for the desired configuration, and by using LUT for bitcell array capacitances and technology characterization parameters at the backend.

The only exception in sizing formulation is for BL precharge (or reset) PFETs. For sizing the reset PFETs, we use the RBL capacitance that is interpolated from the LUT and the delay function for charging a capacitance $t_{ps} = \alpha\left(R_p\right)^K (C)^M$ that is initially characterized for the technology. For the fitting function, total capacitance *C* is $C = 2.C_{RBL} + C_{g_{NAND}}$ with C_{gNAND} the gate capacitance of the skewed NAND gate of the local sense cell, and *Rp* is *Ron* of the PFET. Setting the precharge time as *4xFO4* delays, *Rp* for the PFET is derived from the fitting function, which sets the gate size of reset PFETs. This delay also correlates to the setup time for read enable signal (defined in Chapter 3.2). The BL reset delay can also be set as a parameter if needed.

The 6T bitcell and CAM based bricks have the same cells as 8T memory brick (WL driver, BL reset, control circuits, etc.). Additionally, for the 6T memory brick, sense amps and column muxing cells have fixed sizes. For both 6T brick and CAM brick, there is also additional global BL and select line (SL) to local BL and SL drivers respectively. These cells are sized the same way as WL driver by simply replacing total WL capacitance with total BL or SL capacitances.

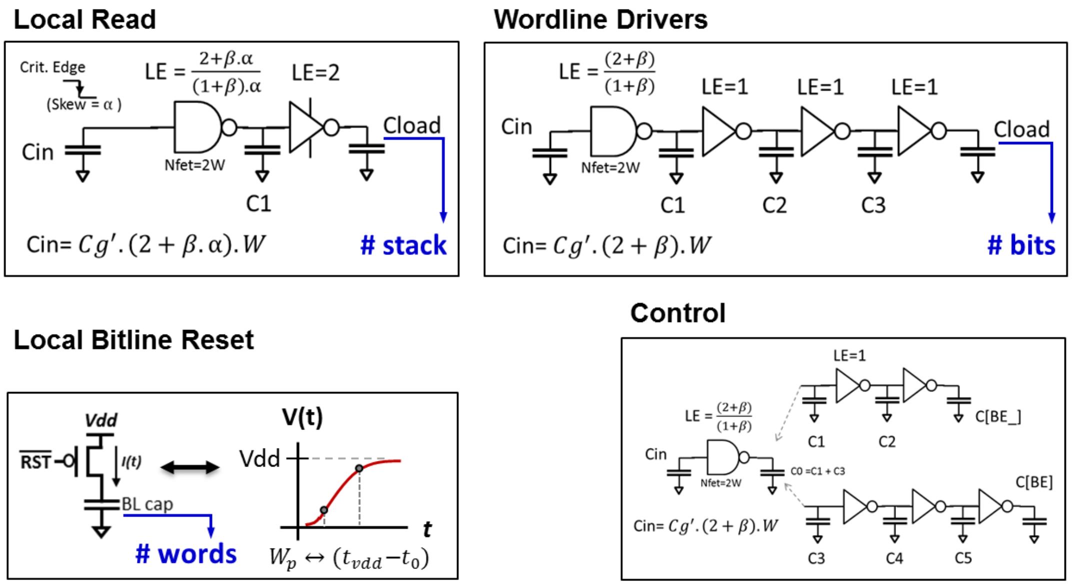


Figure 4.2. Netlist generation for an 8T based 1R1W memory brick based on Logical Effort calculations and curve fitting

By using the logical effort calculations given in this section, the brick netlists are optimally sized for minimum delay with respect to user inputs. As this approach does not require any manual intervention, the formulation automates the netlist generation. This methodology further permits using any other sizing formulation, such as low-power sizing by relaxing the performance. The benefit of this approach is any alternative sizing formulation (such as low power, minimum energy-delay, high speed, etc.) can be easily plugged in to the brick generation flow as a sizing option for the user. Template code in MATLAB for brick generation can be found in Appendix A.

4.1.2 Memory Brick Layout Generator

Once the brick netlist is compiled, its gate sizes are passed to a layout generator. Layout generation is based on "leaf cell" approach, which is similar to conventional SRAM compilers. Leaf cells are pre laid-out template cells that can be modified according to the gate sizes and desired dimensions. Leaf cells are cell layouts of the local periphery circuit blocks, and are initially laid-out in a full custom way. They are pitch-matched to the bitcells, and snap to each other when laid-out in array form with no design rule check (DRC) violations. Brick layout is generated by first modifying leaf cells according to gate sizes set in the netlist, and then arraying these modified leaf cells around the bitcell arrays. Manipulation of the leaf cells are handled in SKILL coding language [61], and the codes are executed in Cadence Virtuoso environment. SKILL coding lets the user to have access to layout shape and layer information, and manipulation of the shapes.

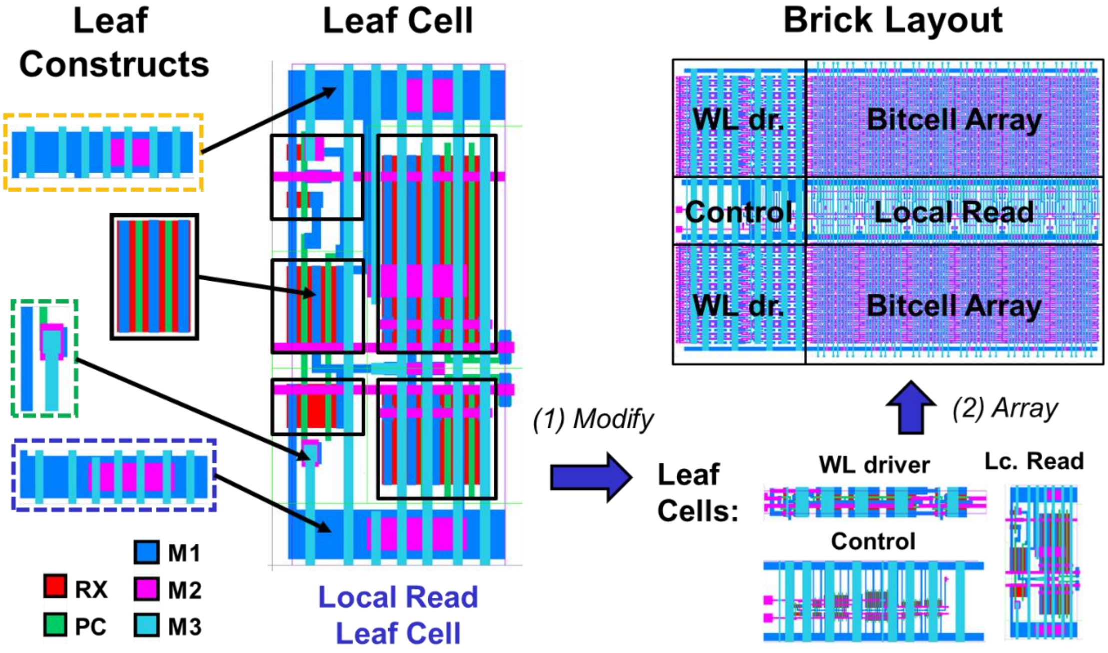


Figure 4.3. Leaf cell based layout generation for an 8T based 1R1W memory brick.

There are no pre-defined functions written in SKILL to easily manipulate layout shapes, so several basic procedures (procedures are equivalent to functions in SKILL) are created initially. These procedures are "stretch", "move", and "copy" for any given shape and layer in any *x*-*y* direction. These basic functionalities are used extensively during the modification of leaf cells. Leaf cells are divided into "leaf constructs" as shown in Figure 4.3, for an easier manipulation of the layout shapes. There are three types of leaf constructs defined; active, connection, and fixed constructs. An "active construct" includes an active region (RX) and poly (PC) to define a transistor. A "connection construct" is used for connecting transistors and includes non-active shapes (PC, metals, or vias) and only used for connections. Similarly, "fixed constructs" are also used for connections and don't include active shapes, but they are fixed layout instances like VDD / VSS rails, or a particular layout shape that never changes with respect to netlist. Leaf cells (and in parallel leaf constructs) are designed and laid-out with minimum possible gate sizes so that they are either enlarged or left un-modified. This approach ensures the final layout to be DRC clean in terms of RX enclosures and minimum gate area rules.

Layout generation starts with modifying the leaf cells with respect to input gate sizes. Consider the local read cell of an 8T bitcell based memory brick, as shown in Figure 4.3. First the active constructs are modified with respect to the transistor sizes. RX, PC and metal layers are stretched to match the new gate size, and via arrays are added if needed. When adding new via shapes, enclosure rules have to be always met so that there is no DRC errors afterwards. Once every active leaf construct of the leaf cell are modified, they are placed to their dedicated slots within the leaf cell as depicted in Figure 4.3. Any fixed constructs that are abutting the active constructs are also placed at this point. Then the new dimensions of the leaf cell under modification is checked. With respect to the new dimensions, connection constructs are modified and placed into their slots in the leaf cell, followed by placing and stitching the remaining fixed constructs. This assures that a modified leaf cell layout is comprised of the updated gate sizes, as shown in Figure 4.3. Modify-and-stich flow ensures no violation of design rules.

The same flow is repeated for all the leaf cells for the given brick. Then the bitcells are arrayed with respect to the array size, and all the modified leaf cells are arrayed around the bitcell arrays accordingly (Figure 4.3). Since all the leaf cells are pitch-matched to the bitcells, no DRC error results from tiling the cells together. Finally, any fixed "wrapper" layout that the brick may require are also put around the final layout. Wrapper layouts can generally include body connections, supply rails, or simply empty "pr boundary" boxes that make the final layout dimensions multiples of standard cell library gate dimensions. For both 6T bitcell based and CAM bricks, the layout generation flow is exactly identical. For the 6T bricks, sense amps and column muxing instances have always fixed gate sizes, so they are placed as non-modified leaf cells. A pseudo-code in SKILL for leaf cell modifications are given in Appendix A.

4.1.3 Performance and Energy Estimation of Memory Bricks

After an optimized netlist and its corresponding layout are generated, the final step in automation is to generate a parameterized library model of the brick. Brick library file includes critical path, energy, area, and setup & hold times that are needed for use in the subsequent synthesis flow. The gate components within the brick netlist are each represented by LUT models based on bilinear interpolation and curve fitting for delay and energy as a function of fanout and slew rate. The LUT based estimation method is similar to tabulated empirical gate delay modeling (or nonlinear delay modeling [2]) approach that is extensively used in conventional ASIC synthesis flows for performance estimations [58].

I) Modeling

Simple RC based estimation for gate delays are not accurate enough as it does not cover the effects of input slew rate, short-circuit in between PFET and NFET network, dynamic effects, transistor stacking effects, or effective capacitance. For an accurate estimation for generating a library, we need another approach. Since brick topology is known, all possible gate types that will be used in a netlist is also known. Physical information (such as number of gate fingers) for each gate in the netlist are known as well from the pre laid-out leaf cells. Then we can characterize each gate type by using SPICE simulations for delay and energy, store the characterizations in a LUT with respect to input driver and output load information, and then use interpolation to estimate the performance of a gate in the netlist. This characterization, however, is technology and PDK dependent, so it has to be carried out initially.

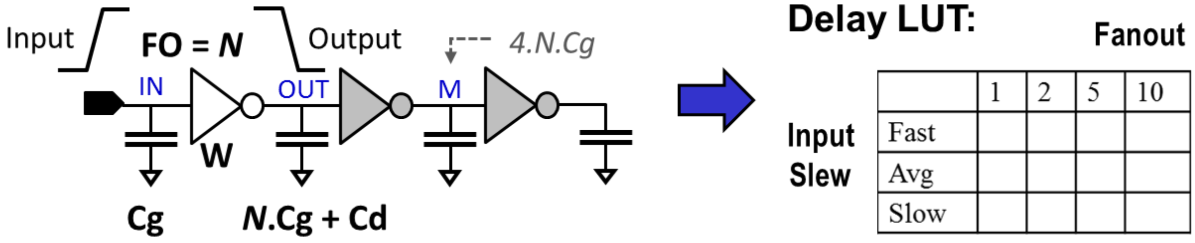


Figure 4.4. SPICE simulation results for delay characterization of a gate are stored in a look-up table (LUT), in form of input slew vs. fan-out of the gate.

Our SPICE-simulation-based gate characterization test-bench is illustrated in Figure 4.4. The gate with width *W* drives the same type of gate with width *N x W*, which sets the fanout of the gate under characterization to be *N*. Input gate capacitance is denoted as *Cg* and the parasitic load of the gate is denoted as *Cd*, which makes the total load capacitance to be *"N x Cg + Cd"*. Input slew of the gate is driven by a pulse generator, and the shape of the signal rise/fall type is set to be half-sine for a realistic input drive. An extra gate with total width of *4 x (N x W)* is placed to the output of the load gate to eliminate Miller Effect [2], such that *node M* does not dynamically affect *node OUT* through parasitic *Cgd* capacitance in an unrealistic way by switching very fast. Slew of the input can be fast, average, or slow (1xFO4, 2xFO4, and 4xFO4 delays respectively), with slew time defined as signal rise/fall from 20% to 80% of the rail. Definition of fast, average, and slow slews are technology, PDK, and standard cell library dependent, and can be changed without affecting the flow. Using this test infrastructure, a SPICE level parameter sweep is performed for the gate, sweeping input slew rate (fast, avg, and slow) and fanout (from 1 to 10, by incrementing 1). Mid-to-mid delays for high-to-low and low-to-high, and output slew for rise and fall times are then all stored in respective LUTs. Mid-to-mid delay is defined as delay from 50% of input signal to 50% of output signal. Gate finger count (pc = 1, 2, 6) is also added as an extra parameter to the sweep when needed.

The same characterization mechanism is used for energy estimation by checking total energy drawn from the supply for a full switching event of gate charging and then discharging the load capacitance of *node OUT*. Then energy is calculated by taking the integral of instantaneous power over the switching time interval. Equations for the energy calculations are as follows:

$$P_{instantenous}(t) = I(t).V(t) \quad , \quad P_{average} = I_{average}.Vdd \qquad Eq.\ 4.6$$

$$E_{switching} = \int_{0}^{T_{switching}} P_{instantenous}(t).dt = \int_{0}^{T_{switching}} -I_{supply}(t).V_{supply}(t).dt \qquad Eq.\ 4.7$$

$$P_{average} = \frac{E}{T} \Rightarrow E_{switching} = T_{switching}.\left(I_{average}.Vdd\right) \qquad Eq.\ 4.8$$

Based on Eq. 4.6, either Eq. 4.7 or Eq. 4.8 can be used to tabulate the energy of a switching event in SPICE simulations. We use both Eq. 4.7 and Eq. 4.8 as a double-checking mechanism for the validity of the results gathered from the simulations. Note that for a full switching event, first the

PFET network is ON and draws current from the supply and charges the capacitance. When PFET network is OFF and NFET network is ON, the charge stored in the capacitance is discharged to the ground, without drawing supply current. Using this test-bench, gate width and fanout are swept in parametric SPICE runs. Input slew is also added as the third parameter to the sweep. We found that energy based characterization is easier, as the power numbers that are needed for the brick library can be calculated using Eq. 4.8 with respect to the brick frequency.

By using this characterization approach, all circuit level effects are captured within the simulations, namely input slew rate to output driving time, parasitic capacitances of the transistors, effects of transistor stacking, energy lost on short-circuit power, and effect of input slew to short-circuit power. Characterized gates under this approach are inverter (pc=1, 2), NAND (regular Vt and low Vt, with pc=1,2), Tri-state buffer (RVt and LVt, pc = 1, 2, 6, and with load capacitance of C_{ARBL} *x Stack#*), and Sense Amp (fixed size, with load RVt tri-sate buffer gate capacitances).

One modeling phenomena that cannot be captured by this test bench is the effective capacitance of long RC interconnects [62]. As a wire gets longer, its parasitic metal wiring resistance gets larger and becomes dominant relative to the *Ron* of the driver transistor. Large resistance of the RC network then shields some of the capacitance at the end of the network, changing the effective capacitance that the driver "sees" and ultimately changes the gate delay. As large bitcell array sizes can present large RC for the WLs or BLs, the impact of effective capacitance has to be embedded into our LUTs for efficient and accurate estimation. This issue is solved by replacing the load gate with a single row or column of RC extracted bitcell arrays. Instead of sweeping for fanout for the output load, the sweeping parameter for the LUT becomes the number of bits per row for WL, and words per column for BL. Extra LUT cases that come from this approach are an inverter driving a WL (pc=2, variable wire load = number of bits per row), single bitcell driving a BL (fixed bitcell, variable wire load = number of rows per column), and reset PFETs pre-charging BL (pc=1, variable wire load = number of rows per column). Bitcells for the bricks are also chosen accordingly. For 8T brick or CAM brick, an 8T bitcell drives the load of LVt NAND gate capacitance. For a 6T brick, a 6T bitcell drives the load of fixed sense amp. With this approach, both RC effects of the BL and WL, and delay/energy dependency on the size of the array is embedded into the corresponding LUTs.

II) Estimation

By using our LUT based energy and delay modeling approach, every possible gate combination that will be encountered in a given brick netlist is covered. For each gate and possible loading scenarios, LUTs storing mid-to-mid delay, output slew, and switch energy are ready. Output slew of a gate is the input slew of the consecutive gate in the path. Since brick netlist and layout are generated before the estimation phase, size of each gate, their paths, and their final loads are known. By performing bilinear interpolation and curve fitting based on the LUTs, delay and energy for each gate are estimated. Fitting functions in form of $y(x) = a.x^b + c$ are added for certain gate delays that are showing exponential delay behavior for better accuracy. By summing up individual energy and delay estimation numbers of every gate under a brick operation, critical path delay, and read / write / no-operation energies for a given brick netlist are automatically estimated. Leakage for the inactive rows are also included into the energy estimations by using a separate LUT that stores leakage energy in terms of number of bits per row. Leakage coming from inactive bits within an active row are embedded in the read and write energy LUTs.

As an example, Figure 4.5 illustrates how path delay is estimated for the WL driver. At the start of the operation, Decoded WL (DWL) signal is ready at high and the operation starts with clock signal (CLK) going high. So the delay estimation for the this signal path starts with input rising and NAND output falling, and then signal traversing all the way to WL load, and rising the WL node. For worst case scenario, any signal coming from outside of the brick is assumed to be received with a slow slew. So for NAND gate, input slew is slow, fanout is W_2/W_1, and gate has single poly finger (pc=1). With these known parameters, NAND high-to-low delay *t1*, and output fall slew *s1* are interpolated from the appropriate LUTs. The same procedure is then repeated for INV2 and INV3 in the path with input slew coming from the output slew of the previous gate. Since INV4 is driving the WL, its delay and slew numbers are interpolated from the bitcell based LUT by using number of bits per row for the load. At the end, every delay is summed up to get the path delay, and the output slew of the path is equal to *s4*. Energy of WL driver switching WL to 1 then to 0 is calculated with the same approach by using energy LUTs.

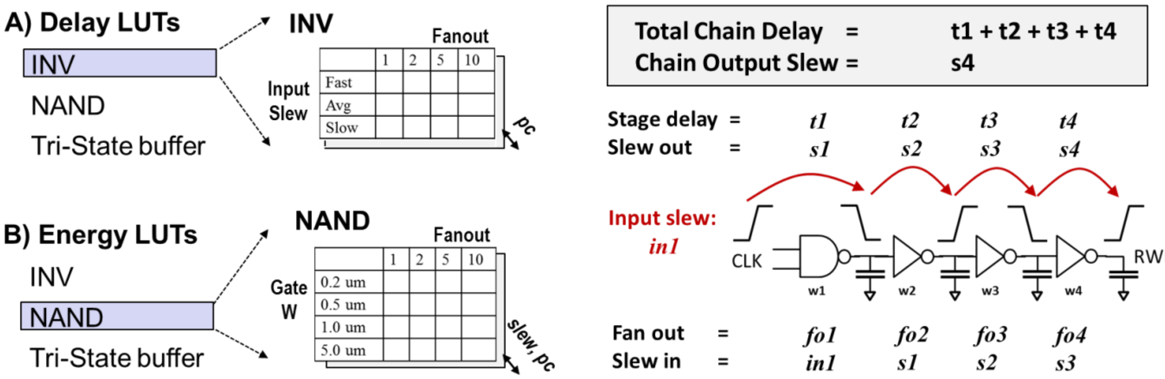


Figure 4.5. LUT based delay and energy estimation for read wordline (RWL) driver stage

The same path delay and energy estimations are performed on all the activated paths depending on the brick operation, and then are summed up to calculate the critical path delay and energy of the brick. Energy calculations assume reading and writing alternating bits of <10101…>, with last bit always set to 0 for a worst case delay from the farthest placed bit. Energy is estimated for read, write, read-write, read-write with area windowing (only for 6T bricks), and no-operation. Critical path, output slew, and energy for all operations are then put into a library file. With this approach, the library file of the brick is generated instantaneously without the need for any simulations or manual intervention. A pseudo-code in MATLAB is provided in Appendix A.

Liberty format (.lib) is used for the library files for compatibility with synthesis tools. Area information and routing blockages are generated by the layout generator and are also part of the brick library model. All input pin capacitances are known from leaf cells. For an 8T brick, there is a hold time restriction on write BLs and setup time restriction on read enable signal. During write operation, there is a delay between CLK going low and WL going low to shut off the bitcells. During this small time window where the WL is still high and bitcells are "writable", the write BLs should hold their current values; otherwise the bitcells can go into a metastable state or flip to the wrong logic state, causing a logical-failure. This indicates the need of a hold time on write BLs with respect to CLK falling-edge. Hold time for write BLs is estimated for WL going back to 0 after CLK goes low. As for read enable signal, all the read BLs have to be precharged to Vdd before a read operation can start. There is, however, a delay between read enable signal going high and the reset PFETs finish precharging the BLs. This indicates a setup time for read enable signal with respect to CLK rising-edge. Setup time for read enable is estimated from the total delay of reset PFETs pre-charging BLs plus the delay of reset control signal. Other types of setup and hold constraints for 6T and CAM bricks are calculated the same way.

III) Accuracy

To validate the accuracy of our estimation tool, two memory bricks with 16x10bits and 32x12bits sizes are generated by the compiler, and their estimated read path delays and read energies are compared to SPICE simulations with RC extracted bitcell array layouts. Results for critical path delay (read delay) and read energy are summarized in Table 4.1 for different bank sizes of 1x, 2x, 4x, and 8x stacked bricks, and for reading a word of alternating bits (<1010...10>). For both bricks, error rates are within 2-7% for critical path estimation, within 0-4% for read energy estimation, and 0-2% for write energy estimation. General trend of results indicates that our estimation accuracy gets better when array sizes are larger and/or there are more stacked bricks. Breakdown of the results for tool estimation and SPICE simulations are summarized in Table 4.2 (in percentages). CLK refers to clock received, RWL refers to driving read word line, and RBL refers to driving read bitline. Ratios of estimated delay/energy of a sub-operation to the overall estimated delay/energy are within +/- 1% of SPICE simulation breakdown. This accuracy enables the user to get valuable feedback from the estimation tool on where the bottlenecks are happening within a brick to better optimize circuit-level choices.

# of stacked bricks	**Brick 16x10bits**			**Brick 32x12bits**		
Critical path [ps]	**Tool**	**SPICE**	**Error**	**Tool**	**SPICE**	**Error**
1x	247	265	**6.6%**	295	307	**4.0%**
2x	256	273	**6.2%**	305	316	**3.4%**
4x	269	285	**5.5%**	322	331	**2.6%**
8x	292	307	**4.9%**	353	359	**1.8%**
Read Energy [pJ]	**Tool**	**SPICE**	**Error**	**Tool**	**SPICE**	**Error**
1x	0.54	0.54	**-0.1%**	0.65	0.63	**3.1%**
2x	0.59	0.59	**0.5%**	0.73	0.70	**3.5%**
4x	0.71	0.70	**1.1%**	0.88	0.85	**3.6%**
8x	0.93	0.92	**1.3%**	1.19	1.16	**2.8%**

Table 4.1. Tool estimation vs SPICE simulation (on RC extracted arrays) for read delay and energy.

Performance	**CLK to RWL**	**RWL to RBL**	**RBL to OUT**
Tool estimated	38%	25%	37%
SPICE	37%	26%	37%
Read Energy	**Driving RWL**	**Local Sense**	**Control**
Tool estimated	2%	74%	24%
SPICE	2%	73%	25%
Write Energy	**Driving WWL**	**Write-in Array**	
Tool estimated	55%	45%	
SPICE	55%	45%	

Table 4.2. Performance and energy breakdown comparison for stages.

4.2 Automated Synthesis of LiM designs

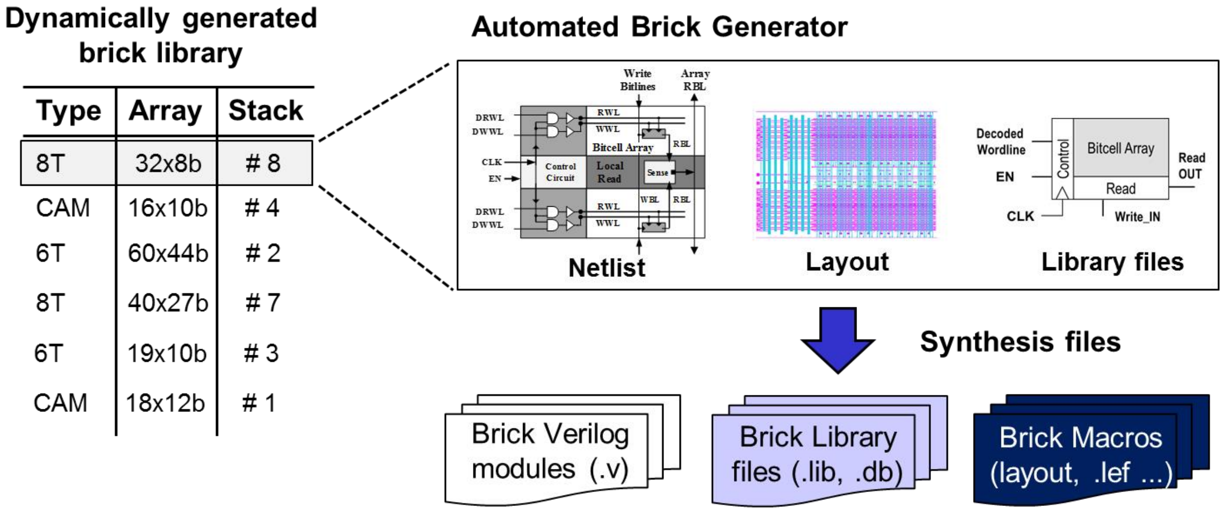
Dynamically generated brick library covers all memory brick sizes, types, and aspect ratios whereby bricks are generated automatically and instantaneously with respect to user inputs. Automated LiM synthesis framework is realized by integrating the dynamically generated brick library with the conventional ASIC synthesis tools. Bricks are generated as macro cells before starting the synthesis. There is no restriction on number of rows or bits per brick, or the number of stacked bricks, as every gate is sized optimally using LE calculations. With this approach, any unconventional bit, row, and stacking numbers (non-power of 2) are permitted in our methodology. This opens up many interesting possibilities for algorithm level optimizations.

There is no design-cost for the user to generate a memory brick. As circuit compilation, layout generation, and performance estimation are technology dependent, however, there is an initial design-cost of technology characterization. Technology related characterization of delay-energy LUTs and leaf cells have to be re-implemented when the designer moves to a new technology or a new PDK. This cost, however, is one time only, and vendor supplied memory compilers go through the same iteration as well.

4.2.1 Overview of the Automated Synthesis Flow

For every generated memory brick, the tool also creates necessary synthesis files as shown in Figure 4.6. Bricks are represented in Verilog modules (.v) for integrating with RTL description of the system. Netlist generator passes the array size to the Verilog file. For logical and physical synthesis tools, the performance, energy, and area information is passed through library files in Synopsys Liberty and database formats (.lib, .db). Generated layout is first transformed into an abstract file using Cadence Abstract generator, and then Library/Library Exchange Format (.lef) and Milkyway format files are generated. Layout is also streamed out as Graphic Database System (GDSII, or .gds) format so that it can be used in any compatible tool. Template .lib file for a brick is given in Appendix A.

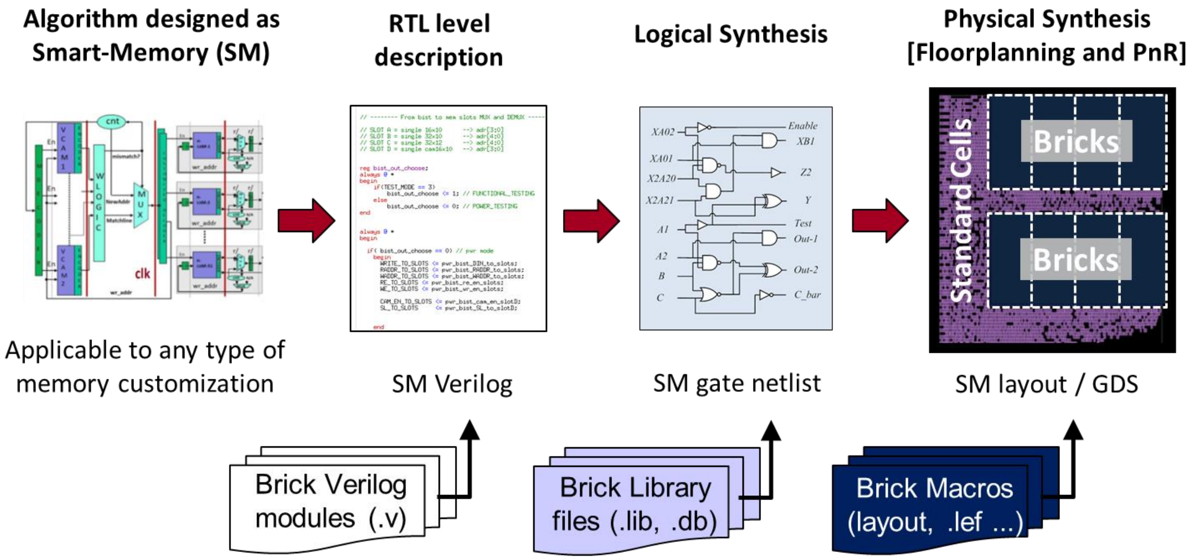
Once the brick synthesis files are generated, it is straightforward to integrate them with the existing ASIC synthesis flows since the files are compatible with them. Bricks are used and integrated as macro files into the ASIC synthesis flow. Overview of the LiM synthesis flow is shown in Figure 4.7. The user first designs the system conceptually as a smart-memory (SM) and defines what brick instances are needed. Then these brick sizes and types are given as user inputs to the brick library. Brick library generates all the desired bricks automatically without any design cost. The user then implements the SM in HDL and the bricks are instantiated as Verilog modules. The SM goes through logical synthesis to create the gate-netlist, and bricks are integrated as .lib (and .db) files. Finally, the gate-netlist goes through physical synthesis wherein the bricks are integrated as library (.lib, .db) and macro cell (.lef, milkyway) formats. SM layout is implemented by going through the conventional flow of floor planning, place and routing (PnR), power and clock network implementation, and verification. Bricks are placed by the user.



Dynamically generated brick library

Type	Array	Stack
8T	32x8b	# 8
CAM	16x10b	# 4
6T	60x44b	# 2
8T	40x27b	# 7
6T	19x10b	# 3
CAM	18x12b	# 1

Figure 4.6. Dynamically generated brick library and synthesis files. Brick array sizes and stack numbers are chosen arbitrarily to show possible examples.



Algorithm designed as Smart-Memory (SM)
RTL level description
Logical Synthesis
Physical Synthesis [Floorplanning and PnR]
cnt
clk
XA02
XA01
X2A20
X2A21
A1
A2
B
C
Enable
XB1
Z2
Y
Test
Out-1
Out-2
C_bar
Standard Cells
Bricks
Bricks
Applicable to any type of memory customization
SM Verilog
SM gate netlist
SM layout / GDS
Brick Verilog modules (.v)
Brick Library files (.lib, .db)
Brick Macros (layout, .lef ...)

Figure 4.7. Overview of automated LiM synthesis flow.

An important result of this flow is that it enables the user not to finalize the decision on the memory sizes initially. Since there is no extra design cost of generating memory bricks, the user can perform a rapid design-space exploration to find the optimum sizes and memory configurations for the chosen application. A simplified example for rapid design-space exploration is given in the following section, and a system-level case study is discussed in detail in Chapter 6.

4.2.2 Rapid Design-Space Exploration Example

Enabled by the automated brick generation, we performed rapid design-space exploration to compare various system-level tradeoffs for a simplified case study in Figure 4.8. Three SRAMs with single partitioning of sizes 128x8bits, 128x16bits, and 128x32bits were created. To analyze the impact of brick array size on memory performance, each of the 128xN bit SRAM partitions were built with three different bricks of sizes 16xN bit, 32xN bit, and 64xN bit by stacking them 8x, 4x, and 2x times respectively. Overall, 9 different bricks are compiled with optimum gate sizing (word numbers: 16, 32, 64 and bits: 8, 16, 32).

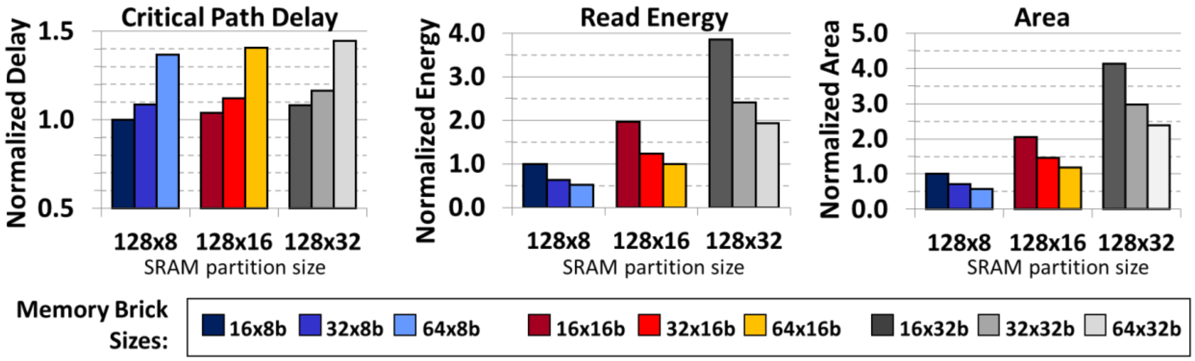


Figure 4.8. Simplified design-space exploration example for different SRAMs with single partitions, all synthesized by using different sizes of memory bricks

Performance, energy, and area consumption of these partitions are estimated within seconds by our library generation tool, and the normalized results are summarized in Figure 4.8. As the brick size gets larger, critical path also increases since a brick with larger array size has longer local RBLs. Within the same sized partitions, however, partition with larger bricks consume less energy and area as they have less number of local sense and control blocks per number of words. More interesting results are observed when different memory sizes are cross-analyzed. For instance, 128x16bit memory built with 16x16bit bricks is still faster than 128x8bit memory built with 64x8bit bricks, while it consumes nearly the same energy as the 128x32bit memory built with 64x32bit bricks. These results show that array size of the brick and the number of bricks per partition have equally important impact on the overall performance as to overall memory size itself. For this analysis, compiling the netlists and generating the library estimations were finalized within 2 seconds of wall clock time. Thus, the same analysis can be done over a finer resolution of row numbers and bit length without any design cost. Library files for any chosen configuration can be then simply fed into a gate-level synthesis tool (e.g. Synopsys DC) to perform system-level analysis.

To capture the impact of application-specific changes on the system, timing and power analysis are done after logic synthesis. For analyzing the tradeoffs of various memory partitioning and floor planning choices of bricks, a parameterized Verilog code with parameters describing different memory configurations is used (such as array sizes, banking, memory hierarchy, etc.). For analyzing modifications to the algorithm, the design itself is parameterized by using object oriented tools like "Stanford Chip Generator" [63] [64]. Different RTLs can be generated by scripts to explore different corners of the algorithm with varying bus widths, number of parallel cores, or choice of the used arithmetic blocks. A system-level design-space exploration study is discussed in great detail in Chapter 6.

4.3 Accuracy of Estimated Library Generation

To validate our automated synthesis flow we implemented a LiM based test-chip that includes different sizes and configurations of synthesized 1R1W SRAMs in a commercial 65nm CMOS technology. Using the same 8T bitcell based memory brick of array size 16x10bits, we implemented different sizes and configurations of SRAMs (Figure 4.9). By stacking the 16x10bit brick for 1x, 2x, 4x, and 8x times to form a single partition, we implemented 1R1W SRAMs with sizes 16x10bits, 32x10bits, 64x10bits, and 128x10bits respectively (configurations *A*, *B*, *C*, and *D*). We also constructed an SRAM of size 128x10bits with 4 partitions (configuration *E*) such that each bank has a size of 32x10bits formed by stacking two 16x10bit bricks. Chip measurements for performance and power are aggregated from multiple chips, under nominal Vdd of 1.2V and room temperature. Simulations on synthesized netlists are done in Synopsys PrimeTime (PT) using standard cell libraries, generated brick libraries, RC parasitic file for routing (.spef), and switching activity file (.saif) that is generated in Modelsim for accuracy.

Comparison of chip measurements and simulations based on the estimated brick libraries are summarized in Figure 4.10 for the SRAM configurations. Performance is reported in GHz, and chip measurements are averaged out of multiple chips with maximum and minimum tested speeds shown as bars. Simulation results are shown for best, worst, and nominal cases. Energy numbers are reported for the maximum respected frequencies, and are normalized with respect to the smallest SRAM for ease of analysis.

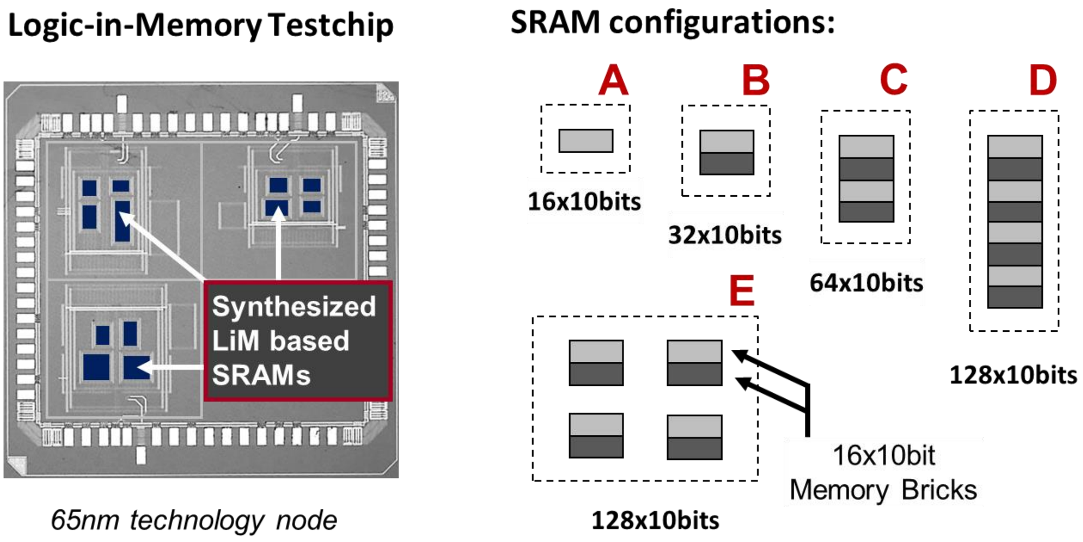


Figure 4.9. LiM test chip containing different sizes and configurations of synthesized SRAM blocks.

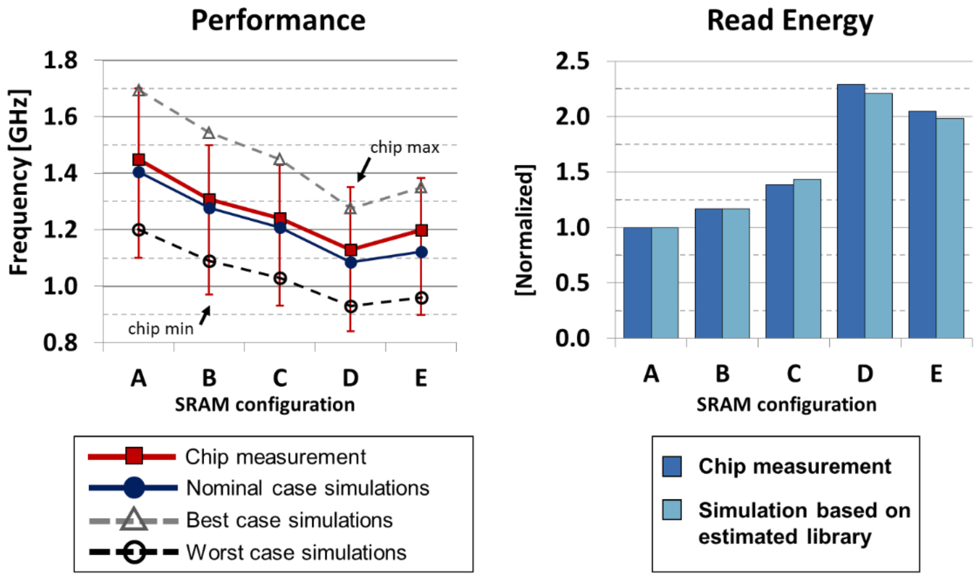


Figure 4.10. Comparison of chip measurements to estimated library based simulations for taped out SRAM configurations.

When we analyze both performance and energy, we observe that simulation results are in line with chip measurements and capture the trend of chip results over the range of different configurations within a small error rate. As SRAM size increases for a single partition (from *A* to *D*), performance drops and energy increases as it is expected. For the same size of SRAMs *D* and *E*, partitioning results in faster performance in *E*. Although individual partitions of *E* have the same size with *B, E* is still slower than *B* due to its slower decoder and global signal routing coming from its larger size. Banks of *E* are implemented such that only the bank with the read address hit is activated during read, thus making *E* consume less energy compared to *D*. This gain in energy and performance of *E* however is traded off with larger area consumption that inherently comes from partitioning, when compared to *D*.

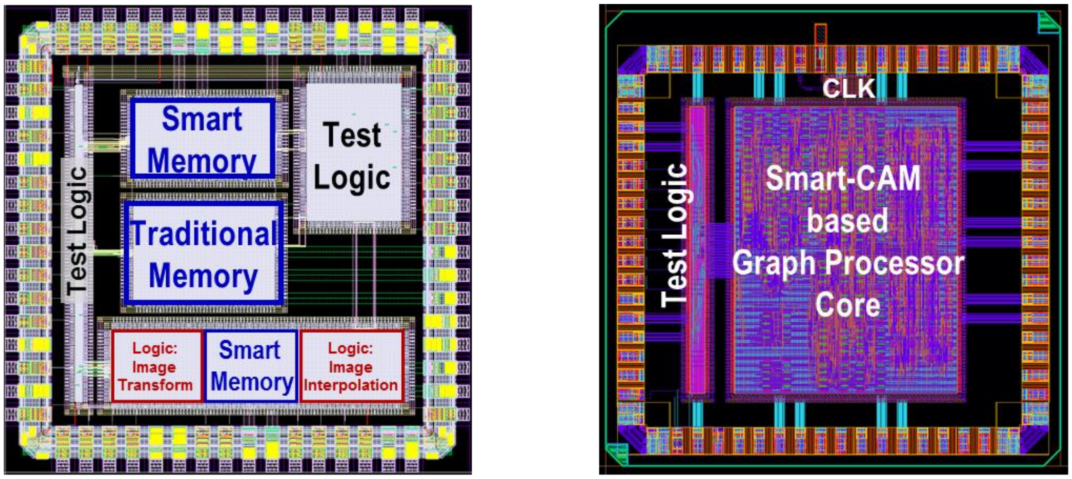
These results validate the accuracy of our automatically generated libraries and capture the circuit behavior of the memory bricks efficiently. Error rates are also consistent with the circuit-level results given in Table 4.1. As system-level simulations based on dynamically generated libraries can capture the trade-offs in between different configurations efficiently, our flow opens up many opportunities for design-space exploration.

4.4 Algorithm and Hardware Co-design

Building a customized memory provides benefits at the circuit-level, but LiM synthesis facilitates broader benefits at the system-level. At the low-level, all memory and logic are represented at the same level of abstraction, and the white-box memory elements have no hard boundaries. Therefore, all architecture and hardware customizations can be efficiently realized at the RTL. Memory arrays of any size can be integrated with logic in a fine-grained manner, thereby enabling efficient data-streams to be built – even for large datasets. Now that the hardware is customizable beyond what a conventional ASIC flow permits, the algorithm constructions can exploit application-level knowledge more aggressively at the high level. With the LiM synthesis flow facilitating the hardware and algorithm co-optimization, superior system-level designs can be realized.

5 Silicon Validation for Two Data-Intensive Applications

To demonstrate the efficacy of LiM synthesis we have implemented two data-intensive applications in silicon. First implementation is a smart-memory based Synthetic Aperture Radar (SAR) image reformatting block that is built in commercial 130nm CMOS technology. Second implementation is smart-CAM based low-power accelerator core for graph processing algorithms that is built in commercial 65nm CMOS technology. Chip-level layouts for both systems are shown in Figure 5.1.



Synthetic Aperture Radar Image Reformatting

Low-power Accelerator for Graph Processing

Figure 5.1. Chip layouts for LiM based system demonstrations

Both systems are implemented using the LiM synthesis flow, and optimal design points were identified after performing design-space exploration. Our silicon results show that the circuit-level customizations provide significant benefits at the system-level, but that co-optimization of the algorithm with the hardware provides dramatic system-level benefits.

5.1 Synthetic Aperture Radar Image Reformatting

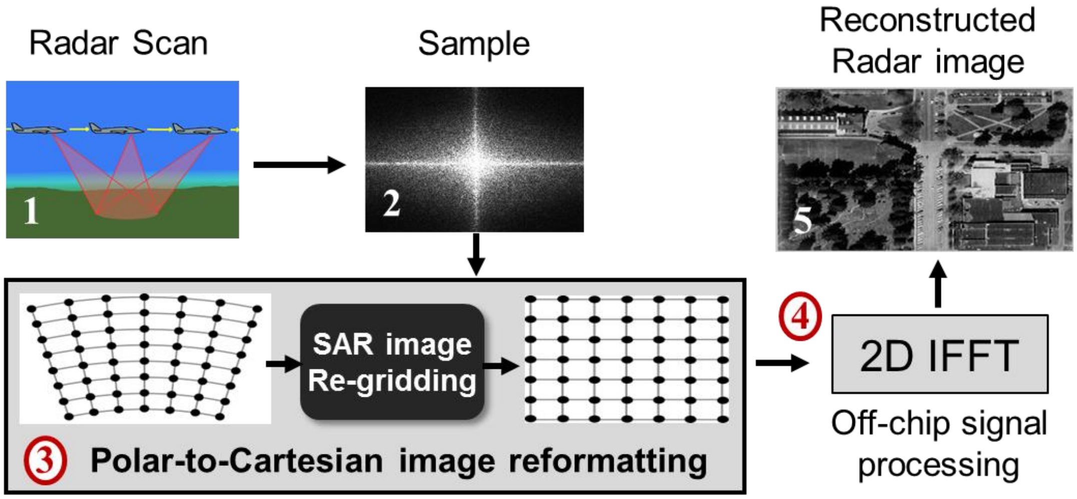

Radar Scan
1
Sample
2
Reconstructed Radar image
5
SAR image Re-gridding
③ Polar-to-Cartesian image reformatting
④
2D IFFT
Off-chip signal processing

Figure 5.2. Overview of Synthetic Aperture Radar (SAR) polar-to-Cartesian image reformatting. Flow picture modified with permission from [18].

Synthetic Aperture Radar (SAR) is a data- and compute-intensive application that is similar to taking a picture using radar data [65] [66]. SAR requires translation of sampled radar data from Polar to Cartesian coordinates that can be fed into a 2-D inverse Fast Fourier Transform (FFT) to reconstruct an image as shown in Figure 5.2. This polar to rectangular grid conversion using FFT-based interpolation is an essential step for SAR [65]. FFT-based algorithms, however, have multiple non-local passes on stride-accesses leading to high data traffic, thereby creating energy and performance challenges [39].

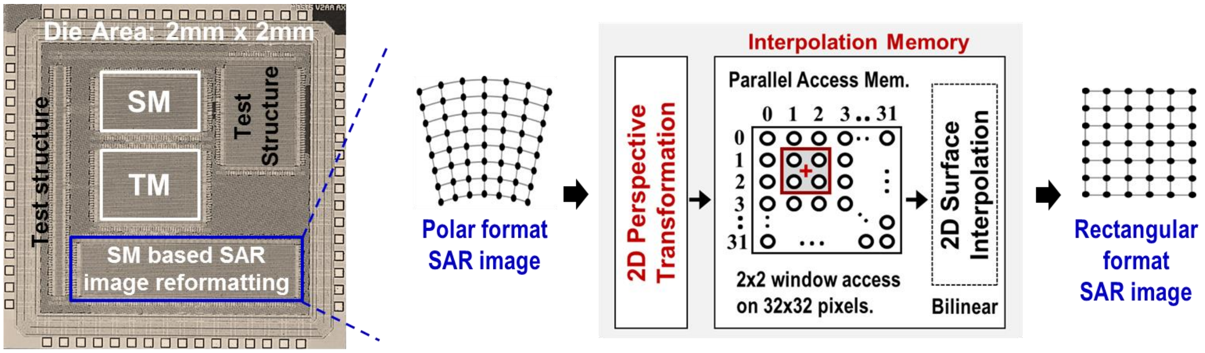
To address this challenge, Zhu et al. proposed a LiM based design approach for a localized polar to rectangular conversion with an error rate comparable to FFT-based conversion in [39] [38] [20]. They used a mapping function that relies on simple arithmetic operations and a smart interpolation memory. The interpolation memory is a “seed table” based on localized bilinear interpolation that performs as if it readily stores all interpolated data, but it actually computes the data on the fly by using a smaller seed table and an arithmetic unit. It is further shown in [40] that a low-power LiM layer based on the same concept can be built for a 3D IC stack to accelerate the SAR by co-designing the algorithm and system architecture to reduce the data-intensity of the application.

5.1.1 LiM Based SAR Architecture and Test-chip

For a 2D interpolation operation, a constant number of spatially neighboring elements within a rectangular window are needed. This makes a parallel-access memory (as discussed in Chapter 2.1) an ideal candidate to implement a single-cycle interpolation-memory that would increase the system performance and throughput. The parallel access memory stores a 2D image pixel array with a size of $K \times L$, and allows random access of pixels in a window of $m \times n$ in a single cycle (where $m<K$ and $n<L$). For a traditional ASIC approach, parallel access memory is realized by implementing logic blocks next to parallel accessible SRAM banks. 2D pixels are distributed to $m.n$ parallel memory banks for a conflict-free access, however, this does not exploit the address pattern commonality between the accessed pixels. Moreover, area and energy penalties are incurred when the image or access window size is large.

For the same functionality, the smart-memory in [32] exploited the address pattern commonality and implemented an application-specific SRAM with shared and customized decoders. Row decoders are shared between *m* banks and customized to activate *n* adjacent wordlines with respect to single address. A column decoder is added under each bank group to select a single element per column from the activated multiple rows. With this customization, the same parallel access functionality can be handled inside the memory block with significantly less power and area.

Using the LiM synthesis framework (Chapter 3 and Chapter 4), it is straightforward to synthesize a smart parallel-access memory. At the RTL level, row decoders are customized to fire up *n* adjacent wordlines with respect to incoming address. A column decoder and AND gates are added to choose any elements within the *n* activated rows. Memory bricks are grouped and stacked to form *m* partitions, and row decoders are shared between the partitions to exploit address commonality to save area. Single-cycle parallel-access memory is then combined with a 2D bi-linear interpolation block to build the interpolation memory.



Commercial 130nm technology

Polar-to-Cartesian image reformatting

Figure 5.3. SAR test-chip and LiM based architecture for image reformatting

To validate the LiM synthesis approach for SAR, we implemented a LiM based image reformatting chip in a commercial 130nm CMOS technology. As shown in Figure 5.3, the architecture of the LiM based SAR image reformatting system is a perspective transformation block followed by an interpolation memory, which is the parallel-access memory that feeds data to surface interpolation logic. Chosen image size is 128x128x16bit pixels, and it is tiled into 16 tiles. For the chosen tiling size, the LiM system has 0.41 mm^2 area.

In addition, we also implemented smart-memory based (SM) and traditional banked memory (TM) based 2x2 parallel access memories on the same test chip for comparison purposes. Both memory blocks are implemented by utilizing the same memory bricks and synthesis flow for a fair comparison. A micrograph of the taped-out chip of area 2mm x 2mm is shown in Figure 5.3.

5.1.2 SAR Test-chip Results

When we first compare the benefits of using a smart-memory (SM) based parallel-access memory over a traditional banked memory (TM), we observe that SM is more power and area efficient, whereas TM has a better performance as shown in Figure 5.4.

In the SM based parallel-access memory, row decoders are shared. Although there are additional column decoders, area penalty coming from partitioning a monolithic memory block is minimized in SM when compared to TM. Therefore, SM based parallel-access memory is 31% smaller. Furthermore, the customization embedded inside the SM combined with smaller area results in less global signal traffic, leading to 29% less power consumption. However, SM has a longer critical path compared to TM as it has a customized decoder that activates two adjacent wordlines shared by two partitions and column decoders performing parallel selection. As the memory read path is longer, SM is 24% slower than TM. Therefore, in this specific smart memory implementation, we are trading-off performance for power and area benefits when compared to a banked memory. It is worth noting, however, that by carefully timing different operations in the SM read path the performance penalty can be minimized. These results demonstrate that our LiM synthesis framework enables efficient circuit-level customization of memory blocks.

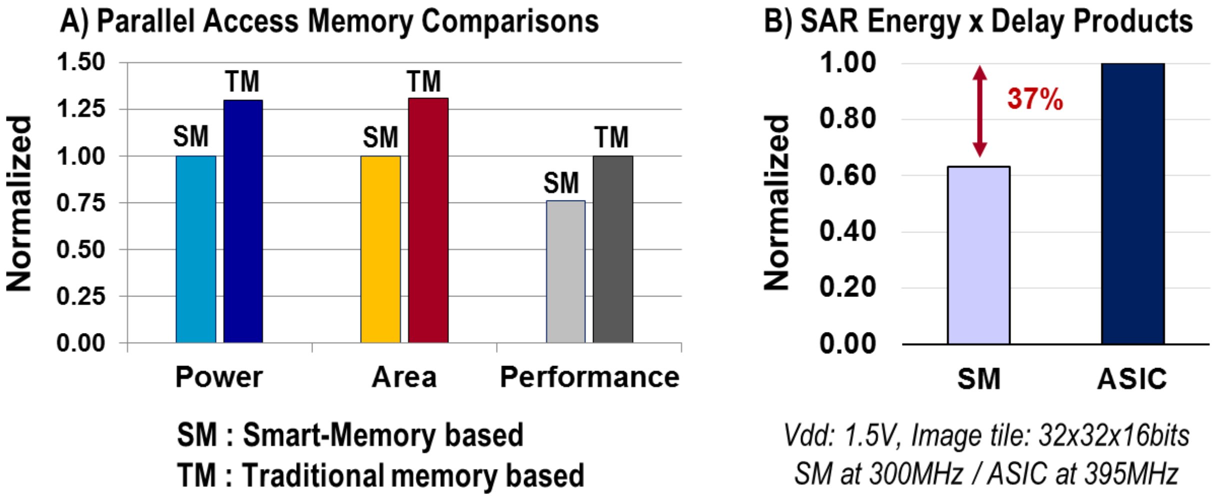
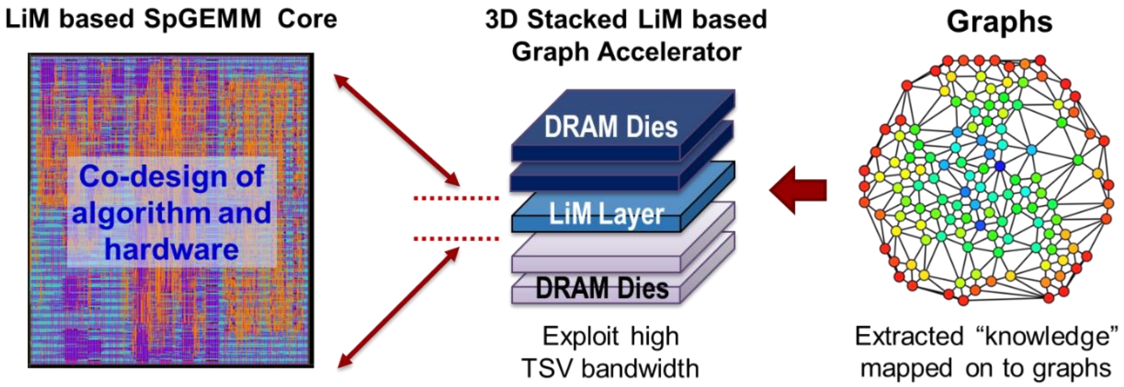


Figure 5.4. SAR chip results. (a) Comparison of SM and TM based parallel access memories. (b) SAR image reformatting Energy x Delay product results for SM based and conventional approaches.

Functionality of the SAR image reformatting system is verified on tiles of a test image, with the tile size of 32x32x16bits. The synthesized LiM system works correctly within the error rate that is provided in [20]. System-level energy comparison is done by back annotating the chip results of 2x2 parallel-access SM and TM for SAR image reformatting system. At 1.5V nominal supply voltage and for an image size of 128x128x16bits, SM based SAR implementation working at 300MHz clock frequency consumes 37% less energy in overall to process and reformat an image tile of 32x32x16bits when compared to the conventional ASIC approach that uses a banked memory working at 395MHz clock frequency, as highlighted in Figure 5.4.

SAR system-level energy results are directly in-line with parallel-access SM and TM power comparison results. For the two compared SAR systems of LiM based implementation and conventional ASIC implementation, two main logic blocks (i.e. perspective and surface interpolation blocks) are the same. Therefore, the difference between these two systems indicates how the parallel-access memory was implemented, as SM or TM. This demonstrates that by using SM we get significant circuit-level benefits. However, LiM synthesis methodology opens up more opportunities than just replacing traditional memories with smart memories. Dramatic benefits are possible by co-designing the algorithms and hardware.

5.2 Graph Processing Accelerator



LiM based SpGEMM Core
Co-design of algorithm and hardware
3D Stacked LiM based Graph Accelerator
DRAM Dies
LiM Layer
DRAM Dies
Exploit high TSV bandwidth
Graphs
Extracted “knowledge” mapped on to graphs

Figure 5.5. Overview of LiM based graph processing core.

To demonstrate the efficacy of algorithm-hardware co-design that is enabled by our LiM synthesis approach we have implemented a low-power accelerator for Generalized Sparse Matrix Sparse Matrix Multiplication (SpGEMM) that is intended to be used in a 3D IC stack (Figure 5.5). SpGEMM is a core function for accelerating graph problems and is inherently a data-intensive application due to high data-traffic coming from sparse matrix operations.

Graphs are the unified representation of large data structures for modeling networks, data analysis problems, or high-level features of extracted objects in advanced imaging applications. As large graphs are sparse, efficient processing of graphs translates into proper manipulation of sparse matrices [67]. SpGEMM is a core kernel in sparse matrix algorithms such as shortest-path search or graph contraction [68] [69]. However, since sparse matrices have highly unpredictable data access patterns and can be structurally large, SpGEMM operations inevitably cause very high and unpredictable data traffic, leading to serious energy and performance related issues. One way to reduce the data traffic in SpGEMM operations is by using column-by-column multiplication [70], whereby only non-zero elements at the intersections are accessed and processed. Conventional ways to implement this algorithm is a heap based design (priority queue) for computing the columns by using multi-way merging [70], that can be built by first-in first-out (FIFO) based SRAMs. FIFO SRAMs cause latency problems, however, due to sequential read/write operations for shifting, which ultimately wastes energy.

To improve the column-by-column algorithm for SpGEMM, Zhu et al. explored the data storage and access patterns in [19] [43] and showed that the SpGEMM operations can be effectively mapped to LiM based content addressable memory (CAM) blocks. As matrix sparsity requires storing only the non-zero elements that are accompanied by their row and column indices, the single cycle "matching" capability of CAMs facilitates index comparison and alignment. Consider a sparse matrix multiplication $C = A \times B$. In this LiM based algorithm, multiplication is separated into two main parts, forming all the columns of resulting matrix C in parallel and assembling them into C. Non-zero elements of a single column of C are formed by using the proposed CAM based architecture for multi-way merging. Row indices of each non-zero element are stored in a CAM array, and their corresponding values are stored in an SRAM array. By using single-cycle CAM matching for cross-checking the intersection of elements in *A* and *B* columns, “multiply and add” or “new entry” operation is decided and executed. Since this architecture assembles row indices of each *C* column, it is called a “horizontal CAM”. A similar operation is performed for assembling *C* by using a single “vertical CAM,” which activates individual horizontal CAM blocks only if their corresponding column indices are matched.

High-level system simulations in [19] [43] show that such a LiM based CAM-SpGEMM core can be used as a low-power hardware accelerator in 3D IC stacks. Sparse matrices are decomposed into sub-blocks and then mapped to DRAM rows for maximizing off-chip DRAM row buffer hit. By this approach, access patterns are rendered predictable, thereby maximizing bandwidth of through silicon vias (TSV) for the 3D stack. Sub-blocks of source matrices *A* and *B* are stored in the on-chip memory, and the result matrix *C* is overwritten as it is computed.

5.2.1 LiM Based SpGEMM Architecture and Test-chip

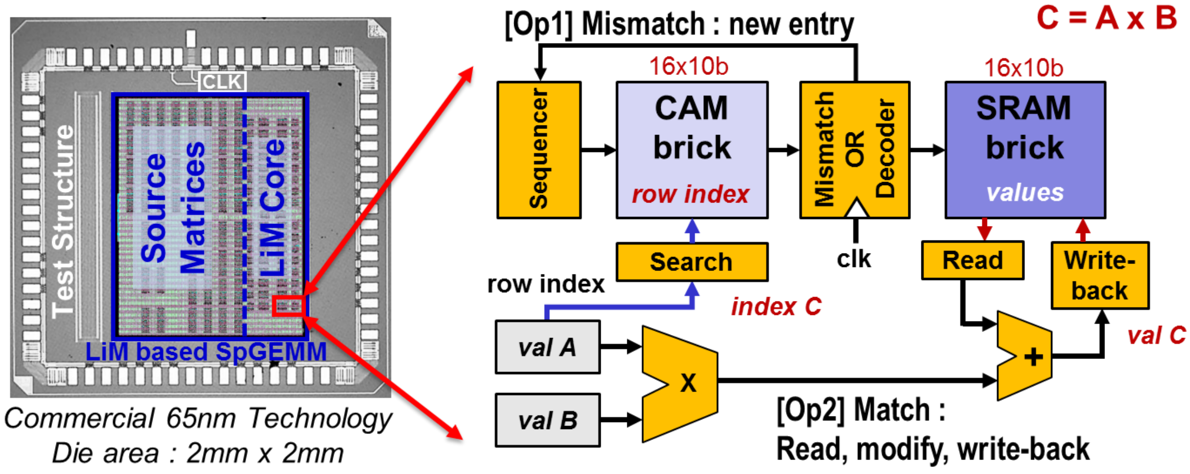


Figure 5.6. LiM based SpGEMM CAM architecture built with distributed SRAM and CAM memory bricks

We synthesized a CAM based SpGEMM chip in a commercial 65nm CMOS technology. CAM bricks are compiled with the same circuit formulation as SRAM bricks. Optimum numbers for tile and array sizes for CAM and SRAM bricks are chosen by sweeping array size parameters in gate-netlist simulations on various SpGEMM benchmarks. As a result of this design-space exploration, row index and data array sizes are chosen as 16x10bits, and column number *N* for sub-blocks is chosen as 32, both consistent with [19]. In contrast to a conventional ASIC flow, since generated CAM and SRAM bricks with such small array size are still very area efficient, the system is synthesized in a fine grained manner from its RTL into 32 horizontal CAMs and 1 vertical CAM with 32 entries.

The CAM-SpGEMM architecture is shown in Figure 5.6, which is basically the hardware realization of the novel LiM based algorithm. The CAM brick holds the row indexes of non-zero elements in the column and the SRAM brick holds their associated data values. For CAM peripherals, a customized mismatch detection block and a sequencer instead of a decoder is built. When there is a mismatch of row indexes, the index of the new data is added to the CAM brick by using the sequencer as the address pointer. When there is a row index match in the CAM brick, the detection block acts as a priority decoder for the SRAM brick. The SRAM brick is designed as a scratch pad with its customized periphery capable of updating or placing new entries. For updating an SRAM entry, a multiply and add block is integrated with a write-back driver. Data is modified (read, modify, write-back) if there is a match, and a new data is added if there is a mismatch. The smart-CAM architecture is distributed throughout the SpGEMM core for 32 columns. The overall SpGEMM core is synthesized without the limitations of black-box memory instances as the LiM synthesis permits fine-grained integration of logic and memory bricks.

The resulting LiM based SpGEMM chip area is 1.3mm^2, with a 0.39mm^2 LiM computation core block. A second chip was implemented based on a standard heap based SpGEMM design for comparison. It consumed 1.24mm2 total area and a 0.33mm2 computation core block. On-chip SRAM blocks for storing source matrices A and B are the same in both chips for a fair comparison. Chip micrograph is shown in Figure 5.6.

5.2.2 SpGEMM Test-chip Results

Both the Heap-based and LiM-based implementations were fully synthesized with our approach and the fabricated chips were fully functional. Test-chip measurement results show that our proposed LiM synthesis methodology provides dramatic system-level benefits for the chosen data-intensive application.

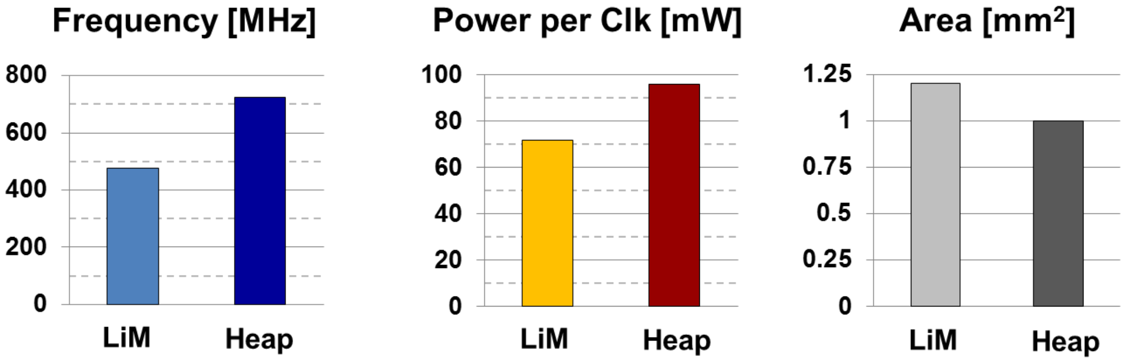
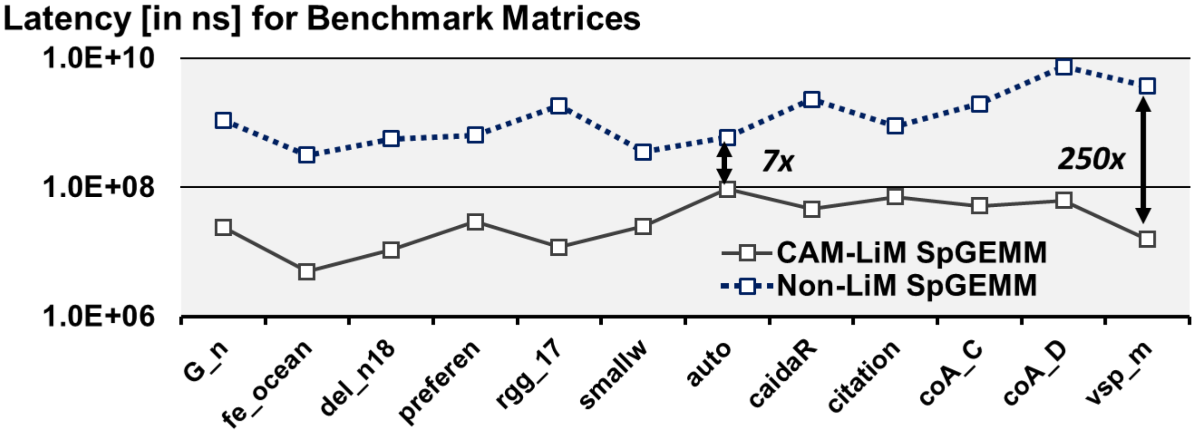


Figure 5.7. SpGEMM chip results for performance comparison of LiM based and Heap based (baseline) implementations

Circuit-level results for both chips are summarized in Figure 5.7. For the same array size of 16x10bits, the CAM brick area is 83% larger than SRAM brick area, and 26% slower. A single read for the SRAM brick consumes 0.73mW power, whereas it is 0.87mW for read and 1.94mW for matching for a CAM brick (at 0.8GHz clock). As a result of these circuit-level differences and added customizations in the CAM based architecture, silicon measurements show that the maximum frequency for LiM based SpGEMM chip is 475MHz, whereas it is 725MHz for the non-LiM SpGEMM chip. Furthermore, the LiM computation core block consumes 20% more area. At the system level we measured the maximum frequencies of the two designs for a nominal Vdd of 1.2V at room temperature. At their respective maximum frequencies, the LiM chip consumes 72mW per clock while the non-LiM based chip consumes 96mW per clock (for both chips, averaged out of multiple test vectors).

When these numbers are back-annotated for benchmark sparse matrix operations that are taken from University of Florida sparse matrix collection [71], the LiM based chip offers dramatic energy and performance benefits. Overall latency and energy results for completing SpGEMM benchmarks are summarized in Figure 5.8. Although the maximum frequency of the LiM chip is 35% slower than the non-LiM chip, the completion time of benchmarks are 7x to 250x faster for LiM chip. Moreover, LiM chip consumes 10x to 310x less energy overall. Utilization of single-cycle CAM matching for multi-way merging drastically reduces the completion time and energy for LiM based SpGEMM chip. Whereas, re-arrangement of FIFO based SRAM arrays at every column computation causes long latency in overall completion time, and higher energy consumption for non-LiM SpGEMM chip.



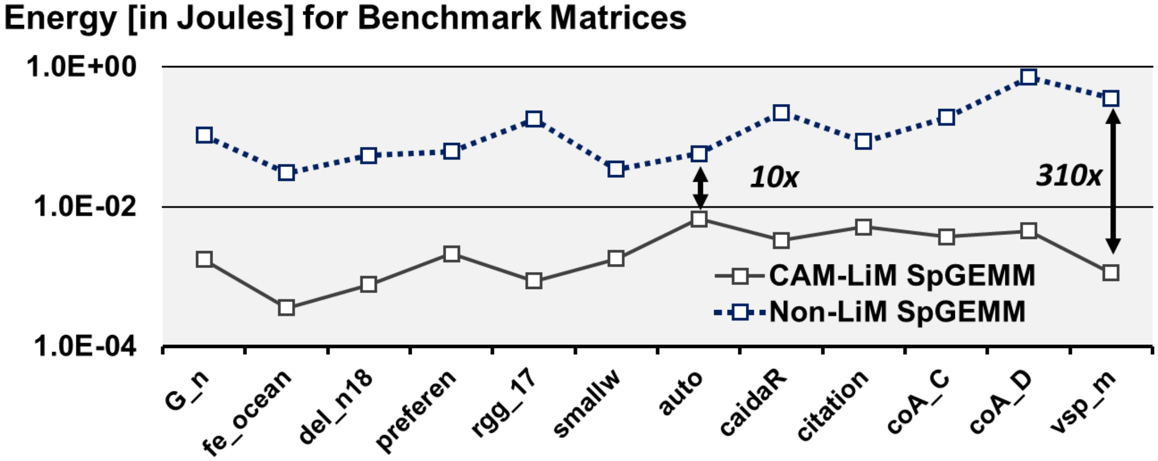


Figure 5.8. Silicon results of latency and energy for LiM based CAM-SpGEMM and standard non-LiM SpGEMM chips.

5.3 Analysis of the Results

Our silicon results show that superior energy and performance results of the LiM based architecture comes from the co-design of algorithm and hardware. The LiM based SAR design gives significant energy benefits that are coming from a synthesized smart-memory, whereas LiM based SpGEMM gives dramatic system-level benefits coming from embedding the application specific knowledge into the overall hardware. We have further shown that LiM synthesis flow provides area efficient integration of fine-grained memory arrays based on white-box primitives of logic and memory bricks. Such logic and memory granularity would be impractical and inefficient with a traditionally compiled embedded memory block approach. Flat synthesis of LiM designs can provide even more area savings when compared to the approach with traditionally compiled memory blocks.

Chapter 3 and Chapter 4 show how the automated memory-brick generation enables affordable synthesis of LiM systems. Chapter 5 demonstrates that LiM synthesis enables fine-grained distribution of bitcell arrays into logic permits co-optimization of application-level knowledge and system design without the black-box limitations of a compiled memory. Another major benefit that LiM synthesis methodology enables is rapid design-space exploration whereby the LiM blocks can be finely optimized and tuned without an extra design cost. A design-space exploration case study on system-level is discussed in detail in Chapter 6.

6 Design-Space Exploration

Our LiM synthesis methodology enables exploration of the design-space with high accuracy to better tune the overall system performance. Using this methodology, the user can narrow down possible floor planning options, explore the trade-offs between energy, delay, and area coming from different memory configurations, and fine tune application-specific "knobs" such as memory partitioning, band-width, number of distributed computational blocks, or number of through silicon vias (TSV).

To demonstrate how automated LiM synthesis flow facilitates design-space exploration, we look at Sparse-Matrix Vector (SpMV) multiplication application as a case study. Similar to SpGEMM application that is discussed in the previous chapter, SpMV is a data-intensive and band-width bound problem on traditional architectures. Combining 3D integration with a low power and fast processing core, this problem can be transformed into a compute bound problem. An application-specific LiM core can efficiently handle the sparse data structure of SpMV and provide fast and energy efficient computation. To accelerate SpMV kernel, a LiM based smart streaming-buffer is designed as a logic layer in a 3D IC stack. Tuning various co-dependent algorithm and hardware parameters for such a large system, however, could impact the design turn-around time. In this section, we will show how LiM synthesis based design-space exploration enables the user to co-optimize the algorithm, architecture, and the LiM based hardware.

The algorithm-level novelties and the finalized implementation results of the overall LiM based SpMV system is beyond the scope of this thesis. The LiM based SpMV architecture is discussed as a case study and the core focus of this chapter is LiM synthesis enabling a design-space exploration flow.

The LiM based 3D architecture for the SpMV system is proposed by Fazle Sadi, Larry Pileggi, and Franz Franchetti. All the algorithm, design, and architecture related details and implementation results of the proposed SpMV system can be found in the accompanying work and doctoral thesis of F. Sadi.

6.1 Spares-Matrix – Vector Multiplication (SpMV)

SpMV is a core kernel that is widely used in various data intensive and scientific applications such as graph algorithms, iterative linear equation solvers, finite element analysis, and analysis of various networks [68]. SpMV algorithms in current architectures, however, are heavily memory bound problems. SpMV applications have very low ratio of computation to memory accesses, leading to very low utilization fraction of the peak performance of the processor (going as low as 10%) [72]. The reason for this low performance is two-fold. For a given matrix-vector multiplication, temporal locality of the elements in a matrix is very low as they are only needed once and not re-used. As a result, conventional memory hierarchy and caching mechanisms cannot be efficiently used. Furthermore, when the source matrix is sparse, the spatial locality of the matrix elements also decreases as sparse matrices are structurally large and non-predictable. With low temporal and spatial locality, SpMV kernel operations cause high and wasted data transfer within the memory sub-system of traditional architectures.

There is existing work on accelerating SpMV applications on CPU [73] and GPU [74] platforms, however, they all suffer from the memory-bound performance and energy issues when the problem size is big enough [75]. A LiM based design, on the other hand, is an ideal approach for such a data-intensive problem with its highly customizable and finely distributed memory instances. Similar to LiM based SpGEMM approach (Chapter 5), a LiM based accelerator layer in a 3D IC stack can exploit the high TSV band-width of the 3D DRAM that is crucial for a memory-bound problem. Unlike SpGEMM, however, there is a dense vector with random access to a large data space, which can add more complexity to both the algorithm and hardware.

6.1.1 LiM Based SpMV Architecture

To alleviate the memory-bound SpMV problem, we implement a LiM based smart streaming buffer, or a sequential access memory, to continuously feed in sparse matrix data to the SpMV core on the logic accelerator layer. The conceptual 3D architecture is depicted in Figure 6.1. An eDRAM cache layer with random access capability fetches and holds the vector data. By modifying the algorithm accordingly for this architecture and LiM based approach, all the accesses to the 3D DRAM that holds the sparse matrix data are rendered sequential and predictable. With this approach, peak DRAM performance can be now achieved with maximum row buffer hits by utilizing the high band-width provided by fast TSV buses to the DRAM layers. As a result, memory bound problem of SpMV algorithm can be transformed into a compute bound problem with high computation to memory access ratios.

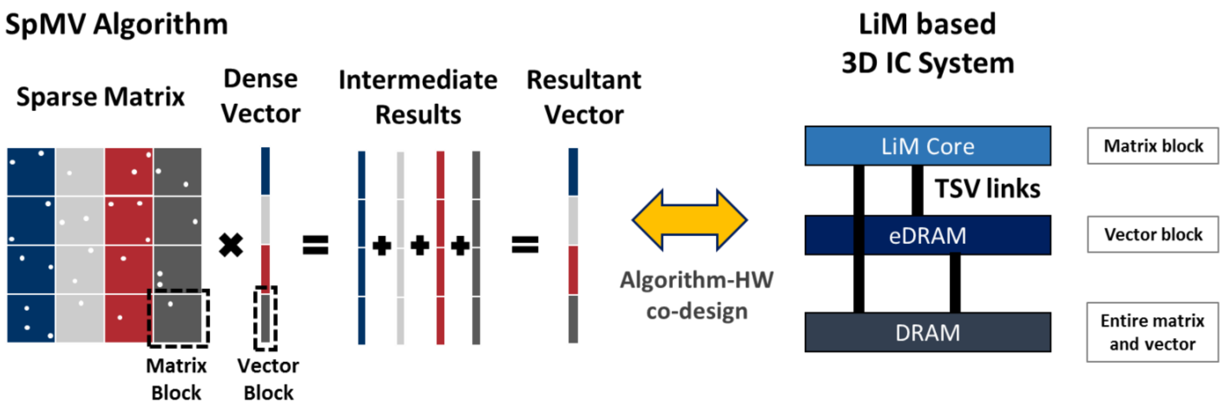


Figure 6.1. Algorithm-hardware co-design of LiM based SpMV system on a 3D IC stack

Consider the sparse matrix - dense vector multiplication of “$y = Ax + y$” where A is sparse matrix and y is the dense vector. The vector is divided into segments (horizontal), and the source matrix is divided into equal number of vertical stripes accordingly, whereby the multiplication is done in a segmented way (Figure 6.1). Within the 3D stack, DRAM layers hold the source matrix and the dense vector. An eDRAM layer acts as a scratch pad with fast random access to hold a given segment of the vector. A LiM based accelerator layer is connected to both eDRAM and DRAM layers through TSVs. The eDRAM feeds the dense vector segment to the LiM layer with random access capability.

The LiM layer is designed as a smart stream buffer that fetches a block of the source matrix from the corresponding vertical matrix stripe. It computes the multiplication of the dense vector segment and the matrix block, where the source matrix elements are only needed once. With this approach, all the DRAM accesses are rendered sequential and predictable although the matrix itself is sparse and large, thereby enabling prefetching matrix blocks intelligently and hiding the fetch delay behind the ongoing multiplication. To better utilize the DRAM bandwidth, sparse matrix elements are stored in CSR format accompanied by their row and column indices.

By co-designing the algorithm and hardware, a novel compressed meta-data that accompanies each element is created to allow pointer based accesses and element tracking. For this implementation, elements are chosen to be stored in 32 bits single-precision floating point format. Resulting compressed meta-data for 32bits is 6bits for row and column indexing each, making the stored data to be 44bits. As the automated brick generation permits any unconventional bit-lengths, this algorithm choice does not present any “wasted memory bit” problems as it would in conventionally compiled SRAMs due to power of two bit-lengths.

6.1.2 Design-Space of the LiM Based SpMV Architecture

Finding the optimum ratio of the matrix block size for the minimum data transfer given the storage size requires exploration of the design-space. Efficient utilization of the available high bandwidth further requires careful tuning of the hardware. Different design-points such as floor planning, memory partitioning, architectural choices, TSV distribution and access can all lead to different utilization rates of the available bandwidth.

For the LiM based SpMV case study, we have defined several application-specific design parameters:

1. <u>Memory size for the smart stream buffers</u>

   Size of the fetched source matrix block from the DRAM depends on the memory size that is chosen for the stream buffer. The decision for the memory size, however, can be a choice in the algorithm, or it can be dictated by the physical area limitations of the LiM layer.

2. Aspect ratio of the distributed memory instances

TSVs are allowed to be placed on a certain pitch in a 3D or 2.5 IC stack. Arbitrary aspect ratios or large memory areas can overlap with TSV pitch, decreasing the number of possible TSVs used and impacting the overall bandwidth. Furthermore, these technology and PDK related information are generally not readily available before starting the actual fabrication process and signing the agreements with the foundry.

3. Memory partitioning and banking within the stream buffer

Different partitioning, floor planning, and parallelism of memory configurations result in different frequency, energy, area, and aspect ratios for the streaming memory. Stream buffer frequency can directly affect the overall performance of the LiM layer if it becomes the critical path. Energy of the stream buffer affects the energy efficiency and MIPS/W of the system. Memory area and aspect ratios affect physical contact points of TSVs, and thus, the available band-width.

4. Single-port vs. multi-port memory instances

Modified SpMV algorithm requires performing read and write-back operations in a single cycle as modified elements may be needed in the consecutive cycle. Performing read and write-back operations in two cycles directly impacts the overall latency. As a result, using a multi-port or single-port memory affects the overall latency and energy-delay product.

As there are multiple co-dependent parameters that can impact the overall system performance, defining a problem size and hardware specifications for this case is not straightforward. Finding an optimum point for many interacting parameters necessitates a design-space exploration. Other than the parameters mentioned previously, there are also number of algorithm related parameters such as the number of parallel processing cores for SpMV computations, storage and compression formats for the elements, and the resulting meta-data size. For this case study, these algorithm-dependent parameters are assumed to be set beforehand.

6.2 Design-Space Exploration Flow

The automated brick generation flow that was presented in Chapter 4 enables a parameterized brick library. By further parameterizing the generation of the RTL of the system, a design space is generated by synthesizing the LiM blocks in an automated fashion. As it is demonstrated in Chapter 4.3, LiM synthesis simulations are efficiently accurate when compared to actual silicon data. By analyzing the impact of different system and memory configurations accurately and affordably, the user can now better optimize the overall system.

Certain variables in HDL coding are already permitted to be parameterized such as bus widths or bit-width going into an instance. Hardware related parameterization such as number of instantiated modules depending on a case, however, are not permitted and synthesizable. To be able to parameterize different memory and system configurations, hardware parameterization is also needed for the design-space exploration. This is achieved by using scripting languages to automate the generation of the HDL codes. A main script that can call other scripts, programs, and configure their input/outputs is created. This script can be in any type or coding language.

The smart stream buffer can be configured with four different parameters:

- Type of the SRAM memory (8T bitcell vs. 6T bitcell)
- Parallel partitions (distributing total bit numbers into parallel partitions)
- Number of bricks in each partition (stack number per bank)
- Brick array word and bit sizes

The main script then calls the brick library generator in a loop to generate all of the meaningful combinations of these four parameters, as shown in Figure 6.2. It is assumed that Synopsys DC is used for the logical synthesis in the following flow.

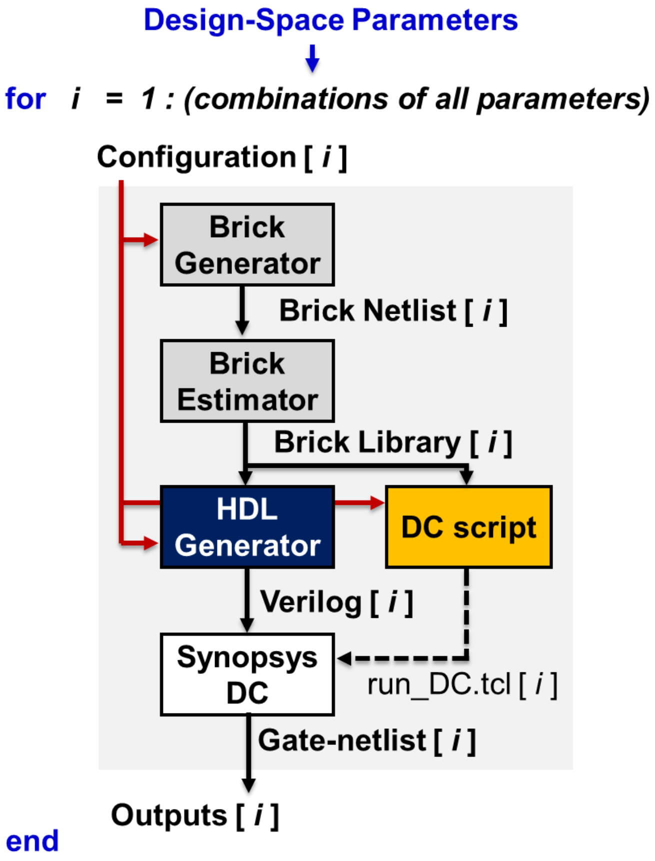


Figure 6.2. Tool flow example for generating a design-space

Within the loop of the main script, the brick generator is called the first to generate an optimally sized memory brick for the memory configuration. For the i^{th} run of the loop, the newly created brick is denoted as *brick[i]*. Then the performance estimator tool is called to generate the energy, performance, and area of *brick[i]*. At the end of the brick generation, a brick synthesis library *lib[i]* is dynamically populated; ready for the memory configuration *[i]*.

After the brick library is generated, a separate script (Python, Perl, Tcl, etc.) that generates Verilog codes for each memory configuration is called by the main script. Using *lib[i]* and memory configuration *[i]*, RTL for the i^{th} system configuration is generated. The HDL generation script modifies an already written and synthesizable Verilog code structure depending on the requirements of the configurations. In the same manner, a second script generates all the corresponding DC .tcl scripts that are needed to run the logic synthesis on the RTLs. These scripts essentially use the critical path of the bricks as CLK period, and feeds in the corresponding brick structures. It also calls the corresponding RTLs and brick library files that were generated within the same iteration.

At the end of the loop, main script runs the generated DC script and collects the simulation outputs. As all the necessary libraries and Verilog files were generated during the same iteration, there is no need for manual intervention. The main script checks whether clock period is met or not for the current design, and re-synthesizes the design with a relaxed clock period if it fails.

After all the different designs are synthesized with their unique system and memory configurations, their corresponding performance results are collected and compiled into a table. With this approach, any synthesis related result that is desired to be analyzed can be collected at the end of the synthesis of each design point iteration. These results can be (and not limited to) energy, frequency, and area numbers, or more detailed results such as clock network power, dynamic and static power consumptions, or physical characteristics such as gate counts. Application-specific metrics such as total band-width, pre-defined figure-of-merits, etc., can also be collected in the same manner. To generate these application-specific numbers, however, the user has to add several extra design-specific functions to the main script loop. At the end of the run, the results of each design-point iteration spans to a design-space.

Enabled by the automated LiM synthesis methodology, design-space exploration flow provides an early feedback mechanism in terms of optimizing the floor plan and memory configurations. The user can now narrow down possible system configurations affordably with good enough accuracy for the comparisons. By setting minimum and maximum targets for any of the design parameters, the system can be optimized even before going into physical synthesis phase. For instance, by setting a minimum frequency, maximum energy and area, or minimum band-width, low performing floor plans and system configurations can be eliminated. By further analyzing how the overall design behaves under different system, memory, and floor planning configurations, the user can start the physical synthesis process better informed on the optimum design choices for desired target metrics. For example, by looking at the designs that give the best and worst results for a chosen specification, the user can gain valuable understanding on the system-level impact of any of the design choices. This feedback mechanism is tested on the SpMV system in the next section.

6.3 Design-Space Exploration on SpMV Architecture

6.3.1 Creating a Design-Space

Using the flow that was described in the previous section, we explored the design-space of smart stream buffers for the SpMV system. Source matrix storage format is CSR, stream in/out data is 44bits (32bit single-precision and 12 bit meta-data), and the system is built with a single SpMV core. As LiM permits fine-grained distribution of memory instances, SpMV multiplication core is integrated with the smart-stream buffer. Variable design parameters for the exploration run are as follows:

- Matrix block size to fetch from DRAM: 32x44bits, 64x44bits, 128x44bits
- Partitioning and floor planning of the stream buffer: Single, 2x, and 4x
- Number of bricks stacked per bank: 1x, 2x, 4x, 8x.
- Memory types: 1R1W 8T brick, R-before-W 6T brick, 1R1W 6T brick, 6T brick

Matrix block-size parameter also defines the memory size of the stream buffer. Row numbers are chosen as power of two numbers for optimum decoder organization. Bricks for the single monolithic memory are 44bits long, whereas 2x parallel partitions are 22bits long and 4x parallel partitions are 11bits long distributed to each partition. 8T memory brick is inherently 1R1W capable. Single cycle read-before-write (R-W) and 1R1W capabilities for 6T brick is handled by area windowing scheme at the RTL (Chapter 3.3). For the conventional single port 6T brick, read and write operations are done in two consecutive cycles thereby doubling the overall latency of SpMV multiplication. As a result of these chosen variables, there are 81 resulting different combinations of possible configurations for the smart stream buffer for this exploration run.

6.3.2 Design-Space Exploration

By using our design-space creation flow, frequency, energy, total area, aspect ratio of partitions, and energy-delay product are collected for every configuration. Isolated results for stream buffers are summarized in Figure 6.3 to Figure 6.7 for a better analysis of memory configuration.

Naming convention for the results are {memory type | configuration: # of partitions, # of stacked bricks | memory size}. For memory types, 1R1W refers to 8T bitcell based 1R1W capable memory brick, R-W refers to 6T bitcell based memory brick with area windowing for read-before-write operation, 1R1W_6T refers to 6T bitcell based memory brick with area windowing for 1R1W operation (with extra penalty on its critical path), and SP refers to 6T bitcell based memory brick with conventional single port operation. For instance, “1R1W c1.1 32x44b” refers to the memory configuration of 8T bitcell based 1R1W memory.

Results are further clustered into groups of three in every plot for space considerations. There are 27 labels on the x-axis to represent the total of 81 configurations. Each label refers to three bars for 1x, 2x, and 4x parallel partitions from left to right respectively. For instance for the label “1R1W c1.1 32x44b”, left bar is representing the single partition configuration (c.1.1), middle bar is representing the configuration of 2x parallel partitions (c.2.1), and the right bar is for the configuration of 4x parallel partitions (c.4.1).

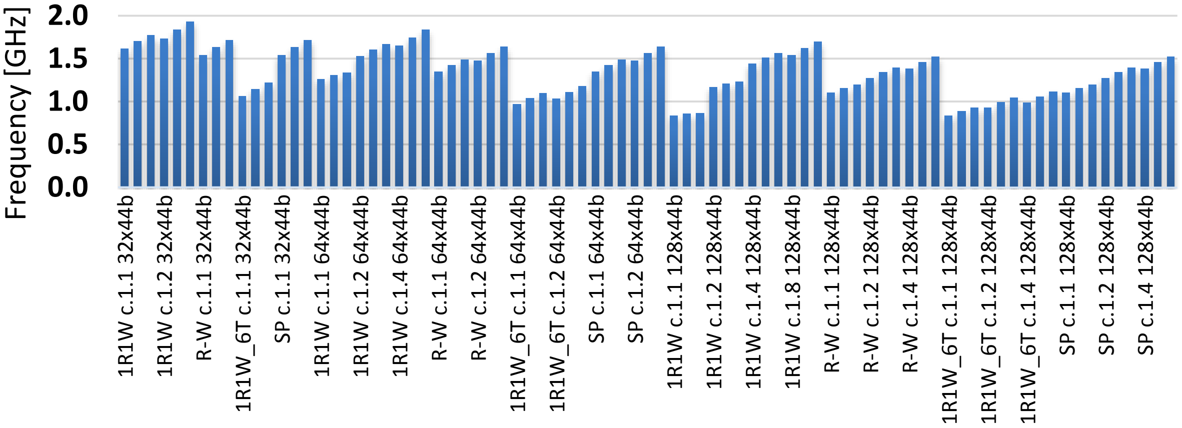


Figure 6.3. Design-space for frequency

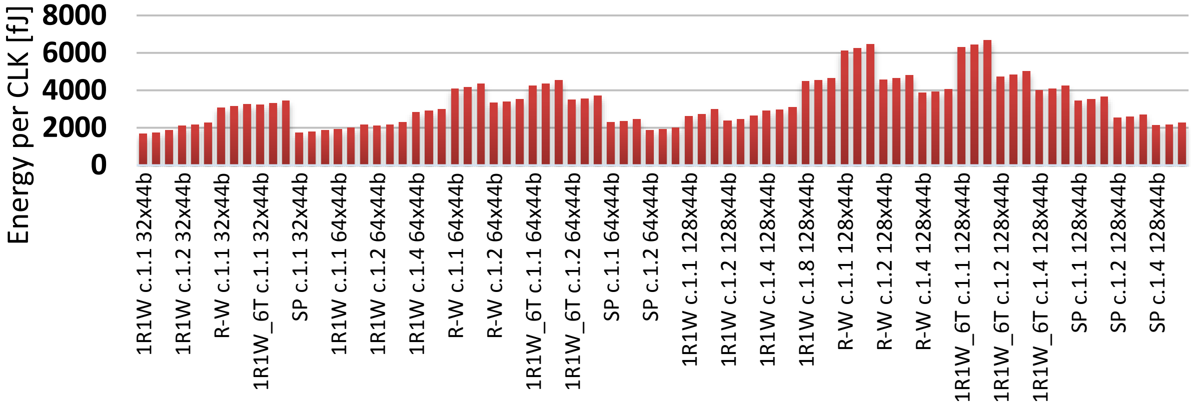


Figure 6.4. Design-space for energy

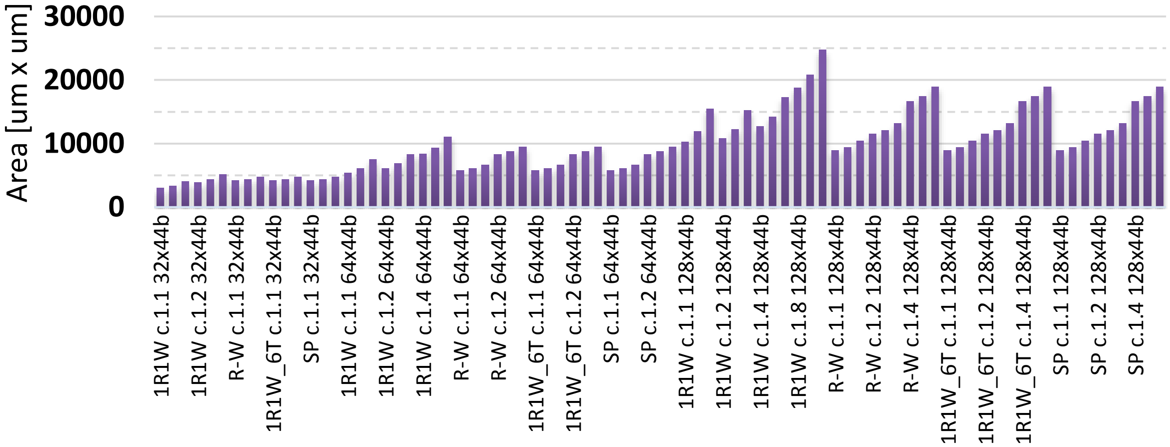


Figure 6.5. Design-space for area

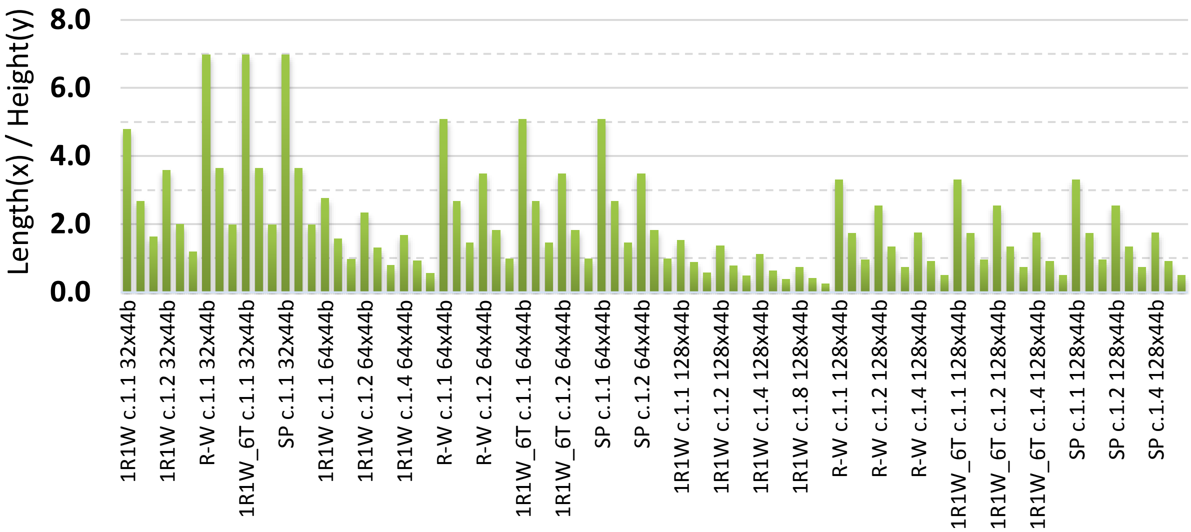


Figure 6.6. Design-space for aspect-ratios.

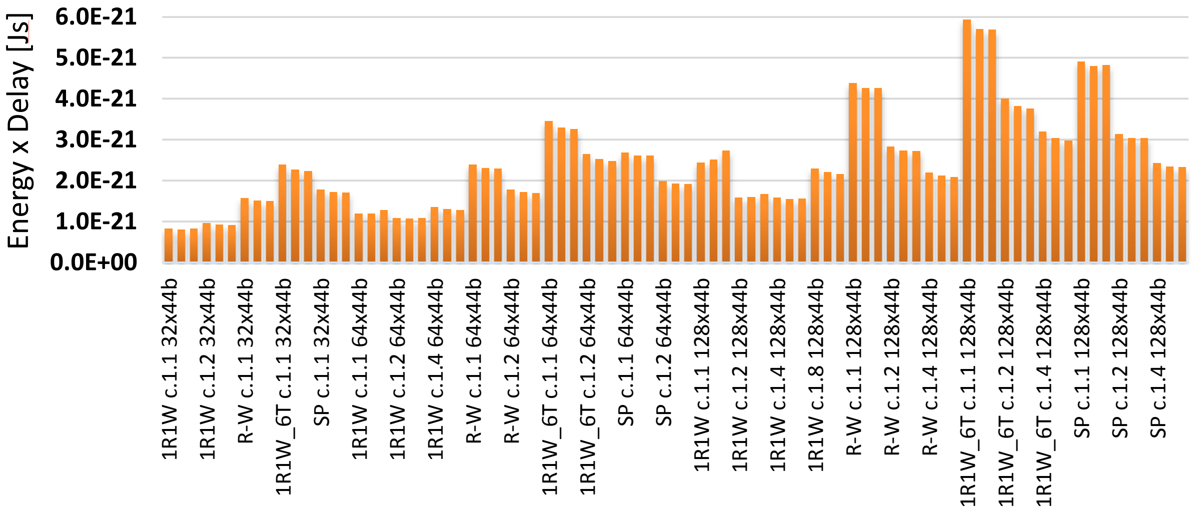


Figure 6.7. Design-space for energy-delay product.

As expected, LiM synthesis provides significant system customization benefits when compared to conventional synthesis approach, as memory arrays with such small sizes are not possible to implement by using a traditional SRAM compiler. The main interest in this chapter, however, is to perform design-space exploration to analyze the impact of different system configurations and getting an early feedback from the tool.

6.3.3 Analysis of the Design-Space

To better analyze the design-space, a combination of different variables should be cross-examined. By defining minimum and maximum target specifications, the user can narrow down the design-points even before moving in to physical synthesis. To narrow down the resulting design-space, we created an example case. Assume that minimum FLOPS/W metric expected from the overall SpMV system, the TSV pitch given from the foundry, maximum DRAM band-width, and available silicon area allocated for the LiM accelerator layer co-dictate the following target specifications for the stream-buffer:

1. Memory size: 64x44bits
2. Maximum Energy x Delay product: 2.5 x 10^{-21} J.s
3. Maximum Area: 7000 um2
4. Maximum aspect ratio of a partition for the memory size 64x44bits (in X:Y) $\leq$ 2:1

These target specifications are then set as minimum / maximum filters for the design-space to eliminate any configuration that fails any of them. Visualization of the target specifications as horizontal bars on the 64x44bit memory size configurations are shown in the following plots (Figure 6.8 to Figure 6.10). Any configuration that passes the maximum allowed target specification are eliminated as a floorplan and memory type choice.

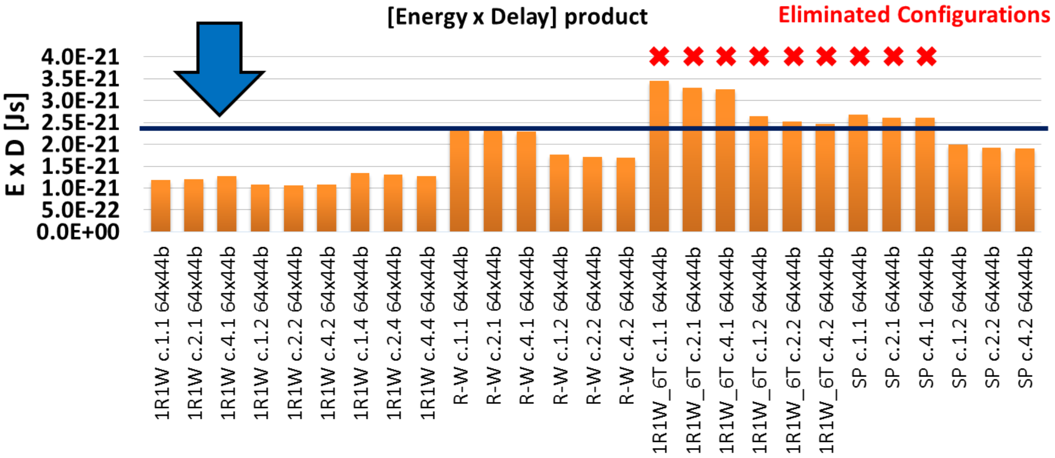


Figure 6.8. Design-space for energy-delay under specific target metrics.

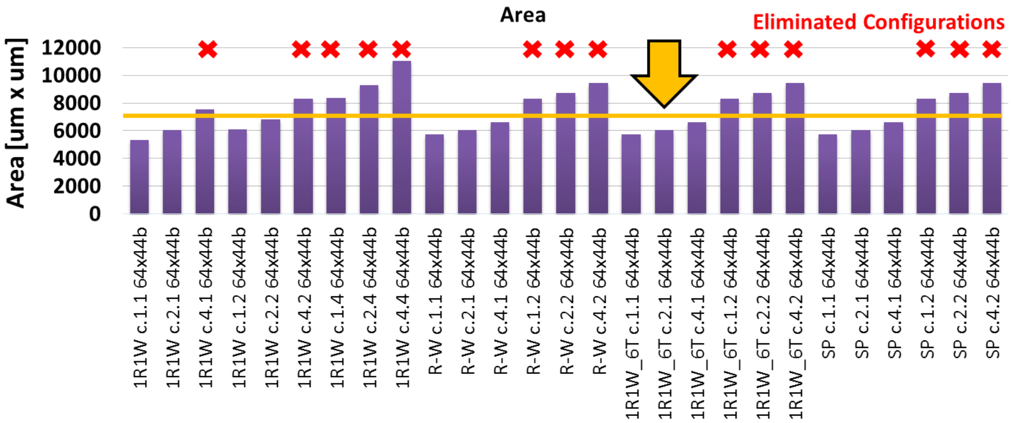


Figure 6.9. Design-space for area under specific target metrics.

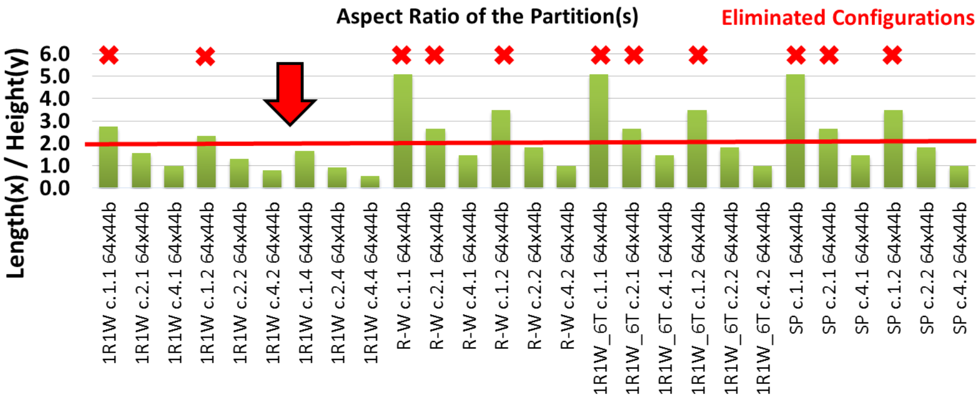


Figure 6.10. Design-space for aspect-ratio of partitions under specific target metrics.

As a result of the parameters set by this example case, possible floor-planning and memory type configurations for the 64x44bit stream buffer are narrowed down to the configurations that are summarized in Table 6.1. The same filtering procedure can be done to obtain the best in class configurations as well. For 64x44bit memory size, feedback from the design-space exploration flow suggests that using an 8T bitcell type 1R1W memory with 2 partitions with either single or two stacked bricks per partition, or a 6T bitcell type Read-before-Write memory with 4 partitions with single brick in the stack will meet the target specifications set by the user.

Configuration	*Memory Type*	*Partitioning*	*Bricks stacked per partition*
1R1W c.2.1 64x44b	8T	2x Partitions (22 bits)	1x Brick (64x22bit)
1R1W c.2.2 64x44b	8T	2x Partitions (22 bits)	2x Bricks (32x22bit)
R-W c.4.1 64x44b	6T	4x Partitions (11 bits)	1x Brick (64x11bit)

Table 6.1. Smart-stream buffer configurations meeting the target metrics.

We can gather further information from the design-space exploration data on how the system behaves when certain parameters are changed. For instance increasing brick number per stack to four bricks for the 1R1W 8T memory type does not impact the energy-delay product much, but can cause problem with the maximum area limitations. On the other hand for R-W 6T memory type, decreasing stacked brick number from four violates the target aspect ratio specification.

This case study demonstrates that our automated LiM synthesis methodology enables detailed exploration of the design-space without any extra full-custom extra design cost. Moreover, the tool itself gives crucial feedback to the user in terms of floorplan and memory configuration assistance. Using the design-space exploration flow as a tool, the user can now optimize the system better. With all simulation results, assistance in floorplan and memory configurations, and a better grasp of the system knobs, the user can move on to the physical synthesis with reference system configurations.

7 Scaling, Future Work, and Conclusions

In the final chapter of this thesis, applicability of LiM synthesis methodology to future technology nodes and SoC trends is discussed. As this dissertation attempts to implement a methodology rather than a specific accelerator design depending on a technology, device, or architecture, we can argue that the LiM synthesis flow remains applicable as a tool to localize computation for memory-intensive applications in foreseeable future IC trends. Possible future work for the implemented methodology and final conclusions drawn out from the dissertation are also highlighted at the end of this chapter.

7.1 Technology Scaling and Future Trends

The work implemented in this dissertation aims to formulate a methodology and not a technology or architecture dependent design. Therefore, we can argue that the applicability of this work holds true as long as the design intent is building an in-memory processing block.

7.1.1 Technology Scaling Below 14nm Node

Layout patterns are getting more restricted due to manufacturing limitations as technology scales beyond 14nm node. Since the restricted pattern constructs are the enabling technology for LiM synthesis, there is no foreseeable obstacles that would cause LiM synthesis approach to fail due to scaling. To validate a synthesized LiM block on silicon at a deeply scaled technology node, a synthesized smart parallel-access SRAM is demonstrated at a 14nm process from IBM in [36]. A traditional compiled-like SRAM with parallel accessible banks is implemented at the same node as a baseline for comparison. The results validate the hypothesis of restricted constructs enabling tight logic and memory integration (as it is shown in Chapter 2.2), and further verifies the significant circuit-level benefits coming from the smart-memory implementation (in accordance to the results demonstrated in Chapter 5.1). Moreover, the devices in the 14nm process that are used in [36] are fin-type field effect transistors (FinFETs), which shows that the synthesis of smart-memories is applicable to any device type that will be used down the scaling path of digital IC industry.

Restricted patterns are a necessity coming from the limitations of utilizing 193nm wavelength immersion lithography in modern IC manufacturing. A cost-efficient breakthrough in extreme UV wavelength or electron-beam lithography technologies may relax the restrictions imposed on future layout patterns. Benefits of fine-grained integration of logic and memory, however, is independent of the layout patterns or the technology node. As LiM synthesis methodology enables co-design of algorithm and hardware and detailed design-space exploration capabilities, the work presented in this dissertation would be still applicable to next generation technologies.

7.1.2 Future SoC Trends

“Big data” applications

As the amount of data to capture, store, process, and send is growing exponentially in the upcoming era of "big data" applications [76], on-chip localization of computation, data-intensive processing methods, and minimizing the data-transfer bandwidth of chip to chip/server communications will be even more crucial. Customized smart-memories are built for accelerating specific application-domains in the LiM synthesis paradigm, and it is an efficient design approach to transform memory-bound problems into compute-bound ones. In the big data applications era, LiM synthesis methodology would arguably provide designers an affordable design approach for low-power application acceleration.

"Dark silicon" era

Clock and power gating techniques, wherein the clock signal or the supply voltage are turned off for an inactive block to save power, are highly used circuit techniques to meet the tight power budget of an SoC [2]. As technology scales down to sub-10nm regions and the number of cores on a single die piles up, however, it is predicted that more than half of the chip will be required to be switched off at a given time to meet the ever-tightening power budgets (commonly known as "dark silicon" [77]).

To address this challenge while making a better use of the chip real-estate for optimum performance and power efficiency, [78] proposes to embed heterogeneous hardware accelerators on the chip for various tasks instead of copying more cores that will eventually stay idle. It is further shown in [78] that a diverse class of accelerators all contain significant amounts of memory (reported up to 90%). As this dissertation demonstrates that LiM synthesis methodology enables affordable implementation of finely tailored low-power accelerators for various data-intensive applications, we can argue that LiM synthesis methodology will remain to be an efficient and affordable tool in the dark silicon era of IC industry.

Advent of 3D Integration

As the 3D die stacking is emerging to become a commercially available technology such as the examples of hybrid-memory cube (HMC) [29], in-memory processing designs are attracting more interest as they present simple and energy-efficient acceleration solutions for heavy data processing [79] [17]. It is already demonstrated in system-level simulations that LiM based logic layers in 3D IC stacks are ideal platforms for energy efficient performance enhancement in [43] [19] [40]. It is also interesting to note that 3D stacked dies can be heterogeneous with different technology nodes and/or different memory types (SRAM, DRAM, eDRAM, STT-based). LiM based accelerator layer(s) can still be used without any integration issues as long as the data signals reach to the LiM die layer(s) through TSVs.

7.2 Future Work for the Synthesis Methodology

This dissertation opens up many interesting opportunities and use cases for the implemented synthesis methodology. Here we discuss three possible future directions for this work, but many more interesting applications may emerge in the future.

a) Circuit level: DRAM based LiMs

This dissertation does not address building memory bricks based on DRAMs or embedded DRAMs (eDRAM). DRAM-based synthesized LiM blocks may open up brand new application domains and architectures as DRAM characteristics are different than SRAM characteristics. Since DRAMs are heavily used as off-chip memory, specialized computation can be then off-loaded to the off-chip memory as well to create smart off-chip memories. As eDRAMs offer more density with a comparable performance to on-chip SRAMs, eDRAM based LiM blocks may also present interesting alternatives in the design-space. DRAM technology, however, requires more constraints on the circuit design and it is harder to implement and verify DRAM based LiM blocks on silicon. Moreover, periodical refresh requirements of DRAMs may present challenges on the brick design from an architectural stand-point.

b) Tool & synthesis level: Bricks as standard cells

Memory bricks are currently integrated as macro cells in the physical synthesis flow, and an existing brick library is needed before starting the synthesis. Therefore synthesis tools such as Synopsys DC, ICC, or Cadence Encounter do not have the ability to improve the design by compiling the bricks on-the-fly. One future work for this methodology is to enhance the design flexibility by allowing the selection of memory bricks to be optimized like standard cells. With such an approach, the synthesis tools could choose and optimize the array size and placement of the memory bricks in a standard cell like manner. The LiM synthesis methodology can be then inserted as a “plug-in” to the commercially available synthesis tools. With this approach, current design-space exploration flow can be further improved to a “LiM based chip generator” whereby different system configurations are optimally implemented by the physical synthesis tools.

c) Application-level: Exploration of different systems and architectures

Existing work on LiM designs are comprised of several data-intensive applications. Now that this dissertation presents a synthesis tool, a wide range of different applications and architectures can be explored quickly and affordably using this tool. A detailed research on benefits and challenges of utilizing LiM designs on a pool of application domains can then give insightful information to IC designers.

7.3 Conclusion

In today's SoCs, it is a necessity to incorporate more memory on chip to overcome the well-known memory-wall and power-wall challenges. As embedded memory consumes more than half of the IC real estate, however, it is essential to integrate application level knowledge into the memory blocks to improve processing efficiency for data intensive applications. Many core architectures in the era of big-data, dark silicon, and 3D die stacking call for even more integration of heterogeneous smart-memory based accelerators on the die. Logic-in-Memory designs are one of the ideal platforms to overcome the multi-faceted challenges of today's and tomorrow's data-intensive applications.

This work aims to formulate a Logic-in-Memory synthesis approach that enables the IC designers to reliably and affordably synthesize application-specific smart-memory blocks with a cell-based design flow. This dissertation demonstrates that the LiM synthesis flow, by construction, is scalable, manufacturable, customizable, verifiable, and provides efficient co-design of algorithm and hardware as demonstrated by the chip implementation. Automated generation of memory-cell instances further enables a rapid system-level design-space exploration flow as a pre-floorplanning system optimization tool.

References

[1] B. Mohammad, Embedded Memory Design for Multi-Core and Systems on Chip, New York: Springer New York, 2014.

[2] N. Weste and D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective (4th edition), Boston: Pearson, 2010.

[3] "International Technology Roadmap for Semiconductors - System Drivers," ITRS, 2011.

[4] E. J. Marinissen, B. Prince, D. Keltel-Schulz, and Y. Zorian, "Challenges in embedded memory design and test," *Design Automation and Test in Europe (DATE),* vol. 2, pp. 722-727, 2005.

[5] "Semico: System(s)-on-a-Chip – A Braver New World.," Semico Research Corp, 24 October 2007. [Online]. Available: http://www.semico.com/content/semico-systems-chip-%E2%80%93-braver-new-world. [Accessed 14 June 2015].

[6] "ISSCC Technology Trends 2015," [Online]. Available: http://isscc.org/doc/2015/isscc2015_trends.pdf. [Accessed 14 July 2015].

[7] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious,"

SIGARCH Comput. Archit. News, vol. 23, no. 1, pp. 20-24, Mar. 1995.

[8] S. Naffziger, "High-Performance Processors in a Power-Limited World," *Symposium on VLSI Circuits, 2006. Digest of Technical Papers,* pp. 93-97, 2006.

[9] F. J. Pollack, "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies," *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture,* pp. 2-., 1999.

[10] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration," *23rd Annual International Symposium on Computer Architecture,* pp. 90-90, 1996.

[11] M. Horowitz and W. Dally, "How Scaling Will Change Processor Architecture," *Digest of Technical Papers. ISSCC. 2004 IEEE International,* vol. 1, pp. 132-133, 2004.

[12] B. S. Amrutur and M. A. Horowitz, "Speed and power scaling of SRAM's," *IEEE Journal of Solid-State Circuits,* vol. 35, no. 2, pp. 175-185, 2000.

[13] R. J. Evans and P. D. Franzon, "Energy consumption modeling and optimization for SRAM's," *IEEE Journal of Solid-State Circuits,* vol. 30, no. 5, pp. 571-579, 1995.

[14] G. Northrop, "Design technology co-optimization in technology definition for 22nm and beyond," *2011 Symposium on VLSI Technology (VLSIT),* pp. 112-113, 2011.

[15] D. Morris, V. Rovner, L. Pileggi, A. Strojwas, and K. Vaidyanathan, "Enabling application-

specific integrated circuits on limited pattern constructs," *Symposium on VLSI Technology (VLSIT),* pp. 139-140, 2010.

[16] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro,* vol. 17, no. 2, pp. 34-44, Mar. 1997.

[17] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," *Proceedings of the 35th Annual International Symposium on Computer Architecture,* pp. 453-464, 2008.

[18] Q. Zhu, "Application Specific Logic-in-Memory," *Ph.D. dissertation*, Dept. Elect. and Comp. Eng., Carnegie Mellon Univ., Pittsburgh PA, 2013.

[19] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing," *IEEE International 3D Systems Integration Conference (3DIC),* pp. 1-7, 2013.

[20] Q. Zhu, K. Vaidyanathan, O. Shacham, M. Horowitz, L. Pileggi, and F. Franchetti, "Design automation framework for application-specific logic-in-memory blocks," *IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP),* p. 125–132, 2012.

[21] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The Architecture of the DIVA Processing-in-memory Chip," *Proceedings of the 16th International Conference on Supercomputing,* pp.

14-25, 2002.

[22] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping Irregular Applications to DIVA, a PIM-based Data-intensive Architecture," *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing,* 1999.

[23] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick, "Hardware/compiler codevelopment for an embedded media processor," *Proceedings of the IEEE,* vol. 89, no. 11, pp. 1694-1709, Nov. 2001.

[24] B. R. Gaeke, P. Husbands, X. S. Li, L. Oliker, K. A. Yelick, and R. Biswas, "Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines," *Proceedings of the 16th International Parallel and Distributed Processing Symposium,* pp. 203-, 2002.

[25] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," *Proceedings of the 25th Annual International Symposium on Computer Architecture,* p. 192–203, 1998.

[26] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational RAM: implementing processors in memory," *IEEE Design Test of Computers,* vol. 16, no. 1, pp. 32-41, Jan. 1999.

[27] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational Ram: A Memory-SIMD Hybrid And Its Application To DSP," *Proceedings of the IEEE 1992 Custom Integrated*

Circuits Conference, pp. 30.6.1-30.6.4, 1992.

[28] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the Terasys massively parallel PIM array," *Computer,* vol. 28, no. 4, pp. 23-31, Apr. 1995.

[29] J. T. Pawlowski, "Hybrid memory cube (HMC)," *Hot Chips,* vol. 23, 2011.

[30] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro,* vol. 34, no. 4, pp. 36-42, Jul. 2014.

[31] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee, "An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," *IEEE 16th International Symposium on High Performance Computer Architecture (HPCA),* pp. 1-12, 2010.

[32] Y. Murachi, T. Kamino, J. Miyakoshi, H. Kawaguchi, and M. Yoshimoto, "A power-efficient SRAM core architecture with segmentation-free and rectangular accessibility for super-parallel video processing," *IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT),* pp. 63-66, 2008.

[33] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," *Proceedings of the 27th Annual International Symposium on Computer Architecture,* pp. 161-171, 2000.

[34] K. Vaidyanathan, R. Liu, L. Liebmann, K. Lai, A. Strojwas, and L. Pileggi, "Rethinking ASIC design with next generation lithography and process integration," *SPIE Advanced*

Lithography, pp. 86840C-86840C, 2013.

[35] K. Vaidyanathan, "Exploiting Challenges of Sub-20 nm CMOS for Affordable Technology Scaling," *Ph.D. dissertation*, Dept. Elect. and Comp. Eng., Carnegie Mellon Univ., Pittsburgh PA, 2013.

[36] K. Vaidyanathan, Q. Zhu, L. Liebmann, K. Lai, S. Wu, R. Liu, Y. Liu, A. Strojwas, and L. Pileggi, "Exploiting sub-20-nm complementary metal-oxide semiconductor technology challenges to design affordable systems-on-chip," *Journal of Micro/Nanolithography, MEMS, and MOEMS,* vol. 14, no. 1, pp. Journal of Micro/Nanolithography, MEMS, and MOEMS, Dec. 2014.

[37] Q. Zhu, E. L. Turnerz, C. R. Bergery, L. Pileggi, and F. Franchetti, "Application-specific logic-in-memory for polar format synthetic aperture radar," *IEEE Conference on High Performance Extreme Computing (HPEC),* vol. 44, 2011.

[38] Q. Zhu, C. R. Berger, E. L. Turner, L. Pileggi, and F. Franchetti, "Polar format synthetic aperture radar in energy efficient application-specific logic-in-memory," *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* p. 1557–1560, 2012.

[39] Q. Zhu, C. R. Berger, E. L. Turner, L. Pileggi, and F. Franchetti, "Local Interpolation-based Polar Format SAR: Algorithm, Hardware Implementation and Design Automation," *Journal of Signal Processing Systems,* vol. 71, no. 3, p. 297–312, 2013.

[40] F. Sadi, B. Akin, D. T. Popovici, J. C. Hoe, L. Pileggi, and F. Franchetti, "Algorithm/hardware co-optimized SAR image reconstruction with 3D-stacked logic in memory," *IEEE High Performance Extreme Computing Conference (HPEC),* pp. 1-6, 2014.

[41] Q. Zhu, L. Pileggi, and F. Franchetti, "A Smart Memory Accelerated Computed Tomography Parallel Backprojection," *VLSI-SoC: From Algorithms to Circuits and System-on-Chip Design,* pp. 21-44, 2013.

[42] Q. Zhu, L. Pileggi, and F. Franchetti, "Cost-effective smart memory implementation for parallel backprojection in computed tomography," *IEEE/IFIP 20th International Conference on VLSI and System-on-Chip (VLSI-SoC),* p. 111–116, 2012.

[43] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," *IEEE High Performance Extreme Computing Conference (HPEC),* pp. 1-6, 2013.

[44] H.E. Sumbul, K. Vaidyanathan, Q. Zhu, F. Franchetti, and L. Pileggi, "A Synthesis Methodology for Application-Specific Logic-in-Memory Designs," *52nd ACM/EDAC/IEEE Design Automation Conference (DAC),* 2015.

[45] D. M. Fried et al., "Aggressively scaled (0.143 mu;m2) 6T-SRAM cell for the 32 nm node and beyond," *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International,* pp. 261-264, 2004.

[46] E. Grossar, M. Stucchi, K. . Maex, and W. Dehaene, "Read Stability and Write-Ability

Analysis of SRAM Cells for Nanometer Technologies," *IEEE Journal of Solid-State Circuits,* vol. 41, no. 11, p. 2577–2588, Nov. 2006.

[47] L. Chang, R. K. Montoye, Y. Nakamura, K. A. Batson, R. J. Eickemeyer, R. H. Dennard, W. Haensch, and D. Jamsek, "An 8T-SRAM for Variability Tolerance and Low-Voltage Operation in High-Performance Caches," *IEEE Journal of Solid-State Circuits,* vol. 43, no. 4, p. 956–963, Apr. 2008.

[48] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey," *IEEE Journal of Solid-State Circuits,* vol. 41, no. 3, p. 712–727, Mar. 2006.

[49] B. S. Amrutur, "Design and analysis of fast low power SRAMs," *Ph.D. Dissertation, Stanford University,* 1999.

[50] B. S. Amrutur and M. A. Horowitz, "A replica technique for wordline and sense control in low-power SRAM's," *IEEE Journal of Solid-State Circuits,* vol. 33, no. 8, p. 1208–1219, Aug. 1998.

[51] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, and T. Nakano, "A 64Kb full CMOS RAM with divided word line structure," *Solid-State Circuits Conference. Digest of Technical Papers. 1983 IEEE International,* p. 58–59, 1983.

[52] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, and T. Nakano, "A divided word-line structure in the static RAM and its application to a 64K

full CMOS RAM," *IEEE Journal of Solid-State Circuits,* vol. 18, no. 5, p. 479–485, Oct. 1983.

[53] T. Hirose, H. Kuriyama, S. Murakami, K. Yuzuriha, T. Mukai, K. Tsutsumi, Y. Nishimura, Y. Kohno, and K. Anami, "A 20 ns 4 Mb CMOS SRAM with hierarchical word decoding architecture," *Solid-State Circuits Conference, 1990. Digest of Technical Papers. 37th ISSCC., 1990 IEEE International,* p. 132–133, 1990.

[54] A. Karandikar and K. K. Parhi, "Low power SRAM design using hierarchical divided bit-line approach," *International Conference on Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings,* p. 82–88, 1998.

[55] K. Osada, H. Higuchi, K. Ishibashi, N. Hashimoto, and K. Shiozawa, "A 2-ns-Access, 285-MHz, Two-Port Cache Macro Using Double Global Bit-Line Pairs," *IEICE TRANSACTIONS on Electronics,* vol. E83–C, no. 1, p. 109–114, Jan. 2000.

[56] S. Hsu, A. Agarwal, M. Anders, H. Kaul, S. Mathew, F. Sheikh, R. Krishnamurthy, and S. Borkar, "A 2.8GHz 128-entry x 152b 3-read/2-write multi-precision floating-point register file and shuffler in 32nm CMOS," *Symposium on VLSI Circuits (VLSIC),* pp. 118-119, 2012.

[57] Y. Xie, "Modeling, Architecture, and Applications for Emerging Memory Technologies," *IEEE Design Test of Computers,* vol. 28, no. 1, pp. 44-51, Jan. 2011.

[58] E. Brunvand, Digital VLSI chip design with Cadence and Synopsys CAD tools, Addison-Wesley, 2010.

[59] I. E. Sutherland, R. F. Sproull, and D. F. Harris, Logical Effort: Designing Fast CMOS Circuits, Morgan Kaufmann, 1999.

[60] I. E. Sutherland and R. F. Sproull, Logical effort: Designing for speed on the back of an envelope, MIT Press, 1991.

[61] T. J. Barnes, "SKILL: a CAD system extension language," *27th ACM/IEEE Design Automation Conference, 1990. Proceedings,* p. 266–271, 1990.

[62] J. Qian, S. Pullela, and L. Pillage, "Modeling the "Effective capacitance" for the RC interconnect of CMOS gates," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 13, no. 12, p. 1526–1535, Dec. 1994.

[63] O. Shacham, "Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms," *Ph.D. Dissertation, Stanford University,* 2011.

[64] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking Digital Design: Why Design Must Change," *IEEE Micro,* vol. 30, no. 6, pp. 9-24, Nov. 2010.

[65] W. G. Carrara, R. M. Majewski, and R. S. Goodman, Spotlight Synthetic Aperture Radar: Signal Processing Algorithms, Boston: Artech Print on Demand, 1995.

[66] D. S. McFarlin, F. Franchetti, M. Püschel, and J. M. F. Moura, "High-performance synthetic aperture radar image formation on commodity multicore architectures," *SPIE Proceedings,*

vol. 7337, p. 733708–733708–12, 2009.

[67] J. R. Gilbert, V. B. Shah, and S. Reinhardt, "A Unified Framework for Numerical and Combinatorial Computing," *Computing in Science Engineering,* vol. 10, no. 2, pp. 20-25, 2008.

[68] J. Kepner and J. Gilbert, Graph Algorithms in the Language of Linear Algebra, Philadelphia: SIAM, 2011.

[69] A. Buluc and J. R. Gilbert, "Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication," *37th International Conference on Parallel Processing (ICPP '08),* p. 503–510, 2008.

[70] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008),* pp. 1-11, 2008.

[71] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software,* vol. 38, no. 1, pp. 1:1-1:25, 2011.

[72] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," *ACM Proceedings of the conference on High Performance Computing Networking, Storage and Analysis,* p. 18, 2009.

[73] J. B. White III and P. Sadayappan, "On improving the performance of sparse matrix-vector

multiplication," *IEEE High-Performance Computing,* pp. 66-71, 1997.

[74] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining," *Proceedings of the VLDB Endowment,* vol. 4, no. 4, pp. 231-242, 2011.

[75] J. D. Davis and E. S. Chung, "Spmv: A memory-bound application on the GPU stuck between a rock and a hard place," *Microsoft Research Silicon Valley, Technical Report,* Sept. 2012.

[76] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Byers, "Big data: The next frontier for innovation, competition, and productivity," May 2011.

[77] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *38th Annual International Symposium on Computer Architecture (ISCA),* pp. 365-376, 2011.

[78] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The Accelerator Store: A Shared Memory Framework for Accelerator-based Systems," *ACM Trans. Archit. Code Optim.,* vol. 8, no. 4, pp. 48:1-48:22, Jan. 2012.

[79] G.H. Loh, N. Jayasena, M.H. Oskin, M. Nutter, D. Roberts, M. Meswani, D.P. Zhang, and M. Ignatowski, "A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM," *Workshop on Near-Data Processing (WoNDP),* 2013.

Appendix A: Memory Brick Generator Codes

A.1 Memory Brick Verilog Module

A template Verilog module code for an 8T bitcell based SRAM memory brick with size 16x10bits is as follows:

```
// Verilog module for 8T bitcell based 1R1W capable SRAM brick of 16x10bits
module   sram_brick_16_10   (CLK, R_EN, DRWL, DWWL, WBL, WBL_B, ARBL);

input CLK, R_EN ;
input [9 : 0] WBL, WBL_B;
input [15: 0] DRWL, DWWL;
output reg [9 : 0] ARBL;

// Create a dummy register array and dummy encoders for Read / Write addr
reg [9 : 0] mem_brick[15 : 0];
wire [3 : 0] read_addr, wr_addr;
wl_encoder   dummy_enc_R  (.in_dwl(DRWL), .out_addr(read_addr)    );
wl_encoder   dummy_enc_W (.in_dwl(WRWL), .out_addr(wr_addr)       );

// Read - Write operations w.r.t. CLK
always @(CLK) begin
        if(CLK == 1) begin

                if(|DWWL)                       // WRITE
                        mem_brick[wr_addr] <= WBL & (~WBL_B);

                if(R_EN)                        // READ
                        ARBL <= mem_brick[read_addr];
                else
                        ARBL <= 10'bz;          // for tri-state behavior
        end
end
endmodule
```

A.2 Memory Brick Synthesis Library file

A template synthesis library file in Liberty format (.lib) for an 8T bitcell based SRAM memory brick with size 16x10bits is as follows:

```
library(sram_brick_16_10_lib_TT) {

        -- Start with default definitions (units, supply V, temperature, corners, etc.)

        lu_table_template(ma_16_10_template) {
                variable_1 : input_net_transition;
                variable_2 : total_output_net_capacitance;
                index_1("10, 20, 30, 40, 50, 60, 70");     // example numbers
        /* in ps - scale in FO4: [1/10 , 1/4 , 1/2 , 1 , 2 , 4 , 10] */
                index_2("10, 20, 30, 40, 50, 60, 70");  }  // example numbers
        /* in fF - scale in "extra" ARBL cap driven: [0x, 1x, 4x, 8x, 9x, 12x, 16x]*/

        type (ma_16_10_ARBL) {
                base_type : array;
                data_type : bit;
                bit_width : 10;
                bit_from  : 9;
                bit_to    : 0;
                downto    : true;      }
        type (ma_16_10_DRWL)     { .. }
        type (ma_16_10_DWWL)     { .. }
        type (ma_16_10_WBL)      { .. }
        type (ma_16_10_WBL_B)    { .. }

        cell(sram_brick_16_10) {
                area : <in um2>
                dont_touch : true ;
                dont_use : true ;
                interface_timing : true ;
                cell_leakage_power : <in uW>

                pg_pin (VSS) {      voltage_name : VSS;
                                    pg_type : primary_ground;     }
                pg_pin (VDD) {      voltage_name : VDD;
                                    pg_type : primary_power;      }
```

```
		bus(ARBL) {
			bus_type : "ma_16_10_ARBL";
			direction : output;
			three_state : "!BLK_RE";
			capacitance : <ARBL pin capacitance in fF>
			max_capacitance : <max output driving cap in fF>
			related_power_pin : VDD;
			related_ground_pin : VSS;
			timing () {
				related_pin : "CLK";
				timing_type : rising_edge;
				timing_sense : non_unate;

				cell_rise(ma_16_10_template) {
				values (		"200, 220, 240, 260, 280, 300, 320",\
						"210, 230, 250, 270, 290, 310, 330",\
								..7x7 array ..		");	}
				-- All 7x7 arrays:
				cell_fall(ma_16_10_template)			{ .. }
				fall_transition(ma_16_10_template)		{ .. }
				rise_transition(ma_16_10_template)		{ .. }
			}
		}

		pin(CLK) {
			direction : input;
			max_transition :	<in ps, technology and design dependent>
			capacitance :		<clk pin total capacitance in fF >
			min_pulse_width_high : <in ps, tech and design dependent>
			min_pulse_width_low : <in ps, tech and design dependent>
			clock : true;
			related_power_pin : VDD;
			related_ground_pin : VSS;

			/* Mem Read & Write – 16rows, 10bits */
			internal_power () {
				when : "( (DWWL[15]) | (DWWL[14]) | (DWWL[13]) |
(DWWL[12]) | (DWWL[11]) | (DWWL[10]) | (DWWL[9]) | (DWWL[8]) | (DWWL[7]) |
(DWWL[6]) | (DWWL[5]) | (DWWL[4]) | (DWWL[3]) | (DWWL[2]) | (DWWL[1]) |
(DWWL[0])) & ( (DRWL[15]) | (DRWL[14]) | (DRWL[13]) | (DRWL[12]) | (DRWL[11])
| (DRWL[10]) | (DRWL[9]) | (DRWL[8]) | (DRWL[7]) | (DRWL[6]) | (DRWL[5]) |
(DRWL[4]) | (DRWL[3]) | (DRWL[2]) | (DRWL[1]) | (DRWL[0]) ) ";
				related_pg_pin : "VDD";
				rise_power (scalar) { values (" <R+W power in uW> "); }
				fall_power (scalar) { .. }
			}
```

```
                    /* Mem Write – 16rows, 10bits */
                    internal_power () {
                              when : "( (DWWL[15]) | (DWWL[14]) | (DWWL[13]) |
(DWWL[12]) | (DWWL[11]) | (DWWL[10]) | (DWWL[9]) | (DWWL[8]) | (DWWL[7]) |
(DWWL[6]) | (DWWL[5]) | (DWWL[4]) | (DWWL[3]) | (DWWL[2]) | (DWWL[1]) |
(DWWL[0])) & (!( (DRWL[15]) | (DRWL[14]) | (DRWL[13]) | (DRWL[12]) |
(DRWL[11]) | (DRWL[10]) | (DRWL[9]) | (DRWL[8]) | (DRWL[7]) | (DRWL[6]) |
(DRWL[5]) | (DRWL[4]) | (DRWL[3]) | (DRWL[2]) | (DRWL[1]) | (DRWL[0]) )) ";
                              related_pg_pin : "VDD";
                              rise_power (scalar) { values (" <W power in uW> ");      }
                              fall_power (scalar) { .. }
                    }

                    /* Mem Read – 16rows, 10bits */
                    internal_power () {
                              when : "( (DRWL[15]) | (DRWL[14]) | (DRWL[13]) |
(DRWL[12]) | (DRWL[11]) | (DRWL[10]) | (DRWL[9]) | (DRWL[8]) | (DRWL[7]) |
(DRWL[6]) | (DRWL[5]) | (DRWL[4]) | (DRWL[3]) | (DRWL[2]) | (DRWL[1]) |
(DRWL[0])) & (!( (DWWL[15]) | (DWWL[14]) | (DWWL[13]) | (DWWL[12]) |
(DWWL[11]) | (DWWL[10]) | (DWWL[9]) | (DWWL[8]) | (DWWL[7]) | (DWWL[6]) |
(DWWL[5]) | (DWWL[4]) | (DWWL[3]) | (DWWL[2]) | (DWWL[1]) | (DWWL[0]) ))";
                              related_pg_pin : "VDD";
                              rise_power (scalar) { values (" <R power in uW> ");      }
                              fall_power (scalar) { .. }
                    }
                    /* CLK received, but no operation – 16rows, 10bits */
                    internal_power () {
                              when : "(!( (DRWL[15]) | (DRWL[14]) | (DRWL[13]) |
(DRWL[12]) | (DRWL[11]) | (DRWL[10]) | (DRWL[9]) | (DRWL[8]) | (DRWL[7]) |
(DRWL[6]) | (DRWL[5]) | (DRWL[4]) | (DRWL[3]) | (DRWL[2]) | (DRWL[1]) |
(DRWL[0]) )) & (!( (DWWL[15]) | (DWWL[14]) | (DWWL[13]) | (DWWL[12]) |
(DWWL[11]) | (DWWL[10]) | (DWWL[9]) | (DWWL[8]) | (DWWL[7]) | (DWWL[6]) |
(DWWL[5]) | (DWWL[4]) | (DWWL[3]) | (DWWL[2]) | (DWWL[1]) | (DWWL[0]) ))";
                              related_pg_pin : "VDD";
                              rise_power (scalar) { values (" <No-op pwr in uW> ");   }
                              fall_power (scalar) { .. }
                    }
          }

          bus(DRWL) {
                    bus_type : "ma_16_10_DRWL";
                    direction : input;
                    max_transition : <in ps, technology and design dependent>
                    capacitance : <in fF>
                    related_power_pin : VDD;
                    related_ground_pin : VSS;   }
          bus(DWWL)    { .. same as DRWL .. }
```

```
        bus(WBL_B) {
            bus_type : "ma_16_10_WBL_B";
            direction : input;
            max_transition : <in ps, technology and design dependent>
            capacitance : <in fF>
            related_power_pin : VDD;
            related_ground_pin : VSS;
            timing() { timing_type : hold_falling ;
                rise_constraint (scalar) {values("0");}
                fall_constraint (scalar) {values(" <hold time in ps> ");}
                related_pin : " CLK ";                              }
        }
        bus(WBL) { .. same as WBL_B .. }

        pin(BLK_RE) {
            direction : input;
            max_transition : <in ps, technology and design dependent>
            capacitance : <in fF>
            related_power_pin : VDD;
            related_ground_pin : VSS;
            timing() { timing_type : setup_rising ;
                rise_constraint (scalar) {values(" <setup time in ps> ");}
                fall_constraint (scalar) {values("0");}
                related_pin : " CLK ";                              }

        }

    }           //end of "cell(sram_brick_16_10)"

}               //end of library file
```

A.3 Memory Brick Netlist Generator

A template MATLAB code for 8T bitcell based SRAM memory brick netlist generator is as follows:

```
function [netlist_name, Cload_ARBL] = compiler_8T_brick(word_num, bit_num,
stack_num)

% ----------- INPUTS:
Num_2xRows = word_num;
Num_Cols = bit_num;
Num_Bricks = stack_num;
Num_Rows = Num_2xRows/2;

% ----------- LUT based Wire Cap Estimation:
[RBL_cap, ARBL_cap, RWL_cap, WWL_cap] = CAP_LUT_8T_brick(Num_Rows,
Num_Cols);

% ----------- CONSTANTS:
-- Define technology and bitcell dependent constant parameters:
--       beta, FO4, Gate_C_fFpUM, min_W, bcell transistor sizes (in uM)

% -- Logical Efforts of each gate in the topology

% WL driver
LE_wl_driver = [(2+beta)/3, 1, 1, 1];

% Local sense
skew = 2;
LE_nand = (2+beta*skew)/(skew*(1+beta));
LE_local_sense = [LE_nand, 2];

% Control circuit
LE_CNTRL_rst = [1, (2+beta)/3, 1, 1];
LE_CNTRL_r_en = [1, 1, 1];
LE_CNTRL_r_en_b = [1, 1];
LE_CNTRL_r_nand = [(2+beta)/3];
```

```
% ---------------------------------------------------
% ----------- Start compiling the netlist ---------
% ---------------------------------------------------

% 1) RWL/WWL driver (picking WWL cap as it is the worst case)

Cload_WL_gates = (2 * bcell_pass_N * Gate_C_fFpUM) * Num_Cols;
Cload_WL = Cload_WL_gates + WWL_cap;

% First decide what "Win (Cin)" == "SE of each gate" will be:

% a) check what min W gives:
nand_N = min_W;
nand_P = (min_W / 2) * beta;
Cin_nand = (nand_N + nand_P) * Gate_C_fFpUM;
SE_for_minWin = calculate_SE(LE_wl_driver, size(LE_wl_driver,2), Cload_WL, Cin_nand);

% b) if starting with min W leads to a non-ideal SE, then iterate for a better SE
```

-- *Essentially re-do the problem by first picking an SE, but make sure that nand_N stays relatively small*

-- *An example SE range: 2.5 < SE < 4.5*

```
% calculate all Cgates (and Wgates) with Cin, Cload, SE known:
% Cin_nand = set
Cin_inv2 = SE_wl_driver * (Cin_nand / LE_wl_driver(1,1));
Cin_inv3 = SE_wl_driver * (Cin_inv2 / LE_wl_driver(1,2));
Cin_inv4 = SE_wl_driver * (Cin_inv3 / LE_wl_driver(1,3));

W_NMOS_nand1 =( 2*Cin_nand /(2+beta) ) / Gate_C_fFpUM ;
W_NMOS_inv2 = ( Cin_inv2 /(1+beta) ) / Gate_C_fFpUM ;
W_NMOS_inv3 = ( Cin_inv3 /(1+beta) ) / Gate_C_fFpUM ;
W_NMOS_inv4 = ( Cin_inv4 /(1+beta) ) / Gate_C_fFpUM ;

W_NMOS_WL_driver = [ W_NMOS_nand1, W_NMOS_inv2, W_NMOS_inv3, W_NMOS_inv4];

% 2) Local Sense
```

-- *Same as WL driver sizing, include "total RBL_cap x #stacked_bricks" as the load capacitance*

```
% 3) RBL RESET PMOSES

Cg_read = W_NMOS_lvt_nand * (Gate_C_fFpUM * (2 + beta*skew));
RBL_cap_total = Cg_read + RBL_cap;
% Setting delay to 4xFo4 delays for reset:
rst_delay = 4 * FO4;

% RC Delay Formulas (Fitted from SPICE simulation results)
% Technology dependent and not disclosed
-- t = a x R^b x Cap^c              %with R in kohm, C in fF, t in ps
-- R = (k*Wp^-1)+m                  %with Wp in nm, R in kohm

% With these fitted formulas, then t = RC --> R = t/C
Rp = ( (rst_delay *1e+12)/(a*( (RBL_cap_total *1e+15)^c)))^(1/b); % in kohm
Wp = k /( Rp - m);                                                 % in nm
W_PMOS_RST = round(Wp);


% 4) CONTROL - Reset_EN signal generation
% 5) CONTROL - Read_EN control (Branched into 2)

-- (4) and (5) are same as WL driver sizing
-- Include "total WL cap + (#bits x gate_cap)" as the appropriate load capacitances


% ---- Save the compiled netlist ----
-- Save all the gate sizes to be used in timing & energy estimation

end
```

A.4 Memory Brick Timing Estimator

A template MATLAB code for critical path timing estimation on 8T bitcell based SRAM memory brick netlist is provided. Functions such as “FO_NAND_delay” are LUTs that store the delay and output slew of a gate in terms of input slew vs. gate fan-out.

```
function [ critical_path ] = perf_est_8T( netlist, cload )

load(netlist);  %loads: wl_driver, read & skew_read, array

% ---- Stage 1: WL driver ----
% Assuming DWL is received with a "slow input" slew for critical-path testing.
% calculate Fan-Out of gates:
fo_nand = (wl_driver(1,2)*3) / (wl_driver(1,1)*2) ;       %fo of wl_driver nand
fo_wl = wl_driver(1,3) / wl_driver(1,2) ;                 %fo of inverters in wl_driver

% WL driver of 4 gates: nand, inv1, inv2, inv_driving_RWL
[del_wl_1, slew_wl_1] = FO_NAND_delay(fo_nand*(4/3), 'l', 62) ; % pc=1
[del_wl_2, slew_wl_2] = FO_delay_pc2(fo_wl, 'h', slew_wl_1) ;     % fingered, pc=2
[del_wl_3, slew_wl_3] = FO_delay_pc2(fo_wl, 'l', slew_wl_2) ;      % fingered, pc=2
[del_wl_4, slew_wl_4] = FO_delay_pc2_RWL_drive(array(1,2), wl_driver(1,4),
slew_wl_3);      % fingered, pc=2
del_wl = del_wl_1 + del_wl_2 + del_wl_3 + del_wl_4 ;

% ---- Stage 2: Bitcells ----
% Extra LVT NAND added to the end of the RBL in 8T SRAM is handled inside the RBL
delay LUT as a load
[del_bcell, slew_rbl] = bcell_delay(array(1,1), read(1,1), slew_wl_4) ;

% ---- Stage 3: Local Sense (All LVT gates) ----
% Local sense for 8T sram brick: NAND (pc = 1) + Tri-state (pc = 2)
fo_LVTNAND = (read(1,2)*3) / ( read(1,1)*(1+skew_read) );
[del_read_LVTNAND, slew_read_LVTNAND] = nand_lvt_pc1_skew_interp(fo_LVTNAND,
slew_rbl, skew_read) ;
[del_read_LVTTRI, output_slew] = fo_lvt_tri(slew_read_LVTNAND, cload, read(1,2) );

del_read = del_read_LVTNAND + del_read_LVTTRI;

% ---- Critical Path: Reading a 0 from the last bit of first row  ----
critical_path = del_wl + del_bcell + del_read;

end
```

A.5 Memory Brick Layout Generator

A template SKILL code example for generating the layout of an 8T bitcell based SRAM memory brick is given in this section. First several basic procedures (equivalent of functions in SKILL coding language) are given. Then a code example that generates a leaf construct for the local sense leaf cell is provided. At the end, a code example that generates the modified leaf cell for the local sense block is given.

```
;; ----------------------- Basic Stretch Functions (Example):
procedure(stretch_right_side(cellid layer purpose extend_right)
        shapes=setoff(ARG cellid->shapes ARG->layerName == layer && ARG->
purpose == purpose)
        foreach(ARG shapes
                llx=LLX(ARG->bBox)
                lly=LLY(ARG->bBox)
                urx=URX(ARG->bBox)
                ury=URY(ARG->bBox)

                new_urx=urx+extend_right
                newbBox=list(list(llx lly) list(new_urx ury))
                ARG->bBox=newbBox       )                           )
;; ----------------------- Basic Move Functions (Example):
procedure(move_right(cellid layer purpose to_right)
        shapes=setof(ARG cellid->shapes ARG->layerName == layer && ARG->
purpose == purpose)
        foreach(ARG shapes
                llx=LLX(ARG->bBox)
                lly=LLY(ARG->bBox)
                urx=URX(ARG->bBox)
                ury=URY(ARG->bBox)

                new_llx=llx+to_right
                new_urx=urx+to_right
                newbBox=list(list(new_llx lly) list(new_urx ury))
                ARG->bBox=newbBox       )                           )
;; ----------------------- Getting BBOX corners of a shape:
procedure(LLX(bBox)
        caar(bBox)   )   ;; similarly using cadar for LLY, caadr for URX, cadadr for URY
```

```
;; Create a leaf construct for the leaf cell
procedure(create_read_rst(unique_brick_name new_lib g_W stretch_reset_Mx_up)
        id = strcat( "read_rst_" unique_brick_name)
        cvID_read_rst = dbOpenCellViewByType("SKILL_LIB" "Read_RST" "layout")
        dbCopyCellView(cvID_read_rst new_lib id "layout" nil nil t)
        dbClose(cvID_read_rst)
        cvID_read_rst_modded = dbOpenCellViewByType(new_lib id "layout" "" "a")

        ;; ------- Modifications
        default_RX_w= <non disclosed, in um>
        default_via_pos = <non disclosed, in um>
        default_via_num = <non disclosed, integer>
        stretch_amount=g_W-default_RX_w

        stretch_top_side(cvID_read_rst_modded "M1" "drawing" stretch_amount)
        stretch_top_side(cvID_read_rst_modded "RX" "drawing" stretch_amount)
        stretch_top_side(cvID_read_rst_modded "PC" "drawing" stretch_amount)
        ;; M4, M5, M6 are used as dummy layers for easier manipulation of unique M1
shapes
        move_up(cvID_read_rst_modded "M4" "drawing" stretch_amount)
        move_up(cvID_read_rst_modded "M5" "drawing" stretch_amount)
        stretch_top_side(cvID_read_rst_modded "M5" "drawing" stretch_reset_Mx_up)
        stretch_top_side(cvID_read_rst_modded "M6" "drawing"

        (stretch_reset_Mx_up+stretch_amount))

        ;; ---- Contact (CA) Vias:
        manipulate_CA_vias_UP(cvID_read_rst_modded g_W default_RX_w
default_via_pos default_via_num stretch_amount)

        ;; ----- Mx colors: (Revert all M4,5,6 back to M1)
        shapes_M45_to_M1=setof(ARG cvID_read_rst_modded->shapes ARG->
layerName == "M4" || ARG->layerName == "M5")
        foreach(ARG shapes_M45_to_M1
                ARG->layerName="M1"
        )
        ;; Repeat the same for all high-level Mx, transform all back to “M1”

        ;; ----------- Mods done
        dbSave(cvID_read_rst_modded)
        dbClose(cvID_read_rst_modded)
)
```

```
;; Create the modified leaf cell
procedure(create_local_sense(unique_brick_name new_lib)

        ;; Get the gate widths for Local Sense cell
        gates = <read in gate sizes from a file>

        ;;---- Create all new leaf constructs and modify them
        -- Going through generation of all leaf constructs:
                ...
        create_read_rst_04(unique_brick_name new_lib gate_rst (gate_nand*2.0)
(gate_tri*2.0) )
                ...

        ;;----- Create the modified leaf cell:
        read_rst_name = strcat( "read_rst_" unique_brick_name)
        cvID = dbOpenCellViewByType("SKILL_LIB" "Read_RST_outline" "layout")
        dbCopyCellView(cvID new_lib read_rst_name "layout" nil nil t)
        dbClose(cvID)
        cvID_read_rst_modded = dbOpenCellViewByType(new_lib read_rst_name
"layout" "" "a")

        ;; Start placing leaf constructs cells into the final leaf cell
        x=0.0
        y=0.0

        -- Going through placing all leaf constructs w.r.t. each other’s coordinates:
                ...

        ;; ---- construct #04
        leaf_cell_name=strcat( "read_rst_04_" unique_brick_name)
        sub_leaf_cell=dbOpenCellViewByType(new_lib leaf_cell_name "layout")
        dbCreateInst(cvID_read_rst_modded sub_leaf_cell "inst4" list(x y) "R0")
        bbox=sub_leaf_cell~>bBox
        x=nth(0 upperRight(bbox))
        x_tri=x                                 ;; update x
        y_nand = nth(1 upperRight(bbox))        ;; update y to use later on

                ...

        ;; -------------- Save & close
        dbSave(cvID_read_rst_modded)
        dbClose(cvID_read_rst_modded)
)
```