A Case for Standard-Cell Based RAMs in Highly-Ported Superscalar Processor Structures

Sungkwan Ku[1]*, Elliott Forbes[2], Rangeen Basu Roy Chowdhury[3], Eric Rotenberg[1,4]
[1]North Carolina State University, 890 Oval Drive, Raleigh, NC USA
[2]University of Wisconsin – La Crosse, 1725 State St, La Crosse, WI USA
[3]Intel, 2200 Mission College Blvd, Santa Clara, CA USA
[4]Qualcomm, 8041 Arco Corporate Dr, Raleigh, NC USA
*E-mail: sku2@ncsu.edu

Abstract— Highly-ported memories are pervasive within superscalar processors. Accordingly, they have been targets for full-custom design using multi-ported versions of the 6T SRAM bitcell. Unfortunately, full-custom design of highly-ported memories is becoming exceedingly difficult in deep sub-micron technologies. This paper makes the case for implementing highly-ported memories with standard cells (flip-flops, muxes, clock buffers). In lieu of exotic peripheral circuits for each port, standard-cell SRAMs use muxes. Consequently, area differences between full-custom and standard-cell designs are greatly reduced at a high number of ports. To also compete with full-custom memories in terms of timing and power, we introduce a standard-cell memory compiler with three key features: (i) per-row clock gating, (ii) a new tri-state based mux standard cell, and (iii) a modular layout strategy, which is the centerpiece of the memory compiler. For a 16-read/8-write 128-entry register file, our modular standard-cell memory consumes 13% more area and 4% more power, and is 35% faster, than the custom memory produced by FabMem. The automatic (built-in) robustness of standard cell designs further weigh in their favor, contrasted with exquisite transistor sizing/tuning of intertwined sub-circuits in a full-custom design.

Keywords— Standard-Cell, RAM, Memory Compiler, Superscalar

I. Introduction

The demand for higher performance in computing has led to the adoption of superscalar execution in all tiers of computing, including mobile computing and application processors. As instruction fetch width and the number of parallel execution lanes increase, superscalar memory structures (e.g., physical register file, reorder buffer, scheduler, rename tables, etc.) require correspondingly more ports. Because of the importance of highly-ported memories to core area, performance, and energy consumption, traditionally, they have been targets for full-custom design using multi-ported versions of the 6T SRAM bitcell. Full-custom design of highly-ported memories is becoming exceedingly difficult, however. The design effort to reliably handle PVT variations and narrow noise margins at low operating voltages is challenging for deep sub-micron technology [2], [3]. This is true for single-ported 6T SRAM, and is even more challenging when designing an efficient and reliable twelve-ported register file (eight read and four write ports) required for a 4-way superscalar core. Microarchitectural alternatives, such as replication [5], [6] or banking of fewer-ported memories, have efficiency or performance drawbacks, respectively.

This paper makes the case for standard-cell based SRAMs (flip-flops, muxes, clock buffers) as the solution to the problem of highly-ported deep-submicron memories. In lieu of exotic peripheral circuits for each port (bitline precharge, sense amps, etc.), standard-cell SRAMs use muxes. Consequently, we observe that the area difference between full-custom and standard-cell memories is greatly reduced at a high number of ports. To also compete with full-custom memories in terms of timing and power, we introduce a standard-cell memory compiler with three key features.

1. Per-row clock gating reduces both clock power and switching inside the D flip-flops.
2. A new tri-state buffer based mux cell is added to the standard cell library. Employing it reduces total routing and further reduces switching inside the D flip-flops. The new mux cell presents the memory compiler with another choice for optimizing timing and power.
3. A modular layout strategy reduces total routing. A layout is generated for a smaller building block. A block's layout is made efficient by careful floorplanning. The modular layout allows stacking multiple blocks in series to compose the overall memory.

Our standard-cell memory compiler provides an easy interface for designers to explore the characteristics of highly-ported SRAMs, in which the full-custom memory faces challenges of reliability and high verification effort. The automatic (built-in) robustness of standard cell designs further weigh in their favor, contrasted with exquisite transistor sizing/tuning of intertwined sub-circuits in a full-custom design.

We compare area, timing, and power of layouts of our standard-cell memories and full-custom memories generated by the FabMem highly-ported RAM/CAM compiler [7]. Even though FabMem is not a commercially available tool, the tool adequately evaluates the fundamental structures of highly optimized multi-ported bitcell, bitline precharge, sense amplifier, and dynamic logic decoder that are designed and verified by Shah [7] using the FreePDK 45nm [8] process design kit.

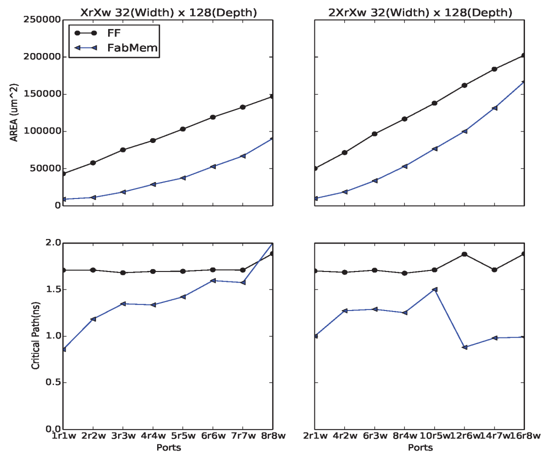


Fig. 1. Pre-layout area and cycle time comparison of FabMem generated RAMs and synthesized flip-flop based RAMs (X read/ X write).

II. Motivation and Contributions

We begin by exploring the area, cycle time, and energy costs of adding ports in two different multi-ported SRAM implementation styles: full-custom (6T SRAM, scaled-up for multiple ports) versus fully synthesized (D flip-flop based). We consider a fixed size of 128 32-bit words (i.e., the SRAM's dimensions are 128 deep by 32 wide) and vary the number of ports from 2 ports (1r1w) to 24 ports (16r8w).

For every port combination, FabMem uses a custom designed bitcell that achieves the optimum characteristics. Different circuit level decisions were made for different bitcells in the FabMem library. FabMem uses a built-in estimation tool to explore different array organization options for a specified SRAM size.

Figure 1 compares area and timing of the FabMem-generated (labeled FabMem) and synthesized (labeled FF, for flip-flop) designs. The graphs on the left of Figure 1 consider X read ports and X write ports (XrXw), varying X. And the right graphs compare 2X read ports and X write ports (2XrXw). Area estimates of synthesized designs include cell area and estimated wire area from the pre-layout synthesis report.

The area of the 1r1w FabMem bitcell (8 transistors, two wordlines, two bitlines) is 2.544 μm^2, compared to 4.522 μm^2 for the 29-transistor D flip-flop. For 1r1w, the total estimated area shows the synthesized design is 4.8 times larger than the FabMem design. However, the synthesized design is 1.62 times larger than FabMem for 8r8w and only 1.2 times larger for 16r8w. The area of the 8r8w FabMem bitcell (36 transistors, 16 wordlines, 16 bitlines) is 19.668 μm^2. While the size of the D flip-flop is unchanged, more mux cells are added for each additional port.

The FabMem bitline length increases linearly with the number of ports (via the wordlines), increasing read latency. If we divide the bitcell array in half (along the length of the bitline) and apply column muxing, read latency may be reduced. FabMem automatically recommends a degree of column muxing that minimizes read latency. As can be seen in the lower right graph of Figure 1, timing is greatly improved for FabMem 12r6w, owing to column muxing. In general, FabMem is faster than FF. However, a full-custom design requires careful static noise margin (SNM) simulation while the synthesized flip-flop design can take advantage of Electronic Design Automation (EDA) tools for static timing analysis.

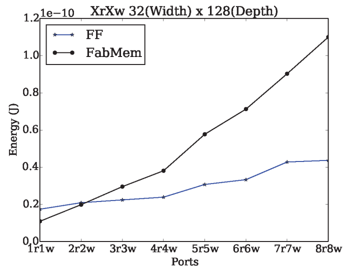


Fig. 2. Pre-layout energy comparison of FabMem generated RAMs and synthesized flip-flop based RAMs (X read/X write).

For a fair comparison of energy consumption, we converted the synthesized gate-level netlist to the transistor-level SPICE netlist, so that both the FabMem and FF designs could be simulated with Synopsys HSPICE. FabMem generates a full SPICE netlist in addition to the layout. Note, we reduced the memory depth to 16 instead of 128 to reduce simulation time. The same test vectors were applied to the read and write ports of both designs over four clock cycles. Energy is shown in Figure 2. FabMem actually consumes more energy than FF for more than four ports (they break even for 2r2w). The percentage of energy consumed by precharge and sense amplifier circuits increases from 53.1% for 1r1w to 72.5% for 8r8w. With more ports, the bitlines are longer and consume more energy for FabMem, whereas more energy is consumed in the additional mux cells for the synthesized design.

These experiments show that for a high number of ports, there is an opportunity for a synthesized SRAM design to achieve performance and area close to that of a full-custom SRAM, with better energy consumption. Given that the full-custom SRAM requires high design effort, we can consider building highly-ported SRAMs by leveraging design automation. Moreover, synthesized SRAMs have the advantage of being very portable. When a design is ported to a new process, the flip-flop based SRAMs can be easily re-synthesized without the need for involved physical design of full-custom SRAMs. The main contributions of this paper are as follows:

i. We identify the important trend that for higher number of ports, synthesized flip-flop based SRAMs become very competitive to custom designed SRAMs.

ii. We propose an automatic standard-cell based memory compiler using hierarchical design automation in order to reduce the design and verification effort of fully synthesized multi-ported SRAMs.

iii. We compare area, timing, and power of layouts of our standard-cell memories with FabMem generated custom memories.

III. Challenges for Flip-Flop Based Designs

The challenge in synthesized flip-flop based SRAMs lies in creating a competitive physical layout. Placing and routing a large number of standard cells using automatic place and route tools results in a wide variation in area utilization. This is due to the unpredictability of placement algorithms. We found two bottlenecks when routing synthesized flip-flop SRAM designs.

First, placing flip-flops and multiplexer cells together is challenging. On one hand, wordline flip-flops should be placed as close together as possible to minimize clock skew and read timing. However, the group of muxes need to be placed in different rows to minimize spanning wires to storage cells. These opposing constraints force CAD tools to spread flip-flops and muxes to be able to completely route nets lowering area utilization and increasing wire delay.

Second, with an increase in SRAM size and port numbers, the design effort to route cells and insert clock buffers/repeaters is increased. With additional cells, the design effort to meet target area and timing constraints requires iterative optimization to tune clock buffers and repeaters.

To obtain a layout of a synthesized SRAM that is competitive with a custom designed SRAM in terms of power, timing and area, we discuss three key optimizations that make the design more efficient and amenable to automatic place and route.

A. Clock gating

In a synthesized flip-flop array, to retain values, the output of each flip-flop is fed back to its input. To change the value, a mux is inserted at the input to select between the new value and the feedback value. A common optimization is to eliminate the mux and feedback path and clock gate the word line. Our standard cell library includes a 3.192μm^2 clock gating cell, which allows us the opportunity to employ this alternative.

B. Custom Mux Cell

Our approach requires adding a large number of multiplexers to the flip-flop arrays. For each read/write data, there is an existing path to access each word line, which requires multiple stages of mux cells. Additional ports linearly increase the number of multiplexers, which increases the wire length and delay. For higher efficiency, we created new large input multiplexer standard cells to reduce routing congestion of complex data paths. Our custom n-to-1 multiplexer cell uses tri-state inverters.

For 8 write ports, an 8-to-1 mux standard cell was implemented to route a write data bit from 8 write ports to a flip-flop 'D' input. Figure 3 shows the schematic and layout of the new 8-to-1 mux cell. Write conflicts from different write ports are avoided architecturally.

For 8 read ports with 128 rows of SRAM memory, each read data port needs to choose a bit from the total number of rows of the SRAM memory; this would require a 128-to-1 mux. In this case, we divide the 128 rows into 8

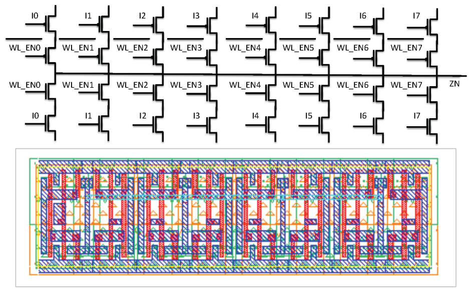


Fig. 3. Schematic and layout of the custom designed for 8-input mux cell.

modules (i.e., each module has 16 rows of the SRAM memory) so that one 16-to-1 mux cell or two 8-to-1 mux cells can be used in each divided module. The new custom n-to-1 multiplexer cell is characterized by Encounter Library Characterizer vers. v13.10.

C. Modular Design

To further aid the CAD tools in automated placement and routing of a flip-flop based SRAM, we propose a structured reusable block based design. A block, called a RAM module, includes a flip-flop storage array and associated decode and muxing logic for read and write ports. Multiple of these RAM modules can then be stacked to build larger multi-ported SRAMs. Each read/write port selects a RAM module and unselected RAM modules are used as bypass buffers. Appropriate placement constraints are used to guide the CAD tools. The modularity of the design is maintained throughout the design flow to reduce the physical design effort by making placement and routing somewhat deterministic. This helps generate a layout that is more efficient in terms of area than a completely unconstrained and unstructured synthesized SRAM design.

Figure 4 (a) shows the block diagram of a 8r/8w RAM module. On the read side, the module has 8 read address input ports and 8 data outputs. Muxes choose between data from outputs of flip-flops in the same RAM module or from another RAM module. Common logic and wires can be shared within a RAM module and between RAM modules. One such example are the read addresses that are connected to other RAM modules using just repeaters. On the write side, 8 write address and 8 data are shared across RAM modules. Only the selected RAM module for each write port decodes the corresponding write address and generates a clock enable signal to update the write data.

IV. Modular SRAM Compiler

To simplify the design and analysis of flip-flop based SRAMs and to make them more usable, we created a "Modular SRAM Compiler". The compiler consists of three main tools (flows):

- A netlist generator script that generates the modular SRAM netlist as per the dimensions and the number of ports specified by the user.

- A layout flow that creates a layout of the SRAM using a hierarchical layout flow and generates a GDS and LEF file for use with standard EDA tools.
- A characterization and library generation flow that generates a Liberty library (LIB file) for the SRAM that can be used in standard ASIC flows for timing and power analysis.

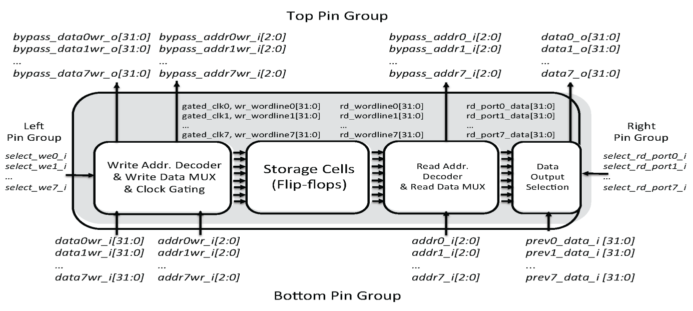
A. Netlist generator

Given an RTL design of the SRAM, synthesis tools can automatically synthesize a gate-level netlist for the SRAM and optimize it to meet a specified target frequency constraint. However, the exact implementation of the gate-level netlist varies from one synthesis run to the next and this inconsistency creates challenges for automated place and route of the generated gate-level netlists. Physical design of such netlists is extremely tedious and requires manual floorplanning to get the best results. To obviate the need for manual floorplanning and to remove inconsistencies in the gate-level netlist, we created a parametrized netlist generator in Perl to directly compose a gate-level netlist for a reusable RAM module. Along with the netlist, the script also generates timing constraints for optimization and placement constraints for deterministic routing to aid in automated routing of the large number of wires. These constraints are then used to perform post-synthesis optimizations of the RAM module using our synthesis tool. The tool inserts buffers and adjusts cell sizes to meet timing requirements.

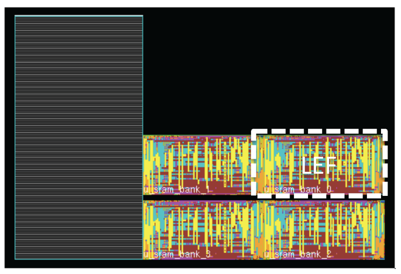
The netlist generator script also generates a top-level wrapper netlist for the required SRAM using a modular approach. The previously generated RAM module is treated as a black box cell and instantiated as many times as needed to compose the required SRAM. The top-level wrapper netlist includes extra decoding logic to enable the corresponding RAM module for each read/write port. The final SRAM top-level wrapper netlist and the associated constraints then feed into the hierarchical layout flow.

B. Hierarchical Layout

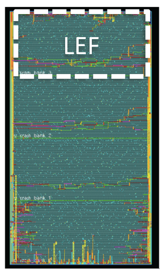
In order to maximize reuse and minimize SRAM generation time, we use a hierarchical layout approach. A reusable RAM module is placed and routed first. The optimized RAM module netlist and associated placement constraints are used for this first pass. The placement constraints guarantee accurate placement of pins, so they line up perfectly when multiple RAM modules are stacked on top of each other to create the required SRAM dimension. This allows for the most efficient use of area in the final SRAM layout. The completed RAM module layout is then streamed into Cadence Virtuoso^{TM}and the Cadence Abstract^{TM}tool to generate a LEF file. The first pass generates a post-layout netlist, and LEF and GDS files for the RAM module. These are used as libraries in the second pass of the flow to lay out the final SRAM. The effort required for the second phase is minimal as it involves placing and routing the combinational logic stitching the RAM modules together. Similar to the first pass, the second pass generates a post-layout netlist, a LEF and a GDS file for the entire SRAM.



(a) Block diagram and pin layout of a reusable block design



(b) Initial floorplan (pre-route) of a modular SRAM



(c) Completely routed layout

Fig. 4. Example of a modular SRAM layout.

Figure 4 (b) shows the initial floorplan of a 32-entry by 32-wide SRAM with 8r8w ports using this modular-driven approach. The place and route tool can stack four modules from bottom to top and place selection decoder cells outside of the stacked area. The routing effort is just in wiring the module selection decoder cells to each module. Figure 4 (c) shows the completed layout. We observed the total wire length to route modules to be greatly reduced.

C. Library Generation

The final process is to get a Liberty library (LIB file) for the generated SRAM design. We use Synopsys Prime Time to analyze the timing and power for the completed layout. In order to populate the LIB file, static timing analysis is run on the post place and route netlist. Parasitic information extracted from the layout as a SPEF file is used by the timing analysis flow. For power analysis, gate level simulation is performed on the routed netlist to get the activity file in Value Change Dump format (VCD). Using this VCD, extracted SPEF, and the routed netlist, we measure power for read/write accesses to the SRAM. The library generator script uses these results to populate a Liberty template to create a final LIB file for the generated SRAM.

V. Experimental Methodology

In order to evaluate the effectiveness of our modular SRAM design, we compare to fully synthesized designs (baseline) and to FabMem compiled designs for various sizes of SRAMs. The baseline designs use an automated CAD tool flow for synthesis, placement and routing under the tightest possible timing constraints. In the baseline designs, we vary the area utilization constraint to meet the timing requirements and to fully route the design.

For our design exploration, we vary the number of ports and SRAM sizes. The bit storage type is either flip-flop based or 6T bitcell (cross-coupled inverter) based. The mux type is a choice between static CMOS mux, tri-state style mux, and dynamic logic based mux. The designs we evaluate have LVS/DRC clean layouts. To analyze timing and power of the baseline and modular designs, we use Synopsys Prime Time PX ^{TM}. We use extracted parasitic information from the layouts (SPEF) as well as value change dump (VCD) from gate-level simulations for the Prime Time flows. To obtain timing and power for FabMem generated designs, we use extracted parasitic information from the layouts (PEX) and run transistor-level SPICE simulations.

VI. Results

In this section, we perform a sensitivity study of our Modular SRAM compiler to understand the impact of the various optimizations and evaluate different SRAM implementation styles for various sizes and ports. We also use our Modular SRAM compiler in case study with a 8-way supersclar out-of-order processor.

A. Sensitivity Study

To study the impact of the various optimizations performed by the Modular SRAM compiler, we perform a sensitivity analysis of highly-ported SRAMs, which are our primary regime of interest. We first study the impact of clock-gating and the custom mux cell. Figure 5 shows the area, cycle time, and power impact of introducing clock-gating and using the new mux cell. In this case we use a 32(W)x32(D) memory with 16r8w ports. The baseline design is an automated place-and-route design with no custom mux cells or clock-gating. The bar labeled M-1 shows the impact that clock-gating alone has when applied to a baseline SRAM. M-2 bar also uses clock-gating but additionally allows the use of the custom mux cell. Compared to both the baseline and to FabMem designs, clock-gating alone (M-1) can save area and power, but comes at the expense of an increase cycle time. The impact on timing is due to the clock buffer placement and routing of the clock tree. However, by introducing the new mux cell (M-2), area, cycle time, and power are all improved compared to both the baseline and FabMem designs. The main sources of improvement are in the reduced wire length and the reduced switching activity.

Since a SRAM can be partitioned into modules in many different ways, in our second study, we study the impact of module granularity, i.e. the number of blocks (modules) in a modular SRAM. These studies do not use the custom mux cell but uses clock-gating. Figure 6 shows the result for a 32(W)x128(D) SRAM with 8r8w ports. The baseline does not use the modular approach. B-2, B-4, B-8, and B-16 refer to the number of blocks: 2, 4, 8, and 16 blocks respectively in a modular SRAM. Increasing the number of blocks improves the area of the modular design compared

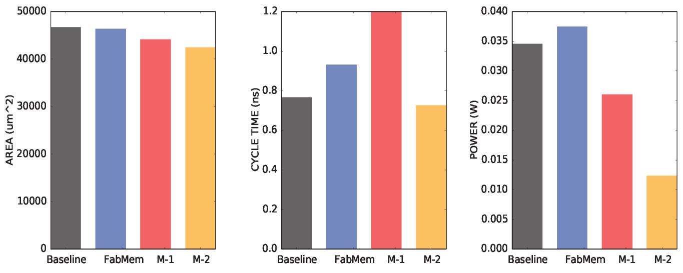


Fig. 5. Impact of clock-gating and custom mux cell.

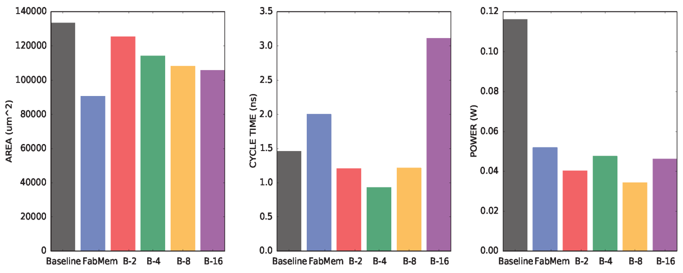


Fig. 6. Impact of module granularity(number of blocks) in a modular SRAM.

to the baseline – approaching that of the FabMem design. Power is not impacted much by using more blocks, but a large cycle time impact is noted when using 16 blocks. This is due to serialization of the read paths from stacking the blocks on top of each other, which increases the wire length and hence, the critical path.

The module granularity has an area and timing trade-off. The clock gating has a timing and power trade-off. The custom mux cell shows impact on power saving. The area, timing, and power of a SRAM depends on a combination of these features. The tool explores all features to arrive at best solution for a given SRAM dimension.

B. Perforamnce, Power, and Area

B.1 Area

Area comparisons are shown in Figure 7 (a). When compared to the Baseline designs, the area of the Modular designs are comparable or slightly better than the Baseline area for all sizes and ports. We observed that Modular designs can achieve even better area if we reduce the block size, which increases the number of stacked blocks. In that case, place-and-route achieves higher utilization with a smaller partition, and hence a higher overall utilization. However, the block size presents a trade-off between area and timing because stacking more blocks in series increases the critical path delay.

When comparing our Modular design to the equivalent FabMem design, we find that the percent of area overhead of the Modular design decreases as the number of ports increases. Furthermore, there is a cross-over point where the area of the Modular design is lower than the area of the FabMem design. As the RAM size is increased, this cross-over point occurs at a higher number of ports. This is evident in the graph for the smallest RAM (cross-over at 8r4w) and medium-sized RAM (cross-over at 16r8w). For the largest RAM, the cross-over point is beyond the maximum number of ports considered, but the area gap is small – just 13% more area.

B.2 **Timing**

Timing comparisons are shown in Figure 7 (b). As the RAM size increases, the cycle time advantage of FabMem decreases with respect to both the Modular and the Baseline designs. Modular designs have better timing than Baseline designs for most datapoints, and have significantly better timing for the largest RAM with 8r4w, 8r8w, and 16r8w ports (the most complex).

The large RAM with 8r4w, 8r8w, and 16r8w ports are key design points, and the Modular designs outperforms FabMem for these RAMs. This cross-over coincides with a discontinuity – the Modular design cycle time decreases from 4r4w to 8r4w ports. This has to do with using either our tri-state mux cells (for 1r1w through 4r4w) or existing mux cells (for 8r4w, 8r8w, 16r8w), whichever is best for timing. For the large RAM and highest number of ports, the tri-state mux cell presents a trade-off between timing and power. The isolated "X" datapoints in the bottom graphs of Figure 7 (b) and Figure 7 (c) illustrate this trade-off. These datapoints are for the large RAM with 8r4w and 8r8w ports, using the tri-state mux cell instead of the existing mux cell. Using the tri-state mux cell reduces power at the expense of timing in these Modular designs.

B.3 **Power**

Power comparisons are shown in Figure 7 (c). Power includes both static and dynamic power. Modular and Baseline designs are fairly comparable in area and timing, owing to their common basis in standard cells. However the Modular design optimizations clearly pay off in terms of power. The sources of power savings are in the gated clocks of D flip-flops, the tri-state data muxes, and minimized routing wires.

The power of Modular designs is always better than FabMem designs. FabMem uses dynamic logic, including the address decoders and bitline precharge/sense circuitry. This is to be competitive in timing, but comes at the expense of power. However FabMem does become increasingly competitive in power as the RAM size increases. In particular, FabMem and Modular designs have comparable power for the large RAM, and have equal power for the large RAM with 16r8w ports.

C. **Case Study: 8-way superscalar processor**

To study the effectiveness of our *Modular SRAM Compiler*, we used it to generate SRAMs for an 8-way superscalar out-of-order processor. The superscalar processor was generated using the AnyCore toolset [4]. Out-of-order superscalar processor designs tend to use many highly multi-ported SRAMs in their pipelines. From the various SRAMs used by the generated design, we selected only the relatively large highly-ported SRAMs for this study.

TABLE I
Highly-ported memories used in our 8-way superscalar processor. Area is in sq. mm, Cycle Time is in ns, and Power is in mW.

RAM Name	Dimensions	Measurement	FabMem	Baseline	Modular
Phy. Register File	32(W)x128(D) 16 read/8 write	Area	0.178	0.205	0.201
		Cycle Time	1.79	1.65	1.33
		Power	83.3	165.0	86.8
Free List	7(W)x128(D) 8 read/8 write	Area	0.046	0.027	0.022
		Cycle Time	1.95	1.28	1.05
		Power	3.46	23.0	7.59
Active List	80(W)x64(D) 8 read/8 write	Area	0.090	0.203	0.151
		Cycle Time	1.36	1.77	1.88
		Power	39.58	84.0	37.28
Issue Queue	164(W)x32(D) 8 read/8 write	Area	0.088	0.205	0.164
		Cycle Time	1.07	2.14	1.94
		Power	81.1	82.2	38.7

Specifically, we compared 3 different implementations of the physical register file, free list, reorder buffer, and the issue queue SRAMS.

Table I shows the dimensions of these SRAMs and also presents the area, cycle times, and power for the three different implementations of these RAMs.

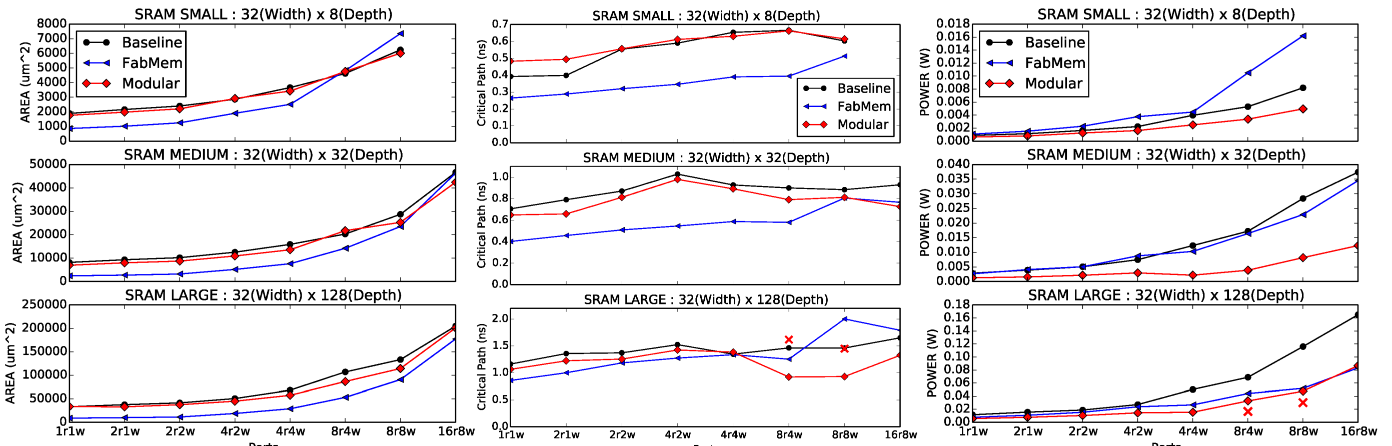
When compared to the Baseline designs (automatically synthesized), the area, cycle time, and power of the Modular designs are better. The only exception is the cycle time of the active list RAM. For this RAM, the Modular SRAM compiler used 4 modules to build a 80(W)x64(D) memory and the read data path becomes the critical path for this configuration due to stacking the modules on top of each other. This serialization degrades the critical path compared to the Baseline synthesized design. Even though timing is affected by this serialization, we chose this configuration as it saved significant amount of area and power compared to the baseline.

Comparing our Modular design to the equivalent FabMem design, the timing of physical register file and free list are better than FabMem due to the depth of memory. FabMem is sensitive to the depth of memory since the longer bitline takes more time for charging and discharging while reading. The power of the Modular designs are comparable or better than FabMem. The area of FabMem is better than the Modular designs except for free list. This is due to the cross-over point explained in section **6-B.1**.

VII. **Related Work**

IBM has implemented various techniques to improve the performance and efficiency of the register files in their POWER7 [10] and POWER8 processors [11]. To support four writes per cycle, the POWER7 writes on both rising and falling clock edges on two physical write ports. Additionally, reads are grouped by thread context and use a double-bit cell with internal muxing to support six reads per cycle (in the vector register file). These techniques allow for a high number of effective read and write operations per cycle, while keeping bit lines short to reduce switching capacitance.

POWER8 introduced a three-level hierarchical bit line implementation. One level of the hierarchy can potentially



(a) Area (b) Critical Times (c) Power

Fig. 7. Area, Cycle Times, and Power Comparision.

reduce read power by only discharging some of the register file entries. This comes at the expense of additional access delay. To overcome some of this additional delay, another level of the hierarchy is used to physically shorten the length of bit lines. The bit cells themselves are implemented using an 8T design. These 8T cells provide separate read and write ports.

IP vendors provide memory compilers to aid in generating custom SRAM memory arrays. However, they typically only support a limited number of ports. For example, ARM licenses a memory compiler for a variety of process technologies, but is limited to a maximum of two read ports and one write port per SRAM array [1]. To overcome the port limitations, banking and replication can be used to trade area and power to support higher port requirements. Banking [9] can use memories with a lower number of ports, with logic to guide reads/writes to the appropriate bank. Additional latency is incurred, however, when multiple readers/writers need to access a conflicting bank. Ports can be mimicked by replicating [5] values for a given entry in multiple physical SRAM arrays. This technique does not suffer from bank conflicts and thus does not increase access latency. However, replication comes at the expense of values being held in SRAM arrays redundantly.

VIII. **Summary**

The findings in this paper support modular standard-cell based memories as a robust and low-effort alternative to fragile and high-effort full-custom memories, in the regime of highly-ported RAMs. We presented a modular standard-cell based memory compiler with three key features for generating competitive multi-ported SRAMs.

IX. **Acknowledgement**

This work is supported by NSF grant CCF-1218608. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] ARM Embedded Memory IP: SRAM, Register File, and ROM Memory Compiler. Available: https://www.arm.com/products/physical-ip/embedded-memory-ip/index.php.

[2] A. Agarwal, B.C. Paul, et al. Process Variation in Embedded Memories: Failure Analysis and Variation Aware Architecture. *IEEE Journal of Solid-State Circuits*, 40(9):1804–1814, September 2005.

[3] L. Chang, D.J. Frank, et al. Practical Strategies for Power-Efficient Computing Technologies. *Proc of the IEEE*, 98(2):215–236, February 2010.

[4] Rangeen Basu Roy Chowdhury, Anil Kumar Kannepalli, Sungkwan Ku, and Eric Rotenberg. Anycore: A synthesizable rtl model for exploring and fabricating adaptive superscalar cores. In *Proceedings of the ISPASS 2016 - IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '16. IEEE Computer Society, 2016.

[5] Brandon H. Dwiel, Niket K. Choudhary, and Eric Rotenberg. FPGA Modeling of Diverse Superscalar Processors. In *ISPASS*, April 2012.

[6] Shen-Fu Hsiao and Pu-Cheng Wu. Design of Low-Leakage Multi-Port SRAM for Register File in Graphics Processing Unit. In *ISCAS*, June 2014.

[7] Tanmay A. Shah. Fabmem: A multiported ram and cam compiler for superscalar design space exploration. Master's thesis, 2010.

[8] James E. Stine, Ivan Castellanos, et al. FreePDK: An Open-Source Variation-Aware Design Kit. In *MSE*, pages 173–174, June 2007.

[9] J.H. Tseng and K. Asanovic. Banked Multiported Register Files for High-Frequency Superscalar Microprocessors. In *ISCA*, June 2003.

[10] D. F. Wendel, J. Barth, et al. IBM POWER7 Processor Circuit Design. *IBM Journal of Res and Dev*, 55(3):6:1–6:8, May 2011.

[11] V. Zyuban, J. Friedrich, et al. IBM POWER8 Circuit Design and Energy Optimization. *IBM Journal of Research and Development*, 59(1):9:1–9:16, January 2015.