Multi-Ported Memories for FPGAs via XOR

Charles Eric LaForest, Ming G. Liu, Emma Rae Rapati, and J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto, Canada
{laforest,steffan}@eecg.toronto.edu, {emma.rapati,minggang.liu}@utoronto.ca

ABSTRACT

Multi-ported memories are challenging to implement with FPGAs since the block RAMs included in the fabric typically have only two ports. Any design that requires a memory with more than two ports must therefore be built out of logic elements or by combining multiple block RAMs. The recently-proposed Live Value Table (LVT) [8] design provides a significant operating frequency improvement over conventional approaches. In this paper we present an alternative approach based on the XOR operation that provides multi-ported memories that use far less logic but more block RAMs than LVT designs, and are often smaller and faster for memories that are more than 512 entries deep. We show that (i) both designs can exploit multipumping to trade speed for area savings, (ii) that multipumped XOR designs are significantly smaller but moderately slower than their LVT counterparts, and (iii) that both the LVT and XOR approaches are valuable and useful in different situations, depending on the constraints and resource utilization of the enclosing design.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Style—*Shared Memory*

General Terms

Design Performance Measurement

Keywords

FPGA, memory, multi-port, parallel, XOR

1. INTRODUCTION

FPGAs are increasingly used to implement complex systems-on-chip that require frequent communication, sharing, queuing, and synchronization among distributed functional units and compute nodes. These high-contention storage mechanisms are often implemented using *multi-ported memories* that allow multiple reads and writes to occur simultaneously. A good example is the register file of an FPGA-based *soft processor*, for which even a simple in-order RISC processor requires one write port and two read ports, while processors that issue instructions more aggressively require even more ports. The challenge is that constructing a multi-ported memory out of FPGA logic elements is inefficient [8]. Furthermore, FPGA substrates typically provide block RAMs (BRAMs) that provide only two ports, and hence memories with more than two ports must be "soft", i.e., constructed using FPGA logic and/or hard BRAMs. However, the ability to construct efficient soft multi-ported memories is important as it frees FPGA vendors from having to include hard BRAMs with more than two ports in their fabrics.

1.1 Prior Approaches

Implementations for FPGA-based multi-ported memories have only recently been formally described and studied [8]; we summarize the conventional approaches here. A straightforward approach is to construct a multi-ported memory using logic elements—for example Altera's adaptive logic modules (ALMs)—enjoying flexibility but at a heavy cost in area and performance. **Replication** enables constructing a memory with any number of external read ports, but can support only a single external write port that must be connected to one of the two ports of each replicated BRAM. **Banking** divides the read and write ports across multiple separate BRAMs, supporting concurrent read and writes but fragmenting and isolating the data across banks. The **Live Value Table** (LVT) approach [8] augments a banked approach with a table that uses output multiplexers to steer reads to the most recently-updated bank for each memory address. The LVT approach improves significantly on the area and speed of comparable designs built using ALMs, although the internal LVT table itself scales somewhat poorly, can consume a lot of area, and usually becomes the critical path. Finally, **Multipumping** can be applied to any memory design to multiply its read and write ports by operating that memory at a multiple of the external clock frequency. Multipumping reduces the area required for a memory with a certain number of ports, but also reduces its maximum achievable external operating frequency.

1.2 An XOR-Based Approach

The XOR operation ($\oplus$) has interesting and useful properties, particularly that $A \oplus B \oplus B = A$. For example, XOR can be used in RAID systems [11] to implement parity and provide data recovery capability should one hard-drive of an array of drives fail. In this paper we present an alternative to the LVT approach that is based on XOR. Similar to the LVT approach, the XOR approach internally uses banking and replication. However, the XOR design avoids the need for a Live Value Table to direct reads and also avoids the corresponding output multiplexing, instead allowing the logic of each read port to consist solely of an XOR of values read from a bank of BRAMs. We demonstrate that XOR designs consume far fewer ALMs but more BRAMs than corresponding LVT designs, and that XOR designs can also be faster and consume less total area than LVT designs for some configurations.

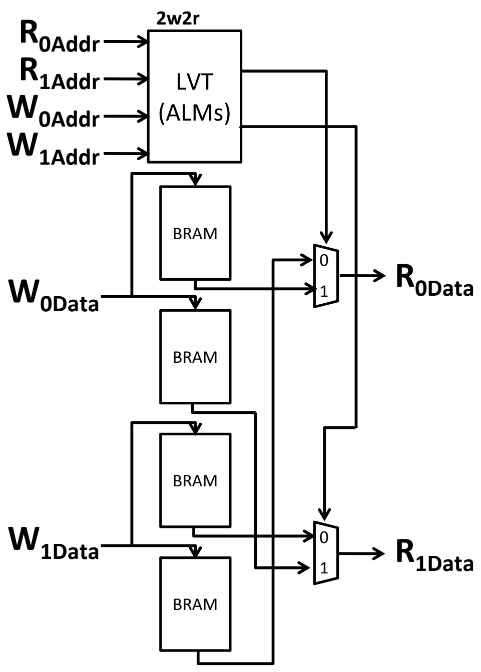


Figure 1: A 2W/2R Live Value Table (LVT) design.

1.3 Contributions

This paper makes the following contributions:

1. we present a novel design for implementing multi-ported memories based on the `XOR` operation;
2. we describe methods to improve the speed of the original LVT design, at an area cost for some designs but an area savings for others;
3. we thoroughly evaluate and compare XOR and improved LVT approaches on an Altera Stratix IV device across a broad range of depths and numbers of ports, as well as multiple factors of multipumping;
4. we demonstrate that the XOR and LVT designs have significant resource diversity, as XOR designs use far fewer ALMs but more BRAMs than LVT designs;
5. we show that both XOR and LVT designs can be the smallest or fastest option, depending on the depth and number of ports of the desired memory.

2. THE LIVE VALUE TABLE DESIGN

The live value table (LVT) multi-ported memory design allows the implementation of a memory with more than one write port to be based on BRAMs, as opposed to being limited to building such a memory solely from logic elements. The basic idea of an LVT design is to augment a banked design with the ability to connect each read port to the most-recently written bank for a given memory location. In this section we briefly summarize the construction and operation of the LVT design; a full treatment is provided by LaForest and Steffan [8].

As a simple example, Figure 1 shows a two-write-two-read (2W/2R) LVT-based memory. Each write port requires its own bank of BRAMs, and each bank requires two BRAMs to provide

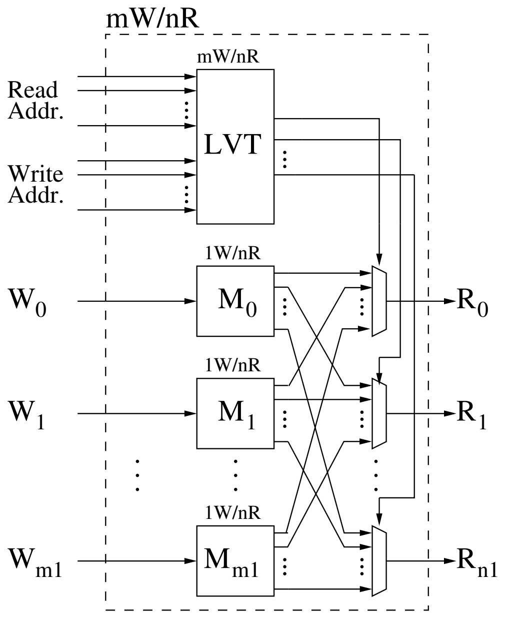


Figure 2: A generalized mW/nR memory implemented using a Live Value Table (LVT)

for the two read ports. At each read port is a multiplexer that is driven by the LVT itself, which selects the most-recently written (i.e., *live*) bank for the read-address. The LVT itself is composed of logic elements (e.g., Altera's ALMs), and itself is a 2W/2R memory, but is only as wide as the log-base-2 of the number of write ports (typically 1-3 bits wide)—and hence can be implemented fairly efficiently.

Figure 2 shows a generalized LVT design. Again, there is a bank of BRAMs per every write port, and each bank is itself a 1W/nR memory composed BRAMs, for a memory with n external read ports. The BRAMs in a bank are arranged using *replication*, where the single write port writes to every BRAM, and the second port for each BRAM is used to provide a read port to the output multiplexers. Hence an LVT design requires a total of $m \cdot n$ BRAMs, plus the logic required to implement the LVT and the multiplexers.

Summary For LVT-based memories, the LVT itself and the output multiplexers together (i) constitute the critical path, and (ii) can require a significant number of logic elements to implement. In the next section we pursue an alternative design that avoids both of these challenges. Later in Section 5 we describe and quantify the ways that we improve on the original LVT design and its multipumped versions to significantly increase its frequency at a small cost in area for shallow designs.

3. AN XOR-BASED DESIGN

In this section we venture to overcome the drawbacks of the LVT design by avoiding the narrow but multi-ported LVT itself required to control the output multiplexers. As introduced earlier, the goal of our design is for each read port to require only the computation of the `XOR` of values read from BRAMs. In this section we review some of the properties of `XOR`, build towards a working XOR-based multi-ported memory design, and then discuss how the XOR-based design can be multipumped to trade speed to save area.

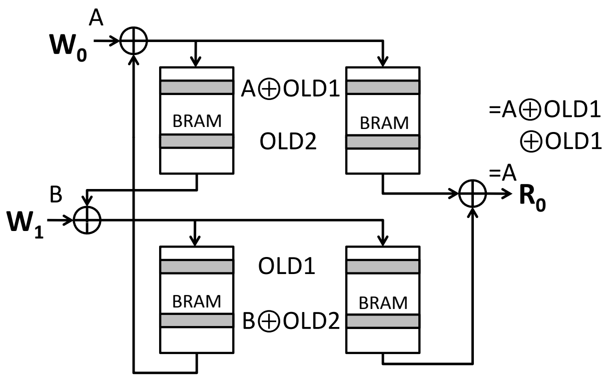


Figure 3: A 2W/1R memory implemented using XOR, with example data values. Note that only data wires are shown, not address wires. W_0 stores the value A XOR'ed with the old contents of the other bank ($OLD1$). Similarly, W_1 stores the value B XOR'ed with the old contents of the other bank ($OLD2$). Reading the location containing A computes ($A \oplus OLD1) \oplus OLD1$ which returns A.

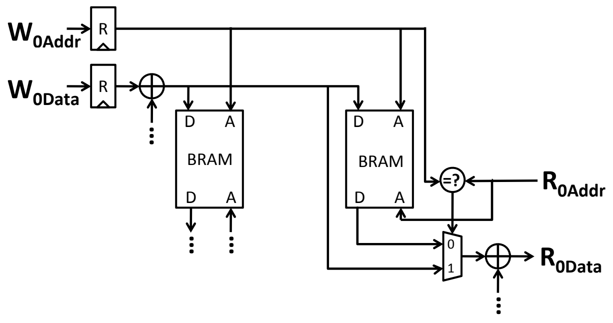


Figure 4: Details of the address wires, registers, and forwarding circuitry used in the XOR design, not shown in other figures for simplicity.

3.1 XOR Properties

The bitwise XOR operation is commutative, associative, and has the following properties[1]:

- $A \oplus 0 = A$
- $B \oplus B = 0$
- $A \oplus B \oplus B = A$

The third property, which follows from the first two, implies that we can XOR two values A and B together, and recover A by XORing the result with B. We can exploit this property to allow the XOR of two instances of a location to return the most recent version. For example, suppose location1 contains some OLD value, and then we save a new value A in location2 by XORing it with the OLD value, i.e., by storing $A \oplus OLD$ in location2; explicitly: $location2 = A \oplus location1 = A \oplus OLD$. We can then recover A, i.e., read the most recently-written value, by simply returning the XOR of the two locations, without having to select between them; explicitly: $output = location1 \oplus location2 = (A \oplus OLD) \oplus OLD = A$. While at first this all seems unnecessary, the key is that it allows two write ports to write two separate BRAMS (or banks of BRAMs) simultaneously (like the LVT design), while read ports need only XOR BRAM locations to return a value (unlike the LVT design which requires output multiplexing).

3.2 Simple XOR Designs

We next build on this basic property of XOR to construct a simple 2W/1R memory out of dual-ported BRAMs, as illustrated in Figure 3. Note that the figure shows only data wires and values, not address wires or values. In the design, each write port has its own bank of two BRAMs, and writes for each are copied to both BRAMs—i.e., corresponding locations in all of the BRAMs in a bank always have the same value. When the write port W_0 stores the value A to the upper locations (in grey), it first XORs A with the old value of the same location in W_1's bank ($OLD1$). Similarly, when the write port W_1 stores the value B to the lower locations (also in grey), it first XORs B with the old value of the same location in W_0's bank ($OLD2$).

For the read port, recall that our main design goal is for the read port circuitry to consist solely of an XOR of BRAM outputs, which is achieved. Reading the upper location computes the XOR of both versions of the upper location (from both the upper and lower banks), which results in isolating the value most recently stored to that location by computing $(A \oplus OLD1) \oplus OLD1$, which returns A.

A challenge for the XOR design is that each write requires reading as well, since we must store the write value XOR the old value of that location from the other bank. This increases the number of BRAMs required to implement the design, since extra read ports must be used internally to service writes. This also potentially complicates the design since writes will effectively take two cycles to complete. However, we can keep the XOR design *black-box-compatible* with previous designs and give the illusion that writes effectively take only one cycle with two additions to the design, as illustrated in Figure 4. First, we register the write port addresses and values. Second, we instantiate forwarding circuitry (via Quartus library) that allows the write data value to flow directly to the read data wires in the event that we read a location in the cycle directly after we write that same location. For simplicity these extra registers and forwarding logic are not shown in figures other than Figure 4.

As shown in Figure 5, we next build a slightly more complex memory by adding another read port, constructing a 2W/2R memory. This design functions similarly to the 2W/1R design, except that another column of BRAMs has been added to provide values for the additional read port.

3.3 A Generalized XOR Design

To summarize the XOR design, each time we write the contents of a given location to a particular memory bank, we XOR the new data with the old contents of the same location from all other banks. To read a location we calculate the XOR of the values for that location across all banks, which recovers the latest value written.

In Figure 6 we present a generalized mW/nR XOR design. Each write port has its own bank (row) of BRAMs. To write a value to a location, that value is first XORed with all of the values for that same location from all other banks, and the result of that XOR is then distributed and written to all BRAMs in the current bank. Hence

[1]Another interesting use of XOR is to swap the contents of two memory locations A and B without the use of a temporary location in three steps: $A = A \oplus B$; $B = A \oplus B$; $A = A \oplus B$.

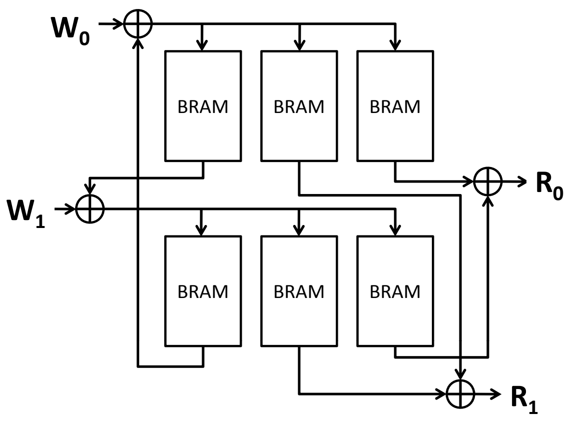


Figure 5: A 2W/2R memory implemented using `XOR`**. Compared with the 2W/1R memory in Figure 3, an additional column of BRAMs is added to supply values for the additional read port.**

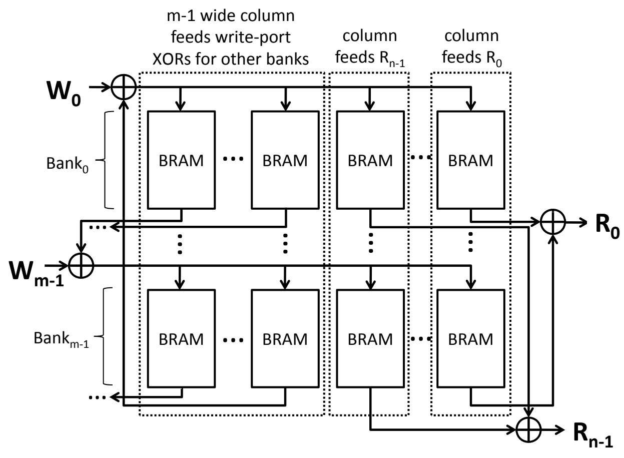


Figure 6: A generalized mW/nR memory implemented using `XOR`**. Each write port requires a bank (row) of BRAMs. A column of BRAMs is required for each read port, as well as a column for each of one less than the total number of write ports.**

the design requires a column of BRAMs that is as wide as one less than the number of write ports, to provide sufficient internal read ports to support writing. Furthermore, a column of BRAMs is required for each external read port. In summary, the XOR design requires $m * (m - 1 + n)$ BRAMs to provide m writes and n reads.

3.4 Multipumping the XOR Design

For any memory design we can trade speed for area by operating the memory at an internal clock frequency that is a faster multiple of the external clock frequency, giving the illusion of having more ports than are actually supported. We can apply this multipumping [8] to either read or write ports, but not both at once. Doing so interleaves the internal reads and writes and breaks the external appearance of reads logically occurring before writes within the same clock cycle. Note that relaxing this constraint would lead to a reduction in hardware (Section 6.3), but at the cost of placing the burden of scheduling reads and writes onto the enclosing system.

Multipumping the write ports provides the best speed/area tradeoff, resulting in (i) fewer memory banks (Figure 6), which leads to (ii) shallower columns for each read port, and (iii) fewer columns to feed the `XOR` logic of other write ports. The resulting multipumped XOR design requires $(m/f) * ((m/f) - 1 + n)$ BRAMs, for a given multipumping factor $f \leq m$ (assuming that both f and m are powers of two).

In operation, a multipumped XOR-based memory performs all reads and the first subset of writes on the first *internal* clock cycle, then performs the remainder of the writes afterwards with all operations completing within one *external* clock cycle. For example, with 2x multipumping, all reads and the first half of all writes happen during the first internal cycle, while the second half of the writes happen during the second internal cycle, whose end coincides with the end of a single encompassing external cycle. Since no read happens after a write, a read always returns the current memory contents as opposed to returning a value being written.

Note that in this paper we only consider even factors of multipumping—e.g., 2x and 4x, which means that for every external cycle there are two or four internal cycles. Note also that a design can be multi-pumped to the point where there are as many internal cycles as there are external write ports, meaning that the internal memory requires only one write port, which does not require the XOR mechanism to function—i.e., one write port and N read ports are trivially supported via replication only. Since such designs are not XOR-based (nor LVT-based), we instead call them *fully multipumped*.

4. EXPERIMENTAL FRAMEWORK

In this section we describe our experimental framework. We evaluate the designs on Altera Stratix IV FPGAs, although we expect similar results on comparable quality-grade Xilinx FPGAs. We provide details on Stratix IV BRAMs, the memory designs under study, our CAD flow, and our method for measuring speed and area.

Stratix IV Block RAM (BRAM) Memory Modern FPGAs often implement BRAMs directly on their silicon substrate. These BRAMs typically have two ports that can each function either as a read or a write port. BRAMs use less area and run at a higher frequency than ones created from the FPGA's reconfigurable logic, but do so at the expense of having a fixed storage capacity and number of ports. The Stratix IV FPGAs mostly contain M9K block RAMs[2], which hold nine kilobits of information at various widths and depths. At a width of 32 bits, an M9K holds 256 elements.

CAD Flow We use Altera's Quartus 10.0 to target the Stratix IV `EP4SE530H40C2` FPGA, a device of the highest available speed grade and containing 1280 M9K BRAMs. We implement all the designs in generic Verilog-2001 without any Altera-specific modules. We place our circuits inside a synthesis test harness designed to both: (i) register all inputs and outputs to ensure an accurate timing analysis, and (ii) to reduce the number of I/O pins to a minimum as larger circuits will not otherwise fit on the FPGA. The test harness also avoids any loss of circuitry caused by I/O optimization. Shift registers expand single-pin inputs, while

[2]These FPGAs also contain M144K and MLAB memories. There are too few M144Ks to fully explore the design space and past work demonstrated that MLABs scale very poorly [8].

registered AND-reducers compact word-wide signals to a single output pin.

We configured the synthesis process to favour speed over area, and enabled all relevant optimizations, including circuit transformations such as register retiming. The impact on area of register retiming varies, depending on the logic found beyond the I/O registers, so the absolute results presented here might not appear in a real system. However, comparing our designs in a real system would yield proportionally similar results. We tested all designs inside identical test harnesses.

We configured the place and route process to make a standard effort at fitting with only two constraints: (i) to avoid I/O pin registers to prevent artificially long paths that would affect the clock frequency, and (ii) to set the target clock frequency to 550MHz, which is the maximum clock frequency specified for M9K BRAMs. Setting a higher target F_{max} does not improve results, and may in fact worsen them if a slower, derived clock exists and thus aims towards an unnecessarily high target frequency, causing competition for fast paths. We assume all clocks to be externally generated and any of their fractions (e.g., half-rate) used in multipumping designs are assumed to be synchronous to the main system clock (i.e., when generated by a PLL).

We report maximum operating frequency (F_{max}) by averaging the results of ten place and route runs, each starting with a different random seed for initial placement. We select the worst-case F_{max} report for the default range of die temperatures of 0 to 85°C. Area does not vary significantly between place and route runs, so we report the first computed result.

Measuring Area When comparing designs as a whole, we report area as the *total equivalent area* (TEA), which estimates the actual silicon area of a design point: we calculate the sum of all the Adaptive Logic Modules (ALMs) used partially or completely, plus the area of the BRAMs *counted as their equivalent area in ALMs*. A Stratix IV ALM contains two Adaptive Lookup Tables (ALUTs), each roughly equivalent to a 6-LUT, two adder and carry-chain stages, and two flip-flops. Wong *et al.* [14] provide the raw layout area data: one M9K has an area equivalent to 28.7 ALMs. This value became known only after publication by LaForest *et al.* [8], which used an estimate.

Designs Considered For simplicity, we consider only the common case of 32-bit-wide memories. We do not consider one-write-one-read (1W/1R) memories as they directly map to a single FPGA BRAM. Similarly, replication trivially enables 1W/nR memories. The challenge lies in creating concurrent multiple *write ports*. We evaluate a representative sample of the range of multi-ported memory configurations with two to eight write ports and four to 16 read ports: 2W/4R, 4W/8R, and 8W/16R. We consider these configurations over memory depths of 32 to 8k entries; some designs with more than 2k entries begin to consume a significant fraction of the large Altera device that we target and hence would likely be impractically large for current-day applications. Although the internal implementations vary, we ensure that all designs are "black-box equivalent" from an outside point of view. Specifically, all ports are unidirectional (read or write only), usable simultaneously within a single external clock cycle, and any latencies between values being written and subsequently readable are equal across designs. Any one design can substitute for another within a system, with clock frequency and resource usage being the only differences. We do not consider memories that may stall (e.g., take multiple cycles to perform a read or write if there exists a resource conflict), although such designs would be compelling future work. Finally, we assume that multiple simultaneous writes to the same address result in undefined behavior and are thus avoided by the enclosing system.

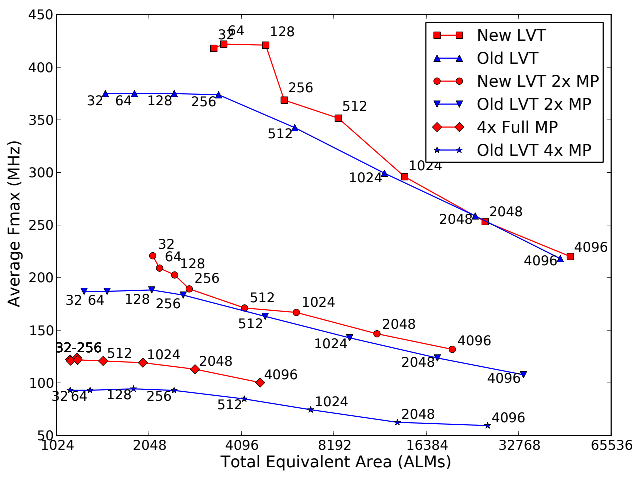


Figure 7: Speed and area of both the original (old) and improved (new) 4W/8R LVT designs. MP means multipumped.

5. IMPROVEMENTS TO THE LVT DESIGN

For our evaluation and comparison with LVT (later in Section 6), we make two significant improvements to the original LVT design [8]: (i) adding forwarding logic, and (ii) multipumping write ports.

Forwarding Similar to that described earlier for the XOR design (Figure 4), we instantiate forwarding logic around the BRAMs of the LVT design to increase clock frequency for an area cost. Recall that the forwarding logic bypasses the BRAMs such that if a location being written is read during the same cycle, the read will return the new write value (as opposed to the old stored value). To remain compatible with the expected behavior of a one cycle read-after-write latency, we also register write addresses and data. This modified design increases the maximum operating frequency of the BRAMs (including forwarding) from 375MHz to 550 MHz, since there is a frequency cost to the Stratix IV implementation of having a BRAM set to return old data during simultaneous read/write of the same location [2]. Overall this makes the modified LVT designs more competitive.

Multipumping As discussed earlier in Section 3.4, for any multi-ported memory design we can exploit multipumping to trade speed for area savings. Details for how to multipump the LVT design are given by LaForest and Steffan [8]—however their described design multipumps across read ports, while we found it to be an improvement to instead multipump across write ports and hence do so for the LVT implementations in this paper. Furthermore, when the multipumping factor equals the number of external write ports, the number of internal write ports reduces to one, eliminating the need for the LVT circuitry entirely, further saving area and gaining speed—as mentioned previously, we call such designs *fully multipumped*.

Impact Figure 7 shows the speed and area impact of our modifications to the LVT design. For small memories (32-128 entries deep), the new LVT is significantly faster although those are also larger than the corresponding old versions. For the 2x-multipumped memories, all memories are faster than the corresponding original

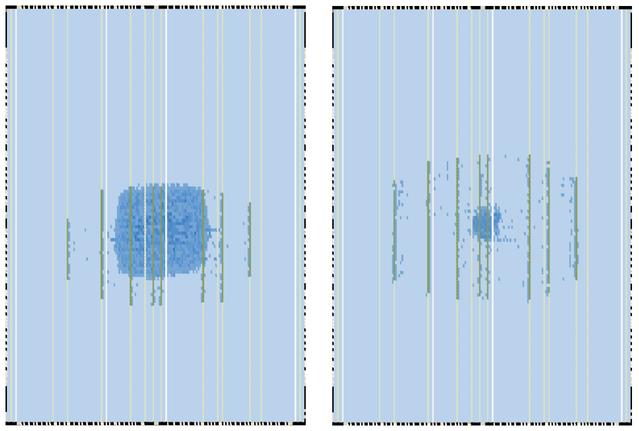


Figure 8: Circuit layout of 8192-deep 2W/4R memories for (a) LVT and (b) XOR designs, as rendered by Quartus. The thin columns are BRAMs or DSPs (darkened indicates in-use), the dots and dot-clouds are ALMs.

versions, and memories deeper than 256 entries are also smaller. For the 4x-multipumped memories, all memories are both faster and smaller than the original versions. Overall, while there are some trade-offs between area and speed that are apparent for some design points, for simplicity and to focus more on speed than area, we concentrate on the new LVT designs for the remainder of this paper.

6. COMPARING LVT VS XOR

In this section we compare the LVT and XOR approaches to implementing multi-ported memories. We begin by visualizing the resource usage and layout of an example of each design. Next we compare in detail the speed and resource usage of a broad range of memories of varying depths and numbers of ports, and then investigate the impact of multipumping all designs. We summarize the design space by highlighting the designs that minimize delay, ALM usage, or BRAM usage.

6.1 Visualizing Layout

To help visualize the resource diversity of the LVT and XOR approaches, in Figure 8 we present the circuit layout of 8192-deep 2W/4R memories for both XOR and LVT designs, as rendered by Quartus. The thin columns represent BRAMs or DSPs, where darkened areas indicate that the BRAMs are being used, and the dots and dot-clouds represent ALMs. We chose 8192-deep memories because they are large relative to the capacity of the chip and emphasize the differences between the designs. Both designs consume resources in a somewhat circular pattern, due to Quartus' efforts to minimize delay and resource consumption. Looking at BRAM usage, the designs are both the same width, but the XOR design consumes more BRAMs and hence has taller columns of in-use BRAMs. For both designs the ALMs used are clustered in the center, but the LVT design consumes far more ALMs than the XOR design. Considering that each multi-ported memory would be inserted into a larger design, one can visualize how the XOR memory would integrate better with an enclosing design that consumes many ALMs, and the LVT design would integrate better with an enclosing design that demands many BRAMs.

6.2 Varying Depths and Numbers of Ports

In Figure 9 we compare LVT and XOR implementations of 2W/4R, 4W/8R, and 8W/16R memories, with depths varying from 32 entries up to 8192, 4096, and 1024 entries respectively—the memories with more ports exhaust the available BRAMs with fewer entries than the 2W/4R memories. For the figures on the left we plot the average *unrestricted* maximum operating frequency (Fmax)[3] versus area. In particular we report the *total equivalent area* (TEA) in terms of ALMs, that accounts for both ALMs and BRAMs used, as described in Section 4. Note that the x-axis (TEA) is logarithmic.

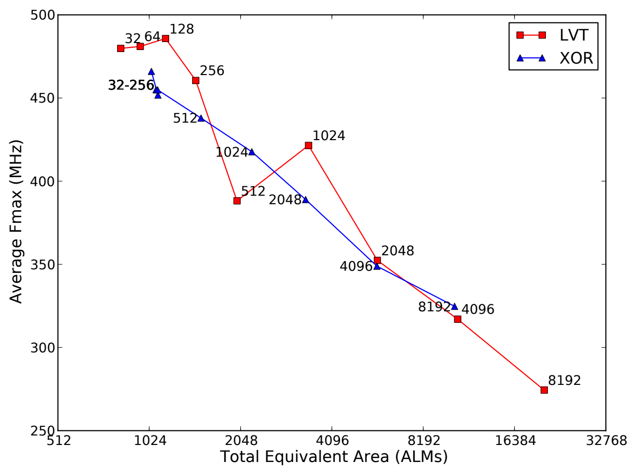
Fmax vs Area Looking at the results for the 2W/4R memories (Figure 9(a)), we observe first that the LVT designs are superior for both Fmax and TEA for 32 and 64 entries. For designs with more than 64 entries the XOR designs consume less TEA, with the relative savings increasing with the number of entries: e.g., for 8192-entry memories the XOR design is 51% of the area of the LVT design. This TEA difference persists in 4W/8R and 8W/16R memories, with the area advantage going to XOR designs with more than 256 entries.

However, both the XOR and LVT designs trigger anomalies in Quartus (despite averaging these results over 10 seeds as described in Section 4): for example, for LVT 2W/4R designs the 128-entry design is faster than the 32-entry design, and the 1024-entry design is significantly faster than the 512-entry design, contrary to the general trend of Fmax linearly decreasing as memory depth increases. Similarly, the 512-entry LVT 8W/16R design has virtually the same Fmax as the 256-entry design, contrary to expectations since going from 256 to 512 entries doubles the number of BRAMs required. Finally, the smallest 4W/8R XOR designs (< 512 entries) see-saw between higher and lower TEAs and Fmax in a manner seemingly unrelated to the memory depth, as does the Fmax of the 64-entry 8W/16R XOR design.

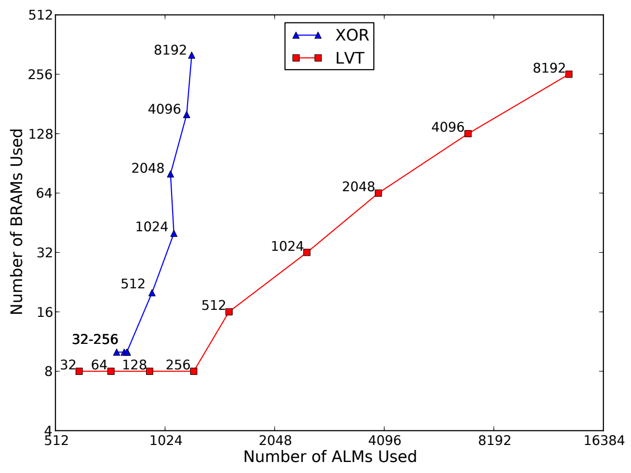
Disregarding the anomalies, it is apparent that for smaller designs the LVT approach is generally faster or equally-fast as the corresponding XOR designs, while deeper designs save more area and gain more speed from the XOR approach.

BRAMs vs ALMs In the figures on the right of Figure 9, we expand on our TEA metric to view the actual numbers of BRAMs and ALMs consumed by each design—note that both axes are logarithmic. Looking at Figure 9(a) as an example, we first note that the resource diversity of the two designs is clearly evident, with XOR designs consuming far fewer ALMs but more BRAMs than the corresponding LVT designs. *Hence the relative availability of ALMs or BRAMs in a given encompassing design plays a large role in the selection of the best choice of multi-ported memory implementation.* Looking in more detail, we observe that the number of BRAMS used is constant for both designs for 32-256 entries, reflecting the native capacity of each BRAM. The number of ALMs used by XOR memories grows very slowly as memory depth increases, since for the XOR design ALMs are used only to implement the `XOR` operations and forwarding logic, both of which grow linearly with memory width and only logarithmically with memory depth. In contrast, the number of ALMs used by LVT memories grows with memory depth since (i) they are used to construct the LVT itself, which grows significantly for deep and/or many-ported memories, and (ii) they implement forwarding logic, which grows with the number of BRAMs used. Due to the extra replicated memories required to support the write port `XOR` operations, XOR designs consume more BRAMS than the corresponding LVT designs—e.g., for the 8192-entry memories the XOR design consumes 25% more BRAMs than the LVT design.

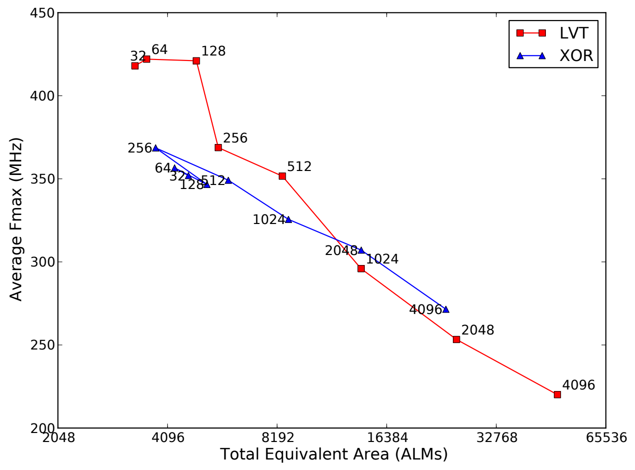
[3]Note that a minimum clock pulse width requirement for the BRAMs restricts the actual Fmax to 550MHz on Stratix IV devices.



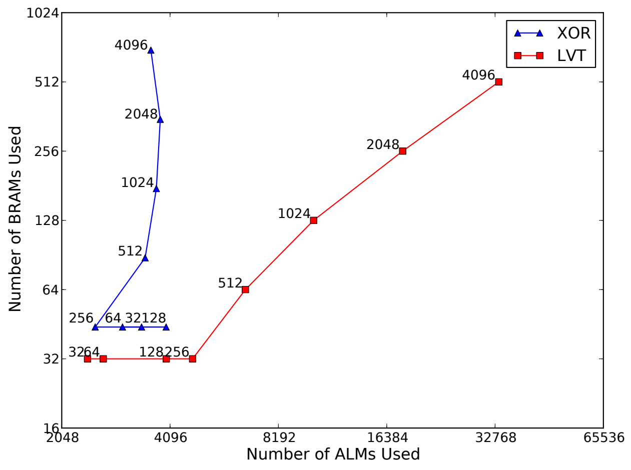
(a) 2W/4R: Fmax vs. Total Equivalent Area



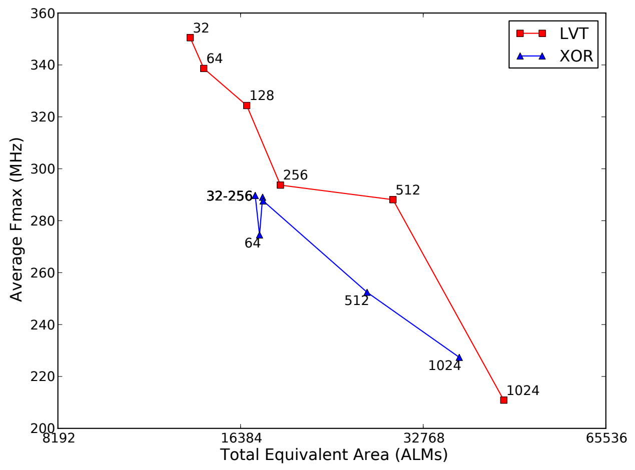
(b) 2W/4R: BRAMs used vs. ALMs used



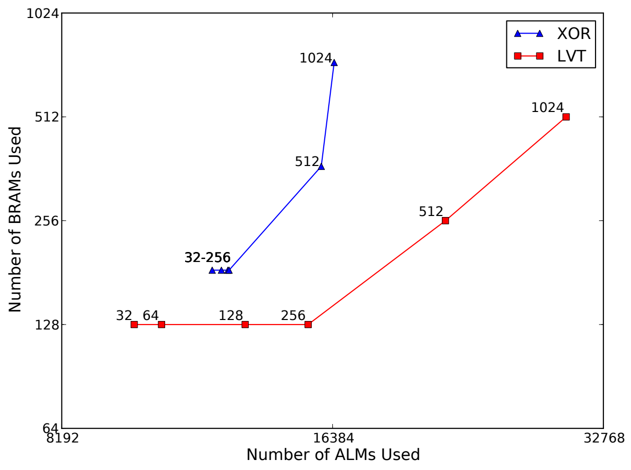
(c) 4W/8R: Fmax vs. Total Equivalent Area



(d) 4W/8R:BRAMs used vs. ALMs used



(e) 8W/16R: Fmax vs. Total Equivalent Area



(f) 8W/16R: BRAMs used vs. ALMs used

Figure 9: Speed and area for LVT and XOR implementations of 2W/4R, 4W/8R, and 8W/16R memories of increasing depth. For each we show average Fmax vs total equivalent area (on the left), as well as BRAMs vs ALMs (on the right).

Depth	Design that minimizes:		
	Delay	ALMs	BRAMs
32	Equal	LVT	LVT
64	LVT	LVT	LVT
128	LVT	XOR	LVT
256	Equal	XOR	LVT
512	XOR	XOR	LVT
1024	Equal	XOR	LVT
2048	XOR	XOR	LVT
4096	XOR	XOR	LVT
8192	XOR	XOR	LVT

(a) 2W/4R

Depth	Design that minimizes:		
	Delay	ALMs	BRAMs
32	LVT	LVT	LVT
64	LVT	LVT	LVT
128	LVT	Equal	LVT
256	Equal	XOR	LVT
512	Equal	XOR	LVT
1024	XOR	XOR	LVT
2048	XOR	XOR	LVT
4096	XOR	XOR	LVT

(b) 4W/8R

Depth	Design that minimizes:		
	Delay	ALMs	BRAMs
32	LVT	LVT	LVT
64	LVT	LVT	LVT
128	LVT	Equal	LVT
256	Equal	XOR	LVT
512	LVT	XOR	LVT
1024	XOR	XOR	LVT

(c) 8W/16R

Figure 10: For each memory depth, listed is the design that minimizes delay (i.e., has the highest Fmax), the number of ALMs used, or the number of BRAMs used, for (a) 2W/4R, (b) 4W/8R, and (c) 8W/16R memories. Results within 5% are considered "Equal".

Increasing Ports Figures 9(c), (d), (e), and (f) plot the results for 4W/8R and 8W/16R memories, which at a high level show similar trends as the 2W/4R memories, but with some notable differences. As the number of ports increases, so does the frequency gap between LVT and XOR designs, with XOR designs becoming comparatively slower. The TEA advantage of XOR designs also diminishes: e.g., for the 1024-entry 8W/16R memories the XOR design is only 18% smaller. Looking at Figure 9(f), we see that the XOR designs now consume a more significant number of ALMs to support `XOR` operations and forwarding logic. The area of these functions increases in proportion to the product of the number of read and write ports: e.g., going from 2W/4R to 4W/8R quadruples the number of ALMS required by XOR designs of the same depth.

Navigating the Design Space From the point of view of a system designer, a key question is "*What is the best memory design to use given my constraints?*". To summarize the design space we list in Figure 10 the design that, for each memory depth, minimizes delay (i.e., has the highest Fmax), the number of ALMs used, or the number of BRAMs used, displayed for (a) 2W/4R, (b) 4W/8R, and (c) 8W/16R memories. Note that any results within 5% of each other are considered to be roughly equal due to normal CAD variation and are labeled as such. First, we note that the LVT designs always use the least BRAMs, regardless of depth or number of ports. In terms of ALM use, designs with 64 or fewer entries are most efficiently implemented via the LVT approach, and designs with 256 or greater entries are most efficiently implemented via the XOR approach. For 128 entries the designs have roughly equivalent ALM usage, although for the 128-entry 2W/4R memory the XOR design has a greater than 5% advantage. In terms of maximizing Fmax, the LVT design is generally faster for shallower memories while the XOR design is faster for deeper memories. The crossover point is around 256-1024 entries, depending on the number of ports, and obscured somewhat by the previously-discussed CAD anomalies.

6.3 Impact of Multipumping

Figure 11 shows the average *unrestricted* Fmax vs TEA for LVT and XOR implementations of (a) 2W/4R, (b) 4W/8R, and (c) 8W/16R memories of increasing depth, including the 2x and 4x multipumped (MP) and fully-multipumped (Full MP) designs. Looking first at Figure 11(a) for 2W/4R memories, since these designs have only two write ports, multipumping by 2x results in a fully-multipumped design (rather than an LVT or XOR design). The resulting impact on speed and area is as expected, resulting in memories that are smaller but slower: for example, the 32-entry fully-multipumped memory is 53% of the speed but also 48.7% fewer equivalent ALMs than the 32-entry LVT memory.

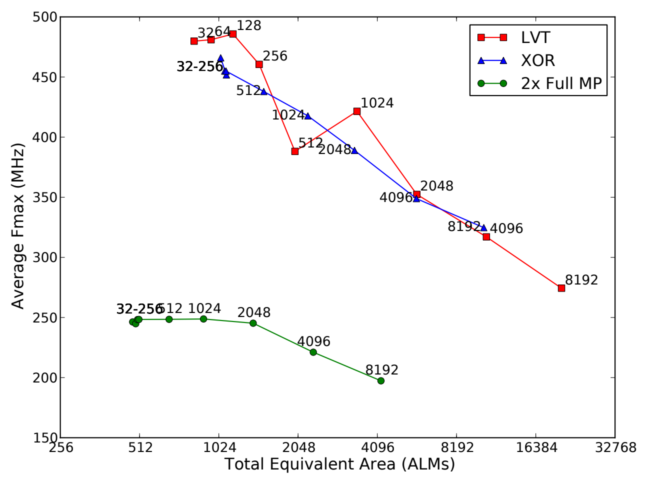
The value of considering both LVT and XOR designs is more apparent from Figures 11(b) and (c), where considering the multipumping factor and choice of LVT vs XOR implementation provides designers with a significant range of options in the speed vs area trade-off space. Looking at Figure 11(b) for 4W/8R memories and focusing on the 2x multipumped (2x MP) designs, we observe that LVT designs are faster but significantly larger than their XOR counterparts. At the extreme, the 4096-entry XOR memory consumes 46.4% of the equivalent ALMs of the 2048-entry LVT design, even though the LVT design has half of the entries. The fully-multipumped design (Full MP) further repeats this significant trade-off between speed and area. Multipumping provides a greater relative area savings for XOR designs than for LVT designs: for XOR designs multipumping reduces the number of replicated BRAMs to a greater extent and correspondingly reduces the `XOR` logic. The range of possibilities provided by these design permutations is significant: for example, for a 4096-entry 4W/8R memory, the design options range from 271MHz using 23,828 equivalent ALMs all the way to 104MHz using 4,875 equivalent ALMs.

For Figure 11(c) for 8W/16R memories we see a similarly-impressive range of design possibilities. Note that we did not include the fully-multipumped implementation for this design as its resulting Fmax is unusably low (i.e., less than 50MHz).

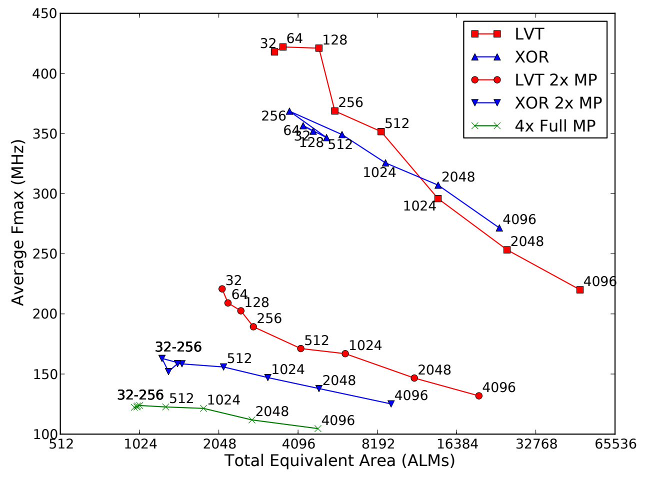
Navigating the Design Space As in the previous section, it is important to have a summary view of the best designs for given constraints. For the multipumped designs, namely the 2x multipumped 4W/8R memories and the 2x and 4x multipumped 8W/16R memories, the result is straightforward: to minimize delay (i.e., maximize Fmax) or to minimize BRAM usage, use the LVT designs; to minimize ALM usage, use the XOR designs.

7. RELATED WORK

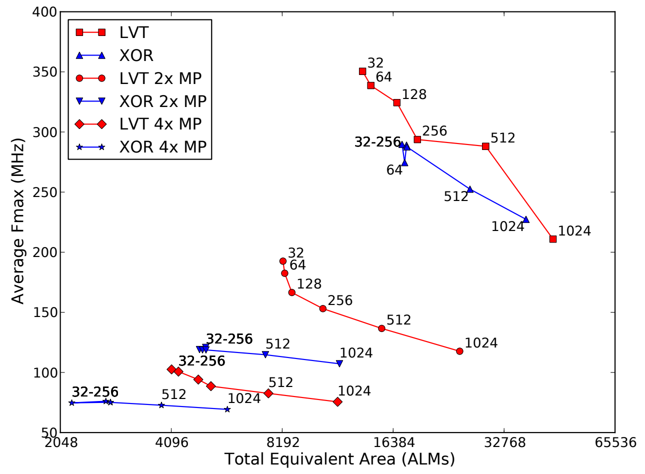
Most prior work on multi-ported memories for FPGAs focuses on register files for soft-processors. Simple replication provides the 1W/2R register file required to support a three-operand ISA [1, 5, 6, 10, 15]. Jones *et al.* [7] implement a VLIW soft processor with a multi-ported register file implemented entirely in logic elements, limiting the operating frequency. Saghir *et al.* [12, 13] also implement a multi-ported register file for a VLIW soft-processor, but use replication and banking of BRAMs; however, their compiler must schedule register accesses to avoid conflicting reads and writes. Manjikian aggressively multipumps memories by performing reads and writes on consecutive rising and falling clock edges within a processor cycle [9]—unfortunately, this design forces a multiple-phase clock on the entire system. More recently, Anjam *et al.* [3] successfully use a LVT-based register file for their reconfigurable



(a) 2W/4R: Fmax vs. Total Equivalent Area



(b) 4W/8R: Fmax vs. Total Equivalent Area



(c) 8W/16R: Fmax vs. Total Equivalent Area

Figure 11: Fmax vs TEA for LVT and XOR implementations of 2W/4R, 4W/8R, and 8W/16R memories of increasing depth, including multipumped designs.

VLIW soft-processor and add one more addressing bit internally to enable splitting a 4W/8R register file into two independent 2W/4R instances. Later work by Anjam *et al.* [4] removes the need for the LVT by avoiding write bank conflicts via compile-time register renaming, but this solution requires more registers than are architecturally visible.

8. CONCLUSIONS

In this paper we introduced an approach to implementing multi-ported memories—composed of the two-ported block RAMs provided on FPGAs—that exploits the properties of the XOR operation to eliminate both the output multiplexing and logic-based lookup table required by the best prior approach, the Live Value Table (LVT) [8]. Targeting an Altera Stratix IV FPGA, we compared over 100 designs that implement both the LVT and XOR approaches and span a broad range of memory depths and numbers of ports. We found that:

- using forwarding logic improves the maximum speed of LVT designs;
- for both XOR and LVT designs, multipumping the write ports instead of the read ports yields a greater reduction in area;
- the XOR designs use far less logic but more block RAMs than the LVT designs, demonstrating a resource diversity between the two designs that makes them each desirable for different use-cases;
- for shallower designs the LVT approach is generally faster or equally-fast as the corresponding XOR designs, while the XOR designs with more than 64-entries are smaller, and with more than 512 entries are also faster than the corresponding LVT designs;
- exploiting multipumping greatly expands the possible design space with a large range of area and speed trade-offs;
- and multipumped XOR designs are significantly smaller but slower than their LVT counterparts.

To summarize, both the LVT and XOR approaches are valuable and useful in different situations, depending on the constraints and resource utilization (block RAMs vs logic) of the enclosing design. Designers can use the results of this work as a guide when choosing an appropriate design to implement a multi-ported memory.

9. FUTURE WORK

The results of this paper suggest several compelling avenues for future work.

Port to Xilinx FPGA Devices Although the design principles of XOR and LVT memories are generic and applicable to any FPGA device with block RAMs, we have only measured their frequencies and resource usage on Altera FPGAs. These CAD results and hence the trade-off space could be different for Xilinx or other FPGA devices.

Pure Multi-Pumping inside XOR Memory We could reduce the number of BRAM columns and thus the amount of XOR logic required for the XOR design by using the BRAMs in *True-Dual-Port* mode (where each port can perform either a read or a write each cycle), and then multipumping them to appear as a 1W/2R memory each—although True-Dual-Port mode suffers a significant reduction in clock frequency.

Different Ratios of Read/Write Ports For this paper we focused on memories with twice as many read ports than write ports, which are common for many applications such as processor register files. Memories with other ratios of read/write ports should be studied.

Stalling Designs For this paper we also focused on memories that do not stall, meaning that all read and write ports complete their requests every cycle. Designs that trade this restriction for area savings or speed, for example by having reads that may take multiple cycles to service, would also be interesting to investigate.

Power Analysis Compared to XOR memories, LVT memories (i) use fewer BRAMs but more logic, and (ii) access BRAMs with different read/write patterns; hence it would be interesting to determine how the dynamic power power consumption of the two approaches compares.

10. ACKNOWLEDGMENTS

The authors thank Altera and NSERC for financial support. We also thank Vaughn Betz, Jonathan Rose, and Ketan Padalia for help with tuning Quartus and our test harness, and the anonymous reviewers for constructive feedback.

11. REFERENCES

[1] Nios II Processor Reference Handbook. `http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf`, March 2009. Version 9.0, Accessed Sept. 2009.

[2] DC and Switching Characteristics for Stratix IV Devices. `http://www.altera.com/literature/hb/stratix-iv/stx4_siv54001.pdf`, June 2011. Version 5.1, Accessed Aug. 2011.

[3] F. Anjam, M. Nadeem, and S. Wong. A vliw softcore processor with dynamically adjustable issue-slots. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 393 –398, dec. 2010.

[4] F. Anjam, S. Wong, and F. Nadeem. A multiported register file with register renaming for configurable softcore vliw processors. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 403 –408, dec. 2010.

[5] R. Carli. Flexible MIPS Soft Processor Architecture. Technical report, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, June 2008.

[6] B. Fort, D. Capalija, Z. Vranesic, and S. Brown. A Multithreaded Soft Processor for SoPC Area Reduction. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 131–142, April 2006.

[7] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster. An FPGA-based VLIW processor with custom hardware execution. In *International Symposium on Field-Programmable Gate Arrays*, 2005.

[8] C. E. LaForest and J. G. Steffan. Efficient Multi-ported Memories for FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 41–50, New York, NY, USA, 2010. ACM.

[9] N. Manjikian. Design Issues for Prototype Implementation of a Pipelined Superscalar Processor in Programmable Logic. In *PACRIM 2003: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, volume 1, pages 155–158 vol.1, Aug. 2003.

[10] R. Moussali, N. Ghanem, and M. A. R. Saghir. Supporting multithreading in configurable soft processor cores. In *CASES '07: Proceedings of the 2007 international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 155–159, New York, NY, USA, 2007. ACM.

[11] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988.

[12] M. Saghir and R. Naous. A Configurable Multi-ported Register File Architecture for Soft Processor Cores. In *ARC 2007: Proceedings of the 2007 International Workshop on Applied Reconfigurable Computing*, pages 14–25. Springer-Verlag, March 2007.

[13] M. A. R. Saghir, M. El-Majzoub, and P. Akl. Datapath and ISA Customization for Soft VLIW Processors. In *ReConFig 2006: IEEE International Conference on Reconfigurable Computing and FPGAs*, pages 1–10, Sept. 2006.

[14] H. Wong, V. Betz, and J. Rose. Comparing fpga vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 5–14, New York, NY, USA, 2011. ACM.

[15] P. Yiannacouras, J. G. Steffan, and J. Rose. Application-specific customization of soft processor microarchitecture. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field Programmable Gate Arrays*, pages 201–210, New York, NY, USA, 2006. ACM.