Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2015, Article ID 826283, 12 pages
http://dx.doi.org/10.1155/2015/826283

Research Article

A Scalable Unsegmented Multiport Memory for FPGA-Based Systems

Kevin R. Townsend, Osama G. Attia, Phillip H. Jones, and Joseph Zambreno

Reconfigurable Computing Laboratory, Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA

Correspondence should be addressed to Joseph Zambreno; zambreno@iastate.edu

Received 30 June 2015; Revised 13 October 2015; Accepted 10 December 2015

Academic Editor: Miriam Leeser

On-chip multiport memory cores are crucial primitives for many modern high-performance reconfigurable architectures and multicore systems. Previous approaches for scaling memory cores come at the cost of operating frequency, communication overhead, and logic resources without increasing the storage capacity of the memory. In this paper, we present two approaches for designing multiport memory cores that are suitable for reconfigurable accelerators with substantial on-chip memory or complex communication. Our design approaches tackle these challenges by banking RAM blocks and utilizing interconnect networks which allows scaling without sacrificing logic resources. With banking, memory congestion is unavoidable and we evaluate our multiport memory cores under different memory access patterns to gain insights about different design trade-offs. We demonstrate our implementation with up to 256 memory ports using a Xilinx Virtex-7 FPGA. Our experimental results report high throughput memories with resource usage that scales with the number of ports.

1. Introduction

Following Moore's Law, transistor densities on FPGAs continue to increase at an exponential rate [1]. This allows for increasingly complex System-on-Chip implementations that require high throughput communication and shared memory on networks of distributed compute nodes. ASIC designs used as accelerators in the field of high-performance computing often utilize multiport memories for communication. By comparison, FPGA vendors do not provide scalable multiport memories in their fabric, with most device families being limited to using dual-port memories [2, 3]. In these cases designers must use a multiport memory constructed with FPGA logic and RAM blocks. These soft IP cores consume significant resources if the memory must behave identically (in terms of performance) as an ASIC equivalent. As an alternative design strategy, if the multiport memory can stall, exhibit variable multicycle latencies, and not have strict memory coherence, substantially fewer resources can be used. Heavily pipelined processing elements can often tolerate these restrictions.

In this paper, we present two FPGA-based multiport memory designs that allow for scalability in terms of the number of ports as well as the addressable memory space. By providing a banked RAM block architecture, our designs allow for implementations that support up to 256 ports and 1 MB of memory space on current-generation FPGA devices. In contrast to previous banked implementations in the research literature, our multiport memories utilize buffering and reordering of memory requests. This results in throughput that approaches the ideal-case throughput, while providing an unsegmented address space. An unsegmented address space allows for simpler integration with the rest of an accelerator implementation. This buffering and unsegmented address space does introduce issues including nonideal throughput, variable multicycle latency, and a lack of memory coherence across different ports. We attempt to minimize these issues.

The rest of this paper is organized as follows. Section 2 provides an overview of related work in the field of multiport memory design. Section 3 presents our two designs (the fully connected and Omega memories), which provide for an explicit trade-off between implementation complexity and achievable throughput. Section 4 discusses our evaluation methodology. Section 5 follows with an analysis of resource usage and performance. Finally, Section 6 concludes with a detailed view towards planned future work.

2. Related Work

If a multiport memory only requires a small amount of memory space, one can synthesize the multiport memory using only FPGA logic resources, as seen in [4]. Otherwise, soft multiport memories utilize the dual-port RAM blocks available on most FPGAs. A review of the research literature illustrates the four design strategies using dual-port RAM blocks: *multipumping*, *replication*, *LVT/XOR*, and *banking*.

Multipumping, seen in [5–7], gains ports by using an internal clock and an external clock, with a clock speed of a constant multiple slower than the internal clock. This way the RAM block can process the requests of multiple ports. This approach limits the number of ports, as each added port decreases the maximum clock frequency (as seen in Figure 3).

Replication, seen in [8–10], gains read-only ports by connecting the write ports of multiple RAM blocks together. This approach does not sacrifice clock speed or FPGA logic resources. However, each extra RAM block just provides one read-only port. Also, increasing the RAM blocks does not expand the storage space of the memory, since the data is explicitly replicated among the blocks.

LVT/XOR, seen in [11–13], gains ports by using a quadratically growing number of RAM blocks. Live value table (LVT) is the first published paper using this method [11]. An LVT-based memory with M write port and N read ports requires $M \times N$ RAM blocks. Similar to replication, live value table connects the write ports of several RAM blocks. A write request writes to N RAMs. A write on a different port writes to N different RAMs. During a read M RAM blocks are read from. The most recent of the M values is valid and is outputted. The "live value table" is a module that keeps track of the most recent values.

The "live value table" module can be eliminated by using XORs. The idea is that by storing the new data XORed with the old data on other rows (e.g., new $\oplus$ old$_0$ $\oplus$ old$_1$) it is easy to extract to newest data during a read by XORing one value from each row (e.g., (new $\oplus$ old$_0$ $\oplus$ old$_1$) $\oplus$ old$_0$ $\oplus$ old$_1$ = new).

We show an implementation, which we call bidirectional XOR memory. (Note that this is not exactly the same as the XOR memory in [14].) The setup consists of $N \times N$ RAM blocks for N bidirectional ports. We illustrate a 3-port bidirectional XOR memory in Figure 1. During a write, a port writes to all the RAM blocks in one row. During a read, a port reads from all the RAM blocks in one column. In order to read a value, all the output ports of the column corresponding to that port are XORed together. In order to write a value, all the output ports of the column corresponding to the port minus the RAM block on the (upper left to lower right) diagonal are XORed together with the (1-clock cycle delayed) incoming value. This value is written to all the RAM blocks on the row corresponding to that port. An example is shown in Figure 2. Because of the one-clock cycle write delay, this means if we

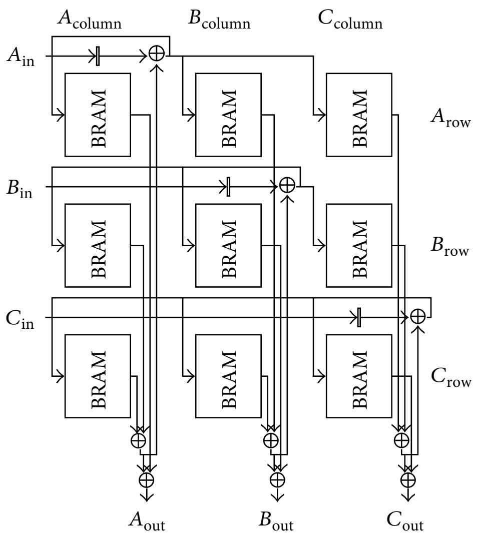


FIGURE 1: This XOR memory has 3 ports. B_{in} and B_{out} are part of the same port. Port B controls the address of each BRAM's read port in B_{column} and each BRAM's write port in B_{row}. Writing to memory using port B requires reading from BRAMs in B_{column} (except the one in B_{row}) as well as writing to the BRAMs in B_{row}. Reading from memory requires reading from all the BRAMs in B_{column}. Because the same read port on the BRAMs is used for reading and writing to the memory, it is not possible to split port B into a read-only port and a write-only port as it is with LVT.

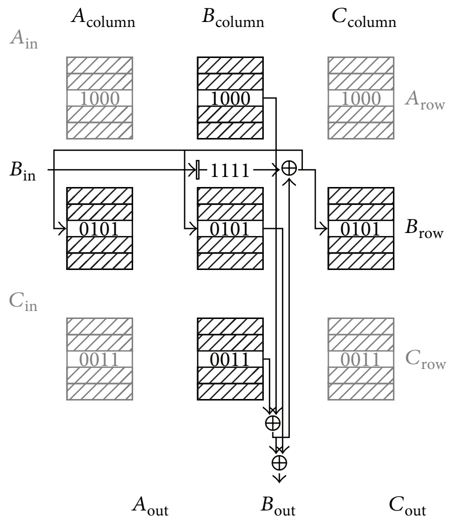


FIGURE 2: In this XOR memory example 1111 is being written on port B. This results in $1000 \oplus 0011 \oplus 1111 = 0100$ being written on all the RAM blocks on B_{row}. The next time this address is read the result will be $1000 \oplus 0100 \oplus 0011 = 1111$.

use read-after-write RAM blocks, the behavior of the memory would be write-after-read. This can be rectified if needed using the same technique in [14].

The work presented in [14] has similarities to our approach. But the main difference is the addition of the RAM blocks on the (upper left to lower right) diagonal that enable reading or writing on a port rather than only writing. If we

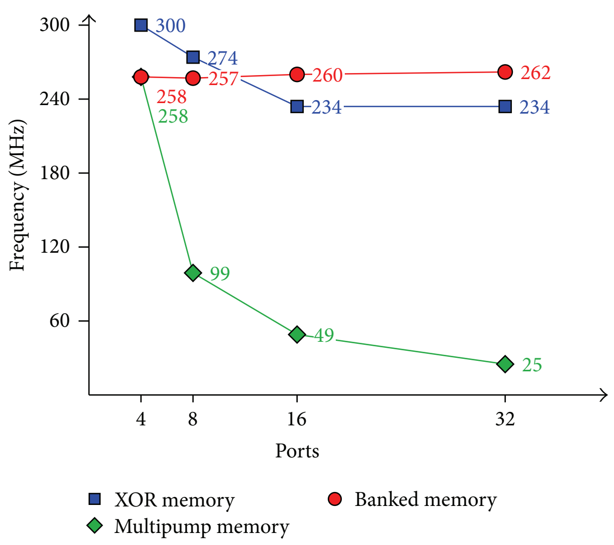


Figure 3: Max frequency for XOR, multipumped, and banked memory designs as the number of ports is varied. We used the results of our Omega memory in Table 1 for the banked memory.

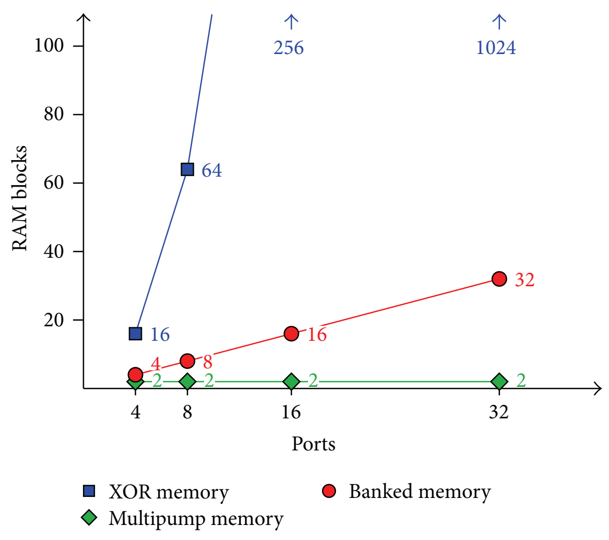


Figure 4: RAM block usage for XOR, multipumped, and banked memories as the number of ports is varied. We used the results of our Omega memory in Table 1 for the banked memory. In this figure we use the minimum number of block RAMs for each internal memory, which is 1 except in the multipump memory where it is 2. This results in the memories having different depths. The XOR memory has a depth of 512 (4 KB memory space). The multipump memory has a depth of 1024 (8 KB memory space). And the banked memory has a depth equal to 512 times the number of ports (16 KB to 128 KB memory space).

wanted write-only ports, we would eliminate the BRAMs on the diagonal. Also using the same approach in [14] additional columns can be added to create additional read-only ports.

These approaches have the advantage of low latency and working at relatively high frequencies. A significant drawback is the resource usage, since the number of extra RAM blocks scales quadratically with the number of ports (as seen in Figure 4). Another drawback, which this approach shares with replication, is that these added RAM blocks do not expand the storage space of the memory.

Banking, seen in [15–17], gains ports by adding RAM blocks to expand the memory. This is also the approach in this paper. This approach allows the memory to gain a full port and increase the memory space. However, multiple ports can not access the same RAM block (and therefore the unique memory space it holds) at the same time. Also, some type of network must route signals between the ports and the banks. These previous banking approaches do not buffer requests and therefore restrict the access of each port to a fraction of the memory space, explicitly segmenting each bank. By comparison, the work presented in this paper buffers and reorders messages, allowing unsegmented access to memory.

3. Architecture

As previously mentioned we implemented two different memory designs, hereafter referred to as the *fully connected* and *Omega* multiport memories. As will be further demonstrated in Section 5, a key differentiator is that the fully connected memory has better throughput, while the Omega memory scales better in terms of resource usage. However, both memories share common characteristics. Both designs use single-port RAM blocks for the memory banks and utilize reorder queues. Also, both utilize a network structure for routing between ports and banks and buffer requests to resolve contention.

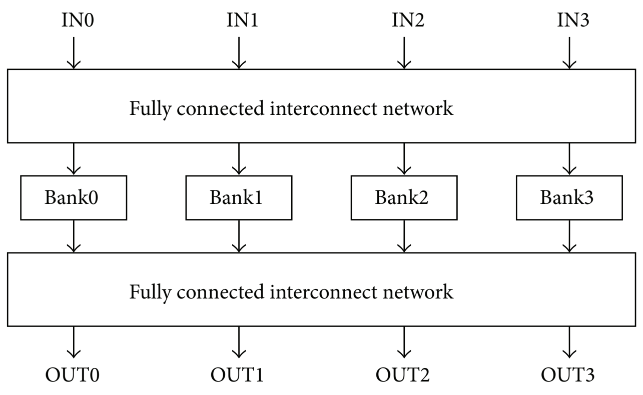

IN0
IN1
IN2
IN3
Fully connected interconnect network
Bank0
Bank1
Bank2
Bank3
Fully connected interconnect network
OUT0
OUT1
OUT2
OUT3

Figure 5: In the fully connected multiport memory all the buffering occurs in the fully connected interconnect networks.

3.1. Fully Connected Multiport Memory.

3.1. Fully Connected Multiport Memory. The fully connected multiport memory (Figure 5) received its name because it uses fully connected interconnect networks. The first of two fully connected networks routes requests from the input ports to the banks. The second routes read responses from the banks to the output ports. Compared to other networks, fully connected networks provide better contention resolution and handle uneven requests to memory banks better.

3.1.1. Memory Banks. For any banking approach, a memory with N ports requires at least N RAM blocks. Each RAM block holds a unique segment of the total memory space. We have multiple options to decide how to segment the memory

Table 1: Analysis of the two multiport memory designs.

	Ports		4	8	16	32	64	128	256
	Memory space		16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	1 MB
			Fully connected multiport memory						
Small resource: linked list FIFO and reorder queue depth set to 64	Resource	Registers	4 K	14 K	50 K	190 K	728 K		
	Utilization	LUTs	5.7 K	18 K	61 K	241 K	906 K		
	Virtex 7	BlockRAM	4	8	16	32	64		
	V2000T^2	Clock frequency	345 Mhz	313 Mhz	256 Mhz	273 Mhz	230 Mhz		
	Sequential	Throughput	100%	100%	100%	100%	50%		
		Latency^1	16	20	36	64	128		
	Random	Throughput	97%	93%	88%	72%	49%		
		Latency^1	66	65	85	97	144		
	Congested	Throughput	25%	13%	6%	3%	2%		
		Latency^1	105	230	490	1034	2780		
	Segregated	Throughput	100%	100%	100%	100%	100%		
		Latency^1	16	24	34	63	61		
			Omega multiport memory						
	Resource	Registers	3 K	9 K	22 K	53 K			
	Utilization	LUTs	5 K	11 K	24 K	53 K			
	Virtex 7	BlockRAM	4	8	16	32			
	V2000T^2	Clock frequency	258 Mhz	257 Mhz	260 Mhz	262 Mhz			
	Sequential	Throughput	100%	100%	100%	100%			
		Latency^1	17	25	37	56			
	Random	Throughput	94%	83%	68%	52%			
		Latency^1	72	110	131	193			
	Congested	Throughput	25%	13%	6%	3%			
		Latency^1	250	462	786	1046			
	Segregated	Throughput	25%	13%	6%	3%			
		Latency^1	247	461	756	1043			

Table 1: Continued.

	Ports		4	8	16	32	64	128	256
	Memory space		16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	1 MB
			Fully connected multiport memory						
	Resource	Registers	4.2 K	14 K	50 K	191 K	744 K		
	Utilization	LUTs	5.3 K	17 K	60 K	241 K	928 K		
	Virtex 7	BlockRAM	7	13	25	48	96		
	V2000T[2]	Clock frequency	352 Mhz	315 Mhz	253 Mhz	271 Mhz	230 Mhz		
	Sequential	Throughput	100%	100%	100%	100%	100%		
		Latency[1]	16	20	36	68	100		
	Random	Throughput	96%	95%	94%	99%	98%		
		Latency[1]	134	296	512	553	616		
	Congested	Throughput	25%	13%	6%	3%	2%		
		Latency[1]	105	231	491	1019	2750		
	Segregated	Throughput	100%	100%	100%	100%	100%		
Large resource: linked list FIFO and reorder queue depth set to 512		Latency[1]	16	23	38	61	63		
			Omega multiport memory						
	Resource	Registers	3.0 K	8.4 K	23 K	53 K	125 K	300 K	677 K
	Utilization	LUTs	7.3 K	16 K	36 K	77 K	163 K	345 K	746 K
	Virtex 7	BlockRAM	9	17	32	65	129	257	513
	V2000T[2]	Clock frequency	234 Mhz	234 Mhz	230 Mhz	230 Mhz	225 Mhz	202 Mhz	175 Mhz
	Sequential	Throughput	100%	100%	100%	100%	100%	100%	100%
		Latency[1]	17	25	37	57	93	161	293
	Random	Throughput	100%	99%	96%	89%	78%	63%	48%
		Latency[1]	312	580	566	751	1182	1397	2072
	Congested	Throughput	25%	13%	6%	3%	2%	1%	0.4%
		Latency[1]	2040	4046	7954	15351	28698	49182	65556
	Segregated	Throughput	25%	13%	6%	3%	2%	1%	0.4%
		Latency[1]	2037	4039	79530	15331	28593	49174	65388

[1]This measures the number of clock cycles between the end of the benchmark and when the last response of the last request gets received. In the worst case scenario several FIFOs queue data that has to wait for access to the same bank.
[2]This particular chip has 2.4 M registers, 1.2 M LUTs, and 1.3 K RAM blocks.

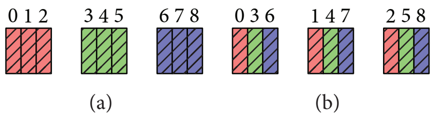


Figure 6: The above 2 sets of memory banks illustrate a simple addressing scheme and an interleaving addressing scheme. Certainly more complex addressing/hashing exists but is beyond the scope of this paper.

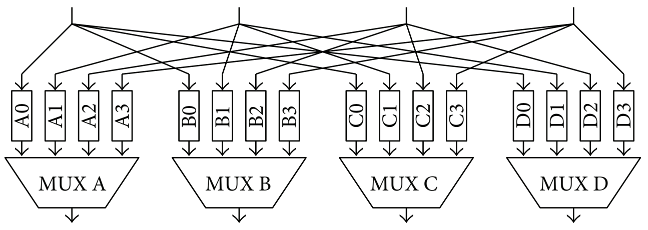


Figure 7: Fully connected interconnect network.

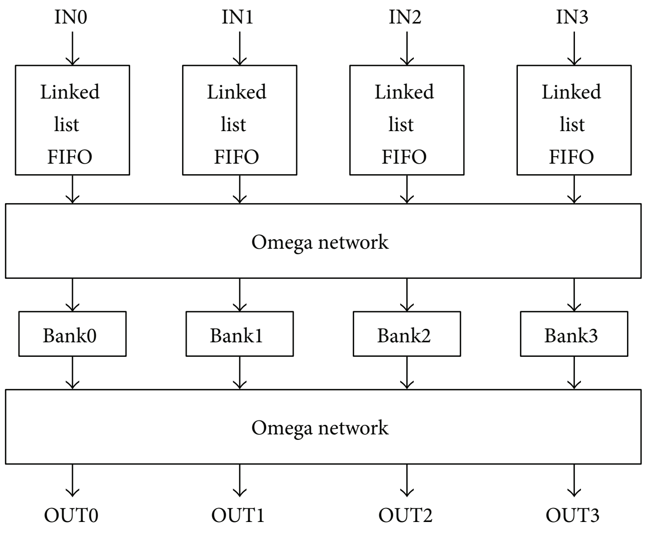


Figure 8: In the Omega multiport memory all the buffering occurs in the linked list FIFOs. The use of multistage interconnect networks, in this case Omega networks, helps reduce the area of the design.

space. The simplest option assigns the first Nth of the address space to Bank0, the next Nth to Bank1, and so on (Figure 6(a)). However, this approach can easily cause bottlenecks. For example, assume that all the processing elements start to read from a low address located in Bank0 and continue to sequentially increment the read addresses. All the requests would route to Bank0, necessitating multiple stalls. The interleaving memory address space that our design uses decreases the chance that these specific types of bottlenecks occur (Figure 6(b)).

3.1.2. Fully Connected Interconnect Network. A fully connected interconnect network (Figure 7) consists of multiple arbiters. An arbiter routes data from several inputs to one output and typically consists of simple FIFOs, a multiplexer, and some control logic. In Figure 7, FIFOs A0 to A3 and MUX A construct one of the four arbiters needed for a four-port fully connected interconnect network.

We use a simple round robin scheme to resolve contention. Assuming the arbiter most recently read from FIFO A0, the arbiter would continue to read from FIFO A0 until it is empty. Then, the arbiter reads from FIFO A1. Again, the arbiter continues to read from FIFO A1 until it is empty, and the cycle repeats. To reduce area and simplify the design we spend at least one clock cycle on each FIFO even if it is empty. This "vertical" control scheme limits the wasted time spent processing empty FIFOs.

Generally, N-by-N fully connected interconnect network consists of N, N-to-1 arbiters. The arbiters act independently. This independence allows for good arbitration schemes that achieve high throughput and low latency. Unfortunately, as the size of a fully connected interconnect network grows, the FIFOs and multiplexers require more space. An 8-to-1 multiplexer requires approximately twice the number of resources of a 4-to-1 multiplexer. This means that the area the multiplexers require grows by around N^2. The number of FIFOs grows by N^2 as well.

3.2. Omega Multiport Memory. The Omega multiport memory (Figure 8) has hardware structures designed for scaling. Instead of using fully connected interconnect networks, area efficient Omega networks (a type of multistage interconnect network) route signals to and from the memory banks. In addition, this memory uses N linked list FIFOs to buffer incoming requests, instead of N^2 FIFOs. Omega networks and linked list FIFOs pair well, because they both save logic resources. However, these structures are more restrictive than fully connected networks.

3.2.1. Omega Network. An Omega network consists of columns of Banyan switches [18, 19]. A Banyan switch is synthesized to two multiplexers. In the ON state, the switch crosses data over to the opposite output port. As an illustrative example, the second column in Figure 9 only contains switches in the ON state. In the OFF state, the switch passes data straight to the corresponding output port. The first and last columns in Figure 9 only contain switches in the OFF state.

The Omega network has features that make it attractive in a multiport memory design. If we switch whole columns of Banyan switches ON or OFF, we can easily determine where signals route by XORing the starting port index with the bits controlling the columns. For example, in Figure 9, the control bits are equal to 010_2 (with the least significant bit controlling the right most column) or 2 and input port 2 (010_2) routes

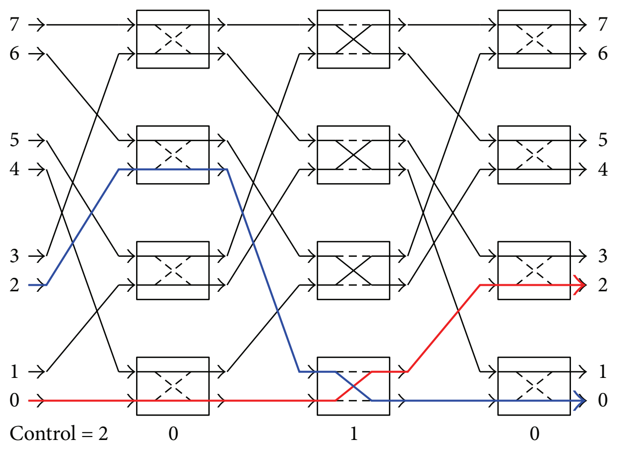


Figure 9: An 8-by-8 Omega network. We turn columns ON or OFF to rotate between different routing configurations.

to output port 0. Not coincidentally, the same configuration routes in reverse. Input port 0 routes to output port 2 and input port 2 routes to output port 0. This means that the design can use identical control bits for both the receiving and sending Omega networks.

In the Omega multiport memory design, the control for this network increments every clock cycle. As an example, input port 5 would connect to output port 5 and then ports 4, 7, 6, 1, 0, and so forth, until it cycles around again. This means that each input connects to each output an equal number of times.

3.2.2. Linked List FIFO. The partnering hardware structure, the linked list FIFO (Figure 10), behaves similarly to a regular FIFO with one major exception, when data is pushed or popped the "color" of the data must be specified. Each color has its own linked list in a shared RAM block. Similar to a software linked list, there exists a free pointer that points to the beginning of the free space linked list. Linked list FIFOs have previously been used in multicore CPU [20] and FPGA [21] designs.

Due to the linking pointers, the size of the shared RAM block now needs $O(N \log N)$ space to store N elements. However, $\log N$ grows slowly. For example, data stored in a 64-bit-wide by 1024-entry RAM would need an additional 11-bit-wide by 1024-entry RAM for the linking pointers. An illustrative example of the linked list FIFO is shown in Figure 10, which uses a 12-entry RAM and 3 colors.

In the initial state (Figure 10(a)), the red box and blue box FIFOs have no messages (read or write requests). The green box FIFO has 4 messages (marked as green striped rectangle). However, every FIFO reserves one empty space for the next incoming value. This limits the total available space in the linked list FIFO to TOTAL_DEPTH − FIFO_COUNT.

(i) On the first clock cycle (Figure 10(b)), the linked list FIFO receives a push containing a green message. The new green message gets stored in the reserved space at the tail of the green linked list. The free linked list pops one space. That space gets pushed onto the green linked list.

(ii) On the second clock cycle (Figure 10(c)), the linked list FIFO receives a pop for a green message. A green message gets popped from the head of the green linked list. The newly freed space gets pushed onto the free linked list.

(iii) On the third clock cycle (Figure 10(d)), the linked list receives a pop for a green message and a push for a red message. In this case the space that the green message was popped from gets pushed onto the red linked list. The free space linked list stays the same.

3.3. Reorder Queue. The buffering in both the fully connected and Omega memory ensures relatively high throughput; however, this buffering causes a problem for both memories, as read responses from different banks from the same port may come back out of order. Although out of order reads do not always cause an issue, to alleviate this issue we add reorder queues (Figure 12) to both multiport memory designs.

A reorder queue behaves similarly to a FIFO. However, some of the messages in between the head and tail pointer exist "in flight" (meaning the read request has been sent to a memory bank and has not yet come back) and not at the reorder queue memory. The reorder queue keeps track of the presence of messages with a bit array (a 1-bit-wide RAM).

Figure 11 shows an example with 5 clock cycles of operation. In the initial state (Figure 11(a)), the reorder queue has one present message and one in flight message.

(i) On the first clock cycle (Figure 11(b)), the present message at the head gets popped from the queue. A new message increments the tail, but the message remains in flight until it arrives at the reorder queue.

(ii) On the second clock cycle (Figure 11(c)), a new message arrives at the reorder queue; however, it does not arrive at the head of the queue so no message can get popped.

(iii) On the third clock cycle (Figure 11(d)), a message arrives at the head of the reorder queue.

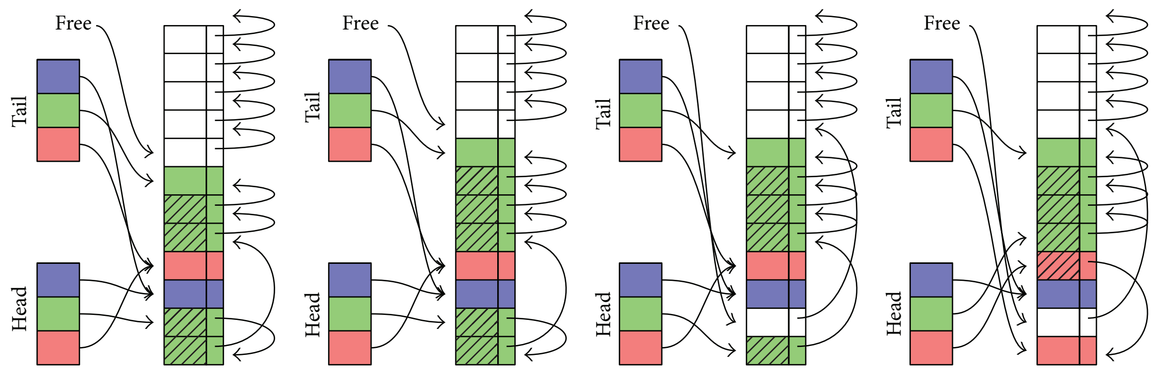
(iv) On the fourth clock cycle (Figure 11(e)), this message at the head of the reorder queue gets popped. If the reorder queue did not exist, the message that appeared on clock cycle 2 would have reached the output first even though it was sent later.

(v) On the fifth clock cycle (Figure 11(f)), a message arrives.

4. Evaluation Methodology

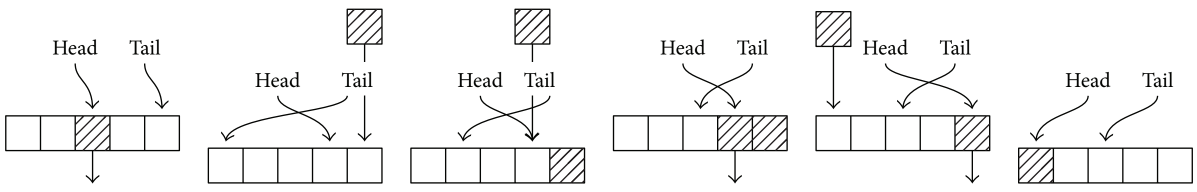
We implemented a small resource and a large resource version of each memory. The small version does not use RAM blocks for buffering and limits linked list FIFOs and reorder queues to a depth of 64. The large version does use RAM blocks for buffering and limits these memories to a depth of 512. However, in both cases we limit the depth of the FIFOs in the fully connected interconnect network to 32 since the number of FIFOs in it grows by $O(N^2)$.

Unless otherwise specified we use a data width of 64. We set the depth of the memory to PORTS × 512. This depth is



(a) Initial example (b) First clock cycle (c) Second clock cycle (d) Third clock cycle

FIGURE 10: A linked list FIFO during 3 clock cycles of operation.



(a) Initial example (b) First clock cycle (c) Second clock cycle (d) Third clock cycle (e) Fourth clock cycle (f) Fifth clock cycle

FIGURE 11: Reorder queue example.

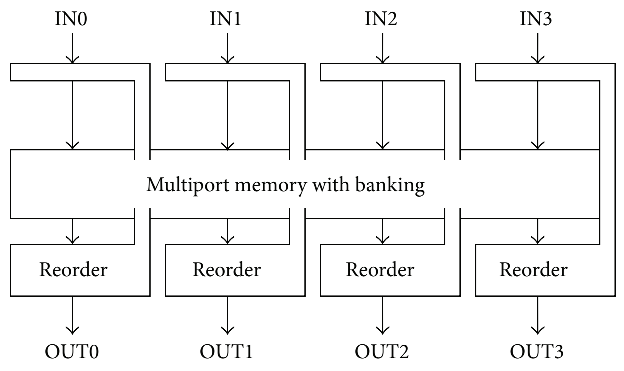


FIGURE 12: A reorder queue tags incoming read requests with an ID (which is the address into the reorder queue memory); this allows the reorder queue to correct the order of the read responses.

where each bank is the size of 1 RAM block. A smaller depth will still cause the bank to consume a whole RAM block. A larger depth would cause the banks to consume more RAM blocks.

For synthesis, we used Xilinx Synthesis Tools (XST) and targeted the Xilinx Virtex-7 V2000T. The V2000T has a relatively large number of logic blocks and RAM blocks, which helps us push the limits of these designs. These designs should also work well on Altera FPGAs, given the commonalities between the devices. Both Xilinx and Altera use a base depth of 32 for distributed RAM and a base depth of 512 for RAM blocks.

We used the ModelSim logic simulator to evaluate the throughput and latency of each configuration. The test bench used for evaluation consists of four benchmarks. Each benchmark tests the read performance of different memory access patterns: *sequential*, *random*, *congested*, or *segregated*.

The *sequential* benchmark begins by sending a read request to memory address 0 on each port. On the next clock cycle each port requests data from memory address 1. This continues unless a port stalls. On a stall, the memory address of that port does not increment until the memory resolves the stall.

The *random* benchmark begins by sending a read request to a random memory address on each port. On the next clock cycle, every port gets a new random memory address to read from. This continues until the end of the benchmark.

The *congested* benchmark begins by sending a read request to memory address 0 on each port. On subsequent clock cycles, the memory address does not change. This results in all the ports attempting to access the same memory bank. The purpose of this benchmark is to demonstrate the worst-case performance for any type of multiport memory with banking, including our designs.

The *segregated* benchmark begins by sending a read request to memory address i on each port, where i equals the index of the requested port. This address does not change on subsequent clock cycles. This results in all the banks receiving an equal number of requests. However, unlike the sequential and random benchmarks, there exists an uneven distribution of requests among all N^2 port to bank connections.

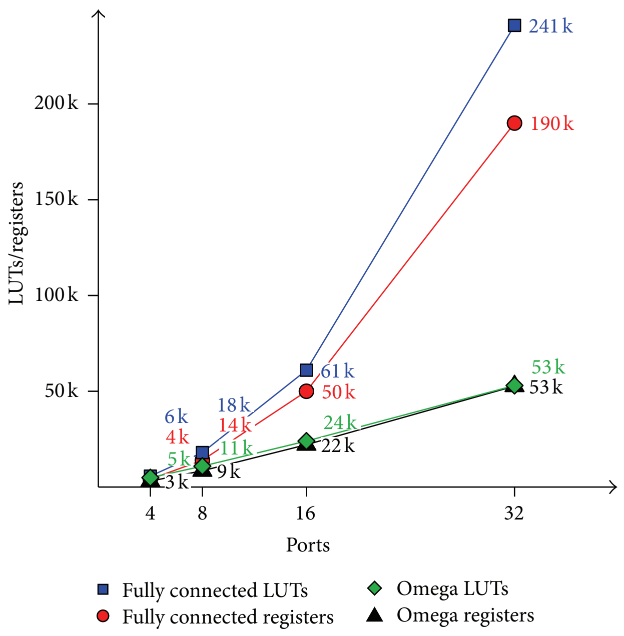


FIGURE 13: The effect of varying the number of ports on FPGA resource utilization (area) on the small resource memories. The fully connected memory grows by approximately N^2 and the Omega memory grows almost linearly.

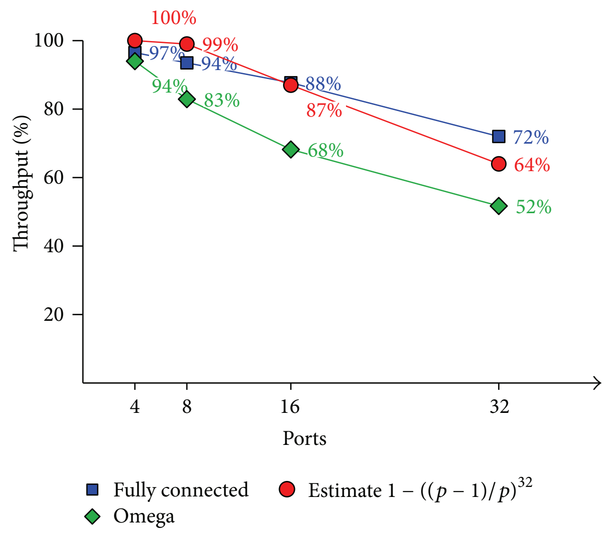


FIGURE 14: The effect of varying the number of ports on throughput of the random memory access benchmark on the small resource memories.

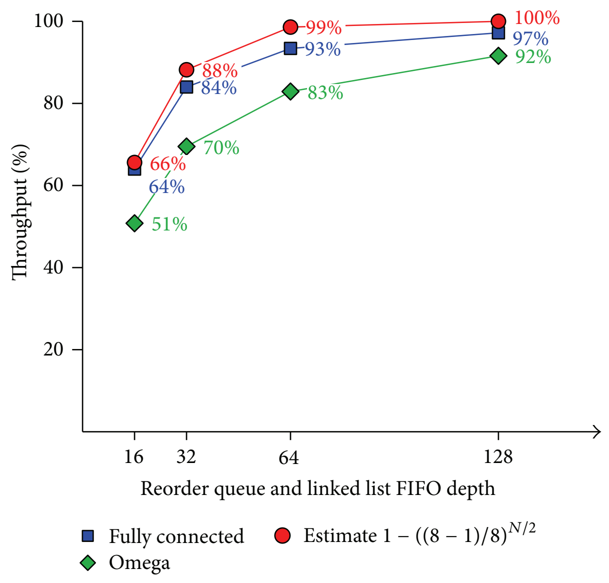


FIGURE 15: The effect of varying the depth of the linked list FIFOs and reorder queues on throughput of the random memory access benchmark (using 8-port memories).

We calculate the throughput of a given benchmark by measuring the ratio of read requests to potential read requests. If no stalls occur, the throughput equals 100%. We calculate the latency by measuring the number of clock cycles between the last read request and the last read response.

5. Results and Analysis

We present most of our results of different memory configurations in Table 1. We also use graphs in Figures 13, 14, 15, and 16. From a quick analysis of the results we can confirm several expected outcomes. The Omega memory uses fewer FPGA resources than the fully connected memory, particularly for memories with more ports. The fully connected memory achieves better throughput particularly for the segregated benchmark. We further analyze the effect of varying the number of ports, the depth of the buffering structures, and the bit width of the data.

5.1. Varying the Number of Ports. In terms of area, Figure 13 shows the effect on FPGA logic resources due to varying the number of ports. As expected, the fully connected memory consumes resources at a rate of approximately $O(N^2)$. The Omega memory consumes resources at a slower rate of approximately $O(N \log N)$. At 8 ports, the fully connected memory consumes 50% more resources than the Omega memory, and either design is viable on the target device. However, as the number of ports is increased, the fully connected memory runs out of resources quicker than the Omega memory. In Table 1 the cells reporting the performance of the fully connected memory with 128 and 256 ports are empty, because the fully connected memory for these configurations uses more than the available amount of resources on the Virtex-7 V2000T. The Omega memory does not reach this limit until it has more than 256 ports.

In terms of throughput, Figure 14 shows that increasing the number of ports decreases throughput, and Table 1 shows that increasing the number of ports increases latency. As expected, throughput decreases a little faster for the Omega memory. In both memories the latency grows almost linearly with the number of ports, because of the round robin contention resolution scheme in both. On average it takes $N/2$ clock cycles to start processing the first memory request.

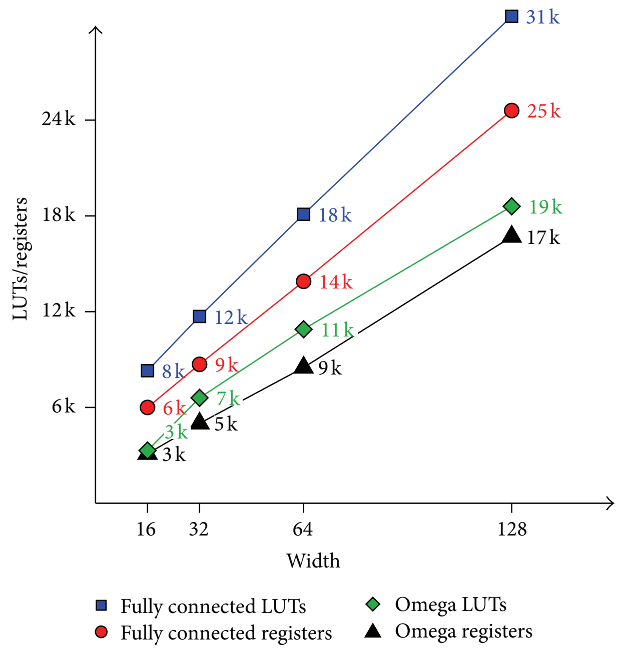


Figure 16: The effect of varying the bit width of the memory on FPGA resource utilization.

5.2. Varying the Buffer Depth. Increasing the buffer depth, that is, the reorder queue depth and the linked list FIFO depth, increases the throughput of the memories. Figure 15 shows that the throughput increases by around $O(1-((p-1)/p)^{N/2})$, where p equals the number of ports and N equals the buffer depth. $1-((p-1)/p)^{N/2}$ equals the probability that at least one of the last $N/2$ memory requests requested data on bank0 (or any specific bank). This approximately equals the probability that the next FIFO in the round robin has at least one message.

The buffer depth imposes a hard limit on the number of ports the Omega memory has. The Omega memory must have a buffer depth larger than the number of ports, because of the empty slots required in the linked list FIFOs. In Table 1 the cells reporting the performance of the small resource Omega multiport memory, which have 64, 128, or 256 ports, are empty because the linked list FIFOs have a depth of 64.

The fully connected memory does not share this restriction; however, an interesting anomaly occurs in the same situation. Table 1 shows that the sequential throughput of the small resource fully connected memory with 64 ports equals 50%. In this case, the messages wait 64 clock cycles to pass through the first interconnect network and then another 64 clock cycles to pass through the second. This happens because messages keep arriving at the FIFO the arbiter had just finished processing, similar to arriving at a bus stop just as the bus leaves. Since the reorder queue only allows 64 in flight messages, the 128-clock cycle latency only allows for 50% throughput. The problem disappears when the reorder queue is of size 128 or greater.

Increasing the buffer depth increases the latency. The buffers fill up over time as they attempt to prevent the memory from stalling. Full buffers mean that latency increases by the depth of the buffer. So, in benchmarks with contention, the latency increases linearly with the buffer depth.

Increasing the buffer depth increases FPGA utilization. The increase in buffer depth affects the Omega memory more since the fully connected memory does not have linked list FIFOs. If we use RAM blocks for buffers, any depth less than 512 results in using approximately the same number of resources. Consequently in this configuration we only implement FIFO depths of 64 and 512. Unsurprisingly, if buffers consist entirely of distributed RAMs (LUT resources), FPGA utilization increases linearly with the buffer depth.

5.3. Varying the Data Bit Width. Data width only effects resource utilization. As Figure 16 shows, FPGA utilization scales linearly with the data bit width. However, bit width does effect throughput when measuring by bytes per second instead of by percentage. The bytes per second measurement equals PERCENT_THROUGHPUT × PORT_COUNT × BIT_WIDTH × CLOCK_FREQUENCY. For example, the throughput on the random benchmark of the Omega memory with 256 ports is 172 GB/s.

6. Conclusions and Future Work

In total, the contributions of this work include the design of two multiport memories with banking and the components used to construct them. Based on the analysis of the results we recommend the fully connected multiport memory for designs that require low latency and high throughput. However, if the design requires 16 or more ports we recommend the Omega multiport memory. In other cases, either version should work fine.

Overall, we find that multiport memories with banking can provide high throughput communication to distributed compute nodes. We also find that these memories provide a substantial amount of shared memory. Upon the time of publication we will provide the source code for the memory designs on our research group's GitHub page (http://www.github.com/ISURCL/A-Scalable-Unsegmented-Multi-port-Memory-for-FPGA-based-Systems), so that other researchers can further analyze our work and integrate these cores into future high-performance reconfigurable computing applications.

In terms of planned future work, we are currently considering four main ideas: more aggressive pipelining, better contention resolution, memory coherence, and use of dual-port RAM blocks.

6.1. Improvements in Pipelining. Currently most of our design configurations achieve an estimated clock frequency between 200 Mhz and 300 Mhz. The Virtex-7 V2000T (with −2 speed grade) has a maximum achievable frequency of approximately 500 Mhz. Going through the design and pipelining the critical paths should increase the maximum frequency by 100 Mhz.

6.2. Better Contention Resolution. Currently the arbiters in the fully connected multiport memory process the FIFOs in a round robin scheme. In this scheme, processing empty FIFOs

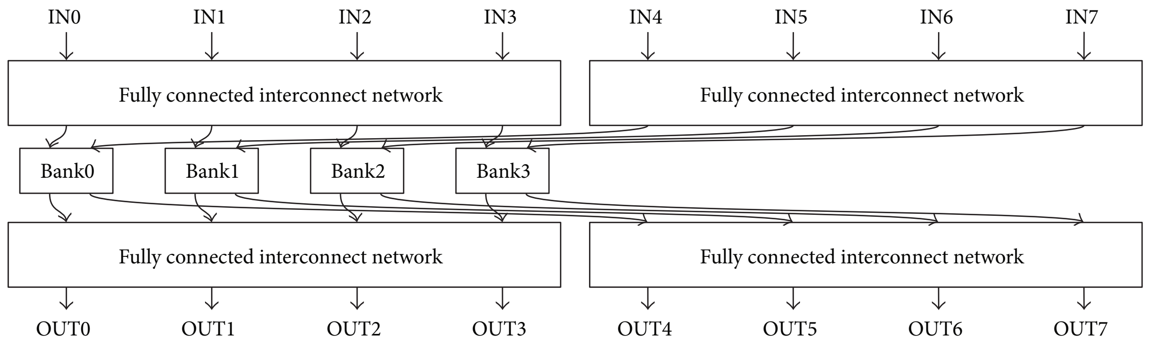

IN0
IN1
IN2
IN3
IN4
IN5
IN6
IN7
Fully connected interconnect network
Fully connected interconnect network
Bank0
Bank1
Bank2
Bank3
Fully connected interconnect network
Fully connected interconnect network
OUT0
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
OUT7

FIGURE 17: A banked multiport memory using single-port RAM blocks can be modified to use dual-port RAM blocks by replicating the design and sharing RAM blocks.

still consumes one clock cycle. Skipping empty FIFOs would improve latency and throughput [22]. Allowing the arbiter to skip empty FIFOs would also enable a horizontal arbitration scheme that would prevent starvation issues.

Improving the contention resolution of the Omega memory has more complications. The Omega network limits the number of routes from input ports to the banks. One avenue to explore is setting the control bits of the Omega networks so that the fullest FIFO is always being processed.

6.3. Memory Coherence. Memory coherence is an issue with these memories because of the variable latency. For example, if port 0 sends a write request to address 42 and takes 5 clock cycles to reach the appropriate bank and port 1 sends a read request for address 42 and takes 3 clock cycles to reach the bank then the read request will retrieve old data.

The fully connected memory can be made memory coherent by making arbiters give priority to requests sent first. This would require memory requests to be time-stamped. The timestamp would need enough bits to cover the largest latency difference between two active requests. The $N - 1$ compares to determine the earliest request probably need more than one clock cycle to complete, so a min-heap binary tree hardware structure can be used to pipeline the arbiter. However, this does not work in the Omega memory because of the more restrictive hardware structures used.

6.4. Use of Dual-Port RAM Blocks. In either version (fully connected or Omega), designing N-port memory as two $N/2$-port memories with banks connected by dual-port RAM blocks, instead of using single-port RAM blocks, can achieve superior results (Figure 17). The matching memory banks (by their index) of the two smaller designs would combine to make dual-port RAM blocks. This makes the estimated throughput and latency equal to the memory with half the number of ports in Table 1. This also makes the estimated resource utilization twice the resource utilization of the memory with half the number of ports in Table 1. For example, a small resource fully connected multiport memory with 32 ports utilizing dual-port RAM blocks would use around 100 K registers, 122 K LUTs, and 32 RAM blocks, versus 190 K registers, 241 K LUTs, and 32 RAM blocks when only utilizing single-port RAM blocks.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work is supported in part by the National Science Foundation (NSF) under the Awards CNS-1116810 and CCF-1149539.

References

[1] V. Betz and L. Shannon, Eds., *FPGAs in 2032: Challenges and Opportunities in the next 20 Years*, 2012.

[2] Stratix V Device Overview, SV51001, Altera, 2015, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_51001.pdf.

[3] Xilinx, *7 Series FPGAs Overview, DS180*, Xilinx, 2014.

[4] A. K. Jones, J. Fazekas, R. Hoare, D. Kusic, and J. Foster, "An FPGA-based VLIW processor with custom hardware execution," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA '05)*, pp. 107–117, Monterey, Calif, USA, February 2005.

[5] N. Manjikian, "Design issues for prototype implementation of a pipelined superscalar processor in programmable logic," in *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and signal Processing (PACRIM '03)*, vol. 1, pp. 155–158, IEEE, Vancouver, Canada, August 2003.

[6] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in FPGA high-level synthesis," in *Proceedings of the IEEE Design, Automation & Test in Europe (DATE '13)*, pp. 194–197, Grenoble, France, March 2013.

[7] H. E. Yantir, S. Bayar, and A. Yurdakul, "Efficient implementations of multi-pumped multi-port register files in FPGAs," in *Proceedings of the Euromicro Conference on Digital System Design (DSD '13)*, pp. 185–192, IEEE, Los Alamitos, CA, USA, September 2013.

[8] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 131–140, IEEE, Napa, Calif, USA, April 2006.

[9] R. Moussali, N. Ghanem, and M. A. R. Saghir, "Supporting multithreading in configurable soft processor cores," in *Proceedings*

of the ACM International Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES '07), pp. 155–159, September 2007.

[10] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '06)*, pp. 201–210, Monterey, Calif, USA, February 2006.

[11] 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)*, pp. 41–50, ACM, Monterey, Calif, USA, February 2010.

[12] F. Anjam, S. Wong, and F. Nadeem, "A multiported register file with register renaming for configurable softcore VLIW processors," in *Proceedings of the International Conference on Field-Programmable Technology (FPT' 10)*, pp. 403–408, IEEE, Beijing, China, December 2010.

[13] A. M. S. Abdelhadi and G. G. F. Lemieux, "Modular multi-ported SRAM-based memories," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '14)*, pp. 35–44, ACM, February 2014.

[14] C. E. LaForest, M. G. Liu, E. R. Rapati, and J. G. Steffan, "Multi-ported memories for FPGAs via XOR," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA '12)*, pp. 209–218, Monterey, Calif, USA, February 2012.

[15] J. Moscola, R. K. Cytron, and Y. H. Cho, "Hardware-accelerated RNA secondary-structure alignment," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 3, article 14, 2010.

[16] M. A. R. Saghir, R. Naous, and M. A. R. Saghir, "A configurable multi-ported register file architecture for soft processor cores," in *Reconfigurable Computing: Architectures, Tools and Applications: Third International Workshop, ARC 2007, Mangaratiba, Brazil, March 27–29, 2007. Proceedings*, pp. 14–25, Springer, Berlin, Germany, 2007.

[17] M. A. R. Saghir, M. El-Majzoub, and P. Akl, "Datapath and ISA customization for soft VLIW processors," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig '06)*, pp. 1–10, IEEE, September 2006.

[18] C. L. Wu and T. Y. Feng, "On a class of multistage interconnection networks," *IEEE Transactions on Computers*, vol. 29, no. 8, pp. 694–702, 1980.

[19] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1145–1155, 1975.

[20] S. Bell, B. Edwards, J. Amann et al., "TILE64TM processor: a 64-core SoC with mesh interconnect," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC '08)*, pp. 88–598, February 2008.

[21] A. Nikologiannis, I. Papaefstathiou, G. Kornaros, and C. Kachris, "An FPGA-based queue management system for high speed networking devices," *Microprocessors and Microsystems*, vol. 28, no. 5-6, pp. 223–236, 2004.

[22] M. Weber, "Arbiters: design ideas and coding styles," in *Proceedings of the Synopsys Users Group Boston Conference (SNUG '01)*, Boston, Mass, USA, September 2001.