Modular Multi-ported SRAM-based Memories

Ameer M.S. Abdelhadi and Guy G.F. Lemieux
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, B.C., V6T 1Z4, Canada
{ameer,lemieux}@ece.ubc.ca

ABSTRACT

Multi-ported RAMs are essential for high-performance parallel computation systems. VLIW and vector processors, CGRAs, DSPs, CMPs and other processing systems often rely upon multi-ported memories for parallel access, hence higher performance. Although memories with a large number of read and write ports are important, their high implementation cost means they are used sparingly in designs. As a result, FPGA vendors only provide dual-ported block RAMs to handle the majority of usage patterns. In this paper, a novel and modular approach is proposed to construct multi-ported memories out of basic dual-ported RAM blocks. Like other multi-ported RAM designs, each write port uses a different RAM bank and each read port uses bank replication. The main contribution of this work is an optimization that merges the previous live-value-table (LVT) and XOR approaches into a common design that uses a generalized, simpler structure we call an invalidation-based live-value-table (I-LVT). Like a regular LVT, the I-LVT determines the correct bank to read from, but it differs in how updates to the table are made; the LVT approach requires multiple write ports, often leading to an area-intensive register-based implementation, while the XOR approach uses wider memories to accommodate the XOR-ed data and suffers from lower clock speeds. Two specific I-LVT implementations are proposed and evaluated, binary and one-hot coding. The I-LVT approach is especially suitable for larger multi-ported RAMs because the table is implemented only in SRAM cells. The I-LVT method gives higher performance while occupying less block RAMs than earlier approaches: for several configurations, the suggested method reduces the block RAM usage by over 44% and improves clock speed by over 76%. To assist others, we are releasing our fully parameterized Verilog implementation as an open source hardware library. The library has been extensively tested using ModelSim and Altera's Quartus tools.[1]

Categories and Subject Descriptors

B.3.2 [**MEMORY STRUCTURES**]: Design Styles – *Cache memories, Shared memory*; C.1.2 **[PROCESSOR ARCHITECTURES**]: Multiple Data Stream Architectures (Multiprocessors) – *Interconnection architectures, Parallel processors (e.g. common bus, multiport memory, crossbar switch)*

Keywords

Embedded memory; block RAM; multi-ported memory; shared memory; cache memory; register-file; parallel memory access

1. INTRODUCTION

Multi-ported memories are the cornerstone of all high-performance CPU designs. They are often used in the register files, but also in other shared-memory structures such as caches and coherence tags. Hence, high-bandwidth memories with multiple parallel reading and writing ports are required. In particular, multi-ported RAMs are often used by wide superscalar processors [1], VLIW processors [1][2], multi-core processors [3][4], vector processors, coarse-grain reconfigurable arrays (CGRAs), and digital signal processors (DSPs). For example, the second generation of the Itanium processor architecture employs a 20-port register file constructed from SRAM bit cells with 12 read ports and 8 write ports [3]. The key requirement for all of these designs is fast, single-cycle access from multiple requesters. These multiple requesters require concurrent access for performance reasons.

One way of synthesizing a multi-ported RAM is to build it from registers and logic. However, this is only feasible for very small memories. Another way is to alter the basic SRAM bit cell to provide extra access ports, but area growth is quadratic with the number of ports, so this requires a custom design for each unique set of parameters (number of ports, width and depth of RAM). Since FPGAs must fix their RAM block designs for generic designs, it is too costly to provide highly specialized RAMs with a large number of ports. A multi-ported RAM can also be emulated through banking or multi-pumping. Banking uses hashing and arbitration to provide access, but it leads to unpredictable (multi-cycle) access latencies under collisions; this complicates system design and compromises performance. Multi-pumping provides a few extra ports, but it is limited by the amount of overclocking. Hence, a method of composing arbitrary, multi-ported RAMs from simpler RAM blocks is required.

In this paper, a modular and parametric multi-ported RAM is constructed out of basic dual-ported RAM blocks while keeping minimal area and performance overhead. The suggested method significantly reduces SRAM use and improves performance compared to previous attempts. To verify correctness, the proposed architecture is fully implemented in Verilog, simulated using Altera's ModelSim, and compiled using Quartus II. A large variety of different memory architectures and parameters, *e.g.* bypassing, memory depth, data width, number of reading or writing ports are simulated in batch, each with over million random memory access cycles. Stratix V, Altera's high-end performance-oriented FPGA, is used to implement and compare the proposed architecture with previous approaches. Major contributions of this paper are:

- A novel I-lVT architecture to produce modular multi-ported SRAM-based memories. It is built out of dual-ported SRAM blocks only, without any register-based memories. To the authors' best knowledge, compared to other multi-ported approaches, the I-LVT consumes the fewest possible SRAM cells. It also provides improved overall performance.

[1] http://www.ece.ubc.ca/~lemieux/downloads/

- A fully parameterized Verilog implementation of the suggested methods, together with previous approaches. A flow manager to simulate and synthesize various designs with various parameters in batch using Altera's ModelSim and Quartus II is also provided. The Verilog modules and the flow manager are available online [5].

Notation and abbreviations used for the rest of the paper are listed in Table 1. The rest of this paper is organized as follows. In section 2, conventional RAM multi-porting techniques in embedded systems are reviewed. Previous attempts to provide multi-ported memories are reviewed in section 3. The proposed invalidation-based live-value-table method is described in detail and compared to previous methods in section 4. The experimental framework, including simulation and synthesis and results, are discussed in section 5, and conclusions are drawn in section 6.

Table 1. List of notations and abbreviations

n_W	Write ports number	WAddr	Write address
n_R	Read ports number	RAddr	Read address
w	Data width	WData	Write data
d	Memory depth	RData	Read data
n_{M20K}	Number of M20K blocks	RBankSel	Read bank selector
n_{BypReg}	Number of bypass registers	LVT	Live-value-table
f_{fb}, f_{out}	LVT feedback/out functions	I-LVT	Invalidation LVT

2. RAM MULTI-PORTING TECHNIQUES IN EMBEDDED SYSTEMS

This section provides a review of current methods of creating multi-ported RAMs in embedded systems. Creating multi-ported access to register-based and SRAM-based memories is described in subsection 2.1. Multi-pumping is described in subsection 2.2. Replicating a memory bank to increase the number of read and write ports is described in subsections 2.3 and 2.4, respectively.

2.1 Register-based RAM

Multi-ported RAM arrays can be constructed using basic flip-flop cells and logic. As depicted in Figure 1, each writing port uses a decoder to steer the relevant written data into the addressed row. Each read port uses a mux to choose the relevant register output. This method is not practical for large memories due to area inflation, fan-out increase, performance degradation, and a decline in routability.

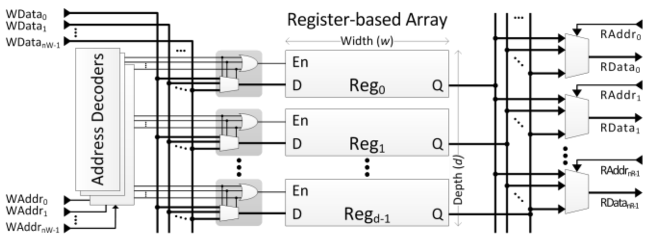

Figure 1. Register-based multi-ported RAM.

2.2 RAM Multi-pumping

A time-multiplexing approach can be applied to a single dual-ported SRAM block to reuse access ports and share them among several clients, each during a different time slot. As depicted in Figure 2, addresses and data from several clients are latched then given round-robin access to a dual-ported RAM. The RAM must operate at a higher frequency than the rest of the circuit. If the maximum RAM frequency is similar to the pipe frequency, or a large number of access ports are required, then multi-pumping cannot be used. A number of designs utilize multi-pumping to gain additional access ports while keeping area overhead minimal [6][7]. The 2.3GHz Wire-Speed POWER processor uses double-pumping to double the writing ports [8].

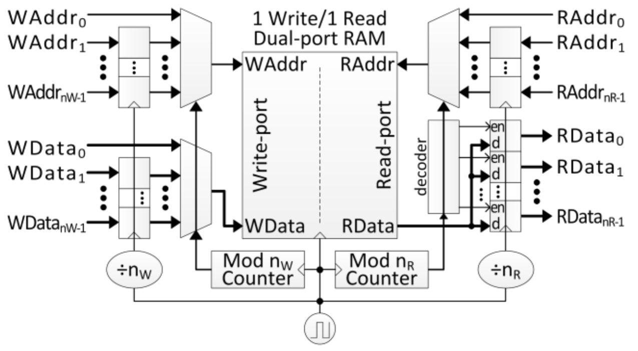

Figure 2. Multi-pumping: RAM is clocked faster than the pipeline in the periphery, allowing multiple accesses during one pipeline cycle.

2.3 Multi-read RAM: Bank Replication

To provide more reading ports, the whole memory bank can be replicated while keeping common write address and data as shown in Figure 3. Any data will be written to all bank replicas at the same address, hence reading from any bank is equivalent. This method incurs high SRAM area and consumes more power. However, the replication approach has two strong advantages over other multi-porting approaches. The first is the simplicity and modularity of bank replication. The second is that read access time is unaffected as the number of ports increases; only write delays increase due to fan-out, but this can be hidden via pipelining and bypassing. The bank replication technique is commonly used in state-of-the-art processing architectures to increase parallelism. The 2.3GHz Wire-Speed POWER processor replicates a 2-read SRAM bank to achieve 4 read ports [8]. Each of the two integer clusters of the Alpha 21264 microprocessor has a replicated 80-entry register file, thus doubling the number of read ports to support two concurrent integer execution units. Similarly, the 72-entry floating-point register file is duplicated, supporting two concurrent floating-point units [9].

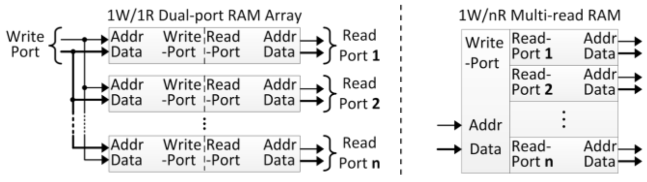

Figure 3. (left) Replicated dual-ported banks with a common write port. (right) Symbol used in this paper equivalent to a multi-read RAM block.

2.4 Multi-write RAM: Emulation via multi-banking

Multi-ported memories are very expensive in terms of area, delay, and power for a large number of ports. The overhead of multi-porting can be reduced by multi-banking if one relaxes the guaranteed access delay constraint. As depicted in Figure 4, the total RAM capacity can be divided into several banks, each with few ports (*e.g.* dual-port). A fixed hashing scheme is used to match each access to a single bank; often, the address MSBs are used. Arbitration logic steers access from multiple ports to each bank. Since two ports can request access to data in the same bank at the same time, a conflict resolving circuit determines which port grants access to a specific bank. The other port will miss the arbitration and is required to request access again. Not only does the multi-banking approach provide unpredictable access latency due to the arbitration miss, but it also degrades delay due to the additional access circuitry. Several approaches have been proposed to improve multi-banking [10][1][11][12]. State-of-the-art memory controllers and processor caches are based on multi-banking due to area and power efficiency. For example, the Pentium 5 has a data cache with 8 interleaved banks and two access ports [13].

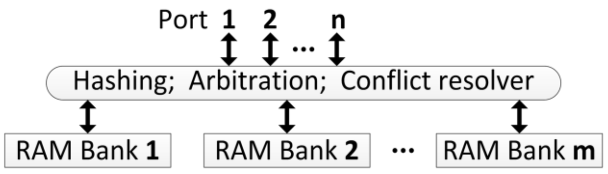

Figure 4. Multi-banking: RAM capacity is divided into several banks. Ports access a RAM bank with a fixed hashing scheme.

3. MODULAR MULTI-PORTED SRAM-BASED MEMORIES: PREVIOUS WORK

In this section a review of two previous modular designs of multi-ported SRAM-based memories are provided. The first approach is based on multi-banking with a live-value-table (LVT) [14] and is described in subsection 3.1. The second approach retrieves the latest written data by utilizing logical XOR properties [15] and is described in subsection 3.2.

3.1 LVT-based Multi-ported RAM

For each RAM address, the LVT stores the ID of the bank replica that holds the latest data. As depicted in Figure 5 (left), an LVT-based multi-ported RAM uses a different bank replica for each writing port, while each bank has several reading ports. All banks are accessed by all read addresses in parallel; the LVT helps to steer the read data out of the correct bank since it holds the ID of last accessed (written) bank for each address.

Actually, the LVT itself is a multi-ported RAM with the same memory depth and number of writing ports as the implemented multi-ported memory. However, since the LVT stores only bank IDs, the data (line) width of the LVT table is only $\lceil log_2 \rceil$ of the number of banks, which is equal to the number of writing ports. Furthermore, the LVT doesn't have write data, instead it writes a fixed bank ID for each port as described in Figure 5 (right).

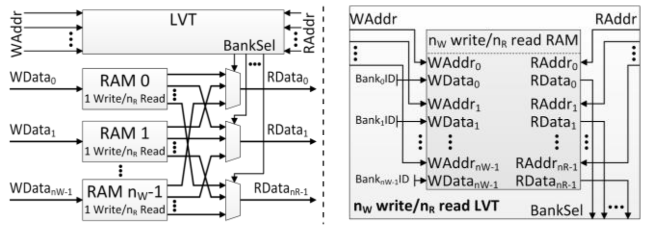

Figure 5. (left) LVT-based multi-ported RAM. (right) An LVT implemented using multi-ported RAM.

Since an LVT is a narrow, multi-port memory, it is implemented as a registered-based, multi-ported RAM. As explained in subsection 2.1, register-based RAM is not suitable for building large memories. While the LVT width is only $\log_2$ of the number of writing ports, the depth of the LVT is still similar to the depth of the original RAM. *This is the main cause of the area overhead*. In this paper, to reduce this area overhead, two methods of constructing SRAM-based LVTs are described. The methodology of constructing SRAM-based LVTs is also generalized. To the authors' best knowledge this is the first attempt to build an LVT out of SRAM blocks only.

Assuming that bank IDs are binary encoded, the total number of registers required to implement the LVT is

$$d \cdot \lceil \log_2 n_W \rceil. \qquad (1)$$

For deep memories, the large number of registers and huge read multiplexers make register-based LVTs impractical. For example, on a Stratix V GX A5 device (185k ALMs), Quartus II failed to synthesize a 16k-deep memory with two write ports.

A register-based LVT with SRAM banks requires n_W multi-read banks for each write port. Each multi-read bank supports n_R reading ports, allowing the LVT to select the required data block. The total number of SRAM cells is

$$d \cdot w \cdot n_W \cdot n_R. \qquad (2)$$

Using Altera's Stratix V M20K block RAMs, the total number of required M20K blocks is

$$n_{M20K}(d,w) \cdot n_W \cdot n_R. \qquad (3)$$

Where $n_{M20K}(d,w)$ is the number of M20K Blocks required to construct a RAM with a specific depth and data width. This value, described by equation (4), is derived from Figure 6, which shows how Altera's M20K blocks can be configured into several RAM depth and data width configurations [16]. The total amount of utilized SRAM bits can be either 16Kbits, or 20Kbits. Assuming that the RAM packing process minimizes the number of blocks cascaded in depth to avoid additional address decoding, each 16K lines will be packed into single bit-wide blocks, and the remainder will be packed into the minimal required configuration as follows

$$n_{M20K}(d,w) = \left\lfloor \frac{d}{16k} \right\rfloor \cdot w + \begin{cases} d\%16k > 8k & w \\ 8k \geq d\%16k > 4k & \lceil w/2 \rceil \\ 4k \geq d\%16k > 2k & \lceil w/5 \rceil \\ 2k \geq d\%16k > 1k & \lceil w/10 \rceil \\ 1k \geq d\%16k > ½k & \lceil w/20 \rceil \\ ½k \geq d\%16k & \lceil w/40 \rceil \end{cases}. \qquad (4)$$

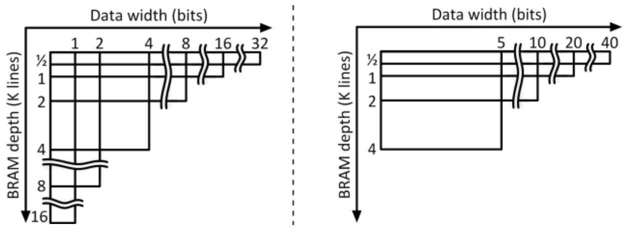

Figure 6. Altera's M20K BRAM configurations (left) 16Kbit (right) 20Kbit.

3.2 XOR-based Multi-ported RAM

While the LVT-based multi-port RAM just shown implements its LVT as a register-based multi-ported RAM, the XOR-based multi-ported RAM is implemented using SRAM blocks [15]. This makes it more efficient for deep memories. However, as will be shown, it is inefficient for wide memories.

The XOR-based method utilizes the special properties of the XOR function to retain the latest written data for each write port. XOR is commutative $a \oplus b = b \oplus a$, associative $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, zero is the identity $a \oplus 0 = a$, and the inverse of each element is itself $a \oplus a = 0$.

As illustrated in Figure 7, each write port has a bank with multi-read and a single write. Part of the read ports are used as a feedback to generate new data and rewrite a specific bank, while the other read ports generate the data outputs. To perform a write, the new data is XOR'ed together with all the old data from the other banks; the result is rewritten to the corresponding bank. Hence if an address A is written through write port i with data $WData_i$, $Bank_i$ will be rewritten with

$$Bank_i[A] \leftarrow Bank_0[A] \oplus Bank_1[A] \oplus \ldots \oplus WData_i \oplus \ldots \oplus Bank_{n_W-1}[A]. \quad (5)$$

A read is performed by XOR'ing all the data for the corresponding read address from all the banks, hence,

$$RData_i[A] \leftarrow Bank_0[A] \oplus Bank_1[A] \ldots \oplus Bank_{n_W-1}[A]. \quad (6)$$

Substituting $Bank_i[A]$ from (5) into (6) and applying commutative and associative properties of the XOR shows that each bank appears twice in the XOR equation, hence will be cancelled since XORing similar elements is 0. The only remaining item will be $WData_i$, the required data.

The XOR-based multi-ported RAM requires n_W multi-read banks for each write port. Each multi-read bank supports $n_W - 1$ read ports to feedback the other ports via XORs, and n_R read ports. Each feedback read port is of width d, to match the write data, so these feedback memories can be quite large. The number of required SRAM cells is

$$d \cdot w \cdot n_W \cdot (n_R + n_W - 1). \quad (7)$$

Using Altera's Stratix V M20K block RAMs, the total number of required M20K blocks is

$$n_{M20K}(d, w) \cdot n_W \cdot (n_R + n_W - 1). \quad (8)$$

Since FPGA block RAM is synchronous, data feedbacks are read with a one-cycle read delay. Hence, the written data, their addresses and write-enables must be retimed to match the feedback data. This requires the following number of registers

$$n_W \cdot (\mathrm{w} + \lceil \log_2 d \rceil + n_W). \quad (9)$$

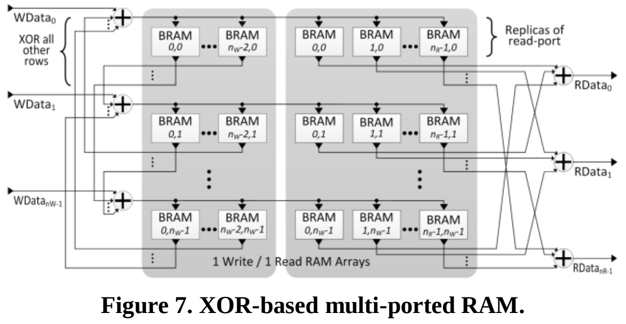

Figure 7. XOR-based multi-ported RAM.

4. INVALIDATION TABLE

As described in the previous section, the XOR-based multi-ported memories requires $n_W \cdot (n_R + n_W - 1)$ manipulated copies of the RAM content, while the LVT approach requires another register-based multi-ported memory with the same number of read and write ports for bank IDs.

This work proposes to implement LVTs using SRAM blocks only, which has a major advantage over register-based LVTs and a lower SRAM area compared to the XOR-based approach. Instead of requiring multiple write ports to each multi-read bank in regular LVT method, we suggest a design with a single write port each like the XOR method. This makes it feasible to implement the LVT using standard dual-ported RAMs. However, writing an ID to one bank requires also invalidating the IDs in the other banks, which produces the need for the multiple write ports. Instead, we suggest writing an ID to only one specific bank and invalidating all the other IDs with a single write by using an invalidation table. Since the *invalidation table* has the same functional behavior as an LVT, we call it an invalidation-based LVT, or I-LVT.

The I-LVT doesn't require multiple writes to indicate the last-written bank. Instead, as described in Figure 8, the I-LVT reads all other bank IDs as feedback, embeds the new bank ID into the other values through a feedback function f_{fb}, then rewrites the specific bank. To extract back the latest written bank ID, all banks are read and data is processed with the output function f_{out} to regenerate the required ID. Selection of these two functions, f_{fb} and f_{out}, is what distinguishes different I-LVT implementations.

The I-LTV requires n_W multi-read banks, each with n_R read ports for output extraction. Furthermore, an additional $n_W - 1$ read ports are required in each bank for feedback rewriting. The data width of these read ports varies depending on the feedback method and the bank ID encoding. In this paper, two bank ID encoding methods are presented, binary and one-hot. The binary method employs exclusive-OR functions to embed the bank IDs, while the second uses mutual-exclusive conditions to invalidate table entries and generate one-hot-coded bank selectors. The two methods are described in subsections 4.1 and 4.2, respectively.

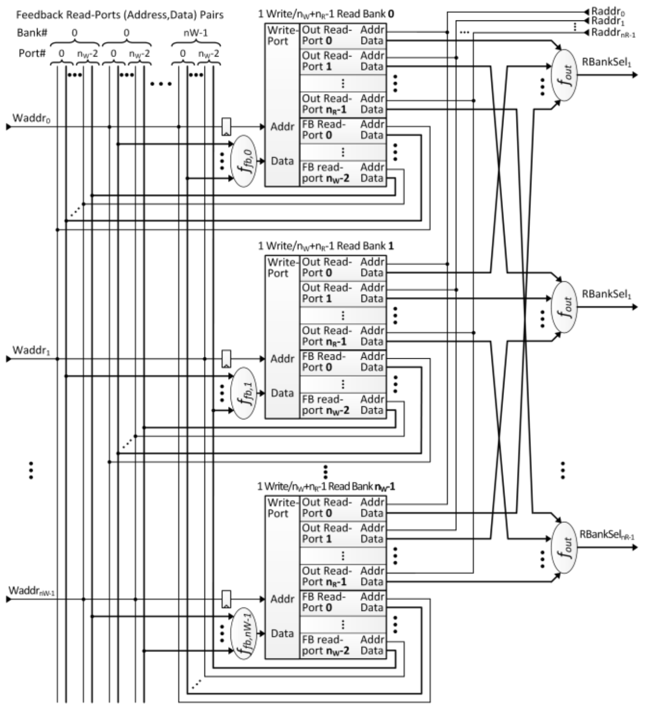

Figure 8. Generalized approach for building the I-LVT.

4.1 Bank ID Embedding: Binary-coded selectors

This approach attempts to reduce the SRAM cell count in the I-LVT by employing binary-coded bank IDs. The special properties of the exclusive-OR function are utilized to embed the latest written bank ID, hence invalidating all other IDs. The current written bank ID is XOR'ed with the content of all the other banks from the same write address as described in the following feedback function,

$$f_{fb,k} = k \oplus_{\substack{0 \le i < n_W \\ i \ne k}} Bank_i[WAddr_k], \tag{10}$$

where k is the ID of the currently written bank.

Similar to the XOR-based method described in subsection 3.2, the last written bank ID is extracted by XOR'ing the content of all the banks from the same read address as described in the following output extraction function

$$f_{out,k} = \oplus_{0 \le i < n_W} Bank_i[RAddr_k]. \tag{11}$$

Without loss of generality, if address A in bank k is written with the feedback function from Equation (10), then

$$Bank_k[\mathrm{A}] = k \oplus_{0 \le i < n_W;\, i \ne k} Bank_i[\mathrm{A}]. \tag{12}$$

If one of the read ports, say read port r, is trying to read from the same address, namely $RAddr_r = A$, then the read bank selector will be generated using the same output extraction function from (11), hence

$$RBankSel_r = \oplus_{0 \le i < n_W} Bank_i[A]. \tag{13}$$

Due to XOR operation associativity, $RBankSel_r$ from (13) can be expressed as

$$RBankSel_r = Bank_k[\mathrm{A}] \oplus_{0 \le i < n_W;\, i \ne k} Bank_i[A], \tag{14}$$

Substituting $Bank_k[\mathrm{A}]$ from (12) into (14) provides

$$RBankSel_r = k \oplus_{0 \le i < n_W;\, i \ne k} Bank_i[\mathrm{A}] \oplus_{0 \le i < n_W;\, i \ne k} Bank_i[A]. \tag{15}$$

The last two series in (15) can be reduced revealing that $RBankSel_r = k$, the ID of the latest writing bank into address A, as required.

Figure 9 provides an example of 2W/2R binary-coded I-LVT. (As will become apparent in the next section, when there are only 2 write ports, the binary-coded and one-hot-coded I-LVTs are identical.) Figure 10 shows a 3W/2R binary-coded I-LVT.

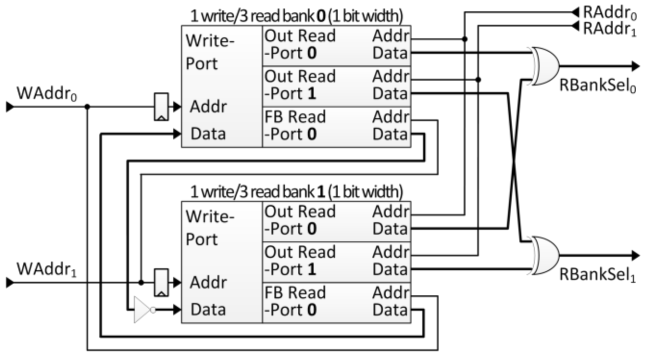

Figure 9. A 2W/2R SRAM-based I-LVT; identical for binary-coded or one-hot-coded bank selectors.

The required data width of the I-LVT SRAM blocks is $\lceil \log_2 n_W \rceil$. Also, n_W multi-read banks are required each with n_R output ports for ID extraction and $n_W - 1$ feedback ports for ID rewriting. Hence, the number of required SRAM cells is

$$d \cdot \lceil \log_2 n_W \rceil \cdot n_W \cdot (n_W + n_R - 1). \tag{16}$$

Respectively, the number of required M20k block RAMs is

$$n_{M20K}(d, \lceil \log_2 n_W \rceil) \cdot n_W \cdot (n_W + n_R - 1). \tag{17}$$

Similarly, the number of registers required for retiming is

$$n_W \cdot (\lceil \log_2 d \rceil + 1). \tag{18}$$

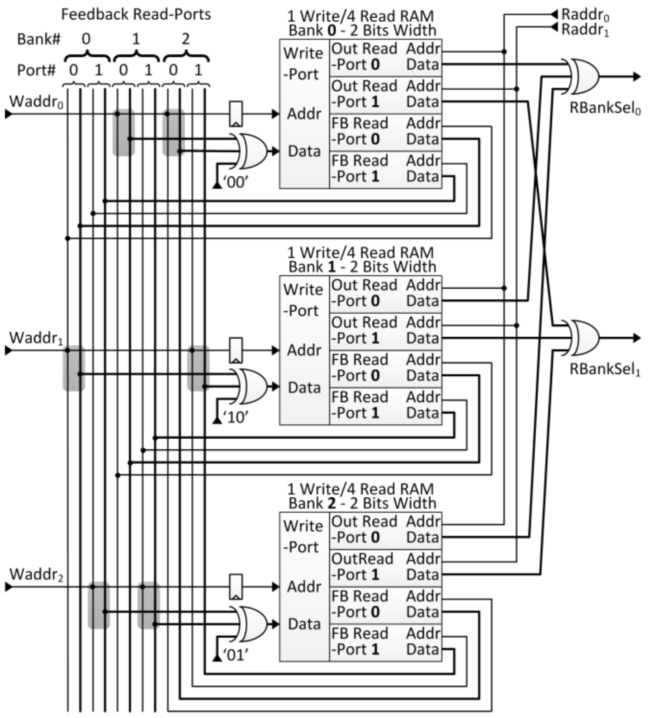

Figure 10. A 3W/2R SRAM-based I-LVT with binary-coded selectors.

4.2 Mutual-exclusive Conditions: One-hot-coded Selectors

The previous binary-coded I-LVT incurs a long path delay through the feedback and output extraction functions, which causes a performance reduction in structures with more ports. Employing a one-hot thermometer ID encoding reduces the feedback paths to just a few inverters, from the n_W-wide XOR used earlier.

Mutual-exclusive conditions are used to rewrite the RAM contents. A specific bank is written data that contradicts all the other banks, hence only this specific bank will be valid and all the others are invalid. By checking the appropriate mutual-exclusive condition for each bank, only the latest written bank will hold the valid data.

Equations (19), (20), and (21) describe mutual-exclusive feedback functions for n_W values 1, 2, and 3, respectively. The angle brackets in theses equations are used for bit selection and concatenation, while the square brackets in other equations are used for RAM addressing. As can be seen from these equations, writing to one bank will invalidate all the other banks at the same address since one mutual negated bit is shared between each two lines. For example, writing to bank 2 when $n_W = 3$ (Equation (20)) will write $Bank_1\langle 0\rangle \leftarrow \overline{Bank_0\langle 0\rangle}$ which will invalidate bank 0, and $Bank_1\langle 1\rangle \leftarrow Bank_2\langle 1\rangle$ which will invalidate bank 2.

$$n_W = 2: \begin{cases} f_{fb,0}: Bank_0\langle 0\rangle \leftarrow Bank_1\langle 0\rangle \\ f_{fb,1}: Bank_1\langle 0\rangle \leftarrow \overline{Bank_0\langle 0\rangle} \end{cases} \quad (19)$$

$$n_W = 3: \begin{cases} f_{fb,0}: Bank_0\langle 1{:}0\rangle \leftarrow \langle Bank_2\langle 0\rangle, Bank_1\langle 0\rangle\rangle \\ f_{fb,1}: Bank_1\langle 1{:}0\rangle \leftarrow \langle Bank_2\langle 1\rangle, \overline{Bank_0\langle 0\rangle}\rangle \\ f_{fb,2}: Bank_2\langle 1{:}0\rangle \leftarrow \langle \overline{Bank_1\langle 1\rangle}, \overline{Bank_0\langle 1\rangle}\rangle \end{cases} \quad (20)$$

$$n_W = 4: \begin{cases} f_{fb,0}: Bank_0\langle 2{:}0\rangle \leftarrow \langle Bank_3\langle 0\rangle, Bank_2\langle 0\rangle, Bank_1\langle 0\rangle\rangle \\ f_{fb,1}: Bank_1\langle 2{:}0\rangle \leftarrow \langle Bank_3\langle 1\rangle, Bank_2\langle 1\rangle, \overline{Bank_0\langle 0\rangle}\rangle \\ f_{fb,2}: Bank_2\langle 2{:}0\rangle \leftarrow \langle Bank_3\langle 2\rangle, \overline{Bank_1\langle 1\rangle}, \overline{Bank_0\langle 1\rangle}\rangle \\ f_{fb,3}: Bank_3\langle 2{:}0\rangle \leftarrow \langle \overline{Bank_2\langle 2\rangle}, \overline{Bank_1\langle 2\rangle}, \overline{Bank_0\langle 2\rangle}\rangle \end{cases} \quad (21)$$

Equation (22) generalizes the feedback function to

$f_{fb,k}\langle i\rangle\big|_{0 \le i < n_W - 1}$:

$$Bank_k[WAddr_k]\langle i\rangle \leftarrow \begin{cases} i < k & \overline{Bank_i[WAddr_k]\langle k-1\rangle} \\ else & Bank_{i+1}[WAddr_k]\langle k\rangle \end{cases}. \quad (22)$$

This equation shows that each bank is using bits from all other banks to write its own content. To prove that each two banks are mutually exclusive, one bit of these banks should be mutually negated. Suppose $0 \le k_0 \le n_W - 1$ a bank ID, and $0 \le i_0 \le n_W - 1$ a bit index. From Equation (22) if $i_0 \ge k_0$ then another bank ID k_1 and bit index i_1 exist such that $Bank_{k_0}\langle i_0\rangle \leftarrow Bank_{k_1}\langle i_1\rangle$, $k_1 = i_0 + 1$, and $i_1 = k_0$. Hence, $i_1 < k_1$ and from (22) $Bank_{k_1}\langle i_1\rangle \leftarrow \overline{Bank_{k_0}\langle i_0\rangle}$ as required. The proof in case of $i_0 < k_0$ is identical.

The output extraction function checks for each one-hot output selector if the read data from a specific bank matches the mutual-exclusive case. Hence, only one case will match due to exclusivity. The output extraction function consists of an $n_W - 1$ bit wide comparator for each one-hot selector.

An example of a 2W/2R one-hot-coded I-LVT is shown in Figure 9, while a 3W/2R one-hot I-LVT is depicted in Figure 11.

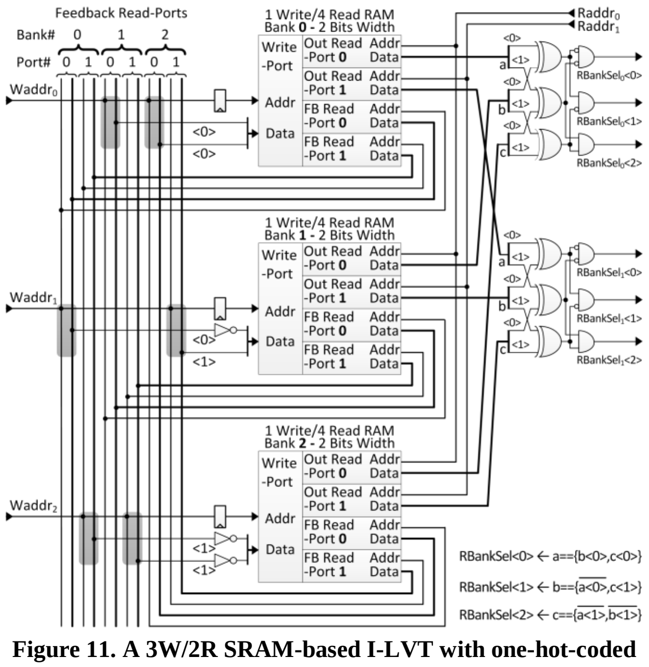

Figure 11. A 3W/2R SRAM-based I-LVT with one-hot-coded selectors.

The one-hot-coded I-LVT requires $n_W - 1$ SRAM bits to save the mutually exclusive cases. However, the feedback read ports requires only one bit, since only one bit is used by the feedback function from each bank. n_W multi-read banks are required each with n_R output ports for one-hot selectors extraction and $n_W - 1$ feedback ports for mutually exclusive cases rewriting. Hence, the number of required SRAM cells is

$$d \cdot (n_W - 1) \cdot n_W \cdot n_R + d \cdot n_W \cdot (n_W - 1). \quad (23)$$

Respectively, the number of required M20k block RAMs is

$$n_{M20K}(d, n_W - 1) \cdot n_W \cdot n_R + B_{M20K}(d, 1) \cdot n_W \cdot (n_W - 1). \quad (24)$$

Similarly, the number of registers required for retiming is equal to the binary-coded case and is described by (18).

4.3 Data Dependencies and Bypassing

The new I-LVT structure and the previous XOR-based multi-ported RAMs incur some data dependencies due to feedback functions and the latency of reading the I-LVT to decide about the last written bank. Data dependencies can be handled by employing bypassing, also known as forwarding.

Figure 12 illustrated two types of bypassing based on write data and address pipelining. Bypassing is necessary because Altera dual-port block RAMs cannot internally forward new data when one port reads and the other port writes the same address on the same clock edge, constituting a read-during-write (RDW) hazard. Both bypassing techniques are functionally equivalent, allowing reading of the data that is being written on the same clock edge, similar to single register functionality. However, the fully-pipelined two-stages bypassing shown in Figure 12 (right) can overcome additional cycle latency. This capability is required if a block RAM has pipelined inputs (*e.g.*, cascaded from another block RAM) that need to be bypassed.

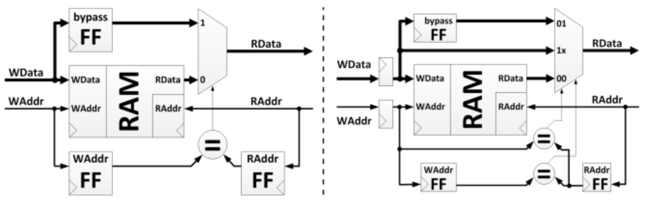

Figure 12. RAM bypassing (left) single-stage (right) 2-stages fully pipelined.

The single-stage and the two-stage bypass circuitry for a w bits data width and d lines depth block RAM requires w registers for data bypassing, two $\lceil \log_2 d \rceil$ wide address registers and one enable register, for a total of

$$n_{BypReg}(d, w) = w + 2\lceil \log_2 d \rceil + 1. \quad (25)$$

The most severe data dependency that I-LVT design suffers from is write-after-write (WAW), namely, writing to the same address that has been written before in the previous cycle. This dependency occurs because of the feedback reading and writing latency. A single-stage bypassing for the feedback banks should solve this dependency.

Two types of reading hazards are introduced by the proposed I-LVT design, read-after-write (RAW) and read-during-write (RDW). RAW occurs when the same data that have been written in the previous clock edge are read in the current clock edge. RDW occurs when the same data are written and read on the same clock edge.

Due to the latency of the I-LVT, reading from the same address on the next clock edge after writing (RAW) will provide the old data. To read the new data instead, the output banks of the I-LVT should be bypassed by a single-stage bypass to overcome the I-LVT latency.

The deepest bypassing stage is reading new data on the same writing clock edge (RDW), which is similar to single register stage latency. This can be achieved by 2-stage bypassing on the output extract ports of the I-LVT or the XOR-based design to allow reading on the same clock edge. The data banks, which are working in parallel with the I-LVT should also be bypassed by a single-stage bypass to provide new data. Table 2 summarizes the required bypassing for data banks, feedback banks and output banks for each type of bypassing of the XOR-based, binary-coded and one-hot-coded I-LVT designs.

Since XOR-based multi-ported RAM requires bypassing for all the $n_W \cdot (n_W + n_R - 1)$ banks to read new data when RAW or RDW, the additional registers required for the bypassing are

$$n_W \cdot (n_W + n_R - 1) \cdot n_{BypReg}(d, w). \qquad (26)$$

RAW for binary-coded method requires bypassing the I-LVT only. Since the I-LVT is built out of $n_W \cdot (n_W + n_R - 1)$ blocks, each with $\lceil \log_2 n_W \rceil$ bits width data, the following amount of additional registers is required

$$n_W \cdot (n_W + n_R - 1) \cdot n_{BypReg}(d, \lceil \log_2 n_W \rceil). \qquad (27)$$

RAW for one-hot-coded method requires bypassing the whole I-LVT, $n_W \cdot (n_W - 1)$ feedback banks with 1 bit width and $n_W \cdot n_R$ output banks with $n_W - 1$ bits width, hence a total registers of

$$n_W \cdot (n_W - 1) \cdot n_{BypReg}(d, 1) + n_W \cdot n_R \cdot n_{BypReg}(d, n_W - 1). \qquad (28)$$

RDW for both binary and one-hot-coded methods require bypassing the $n_W \cdot n_R$ data banks in addition to the I-LVT, hence the following amount of registers is added to the previous count in (27) and (28)

$$n_W \cdot n_R \cdot n_{BypReg}(d, w). \qquad (29)$$

Table 2. Bypassing for XOR-based and binary/one-hot-coded I-LVT multi-ported memories

	XOR-based		I-LVT based		
	Feedback banks	*Output banks*	*Data banks*	*Feedback banks*	*Output banks*
Allow WAW	1-stage	None	None	1-stage	None
New data RAW	1-stage	1-stage	None	1-stage	1-stage
New data RDW	1-stage	2-stage	1-stage	1-stage	2-stage

4.4 Initializing Multi-ported RAM Content

Due to the special structure of the proposed I-LVT-based multi-ported memories and the previously proposed XOR-based method, RAM data may have replicas in several banks. Hence, initializing the multi-ported RAM with a specific content requires special handling.

For the XOR-based multi-ported RAM, the first multi-read bank should be initialized to the required initial content; all the other multi-read banks should be initialized to zero.

The binary/one-hot-coded I-LVT-based multi-ported RAM requires initializing all the I-LVT banks with zeros. The binary-coded I-LVT will generate a selector to the first data bank (indexed zero), since XOR'ing all the initial values (zeros) will generate zero. Similarly, the one-hot-coded I-LVT will be initialized to the first mutually exclusive case, hence the first bank will be selected. Only the first data bank should hold the initial data; the remaining banks can be left uninitialized. The initial values for each bank in the binary/one-hot-coded I-LVT-based and XOR-based designs are shown in Figure 13.

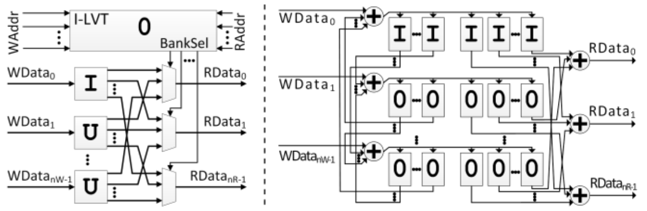

Figure 13. Initial value for each bank (left) I-LVT-based (right) XOR-based. Initial values are 0: zeros, I: initial content, and U: uninitialized.

4.5 Comparison and Discussion

In this section, we compare the previous LVT and XOR approaches to the new I-LVT approaches for building multi-port memories. Using the equations provided, we will illustrate why the I-LVT approach is superior in terms of number of BRAMs required, and number of registers required. Also, between the two I-LVT methods proposed, we will inspect the number of BRAMs and registers used by each bypassing method.

Table 3 summarizes SRAM resource usage for each of the three multi-ported RAM approaches: the XOR-based and the binary/one-hot-coded I-LVT. Both the general SRAM cell count and the number of Altera's M20K blocks are described. Comparing the SRAM cell counts, the XOR-based approach consumes fewer SRAM cells than the one-hot I-LVT if

$$w < n_R + 1. \qquad (30)$$

This inequality is unlikely to be satisfied, since even if the data width is only one byte, the number of reading ports n_R would need to be larger than 8, which is very rare. Hence, for most of the common cases, the one-hot-coded I-LVT approach will consume fewer SRAM cells.

Comparing the XOR-based approach to the binary-coded approach, the XOR-based approach consumes fewer SRAM cells only if

$$w < \left. \frac{\lceil log_2(n_W) \rceil \cdot (n_W + n_R - 1)}{(n_W - 1)} \right|_{n_W > 1}. \qquad (31)$$

Both (30) and (31) show that the XOR-based approach will consume less SRAM cells only for a very narrow data widths which are uncommonly used. Hence, the I-LVT approach will be the choice for most applications. Comparing the two I-LVT approaches, Table 3 shows that the one-hot-coded I-LVT consumes fewer SRAM cells than the binary-coded I-LVT if

$$1 < n_W \leq 3 \;\; OR \;\; n_R < \left. \frac{(n_W - 1) \cdot (\lceil log_2(n_W) \rceil - 1)}{(n_W - 1) - \lceil log_2(n_W) \rceil} \right|_{n_W > 3}. \qquad (32)$$

Table 4 summarizes register usage for all multi-ported RAM architectures and bypassing. Only the register-based LVT architecture is directly proportional to memory depth. As a consequence, it consumes much more registers than other architectures, making register-based LVTs impractical for deep memories.

Table 3. Summary of SRAM-based memory usage

	SRAM bits	**M20K blocks**[1]
Register-based LVT	$d \cdot w \cdot n_W \cdot n_R$	$n_{M20K}(d,w) \cdot n_W \cdot n_R$
XOR-based	$d \cdot w \cdot n_W \cdot n_R + d \cdot w \quad \cdot n_W \cdot (n_W - 1)$	$n_{M20K}(d,w) \cdot n_W \cdot n_R + n_{M20K}(d,w \quad) \cdot n_W \cdot (n_W - 1)$
Binary-coded I-LVT	$d \cdot w \cdot n_W \cdot n_R + d \cdot \lceil \log_2 n_W \rceil \cdot n_W \cdot (n_W - 1) + d \cdot \lceil \log_2 n_W \rceil \cdot n_W \cdot n_R$	$n_{M20K}(d,w) \cdot n_W \cdot n_R + n_{M20K}(d, \lceil \log_2 n_W \rceil) \cdot n_W \cdot (n_W - 1) + n_{M20K}(d, \lceil \log_2 n_W \rceil) \cdot n_W \cdot n_R$
One-hot-coded I-LVT	$d \cdot w \cdot n_W \cdot n_R + d \quad \cdot n_W \cdot (n_W - 1) + d \cdot (n_W - 1) \cdot n_W \cdot n_R$	$n_{M20K}(d,w) \cdot n_W \cdot n_R + n_{M20K}(d,1 \quad) \cdot n_W \cdot (n_W - 1) + n_{M20K}(d, n_W - 1 \quad) \cdot n_W \cdot n_R$

Table 4. Summary of register usage

	No bypass	**Additional registers for single-stage**[2]	**Additional registers for two-stage**
Register-based LVT	$d \cdot \lceil \log_2 n_W \rceil$	None	None
XOR-based	$n_W \cdot (\mathrm{w} + \lceil \log_2 d \rceil + 1)$	$n_W \cdot (n_W - 1) \cdot n_{BypReg}(d,w \quad) + n_W \cdot n_R \cdot n_{BypReg}(d,w \quad)$	None
Binary-coded I-LVT	$n_W \cdot (\quad \lceil \log_2 d \rceil + 1)$	$n_W \cdot (n_W - 1) \cdot n_{BypReg}(d, \lceil \log_2 n_W \rceil) + n_W \cdot n_R \cdot n_{BypReg}(d, \lceil \log_2 n_W \rceil)$	$n_W \cdot n_R \cdot n_{BypReg}(d,w)$
One-hot-coded I-LVT	Same as Binary-coded	$n_W \cdot (n_W - 1) \cdot n_{BypReg}(d,1 \quad) + n_W \cdot n_R \cdot n_{BypReg}(d, n_W - 1 \quad)$	Same as Binary-coded

[1] $B_{M20K}(d,w)$ is the number of Altera's M20K blocks required to construct a RAM with *d* lines depth and *w* bits width and is described in (4).

[2] $Reg_{bypass}(d,w)$ is the number of additional registers required to bypass a RAM with *d* lines depth and *w* bits width and is described in (25).

With a single-stage bypassing, the XOR-based design consumes fewer registers than the binary-coded if

$$w < \lceil log_2(n_W) \rceil. \tag{33}$$

Equation (33) is unlikely to be satisfied. Even if the data width is just one byte ($w = 8$), the number of write ports n_W would need to be larger than 256, which is impractical.

On the other hand, with a single-stage bypass, the XOR-based design consumes fewer registers than the one-hot-coded I-LVT design if

$$w < \left. \frac{1+n_R}{1+\frac{n_R}{n_W-1}} \right|_{n_W>1}. \tag{34}$$

In a typical compute-oriented designs, $n_R = 2 \cdot n_W$. Assuming that $n_R = 2 \cdot (n_W - 1)$ requires that $3 \cdot w - 1 < n_R$; even for a one byte data width, this requires $23 < n_R$ to satisfy (34), which is impractical. Therefore, for a single-stage bypass, the I-LVT based designs will consume fewer registers than the XOR-based design.

Considering two-stage bypassing, I-LVT based designs will consume $n_W \cdot n_R \cdot Reg_{bypass}(d,w)$ more registers, as described in (29). In this case, XOR-based design consumes fewer registers than the binary-coded I-LVT design only if

$$w < \lceil log_2(n_W) \rceil \cdot \left(1 + \frac{n_R}{n_W - 1}\right). \tag{35}$$

On the other hand, XOR-based design consumes fewer registers than the one-hot-coded I-LVT design only if

$$w < n_R + 1. \tag{36}$$

Similar to (30), which is equal to (36), this is unlikely to be satisfied in practical designs. Hence, in the case of two-stage bypassing, the I-LVT-based design will consume fewer bypassing registers than the XOR-based method.

In the next section, we will show these analytical results are in agreement with experimental results.

5. EXPERIMENTAL RESULTS

In order to verify and simulate the suggested approach and compare to previous attempts, fully parameterized Verilog modules have been developed. Both the previous XOR-based multi-ported RAM method, and the proposed I-LVT method have been implemented. To simulate and synthesize these designs with various parameters in batch using Altera's ModelSim and Quartus II, a run-in-batch flow manager has also been developed. The Verilog modules and the flow manager are available online [5]. To verify correctness, the proposed architecture is simulated using Altera's ModelSim. A large variety of different memory architectures and parameters are swept, *e.g.* bypassing, memory depth, data width, number of reading or writing ports, and simulated in batch, each with over million random memory access cycles.

All different multi-ported design modules were implemented using Altera's Quartus II on Altera's Stratix V 5SGXMA5N1F45C1 device. This is a high-performance device with 185k ALMs, 370k ALUTs, 2304 M20K blocks and 1064 I/O pins.

We performed a general sweep and tested all combinations of configurations of the following parameters:

- Writing ports (n_W): 2, 3 and 4 writing ports.
- Reading ports (n_R): 3, 4, 5 and 6 reading ports.
- Memory depth (d): 16 and 32 K-lines.
- Data width (w): 8, 16, and 32 bits.
- Bypassing: No bypassing, single-stage and two-stages.

Following this, we analyzed the full set of results. In this paper, we omit many of the in-between settings because they behaved as one might expect to see via interpolation of the endpoints.

Figure 14 plots the maximum frequency derived from Altera's Quartus II STA at 0.9V and temperature of 0 °C. The results show a higher Fmax for binary/one-hot coded I-LVT compared to XOR-based approach for all design cases. With 3 or more writing ports, the one-hot-coded I-LVT supports a higher frequency compared to all other design styles. Compared to the XOR-based approach, the one-hot-coded I-LVT improves Fmax by 38% on average for all tested design configurations, while the maximum Fmax improvement is 76%.

Figure 15 (top) plots the number of Altera's M20K blocks used to implement each multi-ported RAM configuration. The proposed binary/one-hot-coded I-LVT consumes the least BRAM blocks in all cases. The average reduction of the best of binary/one-hot-coded I-LVT compared to XOR-based approach is 19% for all tested design configurations, while it can reach 44% for specific configurations. The difference of consumed Altera's M20Ks between binary-coded I-LVT and one-hot-coded I-LVT is less than 6%. Both I-LVT methods make a significant improvement in BRAM consumption, but binary-coded I-LVT consumes the least BRAMs for more than 3 writing ports. To clarify the difference in

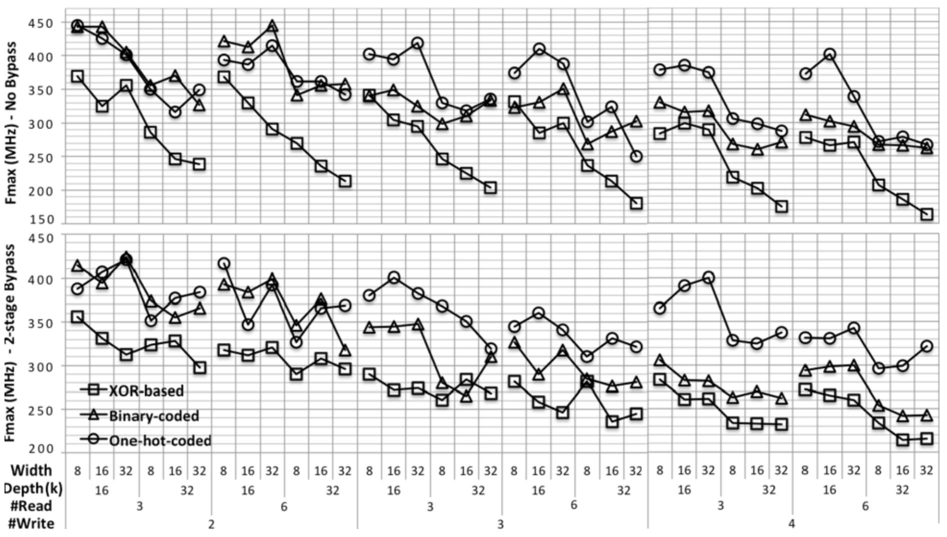

Figure 14. Fmax (MHz) T=0C (top) No bypass (bottom) Two-stage bypass.

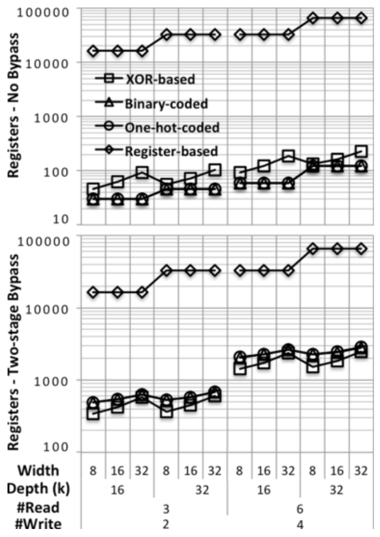

Figure 16. Registers (top) no-bypass (bottom) two-stage bypass.

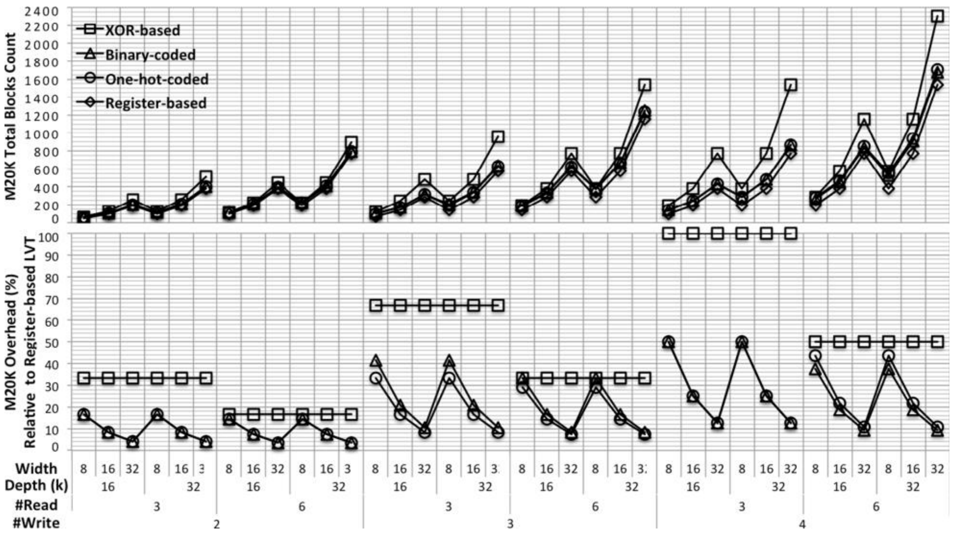

Figure 15. M20K blocks (top) total count (bottom) overhead percentage relative to register-based LVT.

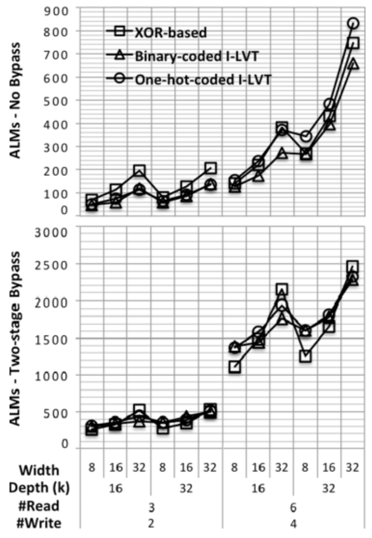

Figure 17. ALMs (top) no-bypass (bottom) two-stage bypass.

BRAM consumption, Figure 15 (bottom) shows the percentage of BRAM overhead above the register-based LVT, which uses the fewest possible BRAMS overall. The XOR-based design consumes more BRAMs in all cases, up to 100% more than the register-based LVT. On the other hand, I-LVT-based methods consume only 12.5% more BRAMs in the case of 32-bit wide memories.

The number of registers required for various designs and bypassing styles is shown in Figure 16. The I-LVT-based methods consume fewer registers compared to the XOR-based method for no bypassing or single-stage bypass cases. For two-stage bypassing, the I-LVT based methods must bypass the data banks, hence the register consumption goes higher than the XOR-based method. However, the register consumption of the register-based LVT method is the highest overall and can be three orders of magnitude higher since it is directly proportional to memory depth. Furthermore, some register-based LVT configurations failed to synthesize on our Stratix V with 185k ALMs.

Figure 17 shows the number of ALMs consumed by each design with different bypassing methods. Single-stage bypassing consumes more ALMs than the non-bypassed version due to address comparators and data muxes. On the other hand, two-stage bypassing requires an additional address comparator and a wider mux; hence it consumes more ALMs than a single-stage bypass. In all bypass modes, as memory data width goes higher, the XOR-based method consumes more ALM's than the I-LVT methods due to wider XOR gates.

Since the register-based LVT approach is not feasible with the provided deep memory test-cases, the register-based LVT trends are derived analytically from Table 3 and 4 and not from experimental results. Hence, the register-based LVT trend was added as a reference baseline to Figure 15 and 16 only.

6. CONCLUSIONS AND FURTHER DIRECTIONS

In this paper, we have proposed the use of an invalidation-based live-value-table, or I-LVT, to build modular SRAM-based multi-ported memories. The I-LVT generalizes and replaces two prior techniques, the LVT and XOR-based approaches. A general I-LVT is described, along with two specific implementations: binary-coded and one-hot-coded. Both methods are purely SRAM based. A detailed analysis and comparison of resource consumption of the suggested methods and previous methods is provided. The original LVT approach can use an infeasible number of registers. Unlike the LVT, the I-LVT register usage is not directly proportional to memory depth; hence it requires magnitudes fewer registers. Furthermore, the proposed I-LVT method can reduce BRAM consumption up to 44% and improve Fmax by up to 76% compared to the previous XOR-based approach. The one-hot-coded I-LVT method exhibits the highest Fmax, while keeping BRAM consumption within 6% of the minimal required BRAM count. Meanwhile, the binary-coded I-LVT uses fewer BRAMs than the one-hot coded when there are more than 3 write ports.

A fully parameterized and generic Verilog implementation of the suggested methods is provided as open source hardware [5]. As future work, the suggested multi-ported memories can be tested with various other FPGA vendors' tools and devices. Furthermore, these methods can also be tested for ASIC implementation using dual-ported RAMs as building blocks, and compared against memory compiler results. Also, to improve Fmax, time-borrowing techniques can be utilized. The goal would be to recover the frequency drop due to the multi-ported RAM additional logic, feedback and bank selection logic. One possible approach uses shifted clocks to provide more reading and writing time [17]. However, adapting this method to multi-ported memories is not trivial due to internal timing paths across the I-LVT.

7. ACKNOWLEDGMENTS

The authors would like to thank Altera for donations of hardware and software, as well as NSERC for funding.

8. REFERENCES

[1] J.H. Tseng and K. Asanovic, "Banked multiported register files for high-frequency superscalar microprocessors," *Int'l Symp. on Computer Architecture (ISCA)*, May 2003, pp. 62–71.

[2] J.A. Fisher, "Very Long Instruction Word architectures and the ELI-512," *Int'l Sym. on Comp. Arch. (ISCA)*, 11(3), June 1983.

[3] E.S. Fetzer and J.T Orton, "A fully-bypassed 6-issue integer datapath and register file on an Itanium microprocessor," *IEEE Int'l Solid-State Circuits Conf.*, vol. 1, Feb. 2002, pp. 420–478.

[4] H. Bajwa and X. Chen, "Low-Power High-Performance and Dynamically Configured Multi-Port Cache Memory Architecture," *Int'l Conf. on Elec. Eng.*, Apr. 2007, pp. 1–6.

[5] http://www.ece.ubc.ca/~lemieux/downloads/

[6] B.A. Chappell, T.I Chappell, M.K. Ebcioglu, and S.E. Schuster, "Virtual multi-port RAM employing multiple accesses during single machine cycle," *U.S. Patent 5 542 067*, Jul. 30, 1996.

[7] H. Yokota, "Multiport memory system," *US Patent 4 930 066*, May 29, 1990.

[8] Ditlow *et al.*, "A 4R2W register file for a 2.3GHz wire-speed POWER™ processor with double-pumped write operation," *IEEE Int'l Solid-State Circuits Conf.*, Feb. 2011, pp. 256–258.

[9] R.E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar./Apr. 1999.

[10] H.J. Mattausch, "Hierarchical N-port memory architecture based on 1-port memory cells," *European Solid-State Circuits Conference (ESSCIRC '97)*, Sept. 1997, pp. 348–351.

[11] J. Weixing, S. Feng, Q. Baojun Qiao, and S. Hong, "Multi-port Memory Design Methodology Based on Block Read and Write," *IEEE Int'l Conference on Control and Automation)*, May 2007.

[12] Z. Wang Zuo, "An Intelligent Multi-Port Memory," *Journal of Computers*, vol. 5, no. 3, pp. 471–478, Mar 2010.

[13] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, 13(3), pp. 11–21, June 1993.

[14] C.E. LaForest and J.G. Steffan, "Efficient Multi-ported Memories for FPGAs," *ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays (FPGA '10)*, Feb. 2010.

[15] C.E. LaForest, M.G. Liu, E.R. Rapati, and J.G. Steffan, "Multi-ported memories for FPGAs via XOR," *ACM Int'l Symp. on Field-Programmable. Gate Arrays (FPGA '12)*, Feb. 2012, pp. 209–218.

[16] Altera Corporation, *Stratix V Device Handbook*, June 2011.

[17] A. Brant, A. Abdelhadi, A. Severance, G. Lemieux, "Pipeline Frequency Boosting: Hiding Dual-Ported Block RAM Latency using Intentional Clock Skew," *IEEE International Conference on Field-Programmable Technology (FPT)*, December 2012.