Development Environment for a RISC-V processor: Cache

João Vieira Rodrigues de Almeida Roque *Electrical and Computer Engineering Department*
Instituto Superior Técnico
Lisbon, Portugal
joao.v.roque@tecnico.ulisboa.pt

***Abstract*—Despite the recent advent of open-source hardware, the available open-source caches have low configurability, limited lack of support for single-cycle pipelined memory accesses, and use non-standard hardware interfaces and Hardware Description Languages. In this work IOb-Cache, a high-performance configurable open-source Verilog cache is proposed and developed. The cache is designed modular and composed of 3 modules. The Cache-Memory module contains the memories and the cache's main controller. The Front-End and Back-End modules isolate the cache design from the processor and memory interfaces, respectively, which enables the fast adoption of new processors or memory controllers. Currently the Front-End module supports the native interface and the Back-End module supports the native and the standard Advanced eXtensible Interface (AXI) interfaces. The cache can be configured to define the number of ways (k) in set-associative designs (k = 1 selects a direct-mapped design), the number of lines and words per line, the replacement policy, etc. The write-policy is currently fixed to Write-Through Not Allocate policy with an internal buffer, limiting the write accesses to word-sized data. The back-end can be configured to read bursts of multiple words per transfer to take advantage of the available memory bandwidth. To the best of our knowledge, IOb-Cache is currently the only configurable Verilog cache that supports pipelined Central Processing Unit (CPU) interfaces and the popular AXI memory bus interface. IOb-Cache is integrated into IOb-SoC Github repository, which has 16 stars, and is being used in 38 projects (forks).**

***Index Terms*—Open-source, Cache, Highly Configurable, Pipeline, AXI, Native.**

I. Introduction

As open-source processors such as the RISC-V architecture become adopted by the industry and compete with commercial solutions such as ARM, the community rushes towards creating the ecosystem for these CPUs to thrive on. These include not only different CPU architectures with different performance, size, and power consumption, but also efficient memory systems, peripherals, and interfaces of all sorts. The software part is even more important as, without compelling user applications and programming tools, no sustainable business can be built with open-source CPUs.

One such key component is a truly configurable cache module, able to support multiple architectural trade-offs. After analysis of the available open-source caches, one finds limitations of the interfaces (no support for de facto standards such as AXI [1]), lack of support for single-cycle pipelined memory accesses, and use of exotic and non-standard Hardware Description Language (HDL), limiting the use of simulation and synthesis tools.

Hence, the need to develop a high-performance configurable cache using a standard bus interface and a standard HDL such as Verilog [2] became evident for IObundle, a company started in 2018 with the aim of developing domain-specific RISC-V Intellectual Property (IP) systems for low power embedded devices. The candidate carried out this work both in the scope of his master's dissertation and as a trainee at IObundle.

The development of a high-performance configurable open-source cache in Verilog with the popular Advanced eXtensible Interface 4th generation (AXI4) interface [1] in the back-end, and with better features than the existing ones is the main objective of this dissertation. To attain this objective the following sub-objectives are pursued:

- Support pipeline architectures. The cache must fulfill 1 request per clock cycle while keeping stalls to a bare minimum. This requires a well designed datapath to correctly implement the cache operation while guaranteeing that loads and store instructions execute in one cycle. A new request and the response to the previous request must be superimposed, given the 1-cycle latency of the RAM modules that constitute the cache memories.
- Modular design. The cache must be composed of 3 modules: front-end; cache core and back-end. This makes it easy to replace the front-end and back-end interfaces, while keeping the core functionality intact, if needed.
- The back-end must implement the Native and AXI interfaces. The flexibility of the back-end interface is indispensable as it can be connected to higher level caches using the Native interface or to 3rd party memory controllers which are likely to be using an AXI interface.
- The back-end must be configured with a different data-width from that of the front-end (asymmetric memory) to take advantage of the available memory bandwidth. Many memory controllers allow wide data buses from designs that need to work at a lower frequency while using a much higher frequency to communicate with the external memory.

In search of HDL open-source cache designs, the most relevant ones are found on the Github platform. The caches on Github are chosen based on their popularity (stars and forks): airin711's Verilog-caches [3]; prasadp4009's 2-way-Set-Associative-Cache-Controller [4]; and PoC.cache, which is part of the Pile-of-Cores (PoC)-Library [5], one of the most popular HDL libraries.

The airin711's Verilog-caches repository houses 3 different set-associative caches: 4-way with Least-Recently-Used (LRU) replacement policy; 8-way with Pseudo-Least-Recently-Used (PLRU) and a run-time configurable 2-to-8-way with PLRU replacement policy. All caches have 4 words per line and only allow configuring the number of lines.

The prasadp4009's Verilog cache repository is a 2-way set-associative cache that uses the LRU replacement policy. Unlike the airin771's caches, it allows configuration of the number of cache lines and words per line, as well as the width of both address and data.

Both caches the airin711's and prasadp4009's caches use write-back write-allocate policy, native memory interface and are unable to either invalidate or flush a cache line. The biggest difference between the two is the fact that airin711's caches require the data memory to be 128-bit wide so that the entire line or memory block can be accessed in a single word transfer. The prasadp4009's cache requires the data memory width to be word-sized, using a counter to receive the memory block or transfer the cache line.

Unfortunately, there is a big issue, which makes these two caches poor choices with more advanced architectures: they need at least 2 clock cycles to process requests, even if the data is already in the cache. These caches implement a Finite State Machine (FSM) that controls all their functions, including the communication with the processor and to the main memory. They only allow requests when their FSM is in the initial state, and have special states for the read and write accesses because of the RAM's 1 cycle read latency upon a cache hit. In the read or write access, if a hit occurs, they acknowledge the request and go back to the initial state. This causes the undesirable 1 clock-cycle latency.

The third cache that was investigated, PoC.cache [5], does allow 1 read-access per clock cycle, which is a big improvement over the others. It is also highly configurable, with various synthesis parameters that characterize the cache dimensions, even allowing 3 types of mapping: direct, set-associative, and full-associative. It uses the LRU replacement policy, an effective but costly policy, especially when compared to others that are cheaper but closely effective. Despite only having a native memory interface, it has access to adapters for other commonly used memories, especially SDRAMs.

The disadvantages of PoC.cache are described in this paragraph. PoC.cache uses a control FSM, which during a read-access, only changes state on a miss. This requires the hit to be checked in the same cycle a request is made. The tag and valid memories are therefore implemented with distributed RAM and registers, respectively. In the presence of a hit, PoC.cache acknowledges the request, but the data is only available in the next clock cycle. In the following clock cycle, when the data is available, it can receive a new request. This allows the cache to operate with the 1 read per clock cycle. Write accesses on the other hand require a change of state in the FSM, resulting in a minimum of 1 write per 2 clock cycles. The cache uses a write-through write-not-allocate policy but does not have a write-through buffer. Instead, it accesses the main memory directly. This means each write-access is dependent on the write-access time of the memory interface controller, which is a big issue given that the write-through policy is expected to generate significant traffic.

Despite being highly configurable, its main memory interface is limited to the size of the cache line. The cache expects to load a line in a single transfer, meaning the memory's data-width needs to be line-sized. This is not a negative point since it maximizes the main memory's bandwidth, but may limit the memory options. One severe limitation is when implementing a multi-level cache, as the higher-level cache needs to have a word-size of the lower-level's line width. The lack of a write-through buffer is also a big limitation since this cache needs to stall during a write-access while the higher-level cache is fulfilling another request.

PoC.cache is written in VHSIC Hardware Description Language (VHDL), unlike the other caches that are written in Verilog. Depending on the synthesis and simulation tools, one language can be advantageous over the other but they are semantically equivalent. Most open-source tools only allow one HDL, generally Verilog, so the entire project needs to use the same language.

Compared to the presently developed IOb-Cache system, PoC.cache also lacks (1) a front-end module to avoid the need to implement an FSM for processor-cache communication; (2) a configurable back-end module that controls the communication with the main memory, freeing the main-controller of unnecessary delays; (3) a universally adapted memory interface like AXI. All these lacking features, plus the fact it written in VHDL, have motivated IObundle to develop a new open-source cache in Verilog.

II. IOb-Cache

IOb-Cache [6] is a configurable open-source pipelined-memory cache, designed in synthesizable Verilog HDL [2], for System-on-a-chip (SoC) implementation.

IOb-Cache is a very configurable IP core: it offers 2 different interfaces for the back-end memory, Native and AXI (4th generation), whose can be different from that of the front end (asymmetric implementation); it can be implemented as Directly mapped or K-Way Set-Associative; there are multiples line replacement policies to choose from, depending on the performance-resources needs. It uses a fixed write-through not-allocate policy,

Performance-wise, it allows 1 request/clock-cycle (pipelined). Each of the following chapters will describe its respective modules, behavior, and implementation.

A. IOb-Cache: top-level

The top-level integrates all the IOb-Cache modules, and is represented in the Fig. 1.

The Front-End connects the cache to a Master (processor). The ports always use the Native Interface, using a valid-ready protocol.

The Back-End connects the cache (master) to the main-memory (slave). Its interface depends on the choice of the top-level module: Native (iob_cache) or AXI (iob_cache_axi).

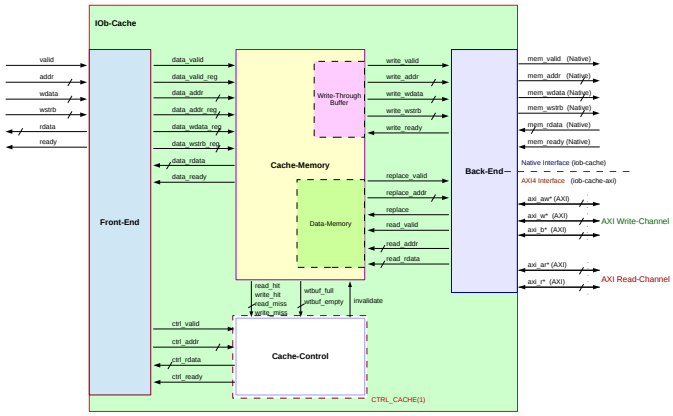

Fig. 1. IOb-Cache top-level module diagram.

The Cache-Memory is shown in between the Front-End and Back-End, and contains all the cache's memories and its main-controller.

Cache-Control is an optional module for an L1 cache that allows performing tasks such as invalidating a data cache, requesting its Write-Through Buffer's status, or analyzing its hit/miss performance. If the "CTRL_IO" macro is set, interfaces for invalidating the cache or observe the Write-Through-Buffer's status are implemented without the Cache-Control module, which is useful for cascading higher-level caches.

To configure the cache, many available synthesis parameters. "FE_ADDR_W" (Front-End Address Width) defines how many bytes are accessible in the main memory. "FE_DATA_W" (Front-End Data Width) defines the cache word-size, needs to be multiple of 8 (byte). "N_WAYS" sets the number of ways (power of 2), if 1 the cache is direct-mapped, if more it is set-associative mapped. "LINE_OFF_W" (Line Offset Width) defines the number of cache lines. "WORD_OFF_W" (Word Offset Width) defines the number of Words per cache line. "REP_POLICY" (replacement policy) sets the replacement policy in use in a set-associative cache. The policies available are LRU, Pseudo-Least-Recently-Used: Most-Recently-Used based (PLRUm), and Pseudo-Least-Recently-Used: (binary) tree-based (PLRUt). "WTBUF_DEPTH_W" (Write-Through-Buffer Depth Width), defines the number of positions in the write-through buffer's FIFO. "BE_ADDR_W" (Back-End Address Width) defines the width of the back-end address port, but the additional bits aren't accessible (access depends on FE_ADDR_W). "BE_DATA_W" (Back-End Data Width) defines Back-End's memory word-size, needs to be multiple of FE_DATA_W. "CTRL_CACHE" implements Cache-Control module, used for performance measurement, write-through buffer status and cache invalidation. "CTRL_CNT" implements Cache-Control's counters needed for performance measurement (hit and miss counters).

B. Front-End

The Front-End module interfaces the processor (master) and cache (slave). In the current design, it splits the processor bus to access the cache memory itself or the Cache-Control module (if present). It also registers some bus signals needed by the cache memory. Its interface is represented in Figure 2, detailing the internal structure. The signals with the "data" prefix are sent to Cache-Memory, and with "ctrl" to Cache-Control.

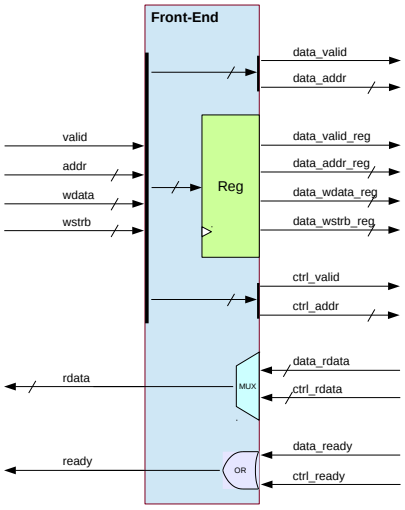

Fig. 2. Front-End module diagram.

The cache requires that during a request (valid), the master's inputs are maintained until the cache signals its conclusion by asserting the ready signal. During the assertion of the ready, a new access can be requested.

The signals required for memory writing and the ready's combinational path are registered. This way, the necessary input data is still available while ready is asserted.

The cache always returns entire words since it is word-aligned. This means the access is word-addressable, so the byte-offset of the CPU address signal (last $\log_2(\frac{\text{FE\_DATA\_W}}{8})$ bits) is not connected to the cache.

In a system with a different CPU interface, only this module requires modification to allow compatibility.

If the optional Cache-Control is implemented, this module also works as a memory-map decoder to select which unit is being accessed, the memory or the the control unit, using then Most Significant Bit (MSB) of the port "addr".

C. Cache-Memory

Cache-Memory is a module that contains the cache's main controller and memories. The available memories are the Tag memory, the Valid memory, the Data memory, the write-through buffer, and, if applicable, the Replacement-Policy memory.

Its main controller accesses specific back-end modules using a handshake valid-ready approach. The ready-signal (data_ready) is always asserted excepts after valid is asserted, where it becomes 0 until the request's conclusion.

Depending on the choice of parameters, the cache's implementation will either be direct-mapped or set-associative based on the number of ways given by N_WAYS.

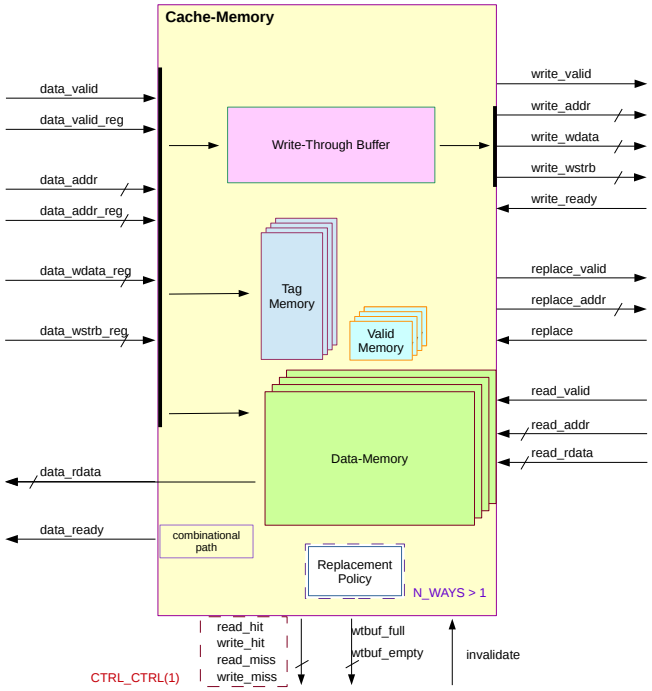

Fig. 3. Cache-Memory module diagram.

Before the Cache-Memory's behavior and design description is presented, the implementation of each memory will be explained.

The Tag memory is inferred using RAMs. There is one Tag memory per cache way. Each of these has Tag-sized width, and depth equal to the total number of cache lines.

The Valid memory is composed of an array of 1-bit registers (register-file), one for each way. Each array's length equals the number of cache lines. This choice of implementation was a simple design choice to set its contents to 0 during either a system reset or a cache-invalidate.

The Tag memory has a 1 clock-cycle read latency (Random-Access Memory (RAM)), therefore the valid memory's output signal needs to be delayed, either by applying a 1-bit output stage-register or using the registered address "data_addr_reg". The latter was chosen because it requires less logic and has no impact on timing. Both Tag and Valid memories' outputs connect to comparators for producing hit/miss results required for memory accesses.

The Data memory is implemented by one RAM for each way and (word) offset. Each RAM has a width FE_DATA_W (cache word-size) and a depth of $2^{\text{LINE\_OFF\_W}}$ (number of cache lines). Since the write-strobe signal selects which bytes are stored, each RAM requires a write enable for each byte. Some synthesis tools can only infer single-enable Block Random-Access-Memory (BRAM)s [7] [8], therefore a RAM will be inferred for each byte.

The Write-Though Buffer is implemented using a synchronous FIFO [9]. It requires the data to be available on its output a clock cycle after being read.

To address the words in the cache's memories, the input address signals are segmented as described in Figure 4.

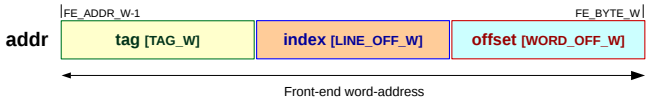

Fig. 4. Address signal segmentation.

The address (data_addr) is only used for the initial addressing (indexing) of the main memories: Valid, Tag, and Data. On the next clock cycle, the stored address (data_addr_reg) will be checked to see if a "hit" occurred (word available in cache), and identifying it within the cache line.

The hit check uses the signal "way_hit". Each of its bits indicates a hit in the respective way. The hit is the result of a tag match.

If any bit of the "data_wstrb_reg" signal is enabled, it is a write-request, otherwise it is a read-request.

During a read-request, if a hit is produced, the respective word is already available in Data-Memory's output, so the request can be acknowledged.

The Data memory allows input Data from both the Front and the Back-End. This selection is done using the signal "replace", which indicates if the replacement on a cache line is in action. While "replace" is not asserted, all accesses are from the Front-End. During a read-miss, the signal "replace" is asserted, which will start the Back-End Read-Channel controller, responsible for line-replacement.

Both Tag and Valid memories are updated when the "replace_valid" (read-miss) signal is high, forcing a hit in the selected way. This allows the replacement process to act similarly to a regular write hit access, reducing the necessary logic. The replaced data, "read_data", is validated with "read_valid", and positioned in the cache line using "read_addr", which depends on the size of the line and the back-end's word-size. The replacement can only start if there are not currently write transfers to the main-memory.

The signals "write_valid" and "write_ready" constitute a handshaking pair for Cache-Memory to write to the Back-End Write-Channel. The former indicates the Write-Through Buffer is not empty, validating the transfer. The latter indicates that the Back-End Write-Channel is idle and thus enables reading the Write-Through Buffer.

The requirement that the replacement only starts after the write transfer is to avoid coherency issues, i.e. storing outdated data in the cache line.

Write requests do not depend on the data being available in the cache, since it follows the write-not-allocate policy. Instead, it depends on the available space in the Write-Through Buffer, which stores the address, write-data, and write-strobe array.

During a write-hit, to avoid stalling, the Data memory uses the registered input signals to store the data, so the cache can receive a new request.

If a read follows a write-access, Read-After-Write (RAW) hazards can become an issue. The requested word may not be available at the memory output, since it was written just the cycle before. This word will only be available in the following clock-cycle, therefore the cache needs to stall.

Stalling on every read-request that follows a write-hit access can become costly performance-wise. Hence, to avoid this cost a simple technique has been employed: the cache stalls only if one wants to read from the same way and (word) offset that has been written before. This results in RAW only signaling when the same Data memory's (byte-wide) RAMs are being accessed.

All the above conditions are implemented in the main controller circuit. The signals data_ready is given combinatorially by Equation 1.

$$\begin{aligned}\text{data\_ready} = (\text{write\_access AND !buffer\_full}) \text{ OR} \\ \text{OR}(\text{read\_access AND hit AND !RAW})\end{aligned} \tag{1}$$

A stall occurs if "data_ready" is 0 after the request, otherwise the request is acknowledged.

Because a new request can be issued in the same cycle as the previous is acknowledged, the registered request signals are used for the tag comparison, memory-writing and word-selection.

Since the line-replacement controller uses the way_hit signal, the hit signal is 1 only when signal replace is de-asserted, as this signal already has a delay to compensate for the Data memory's RAM 1 clock cycle read-latency.

D. Replacement Policy

The line replacement policy in a k-way set-associative cache is implemented by this module. Different available replacement policies can be selected using the "REP_POLICY" synthesis parameter. The module has three main components: the Policy Info Memory (PIM), the Policy Info Updater (PIU) datapath, and the Way Select Decoder (WSD).

The PIM stores information of the implemented policy. Note that replacement policies are dynamic and use data from the past, so memory is needed. The PIM has as many positions as the number of cache sets, addressed by the *index* part of the main memory address. The width of the PIM depends on the chosen policy. The PIM is implemented using a register-file so that during a system reset or cache invalidation, it can be set to default initial values.

When a cache hit is detected, the information stored in the PIM is updated based on the information previously stored for the respective set and the newly selected way. This function is performed by the PIU. When a cache miss is detected the information for the respective cache set is read from the PIM and analyzed by the WSD to choose the way where the cache line will be replaced.

The currently implemented policies are the Least-Recently-Used (LRU) and the Pseudo-Least-Recently-Used (tree and Most-Recently-Used (MRU)-based).

1) Least-Recently-Used: The Least-Recently-Used policy (LRU) needs to store, for each set, a word that has N_WAYS fields of log2(N_WAYS) bits each. Each field, named "mru[i]", represents how recently the way has been used by storing a number between 0 (least recently used) and N_WAYS-1 (most recently used), thus requiring $\log_2$(N_WAYS) bits. In total it requires N_WAYS$\log_2$(N_WAYS) bits per cache set.

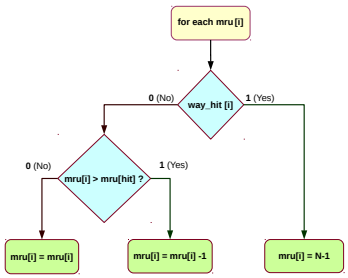

Fig. 5. LRU Encoder datapath flowchart.

The way each mru[i] is updated is represented in Figure 5. Summarizing, when a way is accessed either by being hit or replaced, it becomes the most recently used and is assigned. The other ways with higher mru values than the accessed way get decremented. The ones with lower mru values are unchanged. The selected way for replacement is the one with the lowest "mru" index. This can be achieved by NORing each index, as implemented in Equation 2.

$$\text{way\_select [i]} = \text{!OR(mru[i])}\,. \tag{2}$$

2) Pseudo-Least-Recently-Used: MRU-based: The PLRUm is simpler than the LRU replacement and needs to store, for each set, a word that has N_WAYS bits only. Each bit mru[i] represents how recently the way has been used, storing a 0 (least recently used) or 1 (most recently used), thus requiring $\log_2$(N_WAYS) bits.

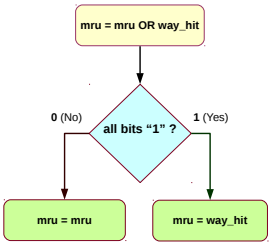

Fig. 6. PLRUm Updater datapath flowchart.

The way each mru[i] is updated is represented in Figure 6. Summarizing, when a way is accessed either by being hit or replaced, the respective bit is assigned 1 meaning it has been recently used. When all ways have been recently used, the most recently assigned remains asserted and the others are reset. This is done by simply ORing the way_hit signal and the stored bits, or storing the way_hit signal if all have been recently used. To select a way for replacement, the not recently used way (mru[i]=0) with the lowest index is selected. This can be implemented by the following logic equation, Equation 3.

$$\text{way\_select [i]} = \text{!mru[i] AND (AND(mru[i-1:0])} \tag{3}$$

3) Pseudo-Least-Recently-Used: binary tree-based: The PLRUt needs to store, for each set, a binary tree with $\log_2$(N_WAYS) levels and N_WAYS leaves, each representing a cache way. Each level divides the space to find the way

in two, creating a path from the root node to the chosen way, when traversed by the WSD. Each node is represented by a bit b[i] where 0 selects the lower half and 1 selects the upper half of the space. For a 8-way example, the binary tree is represented in Figure 7.

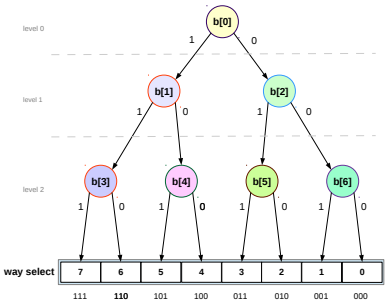

Fig. 7. PLRU binary tree.

To update each node b[i], the first step is to get the slice way_hit[i] from the vector way_hit, relevant for computing b[i]. Figure 8 shows how to compute way_hit[i] for the first 3 notes, b[2:0]. After computing slice way_hit[i], the algorithm shown in Figure 9 is followed. The process is straightforward. If the slice is not hit (all its bits are 0), then b[i] remains unchanged. Otherwise, b[i] is set to 0 if the hit happens in the upper part of the slice and to 1 if the hit happens in the lower part.

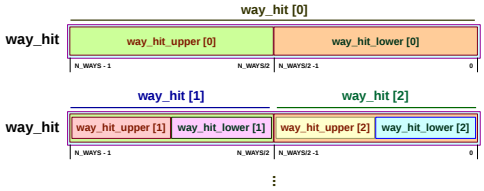

Fig. 8. Computing way_hit slices.

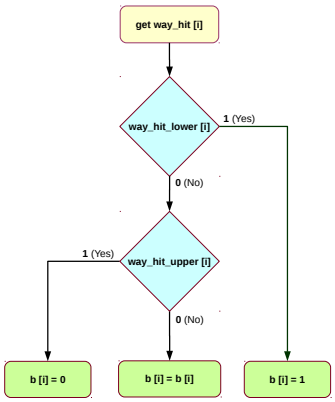

Fig. 9. PLRU way updater.

To select the way for doing the replacement, the binary tree needs to be decoded. This can be done by iterating from the tree levels, from root to leaves, using the b[i] values to point to the next node until the leaf is reached. As explained before the leaf index is the chosen way.

E. Back-End

The Back-End module is the interface between the cache and the main memory. There are currently 2 available main memory interfaces: Native and AXI. The native interface follows a pipelined valid-ready protocol. The AXI interface implements the AXI4 protocol [1], [10].

Although the AXI interface has independent write and read buses, the native interface only has a single bus available. In the native interface, the difference between a write and read access depends on the write-strobe signal (mem_wstrb) being active or not. This requires additional logic to select which controller accesses the main memory. There is no risk of conflict between the read and write channels: reading for line replacement can only occur after all pending writes are done.

The Back-End module has two controllers, the Write-Channel controller and the Read-Channel controller. The Write-Channel controller reads data from the Write-Through Buffer and writes data to the main memory while the buffer is not empty. The Read-Channel controller fetches lines from the main memory and writes them to the cache during a cache line replacement.

1) Write-Channel Controller: The native interface's controller follows the control flow displayed in Figure 10. The controller stays in the initial state while waiting for the write-through buffer to have data. The write-through buffer uses a FIFO, and the FIFO starts the controller when it is not empty. When that happens, signal write_valid asserts, and the FIFO gets read.

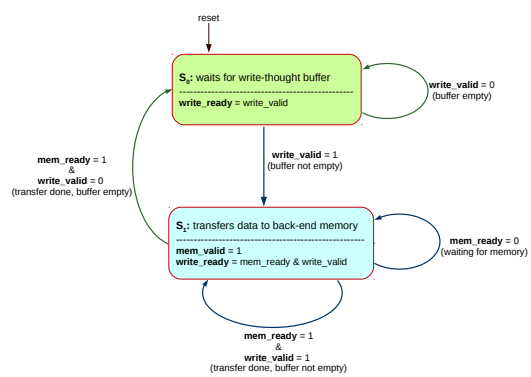

Fig. 10. Back-End Write-channel Native Control-flow.

In the following clock cycle, the required data is available in FIFO's output and the transfer can occur. After each transfer, the FIFO is checked, and if it is not empty, it is read again so the data can be transferred in the following clock cycle. This keeps happening until there are no more available data in the Write Through Buffer, and the controller goes back to its initial state.

The write-through buffer can only be read after each transfer is completed (mem_ready received). Currently, there is no way to pipeline these transfers, which are limited to 1 word per

every 2 clock cycles. While the controller is in the initial state, the memory's write-strobe signal is 0 to not disturb the Read-Channel controller.

The AXI-Interface (Figure 11) has similar behavior but follows the AXI4 protocol. The address valid-read handshake needs to happen before any data can be transferred. After the data is transferred, it is checked to see if it was successful through the response channel (B channel): if axi_bresp does not have the OKAY value (an AXI code), then the transfer was unsuccessful and the data is transferred again.

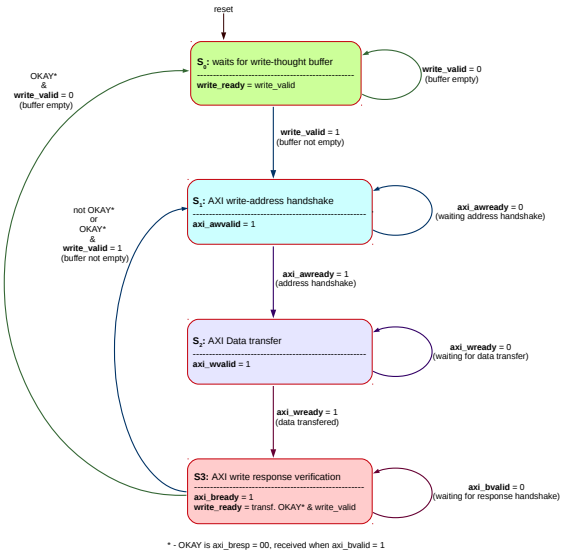

Fig. 11. Back-End Write-channel AXI Control-flow.

If the Back-End's data width (BE_DATA_W) is larger than the front-end's (FE_DATA_W), the data buses require alignment. The address signal becomes word-aligned, discarding the back-end's byte offset bits. These discarded bits are used to align both the write data and strobe (Figure 12).

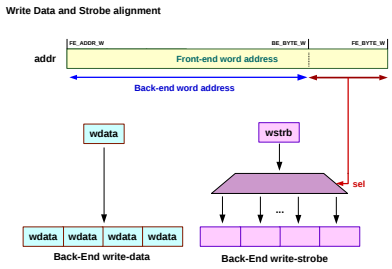

Fig. 12. Back-End Write-channel alignment.

This results in Narrow transfers [1, p. A3-49], allowing the smaller words to be transferred to a larger bus. The Write-Channel data width is, therefore, limited to the cache's front-end word-size. For example, in a 32-bit system, connected to a 256-bit wide memory, each transfer will be limited to 32-bit anyway.

2) Read-Channel Controller: The native interface's controller follows the control flow displayed in Figure 13. The controller stays in the initial state S_0 while waiting for the request of a line replacement. When signal "replace" is asserted, the controller goes to state S_1 requests a word block from the memory and writes it to the cache line at 1 word per cycle after it arrives at the back-end. It requests the base address of the main memory's memory block and uses a word counter to count the received words. After the last word is received the controller goes to state S_2 for a single cycle to compensate for Data-Memory RAM's read latency. Afterward, it goes back to its state S_0, de-asserting signal "replace".

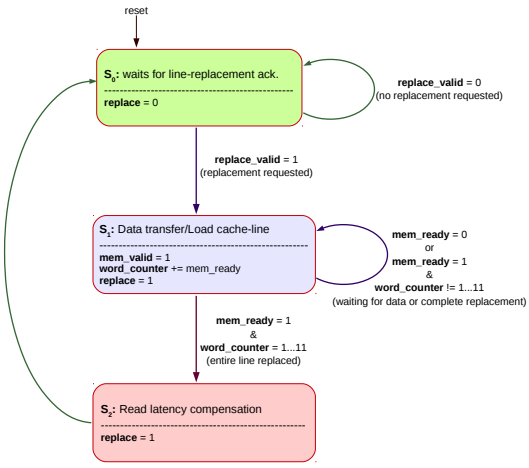

Fig. 13. Back-End Read-channel Native Control-flow.

If the back-end's data width (BE_DATA_W) is multiple the front-end's (FE_DATA_W), the number of words counted is proportionally shorter. If the back-end's data width is the same size as the entire cache line, the burst length is 1 and therefore the word counter is not used.

The AXI Interface controller (Figure 14) has a similar behavior but uses AXI4 burst transfers. The AXI burst parameters are derived for synthesis, using the front-end and back-end data widths, and the cache line's offset width. Instead of using a word counter, the signal axi_rlast is used to know when the line has been fully replaced. During the burst, each beat (transfer) increments signal read_addr automatically.

Unlike the Write-Channel controller, the response signal, "axi_rresp", is sent during each beat (transfer) of the burst. This requires the use of a register which sets in the case at least one of the beats was unsuccessful. After the transfers, the verification of this register can be done at the same time as the read latency compensation.

F. Cache-Control

The Cache-Control module can optionally be implemented using the synthesis parameter "CTRL_CACHE". It is used to measure the cache performance, analyze the state of its write-through buffer, or invalidate its contents. Additionally, the parameter "CTRL_CNT" implements counters for cache hits and misses, for both read and write accesses.

The Cache-Control functions are controlled by memory-mapped registers [11, p. 627], selected through "ctrl_addr". The addresses of the software accessible can be found in the cache's Verilog and C header files.

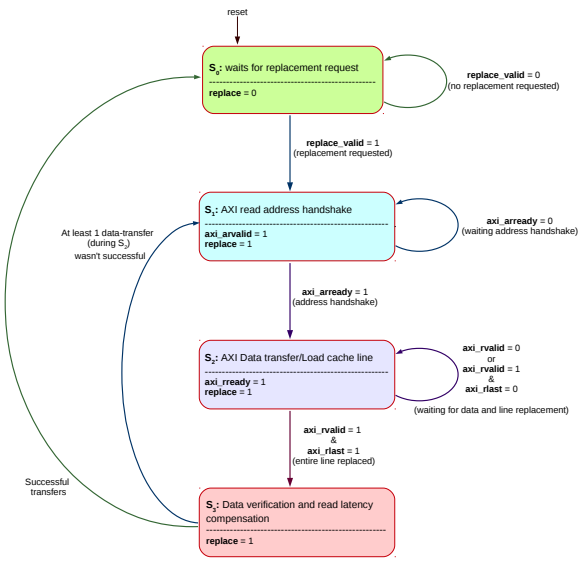

Fig. 14. Back-End Read-channel AXI Control-flow.

The ports write_hit, write_miss, read_hit, and read_miss work as enables that cause the respective counters to increment. The counters can be reset by hardware (global system reset) or by software.

III. Results

This chapter presents results on IOb-Cache's performance. A comparison between IOb-Cache and other open-source caches is also presented.

A. Performance

The Dhrystone [12] benchmark is a useful tool for measuring the performance of processors, using the Dhrystones/s score. The frequency-independent Dhrystone score is called Dhrystone Mega Instructions per Second (MIPS)/MHz or simply DMIPS. Here the Dhrystone benchmark is used to indirectly evaluate the cache as the performance of the system translates the performance of the cache if the processor remains the same. To do that, the Clocks-per-Instruction (CPI) measurement is taken while running the benchmark, as it provides a more direct indication of the cache performance compared to the DMIPS figure.

To efficiently test the cache, a pipelined processor is required, with a performance close to 1 CPI when using an ideal RAM. This way it is possible to analyze the average delay of the cache during memory accesses.

The Dhrystone benchmark has one shortcoming for testing the various policies, it is a small program that can be fitted entirely inside an instruction cache of common size. This shortcoming becomes an advantage for testing the pipeline operation, since it after full it behaves like a RAM, in a system connected to a larger SDRAM. Being a RAM, the correct pipeline operation happens with consecutive loads and stores which should take 1 cycle per instruction. A correct cache design allows for 1-cycle loads and stores whereas a poorer design will need 2 cycles for load/store instructions.

Cache system size	clock cycles	CPI
48 B (16+16+16)	513926	12.971
2 KB (0.5+0.5+1)	185163	4.673
8 KB (2+2+4)	51298	1.294
32 KB (8+8+16)	42397	1.070

TABLE I
FPGA emulation of Dhrystone SSRV (IOb-SoC) 32-bit at 50 MHz. 100 runs. 2-level cache system, sizes are for L1-Instr + L1-Data + L2-Unified.

The tests are run in IOb-SoC [13], using the SSRV [14], [15] multi-issue superscalar RISC-V processor. Despite being multi-issue, the processor was limited to 1 instruction per clock cycle in the tests, which is a simple setup but allows testing the cache. Connected to the IOb-SoC's internal memory (RAM only and no cache), it achieved CPI=1.02, running for 40445 clock cycles. The cache is implemented following a 2-level cache system: an L1-Instruction and L1-Data caches connected to a L2-Unified cache.

The FPGA system is implemented in the XCKU040-1FBVA676 Field-Programmable Gate Array (FPGA) [16], which is part of the Xilinx's Ultrascale FPGA family. The system is run at 50 MHz, 1/4 of the Memory Interface's frequency, so it requires the implementation of the synchronous AXI-Interconnect with a 1:4 clock ratio. Some results are presented in Table I.

During these tests, some results were observed, such as in a 2-way set-associative cache, PLRUt is the best choice since it requires less stored bits while offering the same performance. Using a set-associative in the L2-Unified cache represents the largest performance improvement. The PLRUm policy displays the highest performance in all three caches, while the LRU policy gives the worst performance. This occurs because of the cache's limited size and the fact the PLRU policies lack memory since there is no time locality to exploit. 32 KB cache is large enough to fit the dhrystone program.

B. Resources and Timing

Table II displays the synthesis and timing results of IOb-Cache using the Native interface for 2 different clock frequencies: 100 and 250 MHz. The results for IOb-Cache with AXI Back-End are similar and differ only in 15 LUTs and 2 Flip-Flop (FF)s.

The implementation differs for the 2 clock frequencies. The used memory is enough for BRAMs to be inferred for both the Tag and Data memories. For 100 MHz, the critical-path is from Tag memory output to a Data memory write enable signal. This path is caused by the signal way_hit, which results from the tag comparison and respective validation, and needs to be connected to write enable bits on a write-hit access. However, for 250 MHz the synthesis tool deliberately decides to implement the Tag-Memory with Look-Up-Table Random-Access-Memory (LUTRAM)s, with a stage register at the output, to be able to meet the timing constraint.

Data-Memory infers RAMB18 blocks for each byte in the cache line. Since the RAMB18 block is 18-bit wide, roughly half of it is wasted.

Ways	R.Policy	Lines	Words/line	LUT	LUTRAM	FF	RAMB36	RAMB18	WNS
100 MHz (10 ns)									
4 KB									
1		128	8	431	0	249	1	33	4.047
4	PLRUm	16	16	1727	1068	2407	1	0	3.212
8 KB									
2	PLRUt	128	8	1025	0	509	1	66	2.977
16 KB									
4	PLRUm	128	8	1940	0	1154	1	132	2.158
32 KB									
4	PLRUm	256	8	2961	0	2187	1	132	1.199
1		1024	8	1638	0	1145	1	33	4.003
250 MHz (4 ns)									
4 KB									
1		128	8	510	40	269	1	32	0.398
4	PLRUm	16	16	1730	1068	2407	1	0	0.024
8 KB									
2	PLRUt	128	8	1084	80	549	1	64	0.228
16 KB									
4	PLRUm	128	8	1974	160	1234	1	128	0.103
32 KB									
1		1024	8	1714	272	1162	1	32	0.523
4	PLRUm	256	8	2981	304	2289	1	128	-0.120

TABLE II
IOB-CACHE (NATIVE) RESOURCE AND TIMING ANALYSIS.

C. Open-Source Caches

In this chapter, the IOb-Cache is compared with the configurable PoC.cache design included in the PoC-Library [5] library of open-source cores. PoC.cache is the most competitive open-source cache one was able to find, so the other caches are not evaluated here as clearly they cannot compete with IOb-Cache or PoC.cache. The comparison between the two caches is available in Table III below.

In addition to the information in Table III, the following remarks are needed. The data-width of the back-end of PoC.cache is fixed to the cache line's size, and therefore not configurable to be smaller such as in IOb-Cache. The PoC.cache tag and valid memories are implemented with distributed LUTRAM and registers, respectively, to combinatorially check for a hit and achieve 1 read per clock cycle. Lastly, despite using the Write-Though policy, PoC.cache does not have a buffer and accesses the main memory for write transfers, which is comparatively slower.

Based on the information in Table III, the following conclusions can be drawn. There are 2 points where PoC.cache is better than IOb-cache: (1) the implementation of the cache invalidate function and (2) the implementation of a fully associative cache. PoC.cache can invalidate individual lines whereas IOb-Cache can only invalidate the entire cache. PoC.cache can be configured as fully associative (single set) cache and IOb-Cache needs at least 2 sets. However, besides its theoretical interest fully associative caches are seldom used in practice.

In the remaining features, IOb-Cache is better than PoC.cache: configurable back-end size with AXI4 interface as an option; write-through buffer and independent controller for fast, most of the time 1-cycle writing (PoC.cache only supports 1-cycle reads); more replacement policies to choose from; a modular design that allows changing both front and back-ends without affecting the cache's core functionality.

Both PoC.cache and IOb-cache have the same issue of implementing the Tag-Memory and Policy Info Module using registers, and thus consuming more silicon area than necessary. However, because IOb-Cache is designed to work with the 1-cycle read latency of RAM, it can easily be upgraded to replace these memories with RAMs while PoC.cache needs more drastic design changes.

IV. CONCLUSIONS

In this thesis IOb-Cache, a high-performance configurable open-source cache is developed. IOb-Cache is being used in dozens of projects. It is currently integrated into the IOb-SoC Github repository, which has 16 stars and is being used in 38 projects (forks). In the Github cloud community, it is currently the only Verilog cache found by its search tool, with this level of configurability, that supports pipelined CPU architectures, and the popular AXI4 bus interface.

The cache is composed of 3 modules: Front-End, Cache-Memory, and Back-End. The Front-End interfaces the processor with the cache. The Cache-Memory contains the memories and the cache's main controller. The main controller is implemented by a streamlined datapath that evaluates every necessary condition for read and write-accesses. The Back-End

	PoC.cache	IOb-Cache
HDL	VHDL	Verilog
Configurability		
n. ways, lines, words	Yes	Yes
back-end width	No	Yes
Mapping		
Direct	Yes	Yes
Set-Assoc.	Yes	Yes
Full-Assoc.	Yes	No
Policies		
Write	write-through	write-through
W.T. Buffer	No	Yes
Replacement	LRU	LRU, PLRUs
Back-End Connectivity		
Native	Yes	Yes
AXI	No	AXI4
Implementation		
Main-control	FSM	Data-path
Data-Memory	BRAM	BRAM
Tag-Memory	LUTRAM	BRAM
Valid-Memory	Register	Register
Rep-Pol. Mem	Register	Register
Invalidate	Yes	Yes
Performance		
clk/read (hit)	1	1
clk/write	2	1
Ready	during valid	after valid req.
Read-Data avail.	after ready	during ready
New req.	after ready	during ready

TABLE III
COMPARISON BETWEEN POC.CACHE AND IOB-CACHE.

implements Native and AXI interfaces, allowing flexibility in connecting to 3rd party memory controllers (likely using an AXI interface), and other cache levels (likely using the Native interface). The Native interface follows a pipelined valid-ready protocol, while the AXI4 interface is a full master bus implementation. Each interface has a specific controller for write and read-accesses. The Back-End's Write-Channel module is responsible for the write accesses: it reads data from the write-through buffer and writes them to the main memory. The Back-End's Read-Channel module fetches lines from the main memory and writes them to the cache during a cache line replacement. An optional module called Cache-Control can be selected. This module implements cache performance meters, and analyses write-through buffer states and cache invalidates. These functions are controlled by the CPU using memory-mapped registers. When the Cache-Control module is present, the Front-End module acts as a splitter between accesses to Cache-Memory or the Cache-Control modules, also through memory-mapping.

A. Achievements

In this work, several important achievements deserve to be highlighted. The the following list highlights them.

- The cache supports pipelined memory loads and stores, honoring 1 request per clock cycle. Given the 1-cycle latency present in RAMs, a request can be served while processing the next, which results in a throughput of 1 request per clock-cycle and latency of 2 clock cycles for hit addresses.
- The cache has a modular design that allows keeping its core functionality independent from its interfaces intact, implemented by the Front-End and Back-End modules.
- The cache can pass high frequency (250 MHz) timing requirements for a Xilinx Ultrascale FPGA, in a large number of configurations, including the 32 KB direct-mapped or the 8KB 4-way set-associative configurations.
- If large enough, the results show that its performance is close to that of having a fast on-chip RAM connected to the CPU. Using the multi-issue superscalar SSRV CPU, which has CPI=1.02 when connected to an SDRAM, the cache achieved CPI=1.07.

B. Future Work

IOb-Cache can still be further improved beyond the development time allocated to it. The main improvements are listed below in decreasing priority:

- Increase resource efficiency by reducing the amount of logic and memory used. Both the Valid-Memory and PIM modules would be more efficiently implemented with RAM; the Valid Memory could be merged with the Tag-Memory adding only 1 bit to its width.
- Implementation of the Write-Back Write-Allocate policy, to optimize bandwidth and the general performance for some applications.
- Improve the Cache-Control module, allowing cache line invalidates
- Cache Coherency, a dedicated module, and interface to implement a cache coherency algorithm would significantly expand the usability of IOb-Cache.

REFERENCES

[1] ARM. *AMBA AXI and ACE Protocol Specification*, 2020. Accessed on December 2020.
[2] IEEE 1364-2005 – IEEE Standard for Verilog Hardware Description Language, November 2005.
[3] airin711: Verilog caches, April 2016. Accessed on December 2020.
[4] prasadp4009: 2-way-Set-Associative-Cache-Controller, March 2016. Accessed on December 2020.
[5] Poc - Pile-of-Cores. Accessed on December 2020.
[6] IOb-Cache, December 2020. Accessed on December 2020.
[7] Recommended hdl coding styles, quartus ii 9.1 handbook, volume 1, November 2009. Accessed on October 2020.
[8] Intel Quartus Prime Standard Edition User Guide: Design Recommendations. Accessed on October 2020.
[9] IOb-memories, December 2020. Accessed on December 2020.
[10] Xilinx. *UG1037: Vivado Design Suite - AXI Reference Guide*, July 2017. Accessed on December 2020.
[11] Arlindo Oliveira Guilherme Arros, José Monteiro. *Arquitectura de Computadores – dos Sistemas Digitais aos Microprocessadores*. IST Press, 2nd edition, July 2009.
[12] Alan R. Weiss. Dhrystone benchmark: History, analysis, "scores" and recommendations. November 2002.
[13] IOb-SoC, December 2020. Accessed on December 2020.
[14] risclite. Superscalar-riscv-cpu, 2018. Accessed on October 2020.
[15] IOb-SSRV, September 2020. Accessed on October 2020.
[16] Avnet, Inc. *Kintex UltraScale KU040 Development Board*, 2015.