

Configurable RTL Model for Level-1 Caches

Vahid Saljooghi, Alen Bardizbanyan, Magnus Sjalander and Per Larsson-Edefors

VLSI Research Group, Dept. of Computer Science and Engineering
Chalmers University of Technology, Gothenburg, Sweden

Abstract—Level-1 (L1) cache memories are complex circuits that tightly integrate memory, logic, and state machines near the processor datapath. During the design of a processor-based system, many different cache configurations that vary in, for example, size, associativity, and replacement policies, need to be evaluated in order to maximize performance or power efficiency. Since the implementation of each cache memory is a time-consuming and error-prone process, a configurable and synthesizable model is very useful as it helps to generate a range of caches in a quick and reproducible manner. Comprising both a data and instruction cache, the RTL cache model that we present in this paper has a wide array of configurable parameters. Apart from different cache size parameters, the model also supports different replacement policies, associativities, and data write policies. The model is written in VHDL and fits different processors in ASICs and FPGAs. To show the usefulness of the model, we provide an example of cache configuration exploration.

I. INTRODUCTION

Since data and instructions in executed software exhibit spatial and temporal locality, it is likely that upcoming instructions will access memory addresses that are near the current one. This fact is the rationale for cache memory hierarchies which are extensively used in almost all systems. The part of the software that is likely to be needed by the processor can, thus, be transferred from the big and slow main memory to a smaller and faster memory—the cache—just before it is needed. Now, the processor performance is not constrained by the slow main memory, instead it depends on the speed of the cache. The level-1 (L1) cache of this work represents the fastest cache of the memory hierarchy, located closest to the processor datapath.

The L1 cache can be implemented in a large number of ways and the chosen cache configuration greatly impacts the whole processor system's behavior [1]. In addition, which cache configuration is optimal varies between instruction and data caches [2]. In this perspective, a generic RTL model that allows implementation parameters to be freely selected would facilitate design exploration: Once the cache model parameters have been set, the desired cache configuration can be used in logic simulation. When promising configurations have been identified, these can be synthesized and further evaluated on the metrics that are of importance. Design closure becomes straightforward as the exploration cycle readily can be repeated, using alternate cache parameters.

Our contribution is a versatile L1 cache RTL model that, in contrast to a specialized cache generator such as [3], is not tailored to a specific processor core (CPU). This VHDL model comprises logic and state machines to handle the complex schemes for writing and reading to the cache, providing a complete L1 cache. To support single-bus testing of a processor equipped with an L1 cache, arbitration between the instruction and data cache is also an option in the model. To our knowledge, generic RTL cache models are not publicly available at this time. The generality of the model comes with one disadvantage: Since the model is interface agnostic and since it does not support deeper pipelining, performance co-optimization of CPU and cache will have to be performed once the CPU has been chosen.

II. CACHE MODEL

In order to describe the model, its structure, and its parameters, we give a brief introduction to caches, their function and terminology.

A. Basic Terminology and Function

Fig. 1 shows one example of cache organization [4]. In most systems, and in our model, one cache is devoted to instructions, while another is devoted to data.

The cache is made up of entries that each contains data bits—the *cache line*—and tag bits, where the latter refer to the portion of the main memory address with which the cache line is associated. For the vast majority of caches the cache line contains more than one instruction or data to take advantage of spatial locality in an application. The instruction cache is addressed using the *program counter*, while the data cache uses the address from *load and store* instructions. For both the instruction and data cache, the address is divided into three fields as shown at the top of Fig. 1. The second field—*Index*—identifies the cache line, while the third—*Line Offset*—selects the appropriate word inside the cache line. The first field—*Tag*—contains the most significant bits of the main memory address that is requested.

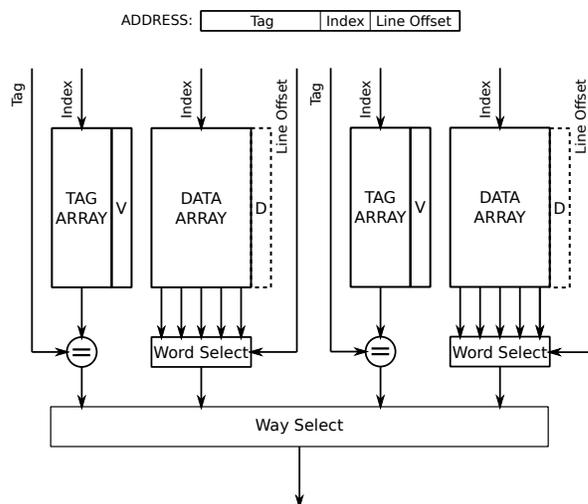


Fig. 1. Cache organization of a 2-way associative cache.

When the CPU needs to write to or read from main memory, it sends a request to the cache, which in turn checks if the requested main memory entry already exists within the cache. If it exists, we have a *hit*, and the CPU's read or write transaction will take place. Otherwise, we have a *miss*, leading to a time-consuming transfer of data from main memory to the cache after which the CPU can read from or write to the appropriate cache line. The CPU is stalled during the completion of the transfer to or from the next level of the memory hierarchy. One such occasion is at the start up of a program; here the CPU is being stalled while the first chunk of instructions and data is transferred to the caches.

Mapping of main memory addresses to cache entries is done by first dividing the address space into blocks that are of the size of a cache line. Then, 2^{Index} of these blocks are grouped together into a *page*. Thus, a page has the same size as the *data array* of a single way of the cache. A particular word within a page can be accessed by using the index to identify the cache line and the line offset to access the particular word within the line. The tag identifies which page a particular cache line resides in. When the CPU is accessing the cache, the address tag field is compared to the tag of the cache entry that is defined by the address index field. If the two tags match and the valid bit (the column marked V in Fig. 1) is set, we have a hit. Otherwise, a cache miss has occurred and data must be retrieved from the next level of the memory hierarchy.

In an *associative* cache, as the 2-way cache in Fig. 1, a cache line (with a particular index) can reside in exactly one position in each of the ways. With two ways the cache can store a cache line with the same index from two different pages. In the event a cache line with the same index would be accessed from a third page, one of the cache lines in one of the ways has to be overwritten. That cache line is said to be *evicted*. If the cache uses a write-back policy, the dirty bit (the column marked D in Fig. 1) of a corresponding line is set on a store operation. During eviction of a line, if the line's dirty bit is set, the line is written back to the next level in the memory hierarchy. The *replacement policy* decides in which order cache entries will be evicted upon a miss, while the associativity defines how main memory entries are associated with each index in the cache. With a higher associativity there is a greater chance that a cache line is not evicted, thus the *hit rate* increases. The downside is that it takes longer time to identify if the cache is holding the requested data. In terms of associativity, the two extremes are *fully associative* and *direct-mapped* caches. In the first cache, the main memory entries are freely associated with any cache entry, while in the latter, which uses a single way, each memory entry is associated with one specific cache entry. Direct-mapped caches tend to have relatively low hit rates, but in exchange their access speed is high.

B. Cache Model Parameters

Our VHDL RTL cache model [5] is developed to be configurable. Therefore, the caches can be used in processors with arbitrary data and address bus widths, and they can communicate with a variety of external memory widths. All configurable parameters, that is, *cache-line size*, *cache-line count*, *associativity*, *replacement policy*, and *write policy*, can be set through VHDL generics when instantiating the top-level cache entity. Currently two replacement policies are supported, that is, *least recently used (LRU)* and *pseudo random*. Since the RTL cache model is modular, it is easy to later add other replacement policies, such as pseudo LRU. As far as data write policies, our model supports write-back and write-through mechanisms, and, on misses, write allocate and no write allocate, in these combinations: *write back (with write allocate)*, *write through with write allocate*, and *write through with no write allocate*. The maximum associativity that can be used is four. If a higher associativity is needed, it is possible to increase this number by performing small modifications to the LRU calculation mechanism. In addition to the above parameters, the cache can be composed of memory modules with an equal number of entries and width, which is useful if the cache-line size is larger than the width of a single memory unit.

C. Cache Operation

In this part we describe instruction and data cache state transitions. The presented finite state machines (FSMs) are simplified versions of the implemented FSMs, only showing the most essential signals.

As shown in Fig. 2, the instruction cache FSM is a Mealy machine composed of three states. When the cache powers up, all valid bits are set to zero to avoid fake hits. The controller goes through all indexes (sets) of the cache and invalidates the tag by clearing its valid bit. During this initial **Flush** state the CPU is stalled. After clearing all valid bits the state is changed to **Comp_Tag**. In this state, the cache performs tag checks to identify if the instruction requested by the CPU resides in the cache or not. If the instruction is available in the cache, a hit occurs and the controller stays in the **Comp_Tag** state and delivers the requested instruction to the CPU. In case of a miss, the state changes to **Read_Mem**. As the name indicates, during this state the instruction cache loads the whole cache line containing the requested instruction from memory. At this point, in the case of an associative cache, the replacement policy dictates from which way a cache line will be evicted. Depending on the memory bus width and the cache-line size, it can take several cycles to load a cache line. The FSM returns to the **Comp_Tag** state once the complete cache line has been read from the next level of the memory hierarchy and asserts a hit for the recently loaded cache line.

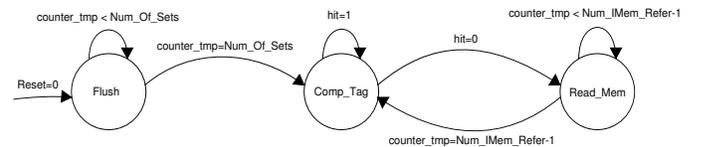


Fig. 2. Instruction cache state machine.

Next we describe the data cache controller that uses the write-back policy. As shown in Fig. 3, the flushing operation is similar to the one mentioned above for the instruction cache. After the **Flush** state, since it is not as frequently accessed as the instruction cache, the data cache goes to the **Idle** state and waits for a request from the CPU. As soon as there is a CPU request for reading ($dc_in.rd = 1$) or writing ($dc_in.wr = 1$), the data cache will change to the appropriate state. These signals should be driven by the result of the ALU if interfaced with a conventional five-stage pipeline. The reason for this is that the memories in the cache are synchronous and will be latched on the rising edge. If the signals are provided to the cache first, a complete cycle will be wasted in the memory stage waiting for the following rising edge of the clock. This also gives the FSM time to change from the **Idle** state to service the requested read or write. For both read and write operations, the data cache checks whether the requested data resides in the data array or not.

Assuming a miss occurs and the victim cache line is dirty ($dirty = 1$), then this line is written back to memory. This means changing to the **Wr_Back** state. The process of writing data back to memory, just like reading from it, takes several cycles as is suggested by the transitions that loop back to the **Wr_Back** state. After the data is written to memory, the controller moves to the **Read_Mem** state, in which the victim line is replaced by new data from the next level of the memory hierarchy. By now, if the CPU request was a read, the task is complete. If the next instruction also is a memory operation ($dc_in.rd = 1$ or $dc_in.wr = 1$), the next state becomes **Comp_Tag**, otherwise the controller changes back to the **Idle** state.

Another scenario is that, when the controller changes to the **Wr_Back** state, the CPU is requesting a write operation. In this case, after reading from memory, part of the cache line—word, half word or byte—is updated by the CPU. Here, **Update_DC** is the state that is reached to complete the CPU's write request. If the next instruction also is a data memory operation ($dc_in.rd = 1$ or $dc_in.wr = 1$), the controller changes to the **Comp_Tag** state, otherwise it changes to the **Idle** state and waits for another CPU request.

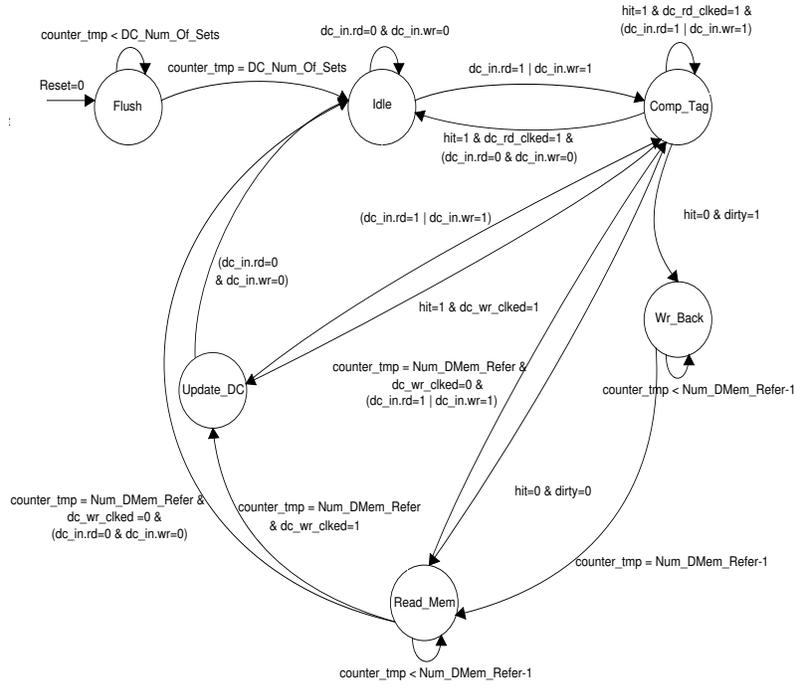


Fig. 3. State machine for data cache using the write-back policy.

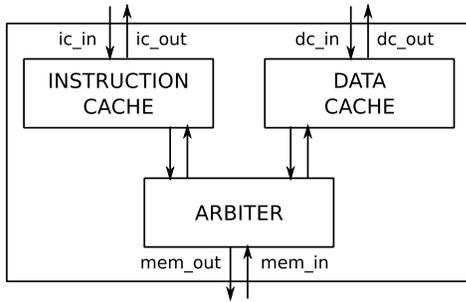


Fig. 4. Cache structure and its interface.

III. SYSTEM LEVEL MODEL

To make a generic RTL model useful to a third-party user, the model must have a structure that is easy to grasp. To make it easier to understand the signal interface between any two components, those interface signals can be encapsulated in VHDL records. Fig. 4 shows the signals encapsulations between the CPU, the caches, the arbiter, and the next level of the memory hierarchy. For example, the *dc_in* record, which was referred to in Sec. II-C, contains address, data, mask, read and write signals, as shown in Table I. The arbiter was developed for single-unit memory environments and performs the task of arbitration between caches to access external memory. If only one of the caches has a memory request, the arbiter grants access to memory immediately. In case of simultaneous requests from both caches, the data cache is given priority over the instruction cache.

To evaluate our RTL cache model, we integrated it with a five-stage MIPS core [4] that was developed in a separate project [6]. To create a complete system that can be deployed on an FPGA we also developed an AMBA™ High-performance Bus (AHB) [7] master interface that connects to the *mem_in/out* records of the arbiter. The AHB master interface connects to an AHB bus with a DDR memory controller that is constructed out of cores from the open-

TABLE I
RECORD DESCRIPTIONS

Record	Signal	Description
ic_in	addr	Address for the instruction cache
	stall	Stall the instruction cache
ic_out	data	The instruction returned from the cache
	stall	The instruction cache is stalled
dc_in	addr	Address to requested data
	data	Data to be written on write requests
	mask	Write mask
	rd	Read request
dc_out	wr	Write request
	data	The data returned from the cache
mem_in	stall	The data cache is stalled
	data	Data returned from main memory
mem_out	stall	The main memory is stalled
	addr	Address to requested data
	data	Data to be written on write requests
	mask	Write mask
	rd	Read request
	wr	Write request

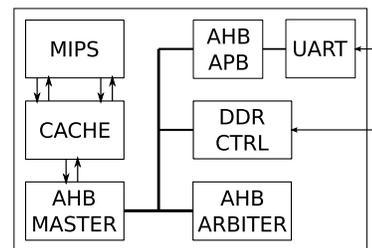


Fig. 5. System evaluation setup.

source GRLIB IP library [8]. The system also has a UART interface that connects to an Advanced Peripheral Bus (APB) bus, which is connected to the AHB bus through an AHB-to-APB bridge. Fig. 5 shows the complete system used in the evaluation presented next.

IV. DEMONSTRATION OF CACHE CONFIGURATION EVALUATION

Here we will present a basic execution-time evaluation using the RTL models of a processor system consisting of a five-stage in-order MIPS processor, L1 instruction and data caches, arbiter, and AHB interface. A time-accurate RTL model of a Micron DDR memory is also included to estimate the main memory latency. As metric we use the average execution time of the following five benchmarks from the EEMBC benchmark suite [9]: *autcor*, *conven*, *fft*, *rgbcmv*, and *viterb*. In terms of cache parameters, instruction and data cache sizes are varied between 512 bytes and 4,096 bytes. The cache-line size is fixed to 32 bytes for all the configurations.

Fig. 6 shows the average execution time for different cache sizes and associativities. The execution time is normalized to the slowest configuration, that is, the 512-byte direct-mapped cache. As expected, when we increase the cache size, the execution time is decreased since the miss rate is reduced. Increasing the cache size further yields diminishing returns, since the miss rate does not decrease significantly after some point. The 2-way associative cache performs more efficiently than the direct-mapped cache. Again, this is expected, since the miss rate decreases due to fewer conflicts. Due to the variation in application working sets, some benchmarks are more sensitive to the cache size than others. For example, the *rgbcmv* benchmark experiences only a 14% execution time improvement when the direct-mapped cache is scaled up from 512 bytes to 4,096 bytes, whereas the *fft* benchmark shows a 73% execution time improvement for the same scaling.

Fig. 7 shows the average execution time for different data write policies. Only a direct-mapped configuration is shown here, but the

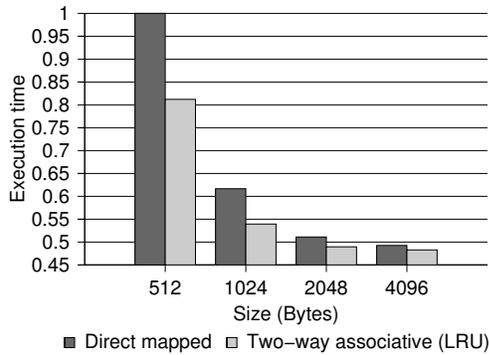


Fig. 6. Execution time for for different cache sizes and associativities, normalized to a 512-byte direct-mapped cache configuration.

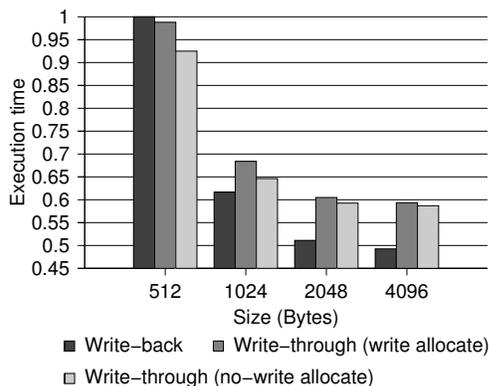


Fig. 7. Execution time for different write policies and cache sizes, normalized to a 512-byte write-back cache configuration.

trend is similar for higher associativities. Because each store operation causes stall cycles until the data are written to main memory, the write-back policy tends to be faster than the write-through policies. The write-through policy actually gives some execution time benefits for very small cache sizes, since the write-back rate is higher here. Although the execution time penalty of the write-through policy can be somewhat mitigated using a write buffer between main memory and the L1 cache, the write-back policy offers the best performance in most cases, which is in line with other reports [10]. The write-through with no-write-allocate policy has shorter execution time than the write-through with write-allocate policy. This is because the no-write-allocate policy reduces the conflicts in the cache, since the store operations that miss in the cache do not cause any allocation in the cache. As the cache size increases, the miss rate decreases and the reduction in the conflicts decreases. Hence, the advantage of the write-through with no-write-allocate policy diminishes.

There are other parameters that should be considered when deciding the cache configuration in a real system. For example, a decreased miss rate does not only improve the execution time, but it also—due to fewer accesses—reduces main memory energy. Another example is an FPGA platform, in which the SRAM blocks are usually of fixed size. To avoid unused space in the SRAM blocks once the resource allocation has been completed, the cache configuration process should consider how memory blocks are implemented.

V. CONCLUSION

We have described an RTL model of a complete L1 cache that enables rapid processor system implementation and evaluation. The RTL model is configurable in a number of parameters (line size, line count, and associativity) and policies (replacement policies and data cache write policies). To show the usefulness of the model, we demonstrated an evaluation of a processor system, which consists of a MIPS processor, the presented cache RTL model, an AHB bus and an AHB arbiter. By varying key parameters, we were able to quickly run a benchmark suite on the whole processor system model, evaluate the results, and identify efficient cache configurations. Synthesis and place and route can readily be performed after the architectural evaluation, improving the evaluation accuracy further.

REFERENCES

- [1] R. Bahar, G. Albera, and S. Manne, “Power and performance tradeoffs using various caching strategies,” in *Proc. Int. Symp. Low Power Electronics and Design (ISLPED)*, 1998, pp. 64–69.
- [2] A. Milenkovic, M. Milenkovic, and N. Barnes, “A performance evaluation of memory hierarchy in embedded systems,” in *Proc. 35th Southeastern Symp. System Theory*, 2003, pp. 427–431.
- [3] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, “Performance and power of cache-based reconfigurable computing,” in *Proc. Int. Symp. on Computer Architecture (ISCA)*, 2009, pp. 395–405.
- [4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. Morgan Kaufmann Publishers, 1998.
- [5] L1 Cache VHDL Code. [Online]. Available: <http://www.flexsoc.org>
- [6] M. Thuresson, M. Sjölander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenström, “FlexCore: Utilizing exposed datapath control for efficient computing,” *J. Signal Processing Systems*, vol. 57, no. 1, pp. 5–19, Oct. 2009.
- [7] *AMBA™ Specification*, ARM, May 1999, rev. 2.0.
- [8] *GRLIB IP Library User’s Manual*, Aeroflex Gaisler, Jan. 2012, ver. 1.1.0, B4113. [Online]. Available: <http://www.gaisler.com/products/grlib/grlib.pdf>
- [9] Embedded Microprocessor Benchmark Consortium. [Online]. Available: <http://www.eembc.org>
- [10] *Cortex-A15 MPCore Technical Reference Manual*, ARM, 2012, sec. 6.4.1, rev. r3p2. [Online]. Available: <http://infocenter.arm.com>