NTNU – Trondheim
Norwegian University of
Science and Technology

Evaluation of cache architectures for a low-power microcontroller system

Vinicius Almeida Carlos

Embedded Computing Systems
Submission date: June 2013
Supervisor: Bjørn B. Larsen, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Abstract

Reducing energy consumption has become a mantra in the Embedded Systems industry. The energy consumption of a memory system of a microcontroller-based embedded system accounts for a great part of the total energy consumption. Furthermore, Flash memories, as the ones usually employed in this kind of system are very power hungry elements. With that premises in mind, this project tries to tackle this problem by analysing the impact of adding a cache system to such existent microcontroller system. It performs this task by going in two directions at the same time: on one of them it evaluates a set of new cache architectures using a high level simulation environment, while on the other it implements a cache system in the register-transfer level with the goal of performing accurate power measurements. This text is the report of how these tasks were accomplished, which results were obtained and what criticisms and conclusions can be derived from it.

Contents

List of Abbreviations

SRAM Static Random-Access Memory

SPM Scratchpad Memory

LSBs Least Significant Bits

DM Direct-mapped

DVS Dynamic Voltage Scaling

DLC Dynamic Loop Cache

PC Program Counter

LCC Loop Cache Controller

PLC Preloaded Loop Cache

HLC Hybrid Loop Cache

ALC Adaptive Loop Cache

BB Basic Block

CAM Content-Addressable Memory

BTB Branch Target Buffer

HBTC History-Based Tag-Comparison

AHB Advanced High-performance Bus

NVM Non-volatile Memory

DUT Device Under Test

HC Hand-coded

RF Register File

RTL Register-transfer Level

Chapter 1

Introduction

This master thesis is a straight continuation of last semester project [2]. And as such, it delves deeper into the matter of whether adding a cache system to a microcontroller-based system can improve its energy efficiency. It does that by attacking the problem in two different fronts: on one hand it evaluates new cache architectures by extending the high level simulation environment developed in [2], while on the other hand it takes the solution proposed in [2] (direct-mapped cache with indexing) and goes all the way down to the hardware implementation with the main purpose of validating the findings obtained in [2].

Moreover, as a development over [2], it includes, for the sake of completeness and continuity, entire sections from last semester project report. However, whenever that happens an acknowledgment is placed on the title of the section, as can be seen in the introduction presented below.

1.1 Problem description (extracted from[2])

It is a well-known fact that energy consumption is of great concern on battery powered embedded systems. Even though improvements on battery capacity have been made in the past years, they are unable to keep up with the increase of energy consumption of such systems. In general terms, many techniques can be devised to tackle this problem, as described for instance in [22]. Nevertheless, the focus here is on microcontroller based systems, in which memory systems are responsible for a great part of the system total energy consumption. In particular, Flash memories, as the one used in the target system of this project, are power hungry elements.

The baseline architecture of the target system of this project, displaying the connection between the microprocessor and the Flash memory, can be seen on figure 1.1.

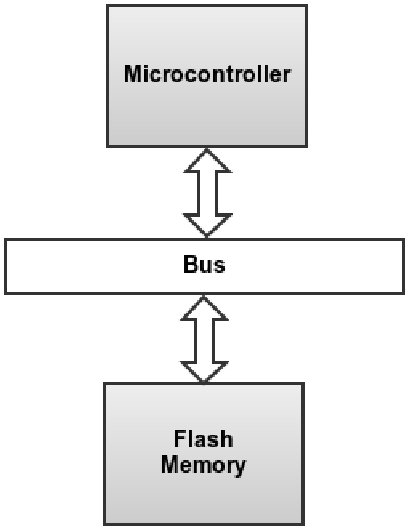

Figure 1.1: Baseline Architecture.

The aim of this project is to extend this architecture by adding a small Static Random-Access Memory (SRAM) to the system, in order to avoid, as much as possible, accesses to the main memory (the Flash memory). The extended version is depicted on figure 1.2.

As can be seen in the figure 1.2, the SRAM added to the system will be used to store instructions only. This small SRAM can then either be an instruction cache (I-cache) or a Scratchpad Memory (SPM). Either way, the purpose remains the same. More importantly, they still represent a great part of the total power consumption of the system, therefore careful design of these elements is paramount to reduce overall energy consumption.

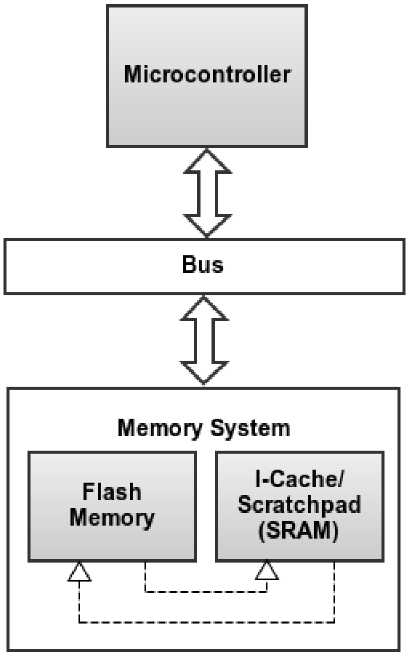

Figure 1.2: Extended Architecture.

1.2 Tasks

In [2] two cache architectures were evaluated in a high level simulation environment with very interesting results, however, as presented in the future work section in [2], there is room for further exploration and questions to be answered. The tasks presented below aim at answering those questions.

Task 1: Evaluate in a high level simulation environment a new cache architecture.

This task comprises the implementation and simulation of the cache architecture named Way-halting in the Python simulation environment developed during the last semester project. For reasons that will be clear later, the Way-halting evaluated here is a 4-way set associative cache, with the four Least Significant Bits (LSBs) as the halting bits.

Task 2: Evaluate the Direct-mapped (DM) with indexing cache architecture against a set of applications.

In [2], due to certain limitations, the indexing solution was evaluated only against one application, however to properly assess the contribution of using indexing in a direct-mapped cache it is important to evaluate it with more applications. Therefore, in this master thesis, two other applications are to be employed for such task.

Task 3: Implement the direct-mapped cache with indexing in SystemVerilog.

A conclusion drawn from [2] was that the direct-mapped cache with indexing is a very good solution in terms of area, latency and energy consumption. However, the high level simulations can give only an estimate of what would be the energy consumption of such architecture, therefore it is highly interesting to evaluate how this cache architecture can perform in real hardware. For example, the high level simulations do not take into account the energy consumption of the cache controller itself, which includes, among other things, the energy dissipated in the hardware responsible for performing the indexing function. For that reason this cache controller is implemented in SystemVerilog and power simulations are used to get the numbers that later are compared to the ones obtained with the high level simulations.

The implementation of this cache architecture in SystemVerilog, which takes most of the time dedicated to this master thesis, serves also as a training exercise in both a new hardware description language and a design methodology employed by Nordic Semiconductor, the company that supports this project.

Task 4: Evaluate the direct-mapped cache with different types of SRAMs.

In [2] only one type of memory was used while performing the high level simulations. In this master thesis the simulations are performed with two types of memories. Moreover, while in [2], due to the memory generator tool (Artisan) constraint, not all possible memory sizes were evaluated, in this master thesis an attempt to find the global minimum in terms of energy consumption is performed.

Task 5: Implement and evaluate the direct-mapped cache as a phased cache.

As will be clear later on this text, phased cache is a different way to implement the tag and data lookup on a cache line. Both the high level and the SystemVerilog versions of the direct-mapped cache are to be implemented and evaluated as phased caches.

Task 6: Implement and evaluate the 2-way and 4-way as a phased cache.

On [5] it was concluded that a DM cache offers better results in terms of energy efficiency compared to their n-way set associative counterparts. However, this project aims at evaluating how they perform when implemented as phased caches. It is mainly interested in checking whether the fact that not all ways are accessed in every memory read helps at all. The 2-way and 4-way are implemented with a Random Replacement policy.

As can be seen by the tasks devised for this project, there are different parallel paths to be followed in the course of this master thesis and this fact will become apparent over the entire text, in which the sections will most of the times be split between the high level and the SystemVerilog implementations. Thus the remainder of this text is organized as follows: Chapter 2 introduces the cache architectures researched last semester with an added section explaining the phased cache concept. Chapter 3 explains the simulation environment developed in Python and the design steps and final result of the SystemVerilog implementation. It also describes the methodology employed to obtain the values for energy consumption in the SystemVerilog implementation. And finally list all the experiments that take place in the scope of this project. Chapter 4 presents the results related to the aforementioned tasks, while Chapter 5 discusses the most relevant points of the project and Chapter 6 draws conclusions and proposes future work.

Chapter 2

Cache Architectures for Low Power (extracted from[2])

(This text assumes the reader is familiar with basics on cache. For an excellent introduction, please read Chapter 7 of [20]).

Caches are usually employed to hide the ever existing latency between the processor and the main memory. However, due to cost constraints, there is a limit to the size of these memories.

Caches are on the critical path of systems, hence contributing to much of a microprocessor system's power and energy consumption. For example, in [31], it is reported that caches may consume approximately 50% of a microprocessor's power. And, as shown in [13], instruction caches consume more energy than data caches, therefore this literature research focused on solutions that were more relevant to instruction caches.

Although the main interest of this project is on architectures for caches that can reduce energy consumption, circuit level techniques were also taken into account, as a possibility for a future work, and thus are also briefly presented in the next section.

2.1 Circuit Techniques

2.1.1 Way-Decay Cache

The idea, as presented in [15], is that after some period of time the cache will start to suffer more misses (spacial and temporal locality decrease overtime), that is when there is opportunity to shut down some of the cache ways to reduce energy consumption. Thus, additional structures (basically counters) are added to the cache architecture to monitor the hits and misses and make a decision on when it is time to shut down some ways or turn them on again. The shutting down mechanism is performed with a circuit technique called gated-GND. An extra transistor is added to the ground path or supply voltage of the SRAM cell, as shown in 2.1, taken from [15]. The rationale is that, by turning off parts of the circuit, static power can be saved (no leakage). For a 8-way-64K bytes instruction cache, under the SPECint95 benchmark, an average of 7.39% of energy saving is reported.

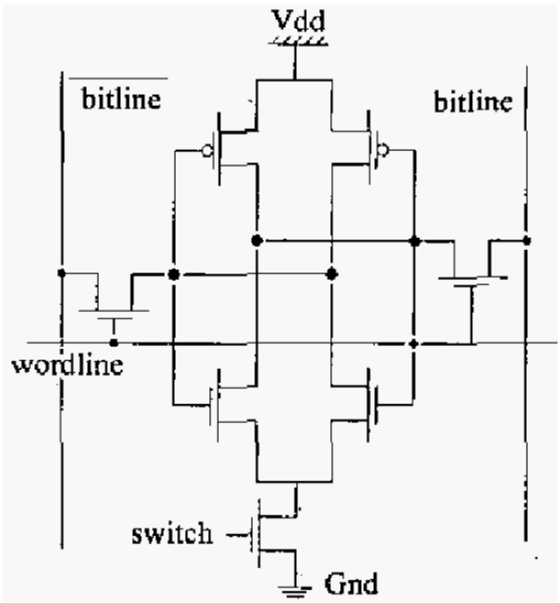

Figure 2.1: Gate-GND Technique

2.1.2 Drowsy Caches

This technique also focus on reducing current leakage, hence reducing static power consumption. As described in [12][4][11], the idea is to implement Dynamic Voltage Scaling (DVS) by changing the basic SRAM cell by introducing 2 transistors to control the voltage (as can be seen in 2.2, taken from [12]), enabling the memory to go to a state-preserving, low power drowsy mode. However, this solution requires that a prediction policy is implemented in order to determine when to put the cache lines in drowsy mode. Furthermore, there is a penalty of waking-up those lines when needed, increasing (slightly) the time to access the data. For a 2-way-32K bytes instruction cache, under SPEC2000, an average of 25% leakage reduction is reported.

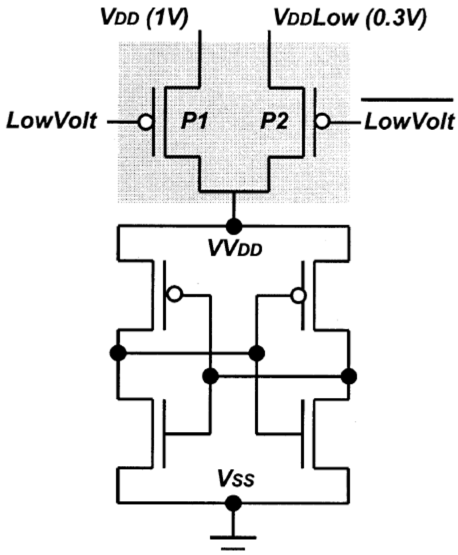

Figure 2.2: Drowsy Memory Cell

2.2 Architectures

2.2.1 Tiny Caches / Loop Caches

The starting point of this approach is the fact that the power per access of a memory begins to increase steeply at a size around 128 or 256 instructions ([7]). Therefore it is interesting to keep the size of the memory to a minimum.

A loop cache is a tagless cache, which stores the instructions of different loops and uses a controller to determine when the instructions should be fetched from the loop cache or from the next level on the memory hierarchy. The four variations available in the literature are presented in the following sections.

2.2.1.1 DLC

In this type of loop cache, the loop detection is done dynamically based on the behavior of the Program Counter (PC). In the original proposal, as described in [7], the Loop Cache Controller (LCC) detects that a loop is happening by observing the PC. When a branch with a negative offset happens, the LCC starts caching the instructions being fetched after this branch instruction. When the same branch instruction is executed again, the LCC knows that a loop is being executed, therefore the instructions are then fetched from the loop cache, hence not from the main memory. The drawback of this implementation is that it only works for loops with straight-line execution, rendering this solution quite limited.

2.2.1.2 PLC

As the name suggests, the idea (proposed in [7]) is to preload into the cache the loops which represent the best candidates in terms of energy saving. This is performed based on profiling of the application. This solution achieves better energy savings than the DLC, however the obvious downside is the need for profiling, which means that it requires support of a set of external tools. Moreover, it is completely application-dependent.

2.2.1.3 HLC

Also proposed in [7], it basically combines DLC and PLC into one cache. It offers even better results than the PLC.

Varying the loop cache sizes from 32 to 512 instructions, around 60% to 70% of reduction of fetches from L1 cache is reported, under the Mibench and Powerstone benchmarks.

2.2.1.4 ALC

Proposed in [23], this solution tries to mimic the results of HLC, but without the need of profiling the application beforehand. It accomplishes that by implementing an enhanced version of the LCC, which takes care of determining during runtime the loops that should be cached. It produces better results than HLC when the cache size is smaller than 128 instructions, otherwise an increase of 5% on average is reported.

2.2.2 Scratchpad

Scratchpad memory (SPM) is a small, fast SRAM memory that can be used to store parts of code or data of an application. It takes up part of the full memory address space, providing fast and low power access to its contents. It is an interesting solution in comparison to caches because it does not contain any of the extra structures a cache needs (tags, muxes and comparators). The challenge then becomes determining which sections of code (or data) should be mapped into the scratchpad, in order to maximize energy savings.

The two basic variants of this approach are regarding whether to perform this allocation dynamically or statically. The former offers more flexibility and explore more opportunities during the lifetime of the application, however, as pointed out in [1], the overhead of moving chunks of memory between the main memory and the SPM might lead to performance degradation in terms of speed and energy in comparison to a static solution.

2.2.2.1 Dynamically allocated

In this type of solution, the set of instructions/data loaded into the SPM changes over time. This change happens with help from the compiler, which places special instructions telling when and what to copy from main memory (or L1 cache, in other words, the next level of memory in the hierarchy).

There are then algorithms focusing on making this dynamic placement as efficient as possible. In [9], basic blocks (BB) are the smallest unit of code that can be transferred to the SPM. Then, based on temporal relationship between the BBs (a metric called concomitance), instructions to move those BBs to SPM are inserted in the code. In [18], “the length of transfer blocks can be adjusted in increments of one instruction”. According to [18], their algorithm “achieves 31% instruction delivery energy reduction over even an ideal coarse-grain algorithm”.

2.2.2.2 Statically allocated:

There are two ways to perform static placement of code/data into SPM: compiler-aware allocation or post-compiling allocation. They both rely on the same principle, which is identifying the most suitable pieces of code/data to be moved permanently to the SPM. The main difference between these two methods is regarding the input for this identification process. For a compiler-aware allocation, the input is the source code, while for the post-compiler, it is a binary file. While the former is more flexible, the latter is more general, since it can be applied to any scenario, without knowledge or customization of the compiling tools.

Post-compiler allocation: In this approach, the following actions are necessary:

1) Identify, in the executable file, all units of code (or data) that are candidates to be placed in the SPM.
2) Given the set of candidates, solve a basic Knapsack ([16]) optimization problem, that is, find the best suitable set of blocks of code that can be placed in the SPM, in order to minimize energy consumption.

3) Patch the binary file, based on the solution found in item 2.

Both [1] and [28] developed their solutions based on these steps. The most important difference between them is the granularity considered when selecting a candidate to be placed in the SPM. In [1], they work with a fine-grain block boundary, with minimum size equal to one single instruction, whereas in [28] only BB and procedures are considered as possible candidates. Unfortunately, these two works cannot be directly compared because they use different benchmarks, however both report a significant energy reduction in comparison to a system without SPM.

2.2.3 Associative Memory / Content-Addressable Memory

An associative memory (or Content-Addressable Memory (CAM)) is a type of memory in which one can search an input data against all values present at the memory in parallel, that is, it compares the input data with all stored values at the same time. That is usually used for devices that need very fast search times. On the other hand, it is a very power hungry type of memory. However, if kept at smalls sizes (namely 32 to 64 entries), it can be used for low power devices as well. The general idea of using CAM for caches is to store the tags in a CAM and the cache blocks in a normal SRAM. That way the tag search can be sped up, giving a hit rate of a 32-way associative memory, while keeping the power consumption to lower levels.

In figure 2.3, taken from [32], a basic structure of a CAM used to store the tag part of a cache entry is shown. As can be seen in the figure, the tag is fed into the CAM and all lines from the CAM will have the result of the search at the same time. If there is a match in any of the entries, its output will be driven to '1' (high/VCC) and that can be used to drive the cache line that contains the desired data.

Although using CAM seems like an interesting solution, besides the fact that size must be kept to a limit, it is also important to mention that CAM memory cells are different from SRAM memory cells at the transistor level, therefore needing special design, making it not possible to use standard memory libraries during the design of the circuit.

In order to cope with the CAM size as the limiting factor, the author on [30] proposes to split the tag memory into two different memories: a CAM and a normal SRAM fully associative memory. A different direction was proposed on [3], in which the idea is to change the CAM cell circuitry to perform serial bit comparisons in order to save energy. It performs the serial bit comparison only for the 4 least signficant bits (LSBs), since most of the hits can be determined by these bits. Then, if it matches, the rest of the comparison is performed in parallel. Problem with this approach: 25% slower than normal CAM.

In [32], the author employs sub-banking, that is, split the memory in smaller banks,

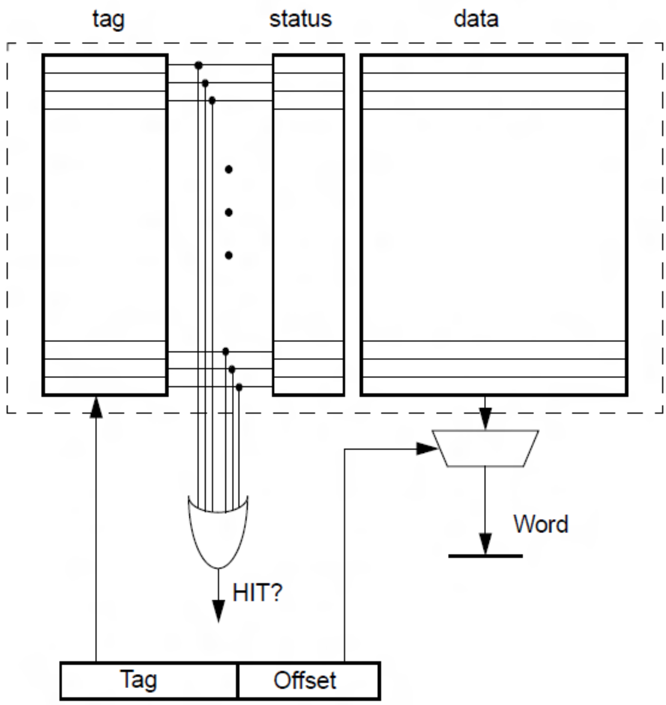

Figure 2.3: CAM Tag.

in which only one bank is active at a time, based on the address being requested. Each sub-bank contains a CAM memory to store the tags and drive the cache lines, when a hit occurs.

2.2.4 Way-Prediction

In set-associative caches, both the tag arrays and the data arrays are accessed in parallel (as can be seen in figure 2.4, taken from [20]). Thus, on a n-way set associative cache, n ways are accessed in parallel and then, based on the tag comparison, the data from the correct way is selected to be the output. However, in every hit, only one way holds the correct data, therefore the other $n-1$ ways accesses are useless, resulting in wasted energy. The idea is then to apply mechanisms that allow to predict the way that should be accessed next, saving the energy of reading from the other ways.

This kind of solution relies on adding extra structures to help with the prediction, such as extra bits to the Branch Target Buffer (BTB). The big disadvantage of this type of solution is the extra latency added to the access time when a miss

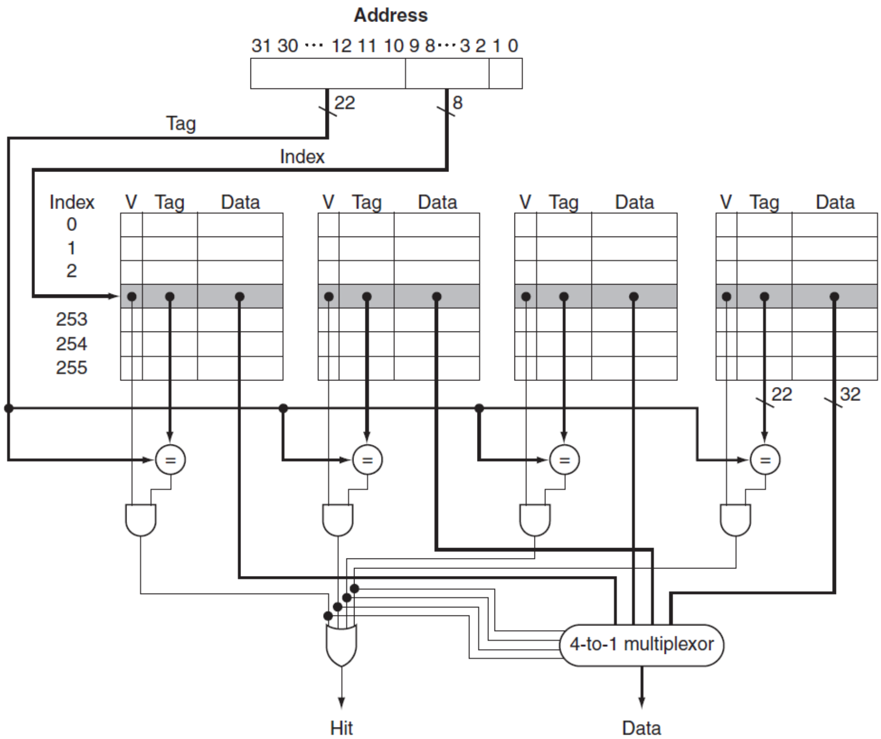

Figure 2.4: Example of 4-way Associative Cache.

prediction happens.

An alternative to way-prediction is to perform serial access to tags and ways, guaranteeing that only the correct way is accessed at all times. The obvious problem is the increase of as much as 60% on access time, as depicted in [21].

2.2.4.1 Way-Halting

A similar idea, but with a slightly different take, is way-halting. Given the same assumption as above, the intention now is, instead of predicting the way, to avoid accessing the wrong ways. This method was proposed on [31].

In order to do that, a small fully associative memory is used to determine the ways that should be accessed. It works in the following way:

The tag bits are split into 2, where the four LSBs are used as address to the fully associative memory. The rationale behind using the four LSBs is that with those bits it is possible to determine, most of the times, whether a hit happens or not. Then the fully associative memory is accessed in parallel with the index decoder.

Thus, only those ways where there was a hit on the fully associative memory will actually be accessed, saving the dynamic power that would be used to read from the other ways.

The interesting advantage over the way-prediction solution is the fact that there is no extra latency added to the cache access, since the fully associative tag comparison happens in parallel with index decoding, as depicted on figure 2.5, taken from [31].

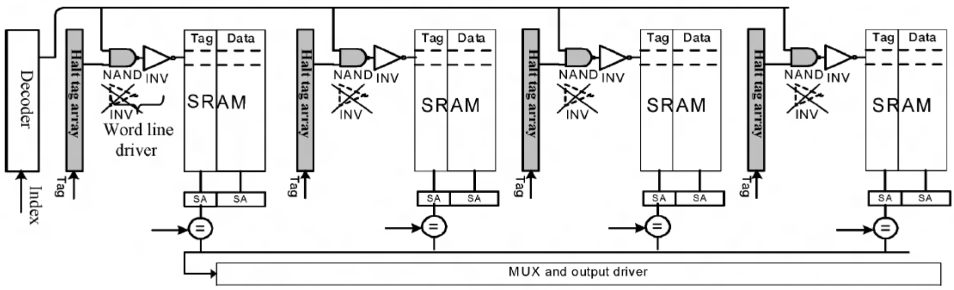

Figure 2.5: Baseline Architecture for Way-Halting Cache

Instead of using CAM, the author implements the fully associative memory using normal SRAM memory cells.

For a 4-way-8K bytes cache, under the Mediabench, Powerstone and Spec2k benchmarks, reductions on power consumption ranging from 45% to 60% are reported.

2.2.5 Indexing / Hashing

As described in [29], a direct-mapped cache has many advantages in comparison to n-way associative caches. Some of them are listed below:

- Less power consumption per access.
- Less area (only one array of data and tags, no multiplexers).
- Faster access times.
- Easier to implement.

However, the biggest problem with DM caches is the higher miss rate, when compared to n-way associative caches. This elevated number of misses comes from the fact that many addresses end up being mapped to the same location in the cache, resulting in what is called conflict misses. These facts are behind the motivation for the solution presented in this section.

It is important then to understand that the mapping from the main memory address to the cache address is basically a hashing function. The standard way to realize this translation between addresses is to take n LSBs from the full address and use it as the address of the cache line, as can be seen on figure 2.6, taken from [20].

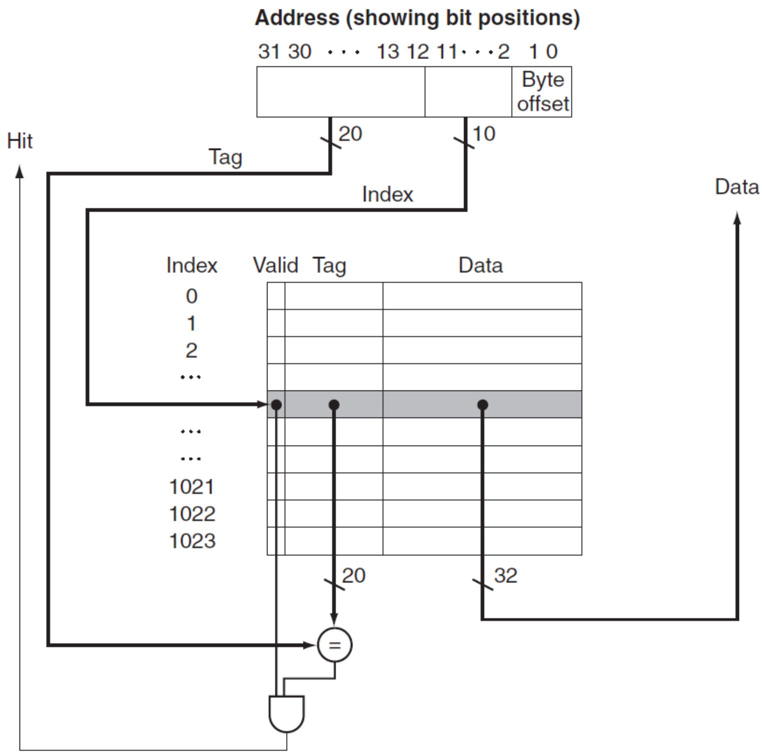

Figure 2.6: Basic Direct-Mapped Cache.

Although very easy to implement, this form of mapping is far from ideal, rendering many conflicts in the cache. The intention of indexing/hashing is then to change the access pattern of the memory address by using some hashing functions with the set of indexes bits. This is the general idea of all hashing/indexing approaches. What may varies is how the implementation is done and rather or not the hashing function is dynamically reconfigurable.

On [29] for example, reconfigurable decoders are used. Through prior profiling of the application, the configuration of the decoders are determined. It is reported an improvement in the number of conflicts, achieving similar numbers of a 2-way cache. On [6], a zero-overhead scheme is presented, where the hashing function is nothing more than a bit-selection function, that is, based also on application profiling, a heuristic algorithm is used to determine the set of address bits that should be used to address the cache. In this specific work, it is reported more improvements for data than to instruction caches. The bit-selection function is called a zero-overhead solution because no extra hardware is needed to perform the hashing function, differently from the XOR-based functions, in which the hashing function relies on XOR gates placed between the main memory address and the cache address. A similar solution to [6], but with an optimal algorithm, is described on [19]. On [27], the algorithm presented on [19] is adapted to XOR-based functions.

2.2.6 Tag Omission

Because caches are much smaller memories than the main memory, it cannot hold all data (or instructions) at all time. That is why every cache line is also composed by the tag, a portion of the address of the original location of the data in the main memory, to allow a checking to be performed every time an access to the cache happens, making sure that the right data is being fetched from the cache. Nevertheless, tag checking during every cache access contributes to the total amount of power consumed by caches. This is the motivation behind the solutions presented in this section. Note that this is somehow intertwined with the idea of loop caches, which are tagless caches, and scratchpads. The key difference is that loop caches and scratchpads do not even have tag arrays in their architecture, while the aim of the solutions presented in this section is to minimize the access to the existent tag arrays of the caches.

The first idea, as presented in [17], is based on the concepts of intrablock and interblock flows. Assume that two instructions are fetched from the cache. If both instructions fall into the same cache line (or intrablock), then for the second instruction it is not necessary to perform tag comparison because it is certain that it will be a hit. Therefore only for interblock flows the tag check must be performed.

In [8] this notion was extended to be able to handle a larger window of memory accesses without the need to check tags. The basic idea behind this approach is the fact that the state of caches changes only when there is a miss, which means that new instructions must be fetched from the main memory. Thus, between two misses, the state of the cache remains stable. Therefore if an instruction is accessed repeatedly during this stable-time (as called by the author of the paper), only at the first reference a tag check has to be performed. To be able to detect the conditions for not performing unnecessary tag checks, the cache proposed in the paper, called History-Based Tag-Comparison (HBTC) cache, records execution footprints in an extended Branch Target Buffer (BTB) [8]. It is important to note that the HBTC cache works only with direct-mapped instruction caches.

For a DM cache with 16KB - 32 bytes on the cache line a reduction of about 90% on tag comparisons was reported, leading to about 15% reduction on energy consumption.

The drawbacks of this solution are the need of a BTB in the microprocessor and the fact that extra cycles are added to the cache access when an invalid footprint is found.

2.2.7 Phased Caches (new)

In a regular direct-mapped cache the tag and data lookup are performed at the same time, even when internally the tag and data arrays are two separate memory elements as it is defined in the CACTI model [14] and portrayed on figure 2.7.

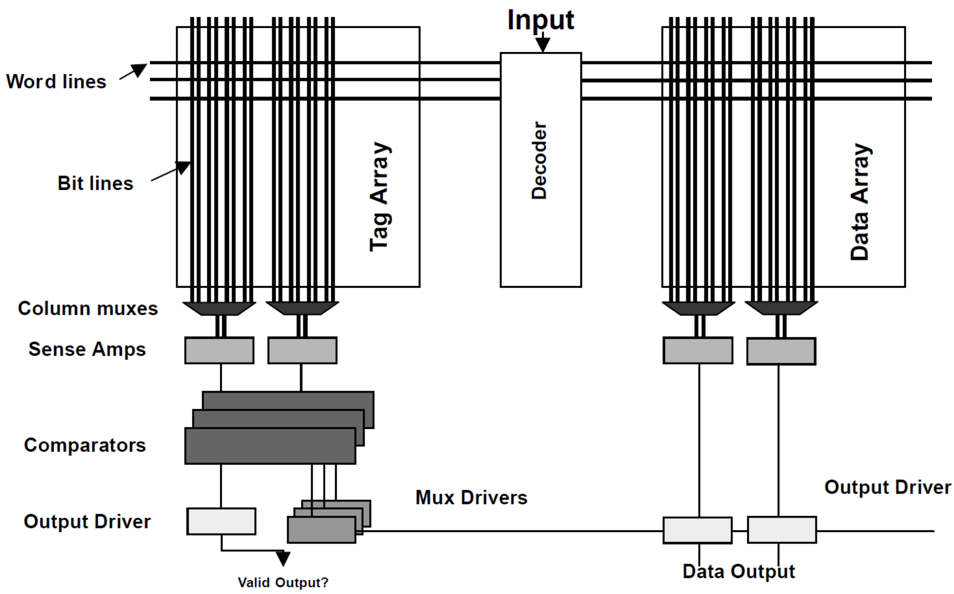

Figure 2.7: Cache memory internal architecture (taken from [14]).

The idea behind a phased cache is to split up the tag and data lookups into two different points in time, in that manner the tag lookup is performed prior to the data lookup which is interesting in terms of energy consumption because when there is a miss, that is, a match was not found in the tag array then there is no need to access the data array, therefore saving energy. The obvious drawback of this design is the fact that the latency of a cache access increases dramatically. For instance if both tag and data arrays are accessed synchronously then two clock cycles are needed to perform one read.

Another important issue regarding phased caches is how the memory is internally organized. On one hand the phased cache can be implemented with two different memories for the tag and the data arrays, or it can be designed as a single entity with phased accesses. By using two different memories the design becomes easier because off-the-shelf SRAM components can be used, however, a specifically designed memory unit is better optimized in terms of latency and particularly in terms of energy consumption because of the internal logic that can be shared between the tag and data arrays.

Chapter 3

Method

In this chapter, the preparations and the methodology employed to perform the desired experiments are described. Firstly, the high level simulation environment developed in Python, along with the new memory systems being evaluated, is described. Later, the details related to the SystemVerilog implementation are provided. Finally, the experiments realized in the scope of this project are presented.

3.1 High Level Simulations

3.1.1 Architectures Selected

In [2] the direct-mapped cache with indexing and the scratchpad memory were evaluated in a high level simulation environment. In this work new cache architectures are added to the simulation environment and the rationale behind such decisions are presented below. (For a full explanation of how the indexing solution works, please refer to [2]).

It is important to mention that the same requirements and constraints applied to [2] when choosing which architectures should be evaluated were also taken into consideration this time. These requirements and constraints, quoted from [2], are:

- The internal signals of the microprocessor are not visible to the memory system, all that is available are the addresses and data (instructions) that come through the bus.
- The whole system should work under the same system clock.
- Preferably, no processor stalling should happen.

- Area and current leakage should be kept to a minimum.
- The microprocessor does not have a Branch Target Buffer (BTB).
- Given the fact that a proprietary compiler is being used, no compiler techniques can be employed.

3.1.1.1 Direct-mapped, two-way and four-way phased caches

As explained before, the basic concept of a phased cache is the separation between the tag and data lookups. The realization that the microcontroller system being evaluated in this master thesis could benefit from a phased cache comes from the observation of how the memory system is connected to the microcontroller. The memory system communicates with a microcontroller through an AHB-Lite bus, using pipelined transactions, which means that each transaction is performed in two different steps composed of an address phase and a data phase. An example of such transaction can be seen in the timing diagram in figure 3.1.

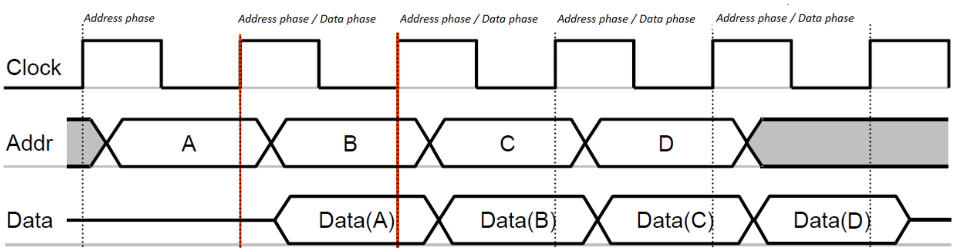

Figure 3.1: Pipelined memory access (taken from [24]).

The idea is then to use the address phase to perform the tag lookup, hence being able to determine whether a hit occurred before it actually needs the data to be available, which is in the next clock cycle. Thus, by the beginning of the data phase it is already known whether the instruction should be fetched from the Flash memory or the data array.

Taking advantage of the fact that the data array is not accessed all the time, the idea of evaluating also the two-way and four-way set associative caches came into being. While in [2] and [5] these two solutions were deemed worse than the direct-mapped version, it was considered worth investigating how they would perform when implemented as phased caches.

As with the other caches evaluated, each line of each way stores only one word (in other words, cache block size: 32 bits).

3.1.1.2 Way-halting

The other cache architecture being evaluated is the Way-halting solution presented in [31]. It has a similar goal as the phased cache, but it is implemented in a completely different way. It also splits the memory access in two different stages, so that in the first phase it performs a parallel lookup in all cache lines using the least significant bits of the address, then only for the ways in which there was a match the final lookup is performed. As explained in section 2.2.4.1 and fully described in [31], the main benefit of this solution comes from avoiding accessing the ways that are known not to hold the desired data.

To better understand how this works and why it is an attractive idea, it is interesting to look at an example (also taken from [31]). In figure 3.2, a snapshot of an access to a given address in the cache is shown. In this scenario, the tag has 21 bits (the addresses are represented in hexadecimal).

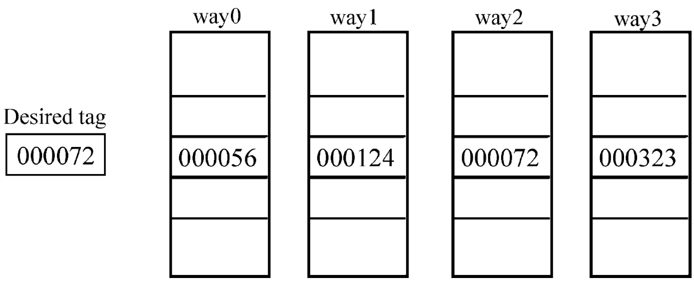

Figure 3.2: Snapshot of access to a way-halting cache (taken from [31]).

In the way-halting architecture described in [31] the 4 LSBs, stored in the fully-associative memory called halt-array, are compared simultaneously in all ways. For the given example, that would result in the following comparisons, in which the 4 LSBs of the desired tag (000072_{16}) are 0010_2:

- $0010_2 == 0110_2$? No, hence way0 halted.
- $0010_2 == 0100_2$? No, hence way1 halted as well.
- $0010_2 == 0010_2$? Yes, way2 not halted.
- $0010_2 == 0011_2$? No, way3 also halted.

Therefore, instead of fully accessing 4 ways, only one way was accessed plus the halt-arrays.

The way-halting cache implemented in this project has the following configuration:

- It is a 4-way set associative cache.

- Each line in each way stores only one word (32 bits).
- It uses the 4 least significant bits (LSB) in the way-halting array.
- It uses the Random Replacement policy.

This setup is a mixture between the configuration proposed by [31] and what is possible to implement in the target microcontroller system. The main restriction applied to the original architecture is the fact that the cache line is composed of only one word, instead of 8 in the original one. This constraint is due to the fact that 32 bits is the width of the connection between the microcontroller and the Flash memory and that in order to fetch more words to put in the cache, more clock cycles would be needed, which is an unwanted situation.

3.1.2 Simulation environment (contains text from[2])

For clarity, the system being simulated is shown again on figure 3.3.

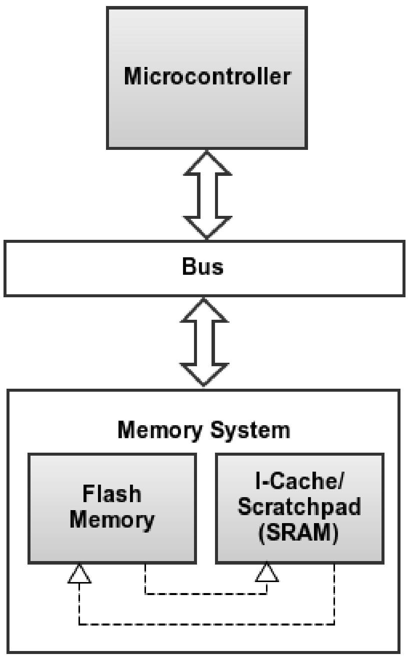

Figure 3.3: Simulated system.

To perform the desired evaluations, a high-level simulation environment that mimics the accesses to the memory system was developed in Python [26]. This environment was used basically to calculate the total energy consumption for an execution of a given program trace. A basic overview of the environment can be seen in the class diagram provided in figure 3.4 (new cache architectures not included in the diagram).

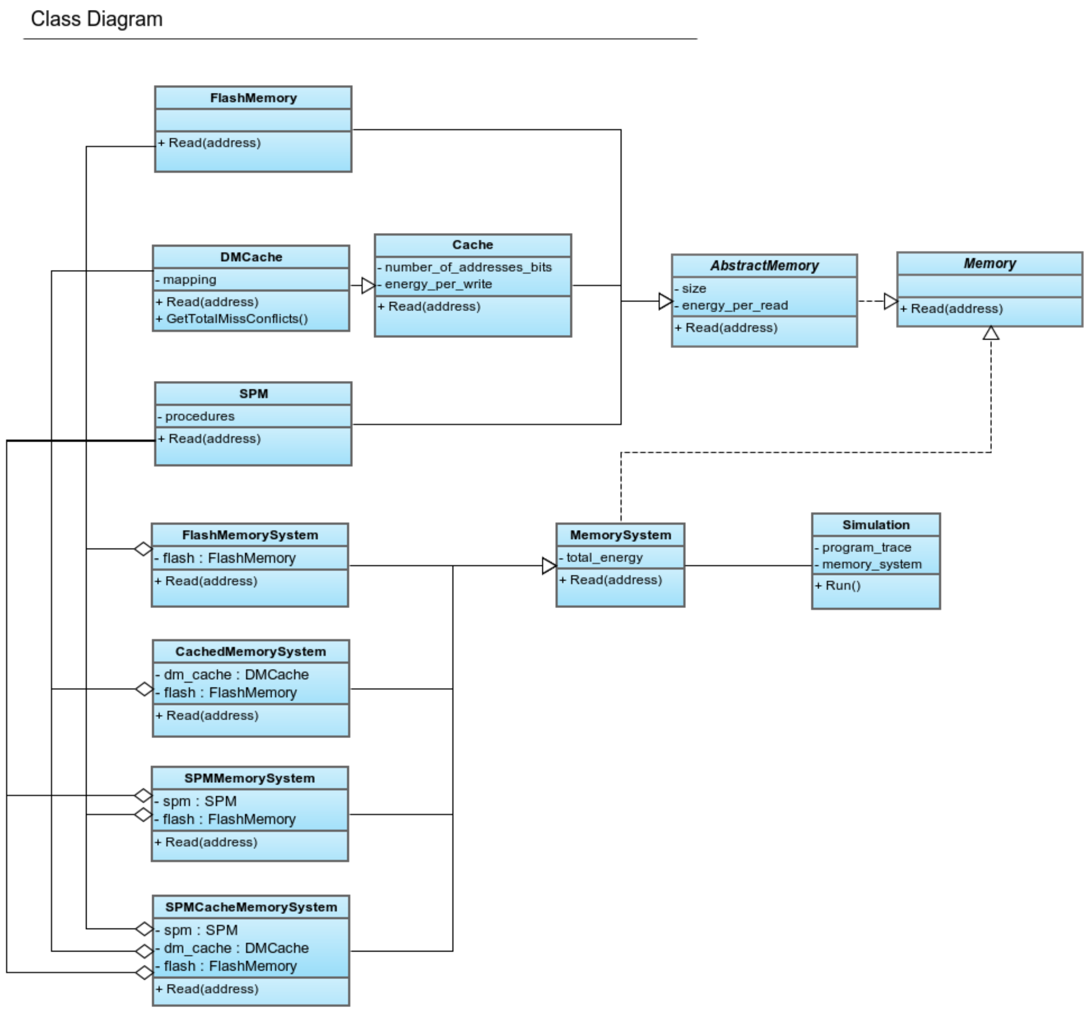

Figure 3.4: Simulation Environment Class Diagram.

In order for a simulation to happen, the following elements should be present:

- Simulation object.
- Program trace.
- Memory System.

The simulation object is in control of the execution. It sets up the system with the desired parameters and runs the simulation. The program trace acts as the processor, providing the addresses of instructions that should be fetched from memory. The memory system is the core of the environment, since it is its behavior that the simulation is trying to reproduce. Its basic functionality is implemented in the `Read` method.

A memory system can be composed of many memory elements, thus being able to represent different hardware configurations. For example, to simulate a system composed of a Flash memory and DM cache, the objects `DMCache` and `FlashMemory` can be combined to form a memory system of the type `CachedMemorySystem`.

It is very important to mention that, although a high-level simulation environment, such as the one used here, increases dramatically the time spent on the evaluation phase (not only because it is faster to run, but also because it is quicker to implement than a more detailed version written in Verilog, for example), it hides essential issues that must not be overlooked when it comes to actual hardware implementation.

For example, for the `CachedMemorySystem`, the `Read` method is implemented as presented below.

```
def Read(self, address):

    hit = self.dm_cache.Read(address)
    self.cache_energy += self.dm_cache.energy_per_read
    if hit == 1:
        return
    else:
        self.miss_count += 1
        self.cache_energy += self.dm_cache.energy_per_write
        self.Flash_energy += self.Flash.energy_per_read
```

What this method is modeling is the following:

1. The DM cache checks whether the contents of `address` is in the cache. If it is, it adds the energy spent reading from cache and no access to the Flash memory is made.

2. If there is a miss, then data must be read from the Flash memory and written into the cache. The energy consumption of both accesses is computed.

What is implicitly assumed in this scenario is the fact that the entire reading operation fits the requirements described in section 3.1.1, in particular those regarding time, which are reproduced below:

- The whole system should work under the same system clock.

- Preferably, no processor stalling should happen.

Therefore, before proceeding any further, an analysis of whether that would be possible to be implemented in hardware was made.

The very same concerns were observed when implementing the new cache architectures being evaluated in this master thesis. Thus, the following memory systems classes were added to the simulation environment: `PhasedDMCacheMemorySystem`, `PhasedCacheMemorySystem` (play the role of 2-way and 4-way phased caches) and `WayHaltingCacheMemorySystem`.

3.2 SystemVerilog Implementation

On [2], the direct-mapped cache with indexing solution was presented and evaluated using the simulation environment described in the previous section. To further validate the findings obtained previously, a register-transfer level implementation of the solution was performed in the scope of this master thesis up to the point in which it was possible to run power simulations in the developed cache system. However, instead of implementing the regular direct-mapped cache controller, it was decided, based on early findings on the high level simulations, that the phased cache version would be implemented. As it will become clear later on, when the results are presented, the gains obtained by using the phased cache are somewhat modest, however, from the designer perspective, implementing the phased cache version is a bit more challenging, therefore more interesting for the purpose of a design exercise.

This section presents the design steps employed in developing the direct-mapped cache with indexing controller along with the explanation and requirements that guided such implementation. It also describes briefly the set of tools utilized in the process.

3.2.1 Design of the Cache Controller

A block diagram of the cache controller with its main elements and connections to the other parts of the system is shown on figure 3.5. The main task of a cache controller is to coordinate the accesses to the given memory elements presented in the system, such as the Flash memory and SRAM memory. Therefore, a critical issue in the design of such a cache controller is to determine which types of memories must be used and the interface between these components.

Because the ultimate goal of Nordic Semiconductor is to include a cache system into one of their microcontroller systems, a fair amount of time of the design process was dedicated to studying existent components and interfaces that would need to

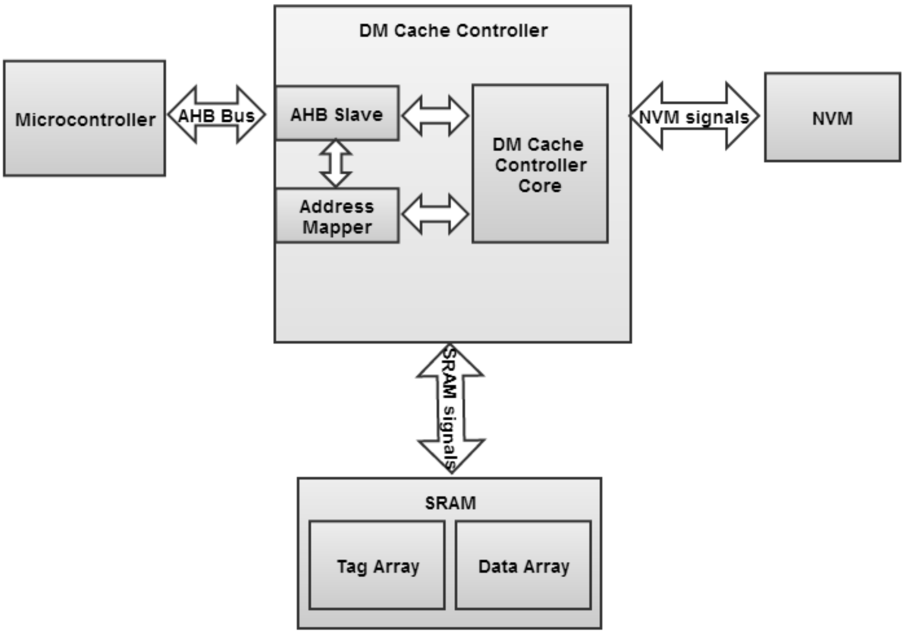

Figure 3.5: Block diagram of the cache system.

be part of the cache system in case it got integrated to Nordic's System-on-Chip. However, after careful consideration, it was opted to isolate the design of the cache system in order to facilitate and speed up the evaluation of its energy efficiency. Thus, a simplified interface to the Non-volatile Memory (NVM) was chosen, leaving more room for the decision of which type of memory and interface should be used for the tag array and the data array, which is how the memory is organized on the phased cache.

3.2.1.1 Defining the Memory System

The first design question that needed to be answered was whether to use a single port or a dual-port memory. However, it is clear to see that, given the pipelined nature of the memory accesses, the best option was to use a dual-port memory.

The next issue was to decide whether an asynchronous or synchronous RAM should be used for the tag array. In order to better understand this challenge, it is interesting to analyze again the timing diagram, shown on figure 3.1, of a basic memory transfer between the microcontroller and the memory system. It is in the address phase (the clock cycle in which the address is setup), that the tag array must be accessed. Nevertheless, it is only some time after the positive edge of the clock signal that the address signal becomes stable, leaving then two ways to access the tag memory: asynchronously, which means that after the address signal becomes stable, it will take n nanoseconds for the contents of the tag array to be available, where n corresponds to the memory reading delay, or synchronously, using then the negative edge to synchronize the access to the tag array.

In this particular case, there are two main issues that should be considered when choosing between these two solutions: timing and power consumption. An asynchronous RAM is by design more power-hungry than its synchronous counterpart, however, by using an asynchronous RAM, the access to the tag array can start before the negative edge of the clock. That results in a less stringent timing requirement, because not only the tag memory should be accessed during the address phase but also the hit signal must be asserted before the end of the clock cycle, which means comparing the output from the tag array with the tag bits from the address signal. Therefore, with the goal of keeping the timing requirement less tight, an investigation of whether an asynchronous RAM would have acceptable power consumption levels was conducted.

Power estimation for Asynchronous RAM

Asynchronous RAMs are not as common as synchronous ones mainly due to their power efficiency, as previously explained. For example, at Nordic Semiconductor there are no asynchronous RAM libraries available, which means no datasheet information. Therefore a procedure was envisioned in order to estimate the power consumption of an asynchronous RAM. It consisted of the following steps:

- Implementing in Verilog an asynchronous RAM module;
- Synthesizing this module in order to get average power consumption values;
- Synthesizing the SRAM models provided by the Artisan tool to get their average power consumption values;
- Defining a relationship between the values obtained through synthesis and the ones from the datasheet from the Artisan tool in order to obtain the energy per read and per write values of the asynchronous RAM.

A different approach to find the energy per operation values would be using the same methodology applied to obtain the power consumption values of the cache system, that will be described later in the text. However, at that time, this procedure was not entirely clear and therefore, in order to speed up the design phase, it was decided in favor of the technique described in this section.

The values obtained with such method, which will be disclosed in the experiments section, were deemed satisfactory at that time, hence validating the use of the asynchronous RAM for the tag array.

3.2.1.2 Detailed Design Description

The block diagram of the cache controller was already shown on figure 3.5 in the previous section and figure 3.6 provides a detailed view of its interface.

Each one of the modules connected to the cache controller core have their signals described in this figure. On column *Sync* there is the information of whether the signal is synchronized to the *ahbHclk* or is an asynchronous one. Note also that some signals are parametrized and these parameters depend basically on the size of the memory connected to the controller (**NB_TAG** for example means the number of bits in the tag).

Timing requirements

As explained before, the *Hit* signal must be determined before the end of the AHB-Lite address phase. This means that the *SRAMaddrR*, *SRAMceTag* and *SRAMreTagR* signals into the SRAM must be ready early enough for a reading to be realized at the tag memory before the end of the clock cycle.

This behavior is summarized in figure 3.7.

Signal	In/ Out	Sync	Description
Core			
arst	In		Asynchronous reset.
AHB-Lite Bus			
ahbHSel	In	ahbHClk	Enable AHB access to this module.
ahbHClk	In		AHB-Lite clock signal.
ahbHWrite	In	ahbHClk	Enable write to this module, otherwise read
ahbHSize[1:0]	In	ahbHClk	Word width of transfer, 2=32 supported.
ahbHTrans[1:0]	In	ahbHClk	AHB-Lite transfer type, HTRANS. Only bit 1 used, i.e. IDLE (00) and non-sequential (10) are implemented
ahbHAddr [NB_HADDR-1:0]	In	ahbHClk	AHB-Lite address bus, byte address.
ahbHWData [NB_HDATA-1:0]	In	ahbHClk	AHB-Lite data bus, write data.
ahbHReadyIn	In	ahbHClk	AHB-Lite status input, not busy.
ahbHRData [NB_HDATA-1:0]	Out	ahbHClk	AHB-Lite data bus, read data.
ahbHReadyOut	Out	ahbHClk	AHB-Lite transfer status output.
ahbHResp	Out	ahbHClk	AHB-Lite status output. Always transfer success (=0).
Cache control signals			
Hit	Out	async	Hit = 1 when there is a hit, 0 otherwise.
SRAM interface			
SRAMaddrR [NB_CACHE_ADDR_BITS-1:0]	Out	async	Address of SRAM read port (used to address each cache line).
SRAMaddrW [NB_CACHE_ADDR_BITS-1:0]	Out	async	Address of SRAM write port (used to address each line).
SRAMdataInR [NB_HDATA-1:0]	In	async	Data coming from the read port of data memory.
SRAMdataOutW [NB_HDATA-1:0]	Out	ahbHClk	Data going to the write port of data memory.
SRAMtagInR [NB_TAG-1:0]	In	async	Tag coming from the read port of tag memory.
SRAMtagOutW [NB_TAG-1:0]	Out	ahbHClk	Tag going to the write port of tag memory.
SRAMvalidBitInR	In	async	Valid bit coming from the read port of valid bit memory.
SRAMvalidBitOutW	Out	ahbHClk	Valid bit going to the write port of valid bit memory.
SRAMreTagR	Out	async	Read enable for tag memory read port.
SRAMweTagW	Out	ahbHClk	Write enable for tag memory write port.
SRAMreDataR	Out	ahbHClk	Read enable for data memory read port.
SRAMweDataW	Out	ahbHClk	Write enable for data memory write port.
SRAMreValidBitR	Out	async	Read enable for valid bit memory read port.
SRAMweValidBitW	Out	ahbHClk	Write enable for valid bit memory write port.
SRAMceTag	Out	async	Chip enable for tag memory.
SRAMceData	Out	async	Chip enable for data memory.
SRAMceValidBit	Out	async	Chip enable for valid bit memory.
NVM interface			
NVMReady	In	async	Signal whether data from NVM memory is ready at the input port.
NVMAddr	Out	ahbClk	Address to the NVM memory.
NVMce	Out	ahbClk	Enable for the NVM memory.
NVMDataIn	In	async	Data coming from the NVM memory.
Register			
indexingConfig	In	ahbClk	Set of bits with the configuration for the mapping from the main memory address to the cache address. This mapping is of the format ""000000000000111100", in which a '1' means that that bit is used in the mapping between the addresses.

Figure 3.6: Cache controller interface.

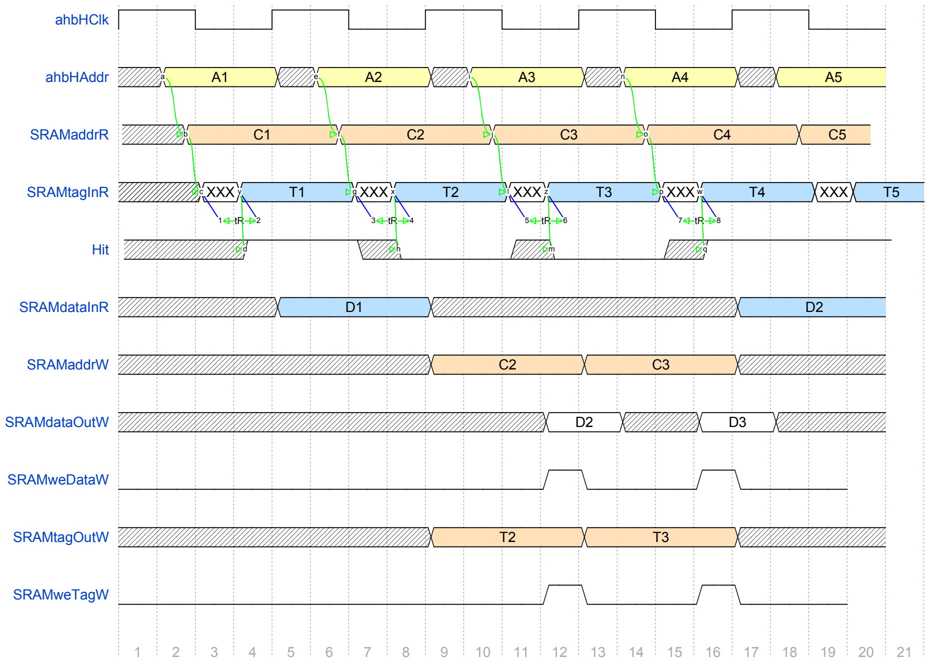

Figure 3.7: Timing diagram for cache controller.

In this figure, all possible scenarios while fetching an instruction from memory are depicted. In the following there is a brief description of each case:

- Hit occurred (time-steps: 2 to 8): note that the cache address (*SRAMaddrR*) was determined as soon as the AHB address signal (*ahbHAddr*) was stable and that lead to the reading of the tag array which was compared to the current address. Since a hit happened, the *Hit* signal was asserted before the end of the first clock cycle. This signal enabled the reading from the data array from the cache and this data is put on *SRAMdataInR*.

- Miss after a Hit (time-steps: 5 to 12): *Hit* signal is determined before the end of clock cycle, which in this case represents a miss. That enables the signals to the NVM memory which fetches the correct instruction. As the data is fetched, the SRAM is prepared to update its contents, as can be seen on signals *SRAMweDataW*, *SRAMweTagW* and *SRAMaddrW*. *SRAMdataOutW* and *SRAMtagOutW* become ready and everything is written into the SRAM at the end of the clock cycle (time-step 12).

- Miss after Miss (time-steps: 13 to 20): same as above.

- Hit after Miss (time-steps: 17 to 20): On this case the data must be read from the data array on the edge of the clock cycle at the end of time-step 16, while at the same time data must be written into the data array, therefore demanding a dual-port SRAM memory.

Cache initialization

Before reset is performed, the cache contains invalid information. That is an issue particularly regarding the valid bit, a bit present in every cache line informing whether or not that cache line contains useful data. In this case, a register file is used to store this valid bit, therefore a basic reset is all that is needed to initialize the cache.

Indexing configuration

The cache controller provides the possibility of configuring the mapping between the address bits (coming from the AHB-Lite bus) and the address bits used to select the cache line. For a thorough explanation on how this works, refer to [2]. In order to perform this configuration, the input *indexingConfig* must be used, as explained in figure 3.6.

Read Operation

As mentioned before, as far as the instruction cache is concerned, only read operations must be handled by the cache controller. Besides the AHB-Lite bus signals, this cache controller also features a signal called *Hit*, that goes high as soon as the data is read and tag-compared, which means that by the end of the address phase of the AHB-Lite bus, the read will be performed and the *Hit* signal will be set as quickly as the read operation at the SRAM happens. This signal can be used to control a second level memory, such as a Flash memory, for example, to determine whether or not a read should be performed in this second level memory.

3.2.1.3 Functional Verification

The verification framework is displayed on 3.8. The first thing to notice about it is the fact that the device under test (DUT) is not actually the cache controller, but what is called a cache system (DMCacheSystem in the block diagram), which is a combination of the cache controller and the volatile memory elements (namely the valid bit, tag and data arrays). The reason for that is that, while the design efforts concentrate on the logic of the cache controller, the controller is nothing more than an orchestrator for the memory elements. Moreover, later on, when carrying out the power simulations, the actual memory elements must be present in order to obtain relevant energy consumption values and having the test bench set up this way facilitates this task. It is also important to bring attention to the fact that the NVM and the microcontroller are part of the test bench and are ultimately just mockup implementations, behaving as the real ones would.

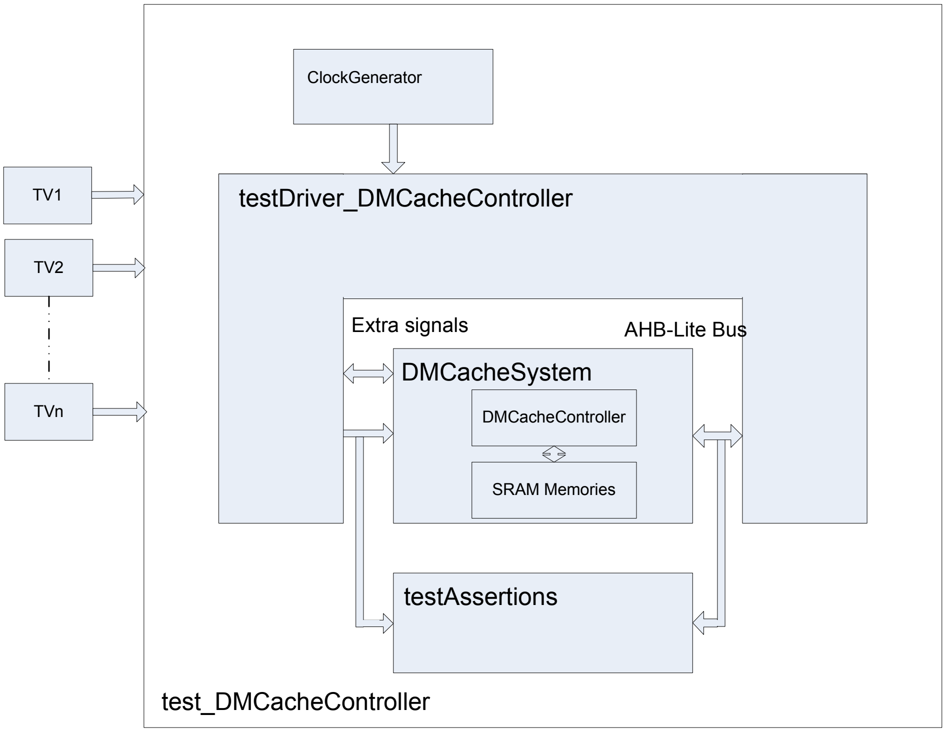

Figure 3.8: Verification framework.

Other elements present in this framework, as can be seen in the figure, are the test driver, which is responsible for generating the relevant signals to the DUT, a separate, configurable clock generator and the test vectors (called TV1, TV2 and TVn in the figure), which corresponds to the various scenarios being tested.

The test vectors provide configuration parameters to the test bench in order to verify as many scenarios as possible. There are in total 32 test vectors. And as it is the case with the high level simulations, these test vectors go through the trace file corresponding to the execution of the Bluetooth Low Energy application and check the contents of each memory element at every clock cycle. It also counts the number of hits that happened during such execution and compare to the expected results.

In order to run all these test cases, the simulation environment provided by Nordic Semiconductor is used. That is comprised of a set a scripts responsible for coordinating the test execution and generating the simulation output reports.

3.2.1.4 Synthesis

As for the verification, the tools and set of template scripts provided by Nordic Semiconductor were used in order to synthesize the cache system. Both logic and memory elements are synthesized to 180 nm technology.

3.2.1.5 Power Simulations

In order to perform the desired power simulations, the tool Primetime PX from Synopsys [25] was used.

The tool allows for different types of power analysis, such as average or time-based power analysis. Different files format are also accepted as input. The basic power analysis flow employed in this project is portrayed in figure 3.9.

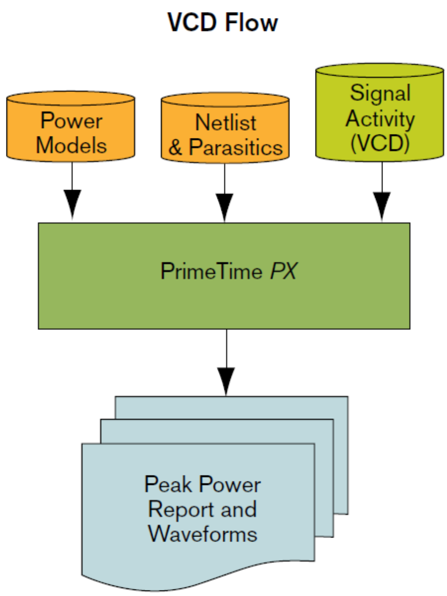

Figure 3.9: Power Analysis Flow (taken from [25]).

According to [25], “the VCD-based analysis is extremely accurate since all the factors contributing to power consumption are supported in an accurate form”. The VCD files used in this flow were obtained by running the experiments described later on in section 3.3.2.

In this analysis mode, the power report outputs the average value of three components of power dissipation, namely: leakage, switching and internal power. A simple but clear explanation of these three components is provided on [25]. The power report also presents the average power for each hierarchical instance inside the module, such as the cache controller core, the AHB-Lite slave, the tag array, the data array and so on.

It is important to mention that the parasitic information, as shown as part of the power analysis flow, was not present when the power simulations were performed. In order to get the parasitic information, it would be necessary to go up to the layout level of the cache system and this project stopped at the gate-level netlist. While this means less accuracy, in terms of final results, previous reports from other cases at Nordic Semiconductor showed that the variation in the outcome is related mostly to the switching power, which in turn does not represent a great part of the total power consumption, as will be clear when the results are presented.

3.3 Experiments

3.3.1 High-level Simulations

For this project, the memory systems modeled and evaluated on the simulation environment described previously are listed below. Between brackets is the name of the class used to model each scenario.

- Flash memory only. Used for comparison against all other scenarios. [`FlashMemorySystem`]
- Flash memory + DM cache (with sizes 128, 256, 512, 1K, 2K, 4K, 8K, 16K and 32K bytes). [`CachedMemorySystem`].
- Flash memory + DM cache with indexing (with sizes 128, 256, 512, 1K, 2K, 4K, 8K, 16K and 32K bytes). [`CachedMemorySystem`].
- Flash memory + DM phased cache (with sizes 128, 256, 512, 1K, 2K, 4K, 8K, 16K and 32K bytes). [`PhasedDMCacheMemorySystem`].
- Flash memory + DM phased cache with indexing (with sizes 128, 256, 512, 1K, 2K, 4K, 8K, 16K and 32K bytes). [`PhasedDMCacheMemorySystem`].

– Flash memory + 2-way set associative phased cache (with sizes 256, 512, 1K, 2K, 4K, 8K, 16K, 32K and 64K bytes). [`PhasedCacheMemorySystem`].

– Flash memory + 4-way set associative phased cache (with sizes 512, 1K, 2K, 4K, 8K, 16K, 32K, 64K and 128K bytes). [`PhasedCacheMemorySystem`].

– Flash memory + 4-way halting (with sizes 512, 1K, 2K, 4K, 8K and 16K bytes). [`WayHaltingCacheMemorySystem`].

All the relevant parameters used throughout the simulations are listed on tables 3.1, 3.2 and 3.3 (please note that E/R means Energy per Read and E/W means Energy per Write in these tables). The parameters related to the Flash memory were provided by Nordic Semiconductor, while the cache parameters were obtained as explained in the following section.

Parameter	Value
Clock	16 MHz
Flash size	256 kB
Flash energy per read	0.500 nJ
Flash leakage power	0 W
Flash access time	30 ns

Table 3.1: Main parameters used during simulations.

Size (bytes)	E/R Tag	E/W Tag	E/R Data	E/W Data	E/R Line	E/W Line
128	0.0084	0.0054	0.0114	0.0072	0.0126	0.0078
256	0.009	0.0066	0.012	0.0084	0.0138	0.0096
512	0.0102	0.0078	0.0144	0.0114	0.0156	0.0126
1024	0.0132	0.0114	0.0186	0.0174	0.0204	0.0192
2048	0.0144	0.0156	0.0276	0.0342	0.0312	0.039
4096	0.021	0.0216	0.0474	0.054	0.054	0.0612

Table 3.2: Register File memory parameters (energy per access in nJ).

3.3.1.1 Energy Models

As presented in [2], the Artisan tool was used to provide the energy per read/write values for the high level simulations. However, for this master thesis different values were obtained. The first reason for that is that, as explained before, there is the need to use a dual-port memory instead of single port, which was the type utilized in [2]. Moreover, in [2] only one type of technology was used, which was the Register File memory type. In this master thesis a different type of technology, a SRAM memory type, was employed

Size (bytes)	E/R Tag	E/W Tag	E/R Data	E/W Data	E/R Line	E/W Line
128	0.0282	0.0312	0.06	0.0702	0.0768	0.09
256	0.027	0.03	0.06	0.0708	0.0756	0.0894
512	0.0258	0.0288	0.0612	0.072	0.075	0.0888
1024	0.0258	0.0282	0.063	0.075	0.075	0.09
2048	0.0264	0.0288	0.066	0.0804	0.0774	0.0948
4096	0.0294	0.0324	0.0726	0.0906	0.0828	0.1044
8192	0.0336	0.0372	0.084	0.111	0.0936	0.1248
16384	0.0342	0.0378	0.114	0.1416	0.1272	0.159
32768	0.0366	0.039	0.1764	0.2058	0.1914	0.2238

Table 3.3: SRAM parameters (energy per access in nJ).

in order to gauge the impact it would have in the cache system. These two types of memories differ in their internal organization, mainly the storage element. In a Register File memory type, the basic storage element is a D flip-flop, while in the other it is the SRAM cell. (A word of caution is advised here to avoid further confusion: in terms of concept, the memory employed in the cache system is called throughout the entire text as an SRAM, such as in the introduction section. However, this conceptual SRAM memory can be implemented in different ways and the two ways cited above are connected to the names the Artisan tool gives to them, which is a Register File memory and a SRAM memory, as just explained).

Yet another change made regarding [2] was in the energy values of memories in which the size was larger than the maximum allowed by the Artisan tool. In [2] a polynomial approximation was used to estimate the energy values of the memories of size larger than 4K bytes. This approximation is disregarded in this project and instead new values were obtained by implementing in Verilog memories of a larger size by combining two or more of the 4K bytes memories and getting average power values after synthesizing them. It turned out that these values were much higher than the ones used in [2], and in fact they were too high to be used at all, hence they were dropped altogether, leaving only the datasheet values obtained with the Artisan tool. Nevertheless, it is worth noting that the SRAM version of the Artisan tool is able to generate larger memories, up to 32K bytes.

Due to the phased cache implementation, energy values were obtained for the different tag and data arrays. Therefore, for each memory type there are values for the tag (with varying number of bits), data (always 32 bits) and full-line, which combines tag and data into one line.

Again, for these high level simulations, as in [2] the datasheet values were divided by 2, following the advice from Nordic Semiconductor.

The final values for energy per access can be seen on tables 3.2 and 3.3.

3.3.1.2 Calculating Energy Consumption

The energy consumption calculation depends on the memory system being evaluated. For the memory systems with cache, it is necessary to know the number of hits and misses. For every miss in the cache, data must be retrieved from the Flash memory and then written in the cache. The calculation is unique for each architecture modeled.

The equations used in these different setups are:

DM cache:

$$FlashEnergyConsumption = NumberOfMisses * FlashEnergyPerRead$$
$$CacheEnergyConsumption =$$
$$NumberOfMisses * (CacheEnergyPerRead + CacheEnergyPerWrite) +$$
$$NumberOfHits * CacheEnergyPerRead$$
$$TotalEnergyConsumption =$$
$$FlashEnergyConsumption + CacheEnergyConsumption$$

Phased DM cache:

$$FlashEnergyConsumption = NumberOfMisses * FlashEnergyPerRead$$
$$CacheEnergyConsumption =$$
$$NumberOfMisses * (CacheEnergyPerReadTag +$$
$$CacheEnergyPerWrite) + NumberOfHits *$$
$$(CacheEnergyPerReadTag + CacheEnergyPerReadData)$$
$$TotalEnergyConsumption =$$
$$FlashEnergyConsumption + CacheEnergyConsumption$$

Phased n-way cache:

$$FlashEnergyConsumption = NumberOfMisses * FlashEnergyPerRead$$
$$CacheEnergyConsumption =$$
$$NumberOfMisses * (CacheEnergyPerReadTag * NumberOfWays +$$
$$CacheEnergyPerWrite) + NumberOfHits * (CacheEnergyPerReadTag *$$
$$NumberOfWays + CacheEnergyPerReadData)$$
$$TotalEnergyConsumption =$$
$$FlashEnergyConsumption + CacheEnergyConsumption$$

Way-halting:

$$FlashEnergyConsumption = NumberOfMisses * FlashEnergyPerRead$$
$$CacheEnergyConsumption =$$
$$[(NumberOfWays - NumberOfWaysHalted) * CacheEnergyPerRead + EnergyPerReadHaltArray * NumberOfWays] * (NumberOfMisses + NumberOfHits) + NumberOfMisses * (CacheEnergyPerWrite + EnergyPerWriteHaltArray)$$
$$TotalEnergyConsumption =$$
$$FlashEnergyConsumption + CacheEnergyConsumption$$

It is important to notice that there is no factor for static power consumption in the equations above. That is the case because the values obtained for energy per read and write for the memories evaluated already include this component. Furthermore, the n-way set associative does not include the energy dissipated in the multiplexer responsible to select the correct way when a hit happens.

3.3.1.3 Benchmark

For this project, a set of applications was provided by Nordic Semiconductor. They are three in total and each one of them uses a different wireless communication protocol. The first one of them is a Bluetooth Low Energy (BLE) application, running a heart-rate monitor profile, which is an example application used in the Bluetooth Software Development Kits available at Nordic Semiconductor. The second and third ones are sample applications using the protocols ANT and Gazelle (a Nordic Semiconductor proprietary wireless protocol) to make the microcontroller communicate with a wireless device.

3.3.2 SystemVerilog

The main purpose of the experiments carried out in the scope of the SystemVerilog implementation was to measure the power consumption of the cache system. In order to do that, two types of tests were envisioned: power simulation test cases and application test cases. Both types are described below.

Power simulation test cases

In this scenario, a small, controlled test is executed in which the cache system behavior can be easily observed. These tests go through different phases of activity such as idle, cache hits in sequence, cache misses in sequence and normal operation, which is basically hits and misses spread out randomly over time. These tests are much quicker to run than the application tasks (minutes versus hours) and give great insight into the power consumption of the cache system as a whole. Furthermore, they were also used to determine the energy per read and energy per write values of the tag and data arrays of the SystemVerilog implementation. This is how this was done: as explained before, the power analysis gives the average power consumption of each module inside the cache system, thus for each phase in the test case, such as cache hits in sequence, which ultimately represents only read operations, the power consumption values for each memory element (tag and data) were obtained and divided by the length of time of such phase (remember that $Power = Energy/Time$). The same was applied for the sequence of misses, which issues writes to the memory every clock cycle. However, due to the pipelined nature of the accesses, when a miss occurs, in the next clock cycle a write will occur on both tag and data arrays, but at the same time a read will take place in the tag array, hence making the final value for energy per write of the tag array not that straightforward to calculate, which is probably the cause for the small imprecision mentioned later in section 4.2.1.

Application test cases

These tests represent the final goal of this project in terms of the SystemVerilog implementation. It measures the power consumption of the cache system while running the Bluetooth Low Energy application, which is one of the applications also executed in the high level simulations. These results are later compared against one another.

The same list of cases is applied to both the application and the power simulation test cases. Each case varies the following variables: memory type and memory size. The application test cases also varies the type of communication with the microcontroller. As explained before, the microcontroller communicates with the cache system using pipelined AHB-Lite bus transactions, however, there might be wait states between transactions. Therefore the application test cases can be of types: pipelined and with wait states.

For the memory type there are three different options, namely a hand-coded (HC) Verilog implementation, a Register File library version provided by the Artisan tool and a SRAM library version also provided by the Artisan tool. Please note that these options apply only to the data array of the memory system since the tag array was defined as an asynchronous RAM implemented in Verilog.

The memory sizes vary differently according to the type of memory. For both the hand-coded and the Register File versions the size varies from 128 bytes to 4096 bytes, while the SRAM version goes up to 16384 bytes.

Chapter 4

Results

As it is the case with other sections of this master thesis, this part of the text is again split between the high level simulations results and the SystemVerilog results. Nevertheless, at the end of this section the two are combined when a comparison between both results is presented.

The results presented here try to answer the questions raised on the introduction of this text.

4.1 High Level Simulations

4.1.1 Hit Rates

4.1.1.1 Benefits of the Indexing Solution

One of the limitations of [2] was the fact that only one application was evaluated in terms of how much improvement could be observed by employing the indexing solution to a DM cache. For this project, other two applications were evaluated. The figure 4.1 presents this result in a summarized manner. In this figure we can observe the percentage of miss reduction obtained by using the indexing solution. The results vary greatly depending on the cache size and application, as expected, since this solution is completely dependent on the mapping from the main memory address and the cache addresses. In general terms it presents only modest improvements, ranging from 0% to around 9% for caches of 8192 bytes. For the Gazelle application the improvements cease at the size of 4096 bytes, that is due to the fact that this is a small application in which its working set can entirely fit into the cache, no matter which mapping is used. Nevertheless, an outstanding improvement can be observed for both the BLE and ANT applications, reaching the peak of almost 60% for the ANT application at cache size of 16384 bytes. This is a clear indication that while no guarantees can be made regarding miss reduction, given the basic nature of the implementation, it is definitely worth trying to apply such technique.

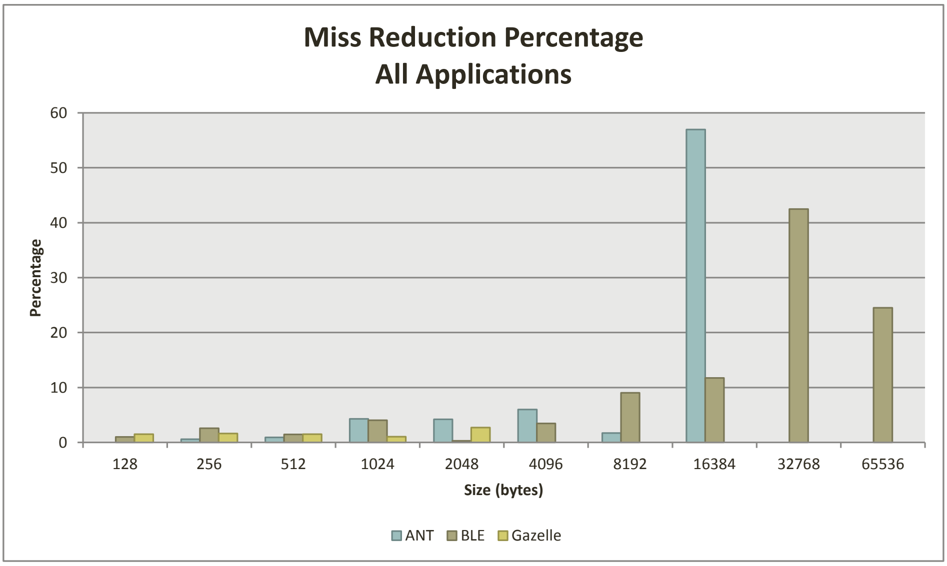

Figure 4.1: Miss reduction comparison.

4.1.1.2 Hits and Misses Comparison

The hit rate and miss rate of caches are extensively used to measure cache performance. It basically represents how successful a cache system is in avoiding accesses to the lower memory levels.

In the following figures, the results comparing all cache implementations against the set of applications available are presented. It is important to note that not all cache architectures can be implemented in all possible sizes. That restriction is in place because of what have already been explained before in section 3.3.1.1, that is, there are limits to the size of a single memory array. Thus, for instance, a 2-way set associative cache has as minimum size 256 bytes, because each way must be at least 128 bytes, which is the minimum size of a single memory array that can be generated by the Artisan tool. It is also worth mentioning that the sizes present in the X axis of each graph is normalized to the number of words that a cache can store. For example, a DM cache of 512 bytes, can store 128 words, which is the same amount of a

2-way set associative of 512 bytes (composed of two ways of 256 bytes). The final size of each cache is ultimately different, which is relevant when looking at the area footprint of each cache architecture. Furthermore, some of the cache architectures were intentionally not implemented in all sizes, such as the way-halting cache architecture. The reason for that will become apparent when analyzing its results more closely in the following sections.

In figure 4.2 we see the number of hits for the ANT application. It is interesting to note that the DM cache with indexing follows closely the performance of the 2-way and 4-way caches up to the size of (even slightly better at some points). Then it is outperformed in the next two cache sizes, but gets close again at the size of 16384 bytes, which is when, as observed in the previous section, the indexing solution provides a big boost in the number of hits. At larger sizes the different basically disappears because the cache is large enough to hold most of the application working set.

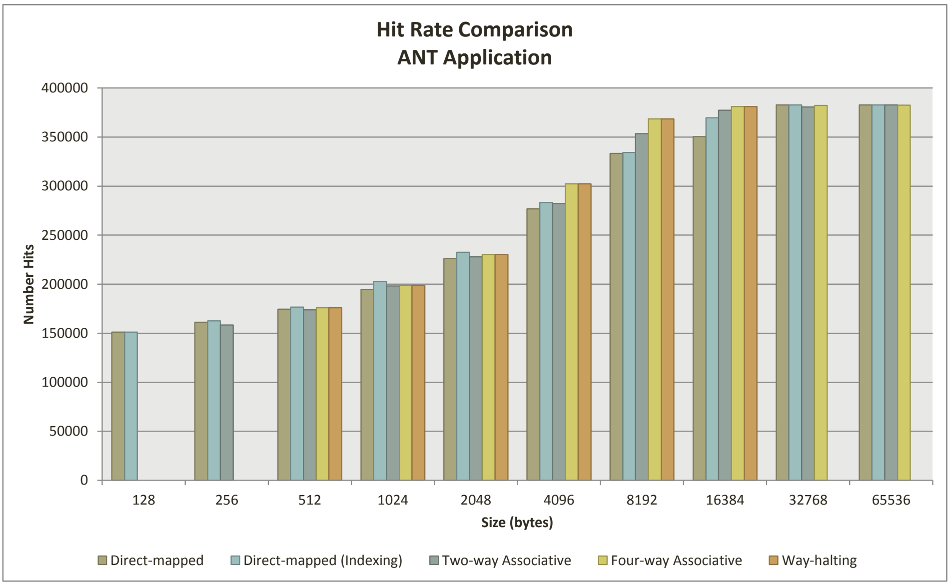

Figure 4.2: Hit rate comparison for ANT application.

A similar trend can be observed in figure 4.3, however as the application working set is significantly larger than the ANT one, the saturation is not fully observed at the large cache sizes.

On the other hand, for the Gazelle application, shown in figure 4.4, this saturation occurs even earlier than in the ANT application.

As a general remark, we can note that the hit rate performance of 2-way and 4-way caches are not significantly better than the DM cache with indexing

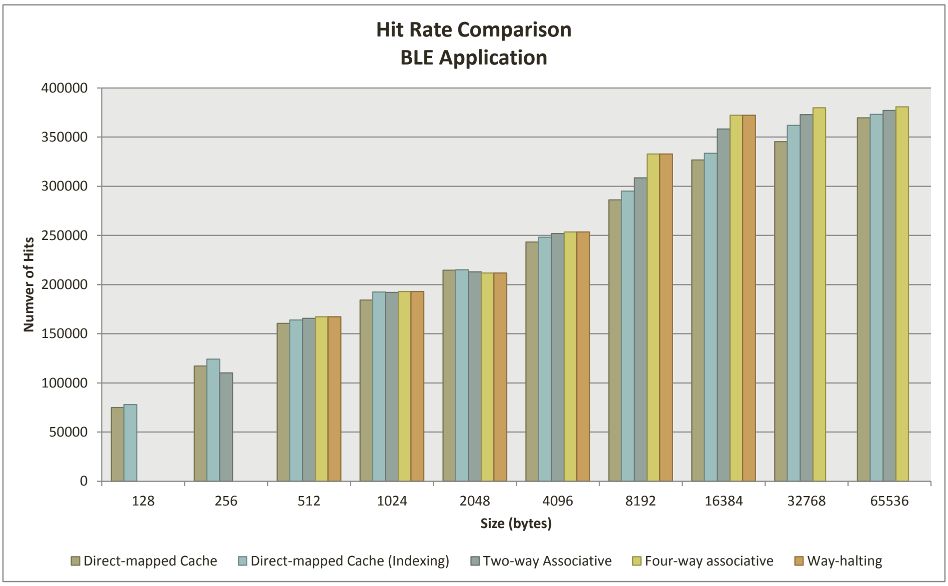

Figure 4.3: Hit rate comparison for BLE application.

(and sometimes are even worse).

A different way to look at the same metric is to analyze the number of misses on each application. Figures 4.5, 4.6 and 4.7 present these results. By looking at these graphs it is simpler to see for each cache sizes most of the misses are actually compulsory ones, that is, misses that will happen independently of the size of the cache, due to cold start (first time cache is being filled with instructions).

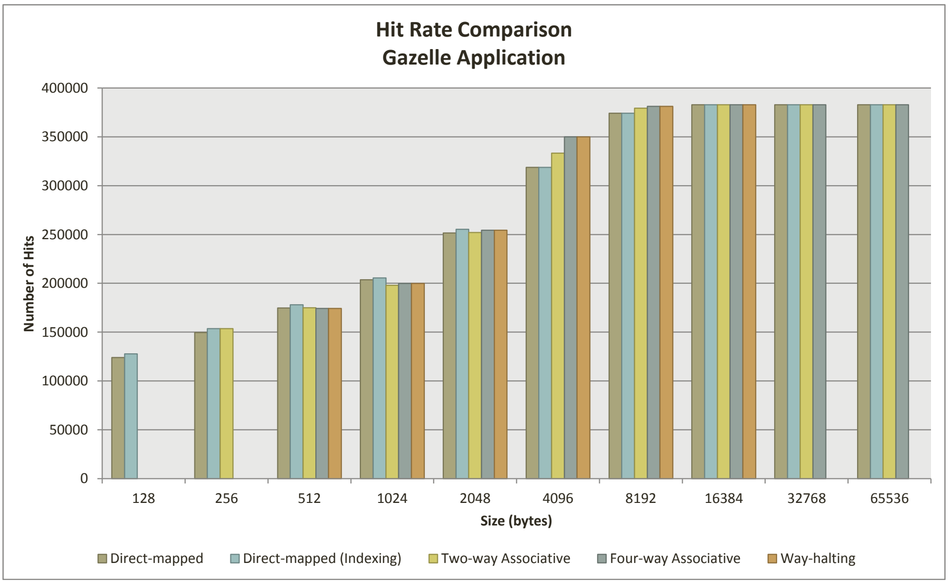

Figure 4.4: Hit rate comparison for Gazelle application.

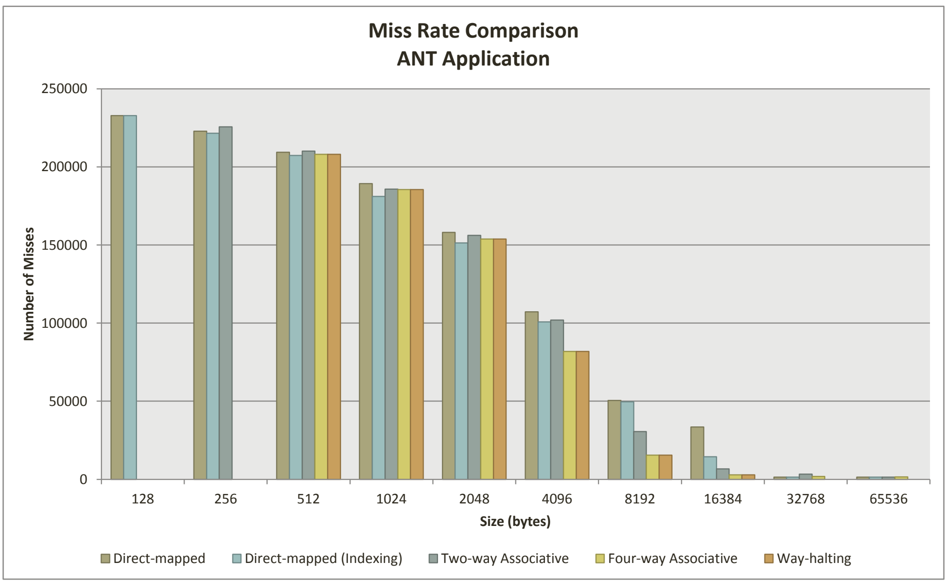

Figure 4.5: Miss rate comparison for ANT application.

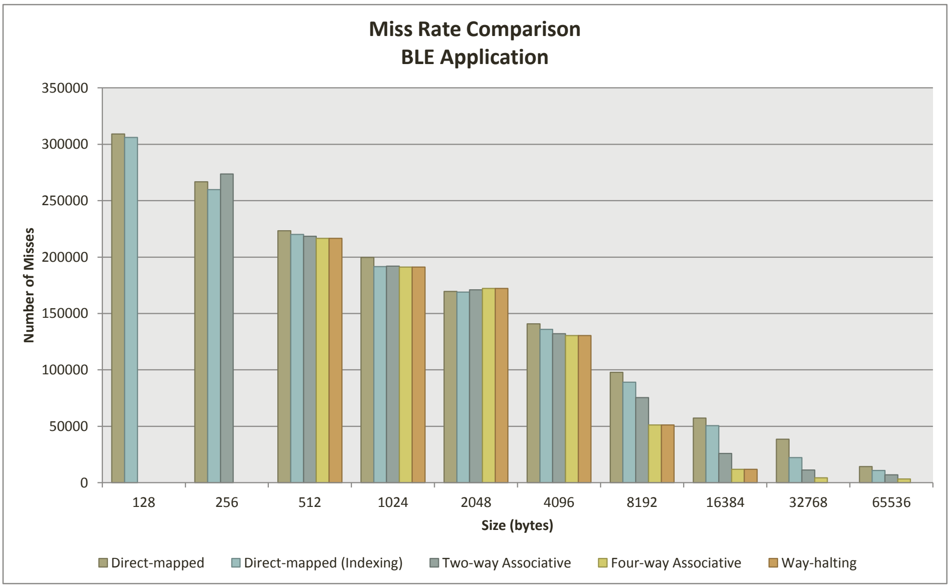

Figure 4.6: Miss rate comparison for BLE application.

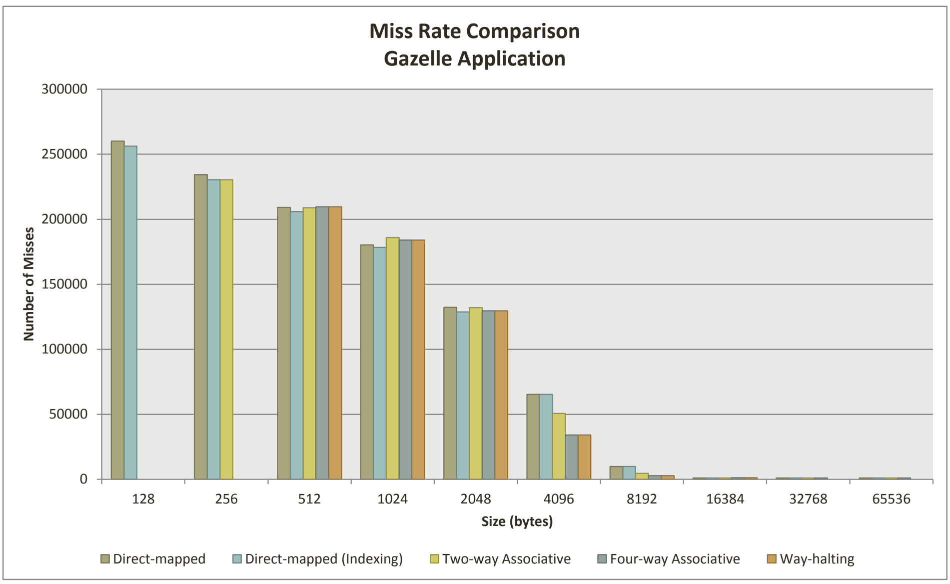

Figure 4.7: Miss rate comparison for Gazelle application.

4.1.2 Energy Consumption

4.1.2.1 Way-halting

Later on this section comparisons among all cache architectures will be presented. However, in order to highlight the results on the Way-halting implementation, this solution was compared to its normal 4-way set associative phased cache counterpart. Remember that the way-halting modeled in this project is also a 4-way cache and therefore, as it can be seen in the previous section, their hit rates are exactly the same.

In figure 4.8 we can see the comparison between these two cache architectures. What is displayed in this graph is the improvement that is obtained by using the way-halting solution instead of a 4-way set associative cache. What can actually be observed is that the way-halting solution performance is absolutely worse than the 4-way set associative phased cache, no matter which application or which cache size is compared.

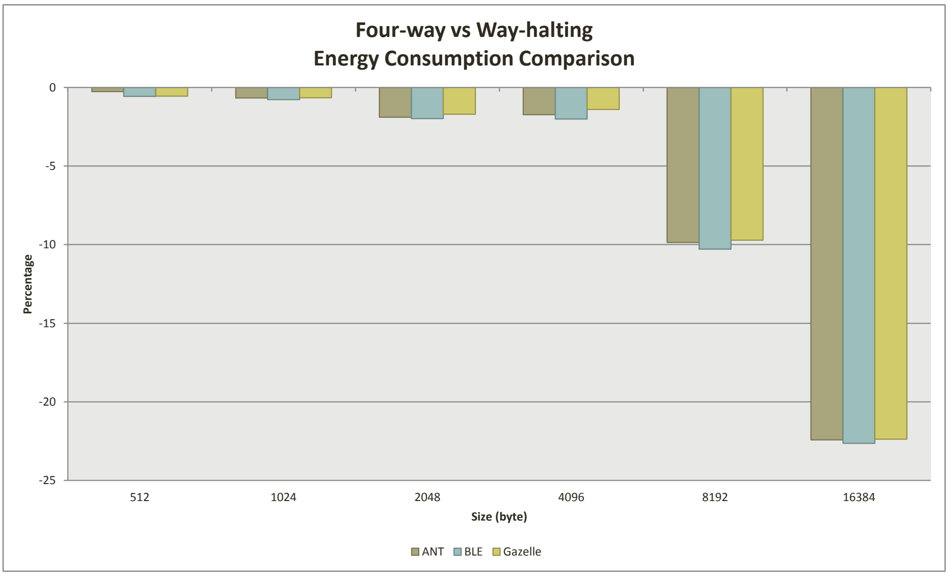

Figure 4.8: Way-halting vs 4-way phased cache comparison.

That might seem surprising at a first glance, but it turns out to be a completely normal result. The problem with this way-halting implementation is the size of the cache line, which is only one word (32 bits). Thus, when such line is fetched from memory, no neighbours come along, therefore addresses with same tags are put in different ways, resulting in less ways being halted. Remember that at each access to the cache, the 4 LSBs are compared in parallel in all ways and only when a mismatch occurs in a given line is that the particular way where this mismatch happened is halted, meaning that the way containing the rest of the tag plus the data is not accessed. But because the principle of locality is not properly explored, given the size of the cache line, the LSBs of the tag do not vary as much as it would if a larger cache line was used. In fact, across all simulations scenarios (different applications and memory sizes), the average number of ways accessed each time is never lower than 3.75.

What it is interesting to perceive is the fact that the 4-way phased cache provides more "haltings" than the way-halting cache, due to the split between the tag and data arrays. Full tags are compared simultaneously, however only the data array from the way which has a tag matched is accessed.

These findings were obtained early enough to prevent further development of the way-halting solution. That is mentioned to explain the fact that the energy consumption of the fully-associative halt tag array existent in the way-halting architecture was not even considered. To obtain energy models for such type of memory would require an effort that clearly is not worth it.

4.1.2.2 Phased Caches

One of the questions to be answered by this project is regarding the energy efficiency of phased caches. In the following figures the results of such analysis are presented.

As explained before, two different types of memories were used in the high level simulations: the Register File and SRAM. In figure 4.9 it is shown the comparison between the normal DM cache and the DM phased cache using the register file memory.

What is displayed in the figure is the amount of improvement obtained in terms of energy consumption by using the phased cache instead of the regular one. As can be seen by the negative values, the phased cache performance is actually worse than the normal cache. Although this is not an expected result, there is a reasonable explanation for that. To help understand why this happened we should look at figure 4.10.

This figure plots the difference in the current values for read and write accesses to the memory elements present in the DM phased cache memory system and the one in the regular DM cache. It is worth remembering that the phased cache memory system is composed of two separate elements: the tag and data arrays. On the other hand, the normal DM cache memory system is made of only one array (called here line array) that stores both the tag and data. The combined size of tag plus data is the same as the full line present

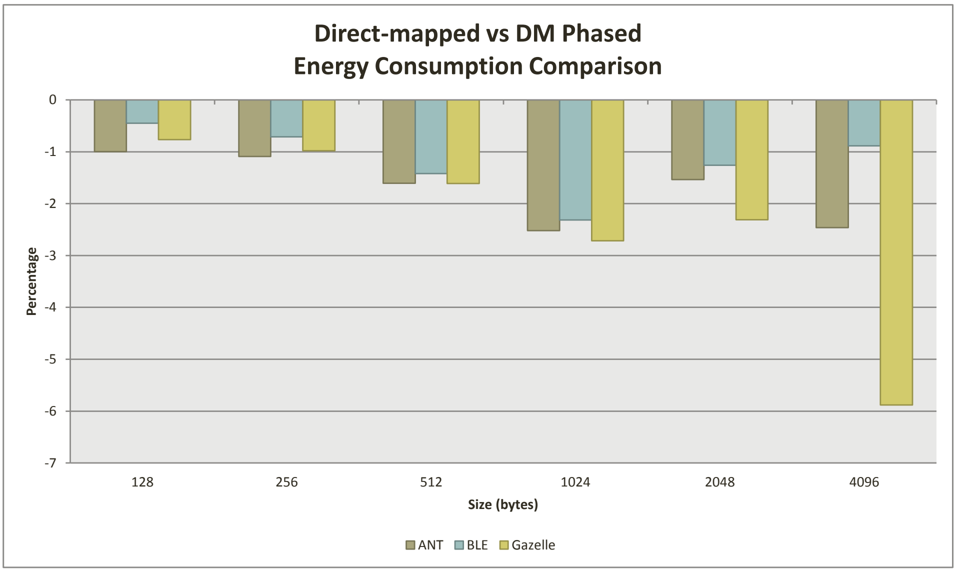

Figure 4.9: DM cache and DM phased cache comparison for Register File memory.

in the DM cache, that is, if the tag array stores tags of size equal 5 bits in the phased cache, then the line array of the normal DM cache stores 37 bits (32 for the data + 5 for the tag). The data array of the phased cache holds always 32 bits.

What can be realized from figure 4.10 is the fact that accessing the two separate arrays from the phased cache is always more energy consuming than accessing the line array from the DM cache. This is perfectly expected, since by splitting the line into two different arrays, some overhead are brought into existence (there two address decoders instead of one, for example). Nevertheless, this difference is big enough to prevent the phased cache to take advantage of the fact that the data array is not always accessed. Furthermore, this difference keeps increasing together with the memory size and so does its energy inefficiency, as shown in figure 4.9 (as a matter of fact, both figures 4.10 and 4.9 present the same trend).

However in figure 4.11, in which the DM and DM phased cache are implemented with the SRAM memory, a slightly different result is observed. In this case, for some sizes of memory, we have an improvement of the phased DM cache over the normal one. Again the same trend can be detected in the difference between current values for the two memory arrangements, as shown in figure 4.12.

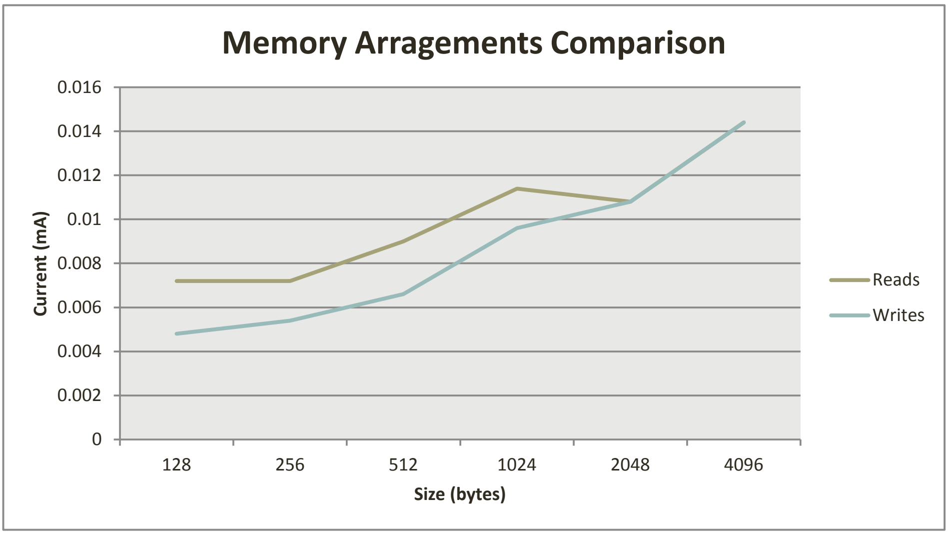

Figure 4.10: Difference in current values between phased and normal memory arrays.

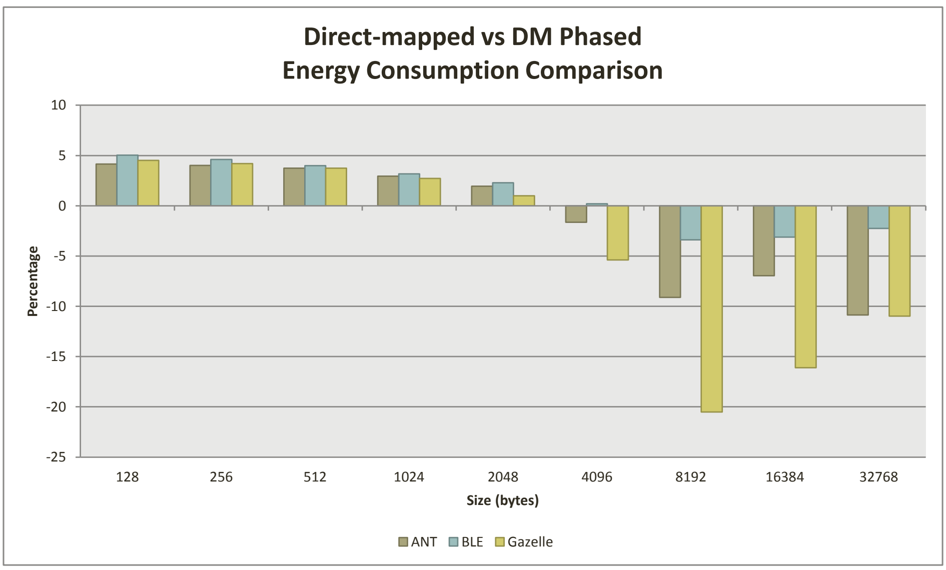

Figure 4.11: DM cache and DM phased cache comparison for SRAM memory.

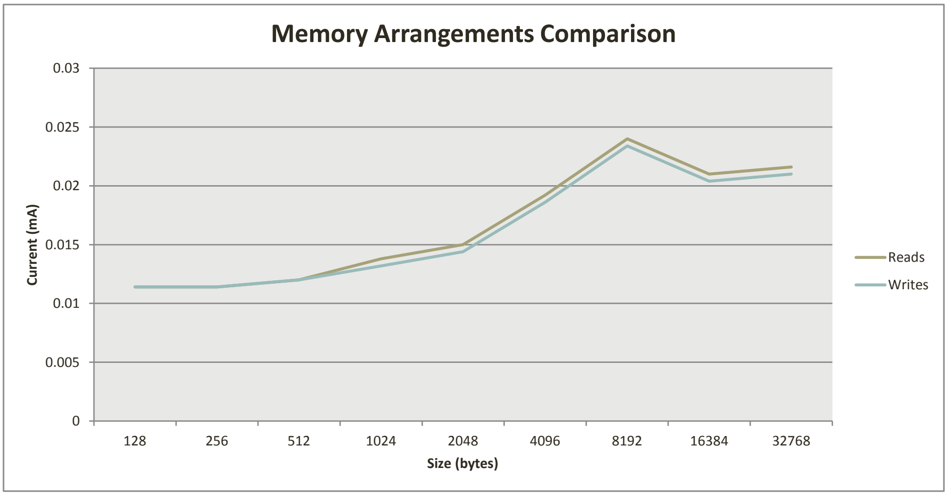

Figure 4.12: Difference in current values between phased and normal memory arrays.

4.1.2.3 Memory Types

As could be seen in the previous section, the type of memory employed in the memory system affects the outcome of the cache system. In this section a more detailed analysis of such influence is presented.

In figure 4.13 is plotted the total energy consumption of running the BLE on a system with a DM cache implemented with the Register File and SRAM memories. This figure presents a feature that will be a constant in all plots showing total energy consumption: a target line, in red, named Flash-only which represents the total amount of energy consumption that would result if no cache was added to the memory system, hence the "-only" in the name. This facilitates the observation of whether a cache system would actually contribute to energy consumption reduction or not.

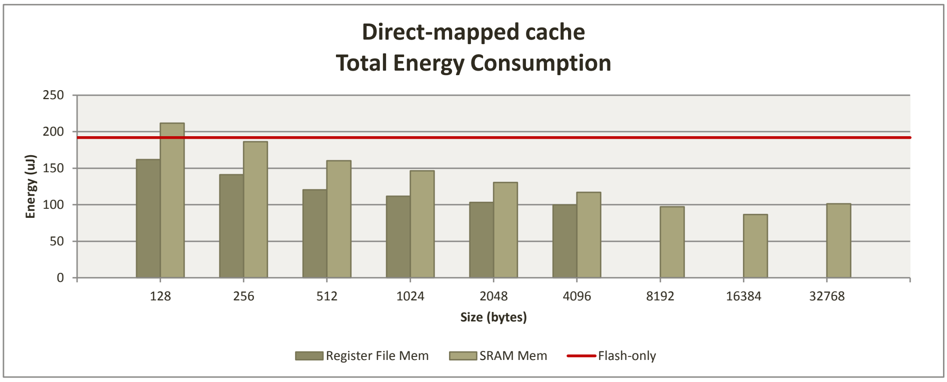

Figure 4.13: Energy consumption of DM cache for different memory types.

As mentioned before, the Register File memory is limited to the size of 4096 bytes and the SRAM to 32768 bytes. It is clear to observe that for all available sizes of the Register File memory it outperforms the SRAM version. Perhaps even more interesting is the minimum point that can be spotted in the vertical bars of the SRAM version. At the size of 16384 bytes this minimum is reached and for the next size, even though the hit rate increases, as seen in figure 4.3, the total energy consumption increases as well.

In figure 4.14 a comparison of all memory types for the DM phased cache running the BLE application is shown. The names of the curves, such *RF + RF*, mean the type of memory used for the tag and data arrays respectively. Here the results from the SystemVerilog and the high level simulations begin to merge. As explained previously, the design of the phased cache for SystemVerilog implementation entails the use of a hand-coded Asynchronous memory for the tag array, and estimations regarding energy consumption of such component were realized prior to the coding phase. Those values were fed to the high level simulations and are represented in this figure by the *Hand-coded + RF* and *Hand-coded + SRAM* curves. As it will be presented later, these values were actually greatly underestimated, therefore the values gathered after running the power simulation experiments were also fed to the high level simulations, although they were also divided by 2, as the ones from the datasheet as explained before (they have the word "Measured" between brackets in the figure).

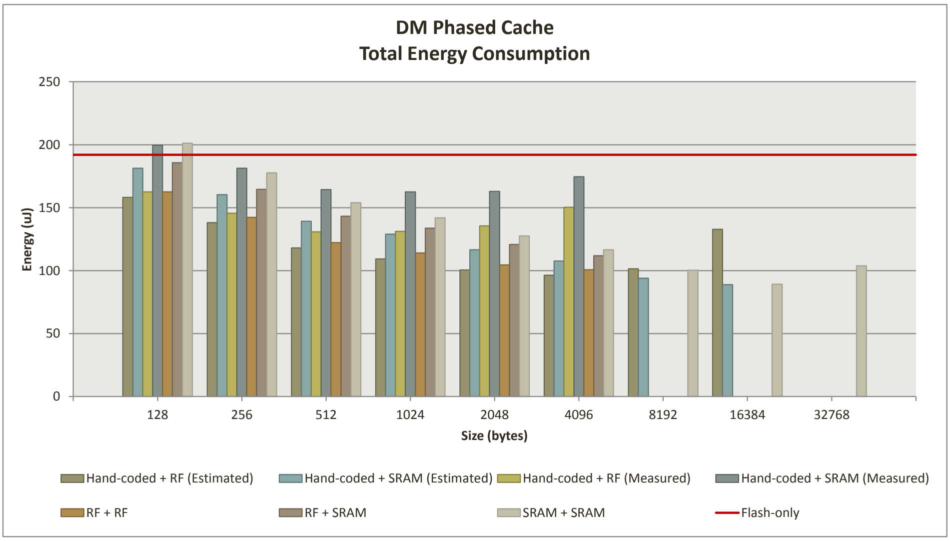

Figure 4.14: Energy consumption of DM phased cache for different memory types.

After this explanation is quite expected that the values with the word "Estimated" next to it are quite low and therefore will not take part in the analysis.

This figure also shows other possible memory arrangements for the tag and data arrays, such as tag array as a Register File memory and data array as SRAM (named *RF + SRAM* in the figure).

As with the regular DM cache, the general observation is that the memory type greatly influences the results and that using Register File memories seem more beneficial, although their maximum size is limited.

The same behavior can be seen on 4.15.

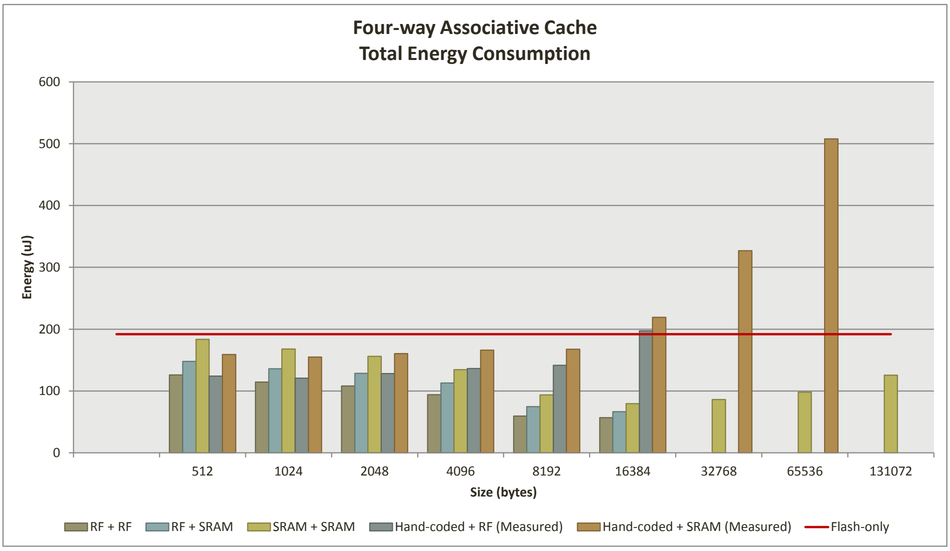

Figure 4.15: Energy consumption of 4-way phased cache for different memory types.

4.1.2.4 All Cache Architectures Compared

So far the analysis focused on specific points of the project. In this section the results of the comparison of all cache architectures modeled in the high level simulation environment are presented.

In figure 4.16 the energy consumption values of all cache architectures for the SRAM memory type are plotted. Although using the Register File memory results in smaller absolute values, for the sake of comparison the data from the SRAM memory type was used because larger cache sizes can be modeled and hence displayed in the graph.

The first interesting observation is the fact that not all cache sizes seem to be beneficial when it comes to improve energy efficiency, when using the SRAM memory. Moreover, the 2-way and 4-way have worse performances in comparison to the DM caches up to the size of 4096 bytes, despite the fact that their hit rates, for most of the cases, are the same or better than the ones from the DM cache. From that point on the energy consumption of all caches (except way-halting) basically evens out. Up to 8192 bytes, the DM caches with indexing (either phased or normal) offer better energy savings than the rest. Then at 16384 bytes both the 2-way and 4-way outperfom the DM caches, which is related to the difference in hit rate that can be observed in figure 4.3. At 16384 bytes it is also the turning point for all cache architectures, since it is after this size that energy consumption stops

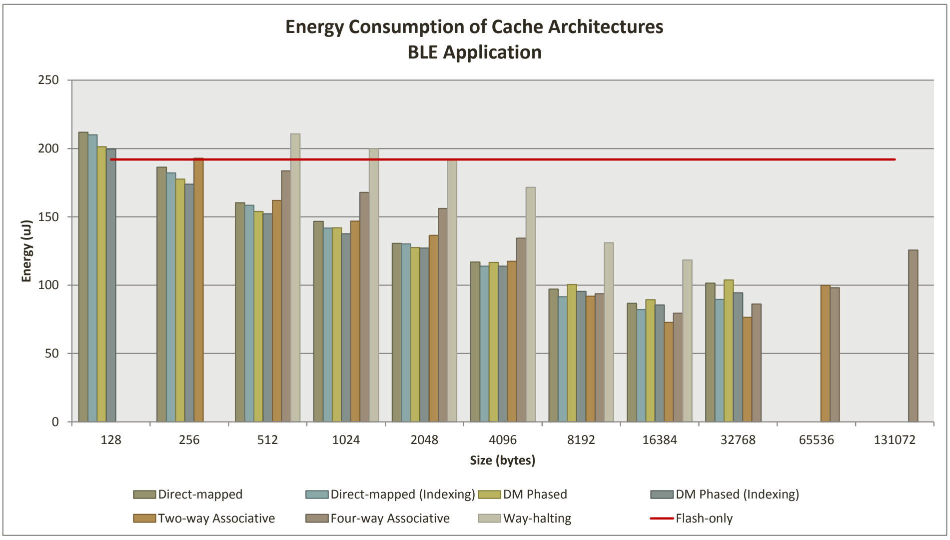

Figure 4.16: Comparison among all cache architectures using SRAM memory.

decreasing. This happens despite the fact the hit rate keeps increasing (and there is a significant increase in the hit rate from 8192 to 16384 bytes for all cache architectures). The explanation for such behavior can be devised after looking at figure 4.17.

As can be seen in this figure, the energy per read and energy per write values remain almost constant up to the size of 4096 bytes, increasing ever so slightly. Nevertheless, after that the growth becomes steeper, having the biggest change from 16384 to 32768 bytes, exactly at the same point in which the energy consumption starts to rise again. Therefore what we see here is the growth in the hit rate being unable to counterbalance the negative effect of the high increase in the energy per read and write values.

However the same phenomenon is not perceived on figure 4.18 which shows the comparison among the cache architectures running the BLE application for the Register File memory. In this case there is a steady decrease in the energy consumption of all cache architectures as their size grows. But it is again the case on figure 4.19, for the 4-way phased cache.

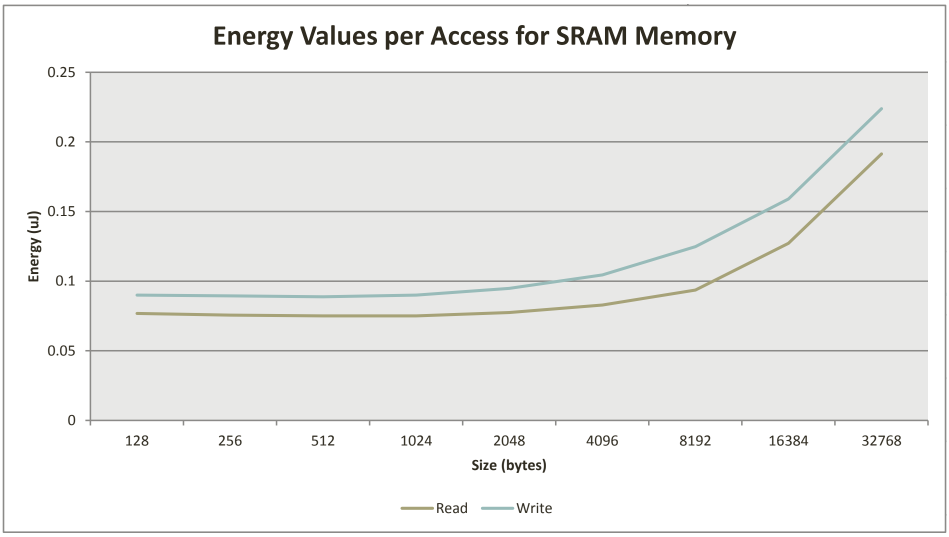

Figure 4.17: Energy values for SRAM memory.

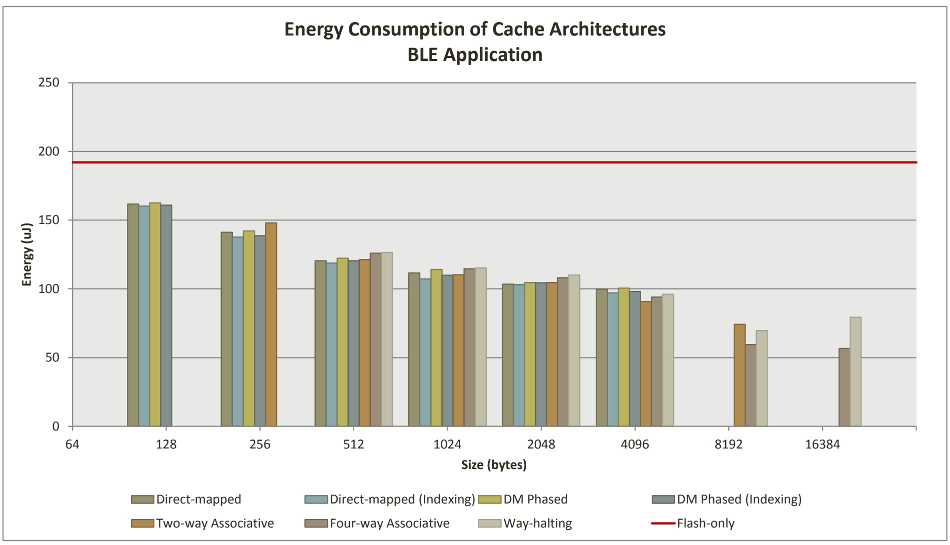

Figure 4.18: Comparison among all cache architectures using Register File memory.

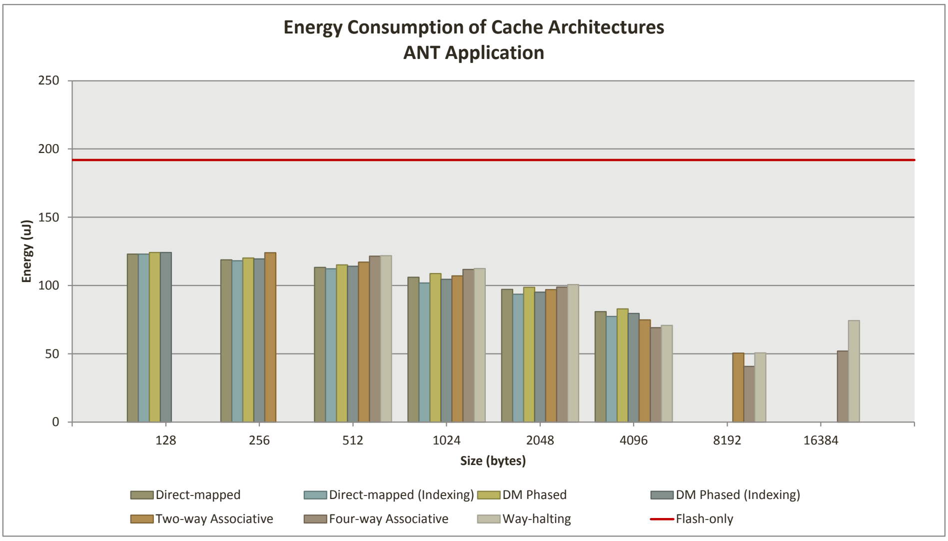

Figure 4.19: Comparison among all cache architectures using Register File memory.

By comparing the cache architectures it is possible to see that the DM caches again outperform the others most of the times. That is not the case only for the caches of size 4096 bytes.

On figure 4.20 the comparisons for the Gazelle application are portrayed. It is interesting to note how much energy savings can be achieved in this case, since this is a small application that fits almost entirely in the cache.

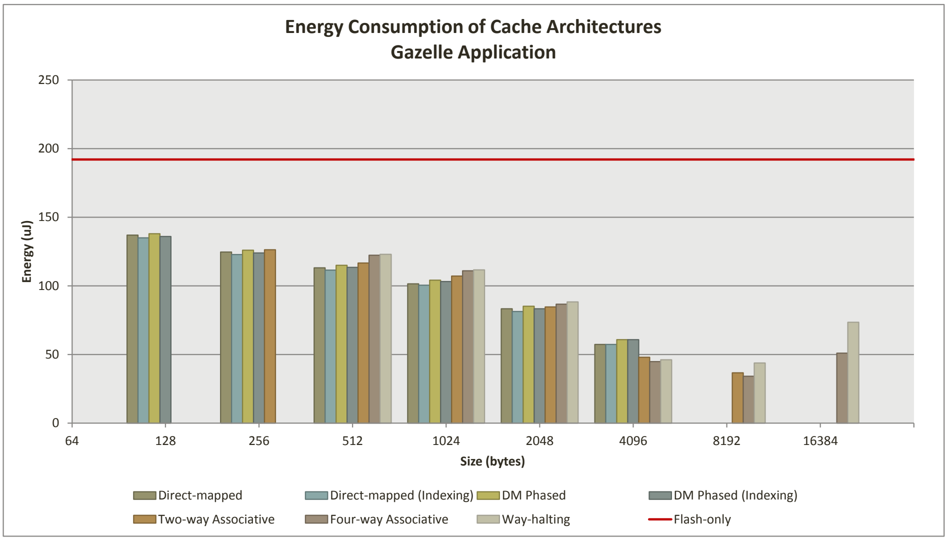

Figure 4.20: Comparison among all cache architectures using Register File memory.

4.2 SystemVerilog Implementation

4.2.1 High Level and SystemVerilog Implementations Compared

The first result presented in this section, shown on figure 4.21 is also one of the main results of this master thesis. It shows the energy consumption of the cache system implemented in SystemVerilog for different cache sizes and different types of memories. As mentioned before in section 3.3.2, these numbers were obtained with PrimeTime PX Synopsys tool. It is important to remember that the tag array in the this implementation is always an asynchronous RAM memory written in Verilog, however, differently from the high level simulations, which uses either a Register File or SRAM memory for the data array, in the SystemVerilog implementation there is also a version that uses a hand-coded synchronous memory for the data array. It is also relevant to mention that the power analysis was performed in the cache system only, therefore there is no information about the energy consumption of the NVM (Flash) memory in the power reports. However, the data presented in the following figures contain the total energy consumption of the memory system (composed of Flash + cache), not only the cache system. The energy consumption of the Flash memory is therefore the same used in the high level simulations.

As expected, we see again the same behavior in terms of energy savings

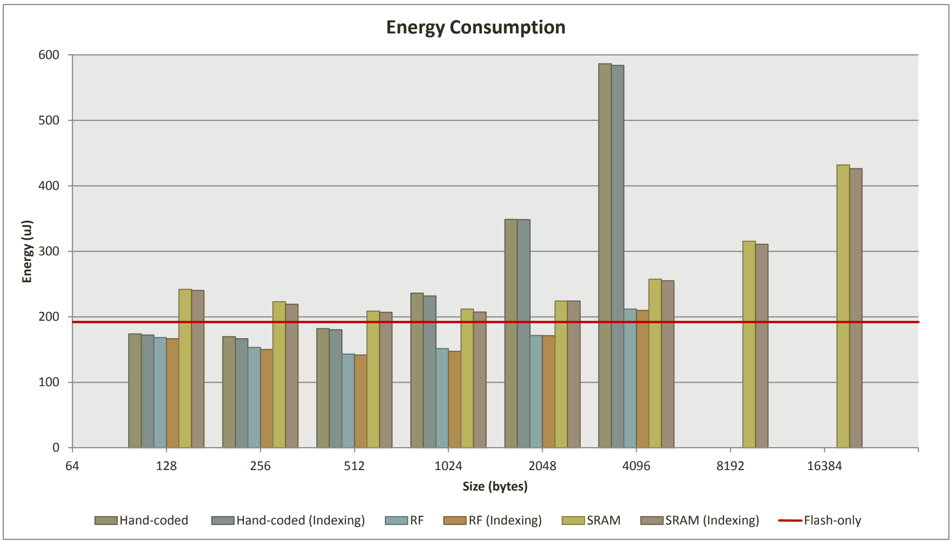

Figure 4.21: Energy consumption of SystemVerilog implementation.

between the regular and the indexing versions of the DM phased caches as observed in the high level simulations. Nonetheless, there is a big difference in terms of absolute values when comparing the values from the high level simulations and the ones from the SystemVerilog simulations. For example, on the high level simulation results for the SRAM memory, it is only at the cache size of 128 bytes that the total energy consumption including a cache system is greater than the total energy consumption without a cache system, however that is the case for all cache sizes in the results obtained with the SystemVerilog implementation. The same big differences can be observed for the Register File memory. In order to understand the reasons behind such discrepancies, we must take a closer at the energy consumption of each separate array (tag and data).

In the figures 4.22, 4.23 we can see the individual energy consumption of the tag and data arrays respectively.

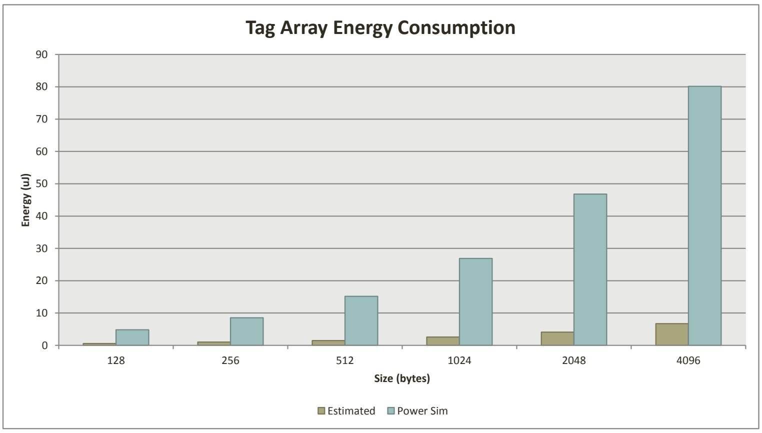

Figure 4.22: Comparison between energy consumption of the tag array.

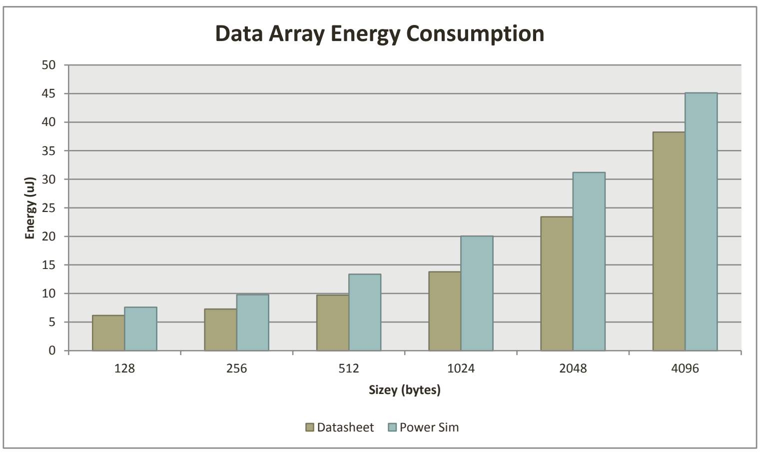

Figure 4.23: Comparison between energy consumption of the data array.

Note that although there are differences in the data array, the biggest discrepancy happens at the tag array. To further analyze the matter, a comparison between the individual energy per access values of the high level and SystemVerilog simulations was made and the results are displayed in the figures 4.24, 4.25, 4.26, 4.27. Before proceeding with the analysis, it is important to make it very clear how the numbers plotted in these figures were obtained.

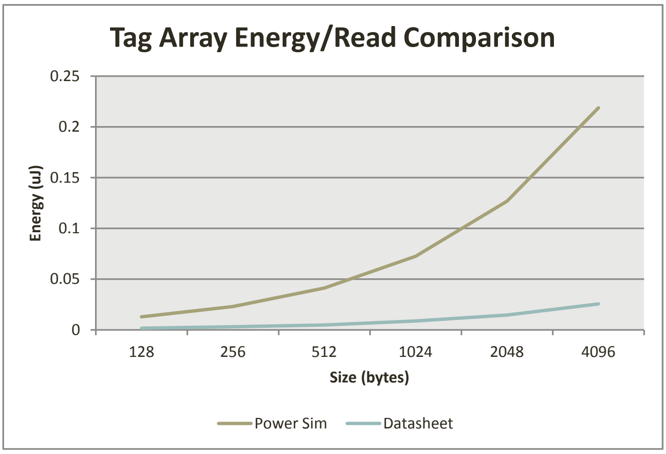

Figure 4.24: Energy per read for tag array.

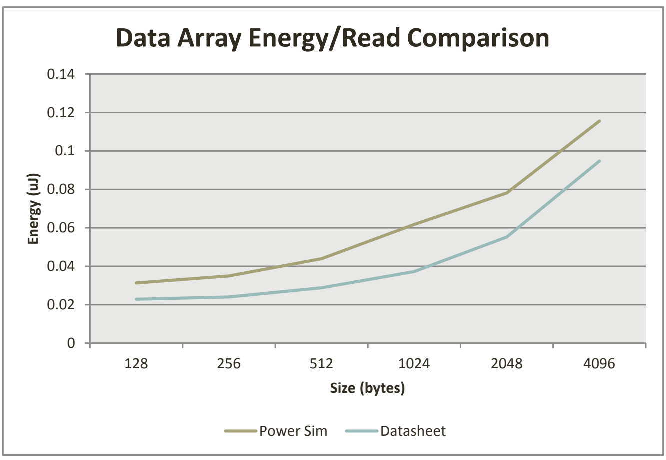

Figure 4.25: Energy per read for data array.

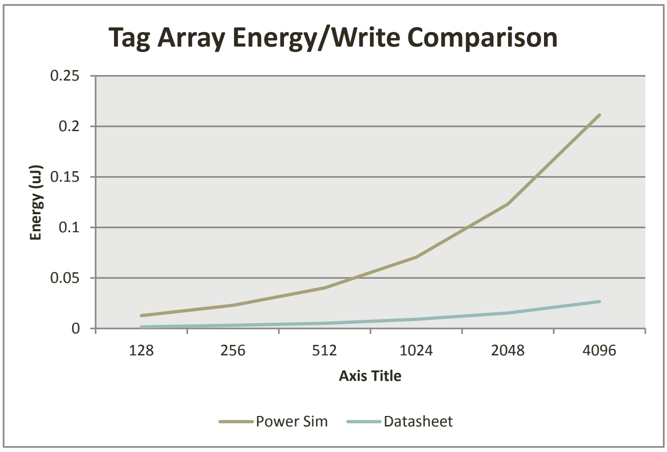

Figure 4.26: Energy per write for tag array.

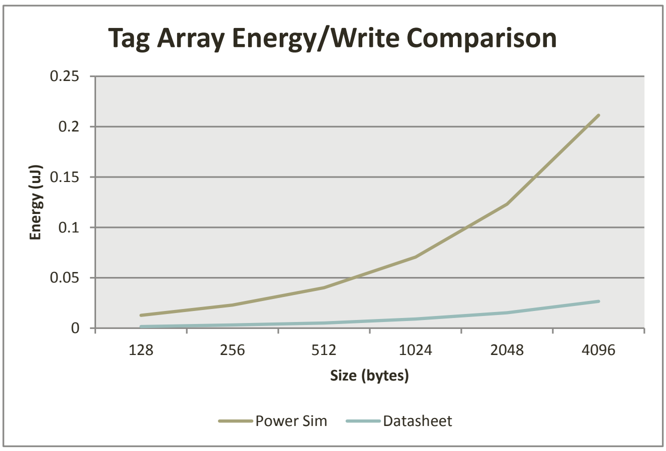

Figure 4.27: Energy per write for data array.

The energy per access values used in the high level simulations for the tag array come from the power estimations previously explained in section 3.2.1.1 and the ones for the data array come from the datasheet (divided by 2). The energy per access values for the SystemVerilog simulations were obtained using the experiments described in section 3.3.2.

Having said that, it now becomes easy to understand why there is such a big gap between their results from the high level simulation and the SystemVerilog simulation. The methodology employed to estimate the energy consumption of the asynchronous memory used in the tag array was flawed and led to unrealistic values. Furthermore, the decision of dividing by 2 the datasheet values for the data array seems to be too optimistic, hence contributing to increase the final difference between these two scenarios.

In possession of this information another set of experiments was performed, which comprised of adjusting the energy per access values of both the tag and the data arrays used in the high level simulation of the DM phased cache and executing the BLE application again. The results are then shown in figure 4.28, in which HC means hand-coded and RF means Register File.

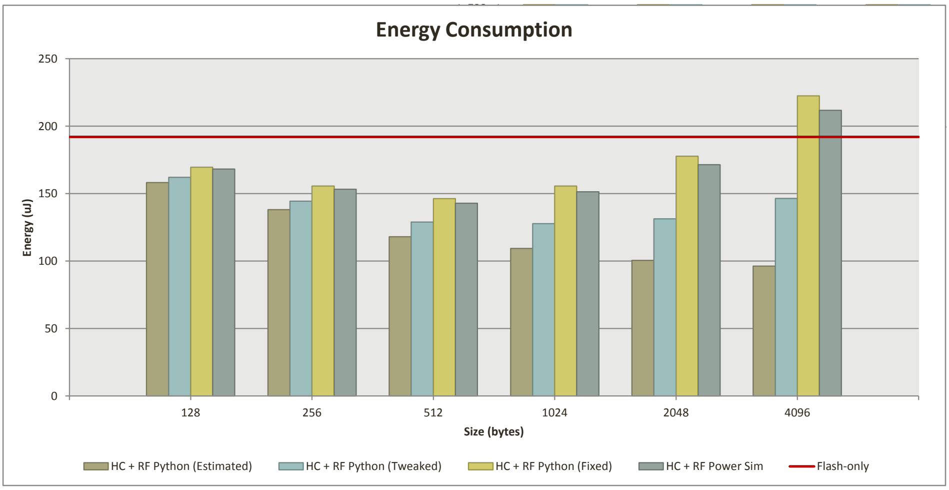

Figure 4.28: Energy consumption with adjusted curves.

As can be seen in this figure, two new curves were generated: one with the word *Tweaked* between parenthesis, in which the energy per access values of the tag array were updated with the ones obtained in the SystemVerilog experiments, and the other with the word *Fixed* between parenthesis, in which not only the tag array uses the values from the SystemVerilog experiments but also the data array uses the datasheet values without dividing them by 2. What can be seen now is that the *Fixed* curve is much closer to the actual values although a little bit higher. A closer look at the data generated for this curve indicates that the energy per read and energy per write values of the tag array obtained as explained in section 3.3.2 also have a slight imprecision.

Since the energy consumption of the asynchronous RAM is actually much higher than the expected, a new scenario was envisioned in which only synchronous memories were employed. Therefore, a simulation in the Python environment using a Register File memory as the tag array was performed, although this time the datasheet values were not divided by 2. The result of that simulation in comparison to the measured values from the power simulation is shown on on figure 4.29. It is clear to see that, in general, the new arrangement offers better prospects in terms of energy consumption.

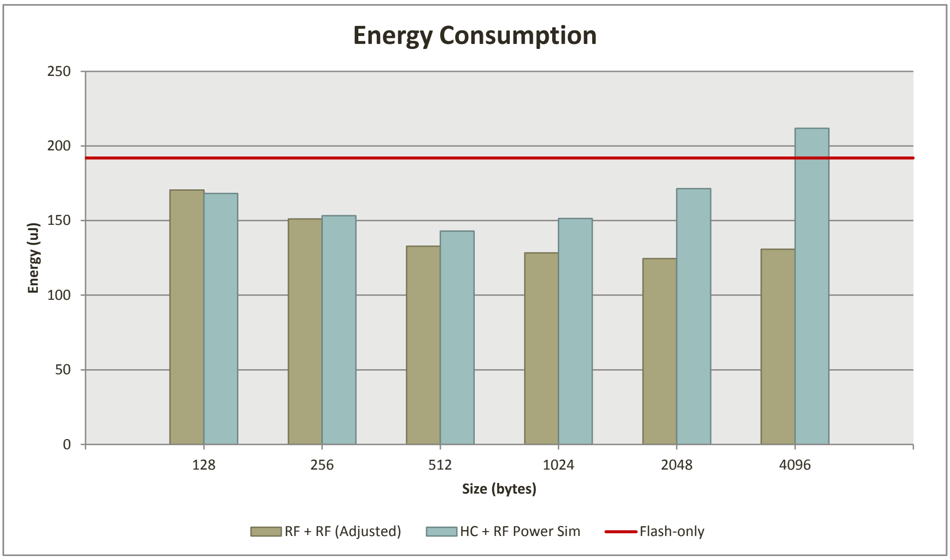

Figure 4.29: Energy consumption with new only-synchronous scenario.

4.2.2 Cache System Module

In this section we take a look at some details regarding the power consumption of the DM phased cache module implemented in SystemVerilog. The first result, shown on figure 4.30, portrays the power breakdown of the main modules that compose the cache system. These modules are:

– Cache Controller Core (named "core" in the figure), which is responsible for coordinating the communication between the microcontroller and the memory elements.

– Tag array (named "tag" in the figure).

– Data array (named "data" in the figure).

– Valid bit array (named "valid" in the figure).

The first interesting fact worth noticing is how little the cache controller core represents in terms of power consumption when compared to the rest of the system. As the memory size grows it becomes basically irrelevant. On the other hand, as could be foreseen after reading the previous section, the tag array is responsible for a great part of the system's power consumption,

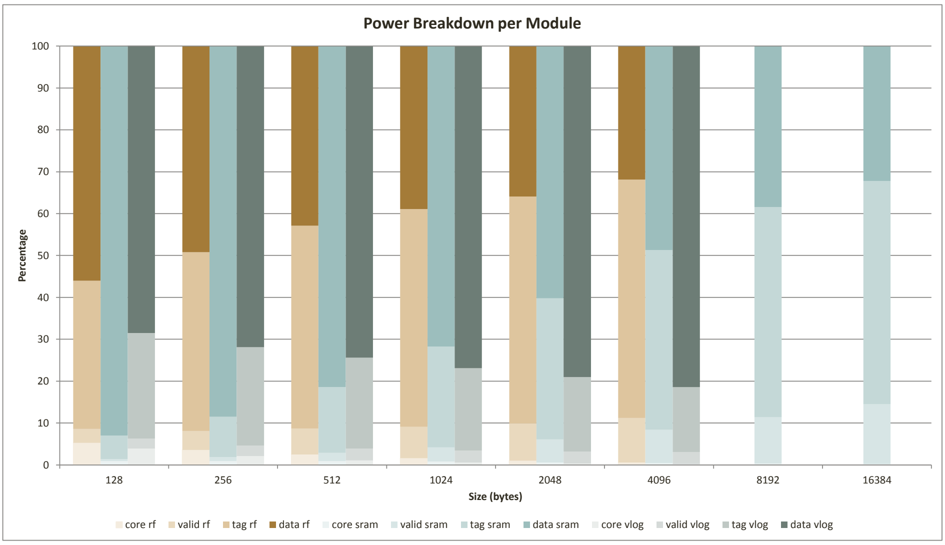

Figure 4.30: Power breakdown of cache system module.

becoming the most power-hungry element on the Register File version after the size of 256 bytes and on the SRAM version after the size of 4096 bytes, corroborating with the findings of the previous section. Interestingly enough, the valid bit array also starts to take up a considerable amount of the total power consumption as its size grows. That is expected since the valid bit array is implemented also as a one-bit register file, which, as the synchronous memory implemented in Verilog, does not scale very well.

In figures 4.31, 4.32 and 4.33 we see the power histogram of the cache system for the three different memory types used as the data array (hand-coded Verilog, Register File and SRAM). The power consumption was broke down into switching power, internal power and leakage. As can be seen in all three figures, the main element is the internal power, which, as explained in [25], "is caused by the charging of internal loads as well as by the short-circuit current between the N and P transistor of a gate when both are on". Being this system synthesized to a 180 nm technology, it was expected that leakage would not represent a major issue.

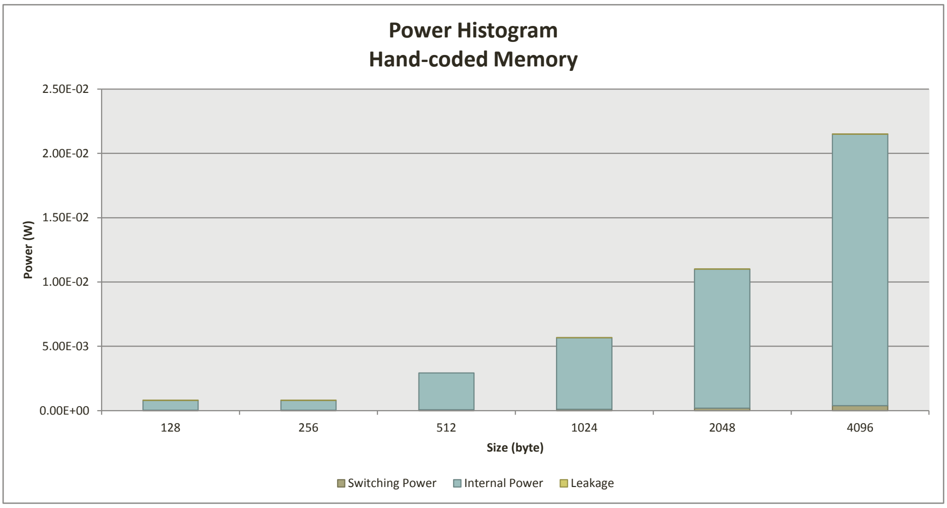

Figure 4.31: Power histogram for hand-coded data array.

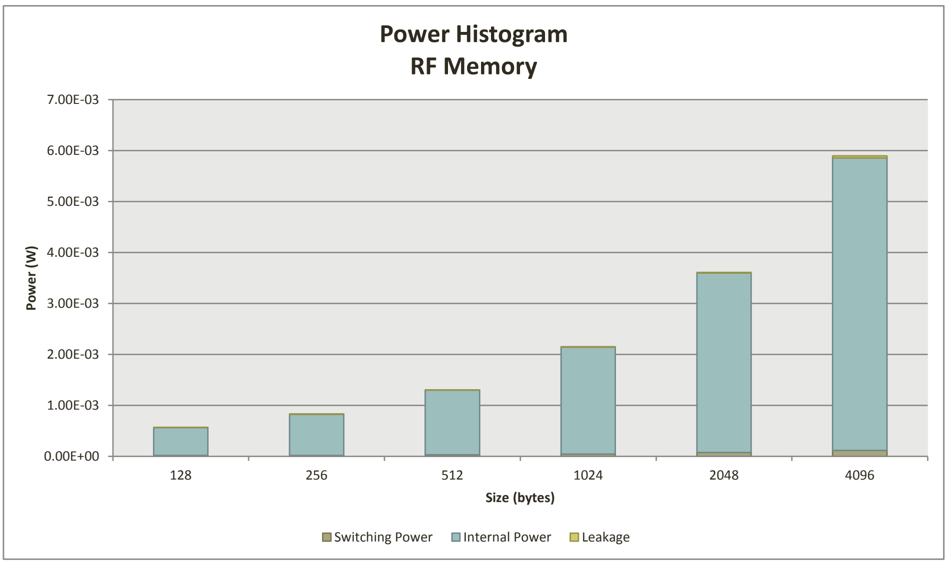

Figure 4.32: Power histogram for Register File data array.

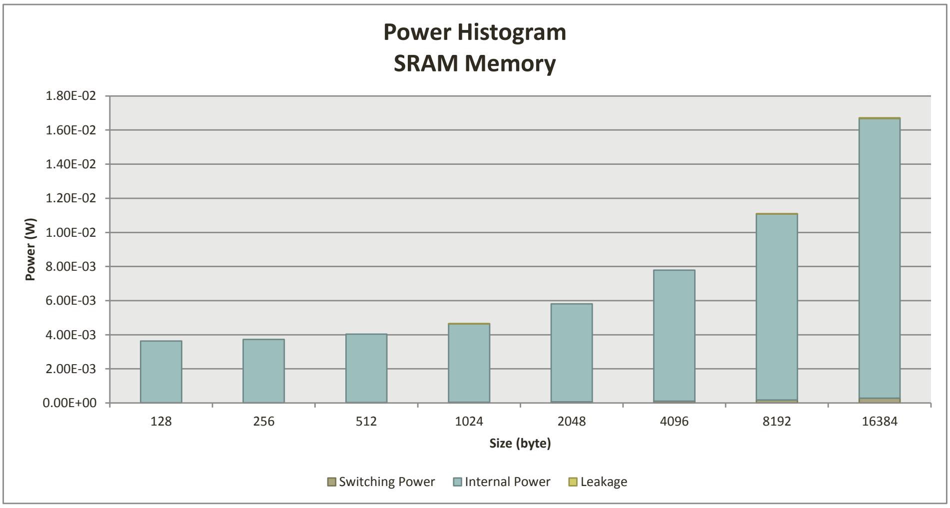

Figure 4.33: Power histogram for SRAM data array.

Chapter 5

Discussion

In this chapter a summary of the main results described in the previous sections are presented, along with an analysis of the key issues of this project.

5.1 Indexing solution

In this project, the indexing solution explained in [2] was further explored. Although the number of applications used (three) was still small in comparison to well-known benchmark suites, it still provided a good insight into this technique capabilities. For example, as presented in section 4.1.1.1, for one specific application and cache size an improvement of over 60% in the hit rate was observed. Therefore, if application profiling is available, applying this solution might result in huge improvements in terms of energy savings, at a really low cost, since the overhead in terms of hardware in the direct-mapped cache controller is minimum.

5.2 Cache Architectures

5.2.1 Energy Consumption

This project presented a broader investigation in terms of cache architectures and their energy efficiency. Not only other five cache solutions were added to the previous project, but also the influence of the type of memory employed was evaluated. And, as can be seen from the results presented in section 4, there is not a single answer as to which option is the best. The solution to this relies in the design space exploration of these implementation, having basically hit rate, energy per access values and area (which was not regarded in this project) as the variables.

On a more specific note, it is worth mentioning the performance of the 2-way and 4-way phased caches, which were included in order to evaluate how the phased cache concept would affect their performances. As seen in section 4.1.2.4, no remarkable gains can be achieved by using such solutions.

5.3 Phased Caches

Phased cache is a central element of this work and it is a concept that fits neatly with the pipelined access to the memory. Nevertheless, as the results proved, the solution did not render the improvements that were expected of it. The main reason for that is the overhead that is present in both separate arrays. That problem exists basically because two separate, standard memory elements were used for each array. If instead a specifically designed memory that combines both tag and data arrays was used, a different result might had been obtained.

5.4 Way-halting

The way-halting is a very interesting cache architecture who aims at taking advantage of the associativity of an n-way cache, while offering energy savings by preventing some of the ways being read in every access. However, the particular implementation used in this project can not benefit from it, since to work properly the principle of locality has to be better explored. The requirement of only one word per cache block imposed by the microcontroller system is the culprit in this case. In hindsight, perhaps a more careful analysis of the architecture prior to the high level implementation could have prevented further exploration. Unfortunately that was not the case, but at least the findings were obtained early enough to avoid additional developments.

5.5 SystemVerilog Implementation

The high level simulations offer a quicker and simpler way to evaluate a cache architecture, in comparison to the SystemVerilog implementation. That is usually the case as one moves along through different levels of abstraction. Nevertheless, the results from the SystemVerilog simulations are far more trustworthy and actually challenge the findings acquired with the high level simulations. All high level simulations were performed using the advice of dividing by 2 the power consumption values found in the datasheet provided by the Artisan tool.

In the figure 5.1 we can see the percentage of increase in energy consumption that would result if the datasheet value was not divided by 2. Obviously this percentage increases as the cache size grows, since it also increases the role of the cache in the total amount of energy consumption of the memory system.

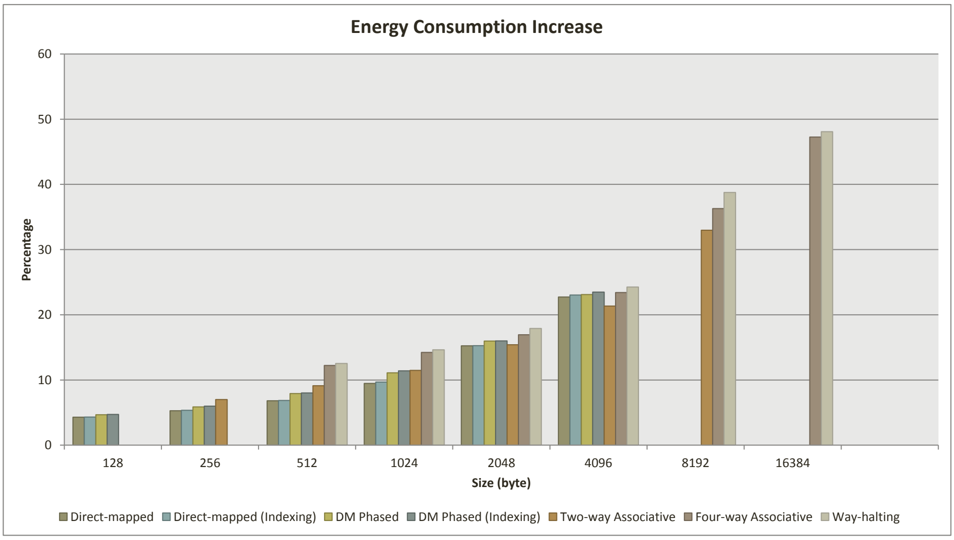

Figure 5.1: Increase in energy consumption for datasheet vs datasheet/2 scenarios.

Although the final results from the SystemVerilog simulations are reliable, the methodology utilized during the design of the cache system was not without shortcomings. The biggest problem, fully analyzed in section 4.2.1, is related to how the estimation of power consumption of asynchronous memories was carried out. In retrospect, it would have been more interesting to spend more time obtaining accurate power consumption values for this type of memory. That would have changed the course of the design of the cache controller, since probably a synchronous RAM would have been used instead. Nonetheless, making this change in the SystemVerilog code is not a very complex task and can be a possible future development of this work.

Finally, it is important to highlight the impact that the logic to implement the cache controller incurs in the total amount of energy consumed by the cache system, which is really low. That means that by using a simple hardware and the correct types of memories, there is room for considerable energy savings.

Chapter 6

Conclusions and future work

This project, as described in section 1.2, was decomposed in different tasks. An account of each of them is provided in the following paragraphs.

Task 1: Evaluate in a high level simulation environment a new cache architecture.

A version of the way-halting cache architecture, presented in [31], was implemented in the high level simulation environment and experiments were performed and thoroughly examined throughout the course of this project. And as can be observed by the results presented previously, this architecture is not suitable for the targeted microcontroller system due to the tight restriction in the number of words pertaining to each cache line (only one), which in turn is related to the connection bandwidth between the Flash memory and the microcontroller. If this restriction is ever lifted, then the way-halting architecture might become a good candidate (and if that happens, the high level simulation environment is ready to perform the evaluations).

Task 2: Evaluate the direct-mapped (DM) with indexing cache architecture against a set of applications.

Two other applications, the ANT and Gazelle sample programs, were added to the Bluetooth Low Energy software used in [2], and again all necessary analyses were carried out and discussed. And, although three is still a small number for a benchmark suite, adding these two applications highlighted the capabilities of the indexing technique in terms of improved hit rate (or miss reduction), leading to the conclusion that this is indeed an attractive solution (however only when profiling an application is a possibility).

Task 3: Implement the direct-mapped cache with indexing in SystemVerilog.

The task that certainly took up most of the time dedicated to this master thesis was fully accomplished as well. All the details needed to explain the results achieved were presented, however, given the fact that the design of the cache module itself was not the main goal of this work and that it in fact does feature any great novelty (in the end, it is a cache controller for a direct-mapped cache, with a configurable mapping), not all the specifics of the final hardware were presented. Furthermore, no emphasis was placed on the design flow and all steps that it entails.

This task helped answer in a more precise way the question of whether adding a cache system to a microcontroller system can improve its energy efficiency. As the numbers presented before suggests, the answer is yes. This observation combined with the results obtained both in this work and previously by [5] in terms of hit rates of direct-mapped caches can give confidence that this is the correct answer.

Moreover, the measurements performed in the scope of this project can be used as a base to further exploration, even in the high level simulations, as briefly hinted on section 5.5.

Task 4: Evaluate the direct-mapped cache with different types of SRAMs.

Another deficiency of [2] was overcome by this project when the effects of choosing a different type of memory to implement the cache system were analyzed. However, finding the global minimum for all scenarios was not possible due to complications regarding the maximum size for a single memory element described in section 3.3.1.1.

The answer to the question of whether the type of memory plays an important role in the cache system is: yes. But the decision regarding which type of memory should be used depends mainly on the size of the cache that one wants to add to the system.

Task 5: Implement and evaluate the direct-mapped cache as a phased cache.

A central topic of this master thesis, the accomplishment of this task is completely depicted in the course of this text. And although the results in terms of energy consumption of this technique were not satisfactory, the rationale behind such findings were fully exploited, being the main problem the overhead created by splitting the memory into two different arrays. Nevertheless, it is safe to conclude that, had a more energy efficient implementation of the tag and data split arrays been used, this solution would certainly outperform the regular direct-mapped cache.

Task 6: Implement and evaluate the 2-way and 4-way as a phased cache.

This task was also finished in its entirety and as in task 5 the final results were not appeasing. And that is even more interesting given the fact that area was not even considered in this work (and 2-way and 4-way caches have a larger area footprint in comparison to direct-mapped caches). Therefore, when left to choose among these three setups: direct-mapped, 2-way and 4-way, direct-mapped seems to be the better choice.

6.1 Future Work

This project finishes a complete cycle in the evaluation of a specific cache architecture (namely the DM cache with indexing), targeting a microcontroller system. The results presented here can be used to guide the decision of whether to include a cache system in the desired System-on-chip or not. However, yet a more relevant outcome would be achieved by integrating the designed cache system into an existent System-on-chip and perform experiments running real applications in such system and gathering the results.

There is always room for evaluation of new cache architectures, which now can benefit not only of the high level simulation environment, but also the environment built around the cache controller implemented in SystemVerilog, offering the possibility of a fast track analysis of the implementation in a lower level of abstraction. A good candidate would be Victim caches, first introduced in [10]. It is a simple concept which fits the requirements of the microcontroller targeted in this project, with an small area footprint.

Since area was just mentioned, another dimension on the design space exploration could be added by analysing the impact in terms of area of each cache solution presented in this work.

In terms of the phased cache solution, it would be very interesting to check its perform with a memory specifically design for this purpose, instead of using two different memories for the tag and data arrays.

Furthermore, the other solution presented in [2], namely the Scratchpad memory, could also be put to test in real applications by integrating it to a System-on-chip. As pointed out in [2], this is a simpler task in comparison to integrating a cache to a System-on-chip because it mainly relies on software techniques.

Bibliography

[1] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pages 259–267, New York, NY, USA, 2004. ACM.

[2] Vinicius Almeida Carlos. Tfe 4520 semester project report. [Report delivered in the scope of the course TFE4520; included in the appendix.], 2012.

[3] A. Efthymiou and J.D. Garside. A cam with mixed serial-parallel comparison for use in low energy caches. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(3):325 –329, march 2004.

[4] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 148 –157, 2002.

[5] Stian Fredrikstad. Saving energy in periodic embedded systems with memory system techniques. Master's thesis, Norwegian University of Science and Technology, June 2012.

[6] Tony Givargis. Zero cost indexing for improved processor cache performance. *ACM Trans. Des. Autom. Electron. Syst.*, 11(1):3–25, January 2006.

[7] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. Tiny instruction caches for low power embedded systems. *ACM Trans. Embed. Comput. Syst.*, 2(4):449–481, November 2003.

[8] Koji Inoue, Vasily Moshnyaga, and Kazuaki Murakami. Dynamic tag-check omission: A low power instruction cache architecture exploiting execution footprints. In Babak Falsafi and T.N. Vijaykumar, editors,

Power-Aware Computer Systems, volume 2325 of *Lecture Notes in Computer Science*, pages 18–32. Springer Berlin Heidelberg, 2003.

[9] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[10] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373, 1990.

[11] Nam Sung Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches. leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 219 – 230, 2002.

[12] Nam Sung Kim, K. Flautner, D. Blaauw, and T. Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(2):167 –184, feb. 2004.

[13] Soontae Kim, N. Vijaykrishnan, Mahmut Kandemir, Anand Sivasubramaniam, and Mary Jane Irwin. Partitioned instruction cache architecture for energy efficiency. *ACM Trans. Embed. Comput. Syst.*, 2(2):163–185, May 2003.

[14] HP Labs. CACTI. `http://www.hpl.hp.com/research/cacti/`, 2012. [Online; accessed 13-November-2012].

[15] Xin Lu and Yuzhuo Fu. Reducing leakage power in instruction cache using wdc for embedded processors. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 1292–1295, New York, NY, USA, 2005. ACM.

[16] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[17] Ramesh Panwar and David Rennels. Reducing the frequency of tag compares for low power i-cache design. In *Proceedings of the 1995 international symposium on Low power design*, ISLPED '95, pages 57–62, New York, NY, USA, 1995. ACM.

[18] Jongsoo Park, James Balfour, and William James Dally. Fine-grain dynamic instruction placement for l0 scratch-pad memory.

In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, pages 137–146, New York, NY, USA, 2010. ACM.

[19] K. Patel, E. Macii, L. Benini, and M. Poncino. Reducing cache misses by application-specific re-configurable indexing. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 125 – 130, nov. 2004.

[20] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[21] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 54–65, Washington, DC, USA, 2001. IEEE Computer Society.

[22] Jan Rabaey. *Low Power Design Essentials*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[23] Marisha Rawlins and Ann Gordon-Ross. Lightweight runtime control flow analysis for adaptive loop caching. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, GLSVLSI '10, pages 239–244, New York, NY, USA, 2010. ACM.

[24] Prof. Dr.-Ing. Dominik Stoffel. Architecture of Digital Systems 2 - Lecture material, 2011.

[25] Inc Synopsys. Expanding the Synopsys PrimeTime Solution with Power Analysis. `http://www.synopsys.com/Tools/Implementation/SignOff/CapsuleModule/ptpx_wp.pdf`, 2013. [Online; accessed 07-June-2013].

[26] Guido van Rossum. Python Official Website. `http://python.org/`, 2012. [Online; accessed 13-November-2012].

[27] H. Vandierendonck and K. De Bosschere. Xor-based hash functions. *Computers, IEEE Transactions on*, 54(7):800 – 812, july 2005.

[28] D.P. Volpato, A.K.I. Mendonca, L.C.V. dos Santos, and J.L. GuÌ and-ntzel. A post-compiling approach that exploits code granularity in scratchpads to improve energy efficiency. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 127 –132, july 2010.

[29] Chuanjun Zhang. An efficient direct mapped instruction cache for application-specific embedded systems. In *Proceedings of the*

3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '05, pages 45–50, New York, NY, USA, 2005. ACM.

[30] Chuanjun Zhang. A low power highly associative cache for embedded systems. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 31 –36, oct. 2006.

[31] Chuanjun Zhang, Frank Vahid, Jun Yang, and Walid Najjar. A way-halting cache for low-energy high-performance systems. *ACM Trans. Archit. Code Optim.*, 2(1):34–54, March 2005.

[32] M. Zhang and K. Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop, MICRO*, volume 33, 2000.

Appendix A

Description of the digital attachments

The practical work performed in the scope of this project in the form of source code, the original versions of the figures and the last semester report ([2]), referenced many times throughout this text, are provided in the file attached to this report.

A.1 Bit-selection (indexing) optimal algorithm

C++ implementation that outputs the optimal indexing to be used for a given trace file and cache size.

A.2 High Level simulation environment

High level simulation developed in Python in which the experiments were carried.

A.3 SystemVerilog implementations

Includes code for the cache controller and memory elements.

A.4 Graphs

Contains high resolution version of the figures presented in Results section of the master thesis.

A.5 TFE 4520 - Semester Project Report

The last semester report ([2]), referenced throughout this master thesis.