Imperial College London

Final year project

Customizable Security-Aware Cache for FPGA-based Soft Processors

---

Author:
Maciej Kurek

Supervisor:
Prof.Wayne Luk

June 15, 2010

Abstract

This report describes our attempt to implement security-aware cache targeting *field-programmable gate array* (FPGA) technology which can be customized to meet various requirements. Our design, based on an architecture resilient to side channel cache timing attacks, involves a range of new address decoders to support efficient cache index re-mapping. Various implementations of our security-aware cache have been developed for the Leon 3 processor, and their performance and resource usage are evaluated.

The main contributions are:

- *A customizable security-aware cache optimized for FPGA technology*, based on a range of new address decoders that utilize remapping register file for index remapping (Chapter 3).
- *Implementation of our security-aware cache on Xilinx FPGAs*, with interface to the Leon-3 soft-processor (Chapter 4).
- *Evaluation of our approach*, showing the resource usage and performance for various implementations (Chapter 5).

We also prepared a guide on how to use security-aware cache (Appendix A).

Acknowledgements

First and foremost my thanks must go to Prof. Wayne Luk. For the countless hours spent on reading my drafts, guiding me through the research process as well as giving me an invaluable lesson on how to clearly present my thoughts.

I would also like to thank Ioannis Ilkos and other friends who helped me by reading countless drafts of my report. I would have not been able to finish this report without their numerous comments and suggestions.

And at last I owe special gratitude to my family, for teaching me not to avoid challenges. Only by facing them, victorious or not, we develop.

Contents

1 Introduction **4**

2 Background **8**
2.1 FPGA Design 8
2.2 Efficient and portable FPGA design 9
2.2.1 CAM memories and FPGAs 10
2.3 Introduction to Soft processors 11
2.3.1 Difference between hard and soft processors 12
2.3.2 Soft processors automatic generation and optimization . . 14
2.3.3 How to quickly and efficiently generate soft processors? . 14
2.3.4 How to optimize the soft processor? 14
2.3.5 Soft processors caches 14
2.4 Brief overview of Side channel cache attacks 16
2.5 Cache Side Channel Timing Attack 17
2.6 Countermeasures for Cache based Attacks 17
2.7 Security-aware Cache Architecture Overview 18
2.7.1 Security-aware cache address decoder 19
2.7.2 Dynamic cache line remapping 19
2.7.3 Line context field 20
2.7.4 Address decoder 20
2.7.5 SecRAND overview 21
2.8 Summary 23

3 Efficient Design of Security-aware Cache on FPGA **24**
3.1 Security-aware cache 26
3.1.1 Increased critical path 27
3.1.2 Using multi-cycle CAM memory 27
3.2 Pipelined security-aware cache index remapping 30
3.2.1 Pipelined security-aware cache based on M-cycle write CAM 32
3.2.2 CAM FIFO buffer architecture 32
3.3 Optimized CAM Decoder, the L-Associative security-aware cache. 37
3.4 Modified Security Algorithm 41
3.5 Design Portability 43
3.6 Summary 43

4 Cache implementation and Leon 3 **44**
4.1 Design choices 44
4.1.1 Why Leon 44
4.1.2 Random number generator 45
4.2 How we modified Leon 46
4.2.1 Stage 1. Identifying the performance crucial components . 46
4.2.2 Stage 2. L-associative security-aware cache 50
4.2.3 Stage 3. Extending Leon 3 with pipelined security-aware cache 54
4.2.4 Stage 4. Enabling Full SecRAND 55
4.3 Summary 58

5 Functional and Performance Evaluation **59**
5.1 Performance Cache Hit Rate 59
5.1.1 Ordinary environment 59
5.1.2 Highly-secure environment 60
5.1.3 Simulation 60
5.2 Performance - Cache decoder delay and resource usage 61
5.2.1 LUT 66
5.2.2 Flip-flops 66
5.2.3 Frequency 66
5.2.4 Block-size 66
5.3 Comparison of pipelined security aware cache with fully associative pipelined cache 68
5.4 Security 68
5.5 Summary 70

6 Future work and conclusions **71**

A Guide: How to use Security-aware cache? **75**

B Leon 3 Configuration Files **80**

C Optimized security-aware cache decoder **87**

Chapter 1

Introduction

This report describes a security-aware cache targeting field-programmable gate array (FPGA) technology which can be customized to meet various requirements. The design, based on an architecture resilient to side channel cache timing attacks, involves a range of new address decoders to support efficient cache index re-mapping. Various implementations of our security-aware cache have been developed for the Leon 3 processor, and their performance and resource usage are evaluated.

The main contributions are:

- *A customizable security-aware cache optimized for FPGA technology*, based on a range of new address decoders that utilize remapping register file for index remapping (Chapter 3).
- *Implementation of our security-aware cache on Xilinx FPGAs*, with interface to the Leon-3 soft-processor (Chapter 4).
- *Evaluation of our approach*, showing the resource usage and performance for various implementations (Chapter 5).

Motivation

Modern processors and their components suffer from many physical security flaws. The flaws vary in the potential risk that they pose and the difficulty in which they can be exploited by the attacker. Side channel cache attacks are difficult to exploit, but can lead to potentially serious information leakage such as revealing cryptographic keys. People have been aware of cache side-channel attacks for a while. A number of methods preventing these attacks have been proposed, like disabling cache, constant timing programming or part cache partitioning [9]. The problem with the above methods of attack prevention is that they often degrade speed and power efficiency. A new way of countering the problem was proposed by Wang and Lee without the drawback of performance degradation [23]. Their cache architecture is not only security-aware, but also has a performance advantage over the traditional cache architecture in the form of increased hit rate.

However, the most novel aspect of the security-aware cache, the new address decoder, is the only part of the design that cannot not be directly ported onto

FPGA. It has been designed using a *transistor-level description*, targeting implementation in *application-specific integrated circuit* (ASIC) technology. This means that a trivial *look-up-tables* (LUT) and flip flops decoders implementation is impossible without the loss of the benefits it poses over traditional decoders. That makes the unmodified security-aware cache in it present form an unsuitable alternative to the ordinary cache for most of the soft processors and many other devices that are based on the field programmable hardware. According to Wang and Lee the security-aware cache is an improvement over traditional cache in virtually every aspect; at the cost of slight area increase [23], as such it seemed reasonable to investigate the possibility of porting the architecture onto *block ram* (BRAM) and LUT based FPGA devices.

Challenges

At the birth of the idea of porting the security-aware cache onto FPGAs we were aware that an resource overhead was going to be present, and that it would be noticeable. The main reason behind this is the fact that the security-aware cache introduces extra logic compared to an ordinary cache. Although this is also the case with the ASIC security-aware cache, the extent of the overhead has to be more significant on FPGA. FPGAs building fabric is *specialized and coarse-grained* in nature opposed to the *fine-grained and generic nature* of the ASIC.

An example relevant to our problem would be the implementation of a tag array with a non trivial address decoder on FPGA. It is very straightforward and consists of a simple BRAM instantiation. The problem is, that the BRAM file has a built in address decoder that cannot be modified to implement a different decoding scheme, which security-aware cache is using. One solution would be to wrap the BRAM file with extra logic to implement a tag array with a more sophisticated decoding scheme, but this can clearly cause redundancy. Although we do not need two decoders, we have to use both of them. In ASIC we can easily deal with this, as everything is implemented using transistors, while on FPGA we can only minimize it.

Furthermore a number of different problems raised while working on cache implementation. The solutions we came up with although very often simple in concept were far more difficult to implement than expected. Not due to complexity of the design, but due to the long hardware design-debug cycle. The challenges that appeared during the design process:

- High resource demand.
- Increased critical path.

The Modified Security-Aware Cache

We present a number of possible modifications of the security-aware cache architecture that make it suitable for the FPGA based devices, two of which are solutions to the challenges mentioned before. The different ports of the security-aware cache architecture are dependent on the target platform as well as the goal performance levels (discussed in detail Chapter 5). There are three different aspects of the design.

- Security-aware cache base on unregistered CAM memory.
- L-associative security-aware cache. Answer to the high resource demand.
- Pipelined security-aware cache. Answer to increased critical path.

The security-aware cache address decoder is based on generic or vendor provided and unregistered single cycle write *content addressable address memory* (CAM). The memory is part of the cache control logic therefore read/hit bits are generated in one cycle and thus it is referred as a zero cycle. The cache performance is highly dependent on the CAM memory used, the most generic version being characterized by portability and ease of modification of existing designs at the expense of high resource cost and increased critical path. The high resource cost is especially prominent if a generic flip flop/LUT based CAM is used.

Cache based solely on the first design with generic CAM is characterized by high resource usage, as such the next logical step was an attempt to lower it. The solution to the first problem is based on the idea of associative cache architecture, but applied to the address decoder. It uses fraction of the resources used by the ordinary (FPGA) security-aware cache, and can be implemented using same CAM memories as the latter. It suffers from the problem of increased critical path as the previous design. A number of problems appeared with implementation of this type of cache. We came up with two different methods of implementing the cache. One efficient but more difficult to implement based on BRAM memory and one based on flip flops.

Although we have found a solution to the resource problem, we still faced the problem of increased critical path. To counter the increased critical path problem, *pipelined design* was approached. The idea can be applied both to the associative and ordinary security-aware cache, depending on the users performance and resource utilization goals. It has the advantage of offering shorter critical path, and therefore higher design operating frequency, at the cost of more complicated state machine as well as decreased portability. The decreased portability is a result of cache being more difficult to implement and having different behaviour than traditional FPGA caches. A similar design has been approached (independently) by Yiannacouras and Rose in the context of *fully associative cache* (FA) [24].

Potential Benefactors

Our modified caches are not solutions to all the problems that modern soft processors designers face. We offer an alternative to fully associative caches, and we offer an alternative solution to current hardware cache timing attack prevention mechanisms.

Most soft processors have a very simple memory hierarchy and therefore are far more vulnerable to the cache timing attacks then complicated x86 or Power cores. The soft processors are also becoming more commonly used, especially in the cryptographic and networking sector. New systems based on massively simple parallel cores are being developed, also based on simple memory hierarchy. There are a number of DSL routers that use simple soft processor cores. The problem of cache timing attack can become a serious issue. Currently the

attacks were only successfully employed as part of research projects, but we believe that it is a question of a year or two when they will become a real threat. It is even possible that they have already been used, but they not been traced...

Contributions

Summary of contributions:

- *A customizable security-aware cache optimized for FPGA technology*, based on a range of new address decoders that utilize remapping register file for index remapping (Chapter 3). We identified a number of potential problems with security-aware cache implementation and we propose different techniques how to counter them. We base our approach on abstracted CAM properties.
- *Implementation of our security-aware cache on Xilinx FPGAs*, with interface to the Leon-3 soft-processor (Chapter 4). Although we did not manage to implement all of the desired features we elaborate and explain on how we could do it (and plan to do it in future).
- *Evaluation of our approach*, showing the resource usage and performance for various implementations (Chapter 5). We evaluate different techniques proposed in Chapter 3.
- *Guide: How to use Security-aware cache,* we present a guide on how to use security-aware cache on FPGA chips. We identified a number of potential problems with security-aware cache while working on Leon 3 and we believe that this guide can be of big help to any potential user (Appendix A).

Chapter 2

Background

This chapter contains basic background information on which this thesis was based. Sections 2.1 - 2.3 contain basic information on soft processors and FPGAs. Sections 2.4 - 2.6 briefly describe different types of side channel attacks and present in detail the principles of cache side channel timing attacks. Section 2.7 is the overview of security aware cache.

2.1 FPGA Design

In the 1960's-1970's software technology started to rapidly change, with the development of first general-purpose computer programming languages like C or Fortran and their associate tools. Thanks to the level of abstraction offered by the new tools, programs became more complicated, consisting of sequences of instructions incomprehensible for the human mind both, due to their ever-increasing size and complexity; software development became relatively independent of hardware. The process reached its pinnacle with the development of JAVA in the 1990's. The new language made the underlying computer hardware so abstract to the programmers that the knowledge of computer architecture became obsolete for majority of users. At the same time, another technological revolution was happening, the development of reconfigurable hardware and logic synthesizers.

The idea of reconfigurable hardware dates back to the very pioneers of computer engineering. Nevertheless it was not till the 1990's when first FPGAs and *complex programmable logic device* (CPLD) devices appeared. At the very beginning they could not compete with ASIC technology, due to both performance and cost issues. However with the development of supporting software and better hardware at the beginning of the 21st century, FPGA became commercially feasible. They offered several advantages over ASIC based technologies (listed in table Table 2.1).

Table 2.1: Design Flow Comparison

ASIC	Reconfigurable Hardware
Full custom compatibility	Short time to market
Lower unit costs	No upfront *NRE* (non recurring expenses)
Smaller form factor	Simpler design cycle
Higher clock cycle	Simpler design cycle
	More predictable project cycle
	Programmability

One of the main reasons behind the success of reconfigurable hardware was the development of logic synthesis tools. Along with several other hardware synthesis tools, they require substantial computing power not available till the late 1990's. Without them, hardware design was extremely complicated and time consuming due to the design mapping process. In exactly the same way as a program can be written in a language like C and ran across a range of hardware platforms we can design a circuit and implement it via logic synthesis on a number of different semiconductor devices. The software code is analogical to hardware description code, logic synthesis and then placing and routing a design onto hardware design is analogical to code being compiled. Depending on the target architecture we use a different compiler - similarly, depending on the target semiconductor device, we use different logic synthesis tools. This gives us the flexibility of porting one hardware design onto a range of different platforms, given that logic synthesis tools are available.

2.2 Efficient and portable FPGA design

Any device that is designed for ASIC can in principle, be implemented on any FPGA device. The challenge is to make the resulting FPGA circuit efficient. The circuit needs to make effective use of FPGA embedded resources such as LUTs, multipliers, flip flops, BRAMs, and *debug support units* (DSU).

The building blocks of FPGAs differ from ASIC; furthermore, FPGAs differ in their embedded resources among vendors and within their respective chip families. This is why the key to efficient design is to abstract the circuit architecture away from the mapping. The architecture should focus on utilizing generic FPGA features, and the mapping should focus on implementing the components efficiently.

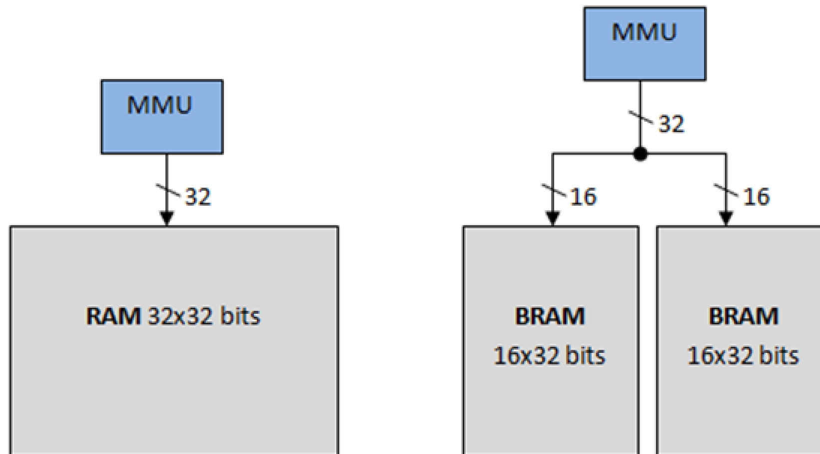
For example in the Figure 2.1 from the architecture point of view we have a memory management unit connected to a 32-bit wide RAM array. On a particular device it might be best to implement it using two 16-bit wide BRAMs. If the mapping is separated from the architecture, we could quickly port the design onto a FPGA device that could only implement the ram as 4 8-bit wide BRAMs, by simply adding mapping constraints for the new device. The example is over simplistic but it illustrates the concepts underlying behind architecture and mapping separation.

Table 2.2: Design Flow Comparison.

[c] ASIC	Reconfigurable Hardware
The designer has to take into account the physical constraints of the device. For example the high leakage current in transistors fabricated in the 90nm process. [21]	The designer has to take into account the architecture of the target device, not its physical implementation.
Requires careful floor planning which can take months.	Hardware synthesis takes up to two or three days for biggest chips with high resource usage designs.

FPGA devices vary in the degree to which they are portable. For example Leon 3 has been implemented using a range of devices from ASIC to CPLD, while some, like Xilinx MicroBlaze / PicoBlaze, target specific devices, as they make heavy use of vendor hardware libraries. The portability/performance ratio is an important aspect of FPGA design, which always has to be taken into account. Vendor/chip specific hardware libraries can provide substantially dedicated circuits like faster CAM memories or *arithmetic logical units* (ALU). For a truly portable design this has to be separated from the architecture.

Figure 2.1: Architecture of circuit and its mapping.



2.2.1 CAM memories and FPGAs

There are a number of techniques available to implement a CAM on an FPGA. The most important fact about FPGA CAM circuits is that they are expensive resource wise. They can be implemented in a number of different what changes

their various properties. We approach several different CAM designs:

- Generic multi-cycle designs.
- Fast vendor specific designs.
- Very fast and area efficient designs based on LUT requiring of chip software control.
- Resource expensive generic single-cycle register file based CAMs.

When discussing various designing and referring to CAMs we will characterize them by abstracted properties. This has to be done as new CAM designs are being approached by various research groups around the world and we want our design to make use of whatever best tools are available across different platforms.

FPGA CAM implementation properties:

- *Control:* On-chip line update or Off-chip line update.
- *Main resources:* LUT, BRAM or SLR.
- *Read/write mode (if read/write is multi-cycle):* Exclusive read/write or Concurrent read/write.
- *Read/write mode (separate read/write port):* Exclusive read/write or Concurrent read/write.
- *Write cycles:* Multi-cycle write or Single-cycle write.
- *Read cycles:* Multi-cycle read or Single-cycle read.
- *Registered outputs:* Outputs are registered, Outputs are not registered or Outputs are optionally registered(specified in design stage).
- *Line match support:* Single match support or Multiple match support.

2.3 Introduction to Soft processors

Soft processors are processors that can be implemented using different Logic synthesis tools. They can be implemented using FPGA, ASIC, CPLD or any device [11]. They are designed using one of the hardware description language (HDL).In almost all cases they follow the previously described method of design where architecture and mapping are separated. Soft processors have few very important features:

- *Parametrizable* - A soft processor can be instantiated with different parameters in the same manner objects are instantiated with different variables. The possible parameters are cache size, cache architecture(disabling different levels), enabling/disabling extra cores like PCI controller or extending the instruction set.
- *Dependable on target device* - Although soft processors can be and usually are portable, there are some physical constraints that can limit that

  - It is impossible to run a large Java program on a 64 MB ram system, and in the same way a huge circuit cannot be mapped onto a small device.
  - Some processors can become very slow due to physical properties of a device. A soft processor might require certain amount of fast on-board memory to implement cache or have certain timing constraints that cannot be satisfied by the target platform.

  It is very common for soft processors to be designed for one range of chips or even a specific semiconductor device. Different LUT/multipliers size might mean that a soft processor might work well on one FPGA while on other it might waste a resources and on a CPLD it might not even be possible to implement it, because of the way the ALU was designed to exploit hardware based multipliers.

- *Modular* - Vast majority of soft processors are developed using *intellectual property cores* (IP cores). Looking at the Figure 2.2. describing soft processors data path we can see how this can potentially benefit us (The diagram is based on part of the MicroBlaze Processor Block Diagram [1]). For example in certain applications it would be wise to disable the divider while in other the potentially saved resources would not balance out the lowered *instructions per cycle* (IPC).

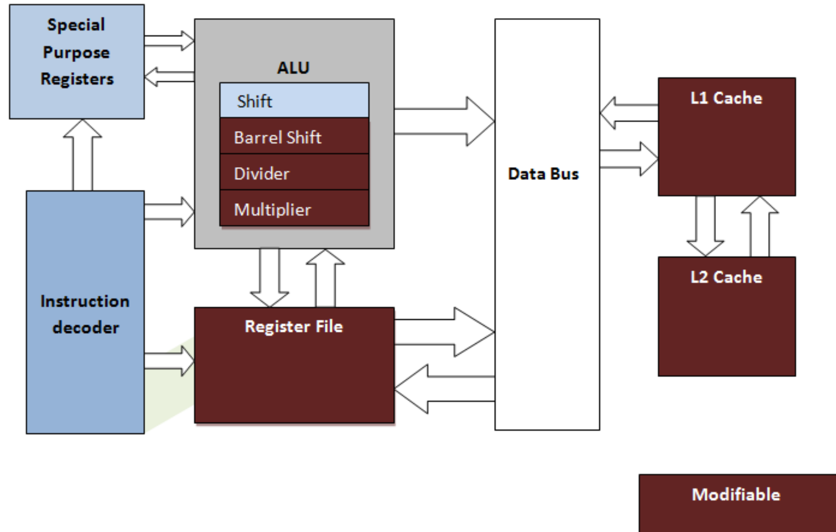
2.3.1 Difference between hard and soft processors

There are number of differences between soft and hard processors. A soft processor can, in theory, be implemented on any device which has a associated logic synthesizer, whereas hard processor is device specific and completely unportable; or the cost is unreasonable high. The dependability of soft processor performance on the target device is crucial to the understanding of the idea of a soft processor. Soft processors are not always implemented using NAND gates. If our target device is FPGA, the micro blocks one would use to implement a processor would consist of LUT, multipliers and flip flops. There are a number of ways of implementing a multiplier on an FPGA and each is a compromise of speed, power efficiency and resource usage. Some soft processors are easily portable (Suns OpenSPARC have been implemented using a range of devices from ASIC to CPLDs [2]), while some (like Xilinx Microblaze/Picoblze) can be only implemented using a small range of chips.

The fact the soft processors are clocked in the range of MHz instead of GHz and consist of far smaller number of logical gates than hard processors means that in a large number of cases they are going to be orders of magnitude slower than their hard counterparts. That is an obvious disadvantage, which is countered by customizing the processors to better fit the goal performance levels. A modern x86 CPU comes with a huge set of instructions, however within one application usually only a very limited subset is used. In the case of a soft processor, a user can customize so that they would not be implemented on the semiconductor device. This saves hardware that can be used for something else, for example including an extra ALU or increasing the precision of floating point units, which might be of greater benefit to the potential user. Overall, soft processors counter the speed disadvantage by task specific customization.

By making more efficient use of hardware, we might be able to add vector integer instructions or implement a faster integer multiplier. One can even design custom instructions depending on a target application [16].

Figure 2.2: An overview of a modular soft processor architecture.



The data-paths relatively slow speed compared to memory access time has a huge impact on soft processor architecture. A memory miss does not take hundreds of cycles. This means that elaborate multilevel cache system is not a necessity. This property has to be taken into account when designing a soft processor for an FPGA or a CPLD device.

Soft processors are far more suitable for low scale systems, where the cost of developing an ASIC chip would be outweigh the potential gains. The cost of producing a single FPGA chip is much higher, but designing an efficient ASIC chips takes months if not years, where else designing a soft processors can take days (given that one is only customizing a ready design). Even a up to bottom full scale soft processor design is less time consuming, mainly due to lower prototyping cost.

Table 2.3: Differences between hard and soft processors.

	Soft processors	Hard processors
1.	Soft processors can be easily ported onto any semiconductor device if there exists a target logic synthesizer.	Designed for a specific device, and usually requires substantial amount of work to be ported onto any other one.
2.	Clocked in MHz	Clocked in GHz
3.	The clock-rate difference between data-path and memory is not as prominent	Data path is orders of magnitude faster than memory
4.	Short design cycle	Long design cycle

2.3.2 Soft processors automatic generation and optimization

The advantage of soft processors is their potential for customization. Soft processors can be customized to match specific performance/power/area usage goals. On the contrary modern general purpose processors are used for a wide range of applications, therefore they cannot be optimized for a small domain of applications. Two problems have to be tackled in order to efficiently use soft processor technology.

2.3.3 How to quickly and efficiently generate soft processors?

After choosing the processor, target device, performance, energy usage, functionality and area usage levels the user has to optimize the processor to meet the specified goal. A number of ways have been proposed to solve this problem. Assuming that average configuration space of a processor lays in the range of thousand of different settings, each requiring many multi-hour simulations, it is infeasible perform a full search on the hypothesis space. A number of approaches have been proposed to solve this problem. From the very simple one of using "common sense" up to more elaborate ones like based on the Design of Experiments Paradigm proposed by Sheldon, Vahidand and Lonardi [12]. What they offer us is a way of finding a processor setting matching the goal performance level, in a reasonable time. According to Sheldon, Vahidand and Lonardi they were able to find a 3x-6x faster processor compared to other approaches in a smaller number of simulations.

2.3.4 How to optimize the soft processor?

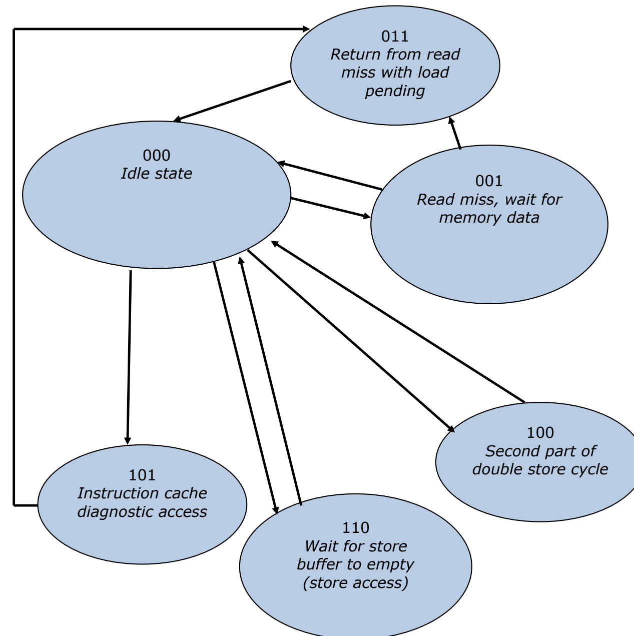
During the processor generation process, a user has to use a number of benchmarks and simulations in order to optimize the processor by investigating the performance impact of various parameters. Different power usage estimation strategies such as a methodology based on instruction level energy profiling [12], performance estimations and verification tools have to be used. For example a system has been proposed which by identifies critical code of a target application and extend the processor instruction set with custom ones which might vastly improve performance (speed-ups between 1.7 and 6.6 times, code-size reductions between 6% and 72%, and area costs ranging between 12 and 256 adders for maximum speed-up)[16]. Ideally all of that should be integrated within the processor generation software package. If the software available is both efficient and accurate it means that the user is going to be able to better fit the processor within the required performance goal.

2.3.5 Soft processors caches

Most caches can be divided into two parts. First is the cache control unit which can be integrated with the *memory management unit* (MMU). The cache state machine for Leon 3 is presented as an example in Figure 2.3. The second building block is usually a memory array. On FPGA this would be a BRAM

file. The scheme is very similar to ASIC implementation, with the difference that the resources are not transistors but BRAM and LUTs.

Figure 2.3: Leon 3 cache state machine.



Due to the high cost of CAM memory fully associative caches are very rarely used. The LUT, BRAM and flip flop cost can be so high that for even a small cache it would have to use resources of an entire chip [3][4]. That is why nearly all designs only support direct mapped and 1, 2 or maybe 4 set associative designs.

The fully associative cache is very likely to increase the designs critical path and power usage because of vast matching circuit. In order to keep the designs operating frequency and/or power usage at reasonable level this might enforce the use of pipelined cache or CAM. For example with a tag matching stage, and tag/validation/context access stage. There are a lot of different approaches to the topic depending on CAM size and performance goals [24][20][23].

2.4 Brief overview of Side channel cache attacks

There are a number of ways of breaking a cryptographic system, we present two of them.

- *Brute force attack*, is an attack which tries to force a security mechanism by traversing a search space of possible keys.
- *Side channel attack*, is an attack trying to utilize information based on physical traits of a target system. This can be anything from acoustic, thermal or timing properties.

Modern processors and their components suffer from many physical security flaws. The flaws vary in the potential risk that they pose and the difficulty in which they can be exploited by the attacker. Side channel cache attacks cache are relatively difficult to exploit, but can lead to serious key/passwords thefts. Below are listed different types of attacks, as mentioned by Isuru [14].

- *Timing attacks* - Based on the timing difference between certain cache access patterns executed at the target system. The technical details of the attacks are out of scope of this paper, only the understanding of the simplified concept presented in the next section is required.
- *Fault Attack* - Information gathered by causing exceptions and faults on an attacked device can help the attacker to breach the system's security.
- *Power Analysis attacks* - Attack based on the information gathered by analysing the power consumption of the target device.
- *Visible light attack* - Monitoring systems have been proven to be capable of "seeing" through walls. This can possibly allow the attacker to see the screen output despite being separated by a wall or any other obstacle.
- *EM attacks* - Exploit the electromagnetic radiation spectrum to attack a processor. Different wavelengths than visible light can be used to "see" what is happening within a processor or what information is transferred through a cable (for example Ethernet cable connecting different parts of a system).
- *Cache based attacks* - Attacks which monitor data that is being moved in and out of a processors cache. There are two types of cache based attacks. The first one is trace driven attack which makes use of cache hits and misses. The second one is a cache timing attack which makes use of the fact that cache misses and hits are handled in different time.

There was a notable case of timing attack (non cache) TENEX OS password checking mechanism being broken using the fact that it compared one character at a time. It terminated immediately if a checked character did not match against the compared password. This enabled the attacked to determine a stopping point, where all the characters before where contained within the password. This allowed for a speed up of the brute force attack thousand-fold [9]. In the same manner a cache timing attack can give a clue to the attacker on secret key values.

The main difference between timing, fault and cache based and the rest of the listed attacks lies in the fact that timing attack can be performed remotely while the latter require some kind of physical monitoring system. This report is mainly concerned with the cache based attacks.

2.5 Cache Side Channel Timing Attack

The relevant characteristic of the cache side channel attack is that it exploits the predictability of cache access patterns. Knowing the cache architecture and the replacement policy of the target processor over a number of processor cycles, an attacker can create a channel that can be used to steal confidential data. There are many examples of various side channel attacks, the most prominent being cache side-channel timing attack against AES. [9][10][18]

We can imagine the timing attack as probing a cipher machine. We know the machine mechanism, but we do not know its cipher key (used to code messages). The key cannot be recovered by comparing inputs and outputs (in a reasonable amount of time) due to sophisticated coding mechanism. What we can do is attack the cipher machine by using a timing attack. By comparing the amount o time necessary to prepare a batch of coded messages we might be able to verify small bits of the cipher key. After comparing large number of messages we might be able to verify the full key (or narrow down the search space and prepare a brute force attack). This is essentially a timing attack. Knowing the structure of the machine, the timing attack uses the information leakage in the form of variation of time necessary to generate output.

2.6 Countermeasures for Cache based Attacks

As people have been aware of the cache side-channel attacks for a while, there is extensive literature covering this topic. A number of countermeasures have been proposed, some of which are listed below.

- *Disabling Cache*, The most obvious solution to the cache based attacks. It has a serious drawback of processor performance degradation. Very inefficient (slow).
- *Constant Timing programming*, Extra instructions which hide timing from the attacker are added to the applications code. This method of attack prevention can slow down the application significantly, and usually only forces the attacker to use a larger number of samples to extract the desired information.
- *Partitioning Cache*, This method works by partitioning the cache, it is designed for a multi-threaded system. By allowing certain threads to access only a subset of cache, the attacker is denied the possibility to steal the information held by another user (thread) using a timing or trace driven attack [19].
- *Security-aware cache*, A way of countering the side-channel cache timing attack has been proposed by Wang and Lee at Princeton University [23].

It is reviewed in detail in the next section. It is based on the idea of dynamic cache partitioning.

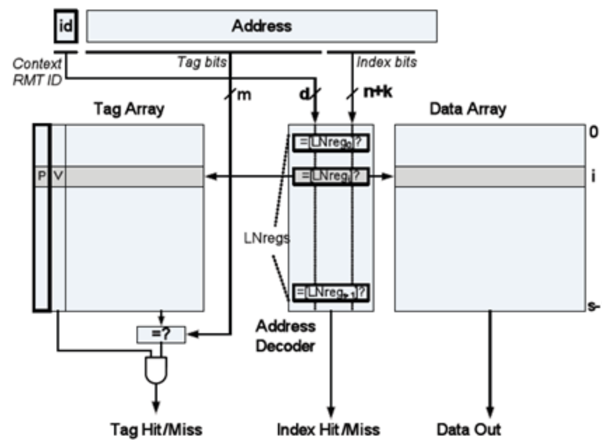
Table 2.4: Comparison of cache timing attack countermeasures.

Disabling Cache	Constant Timing programming	Partitioning cache	Security-aware cache
Degrades Performance	Degrades Performance in case when security features are enabled	Degrades overall performance, compared to standard cache architectures.	Improves overall design performance, when compared to standard architectures
Lowers resource cost	Cost the same	Increases resource cost	Increases resource cost
Hardware based	Software based	Hardware based	Hardware based

2.7 Security-aware Cache Architecture Overview

Security-aware cache is essentially a direct mapped cache with a more elaborate address decoder and a new replacement algorithm named *security-aware cache replacement algorithm* (SecRAND). According to Wang and Lee the security-aware cache nearly matches the *direct mapped cache* (DM cache) address decoder delay; the difference between the two being negligible; 64 KB 0.192ns against 0.197ns, 1 KB 0.149ns against 0.151ns [23]. Furthermore they have shown that the new cache has less conflict misses than a DM cache of a larger size, and almost as few as comparable size fully associative cache. Due to the new decoder the new cache occupies a slightly larger area than DM cache. The overhead depends on the security-aware cache parameters, but should lie around the 5% overhead mark compared to same size DM cache. The security-aware cache is summarized in Table 2.4.

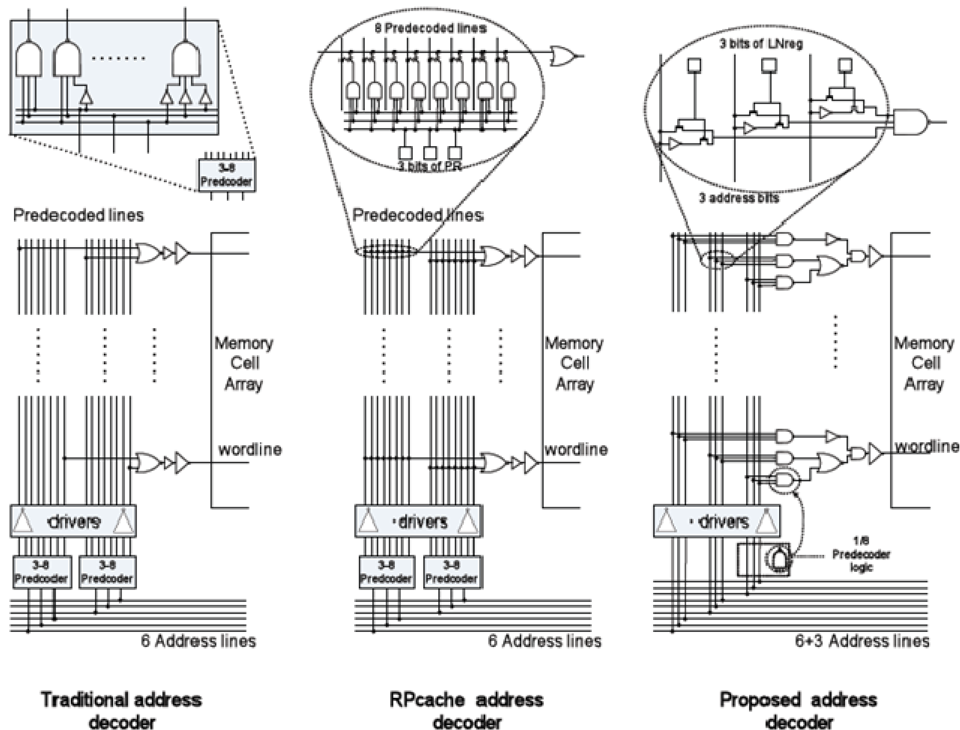
Figure 2.4: Security aware cache. Diagram taken from Wang and Lee [23].



2.7.1 Security-aware cache address decoder

The cache address decoder dynamically partitions the cache. A correct grouping of the processes protects them from cache based attacks. A group of processes that do not "trust" each other should be assigned a different context remapping table id RMT ID. Wang and Lee described a way of implementing a new hardware decoder using traditional transistor representation. The advantage of the design lies in dynamic grouping, which does not cause a performance degradation (given that the partitioning mechanism is fast). The concept is similar to that of partitioned cache, but has the advantage of better memory allocation between different thread groups.

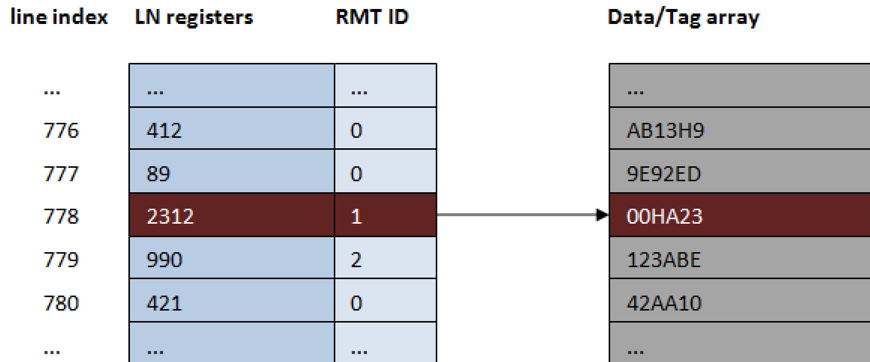
Figure 2.5: Comparing ASIC static ram address decoder circuits. Taken from Wang and Lee [23].



2.7.2 Dynamic cache line remapping

Instead of directly mapping cache indices to tag/data arrays, the lines in the security-aware cache are being dynamically remapped using line number registers called LN registers. Every cache line has an assigned remapping register that is used to access the appropriate cache line in the address decoding stage (the value it stores is the index of the line). The line accessed is determined by

Figure 2.6: Thread with context $ID = 1$ accessing security-aware cache line with $index = 2312$.



the index stored in the remapping registers. For example in the case where register at line number 778 stores *index* 2312, *index* of 2312 would cause an access to $line = 778$. A simplified view of the security-aware cache and an access to line 778 indexed 2312 is presented in Figure 2.6.

The full concept can be clearly seen in Figure 2.4. Simple remapping would neither increase the hit rate nor decrease the decoder latency as an associative search has to be performed on the registers before accessing memory line. This is why the remapping registers can be extended to hold a longer index expanded by k bits. This effectively extends the cache index range, and can significantly lower the miss rate [23]. Any register can hold an arbitrary index, although no two registers can hold the same index. There are n remapping registers, each pointing to one tag and data cache line. The cache has an extended address space of $n+k$, although its physical size is determined by n only.

2.7.3 Line context field

A *line context field* (we will refer to as context ID or simply ID) is associated with every line, so that a process/thread address space can be separated from the rest of the processes/threads if desired. As explained later it is needed if we want to prevent channel creation by a malicious user. Depending on the number of separate number spaces we need, the tag line is extended with d bits, where 2D gives us the number of secure address spaces we can get. In 2.4. the user (process or thread) with access granted to lines with $ID = 1$ selected line indexed with 2312. What would happen on an access to $line = 780$ with $index = 421$ and $ID = 2$ is explained in detail later.

2.7.4 Address decoder

Dynamic cache line remapping and associative memory architecture have been exploited before [15][8][17] and most designs have significantly increased delay. The security-aware cache decoder [23] has been designed using transistor level

representation; this decoder has a marginally higher delay than the decoder used in a traditional direct mapped cache. The transistor level description of different address decoders can be seen in Figure 2.5.

2.7.5 SecRAND overview

The security-aware cache algorithm is based around a random number generator. Cache hits are handled as in ordinary direct mapped cache and there are three types of misses:

- *Index miss* - It occurs if there is no remapping register holding the needed index.
- *Context miss (Tag miss involving protected page)* - If there is an index match, but either the requested or stored line has a different context, then the context is protected. By protected we refer to a block that was written to cache by a process that wants to be safe against channel attacks from processes from a different context. A context is a group of processes that ”trust” each other within their group when cache channel attacks are considered.
- *Tag miss* - An unprotected tag miss. This means that the index is matched with a line but a tag does not match the requested address i.e. the line lives in the same secure context as the process that has requested data from the cache.

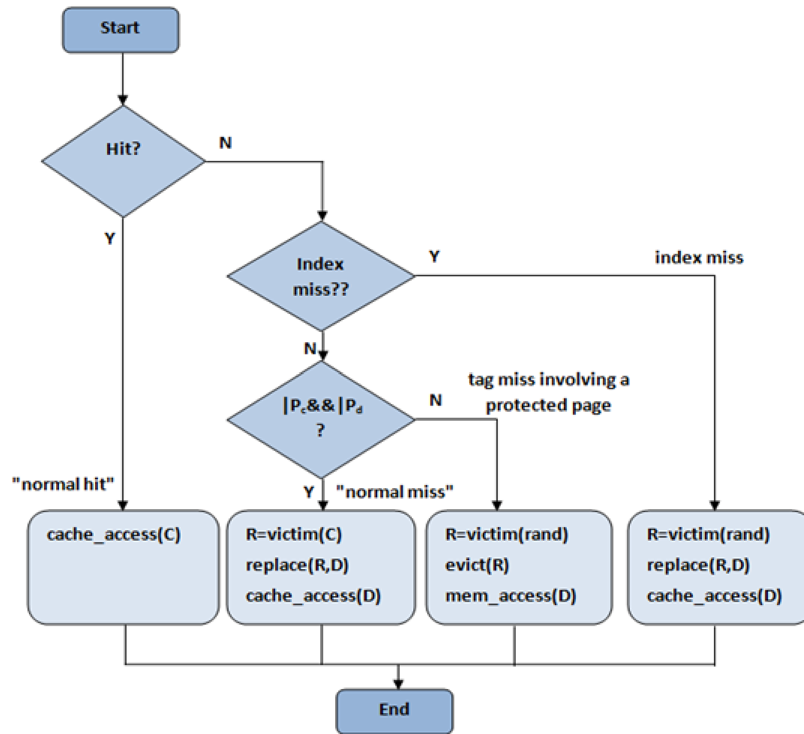
Referring to Figure 2.6 given that a process has access to lines with $context = 0$ or 1. On a index hit a normal access to the cache line is granted; for example $index = 2312$ where the line has $context = 1$. If the user tried to access $line = 561$ that is not stored in the cache, an index miss would occur. This would cause a randomly chosen cache line to be evicted and the requested word would be fetched from memory and stored in cache, it could replace any line no matter the context. On the other hand, if the process referred to word with $index = 990$, and a tag miss would also occur than the behaviour would be very different(*context miss*).

The cache line that stores memory line with $index = 990$ belongs to a different context. A randomly chosen line would be evicted and the data would be accessed directly from memory without a line in cache being allocated for it. This way of handling tag misses with a context conflict hides the information from the potential attacker on whether certain memory words are stored in cache, for example words that hold parts of AES key. The attacker cannot verify whether a word is cached or not by accessing cache lines and measuring the average response time of the system. There is also one more scenario possible, that the process accessed line with $index = 89$. The process lives in $context =$ 0 and 1 as such if $line = 777$ is accessed (with $index = 89$ currently associated with it) but stores a different tag than the requested word, an ordinary tag miss occurs. The SecRAND algorithm is shown in the form of a flow chart in Figure 2.7.

Table 2.5: Notation used in the SecRAND algorithm flowchart [23].

C	The cache line selected by the address decoder (during a cache hit or a tag miss).
D	The memory block that is being accessed.
R	The cache line selected for replacement (victim).
P_x	The protection bit of X. If X is in a cache line, it is the P bit of the cache line. Otherwise it is determined by the PP bit of the page/segment that X belongs to.
cache_access(C)	Access C as in a traditional Direct Mapped cache.
victim(C)	Randomly select any one out of all possible cache lines with equal probability.
victim(rand)	Replace R with D, update LNreg.
replace(R,D)	Write back R if it is dirty; invalidate R.
evict(R)	Write back R if it is dirty; invalidate R.
mem_access(D)	Access to D without caching it.

Figure 2.7: SecRAND algorithm flowchart.



The advantages of security-aware cache

We summarize the advantages of the security-aware cache over traditional cache architecture (ASIC):

- Nearly matches direct mapped cache in terms of access time, the difference is negligible.
- Has hit-rate which is basically equivalent to Fully associative cache (For $k = 4$).
- Has power efficiency that matches Direct mapped cache.
- The resource overhead is small compared to potential benefits. For reasonable k parameter ranges in the area of 5 to 10%.

Random Number Generator

The main problem with ordinary cache when taking under account side-channel attack is its predictability. If the attacker knows the cache replacement scheme, and its underlying architecture he can exploit its features. By randomizing the replacement policy, the predictability problem is solved. The random number generator used in the SecRAND algorithm can be implemented in a number of different ways, with the restriction that attributes like seed and number period should be impossible to recover by the attacker.

2.8 Summary

This Chapter provided the essential background for understanding of the concepts underlying the security-aware cache and its FPGA implementation. The most important topics covered:

- The difference between ASIC and FPGA.
- The differences between soft and hard processors.
- Cache side channel timing attack.
- CAM FPGA implementation and the associated problems. High resource usage, multi-cycle operation and long critical path being most prominent.
- Security-aware cache, and how it works. It is composed of the new replacement algorithm (SecRAND) and the index remapping circuit. The index remapping circuit should be based on CAM memory.

Chapter 3

Efficient Design of Security-aware Cache on FPGA

This Chapter discusses the architecture of the modified security-aware cache. It is assumed that the reader is familiar with the Chapter 2.

The main novel aspects of our design:

- Efficient index re-mapping decoders and their optimization.
- Modified Security Algorithm.
- Design portability.

This Chapter is divided into five sections. Sections 3.1 to 3.3 present the new index re-mapping decoders and their optimization. Section 3.4 describes how the security-aware cache algorithm was modified to work with the new decoders. Section 3.5 summarizes how the design portability was achieved.

The novel aspects of the design and the problems they address:

- *Security-aware cache* - Different building blocks of FPGA and ASIC design, identifying the complications and differences. Design portability and ease of modification.
- *Pipelined security-aware cache* - Solution to the increased critical path.
- *L-associative cache* - An answer to FPGA CAM high resource demand.
- *Modified Security Algorithm* - Modified algorithm addressing the new decoders.
- *Design portability* - Abstraction of CAM implementation.

Overview of the new index re-mapping decoders

The main challenges with security-aware cache FPGA implementation are related to the FPGA CAM implementation problems:

- High resource cost of CAM memories.
- Multi-cycle read/write CAM operations.
- Unregistered CAM increases critical path.
- Difficulty of modification of existing designs.

To implement the remapping registers our design makes use of either a combinational CAM index decoder, or a multi-cycle CAM memory. The first is used when single cycle data/tag read is needed and leads to a cache design which we are going to refer to as *security-aware cache*. It allows for a quick design modification and evaluation. The second is used if line access can be delayed by one cycle, and leads to what we will refer to as *pipelined security-aware cache*. The number of read/write operations per cycle is the same for both designs as both read and write can be issued every cycle despite the results being delayed in the second case. The CAM based address decoders are used to find the correct index, and to issue index hit/miss. They replace the LN register Array in the ASIC security-aware cache [23]. It is an *unregistered* CAM.

To counter the high resource demand of FPGA CAM memory implementation we propose a new cache architecture to which we will refer as *L-associative security-aware cache*. It is based on the associative principle but applied to address decoder.

We will refer to CAM memories with combinational index decoders as *unregistered*. Clearly because the outputs are not registered. The other being called *registered*.

When discussing the new decoders we use the following notation:

- n - The number of index bits.
- k - The number of bits used in security-aware cache to extend index.
- t - The number of tag bits.
- d - The number of bit used to define line context.

3.1 Security-aware cache

Table 3.1: Ordinary security-aware cache overview:

Advantages and Disadvantages	
Utilizes LUTs and registers therefore synthesize-able on any FPGA.	Likely to become the critical path (the likelihood increases with CAM size).
Based on generic CAM. The CAM becomes part of the cache control logic.	Very narrow range of applicable vendor specific CAM memories
The decoder can be easily used to extend any existing device by ”wrapping” tag/data files.	

In our first design we approach the problem of portability and modification of existing designs. In order to make the design portable we need to ensure that it is synthesize-able on any FPGA platform. This is achieved by making use of resources that are synthesize-able on any FPGA platform. We designed the CAM using a register file along with a combinational search circuit, in order to make security-aware cache a cross platform solution. The main novel aspect of the design are:

- Transparency from cache controller point of view.
- Portability.

In order to achieve ease of modification we make the modification transparent from the existing cache state machines. The CAM memory matching circuit extends cache control logic remapping the indices before they are used to access cache lines. The modification is transparent as CAM matching is done in the same cycle as the access to cache line.

Decoders CAM properties:

- *Control:* On-chip line update.
- *Main resources:* LUT.
- *Read/write mode (if read/write is multi-cycle):* Not applicable for read, write is single-cycle.
- *Read/write mode (separate read/write port):* The design is single ported.
- *Write cycles:* Single-cycle write.
- *Read cycles:* Combinational unregistered read.
- *Registered outputs:* Outputs are not registered.
- *Line match support:* Single match support.

The way we aim to modify existing designs is by extending the cache control logic with the remapping register file. Equally it can be seen as wrapping the tag/data arrays. The extension is done by inserting the remapping logic between tag/data array input and circuit responsible for tag/data array access. *Note* that the CAM matching circuit effectively becomes part of the cache control logic, therefore from control perspective read/write operations can be performed in the same way as in unmodified cache. The circuit for this decoder can be seen in Figure 3.4.

When an access is issued to the tag/data array we do not use the usual address index of width n to access a tag/data line. We access the CAM memory decoder with index of width $n+k$. As discussed in Chapter 2 and by Wang and Lee [23] k is the extended index used to improve performance. This index is later mapped to a tag/data array with n lines.

When we issue a write to cache if the index we are searching the CAM for resides in the cache, we use the address provided from the CAM memory to access the appropriate cache line. If it does not reside in CAM, we use the number generated from a random number generator to both index tag/data line as well as map the index to a line in the CAM memory. This way we ensure a single-cycle write to the tag/data lines as well as the CAM memory. In order to decrease the potential critical path we register the write parameters, and perform write to CAM in the following cycle. Doing that we need to check whether a write was issued in the previous cycle, and whether there is a index match. The index from write buffer has precedence over the CAM mapped indices as the written index is more recent.

3.1.1 Increased critical path

It is important to note that the CAM memory physically becomes part of the cache control logic. Although in most designs the *execution stage* (EXE) of a processor data-path is the critical path and determines the frequency at which the processor will operate, this might not be the case with this type of security-aware cache. The CAM circuit extends the cache control circuit by a significant amount of logic, if the CAM exceeds certain size it will become the critical path. The transition can be seen from Figure 3.1 to Figure 3.2. The potential critical path is marked in Figure3.4. From the behavioural point of view this type of cache is identical to the security-aware cache as described by [23].

3.1.2 Using multi-cycle CAM memory

Using a M-cycle write memory is possible, and the concept is clearly explained in the next section. We do not recommend it as it requires more complex control mechanism and this platform should be used either as a temporary fix to cache timing attack (till pipelined design is used) or as a quick evaluation platform.

Figure 3.1: A simple diagram of a data path without security-aware data cache.

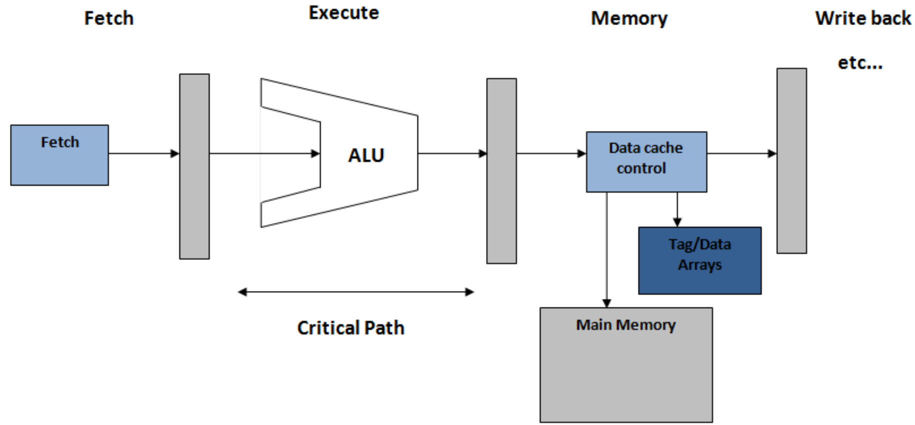


Figure 3.2: A simple diagram of a data path extended with the security-aware data cache logic. It is important to note that the critical path increases as the remapping register file becomes part of the control logic.

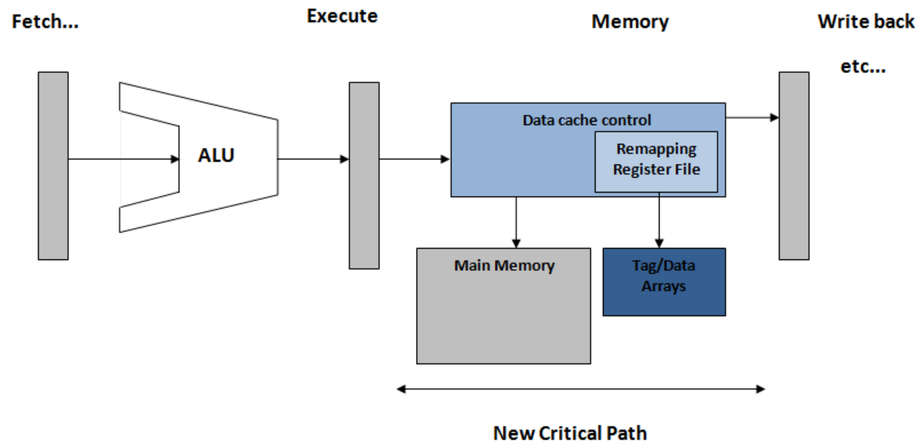
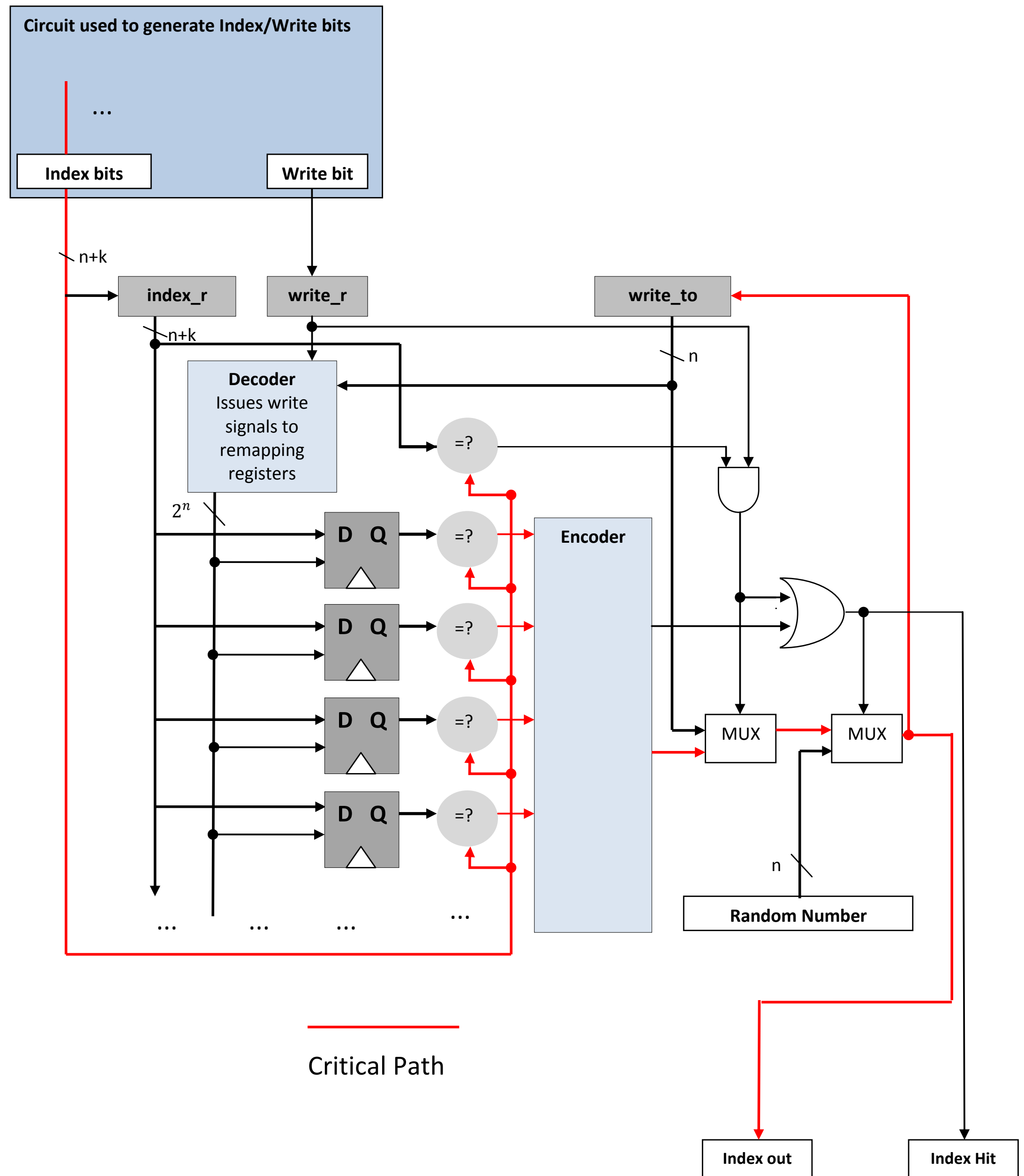


Figure 3.3: Security-aware cache index remapping circuit. The critical path is marked on the diagram. The circuit can be optimized to decrease the critical path, but for readers convince we present the optimized circuit in Appendix A.



3.2 Pipelined security-aware cache index remapping

In order to counter the increased critical path problem we approach a pipelined cache design. The problems the pipelined cache is trying to counter:

- Increased critical path. Pipelined cache *does not* extend the cache control unit, this results in decreased critical path.
- Portability, using a wide range of different CAM memories to implement the remapping register file.

There were a number of attempts to create pipelined associative caches/-CAM memories before (essentially the pipelined decoder). For example by using a hierarchical search scheme as described by Pagiamtzis and Sheikholeslami we could lower the power usage substantially [20]. The motivation behind most pipelined CAMs and caches is to decrease the critical path and power, the motivation behind our cache is portability and throughput.

The main novel aspect of the pipelined security-aware cache:

- CAM FIFO buffer which is used to abstract the cache behaviour from the underlying CAM.
- Portability due to large range of supported CAMs. The cache can be implemented over a wide range of chips using whatever resources are available.
- Pipelined cache can be based on vendor specific CAM memory, therefore achieving better performance and resource utilization. We have a larger freedom in choosing potential CAM decoders as extra logic needed to accumulate them is less likely to affect designs critical path.

The main difference between the pipelined and the security-aware cache lies within the way we use CAM memory. In the security-aware cache we rely on a LUT/register based CAM that extends the cache control logic (state machine circuit). In the pipelined cache we operate in a two stage mode. During the first stage the cache remaps the index using a CAM memory. In the following cycle the previously remapped index is used to access a tag/data line. Effectively a pipelined cache is created with an index remapping stage and an data/tag array access stage.

Decoders CAM properties:

- *Control:* On-chip line update.
- *Main resources:* Any.
- *Read/write mode (if read/write is multi-cycle):* Both exclusive and concurrent are applicable.
- *Read/write mode (separate read/write port):* Both separate and dual port CAM are applicable.
- *Write cycles:* M-cycles (any).

- *Read cycles:* Single-cycle read.
- *Registered outputs:* Both registered and unregistered CAM are applicable.
- *Line match support:* Single and multi-line match support is applicable.

The security-aware cache and pipelined security-aware cache also differ in the type of CAM memory we use. The pipelined cache is based either on the LUT/register CAM (as in ordinary security-aware cache) or on a single-cycle read and M-cycle store memory. This allows us to implement single-cycle read M-cycle store vendor specific CAM memory. In virtually any case the vendor provided CAM is going to be faster and more resource efficient than a generic LUT/register based CAM. Therefore by using vendor specific CAM we improve the cache resource utilization and decrease the critical path length.

A X-cycle read CAM could also be used, although the number of stages would have to be increased. As most of the stages would be "empty" it is not recommended.

On the other hand referring back to the example of hierarchical search CAM [20], increasing number of index decoding stages could result in advantages like improved power efficiency.

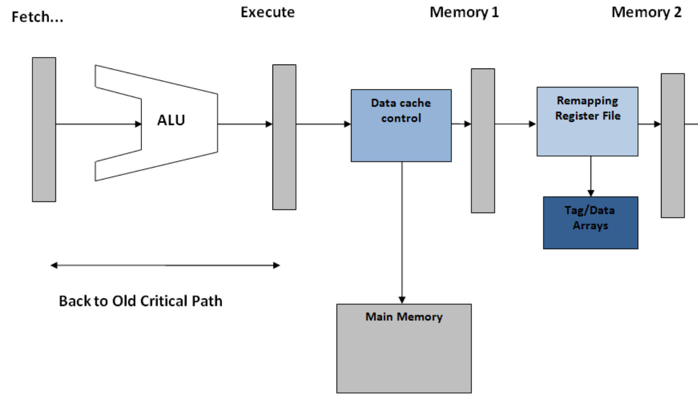
We present the cache in Figure 3.6. A pipelined version of security-aware cache based on the ordinary security-aware cache is presented in Figure 3.7.

Table 3.2: Pipelined cache overview.

Advantages and Disadvantages	
Better resource efficiency.	Two stages, index remapping stage and an data/tag array access stage.
Far less likely to become the critical path, as long as the CAM has a higher operating frequency than unmodified data-path.	More complex design.
Based on vendor specific single-cycle read M-cycle write CAM memory.	

The cache behaviour is altered by the way and type of CAM we use, what has important implications on the design. In the rest of this section we describe techniques on how we counter them. Single-cycle delayed read and M-cycle delayed write CAM enforces a use of buffering mechanism. Furthermore not all CAM memories allow simultaneous read/write operations, potential cause of pipeline stalls.

Figure 3.4: A simple diagram of a data path with a pipelined security-aware cache.



3.2.1 Pipelined security-aware cache based on M-cycle write CAM

In order to perform tag/data array write operation every cycle we make use of a *first in first out* (FIFO) CAM buffer. When an access is issued to tag/data array, CAM memory is used to remap the index. As we are using a single cycle read/M-cycle write memory, we are not going to get the index until the next clock cycle; so we have to delay the tag/data/ID by single cycle. This is why we insert a register between the tag/data/ID input and the arrays to which/from which we are going to read or write. The box between tag/data/id arrays and inputs in Figure 2.2. A similar mechanism is used in nearly all of the modern hierarchical memory systems. For example buffering hard drive or *random access memory*(RAM) operations in modern personal computer systems.

3.2.2 CAM FIFO buffer architecture

What is unique about the FIFO CAM buffer used in pipelined security-aware cache is that the FIFO CAM buffer can be searched through in an associative manner. The FIFO has to be accessible and searched through as otherwise when performing a read operation the cache would be invalid with respect to the write operations that have taken place within the last M-cycles. We present the data flow in Figure 3.5.

It is important to note that the buffer is a CAM memory FIFO. We know which LN registers are going to be updated as the buffer stores the index to which the write is to be issued. It also preserves the write store order therefore we know the current status of the LN registers. In case of a conflict when there are 2 write operations issued using the same random number, we treat the most recent one as valid. This search mechanism can be easily employed using a binary tree structure, with the left branch having precedence over the right one (as it is most recent).

We propose two ways of implementing control logic for the buffer, depending on the frequency of cache write/read operations. The first implementation

requires easier control logic, although it results in some redundancy. We use a simple counter with a cycle of length M that moves the buffered indices every M cycles by one step. The redundancy comes from the fact that if the buffer is empty we could move the incoming write to the front of the FIFO. The second implementation is more sophisticated and differs from first in that it checks the first empty spot within the FIFO and insert the incoming write at that point.

CAM FIFO buffer sizes and handling buffer overflows

The buffer can be of arbitrary size. In case when $M = 1$ we can ensure that the buffer will not ever be filled up (write can be issued every cycle). When $M = 2$ in most caches we can also ensure that the buffer will never fill up. The reason is that cache write operations are usually preceded by read operations, therefore in worst case write is performed every other cycle. This might not be the case when for example cache supports flushing and we nullify all of its entries by writing to cache over large number of cycles. When $M > 2$ determining the buffer size is non trivial and depends on the frequency and length of streams of write operations.

When the buffer fills up, we can either discard any new incoming write messages or stall the cache till the previous write operation finishes. Both methods degrade cache performance but ensure correctness and limit resource requirement. We have to take into account account cache write policy. In case of *write-through* policy the mechanism of discarding messages has a trivial implementation. In case of *write-back* the FIFO control logic has to issue a "discard" signal to cache control logic in order to write back the data into the cache.

The FIFO CAM buffer can be optimized to discard any writes to cache that would invalidate writes that are currently pending or that would not change the state of the cache. If we issue a write with $index = A$ to the first remapping register and then we issue a write to the same register with $index = B$, obviously the old write can be discarded. Note that in case $A = B$, write will not be issued: when we issue a write we search the buffer for stored indices. Again the cache write policy has to be taken into account. With a *write-through* scheme the implementation is again trivial. In case when *write-back* policy is used a more sophisticated control is necessary and as it is case dependent will not be further discussed.

Exclusive read/write CAM

Our solution depends on the memory access patterns and on the M parameter. Clearly M is bigger then 1, as otherwise cache read/write operations would not overlap. We do not recommend exclusive read/write CAM memories unless read/write operations are separated by a reasonable amount of cycles with respect to M, as in order to ensure correctness of cache stalling has to be employed. Pipeline stalling is clearly undesirable as it hinders cache performance. We could also drop write operations, what would also hinder the cache performance. Extra control logic would have to be employed to drop writes to tag/data arrays. Nevertheless we do not recommend these type of CAMs as they hinder the performance significantly.

Figure 3.5: Pipelined security-aware cache operation. We update the cache during the first three and the fifth cycle. During the fourth cycle we perform a read on index 54CFDC, and we get a hit from the CAM and miss from the FIFO CAM buffer giving us a mapping to line 1. In cycle 5 we issue a write to line 1 with a new index. The CAM will not be updated with the index till cycle 9, nevertheless we can read the state of the cache. During the read operation in cycle 6 (77C1DC) the cache will read data/tag from line 1. The FIFO CAM buffer has precedence over CAM mapping and will return us mapping to line 1, despite CAM holding a different index at register 1 (54CFDC). Clearly in a 1 cycle read/write CAM the output from the cache would be the same.

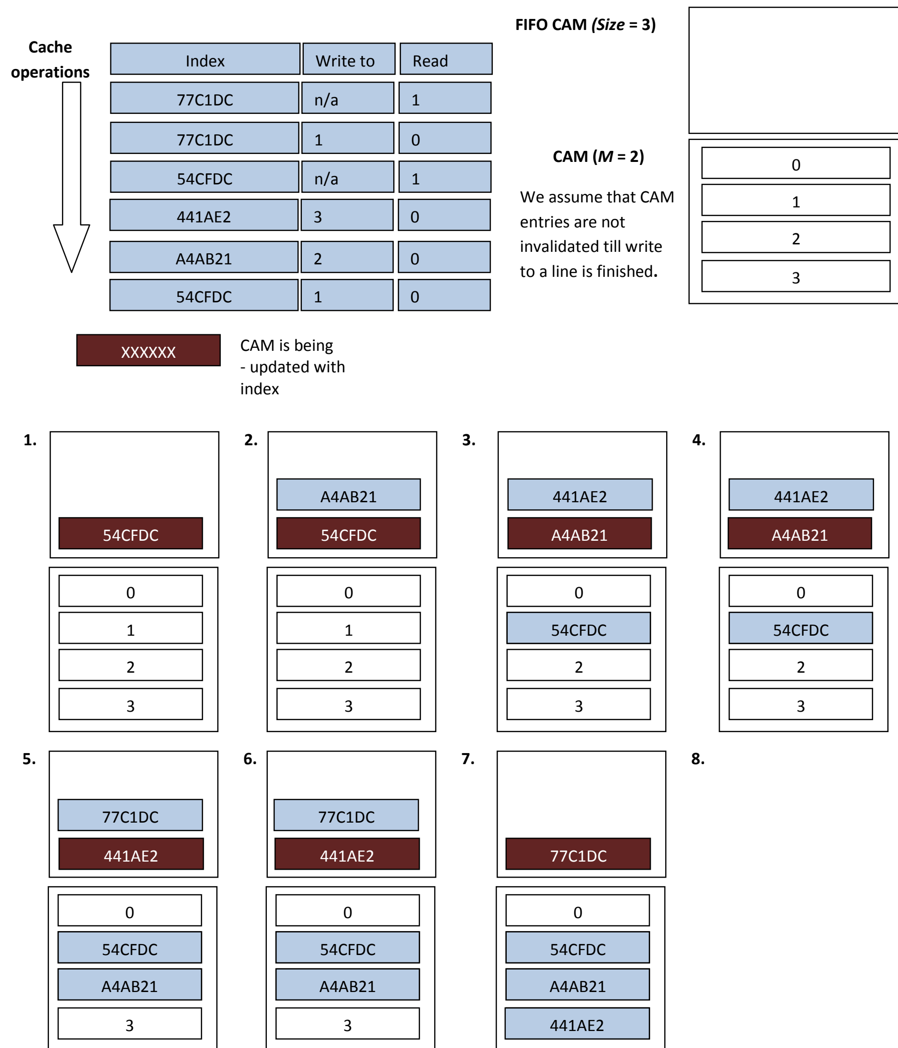


Figure 3.6: Pipelined security-aware cache.

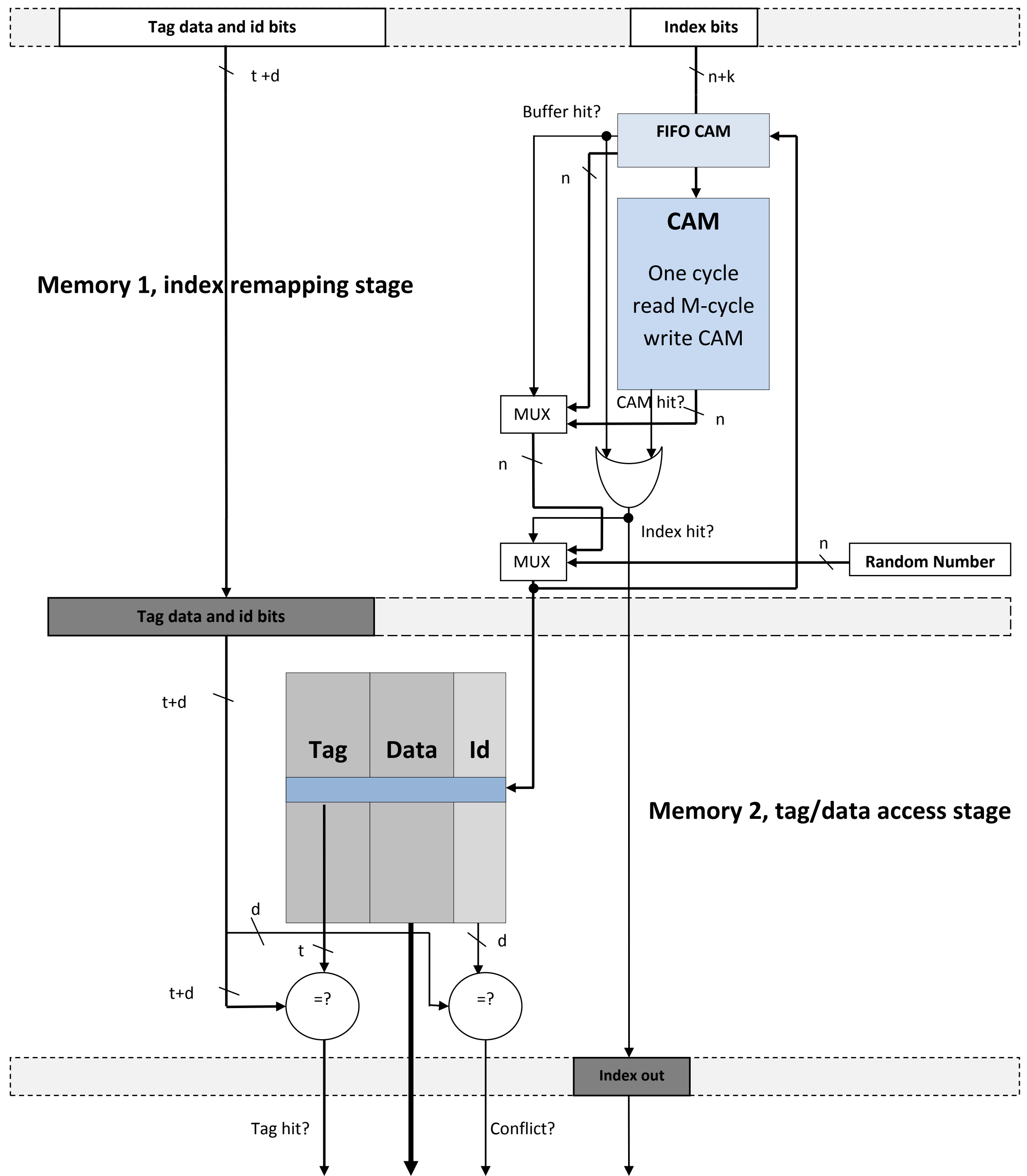
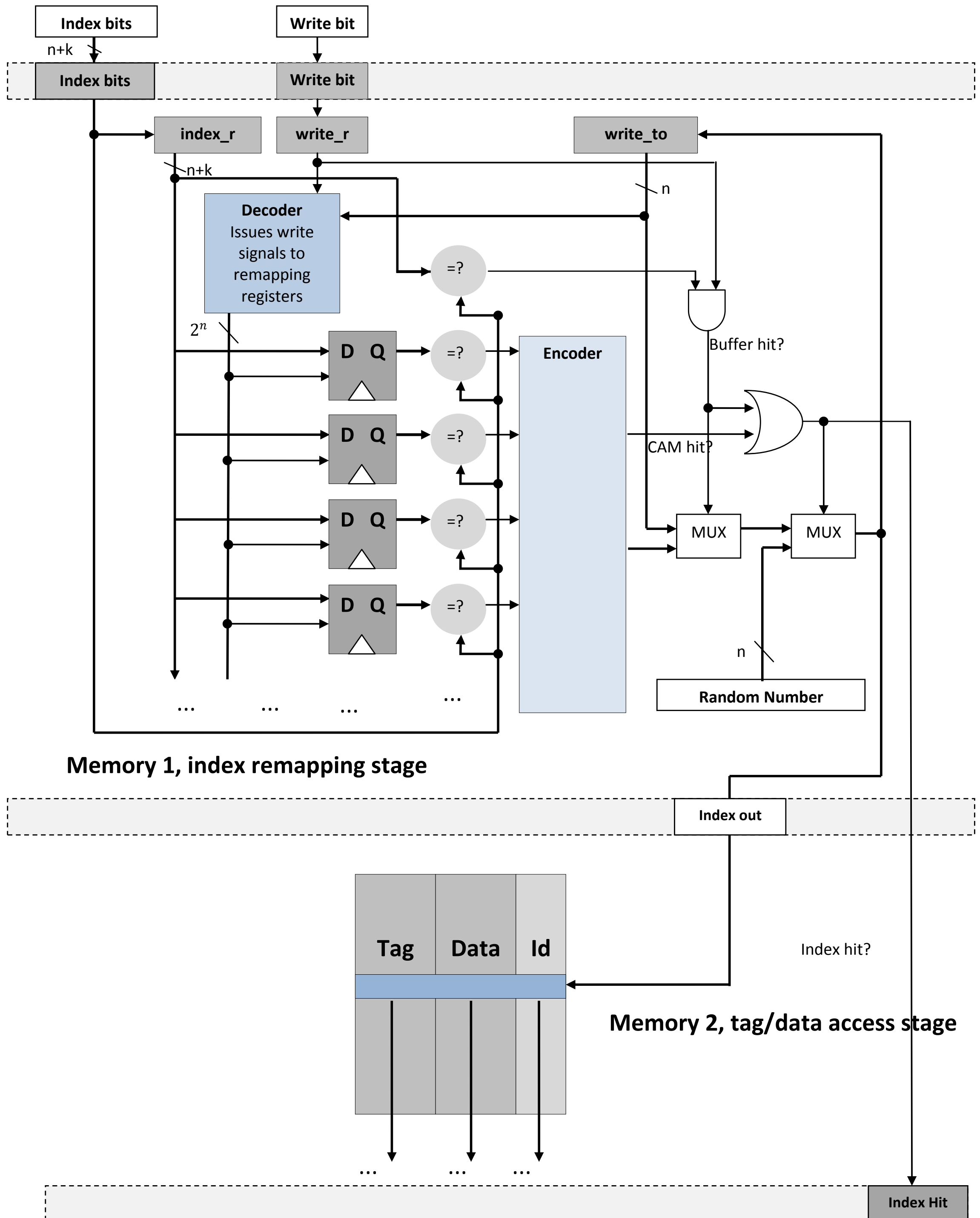


Figure 3.7: Pipelined security-aware cache based on the ordinary security-aware cache.



3.3 Optimized CAM Decoder, the *L*-Associative security-aware cache.

Figure 3.8: *L*-Associative security-aware cache.

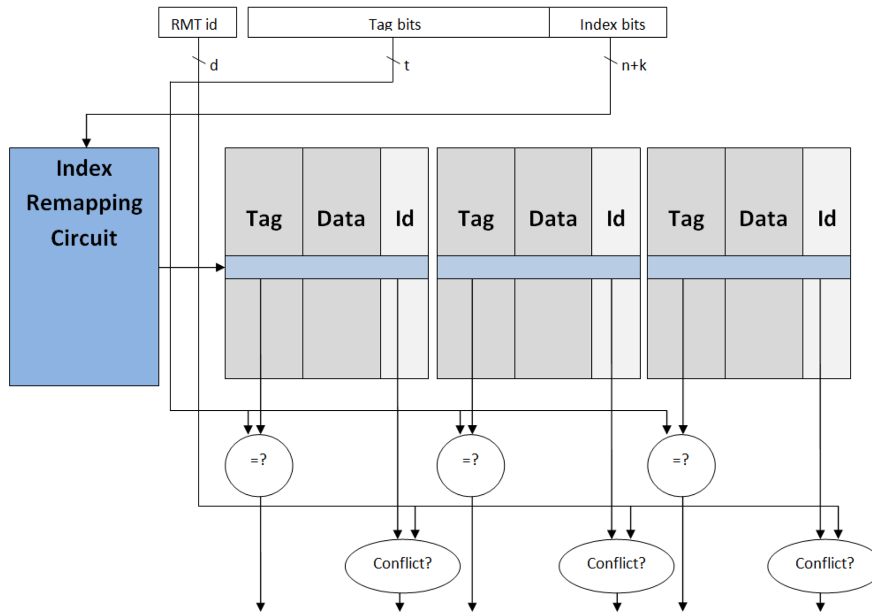


Table 3.3: L-Associative cache overview.

Advantages and Disadvantages	
Can be based either on the pipelined design or ordinary security-aware cache. Shares both the advantages and disadvantages of either.	Not suitable for write-back policy.
Decreased resource usage.	Decreased Hit rate.

Both of the previously described caches can be expensive in terms of BRAMs and LUTs as well as any other resources used to implement CAM. The security-aware cache index remapping circuit can also drastically lengthen the critical path within the cache control. The *L*-Associative security-aware cache (where L is the number of sets) is based on the associative cache architecture where one security-aware cache index remapping scheme (either ordinary or pipelined

cache scheme) is used to map indices in all of the sets. This decreases both the critical path length of the cache $\log L$ and resource usage L times for a fixed size cache, when compared to equivalent capacity security-aware cache. The circuit for this decoder can be seen in Figure 3.8. The cache operation and the explanation behind set invalidation is shown in Figures 3.9 and 3.10.

Overview of the L-Associative security-aware cache

The cache is based on the principle of increasing the block-size. It could be considered as a security-aware cache with *L times larger block-size* but with a small but important difference. The L-associative cache does not load a L times larger block to memory. It only fetches one block of "normal" size from RAM, and stores it within one of its sets. The set that the data is fetched to is determined by modified SecRAND described in Section 3.4.

L-Associative security-aware cache multi-set validation mechanism

As the cache makes use of number of sets, and only one line address decoder, the set validation mechanism is redesigned. The set validation control mechanism can be based on a number of different resources, and is highly dependent on the choice of the underlying CAM based decoder. Nevertheless the scheme is as follows.

Besides having the usual decoder we extend cache with a *valid table*. The valid table stores *sets* (number of sets) bits per line, each indicating whether a set within a line is valid. In the ideal situation when pipelined security-aware cache is used we can easily extend the tag array to incorporate valid tables, making a cheap and fast extension. In case of non-pipelined cache data cache control modification is required in order to manage set invalidation.

Unregistered storage element is applicable for valid table implementation in any CAM based decoder. Registered storage element is not suitable for designs which a user does not want to modify the cache control logic. Users should be aware that the unregistered validation table is an expensive design due to complex routing. The validation bits are being associatively searched for along with CAM what implies a larger CAM. The validation table can be implemented using vendor provided CAM by using do not care bits. Matching is performed on the index bits, while the valid bits are read from the CAM and used to verify valid sets.

Example of L-associative cache write

We have $L = 4$ and $line = 231$ being occupied by $index = 8890$. We need to store a word to $index = 2412$, which is not currently present in the remapping register file. During the store the random number generator provides us with $index = 231$. We invalidate all of the sets within $line = 231$, and change the index of the register to 2412. At the same time we validate the set within the line that we performed the store to. If in the subsequent cycle a write is issued to $index = 2412$, we follow the modified SecRAND and write to one of the sets within the $line = 231$ and update the valid table accordingly.

Figure 3.9: The figure presents set invalidation during 3rd cycle. Line 4 stores now *index* = 77C1DC, and the previous tag H1231241241253 is no longer valid with the current index.

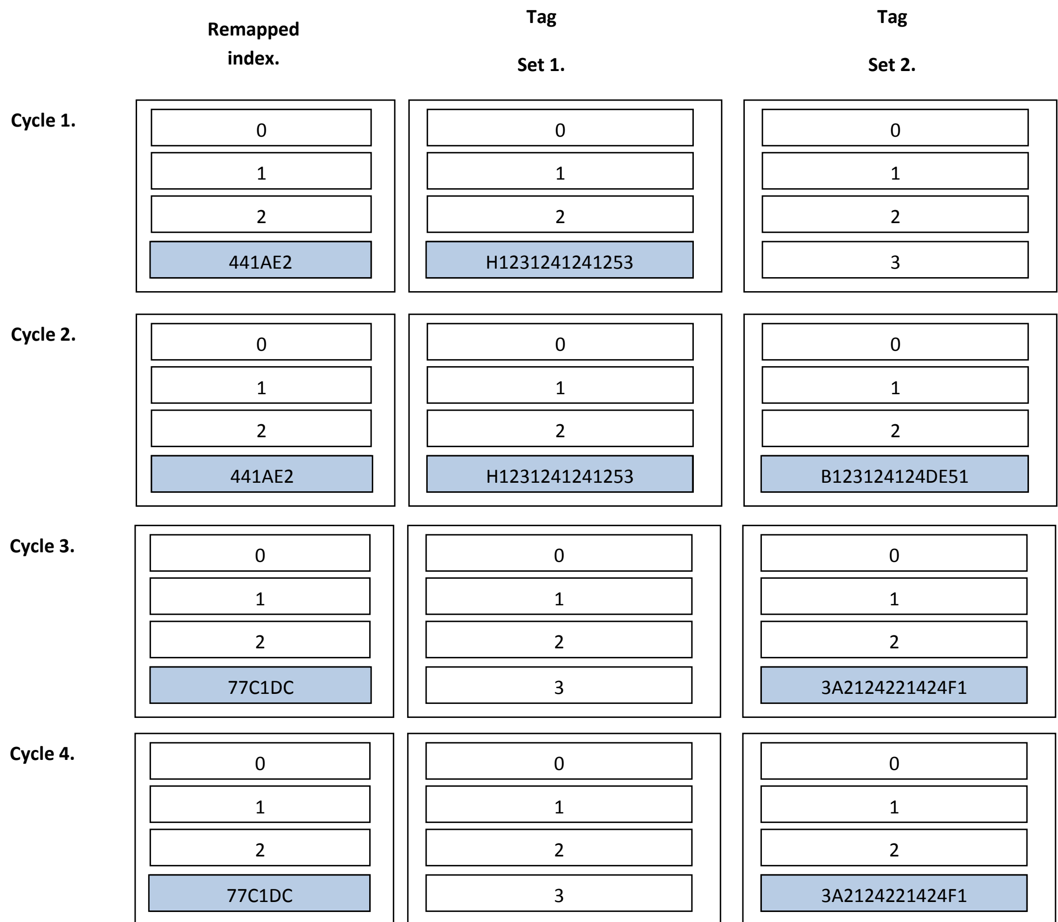
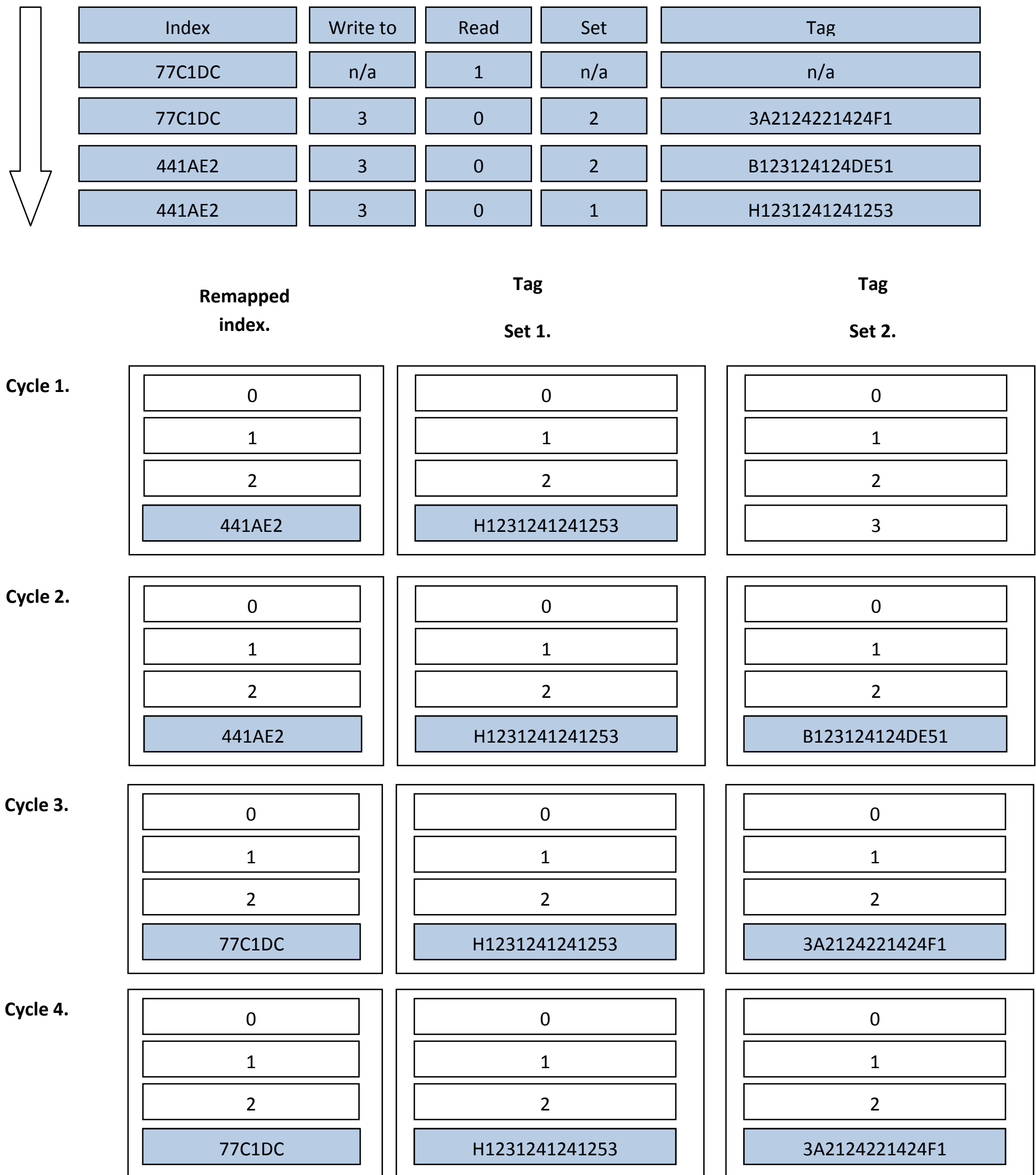


Figure 3.10: The figure presents what would happen without set invalidation. We can see tag H1231241241253 present in 3rd cycle. The cache is wrong state where address with tag H1231241241253 indexed by 77C1DC seems to be valid.



Drawbacks of the *L*-Associative security-aware cache

The drawback of this design is that when a cache line eviction occurs and a remapping register is updated, we invalidate all of the lines that made use of the former index. The hit rate degradation will depend on the application using the cache as well as on the number of sets; the higher the associativity, the higher the miss rate. To visualize it better, the cache has the drawback of larger block-size, without the advantage of smaller *compulsory misses* due to data pre-fetching (it does not make use of the *spatial locality*).

Write-back policy is not recommended for this design. On a cache miss, all of the sets would have to be written back to memory. Depending on the number of misses and CPU:Main memory bus throughput *write-back* policy could cause stalls, or require large buffers.

3.4 Modified Security Algorithm

The replacement policy depends on the decoder we use and on the cache write policy. Both write through and write back policies are supported. If we use write through policy, no dirty block has to be written back to memory during a protected line tag miss.

SecRAND for ordinary security-aware cache

The SecRAND algorithm works as in the original security aware cache [23]. The difference is that context ID bits would be stored and fetched from a resource-wise less expensive tag array.

SecRAND for pipelined security-aware cache

Most of the M cycle CAM memories consist of BRAMs, therefore there is no difference whether the protected bit is stored within the tag CAM line. If we store the *ID* context within the CAM memory, we can detect context miss a cycle earlier. We can act as if it was an index miss. From the attacker's perspective, it looks as if it was an index miss despite the fact that the data accessed from memory would not be written to cache as in the original SecRAND [23].

Modified SecRAND for *L*-associative security-aware cache

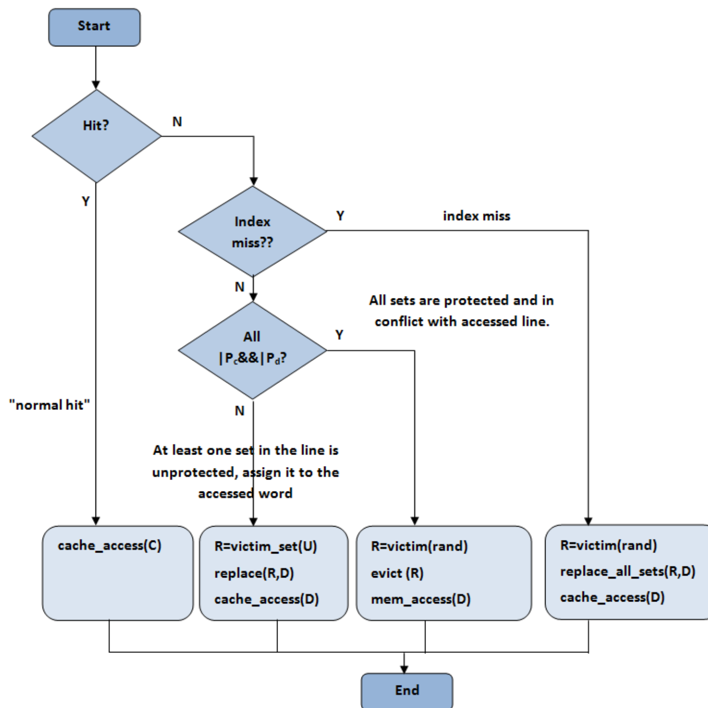
The advantage we have in this case is that there are L sets holding the same index. What happens during a tag miss involving a protected line is different from the original algorithm [23]. Instead of writing to a randomly selected set, we choose one that is unprotected (if there is one) and we write the new tag/data to that set. If all of the sets are used within one line, we operate as if the original SecRAND algorithm is used (with the difference of invalidating one line within all of the sets).

In case we decide to use a multi-cycle CAM in an L-associative security-aware cache, the *ID* bits for every set have to be either stored in CAM memory along the index remapping register, or in the tag line of every set. The first implementation allows us to detect a context conflict one cycle earlier.

Table 3.4: The table describes the notation used in the modified associative SecRAND algorithm.

C	The cache line selected by the address decoder (during a cache hit or a tag miss).
D	The memory block that is being accessed.
R	The cache line selected for replacement (victim).
P_x	The protection bit of X. If X is in a cache line, it is the P bit of the cache line. Otherwise it is determined by the PP bit of the page/segment that X belongs to.
cache_access(C)	Access C as in a traditional Direct Mapped cache.
victim(C)	Randomly select any one out of all possible cache lines with equal probability.
victim(rand)	Replace R with D, update LNreg.
replace(R,D)	Write back R if it is dirty; invalidate R.
evict(R)	Write back R if it is dirty; invalidate R.
mem_access(D)	Access to D without caching it.
U	A set with non conflicting address.
victim_set(U)	Select U as the victim set to be replaced.

Figure 3.11: SecRAND for *L*-associative security-aware cache.



3.5 Design Portability

Design portability has been achieved by abstracting the architecture from its implementation. Depending on the technology used, the cache can be implemented across different vendors FPGAs. The choice a user has to make is whether the cache is going to be used in pipelined mode, associative mode or both. Given that appropriate CAM libraries are often provided for a given platform, any of our cache designs is efficiently portable onto any FPGA chip. In case platform specific CAM is not available, a flip flop based CAM memory can be used. This also holds for our pipelined security-aware cache.

3.6 Summary

This Chapter described the novel aspects of the design and the problems they address. The reader should be accustomed with the following ideas before further reading.

- *Security-aware cache* - Different building blocks of FPGA and ASIC design, identifying the complications and differences.
- *Pipelined security-aware cache* - Solution to the increased critical path. The cache is based on the idea of a pipelined cache design, in separate CAM search circuit from cache control. The cache should be used in two stage mode. During the first stage CAM memory is used to remap indices. During the second stage Tag/Data access is performed. The user has a large freedom when choosing the underlying CAM. Single cycle read CAM is recommended. X-cycle read CAMs can be used although result in more complicated pipeline and are likely to degrade performance.
- *L-associative cache* - An answer to FPGA CAM high resource demand. The cache makes use of modified SecRAND. It degrades performance but improves resource utilization.
- *Modified Security Algorithm* - Modified algorithm addressing the new decoders. The algorithm changes when we use of L-associative security-aware cache. Similar to security-aware cache with L times larger block-size, but not making use of *spatial locality.*
- *Design portability* - The design portability is reached by use of generic resources as well as abstraction of CAM implementation (prominently the case when pipelined security-aware cache is used.

Chapter 4

Cache implementation and Leon 3

In this Chapter we discuss the implementation issues of the designs described in Chapter 3 as well as our attempt to modify Leon 3 processor. We present methodology of our work as well as achievements and findings. Currently we managed to implement non-pipelined security aware cache (Both associative and non-associative versions), with several restrictions. We used the platform in order to identify potential problems arising from various security-aware cache designs.

Section 4.1 is an overview of the design choices. In Section 4.2 we discuss the different stages in which the Leon 3 was modified.

4.1 Design choices

The Leon 3 modification is an evaluation platform and is not yet fully validated and for the time being will not be released. The processors data cache is modified to the L-associative and ordinary security-aware cache standard with some restrictions. We extended the processor with the most performance and resource-wise crucial part of the design *the remapping register file*. This provides us with the information on the potential costs and benefits of the new cache. The full SecRAND algorithm implementation requires trivial modification (discussed later).

4.1.1 Why Leon

We wished to show that our security-aware cache works, and it can be as fast as a direct-mapped cache. We needed a platform (soft processor and its utilities) that would provide us with tools to compare ordinary, associative caches and our cache. We required the platform to be portable and easily parametrizable.

When choosing a soft processor we have mainly taken into account the associated tools. Without appropriate generation and optimization tools, the generated processor would not have superior performance. The qualities desired are good support, a broad range of tools and simplicity of design as well

as documentation. The Leon 3 processor is chosen since it meets the above requirements. It is publicly available, and is relatively simple.

Although initially a simple processor extended with security-aware cache would be an easier platform to work on, this approach would have a number of drawbacks:

- The design of appropriate tools would be extremely time consuming, and the exact scope of required tools can never be determined. Putting 90% of available time into building the processor associated tools would be a reasonable estimate. Clearly, this would restrict the amount of time we would be able to invest into extending the processor with security-aware cache.
- The results obtained by builds of various configurations of a simple processor would not be as informative as numbers got on a platform as widely recognized as Leon 3. 2000 extra LUTs mean nothing on a soft processor, 2000 extra LUTs on Leon 3 with a specific design give an insight into how expensive the security-aware cache can be.
- The security-aware cache is an answer to an existing problems, and only by modifying an existing and acclaimed design we get feedback on the potential implementation difficulties. The platform is expected to be used in more advanced work. Performing a cache timing attack requires a real system, one which consists of an operating system, FPGA chip, boards, ram modules etc. Putting all of the components together is an extremely time consuming project, Leon 3 has all of that.
- Although when designing a new system from scratch we would have the freedom to design it without the constraints which made some modifications of Leon difficult, we were not aware of them when the project was launched. Therefore it is very likely that the platform would have to be redesigned, and as such this approach was of no advantage to us.
- Leon 3 is already portable across a vast number of platforms. It is one of the most portable designs available.

4.1.2 Random number generator

For the purpose of simulating and building our design we used *linear feedback shift register* (LSFR) method [5]. This uniform random number generator should provide good quality uniform numbers, and be efficiently implementable on FPGA. As the performance of cache is dependent on quality of uniform random numbers, in future we plan to use a different RNG for benchmarking purposes. LUT-Optimized [13] random number generators would also be suitable for this design. It is cheap LUT-wise, allowing index remapping registers to utilize a large amount of LUTs. The way the current system is modified, exchanging random number generator is equivalent to reconnecting one signal and is not prone to cause any problems.

4.2 How we modified Leon

Before working on the modification we identified and modelled the Leon 3 file and component structure as well as the signal interconnections. Although seemingly an unimportant step, it allowed us to narrow down the extent of required modifications. We also got accustomed with Leon 3 suite, as well as commented the code and used reverse engineering techniques (as the documentation on a number of components was scarce) to model their behaviour.

We approached the Leon 3 modification in a number of testable stages. Every next stage was a solution to the problems that raised in the previous one. At each stage we aimed to verify the resource and timing requirements of each of the cache designs. After evaluating a number of different processors we decided to use Leon 3 as our evaluation platform. The stages are arranged in a chronological order. We present the stage overview:

- *Stage 1.* - Identifying the performance crucial components. (Subsection 4.2.1.)
- *Stage 2.* - L-associative security-aware cache. (Subsection 4.2.2.)
- *Stage 3.* - Extending Leon 3 with pipelined security-aware cache. (Subsection 4.2.3.)
- *Stage 4.* - Enabling Full SecRAND. (Subsection 4.2.4.)

4.2.1 Stage 1. Identifying the performance crucial components

Our next step was to identify the performance crucial components of the new cache and prioritize them accordingly. The most important part of security-aware cache is the remapping register file. CAM memories are very expensive to implement on FPGA and bascially they are the remapping register file.

In order to implement the remapping register file we decided to make use of a cut down Leon 3 core. We based our design on Leon 3 with MMU disabled. As the remapping register file has unregistered outputs we were able to keep the processors data-path unchanged, allowing us to shorten the testing cycle substantially. We decided not to use MMU as we tried to keep the design as simple as possible to narrow down the number of possible bugs. Keeping the core simple decreased the simulation times substantially. Our approach allowed us to implement part of the cache in very short time and reveal a number of challenges. We did not aim to implement full security-aware cache at this point.

Goals for stage 1:

- Identification of Leon 3 components crucial to the security-aware cache.
- Getting accustomed with Leon 3 and SPARC documentation.
- Extending Leon 3 processor and identifying potential problems with remapping register file.

In order to decrease the modification, validation and test cycle length we imposed several constraints on the Leon 3 functionality:

- Cache snooping cannot be used.
- Instruction cache is not security-aware. The reason is as in the two previous cases. In addition, extending instruction cache is not necessary.
- MMU is disabled.
- Only remapping register file is installed.

Efficient Index Re-mapping Based on CAM, first approach to Leon 3 processor

We aim for a portable design. Depending on the vendor and the chip model, FPGAs vary significantly in terms of ratio of BRAM:LUT:DSU. The building blocks are arranged in a different way on every chip which can hinder performance. Vendors offer libraries that can be used to increase efficiency of the design. The key to a secure, efficient and portable design lies in abstracting the design from its implementation, by identifying the components for a cache. For security-aware cache, this is the index remapping circuit, tag and data array and random number generator.

CAM memories (the most important part of our design) have been implemented in several different ways across a wide range of platforms, each implementation having different pros and cons. This is the first reason behind customizing caches based on different index remapping circuits, to maximize the potential of different platforms. The second reason is to vary the degree of modification required from existing caches to match our design. There are a number of techniques available to implement a CAM (LN registers file) on an FPGA(as mentioned in Chapter 2). We approach several different CAM designs, and all of them pose different challenges:

- Generic slow multi-cycle designs based on BRAM, and fast multi-cycle designs based on LUTs.
- Fast vendor specific designs based on BRAMs, SLR (shift registers) etc.
- Very fast and area efficient designs based on LUT.
- Simple register files (resource wise very expensive).

The first implementation is not considered as read/write access time is excessive, in dozens of cycles. The third CAM implementations are not considered as they either require off chip software to be updated (The CAM is written to using off chip software [22]).

For the single cycle cache decoder, we used the 4th type of CAM memory listed above. The characteristics of the cache based on this design are portability and not requiring a complex state machine, as the index remapping circuit makes it possible to perform a single cycle read and write operation to data/tag arrays. A number of soft processors (Leon 3 in particular) have complex state machine. Integrating the new cache could potentially be time consuming if a multi-cycle design is used, and would degrade performance due to enforced stalling/cache write dropping mechanisms.

This cache has the drawback of being resource demanding (it uses LUT/Flip-Flops instead of BRAMs) and having a longer critical path. Logically the decoding combinational circuit becomes part of the cache control instead of being a simple array index decoder(explained later in the Section).

Abstracting the implementation

In order to keep the testing suite and all current software valid we extend the processors data cache in a way which is completely abstract from the Leon 3 programmers point of view. Furthermore to decrease the amount of our interference within the Leon 3 design we use the k parameter to decrease the size of cache instead to increase the range of indices. For example to get a 4 KB cache with $k = 1$ we should set cache size to 8 KB and $k = 1$. The k parameter decreases the size of cache by 2^k. The functionality is the same, but from engineering point of view it allowed us to simplify the design. To the processor the tag/data arrays appear as if they were of the index appropriate size.

Listing 4.1: The tag/data array wrapping.

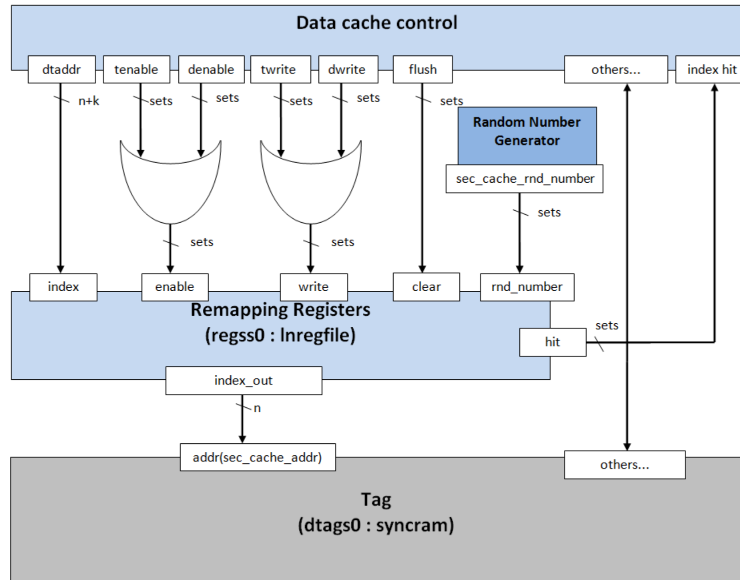
```
1     dme : if dcen = 1 generate
2
3        -- dont use the security-aware cache
4         dtags0 : if ( not DSECCACHE) and (DSNOOP = 0) generate
5             dt0 : for i in 0 to DSETS-1 generate
6                 dtags0 : syncram
7                 generic map (tech , DOFFSET_BITS, DTWIDTH)
8                 port map (clk , dtaddr, dtdatain(i)(DTWIDTH-1 downto 0),
9                 dtdataout(i)(DTWIDTH-1 downto 0), dtenable(i), dtwrite(i
                      ));
10            end generate;
11        end generate;
12
13        -- use security-aware cache
14        dtags2 : if DSECCACHE and (DSNOOP = 0) generate
15
16                -- the remapping register file (lnregfile) has to be
                      updated whenever a write to tag/data array occurs.
17                ln_sig0 :for i in 0 to DSETS-1 generate
18                    sec_cache_enable(i)<= dtenable(i) or ddenable(i);
19                    sec_cache_write(i) <= dtwrite(i) or ddwrite(i);
20                end generate;
21
22
23                regss0 : lnregfile
24                generic map (n=>DOFFSET_BITS-DLNREGS_BITS, k=>
                      DLNREGS_BITS, sets=>DSETS)
25                port map (
26                    index       =>dtaddr,
27                    write       =>sec_cache_write,
28                    clock       =>clk,
29                    hit         =>index_hit,
30                    index_out   =>sec_cache_addr,
31                    rnd_number  =>sec_cache_rnd_number,
32                    clear       =>crami.dcramin.sec_cache_clear,
33                    enable      =>sec_cache_enable
34                );
35
36
37            dt0 : for i in 0 to DSETS-1 generate
38
39                dtags0 : syncram
40                generic map (tech , DOFFSET_BITS-DLNREGS_BITS, DTWIDTH)
41                port map (clk , sec_cache_addr,dtdatain(i)(DTWIDTH-1
                      downto 0),
42                dtdataout(i)(DTWIDTH-1 downto 0), dtenable(i), dtwrite(i
                      ));
43
```

```
44              end generate;
45          end generate;
```

Listing 4.2: The data cache control logic only modification.

```
1   if DSECCACHE then
2       if (dcramov.tag(i)(TAG_HIGH downto TAG_LOW) = dci.
            maddress(TAG_HIGH downto TAG_LOW)) then
3               -- Tag hit
4           hitv(i) := dcramov.index_hit(i);
5       end if;
6   else
7       if (dcramov.tag(i)(TAG_HIGH downto TAG_LOW) = dci.
            maddress(TAG_HIGH downto TAG_LOW)) then
8               -- Tag hit
9           hitv(i) := '1';
10      end if;
11  end if;
```

Figure 4.1: Diagram visualising the Leon 3 data cache modification



The data cache state machine holds for both the modified and non-modified Leon 3. We tried to minimize Leon 3 modification in order to keep its behaviour as close to the original design as possible. This approach allows using standard testing suits, and decreases debugging time due to small scale of code modification. Using registered multi-cycle read CAM memories would require us to create an extra stalling states while waiting for CAM read operations to finish. This could also cause a number of potential problems with the processor control unit. Under this approach, cache can be implemented on any platform.

This design has the advantage of being portable. Register file can be implemented on any FPGA. In case a cache has to be defended against cache timing attack, this architecture can be used to protect it. After the design has been successfully modified, the user could optimize the LN register file to best fit the

target platform. Potentially the register based CAM could be replaced with a vendor provided CAM with the option of not registering output. Although the resource usage might be decreased it is very likely to suffer from elongated critical path.

Indeterministic cache behaviour

While installing the remapping register file we found out a potential complication of this extension. SPARC v8 instruction set allows separate data cache tag and data cache data reads and write using the privileged **LDA** instruction with **ASI** 0xe and 0xf respectively. In order to independent cache tag/data write the remapping register file has to be updated when either the data is written to the tag or data array. Users should bare in mind that has implications on the cache behaviour. For example in the following situation where the data cache has just been flushed and index(*addr1*) = index(*addr2*).

1. **ST** *addr1*, *reg1*
2. **ST** *addr2*, *reg2*
3. **LDA** index(*addr1*), *reg3*, 0xf

User would expect to get the data stored under *addr2* when issuing the **LDA** instruction. This might not be the case as storing the second word to cache has a probability of 1/cache size of being stored in the same line as word with *addr1*. This should not cause any complications, as separate cache tag/data writes are only used for diagnostic reasons, nevertheless potential users should be aware of the problem. A possible way to counter it would be to supply deterministic numbers instead of random number to the remapping registers file in case of diagnostic access.

Increased critical path

The critical path was increased due to remapping register file becoming part of the cache control circuits. The critical path is illustrated in Figures 4.2 and 4.3.

4.2.2 Stage 2. *L*-associative security-aware cache

At the end of Stage 1. we identified a number of potential problems with the security-aware cache.

The two problems were:

- Increased resource usage
- Elongated critical path

Although initially we struggled to find solution to the first problem, we came up with one while running a number of builds to verify cache parameter impact on Leon 3 resource utilization. The clue lied within the block-size. A larger block-size implies a smaller number of remapping registers. This is the case

Figure 4.2: Leon 3 7-stage pipeline.

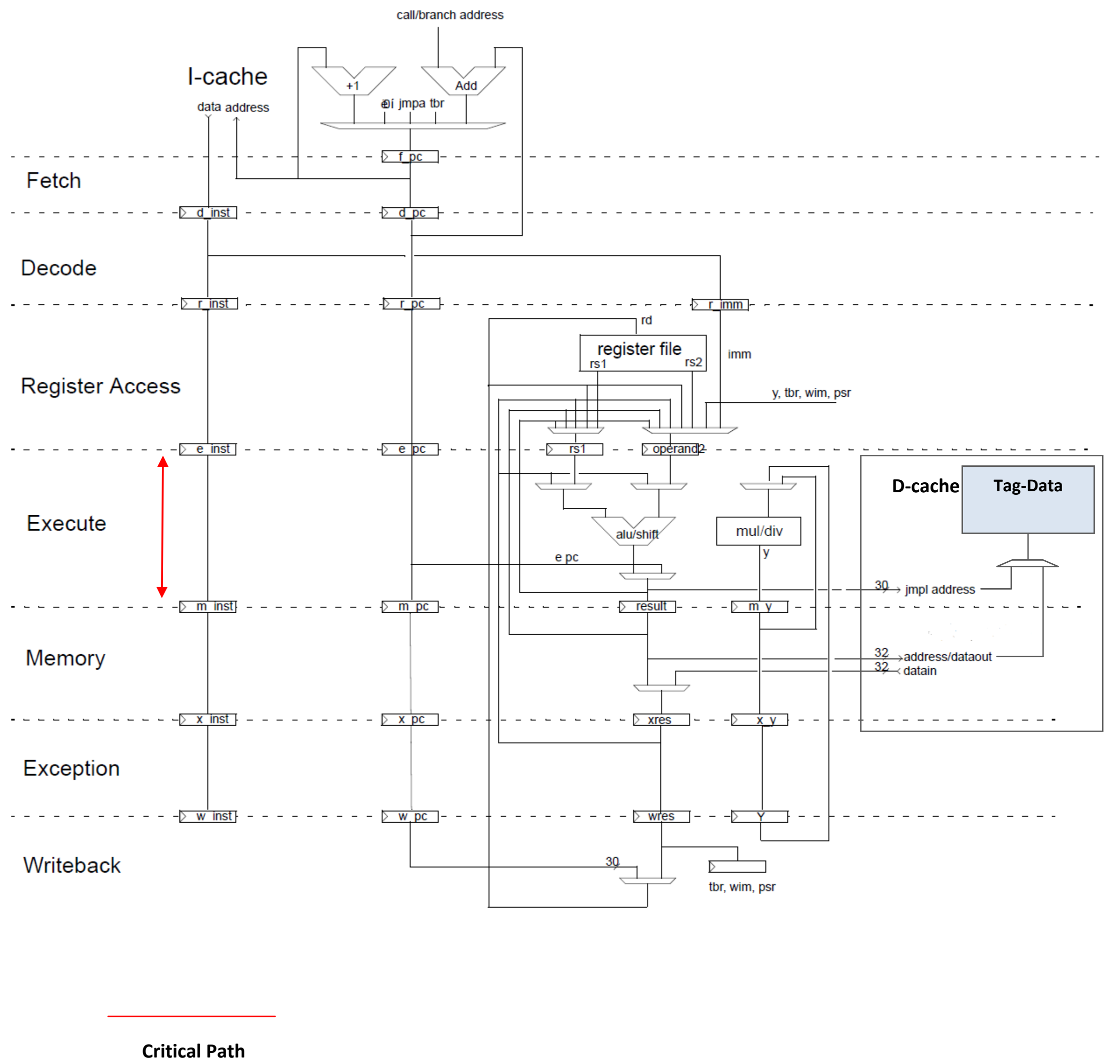
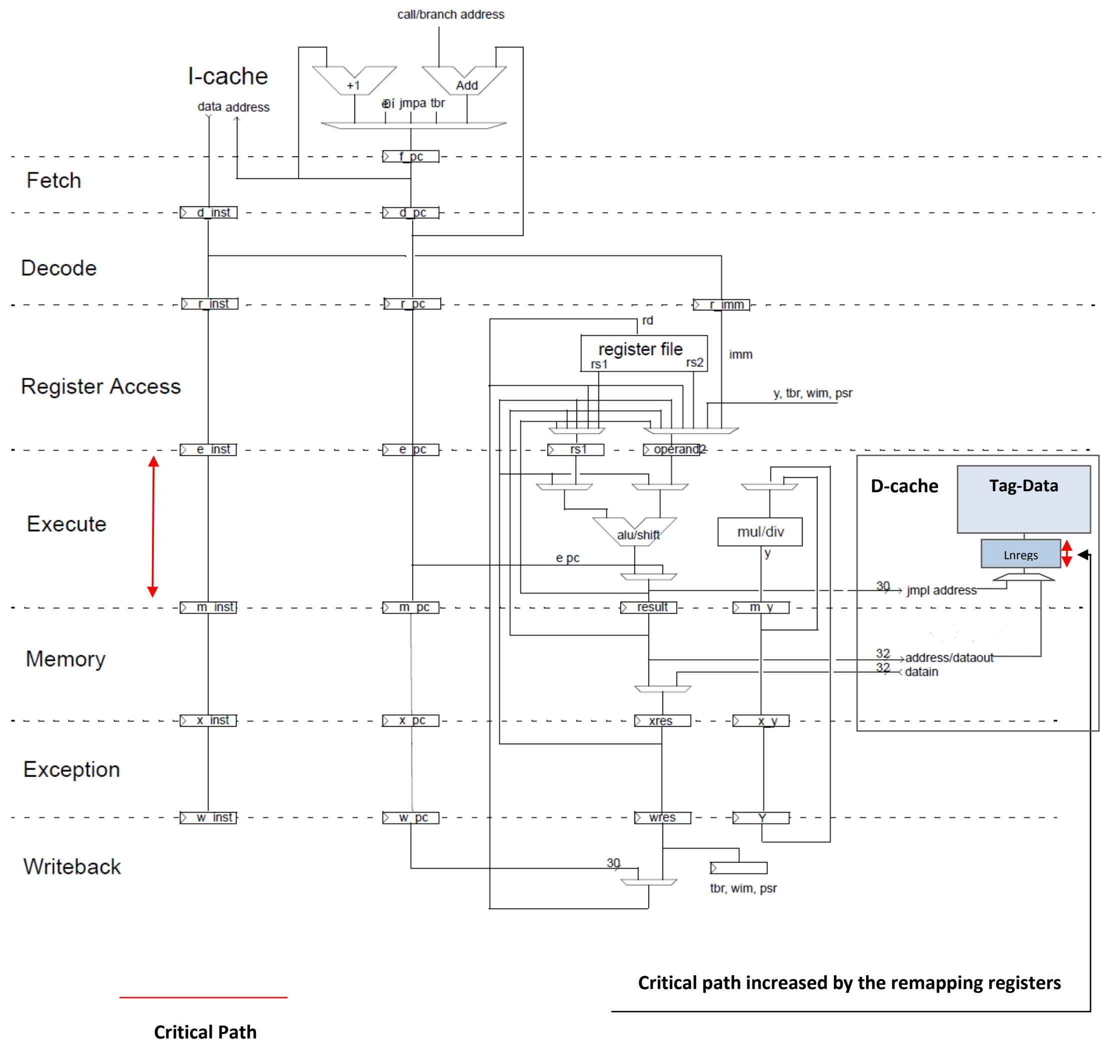


Figure 4.3: Leon 3 extended 7-stage pipeline.



as each cache line stores more words, and we need one remapping register file for each line. Therefore smaller amount of lines results in a smaller number of remapping registers.

There are a number of problems with increasing block-size.

- Larger miss penalty on a cache miss
- Higher memory bandwidth requirement
- Larger memory buses.

Although in ASIC larger memory buses are expensive, the impact is not as severe as it is the case in FPGA. Bus routing and multiplexing can become very expensive. We came up with the idea of L-associative security-aware cache which closely resembles larger block-size, but with a number differences (Chapter 3). The cache performance is carefully evaluated in Chapter 5.

Our goal for this section was to formalize and implement the L-associative cache architecture.

Validation table design choices

As discussed in Chapter 3 we came up with two possible ways on implementing validation tables. Currently we decided only to make use of the unregistered validation table, which in a lot of cases nullifies the potential benefits of the cache due to similar resource requirements as LUT based CAM memory. Unfortunately this project had a limited time-scale and we had a strong belief that registered validation tables would be very resource efficient, but we were not sure about our predictions on unregistered tables. We decided to follow the second approach.

There were two main reasons behind that decision

- Validation tables implemented with tag array require cache control modification
- When implementing validation tables using BRAM the resource usage can be easily estimated. We had very rough idea of performance impact of unregistered validation tables on L-associative security-aware cache.

We implemented the associative cache using existing cache set control logic. We did not implement the associative SecRAND algorithm.

Cache implementation: L-associative Security-aware Cache

Making a generic and efficient CAM based design is difficult using modern FPGA chips. This is the reason behind creating L-associative security-aware cache with one remapping register file for all the sets. We distinguish this design from the traditional N-associative cache as the L parameter has a very different impact on cache performance. This cache can be based on either our

security-aware cache or the pipelined security-aware cache, sharing the advantages and disadvantages of the corresponding design.

The main difference between the L-associative security-aware cache and other caches lies in resource utilization. When we are short of LUTs or BRAMs, we might consider using one decoder for a number of sets. Obviously one would suffer from a higher miss rate (discussed in Chapter 5) as all sets are invalidated on an index miss, but the size of the decoder decreases linearly with the number of sets. Furthermore when we use a single cycle decoder, the critical path length decreases as well. We need to compare L times fewer elements. Since the CAM decoder is based on a binary tree structure (elements are compared using binary tree structure), we can expect logarithmic decrease of the decoder's critical path length(in case pipeline stage includes index decoding) and linear decrease of resource usage compared to a basic cache.

4.2.3 Stage 3. Extending Leon 3 with pipelined security-aware cache

The pipelined security-aware cache was our answer to the increased critical path problem.

The main idea behind the pipelined security-aware cache is to decrease the critical path length. The way to separate search from control is to use a single cycle CAM memory. In ordinary security-aware cache, read is performed in the same cycle as index remapping. In the pipelined cache, we separate the index remapping stage from data/tag access. Obviously to access data and tag arrays, we need to index it correctly during the same cycle as read/write is performed. We achieve this by inserting extra registers, effectively creating a pipelined design. The pipelined cache can be implemented using the register based CAM as mentioned in Chapter 3. Registering inputs to the single cycle index remapping circuit provides us with a single cycle write and single cycle read decoder ($M = 1$).

This cache has the disadvantage of being more difficult to implement than the non-pipelined cache. At the same time it can easily utilize a number chip specific CAM, usually with better speed/power/area characteristics, while decreasing the critical path length. It also allows us to make use of registered BRAM based validation tables, which are faster and less resource demanding the LUT/flip flop based. We have not yet finished implementing the pipelined security-aware cache (discussed later in the Chapter), although the design is in a mature state and the current results show a lot of promise (Chapter 5).

There are a number of reasons why extending Leon 3 with pipelined security-aware cache is more difficult than non-pipelined version. The exact details are irrelevant, the concept is important. The *jmp address* that can be seen in Figure 4.4 is used to select the lines within the tag/data arrays. It is not registered, and it is one of the signals sued to access data/tag arrays. If not for that fact, the cache could be considered pipelined as index is dependant on two registered inputs which are multiplexed using a registered value; the path overhead is very little. Our current idea is to insert an extra index decoding stage after execution stage. From our current estimates based on registering inputs this should allow us to raise the modified Leon 3 operating frequency to match that of original Leon 3. Although it is very likely that the problem has a neat and simple solution, currently there is a number of issues that we are trying to solve.

The concept of extra stage is illustrated in Figure 4.4. It is possible that the approach will lower the processors CPI.

4.2.4 Stage 4. Enabling Full SecRAND

Due to limited time scale of the security-aware cache project we had to postpone the implementation of pipelined security-aware cache. Our next approach to security-aware cache implementation was installation of full SecRAND algorithm. We had two possible ways of approaching this task, one was to enable MMU and extend it with SecRAND. The other one was to modify non-MMU Leon 3 and modify one of the current Leon 3 supported operating systems.

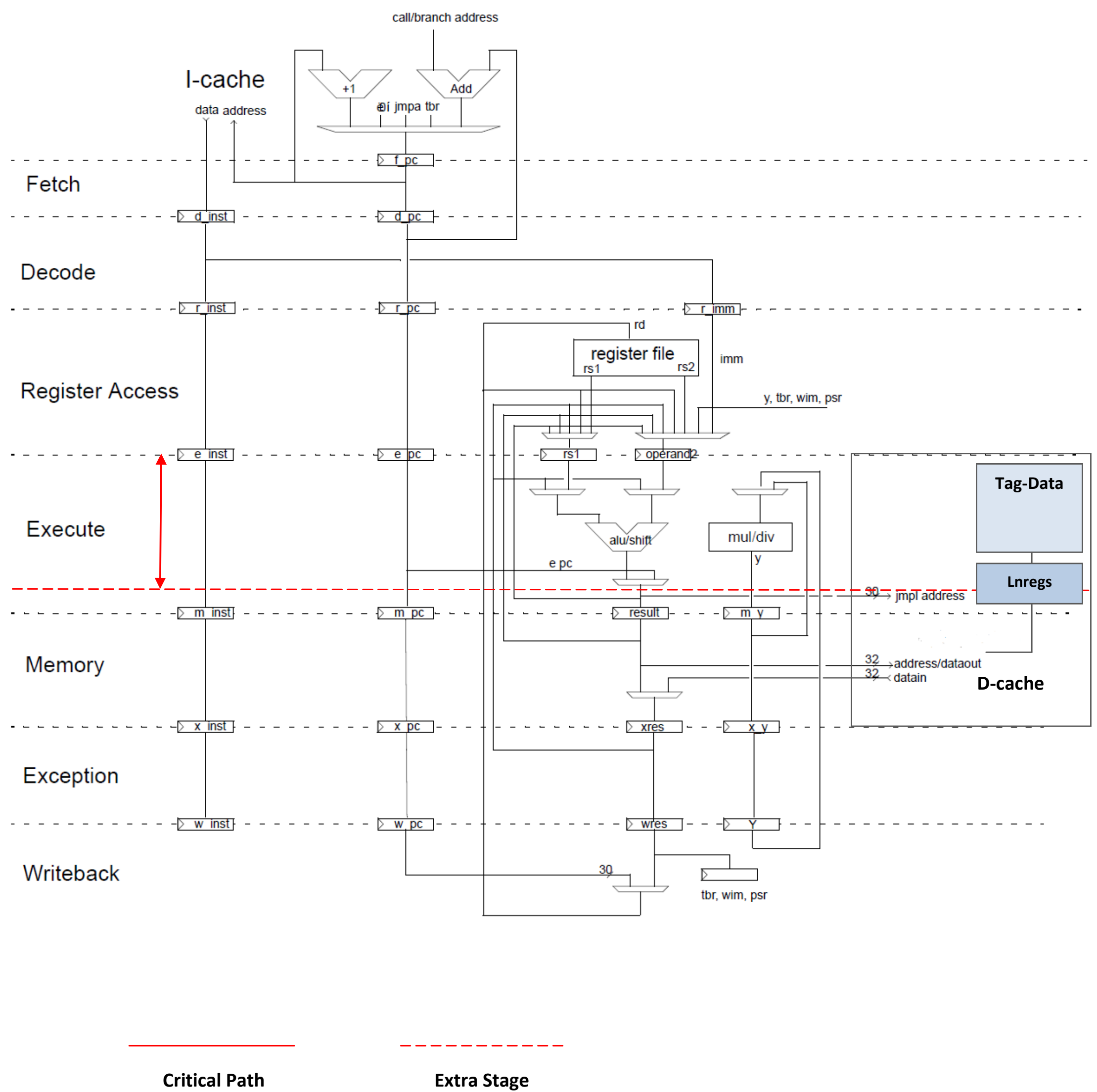
Non-MMU Leon 3

> "Does uClinux support multitasking? What limitations are imposed by not having a MMU?
>
> A. uClinux absolutely DOES support multi-tasking, although there are a few things that you must keep in mind when designing programs...
>
> 1. uClinux does not implement fork(); instead it implements vfork(). This does not mean no multitasking, it simply means that the parent blocks until the child does exec() or exit(). You can still get full multitasking.
>
> 2. uClinux does not have autogrow stack and no brk(). You need to use mmap() to allocate memory (which most modern code already does). There is a compile time option to set the stack size of a program.
>
> 3. There is no memory protection. Any program can crash another program or the kernel. This is not a problem as long as you are aware of it, and design your code carefully.
>
> 4. Some architectures have various code size models, depending on how position independence is achieved." [6]

In order for security-aware cache to work operating system and hardware cooperation is required. Operating system is responsible for separating process by context switching. Without MMU, which in SPARC v8 is responsible for context management, two different process can operate by interleaving. This concept can be clearly seen in the above quote taken from uClinux *frequently asked questions* (FAQ).

Currently full SecRAND is not implemented when MMU is disabled. In order to use full SecRAND replacement algorithm *operating system* (OS) cooperation is required, and due to the limited time scale of the project as well as its different scope we decided to leave this part of the design out. Full SecRAND algorithm would require, depending on the operating system used, a context switch instruction to be executed whenever a new untrusted thread would be initialized by kernel. Although trivial concept the validation phase is very likely to become time consuming. For simplicity and efficiency we recommend adding an **STA** instruction to a new status register (ID register) whenever fork is executed. Currently the OS keeps track of the process ID, and a single **STA** instruction to a processor status register has an negligible execution time and

Figure 4.4: Leon 3 7-stage pipeline. The extra stage is marked with a red dashed line.



no impact on the application performance. SPARC v8 ISA offers a number of address spaces which are undefined in Leon 3 and would be suitable for the implementation.

The required hardware modification consists of:

- Extending tag array with context ID fields.
- Adding an extra context control register. Our idea is to make use of **SDA** instruction(SPARC v8 alternate address space store instruction) and a context register to change the context within which processor is operating.
- Extending cache control logic with context matching mechanism, a simple comparator of current register state and the output from the tag.
- Whenever a context miss is detected, we issue a force missed. Fetch necessary data from memory, and write an invalid tag to a randomly chosen cache line. The cache control logic requires very little modification(couple lines of code), as most of the required functionality is already implemented. Nevertherles a number of currently undefined problems can arise.

The only question we ask is whether work on implementing security-aware cache in MMU disabled Leon 3 going to be fruitful. Without MMU and under the uClinux thread model in most scenarios it would be much easier (and more efficient) to flush cache after every *vfork()* call. When MMU is not used, the context switches are going to be less frequent, therefore the impact of cache flushing on processor should not be microscopic. Flushing the cache would hide any potential timing information leakage. In case higher hit rate is required the remapping register file would be sufficient on its own.

Enabling MMU

With MMU enabled, the context mechanism is already implemented within Leon 3(it is part of the SPARC v8 ISA). Furthermore no software modification is required.

> "A SPARC Reference MMU provides three primary functions: 1) It performs address translation from the virtual addresses of each running process to physical addresses in main memory. This mapping is done in units of 4K-byte pages so that, for example, an 8-megabyte process does not need to be located in a contiguous section of main memory. Any virtual page can be mapped into any available physical page. 2) It provides memory protection, so a process cannot read or write the address space of another process. This is necessary for most operating systems to allow multiple processes to safely reside in physical memory at the same time. 3) It implements virtual memory. The page tables track which pages are in main memory; the MMU signals a page fault if a memory reference occurs to a page not currently resident." [7]

By examining Leon 3 data cache code and SPARC v8 manual it can be clearly seen that the required context collision mechanism is part of Leon 3, the

only difference is that context miss have to be differentiated from other misses. The main advantage behind the MMU version is that it requires no software modification. This does mean, that in theory all current Leon 3 software could be defended against timing attacks. As a research platform, although more complicated and difficult to use, would give a great feedback on real systems performance.

There are a number of problems associated with context misses. Although paper by Wang and Lee describes the new security-aware cache architecture and a new cache replacement algorithm, it does not discuss the issue of how to manage different contexts. This might have a severe impact on cache performance. Our current approach is that there is no mutual trust between context. Any context mismatch results in a context miss. The issue is discussed in a bit more detail in Chapter 6.

Currently the MMU enabled version is not yet fully verified.

The L-associative SecRAND algorithm would require substantial modification to the Leon 3 MMU set replacement circuits. Therefore a version of ordinary security-aware cache is being used (We do not make use of the possible performance advantage by checking which sets are in conflict with the currently access word).

4.3 Summary

This chapter was an overview of the Leon 3 modification process. We approached the modification process in four stages:

1. Identifying the performance crucial components. The problems we discovered; High resource requirement and increased critical path.

2. L-associative security-aware cache. We implemented the unregistered validation tables, as we wanted to investigate the performance and resource utilization impact.

3. Extending Leon 3 with pipelined security-aware cache. We did not finish the pipelined approach. We currently believe that adding an extra stage to Leon 3 pipeline is the way to implement pipelined security-aware cache.

4. Enabling Full SecRAND. There are a number of questions that need to be answered, the most important being context management. Which context trust each other? The modified Leon 3 with MMU needs further verification.

Chapter 5

Functional and Performance Evaluation

In this chapter we evaluate both our implementation and design. Section 5.1 evaluates L-associative cache hit rate. In Section 5.2 we present a number of builds along with comments on our design resource usage and performance. Section 5.3 compares our cache with other pipelined FPGA caches. Last Section 5.4 includes comments on the security features of the new design.

5.1 Performance Cache Hit Rate

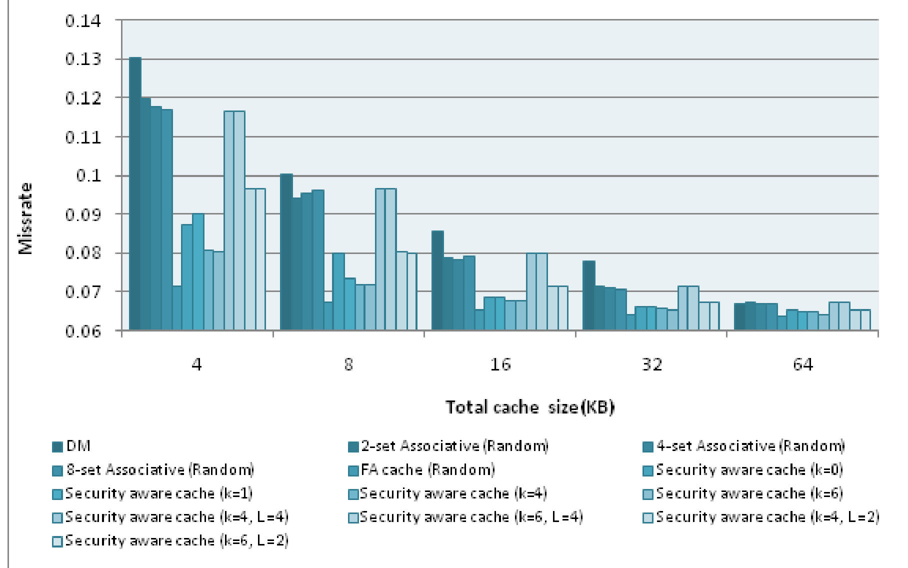
We follow the standard convention of compulsory, capacity and conflict misses. We define a *context* miss to be one that occurs when there is a conflict of context between two accesses. This can have a huge impact on cache performance (in terms of hit rate) in case two or more conflicting applications run in parallel. We discuss cache performance in two different environments. An ordinary environment when there are few processes requiring protection against timing attacks, and a highly-secure environment when the security-aware cache would suffer from a large number of context misses.

5.1.1 Ordinary environment

We analyse our L-associative security-aware cache since our pipelined and ordinary security-aware cache offer the same hit rate as the original security-aware cache [23]. The L-associative security-aware cache always has a higher miss rate than the equivalent capacity and k size security-aware cache due to all set invalidation (compulsory miss causes between one and L evictions instead of 1). Conflict and capacity misses are on a par with other security-aware caches. The hit rate drops with associativity (the opposite of what happens in traditional caches), as although the number of conflict misses decreases, the cache set invalidation effectively cancels it out.

The cache can be viewed as security-aware cache of block-size L times larger than security-aware cache but without the possibility of fetching L blocks at a time. Contrasting the two caches the number of L-associative cache compulsory miss > L times larger block-size security-aware cache compulsory misses due

Figure 5.1: Security-aware cache miss rates with different architectures.



to single block fetching (L times smaller block-size). At the same time cache suffers from the same performance degradation, a full line consisting of L blocks is invalidated. All other cache parameters that affect hit rate are equivalent to L times larger block-size security-aware cache. Therefore an equivalent size security-aware cache with L times larger block-size is an upper bound on caches hit rate.

5.1.2 Highly-secure environment

There are certain applications where there are a number of process/threads running in parallel that do not trust each other, so have to be protected against each other in case of a timing attack. The security-aware cache performance suffers heavily in this environment. Context misses would be common, which means that a lot of blocks would be accessed directly from memory at the same time evicting large numbers of valid lines. The security-aware cache should perform worse than the L-associative security-aware cache, as the latter can be used to store up to the L conflicting context lines. This is the same concept that lies behind N-associative cache applied to context misses instead of conflict misses. As the principle is the same, the speed-up for a highly secure environment (processes only trust themselves) should be similar to speed up achieved by increasing ordinary cache associativity.

5.1.3 Simulation

We use functional simulation to compare different cache architectures in a non-secure environment, with a benchmark based on Jacobi sparse matrix multi-

plication. The results are shown in Figure 5.1. Results from traditional cache and non-associative security-aware cache match the results obtained for the security-aware caches. Our main concern is to validate the principles, not the fine details; as such we decide not to extend CACTI or other more advanced simulators. Our security-aware cache and pipelined security-aware cache follow the same principles as the original security-aware cache [23]. However the L-associative cache differs from other implementations substantially. The cache miss rate grows along with the L parameter. The L-associative cache k parameter has a small impact on cache performance for $k > 1$. The security-aware cache reduces conflict misses, but only when single line is evicted on a miss and the L associativity increases the penalty for the conflict misses nearly L fold. The experiment has shown that conflict misses are more common due to line eviction in L-associative cache no matter what index is being used for remapping (longer k). Nevertheless line remapping still improves cache performance as can be seen in Figure 5.1, where even 4 L-associative cache outperforms all but the ordinary fully-associative cache architectures and security-aware caches.

5.2 Performance - Cache decoder delay and resource usage

Our goal is to investigate the worst case scenario and to explore potential problems with our implementation. We summarize LUT utilization and achievable frequency for Leon 3 extended with our security-aware cache in Figures 5.2 to 5.4. The LUT figures present the overhead introduced after adding security-aware cache.

For comparison, we present the resource usage and maximum frequency for vendor specific CAM in tables 5.1 and 5.4. We prepared a number of builds using a Xilinx Spartan 3 FPGA chip for comparison. The chip is larger then the Virtex 4 chip and as the builds take substantially larger amount of time we made only a limited number of them. The results can be seen in Figure 5.6.

We only present the flip flop utilization for Spartan 3 chip. The flip flop utilization for Virtex 4 chip follows the same pattern as for Spartan 3. We are aware that for some *systems on chip* (SOC) number of available flip flops is important.

Leon 3 with the configuration we are using utilizes around 5,000 LUTs and 1,800 flip flops and equivalent amount of BRAMs (ordinary Leon with equivalent size cache will use as many BRAMs as modified Leon 3) for a Virtex4 chip. The figures we present are the overhead that the security-aware cache introduces. We use Virtex4 as it is a small chip and allowed us to prepare a large number of builds.

The configuration files for Spartan 3 and Virtex 4 Leon 3 are provided in Appendix B.

Figure 5.2: Security-aware cache remapping register file cost for different cache sizes and k parameters. Avnet ML401 board with Xilinx Virtex 4 XC4VLX25 FPGA. The quadratic growth of the number of required LUT is very clear for any k. k value has very little impact on the amount of required LUTs.

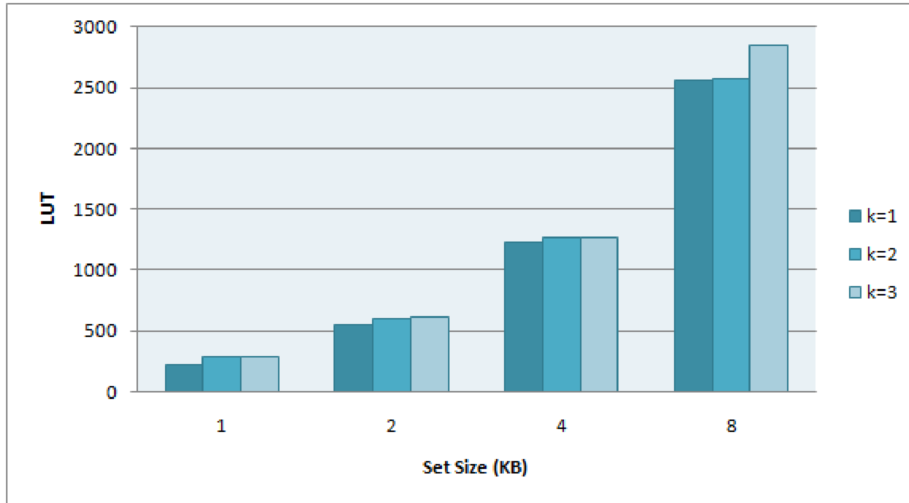


Figure 5.3: Security-aware cache highest achievable frequencies with different cache sizes and k parameters. Non modified Leon achieves 66-69 MHz. Avnet ML401 board with Xilinx Virtex 4 XC4VLX25 FPGA. For cache size of 2 KB and $k = 2$ we can see a clear anomaly, it is most likely due to synthesis tool being able to extract a faster circuit for certain parameters.

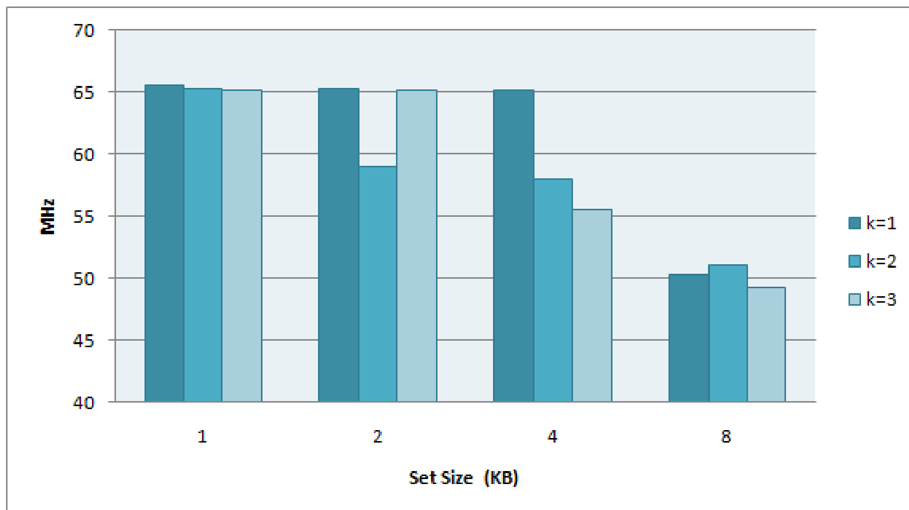


Figure 5.4: Security-aware cache LUT cost different L and cache sizes ($k = 1$). Avnet ML401 board with Xilinx Virtex 4 XC4VLX25 FPGA. The cost clearly follows a linear pattern. The $L = 4$ 4 KB set cache was the largest security-aware cache we were able to synthesize on Virtex 4 chip.

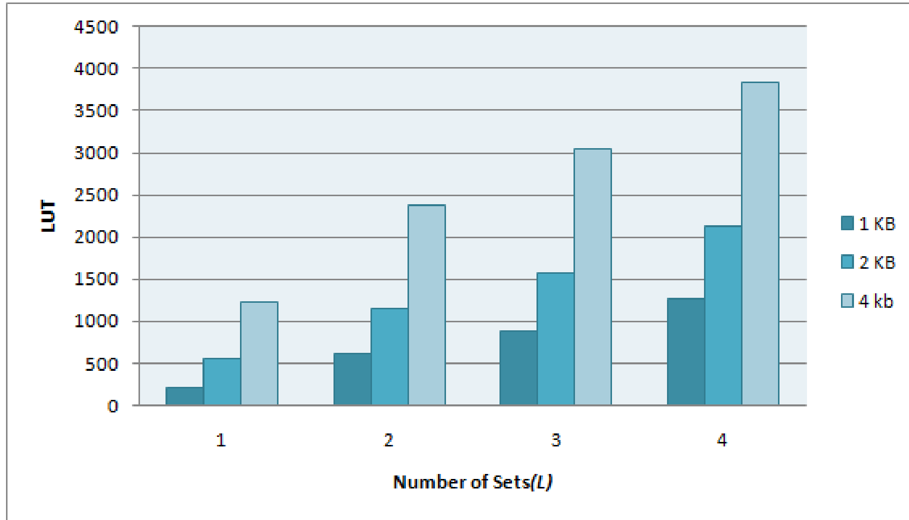


Figure 5.5: Security-aware cache highest achievable frequencies for different L and cache sizes ($k = 1$). Avnet ML401 board with Xilinx Virtex 4 XC4VLX25 FPGA. The frequency drop is proportional to increase in cache set size. The L parameter has very little impact on the achievable frequency.

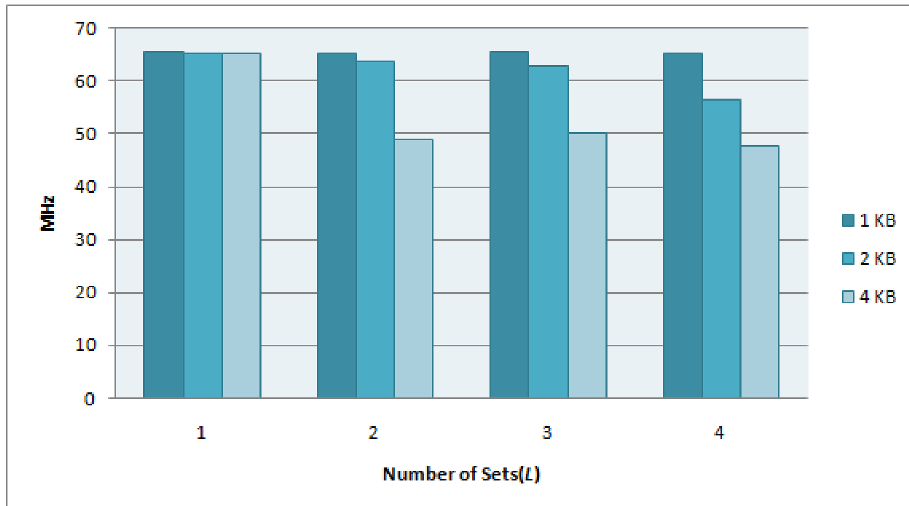
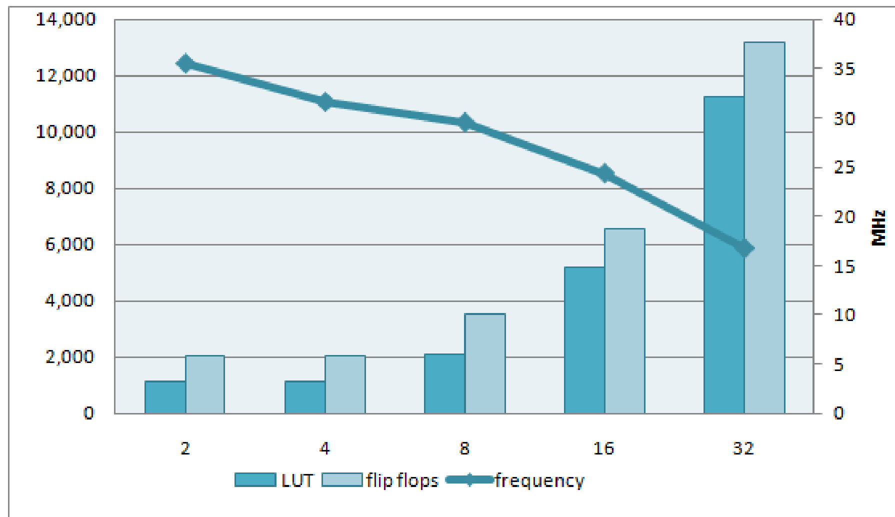


Figure 5.6: GR-XC3S-1800: Resource Utilization and Performance of Security-aware cache with k=1 L=1.



The CAM properties of designs presented in Tables 5.1 to 5.4:

- SRL16 based CAM with a 16 clock-cycle write operation and a one clock-cycle search operation.
- Block RAM based CAM with only a two clock-cycle write operation and a one clock-cycle search operation. The block RAM based CAM also supports an optional additional output register, which adds one clock cycle latency to all read operations.
- The frequency provided is "ideal". The benchmark was performed on a chip with CAM and minimal extra logic.

The tables on the next page present resource utilization and performance of different CAM implementations.

Table 5.1: Virtex-5 FPGA SRL-based CAM Implementation: [4]

CAM Width	CAM Depth														
	16					256					1024				
	SRL16s	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	SRL16s	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	SRL16s	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)
8	32	148	27	190	200	512	1848	36	110	120	2048	7200	39	80	100
32	128	288	52	170	200	2048	3438	60	100	110	8192	13493	63	70	80
64	256	474	84	160	190	4096	5561	93	90	100	16348	21899	101	60	70

Table 5.2: Virtex-5 FPGA Block RAM-based CAM Implementation: [4]

CAM Width	CAM Depth														
	16					256					1024				
	Block RAMs	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	Block RAMS	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	Block RAMs	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)
8	1	95	34	200	210	8	1077	277	130	160	32	4138	1048	100	110
32	4	170	57	170	180	32	1263	302	110	130	128	4677	1072	60	80
64	7	281	91	140	160	56	1721	335	90	110	•(3)	•(3)	•(3)	•(3)	•(3)

Table 5.3: Spartan-3A FPGA SRL-based CAM Implementation: [4]

CAM Width	CAM Depth														
	16					256					1024				
	SRL16s	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	SRL16s	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	SRL16s	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)
8	32	151	27	80	100	512	1727	36	50	70	2048	6646	42	40	50
32	128	299	52	80	100	2048	3333	60	50	60	8192	12892	66	40	40
64	256	499	84	80	90	4096	5464	92	40	50	•(3)	•(3)	•(3)	•(3)	•(3)

Table 5.4: Spartan-3A FPGA Block RAM-based CAM Implementation: [4]

CAM Width	CAM Depth														
	16					256					1024				
	Block RAMs	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	Block RAMS	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)	Block RAMs	LUTs	FFs	Perf. (MHz)(1)	Perf. (MHz)(2)
8	1	124	35	110	130	8	1368	278	70	80	32	5623	1079	50	60
32	4	219	59	90	120	32	2236	303	60	70	•(3)	•(3)	•(3)	•(3)	•(3)
64	8	357	92	90	110	•(3)	•(3)	•(3)	•(3)	•(3)	•(3)	•(3)	•(3)	•(3)	•(3)

5.2.1 LUT

The index remapping register files size grows quadratically with the cache size. This is due to the fact that both the index and depth of cache is increased with cache size. Although vendor provided CAM memories currently have better resource utilization they also do follow quadratic growth.

The L parameter can increase resource efficiency when the validation tables are unregistered, but this is not always the case. For example looking at Figure 5.4 we can see that we need more LUTs to implement a $L = 4$ 1 KB set cache than $L = 1$ 4 KB set cache. The results for L-associative cache are obviously far from satisfactory, and although the cache is unoptimized it gives a good feedback on that the unregistered tables should only be used for evaluation platforms. For example 4 L set 8 KB set L-associative security-aware cache (32 KB cache size) with validation tables implemented as part of tag/data arrays would use equivalent amount of LUTs to a single set 8KB security-aware cache, the current design over-maps on the chip resources. The associative search logic is expensive as it is unoptimized and when $L > 2$ we use unregistered validation tables(evaluated later in the Chapter).

5.2.2 Flip-flops

Our cache has flip flop utilization rate that follows the LUT demand. The flip flop usage scales quadratically with cache size. The k parameter has a slight effect on flip flop usage. On Viretx 4 even for 8 KB set size with $L = 1$ the difference between design with k=3 and k=1 is 300 flip flops (7900 and 7600).

5.2.3 Frequency

It is obvious that the security-aware cache introduces overheads for both flip flop and LUT resources. Our decoder is currently designed for correctness, not speed. The cache CAM decoder has to be optimized to achieve a reasonable performance level.

The frequency drop is quite clear when we increase cache size. The k parameter seems to have little impact on the performance. This should be the case as it only widens the CAM circuit, it does not change its depth.

The frequency also drops with increased L parameter. This is due to increased depth of the circuit imposed by set management logic.

There is a clear irregularity in Figure 5.3. Currently we have not investigated the reason why cache with $k = 2$ performs worse than cache with $k = 3$ although it is likely to be a result of Virtex 4 slice architecture.

5.2.4 Block-size

Although we do not present the impact of block-size on cache performance, we are going to comment on it. A two-fold increase in block-size decreases the depth of the search circuit by one level (assuming binary tree structure) and lowers the resource requirement at *exactly* the same rate as decrease of cache set size.

A cache with 4 byte block-size and 4 KB security aware cache needs basically the same remapping circuit as 8 byte 8 KB security-aware cache. The only

difference will be a change in Tag field width (decreased by 1 bit in case of twice as big block-size).

Using vendor provided CAM memory:

Large security-aware caches and especially marketable designs are not intended to be implemented using the generic flip flop/LUT based CAM. The pipelined security-aware cache CAM FIFO buffer can be based on CAM and this is why we investigate different customizations of the security-aware cache, as logic for both is essentially the same. For the final design, appropriate buffer size should be investigated as it could lengthen the critical path.

We present a study case of using a vendor provided memory.

If the chips we are using have a reasonable number of BRAMs we could decrease the number amount of LUTs down to approximately 75% of what we currently use. Assuming:

- We use a Spartan 3 chip.
- CAM size linearly increases with the word width.
- CAM is 1024 word deep.
- 32 byte block-size.
- Index of width $(n + k)$ 11(k=1; n=10).
- The vendor provided CAM memory imposes pipelined design as it is a single-cycle read design.
- Cache control logic would add to the CAM cost for security-aware cache implementation.

Referring to the Tables 5.1 up to 5.4 it is clear that we would need around 8000 extra LUTs over the original design and around 32-34 extra BRAMs. Our remapping register file uses around 12,000 LUTs for a 32 KB cache with $k = 1$ on a Spartan 3 chip. We could save around 4,000 LUTs at the expense of using a pipelined design and a number of BRAMs.

One could argue that the BRAMs could be used to extend cache, and this can be true in certain cases. Increasing non security-aware cache two folds increases the BRAM requirement quadratically as well, as the cache depth has to be increased two fold and the tag width has to be increased as well. Furthermore the BRAM binding logic has to be included. It might be the case that half sized security-aware cache will deliver similar or higher hit rate at smaller BRAM and higher LUT usage. Depending on the chip parameters, it might be of an advantage. The choice of CAM requires a careful case study depending on the user requirements.

Taken into account that our CAM circuit is unoptimized it might be reasonable to verify the flip flop based CAM designs. If the number of LUTs of our current design was decreased by around 30% the design would be a tough competitor for current Xilinx designs. We believe that current overhead is caused by single bit latches and excessive routing. Although we did not measure the critical path of our CAM, it is very likely that the critical path of Xilinx CAM is shorter. For maximum cache performance it is of no difference as long as

the critical path of CAM is slightly longer than the current designs critical path(given that we use pipelined cache).

5.3 Comparison of pipelined security aware cache with fully associative pipelined cache

A similar non-security-aware FPGA cache has been proposed by Yiannacouras and Rose [24], which has several significant differences from our design. The first difference is that our cache performs associative search only on indices. Associative search is both slow and resource wise very expensive, by decreasing the width of CAM to n+k instead of n+t (t is tag width, $t >> k$) bits. Using a tag array implemented as BRAMs, we have a smaller and therefore faster design, yet offering similar hit rate [23].

The second difference is that our cache can use both *write through* and *write back* policy. First one requires less logic and the second is only really suitable for non L-associative caches. In case write back policy is used we implement the tag/data/id array using dual port BRAMs. Whenever a collision occurs between the line written to the cache and one that is currently being stored, we read the data from the cache to write it back, and we use the second port to store the new data. The indices we need to reference the arrays are known as they are stored within the CAMs.

5.4 Security

Our security-aware cache and pipelined security-aware caches follow the same principles as the original security-aware cache which has been proved to offer enhanced security [23]. The L-associative security-aware cache treats tag misses differently. The miss is a tag miss that involves protected cache lines and there are sets within the line that are not protected by context bits. We replace them with the new block, effectively treating the miss as a tag miss. The miss is a tag miss that involves protected cache lines, and there are no sets within the line with unprotected context. We treat the miss as an ordinary context miss in a security-aware cache (evicting randomly selected line and accessing the data directly from memory). The proposed SecRAND randomization mechanism satisfies the security criteria stated in [23] that if an attacker accesses line i that causes an eviction, an eviction can be observed at any line with equal probability. Thus achieves zero channel capacity.

Proving Modified processors security features

The processor testing suite is currently not designed to take the security-aware features into account thus we present a general study case of an attempted cache timing attack and how it is being prevented by the security-aware cache.

The AES cache-timing attack exploits the following fact(as described in Chapter 2).

> "Cache-Collision Assumption. For any pair of lookups i, j, given a large number of random AES encryptions with the same key, the

> average time when $< i >=< j >$ will be less than the average time when $< i >! =< j >$." [10]

This concept of cache-timing attack can be further generalized to all cache timing attacks.

We assume:

- Leon 3 is running any operating system with MMU enabled. Without MMU processor is modified with context switching mechanism as described in Chapter 4. Therefore has the same secure against cache timing attacks.
- The user process A with context $ID = A$ stored within the processors data-cache.
- Malicious process B is trying to perform a timing attack on line with index I containing part of the protected information.

We take into account any possible attack. We consider attack against one cache line as the principle can be generalized to any case. The code is executed by a malicious process a number of times in order to calculate and compare the average response time. Be it service, program or any other. The cache timing attack would involve executing any one of the following instructions and comparing the execution times.

- index(addr1) = index(addr2).
- **LD** *addr1*, *reg1*. Any memory loading instruction(beginning with LD).
- **ST** *addr1*, *reg1*. Any store to memory instruction(beginning with ST).

Timing attacks exploit the response time of the system, which varies and allows the attacker to determine whether a given cache line is cached or not. This is the information that the attack is trying to obtain, how it is using it is irrelevant and can vary between the attacks. To prevent the attack we have to make the response time for $< i >=< j >$ and $< i >! =< j >$ indistinguishable if i and j are owned by different processes.

Non-modified Leon 3

In case of non-modified Leon 3 any miss(tag or context) will results in a line eviction and a replacement by a new line. Both context and tag miss results in the same state pattern. The evicted line is the one with the matching index. Therefore it is predictable and exploitable by cache-timing attacks.

Modified Leon 3

- *Tag miss*, In case of a tag miss without a context miss match, the index matching line is replaced. No information is leaked.
- *Index miss*, Whenever a index miss occurs, a randomly chosen line is replaced. The victim is randomly chosen and does not leak any information on cache state.
- *Context miss*, Whenever a context miss occurs, the cached line will not be evicted. Instead a word is fetched from memory, and a randomly chosen line is evicted. The procedure follows the same state pattern and as such timing-wise is indistinguishable from an ordinary tag miss. The attacker can not verify whether the line is cached or not.

From attackers perspective it is impossible to distinguish the evicted line index.

5.5 Summary

In this chapter we evaluated the performance of modified Leon 3 processor. Clearly the elongated critical path problem has to be countered by employing the pipelined cache design. Currently for larger caches the frequency impact is significant.

We have shown that the increase of size of the remapping register file is quadratic with respect to cache set size increase. This holds for both the summarized vendor provided CAM memories and our circuits.

We also investigated the impact of k and L parameters on the security-aware cache hit rate. What the current figures tell us is that we should only make use of registered validation tables. Also the k parameter can be easily used to extend the cache by 3 bits, and most likely more, without severe resource overhead.

We summarized our design by showing that it is secure.

Chapter 6

Future work and conclusions

Future work

There are a number of things that we feel should be done. We plan to extend the Leon 3 suite to support all of the caches presented above with a full range of available parameters for customization. The task is to implement them in an efficient and portable manner making the cache attractive to potential users. This includes finishing the pipelined design and making use of vendor provided CAM memories. The obvious issue with the pipelined security-aware cache is the difficulty in modifying the existing designs. The modification should keep or even improve the IPC metric of the processor.

Having all of the designs implemented we plan to install the full SecRAND algorithm with MMU enabled. This would allows us to devise a cache timing attack and test the caches security features on an actual application. It would have to be done on an actual FPGA based system, simulator is not suitable due to the amount of time such an attack requires. Having a Leon 3 modified system running we also plan to run a number of community recognized benchmarks to further verify Leon 3 performance results.

There are a number of potential performance bottlenecks that have to be verified. An extension to Simplescalar, or CACTI would allow us to verify cache functional performance. The more advanced extended simulator would be a perfect performance evaluation platform for any of the new architectures. Our current functional simulator was only intended to give us a quick insight on the new architecture performance. If any new issues arise it could still be used to verify whether a solution is worth any further investigation.

One of the parameters which we did not verify is the buffer size in pipelined security-aware cache. The performance impact of buffer size should be investigated both from theoretical and practical point of view. The theoretical aspect should be modelled using queuing theory. Modelling of impact of CAM FIFO buffer size on performance and resource utilization is an important step. This

should lead to an evaluation tool which would be provided along with extended Leon 3 software suite.

The two different implementations of FIFO CAM buffers used in pipelined security-aware cache should also be investigated. Although the more complicated control might be more resource demanding, in case of certain write/read patterns it could offer a serious performance advantage. We only have a very rough intuition on this manner. Currently we believe that the cache control implementation should depend on the CAM memory (not FIFO CAM as this should be based on generic CAM) parameters, more precisely the M parameter. For large M the buffer size should be appropriately extended, and then the more complex control logic could be offer a serious performance advantage.

We plan to design a processor architecture and memory management unit that would address the pipelined cache problem. The challenge is to develop a processor with a cache that will be only slightly more resource-demanding compared to existing processors, while delivering superior power, speed and security. The architecture could be based on any of the current ISA, even be a development of Leon 3 or the new Leon 4 processor.

We would like to evaluate a more pipelined design. One where the index remapping circuit is pipelined in a similar fashion to what Pagiamtzis and Sheikholeslami [20] proposed. The security-aware cache would become a highly pipelined design. We did not evaluate the power consumption of current design, but is is likely that hierarchical search would surpass it. There are a number of complications with such an approach and they would have to be evaluated. For example increased complexity of the design.

Conclusions

Our results indicate that security-aware cache architecture shows promise for FPGA implementation. One question we ask is whether our cache would match performance expectations when implemented using optimized CAM memories. Is it worth spending a quarter of chip resources to increase security? We believe that it is. Modern soft processors do not fully exploit the potential offered by FPGA technology in terms of embedded flip flops, and we feel that one way to efficiently utilize them is by using a security-aware cache and other highly pipelined architectures.

Bibliography

[1] http://www.xilinx.com/publications/magazines/emb_04/xc_pdf/p18-21_4emb-archide.pdf. Retrieved June 15, 2010.

[2] http://www.opensparc.net/. Retrieved June 15, 2010.

[3] http://www.actel.com/documents/CAM_AN.pdf. Retrieved June 15, 2010.

[4] http://www.xilinx.com/support/documentation/ip_documentation/cam_ds253.pdf. Retrieved June 15, 2010.

[5] http://vhdlguru.blogspot.com/2010/03/random-number-generator-in-vhdl.html. Retrieved June 15, 2010.

[6] http://www.uclinux.org. Retrieved June 15, 2010.

[7] http://www.gaisler.com/doc/sparcv8.pdf. Retrieved June 15, 2010.

[8] Anant Agarwal and Stephen D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 179–190, New York, NY, USA, 1993. ACM.

[9] D.J. Bernstein. Cache-timing attacks on aes. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[10] J. Bonneau and I. Mironov. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems – CHES 2006*, pages 201–215, 2006.

[11] S. Brown and J. Rose. Architecture of fpgas and cplds: A tutorial. http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf. Retrieved June 15, 2010.

[12] F. Vahid D. Sheldon and S. Lonardi. Soft-core processor customization using the design of experiments paradigm. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6, 2007.

[13] L. Howes D.B. Thomas and W. Luk. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of international symposium on FPGAs*, 2009.

[14] R. G. Ragel I. Herath. Side channel attacks: Measures and countermeasures. `http://www.ce.pdn.ac.lk/~escal/papers/herath07side.pdf`. Retrieved June 15, 2010.

[15] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM.

[16] G. Dundar O. Mencer K. Atasu, C. Ozturan and W. Luk. Chips: Custom hardware instruction processor synthesis. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 27, pages 528–541, 2008.

[17] Jih kwon Peir, Yongjoon Lee, and Windsor W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *In Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII*, pages 240–250, 1998.

[18] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005.

[19] D. Page. Partitioned cache architecture as a side-channel defense mechanism. `http://eprint.iacr.org/2005/280.pdf`, 2005. Retrieved June 15, 2010.

[20] Kostas Pagiamtzis and Ali Sheikholeslami. A low-power content-addressable memory (cam) using pipelined hierarchical search scheme. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 39(9):1512–1519, 2004.

[21] Jonathan Rose. Hard vs. soft: The central question of pre-fabricated silicon. *Multiple-Valued Logic, IEEE International Symposium on*, 0:2–5, 2004.

[22] D. Levi S.A. Guccione and D. Downs. A reconfigurable content addressable memory. In *Lecture Notes In Computer Science,Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 882–889, 2000.

[23] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Peter Yiannacouras and Jonathan Rose. A parameterized automatic cache generator for fpgas. In *Proc. Field-Programmable Technology (FPT*, pages 324–327, 2003.

Appendix A

Guide: How to use Security-aware cache?

The security-aware cache has several parameters and features what results in a large number of possible configurations. This chapter is a guide on how to select the parameter depending on different performance and implementation goals. By resource usage we understand the resources used to implement CAM memory and by performance the combined impact of hit rate and achievable frequency. The guide is based on our findings discussed in Chapter 5 along with Wang and Lee paper [23]. The cache configuration utility for modified Leon 3 is presented in Figure A.1. We discuss the impact of individual parameters on cache performance.

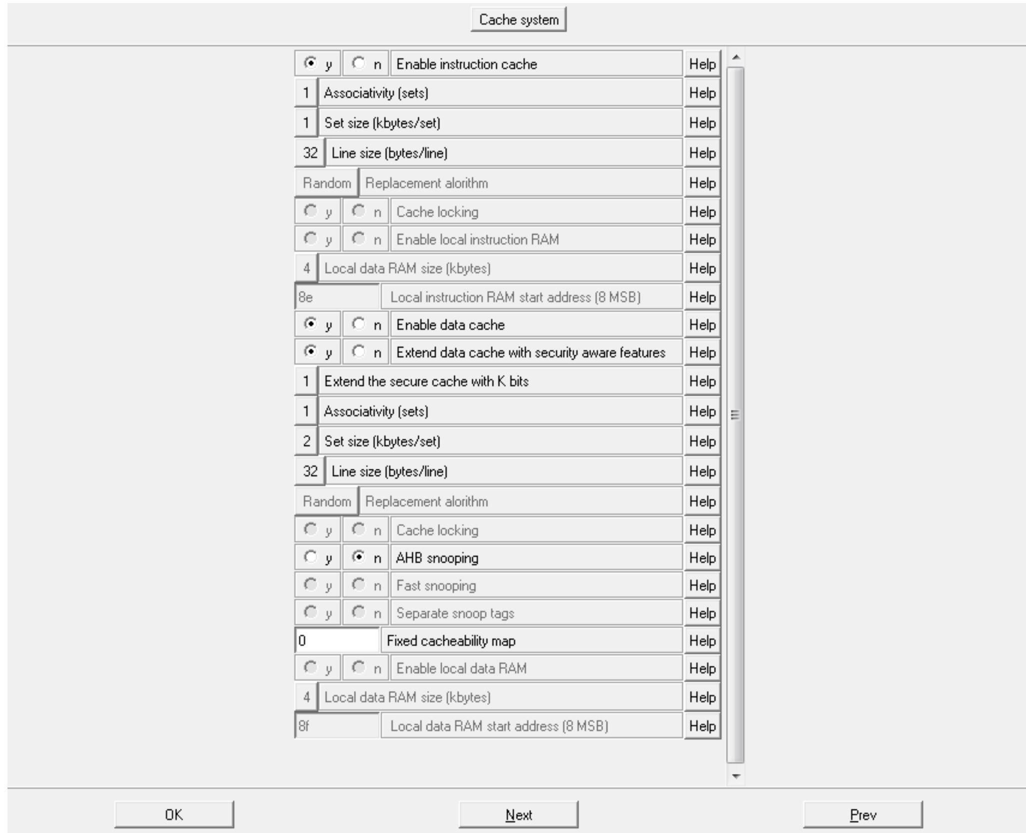
We try to clarify the impact of the following parameters on cache resource usage and performance:

- k - The number of bits used to extend index.
- L - The number of sets.
- b - Block-size.
- *set size* - self explanatory.

In case user is time constrained the non-pipelined security aware cache is recommended. Although design performance will most likely be degraded L-associative principle could lower the resource increased impact. We do not recommend unregistered validation tables, as our study has shown although easier to implement it is a resource demanding solution.

For applications that require high performance and the implementation is not as time constrained, our pipelined security-aware cache should be used. It offers the same cache hit rate while having a higher clock frequency and lower resource usage than a fully associative cache. The resource utilization and performance for a specific customized processor can be balanced by choosing k and the cache depth carefully. The L-associative principle can be applied here as well, if resource utilization becomes an issue.

Figure A.1: Modified Leon 3 cache configuration utility. Currently Several constraints are imposed on the design when using the extended cache. The cache can be fully configured using the Leon 3 configuration utility.



In case one is more concerned with cache size and security than performance, we offer the L-associative cache. By balancing the cache depth, L and k, the L-associative security-aware cache can be customized to use only $(1/L)$ of the resources of an ordinary security-aware cache of equal size, at the cost of a marginally lower hit rate. By varying L in this associative cache, one would also expect a rise in its clock frequency with respect to the non-associative security-aware cache, if the remapping register file was part of the critical path. This requires a careful analysis of specific applications, since the effect of increased miss rate might offset any performance gain.

The rest of the chapters serves as a guide with visualisation of impact of different cache parameters on both resource usage and performance.

Figure A.2: Theoretical impact of k parameter on performance. The impact is non trivial and is a combination of the device achievable frequency and hit rate increase. Depending when (and if) the cache index remapping circuit becomes the critical path the k parameter impact on the design achievable frequency will vary. The frequency will decrease at approximately logarithmic rate and depend on both the cache size and k, due to binary tree structure of CAM matching circuit. The hit rate improvement is included in the graph, it is based on results obtained and discussed in the Chapter 5. Curves show how performance varies with k depending on when the remapping circuit becomes the critical path.

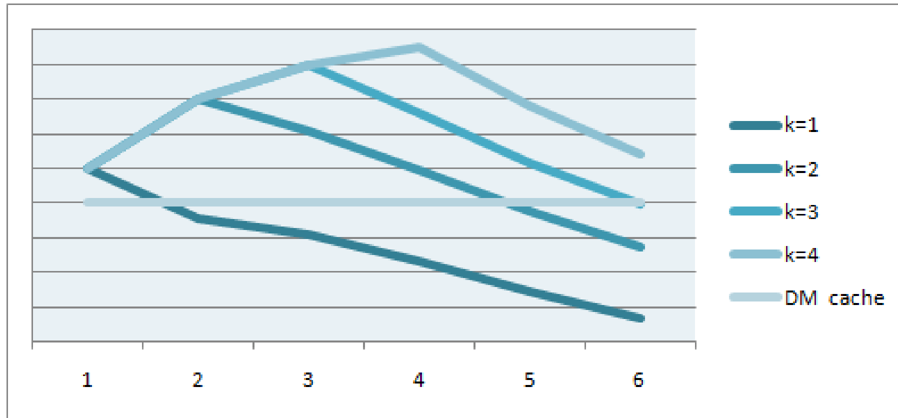


Figure A.3: Theoretical impact of k parameter on resource usage. The relationship is linear, as k extends each register linearly. Every comparison is extended by k extra bits what also implies a linear increase in resource usage. This holds for both BRAM, SLR and LUT implementations. Careful study should be performed in order to maximize resource utilization. Very often CAM memories are implemented by vendors so that they have best efficiency when words have certain width at certain depth.

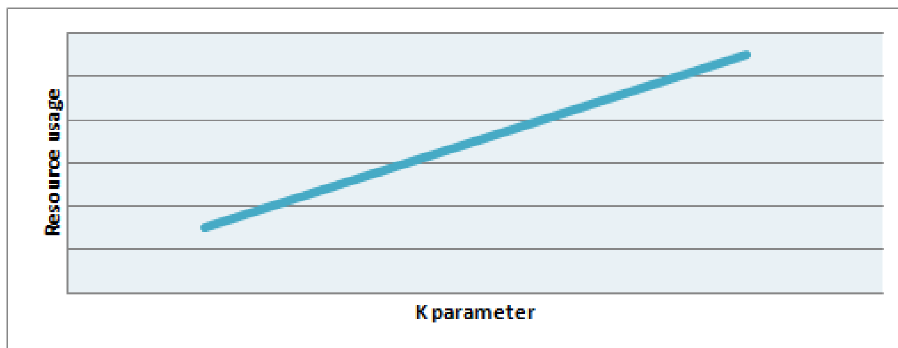


Figure A.4: Theoretical impact of L parameter on performance. What we can see on the graph is performance degradation of cache with respect to the increased associativity. Both the unregistered and registered validation table based caches are included. It is quite clear that unregistered validation tables should be avoided.

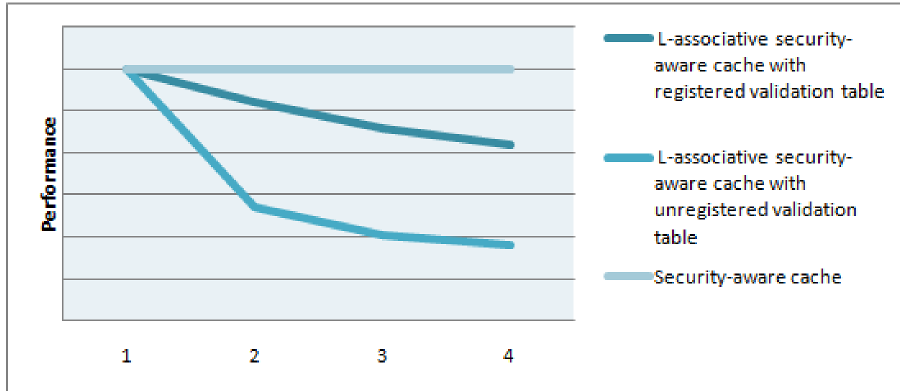


Figure A.5: Theoretical impact of L parameter on resource usage. In ideal situation when the validation table is implemented along with the rest of tag field the L parameter has a linear relationship with resource utilization of cache. The extra overhead in terms of logic required to determine set invalidation is negligible. When unregistered storage is unavailable, L-associative cache is not recommended(refer to Chapter 5). The L-associative cache when implemented with unregistered validation tables can be more efficient than non-associative cache but is far less likely.

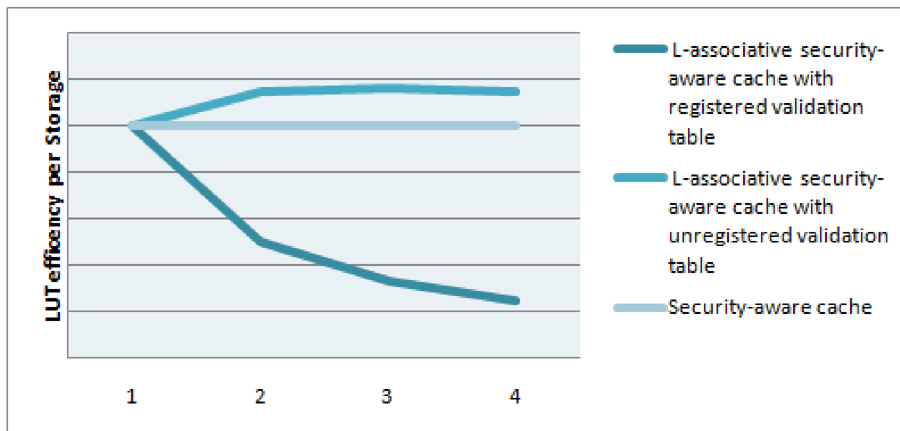


Figure A.6: Theoretical Impact of block-size on resource usage. Increasing the block-size decreases the amount of required remapping registers exponentially. This is due to the fact that increasing block-size twice increases the amount of lines per registers two fold.

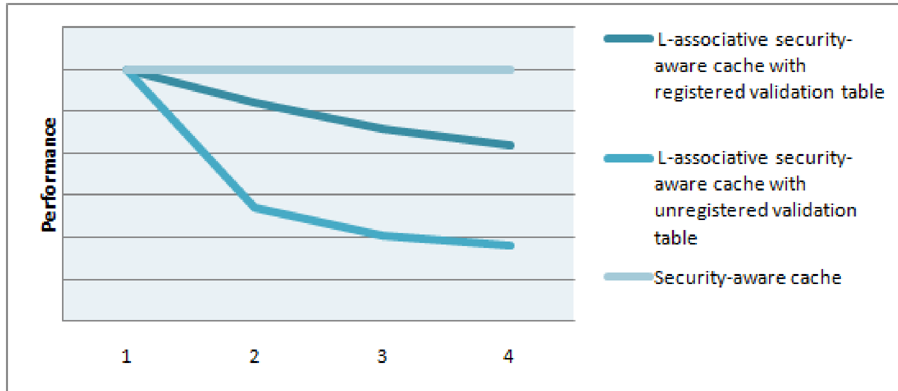
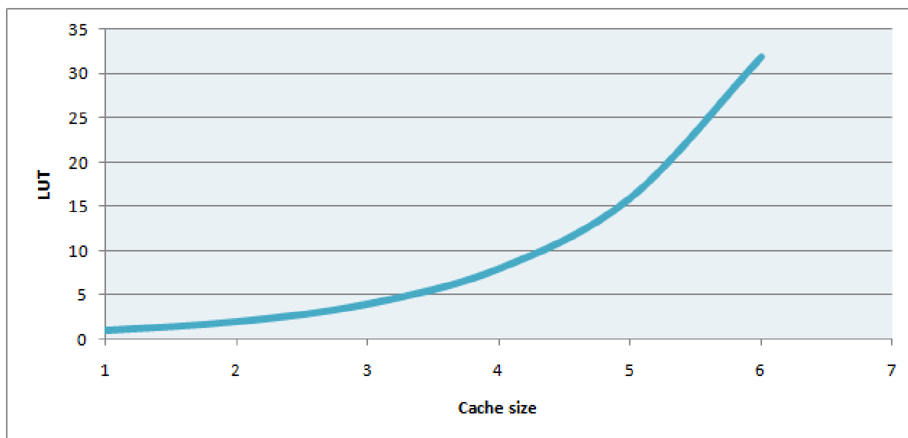


Figure A.7: By increasing cache size we exponentially increase the resources required to implement the CAM circuit. This is due to the fact that we do not only increase the index width but we also increase the depth of the CAM memory.



Appendix B

Leon 3 Configuration Files

Listing B.1: leon3-xilinx-ml403 sample configuration file.

```
1
2  -- -----------------------------------------------------------------------------
3  -- LEON3 Demonstration design test bench configuration
4  -- Copyright (C) 2009 Aeroflex Gaisler
5  -- -----------------------------------------------------------------------------
6
7
8  library techmap;
9  use techmap.gencomp.all;
10
11 package config is
12 -- Technology and synthesis options
13   constant CFG_FABTECH : integer := virtex4;
14   constant CFG_MEMTECH : integer := virtex4;
15   constant CFG_PADTECH : integer := virtex4;
16   constant CFG_NOASYNC : integer := 0;
17   constant CFG_SCAN : integer := 0;
18 -- Clock generator
19   constant CFG_CLKTECH : integer := virtex4;
20   constant CFG_CLKMUL : integer := (13);
21   constant CFG_CLKDIV : integer := (20);
22   constant CFG_OCLKDIV : integer := 2;
23   constant CFG_PCIDLL : integer := 0;
24   constant CFG_PCISYSCLK: integer := 0;
25   constant CFG_CLK_NOFB : integer := 0;
26 -- LEON3 processor core
27   constant CFG_LEON3 : integer := 1;
28   constant CFG_NCPU : integer := (1);
29   constant CFG_NWIN : integer := (2);
30   constant CFG_V8 : integer := 0;
31   constant CFG_MAC : integer := 0;
32   constant CFG_BP : integer := 0;
33   constant CFG_SVT : integer := 0;
34   constant CFG_RSTADDR : integer := 16#00000#;
35   constant CFG_LDDEL : integer := (1);
36   constant CFG_NWP : integer := (1);
37   constant CFG_PWD : integer := 0*2;
38   constant CFG_FPU : integer := 0 + 16*0;
```

```
39    constant CFG_GRFPUSH : integer := 0;
40    constant CFG_ICEN : integer := 0;
41    constant CFG_ISETS : integer := 1;
42    constant CFG_ISETSZ : integer := 1;
43    constant CFG_ILINE : integer := 8;
44    constant CFG_IREPL : integer := 0;
45    constant CFG_ILOCK : integer := 0;
46    constant CFG_ILRAMEN : integer := 0;
47    constant CFG_ILRAMADDR: integer := 16#8E#;
48    constant CFG_ILRAMSZ : integer := 1;
49    constant CFG_DCEN : integer := 1;
50    constant CFG_DSETS : integer := 1;
51    constant CFG_DSETSZ : integer := 32;
52    constant CFG_DLINE : integer := 8;
53    constant CFG_DREPL : integer := 0;
54    constant CFG_DLOCK : integer := 0;
55    constant CFG_DSNOOP : integer := 0 + 0 + 4*0;
56    constant CFG_DKSIZE : integer := 2;
57    constant CFG_DFIXED : integer := 16#0#;
58    constant CFG_DLRAMEN : integer := 0;
59    constant CFG_DLRAMADDR: integer := 16#8F#;
60    constant CFG_DLRAMSZ : integer := 1;
61    constant CFG_MMUEN : integer := 0;
62    constant CFG_ITLBNUM : integer := 2;
63    constant CFG_DTLBNUM : integer := 2;
64    constant CFG_TLB_TYPE : integer := 1 + 0*2;
65    constant CFG_TLB_REP : integer := 1;
66    constant CFG_MMU_PAGE : integer := 0;
67    constant CFG_DSU : integer := 0;
68    constant CFG_ITBSZ : integer := 0;
69    constant CFG_ATBSZ : integer := 0;
70    constant CFG_LEON3FT_EN : integer := 0;
71    constant CFG_IUFT_EN : integer := 0;
72    constant CFG_FPUFT_EN : integer := 0;
73    constant CFG_RF_ERRINJ : integer := 0;
74    constant CFG_CACHE_FT_EN : integer := 0;
75    constant CFG_CACHE_ERRINJ : integer := 0;
76    constant CFG_LEON3_NETLIST: integer := 0;
77    constant CFG_DISAS : integer := 0 + 0;
78    constant CFG_PCLOW : integer := 2;
79 -- AMBA settings
80    constant CFG_DEFMST : integer := (0);
81    constant CFG_RROBIN : integer := 0;
82    constant CFG_SPLIT : integer := 0;
83    constant CFG_AHBIO : integer := 16#FFF#;
84    constant CFG_APBADDR : integer := 16#800#;
85    constant CFG_AHB_MON : integer := 0;
86    constant CFG_AHB_MONERR : integer := 0;
87    constant CFG_AHB_MONWAR : integer := 0;
88 -- DSU UART
89    constant CFG_AHB_UART : integer := 0;
90 -- JTAG based DSU interface
91    constant CFG_AHB_JTAG : integer := 0;
92 -- Ethernet DSU
93    constant CFG_DSU_ETH : integer := 0 + 0;
94    constant CFG_ETH_BUF : integer := 1;
95    constant CFG_ETH_IPM : integer := 16#C0A8#;
96    constant CFG_ETH_IPL : integer := 16#0033#;
97    constant CFG_ETH_ENM : integer := 16#020000#;
98    constant CFG_ETH_ENL : integer := 16#000009#;
99 -- LEON2 memory controller
100   constant CFG_MCTRL_LEON2 : integer := 1;
```

```
101   constant CFG_MCTRL_RAM8BIT : integer := 0;
102   constant CFG_MCTRL_RAM16BIT : integer := 0;
103   constant CFG_MCTRL_5CS : integer := 0;
104   constant CFG_MCTRL_SDEN : integer := 0;
105   constant CFG_MCTRL_SEPBUS : integer := 0;
106   constant CFG_MCTRL_INVCLK : integer := 0;
107   constant CFG_MCTRL_SD64 : integer := 0;
108   constant CFG_MCTRL_PAGE : integer := 0 + 0;
109 -- DDR controller
110   constant CFG_DDRSP : integer := 1;
111   constant CFG_DDRSP_INIT : integer := 1;
112   constant CFG_DDRSP_FREQ : integer := (100);
113   constant CFG_DDRSP_COL : integer := (9);
114   constant CFG_DDRSP_SIZE : integer := (64);
115   constant CFG_DDRSP_RSKEW : integer := (0);
116 -- SSRAM controller
117   constant CFG_SSCTRL : integer := 0;
118   constant CFG_SSCTRLP16 : integer := 0;
119 -- AHB status register
120   constant CFG_AHBSTAT : integer := 0;
121   constant CFG_AHBSTATN : integer := 1;
122 -- AHB ROM
123   constant CFG_AHBROMEN : integer := 0;
124   constant CFG_AHBROPIP : integer := 0;
125   constant CFG_AHBRODDR : integer := 16#000#;
126   constant CFG_ROMADDR : integer := 16#000#;
127   constant CFG_ROMMASK : integer := 16#E00# + 16#000#;
128 -- AHB RAM
129   constant CFG_AHBRAMEN : integer := 0;
130   constant CFG_AHBRSZ : integer := 1;
131   constant CFG_AHBRADDR : integer := 16#A00#;
132
133 -- Gaisler Ethernet core
134   constant CFG_GRETH : integer := 0;
135   constant CFG_GRETH1G : integer := 0;
136   constant CFG_ETH_FIFO : integer := 8;
137
138 -- UART 1
139   constant CFG_UART1_ENABLE : integer := 0;
140   constant CFG_UART1_FIFO : integer := 1;
141
142 -- LEON3 interrupt controller
143   constant CFG_IRQ3_ENABLE : integer := 0;
144   constant CFG_IRQ3_NSEC : integer := 0;
145
146 -- Modular timer
147   constant CFG_GPT_ENABLE : integer := 0;
148   constant CFG_GPT_NTIM : integer := 1;
149   constant CFG_GPT_SW : integer := 8;
150   constant CFG_GPT_TW : integer := 8;
151   constant CFG_GPT_IRQ : integer := 8;
152   constant CFG_GPT_SEPIRQ : integer := 0;
153   constant CFG_GPT_WDOGEN : integer := 0;
154   constant CFG_GPT_WDOG : integer := 16#0#;
155
156 -- GPIO port
157   constant CFG_GRGPIO_ENABLE : integer := 0;
158   constant CFG_GRGPIO_IMASK : integer := 16#0000#;
159   constant CFG_GRGPIO_WIDTH : integer := 1;
160
161 -- I2C master
162   constant CFG_I2C_ENABLE : integer := 0;
```

```
163
164 -- VGA and PS2/ interface
165   constant CFG_KBD_ENABLE : integer := 0;
166   constant CFG_VGA_ENABLE : integer := 0;
167   constant CFG_SVGA_ENABLE : integer := 0;
168
169 -- GRLIB debugging
170   constant CFG_DUART : integer := 0;
171 end;
```

Listing B.2: leon3-xilinx-xc3sd-1800 sample configuration file.

```
1
2 --
   ------------------------------------------------------------------------------
3 -- LEON3 Demonstration design test bench configuration
4 -- Copyright (C) 2009 Aeroflex Gaisler
5 --
   ------------------------------------------------------------------------------
6
7
8 library techmap;
9 use techmap.gencomp.all;
10
11 package config is
12 -- Technology and synthesis options
13   constant CFG_FABTECH : integer := spartan3;
14   constant CFG_MEMTECH : integer := spartan3;
15   constant CFG_PADTECH : integer := spartan3;
16   constant CFG_NOASYNC : integer := 0;
17   constant CFG_SCAN : integer := 0;
18 -- Clock generator
19   constant CFG_CLKTECH : integer := spartan3;
20   constant CFG_CLKMUL : integer := (6);
21   constant CFG_CLKDIV : integer := (20);
22   constant CFG_OCLKDIV : integer := 2;
23   constant CFG_PCIDLL : integer := 0;
24   constant CFG_PCISYSCLK: integer := 0;
25   constant CFG_CLK_NOFB : integer := 0;
26 -- LEON3 processor core
27   constant CFG_LEON3 : integer := 1;
28   constant CFG_NCPU : integer := (1);
29   constant CFG_NWIN : integer := (8);
30   constant CFG_V8 : integer := 2;
31   constant CFG_MAC : integer := 0;
32   constant CFG_BP : integer := 0;
33   constant CFG_SVT : integer := 0;
34   constant CFG_RSTADDR : integer := 16#00000#;
35   constant CFG_LDDEL : integer := (1);
36   constant CFG_NWP : integer := (0);
37   constant CFG_PWD : integer := 0*2;
38   constant CFG_FPU : integer := 0 + 16*0;
39   constant CFG_GRFPUSH : integer := 0;
40   constant CFG_ICEN : integer := 0;
41   constant CFG_ISETS : integer := 1;
42   constant CFG_ISETSZ : integer := 1;
43   constant CFG_ILINE : integer := 8;
44   constant CFG_IREPL : integer := 0;
45   constant CFG_ILOCK : integer := 0;
46   constant CFG_ILRAMEN : integer := 0;
47   constant CFG_ILRAMADDR: integer := 16#8E#;
```

```
48    constant CFG_ILRAMSZ : integer := 1;
49    constant CFG_DCEN : integer := 1;
50    constant CFG_DSETS : integer := 1;
51    constant CFG_DSETSZ : integer := 2;
52    constant CFG_DLINE : integer := 8;
53    constant CFG_DREPL : integer := 0;
54    constant CFG_DLOCK : integer := 0;
55    constant CFG_DSNOOP : integer := 1 + 0 + 4*0;
56    constant CFG_DKSIZE : integer := 1; ---***Maciej*** dksize added
57    constant CFG_DFIXED : integer := 16#0#;
58    constant CFG_DLRAMEN : integer := 0;
59    constant CFG_DLRAMADDR: integer := 16#8F#;
60    constant CFG_DLRAMSZ : integer := 1;
61    constant CFG_MMUEN : integer := 1;
62    constant CFG_ITLBNUM : integer := 8;
63    constant CFG_DTLBNUM : integer := 8;
64    constant CFG_TLB_TYPE : integer := 0 + 1*2;
65    constant CFG_TLB_REP : integer := 0;
66    constant CFG_MMU_PAGE : integer := 0;
67    constant CFG_DSU : integer := 1;
68    constant CFG_ITBSZ : integer := 4;
69    constant CFG_ATBSZ : integer := 4;
70    constant CFG_LEON3FT_EN : integer := 0;
71    constant CFG_IUFT_EN : integer := 0;
72    constant CFG_FPUFT_EN : integer := 0;
73    constant CFG_RF_ERRINJ : integer := 0;
74    constant CFG_CACHE_FT_EN : integer := 0;
75    constant CFG_CACHE_ERRINJ : integer := 0;
76    constant CFG_LEON3_NETLIST: integer := 0;
77    constant CFG_DISAS : integer := 0 + 0;
78    constant CFG_PCLOW : integer := 2;
79  -- AMBA settings
80    constant CFG_DEFMST : integer := (0);
81    constant CFG_RROBIN : integer := 1;
82    constant CFG_SPLIT : integer := 1;
83    constant CFG_AHBIO : integer := 16#FFF#;
84    constant CFG_APBADDR : integer := 16#800#;
85    constant CFG_AHB_MON : integer := 0;
86    constant CFG_AHB_MONERR : integer := 0;
87    constant CFG_AHB_MONWAR : integer := 0;
88  -- DSU UART
89    constant CFG_AHB_UART : integer := 0;
90  -- JTAG based DSU interface
91    constant CFG_AHB_JTAG : integer := 1;
92  -- Ethernet DSU
93    constant CFG_DSU_ETH : integer := 0 + 0;
94    constant CFG_ETH_BUF : integer := 1;
95    constant CFG_ETH_IPM : integer := 16#C0A8#;
96    constant CFG_ETH_IPL : integer := 16#0033#;
97    constant CFG_ETH_ENM : integer := 16#020000#;
98    constant CFG_ETH_ENL : integer := 16#000009#;
99  -- LEON2 memory controller
100   constant CFG_MCTRL_LEON2 : integer := 1;
101   constant CFG_MCTRL_RAM8BIT : integer := 1;
102   constant CFG_MCTRL_RAM16BIT : integer := 0;
103   constant CFG_MCTRL_5CS : integer := 0;
104   constant CFG_MCTRL_SDEN : integer := 0;
105   constant CFG_MCTRL_SEPBUS : integer := 0;
106   constant CFG_MCTRL_INVCLK : integer := 0;
107   constant CFG_MCTRL_SD64 : integer := 0;
108   constant CFG_MCTRL_PAGE : integer := 0 + 0;
109 -- DDR controller
```

```
110   constant CFG_DDR2SP : integer := 1;
111   constant CFG_DDR2SP_INIT : integer := 1;
112   constant CFG_DDR2SP_FREQ : integer := (125);
113   constant CFG_DDR2SP_TRFC : integer := (130);
114   constant CFG_DDR2SP_DATAWIDTH : integer := (32);
115   constant CFG_DDR2SP_COL : integer := (10);
116   constant CFG_DDR2SP_SIZE : integer := (128);
117   constant CFG_DDR2SP_DELAY0 : integer := (0);
118   constant CFG_DDR2SP_DELAY1 : integer := (0);
119   constant CFG_DDR2SP_DELAY2 : integer := (0);
120   constant CFG_DDR2SP_DELAY3 : integer := (0);
121   constant CFG_DDR2SP_DELAY4 : integer := (0);
122   constant CFG_DDR2SP_DELAY5 : integer := (0);
123   constant CFG_DDR2SP_DELAY6 : integer := (0);
124   constant CFG_DDR2SP_DELAY7 : integer := (0);
125   constant CFG_DDR2SP_NOSYNC : integer := 0;
126 -- AHB ROM
127   constant CFG_AHBROMEN : integer := 0;
128   constant CFG_AHBROPIP : integer := 0;
129   constant CFG_AHBRODDR : integer := 16#000#;
130   constant CFG_ROMADDR : integer := 16#000#;
131   constant CFG_ROMMASK : integer := 16#E00# + 16#000#;
132 -- AHB RAM
133   constant CFG_AHBRAMEN : integer := 0;
134   constant CFG_AHBRSZ : integer := 1;
135   constant CFG_AHBRADDR : integer := 16#A00#;
136 -- Gaisler Ethernet core
137   constant CFG_GRETH : integer := 0;
138   constant CFG_GRETH1G : integer := 0;
139   constant CFG_ETH_FIFO : integer := 8;
140 -- UART 1
141   constant CFG_UART1_ENABLE : integer := 1;
142   constant CFG_UART1_FIFO : integer := 1;
143 -- LEON3 interrupt controller
144   constant CFG_IRQ3_ENABLE : integer := 1;
145   constant CFG_IRQ3_NSEC : integer := 0;
146
147 -- Modular timer
148   constant CFG_GPT_ENABLE : integer := 1;
149   constant CFG_GPT_NTIM : integer := (2);
150   constant CFG_GPT_SW : integer := (8);
151   constant CFG_GPT_TW : integer := (32);
152   constant CFG_GPT_IRQ : integer := (8);
153   constant CFG_GPT_SEPIRQ : integer := 1;
154   constant CFG_GPT_WDOGEN : integer := 0;
155   constant CFG_GPT_WDOG : integer := 16#0#;
156
157 -- GPIO port
158   constant CFG_GRGPIO_ENABLE : integer := 1;
159   constant CFG_GRGPIO_IMASK : integer := 16#0000#;
160   constant CFG_GRGPIO_WIDTH : integer := (8);
161
162 -- SVGA controller
163   constant CFG_SVGA_ENABLE : integer := 0;
164
165 -- SPI memory controller
166   constant CFG_SPIMCTRL : integer := 0;
167   constant CFG_SPIMCTRL_SDCARD : integer := 0;
168   constant CFG_SPIMCTRL_READCMD : integer := 16#0#;
169   constant CFG_SPIMCTRL_DUMMYBYTE : integer := 0;
170   constant CFG_SPIMCTRL_DUALOUTPUT : integer := 0;
171   constant CFG_SPIMCTRL_SCALER : integer := 1;
```

```
172     constant CFG_SPIMCTRL_ASCALER : integer := 1;
173     constant CFG_SPIMCTRL_PWRUPCNT : integer := 0;
174
175  -- SPI controller
176     constant CFG_SPICTRL_ENABLE : integer := 0;
177     constant CFG_SPICTRL_SLVS : integer := 1;
178     constant CFG_SPICTRL_FIFO : integer := 1;
179     constant CFG_SPICTRL_SLVREG : integer := 0;
180     constant CFG_SPICTRL_ODMODE : integer := 0;
181     constant CFG_SPICTRL_AM : integer := 0;
182     constant CFG_SPICTRL_ASEL : integer := 0;
183
184  -- GRLIB debugging
185     constant CFG_DUART : integer := 0;
186  end;
```

Appendix C

Optimized security-aware cache decoder

As multiplexing logic can be very expensive we re-reorder the buffer/CAM/random number multiplexing logic so that the output from CAM is multiplexed in the last stage. The hit bit generation logic should have a shorter path than CAM index output. Otherwise the old circuit can be used.

Figure C.1: The security-aware cache decoder with a shorter critical path.