Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Khakim Akhunov

Way-predictive instruction cache access in Rocket Chip processor with RISC-V ISA

Master's thesis in Embedded Computing Systems
Supervisor: Magnus Själander
June 2020

Abstract

The performance of central processing units (CPU) is restricted in energy consumption these days. More efficient CPUs are needed to enable improved performance. Modern computing systems exploit complex CPUs that communicate with the main memory across several levels of cache. In order to minimize the gap between CPU and memory speeds, performance-oriented developers utilize lots of power and chip area to implement a cache hierarchy. Thus, the cache is one of the main energy consumers in the system.

In modern processors, set-associativity is used to improve hit rate. Conventional set-associative Level One (L1) instruction caches (i-cache) achieve low miss rates for common applications but still consume significant energy. In set-associative caches, access time is decreased by accessing all the data ways in parallel with the tag search, while the output of only the matching way is consumed. The energy which is spent to access the other ways is wasted. There exist a large number of cache architectures with the goal of reducing their energy usage, such as phased cache, way-halting, and block buffering. However, most proposed techniques increase latency and complexity, which makes them ineligible for high-performance L1 caches.

This master thesis encompasses the implementation and evaluation of a virtual-address-matching (VAM) mechanism on a RISC-V instruction set architecture (ISA) processor, that enables access to only the way where the data has a high likelihood to reside. The main purpose is to maintain as simple implementation as possible with low area overhead and performance degradation that is an important advantage for manufacturers. In order to test the efficiency of the VAM, and to define overheads that this technique may cause, the Rocket Chip generator and its emulator are used to implement and evaluate the implementation by running existing benchmarks. Our evaluation is based on such criteria as energy consumption, occupied area, complexity, and critical-path delay. The VAM has low performance penalty and adds insignificant complexity to a conventional cache design. The results show that on average, this technique gives an energy reduction of 45%, while the increase of the critical-path delay is 5.3%. The area overhead evaluation is based on the utilization of three components: Look-Up Tables (LUT), Flip-Flops (FF), and Block Random Access Memory (BRAM). This implementation uses around 2% and 1% more LUTs and FFs respectively, but BRAM utilization remains the same compared to the conventional i-cache.

Sammendrag

Ytelsen til sentrale prosesseringsenheter er begrenset i energiforbruk i disse dager. Mer effektive prosessorer er nødvendige for å muliggjøre forbedret ytelse. Moderne datasystemer utnytter komplekse prosessorer som kommuniserer med hovedminnet på tvers av flere cache nivåer. For å minimere gapet mellom prosessor- og minnehastigheter bruker ytelsesorienterte utviklere mye strøm og chip-plass for å implementere et hurtigminners hierarki. Dermed er cachen en av de viktigste energiforbrukerne i systemet.

I moderne prosessorer brukes set-assosiativitet for å forbedre trefffrekvensen. Konvensjonelle set-assosiative nivå én instruksjons hurtigminner oppnår lave glippfrekvenser for vanlige applikasjoner, men bruker fremdeles betydelig energi. I sett-assosiative hurtigminne reduseres tilgangstiden ved å få tilgang til alle data-feltene parallelt med tag søket, mens utdataene fra bare den matchende måten forbrukes. Energien som blir brukt for å få tilgang til de andre måtene blir bortkastet. Det finnes et stort antall hurtigminne arkitekturer med mål om å redusere energiforbruket deres, for eksempel faset hurtigbuffer, måte å stoppe og blokkere buffering. Imidlertid øker de fleste foreslåtte teknikker latens og kompleksitet, noe som gjør dem ikke kvalifiserte for høyytelses nivå én hurtigminne.

Denne masteroppgaven omfatter implementering og evaluering av en virtuell adressematching (VAM) mekanisme på en RISC-V instruksjonssett-arkitektur prosessor, som gir tilgang til bare måten dataene har stor sannsynlighet for å oppholde seg på. Hovedformålet er å opprettholde en så enkel implementering som mulig med lavt arealkostnad og ytelsesforringelse som er en viktig fordel for produsentene. For å teste effektiviteten til VAM, og for å definere overheads som denne teknikken kan forårsake, brukes Rocket Chip-generatoren og dens emulator for å implementere og evaluere implementeringen ved å kjøre eksisterende benchmarks. Evalueringen vår er basert på kriterier som energiforbruk, okkupert område, kompleksitet og forsinkelse av kritisk vei. VAM har lav ytelse og gir ubetydelig kompleksitet til en konvensjonell cache-design. Resultatene viser at denne teknikken i gjennomsnitt gir en energireduksjon på 45%, mens økningen av den kritiske banen-forsinkelsen er 5,3%. Overheadevalueringen er basert på bruk av tre komponenter: Lookup tables (LUT), Flip-Flops (FF) og Block Random Access Memory (BRAM). Denne implementeringen bruker henholdsvis rundt 2% og 1% flere LUTer og FFer, men BRAM-bruken forblir den samme sammenlignet med den konvensjonelle hurtigminne.

Preface

This report is submitted to the Norwegian University of Science and Technology in partial fulfilment of the requirements for an MSc degree in computer science.

This work has been performed at the Department of Computer and Information Science, NTNU, with Magnus Själander as the supervisor.

Acknowledgements

I appreciate the European Commission for the full funding of my master's program and for the given opportunity to meet many interesting and talented people.

I would also like to thank the supervisor of the European Master in Embedded Computing Systems program, Professor Wolfgang Kunz from the Technical University of Kaiserslautern, for choosing me among many other talented applicants and instilling an interest in computer architecture with his fascinating lectures and projects.

Finally, thanks to my supervisor, Magnus Själander from NTNU for his technical and scientific support, and giving valuable feedback on this thesis.

Table of Contents

List of Tables

List of Figures

Abbreviations

ALU	=	Arithmetic Logic Unit
ASIC	=	Application-Specific Integrated Circuit
BHT	=	Branch History Table
BRAM	=	Block Random Access Memory
BTB	=	Branch Target Buffer
CAM	=	Content Addressable Memory
CLB	=	Configurable Logic Block
CPI	=	Cycles Per Instruction
CPU	=	Central Processing Unit
DMA	=	Direct Memory Access
DRAM	=	Dynamic Random Access Memory
DSP	=	Digital Signal Processing
FF	=	Flip-Flop
FPGA	=	Field-Programmable Gate Array
FPU	=	Floating-Point Unit
HDL	=	Hardware Description Language
IoT	=	Internet-of-Things
ISA	=	Instruction Set Architecture
JVM	=	Java Virtual Machine
L0,1,2	=	Level Zero, One, Two
LRU	=	Least Recently Used
LSB	=	Least Significant Bit
LUT	=	Look-Up Table
MRU	=	Most Recently Used
MSB	=	Most Significant Bit
MTU	=	Matching Table Unit
PA	=	Physical Address
PC	=	Program Counter
PPN	=	Physical Page Number
RAM	=	Random Access Memory
RAS	=	Return Address Stack
RISC	=	Reduced Instruction Set Computer
RTL	=	Register Transfer Level
SAWP	=	Sequential Address Way-Predictor
SoC	=	System-on-Chip
SPI	=	Serial Peripheral Interface
SRAM	=	Static Random Access Memory
TLB	=	Translation Lookaside Buffer
TRB	=	Tag Record Buffer
UC	=	University of California
VA	=	Virtual Address
VLIW	=	Very Long Instruction Word
VLSI	=	Very Large Scale Integration
VPN	=	Virtual Page Number
WRB	=	Way Record Buffer

Chapter 1

Introduction

Several decades ago, computation processes were only performed by huge centralized computers, utilized for financial transactions, carrier reservations and logistics, business budgeting, or in the manufacturing. Starting in the early 1990s, personal computers began to appear in the homes of ordinary people. From that point forward, there has been a tremendous advancement in the computing area. Nowadays, embedded systems with microcontrollers can be found in almost all electronic devices, and a significant number of these gadgets, for example, smartphones convey more computing power than what was accessible in the early personal computers. Reducing the size of computers has led to the fact that embedded systems have become more widely used. In our days, many of these systems are powered by a battery and are located in hard-to-reach places, which makes it difficult to frequently charge or replace the battery. With the growing popularity of the Internet-of-Things (IoT), the need for energy-efficient computing systems is more acute than ever. This is the reason the researchers and semiconductor manufacturers are spending a lot of resources in making embedded systems more energy-efficient.

On the other hand, the CPU performance race and a set of mechanisms associated with it, such as reducing Cycles Per Instruction (CPI) by exploiting pipeline, multi-issue policies and Very Long Instruction Word (VLIW), increases the speed gap between processors and memories, as the reduction of average memory access time is limited. The original reason behind this gap is the split of the semiconductor manufacture into microprocessor and memory fields. As their technology is headed in different ways, the former is aimed at increasing the speed, and the latter at increasing the capacity. To solve this access latency gap between a processor and main memory, modern CPUs employ a cache memories hierarchy.

High-performance caches dissipate significant dynamic energy by charging and discharging high-capacity bit lines and sensitivity amplifiers. As a result, caches account for a significant proportion of the total dynamic energy of the chip. A recent study about chip power consumption indicates that the principal amount of chip power has been consumed by the on-chip cache (Zang and Gordon-Ross, 2013). It states that the portion of the total microprocessor system power consumed by the cache and memory subsystem can reach

44%, and in some cases even more. Meanwhile, embedded systems and mobile devices are becoming increasingly popular, taking low power into account, which along with the requirements of high performance has become an important design constraint.

To obtain low miss rates for standard applications, modern microprocessors exploit set-associative caches. Rapid, set-associative cache implementations probe tag and data arrays in parallel followed by selecting a data from only matching way, which is, in turn, defined by the tag array. The matching way is not known, during tag and data arrays pre-charging and reading. Hence, traditional parallel access caches result in wasted dynamic energy dissipation, when pre-charge and read all the ways but select only one of the ways on a cache hit. It means that a four-way set-associative cache rejects three of the four ways on every access, by this wasting approximately 75% of dissipated energy.

There are diverse ways to reduce the cache dynamic energy, resulting in different performance effects. The key idea for reducing energy consumption is to avoid probing all the ways for nothing. One of the techniques called phased cache was proposed by Megalingam et al. (2009). In this method, access to the cache consists of two stages. In the first stage, all tags are explored in parallel. Then if there is a hit, in the next stage, data is accessed for the hit way. This phased cache shows an average 21% power reduction as compared to a conventional parallel set-associative cache architecture. However, serializing the tag and data arrays increases the cache access time, thereby deteriorating performance. This impact on the access time is not appropriate for L1 caches.

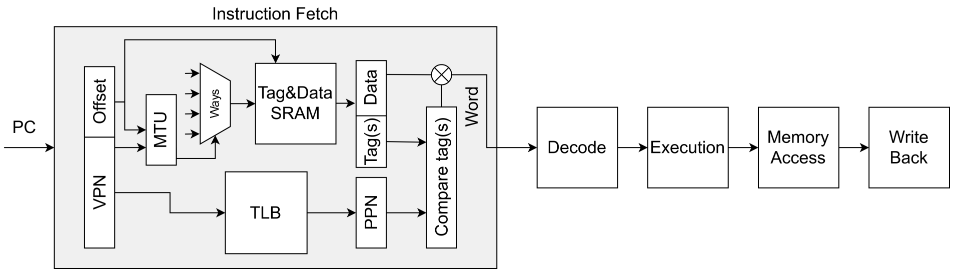


Figure 1.1: Instruction fetch stage execution with the VAM technique.

This work is researching the energy efficiency of the L1 i-cache, by implementing the virtual-address-matching (VAM) technique for predicting the way in which the required data may reside, and thus avoiding waste of the energy spent for accessing unneeded tags and data. The virtual-address-matching (Yang and Li, 2010) is performed in parallel with the matching virtual and physical page numbers in the translation lookaside buffer (TLB). When the CPU conveys an instruction access request, this approach gives the virtual address of the instruction not only to the TLB but also to the matching table unit (MTU) which is similar to a small TLB. The difference is in function and capacity. The role of the MTU is to map the virtual address to the corresponding way of the desired data in a cache. **Fig. 1.1** depicts a high-level overview of the five-stage pipeline where the instruction fetch stage is executed with the VAM technique. The program counter (PC), the address of an instruction requested by CPU, consists of a virtual page number (VPN) and an offset. The VPN is used by the TLB to search the appropriate physical page number (PPN), but simultaneously, the MTU can use the same VPN and offset to define which way to access.

If the definition succeeds, the cache accesses only the predicted way followed by the tags comparison. If it is an MTU miss, all ways are read and then compared with the PPN tag.

In this thesis, we a trying to embed the VAM technique into a conventional cache in such a way that the implementation adds as little complexity as possible while keeping the performance degradation and area overhead at an adequate level. To achieve this, we are exploiting a XOR-based mapping scheme, which maps virtual page numbers to the MTU entries. By doing this, we are also trying to reduce the energy needed to find and read the entry from the MTU. Furthermore, we are testing different configurations of the XOR-based mapping scheme to find a trade-off solution.

As a target architecture, the UC Berkeley's Rocket Chip processor (UCB, 2019a) has been used. The Rocket Chip is an open-source SoC generator that can generate a Register Transfer Level (RTL) RISC-V implementation with virtual memory, a coherent multi-level cache hierarchy and all the infrastructure to communicate with a running system. Hardware design is described in Chisel language which is developed to ease cutting edge circuit generation and design reuse for both Application-Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA) digital logic designs.

The implementation has been evaluated in terms of the consumed energy, occupied area, complexity, and critical-path delay. The results show that the VAM technique adds insignificant complexity and area overhead to a conventional i-cache design. It increases the critical-path delay by 5.3% but reduces the energy consumption by 45% on average.

The rest of the report is organized as follows. In Chapter 2, the overview of memories and their challenges, the conventional cache architecture with the virtual memory, the Rocket Chip generator, the Chisel language, and the RISC-V Instruction Set Architecture (ISA) are given. Chapter 3 describes the implementation of the VAM technique. Chapter 4 describes the evaluation methods used in this work, followed by the evaluation in terms of performance, energy saving and area overhead in Chapter 5. In Chapter 6 we discuss further work. The review of related work is given in Chapter 7, and Chapter 8 concludes this report.

Chapter 2

Background

In this chapter, some background insights are given to make the reader more comfortable in further parts of the report comprehension.

2.1 Hitting the memory wall

Processor speed has generally improved by 50-100% per year since the mid-1980s, while main memory access speed has only improved by 7% per year (Hennessy and Patterson, 2007), doubling the speed gap between processor and DRAM cycle time. If we take into account the effect of speed-up in CPU by using more aggressive pipelines, the speed gap doubles every 1 to 2 years. This effect is well-known as the memory wall and well described in (Wulf and McKee, 1994). Using the simple Equation (2.1), where t_c and t_m are the cache and main memory access times and p is the probability of a cache hit, authors explained how the increasing CPU performance hits the memory wall. If we assume that the cache speed matches that of the processor, then as t_c and t_m diverge, t_avg will grow and system performance will degrade. It means that the system performance is dictated mostly by memory latency.

$$t_{avg} = p \times t_c + (1 - p) \times t_m \tag{2.1}$$

The graph in **Fig. 2.1** shows that over time, it becomes more difficult for memory developers to keep up with the speed of the processor. This circumstance makes the main memory the bottleneck in computer performance. Furthermore, besides the latency, the memory faces more, at least two, additional challenges: increasing bandwidth and high energy consumption (Hennessy and Patterson, 2014).

2.1.1 Memory hierarchy

In an attempt to reduce this gap, cache memory was invented. Cache memories are small, high-speed buffers for storing those parts of the main memory's content which are

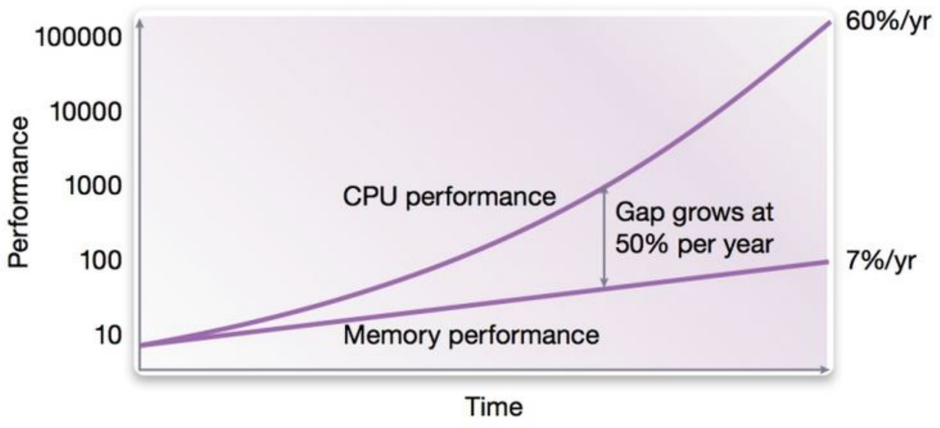


Figure 2.1: Increasing gap between CPU and memory speed.

currently in use. The cache is formed from a small amount of faster and expensive static random access memory (SRAM) and is used to speed-up the greater number of slower and cheap dynamic random access memory (DRAM). This multi-level memory constitutes the memory hierarchy idea, which aims to find a trade-off solution to achieve a cost-effective, high-performance and large memory system (Dinis, 2002). Cache memories complete the structure of the memory hierarchy composing levels closer to the processor. The size of the memory increases with the distance from the CPU but decreases in cost and speed (**Fig. 2.2**).

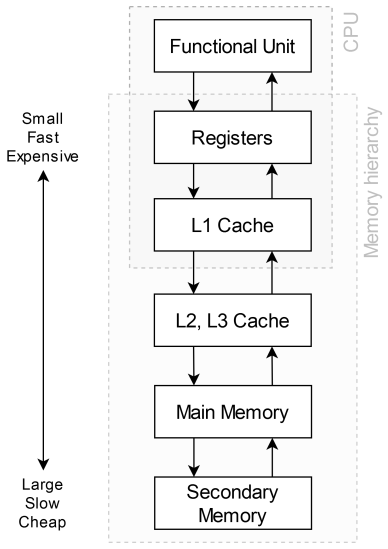


Figure 2.2: Typical modern memory hierarchy.

SRAM vs DRAM

A memory unit is designed as Random Access Memory (RAM) which means that any location can be accessed in almost the same amount of time that is independent of the physical location of data inside the memory. Memory cells are located in such a way that they form an array, in which each cell is able to store one bit of information. A memory cell can consist of several transistors or a single pair of a transistor and a capacitor. In the first case, the cells maintain their state while being supplied with power, so they are called Static RAM (SRAM). In the second case, the cells do not hold their state permanently, due to capacitor leakage, and must be periodically refreshed in order to retain information, which leads to a dynamic behaviour and to the name Dynamic RAM (DRAM).

The design differences are crucial for the price, speed, and dimension factors of the mentioned memory constructions as shown in **Table. 2.1**. In fact, the disadvantage of dynamic behaviour is that the processor is not able to read memory while DRAM is being recharged, sometimes causing the CPU to stall while memory is being recharged. However, DRAM has several times more capacity than SRAM and is cheaper. These are the main reasons why DRAMs are widely used in computer main memory blocks.

	SRAM	**DRAM**
Cell Size (F)	50-120	6-8
Access Delay (ns)	1-10	10-30
Access Power	Low	Low-Medium
Leakage Power	High	Medium
Application	Cache	Cache/Memory
$ per GiB	500-1000	10-20

Table 2.1: Comparison of SRAM and DRAM parameters.

2.1.2 Cache associativity

To obtain maximum efficiency from the cache memory, it must be designed and implemented carefully. Different cache placement policies exist with their strengths and weaknesses. The main three of them are: *a) Direct-mapped Cache* - multiple sets with a single cache line per set; *b) Fully-associative Cache* - single set with multiple cache lines; *c) Set-associative Cache* - trade-off between (a) and (b), multiple sets with multiple cache lines per set. Besides that, choosing an optimal replacement policy is one of the key factors determining the effectiveness of a cache, increasing the hit rate (Al-Zoubi et al., 2004).

It is also important to understand how bits of an address given by the processor are mapped to the cache structure. In **Fig. 2.3** you can see how different address bits correspond to different cache parts. The index part of the address selects the set of the cache while the upper part of the address, tag, is compared against the tags from the set. The comparison is followed by the Hit Logic mechanism which decides if it is a hit or not, and which data way to output. The offset bits define the required portion of the data within the cache line. All cache terminology and structure are well-explained in (Smith, 1982).

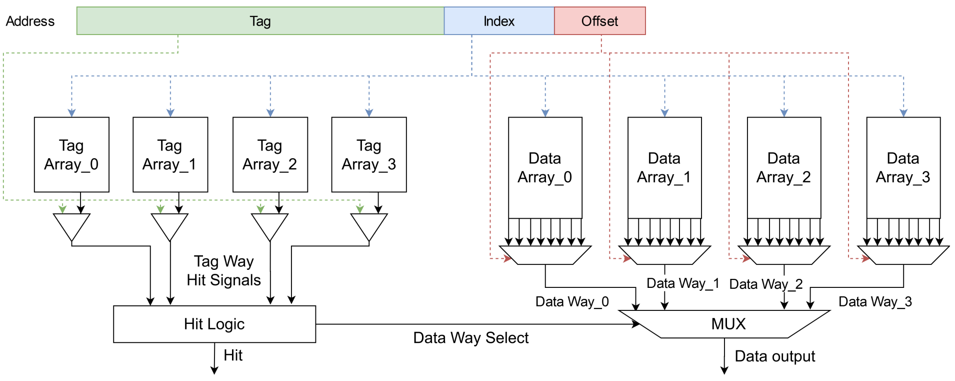


Figure 2.3: Four-way set-associative cache.

2.1.3 Virtual memory

Similarly, how caches provide rapid access to recently used portions of programs instructions and data, the main memory can act as a ”cache” for the secondary memory. This mechanism is called virtual memory. Two main reasons for virtual memory are: to allow efficient and safe memory sharing among multiple programs and to eliminate the programming loads of a limited amount of main memory. In a virtual memory system, each program is compiled in a virtual space, which is dynamically mapped onto the physical memory of the computer at runtime. It means that each program has its own virtual space.

Virtual memory is divided into equal blocks of serial memory locations called virtual pages. These virtual pages are dynamically mapped onto physical pages in the main memory via a set of translation tables called page tables. Pages are brought into page frames on request as programs need them. Since the page table resides in the main memory, the translation process increases memory access latency by having to access memory twice, first to read the page table entry, and then to read the data at the retrieved physical address. To speed up virtual address translation, the system stores current address translations in the translation lookaside buffer (TLB), a separate cache.

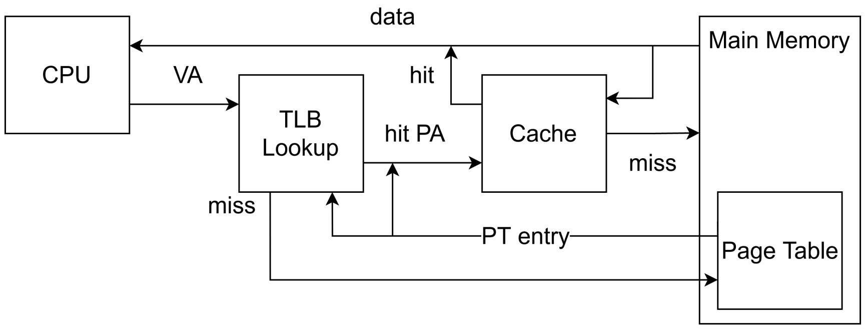


Figure 2.4: Processing the CPU read request.

Fig. 2.4 shows the place of the TLB and page table in handling a CPU request with virtual memory. The virtual address (VA) from the CPU is passed to the TLB. If an entry with such a virtual page number (VPN) exists in the TLB, the physical address (PA) is conveyed to the cache, which handles this address as shown in **Fig. 2.3**. In the case of a TLB miss, it sends the request to the page table (PT) in the main memory in order to retrieve the needed page table entry. It is obvious that increasing the TLB hit rate decreases the time for handling the CPU request.

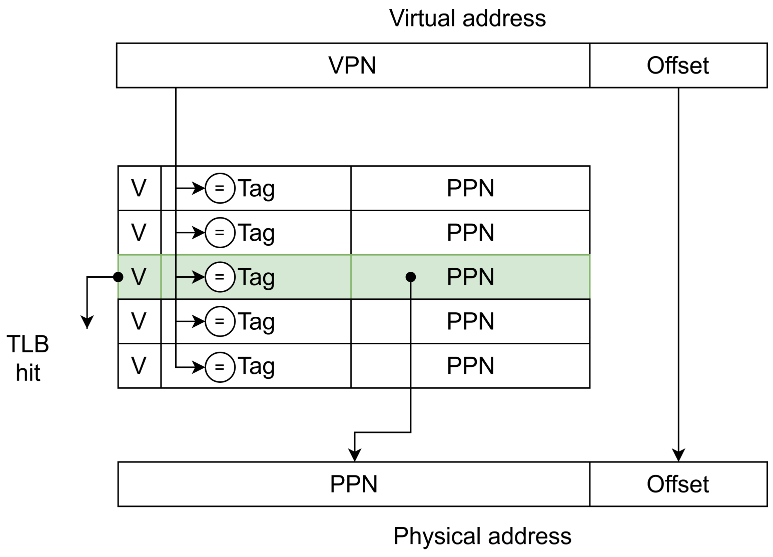


Figure 2.5: High-level overview of a TLB organization.

As mentioned earlier, the TLB contains a subset of virtual-to-physical page mappings that are in the page table. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. **Fig. 2.5** depicts the TLB organization. This diagram focuses on a read with the fully-associative TLB. Implementing such a TLB requires that every TLB tag be compared against VPN since the entry of interest can be anywhere in the TLB. If the valid bit in the matching entry is asserted, the access is a TLB hit, and bits from the PPN together with bits from the offset form the physical address that is used to access the cache.

2.2 The Rocket Chip generator

The Rocket Chip, according to the technical report (Asanović et al., 2016), is an open-source SoC design generator that emits synthesizable RTL. It uses the Chisel hardware-build language to create a library of sophisticated generators for cores, caches and interconnects into an integrated SoC. The Rocket Chip generates general-purpose processor cores that use the open-source RISC-V ISA, and provides both an in-order core generator (Rocket) and an out-of-order core generator (BOOM). The Rocket Chip has been manufactured eleven times and yielded functional silicon prototypes capable of booting Linux. Using RISC-V as an ISA avoids potential licensing constraints from the Rocket Chip and allows the same ISA and infrastructure to be utilized for a wide range of cores, from high-performance out-of-order designs to small embedded processors.

Six main parts compose the Rocket Chip generator (Lee et al., 2016):

- **Core generator:** the scalar core and out-of-order superscalar core generators. Both can contain an optional floating-point unit (FPU), tunable functional unit pipelines, and custom branch predictors
- **Cache generator:** cache and TLB generators whose size, associativity and replacement policies are configurable
- **RoCC-compatible coprocessor generator:** the Rocket Custom Coprocessor interface, a template for application-specific coprocessors that can provide their own parameters
- **Tile generator:** a tile-generator template for cache-coherent tiles
- **TileLink generator:** a generator for networks of cache-coherent agents and the appropriate cache controllers
- **Peripherals:** generators for AMBA-compatible buses and a variety of converters and controllers

More detailed explanation can be found in the official technical report.

It is also worth noting that the Rocket Chip exploits some advanced programming techniques, the aim of which is to implement a powerful and easily configurable system. For instance, *nSets* and *nWays* for the cache are defined in *BaseConfig*[1]. By changing those numbers one can get a Rocket core with different cache parameters. Rocket Chip developers achieve this by using four related code templates such as *Mixins*, *LazyModule*, *Cake pattern*, and *Diplomacy* (Intensivate, 2018).

The Rocket core

The Rocket is an in-order, single-issue scalar processor that includes a six-stage pipeline. The schematic representation of the pipeline is displayed in **Fig. 2.6**. The Rocket core has one integer arithmetic logic unit (ALU) and an optional FPU. An accelerator or coprocessor interface, called RoCC, is also provided.

Fig. 2.7 displays a closer look at the PCGen and Fetch stages of the Rocket core. Instruction fetch is assisted by a branch history table (BHT), which acts as a predictor of the next instruction, a return address stack (RAS) and a branch target buffer (BTB). On instruction request, the index bits of a virtual address are used by the tag and data arrays to define the set and read all the ways from the set. In the next clock-cycle, when the virtual address is translated to the physical address by the TLB, the tag bits from the physical address are compared against the tags from all ways in order to define a hit or a miss and select appropriate data in case of a hit. The data is available for the CPU one clock-cycle

[1]Available: https://github.com/chipsalliance/rocket-chip/blob/master/src/main/scala/system/Configs.scala

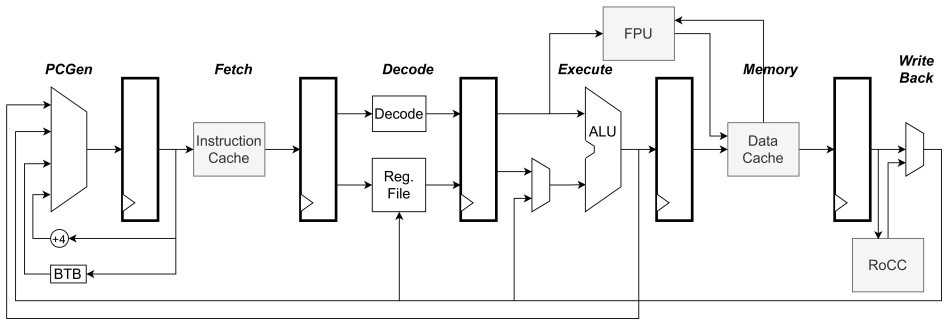


Figure 2.6: The Rocket core 6-stage pipeline.

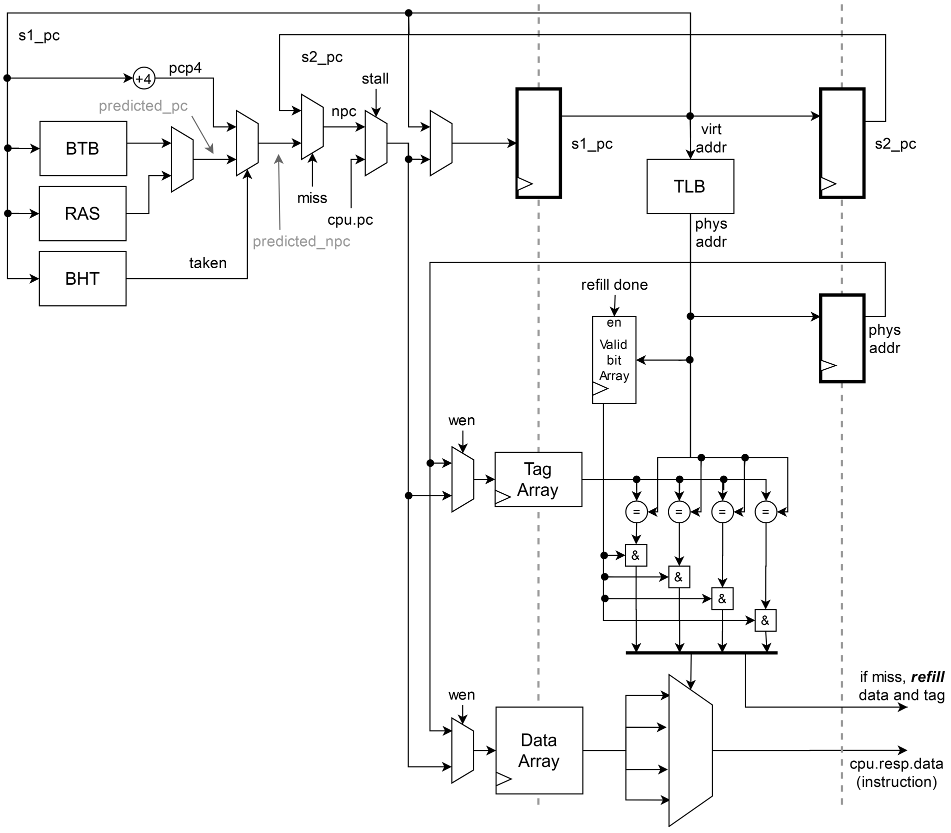


Figure 2.7: PCGen and Fetch stages of the Rocket core.

later. It means that the i-cache latency in case of a hit is two clock-cycles. This access mechanism is called virtually indexed and physically tagged.

To make clear how a read operation in the i-cache is done, **Fig. 2.8** (a larger version is available in the appendix Fig.8.1) displays how the address bits are assigned and used to retrieve the required instruction. For the sake of simplicity, only one 32-bit address is presented, while in reality, bits from both virtual and physical addresses are used for access.

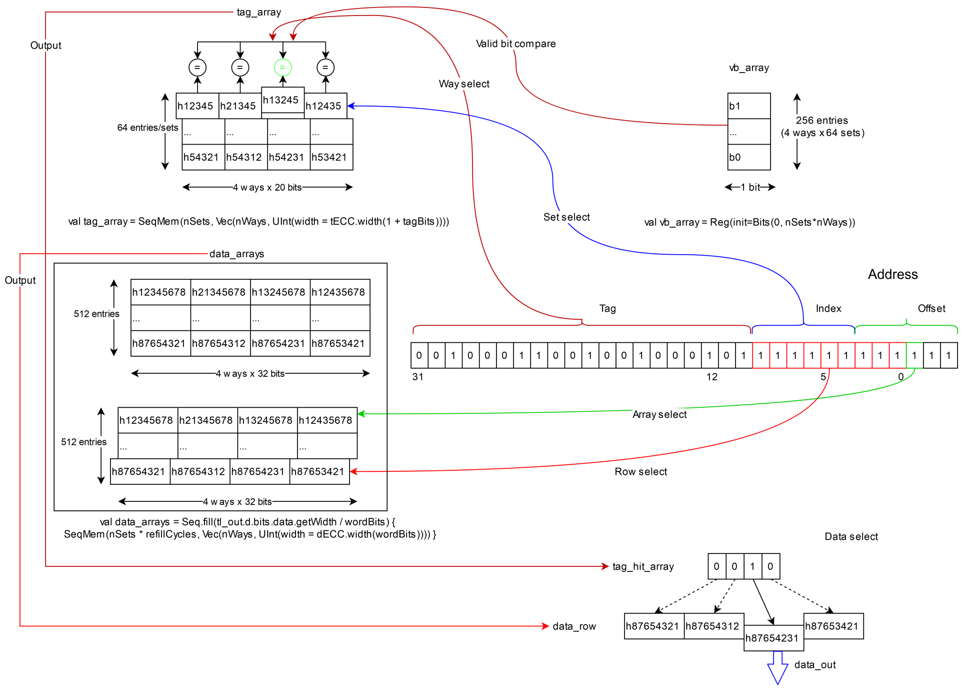

tag_array
Output
Valid bit compare
Way select
vb_array
64 entries/sets
4 ways x 20 bits
256 entries (4 ways x 64 sets)
1 bit
val tag_array = SeqMem(nSets, Vec(nWays, UInt(width = tECC.width(1 + tagBits))))
Set select
val vb_array = Reg(init=Bits(0, nSets*nWays))
data_arrays
Output
512 entries
4 ways x 32 bits
Address
Tag
Index
Offset
Array select
Row select
val data_arrays = Seq.fill(tl_out.d.bits.data.getWidth / wordBits) { SeqMem(nSets * refillCycles, Vec(nWays, UInt(width = dECC.width(wordBits)))) }
Data select
tag_hit_array
data_row
data_out

Figure 2.8: The Rocket core i-cache reading. (Below each storage, the corresponding Chisel line of code for initialization is given).

This illustration reflects a read access for the default-parametrized i-cache. Namely, 64 sets, 4 ways, 32 bits word, 64 bytes block. The incoming address is 32 bits width, six Least Significant Bits (LSB) are reserved for the block offset, the next six bits for the index, and the rest 20 bits for the tag. The i-cache operates with three main storages: (1) the SRAM for the tag array with 64 entries, which corresponds to the number of sets with four tags (ways) for each; (2) the SRAM for the data array which is divided into two sub-arrays of 512 elements of four words each; (3) the register for valid bits array containing 256 single bits for all tags in the tag array. To find the requested instruction, the tag and data arrays are accessed simultaneously. The index part of the address defines the set from the tag array, while the tag field is used to probe each way in the set considering the tag's valid bit. Meanwhile, the combination of the index and offset bits is used to access the data array, it means that the third offset bit corresponds to the sub-array selection, while six bits of the index and three Most Significant Bits (MSB) of the offset are used to choose the entry from that sub-array. As a result, we have the tag hit array which detects the selected way and the vector of 128 bits from the data array. Then, these two parts are multiplexed outputting the desired instruction of 32 bits.

2.2.1 The RISC-V ISA

RISC-V is an open-source hardware ISA based on established Reduced Instruction Set Computer principal. As the designers of RISC-V state, this instruction set is for practical computers despite its academic background. Designers claim that it has features to increase computer speed, yet decrease power use and cost (Waterman and Asanovic, 2019). First, RISC-V exploits load-store architecture, it means that only load and store instruction access memory and arithmetic instruction only uses CPU registers. Second, it has a simplistic standards-based floating-point. Furthermore, this instruction set placed most-significant bits at a fixed location to speed-up sign extension.

RISC-V supports three widths for the word: 32, 64 and 128 bits. A variety of subsets also exists, which supports the range of machines from huge rack-mounted parallel computers to small embedded systems. The instruction set has a modular design with added optional extensions providing alternative base parts. RISC-V machines have an option to implement a compact extension to reduce power consumption, code size and memory usage.

Four instruction formats (R/I/S/U), shown in **Fig. 2.9**, are the core of the RISC-V ISA. Each of them is a fixed 32 bits in length, which must be aligned on a four-byte boundary in memory. The same position for the source (*rs1* and *rs2*) and destination (*rd*) registers are kept in order to simplify and accelerate decoding.

Furthermore, designers of RISC-V run the Foundation project, which has already attracted 325 members most of whom are world-famous (UCB, 2019b). The key benefits that proposed by technology are: *a) Software architects/developers* - the base instructions and optional extensions are frozen, aka stable; *b) Chip designers/System architect* - ISA open-source nature similar to everyone having microarchitecture license. Custom optimized design; *c) Board designers* - RISC-V is royalty-free this creates significant flexibility to port a RISC-V based design from an FPGA to an ASIC or another FPGA without any software modifications.

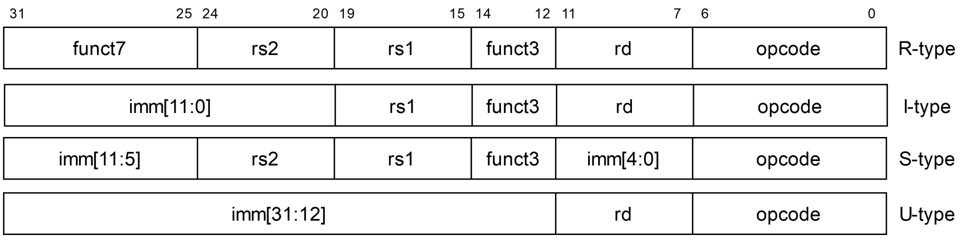


Figure 2.9: RISC-V base instruction formats (Waterman and Asanovic, 2019).

2.2.2 Chisel

The RISC-V design software includes a design compiler based on the Scala programming language, Chisel, which can convert the design to Verilog for use in a device. Instead of building a new Hardware Description Language (HDL), UC Berkeley developers decided to embed hardware construction primitives within the Scala language. There are several reasons why developers preferred Scala, including its power with features, domain-specialty, compilation for Java Virtual Machine (JVM), large and growing user community. The first prototype of the processor was written using Chisel by UC Berkeley developers for educational purposes. Now, two of the cores written in Chisel, the BOOM and the Rocket, are even used for commercial purposes by SiFive fabless semiconductor company established in 2015 by three researchers from UC Berkeley.

Chisel was presented for the first time in 2012 (Bachrach et al., 2012). It was introduced as a simple platform that supplies modern programming language features for thoroughly specifying low-level hardware blocks, meanwhile, it can be easily extended to cover many helpful high-level hardware design schemes. Chisel can generate rapid cycle-accurate emulators for design as well as produce low-level Verilog code which is suitable for an FPGA or an ASIC simulation and synthesis using standard tools.

2.3 Field-programmable gate array

In the mid-1980s manufacturers of integrated circuits came up with the smartest idea. The idea was to allow customers to customize the implemented logic of the circuit (Willert, 1999). The FPGA industry sprouted from this clever idea being one of the fast-growing nowadays. Modern FPGAs compete very well on performance, price, and usability against many standard off-the-shelf devices. FPGA offers the customer the key advantage of profitability - period, it means that users can get their products to market sooner and keep it in the market longer than with any other technique.

FPGA consists of basic logic circuits like encoder, decoder, multiplexers and several Look-Up Tables (LUT) integrated into a Configurable Logic Block (CLB). Designers can specify the operations to perform by CLB writing a program in any hardware description language including Chisel (Lennon and Gahan, 2018). In turn, software programs are used to describe connections and interface signals for each module and the functionality of the design.

Contemporary FPGAs are being manufactured with embedded hardware in which designers can add intelligence systems through software. Operations can be executed in real-time programmable hardware and system interfaces through programmable Input/Output that makes an FPGA a complete SoC solution (Xilinx, 2014).

Chapter 3

Implementation

The goal of this thesis was to integrate the virtual-address-matching technique into the Rocket core architecture to evaluate its efficiency and ability to lower the energy dissipation. This chapter covers the implementation of the VAM and i-cache modifications needed to perform this implementation.

3.1 Virtual-address-matching (VAM)

The virtual-address-matching mechanism seeks out the set and way which contains the desired data via virtual address matching. The virtual tag and virtual index are used in the VAM to determine the location of the valid data. Furthermore, virtual tag bits of a virtual address are translated into a physical address through the TLB. A matching table unit (MTU) is added as shown in **Fig. 3.1**, which is similar to a small TLB. The difference is in function and capacity. The function of the TLB is to convert the virtual address from the processor into the physical address which can be used directly for the cache and memory access, while the MTU is to map the virtual address to the corresponding way of the desired data in the cache. As the TLB, the MTU is fully-associative mapping but much smaller than the TLB. For indicating whether there is a mapping of the i-cache block in the MTU, a one-bit M field is added for each block in the cache.

When the CPU requires an instruction, it passes the virtual address of the instruction not only to the TLB but also to the MTU. As shown in **Fig. 3.1**, the TLB utilizes only the tag of the virtual address, aka the VPN, while the MTU uses the tag+index bits. If the tag+index of the requested virtual address exists in the MTU, the MTU search gives a match quickly and an MTU hit is asserted. In this scenario, the CPU will not wait for the TLB searching and directly access the data in the cache line to which the matched entry in the MTU is mapped. Therefore, the energy for conventional tag-matching and redundant way-precharged is eliminated. In contrast, if the tag+index is not in the MTU, an MTU miss is produced. In this case, the i-cache will be accessed as the conventional set-associative cache. As a result, if an MTU miss occurs, the energy consumption is the same as that of a conventional set-associative cache. However, since the MTU searching

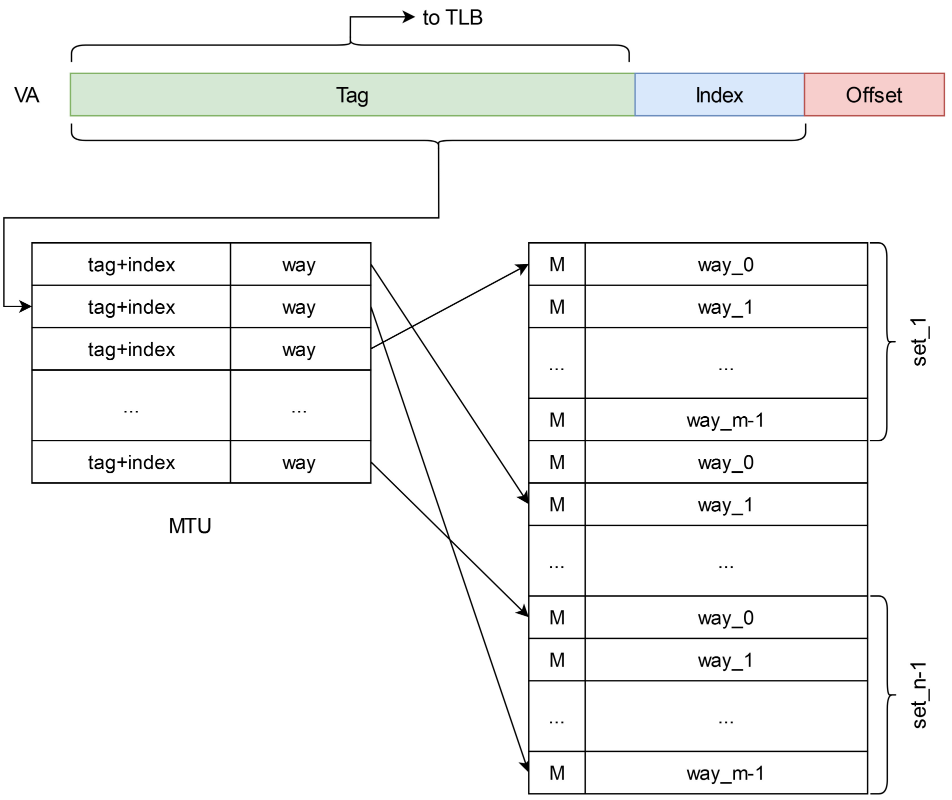


Figure 3.1: The VAM working principal.

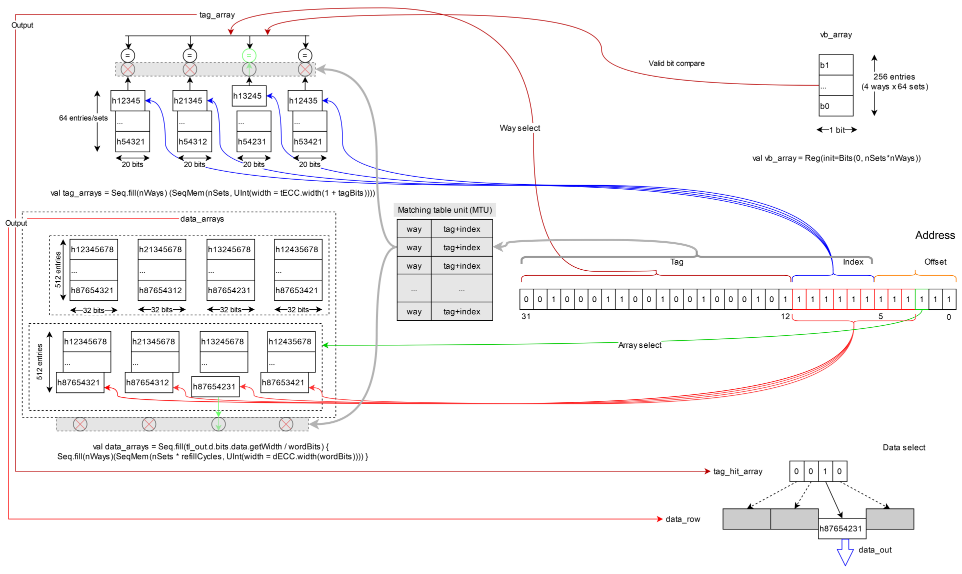
is parallel to the TLB's address translating, the i-cache access can finish within one cycle whether the MTU hits or misses.

In the original paper (Yang and Li, 2010), the MTU is implemented as content-addressable memory (CAM) (Kenneth, 1997), however, how to find the appropriate entry in the MTU when the system needs to remap it in the case of a tag replacement in the tag array is not described clearly. In our implementation, the MTU is built on registers, and in the case of a cache miss, we keep the virtual address until the new tag is written to the tag array, so we can find the appropriate entry in the MTU to remap.

3.2 Rocket Chip i-cache structure modification

Since the way-prediction mechanism is not provided for the Rocket core, the tag and data arrays represented as SRAMs, are coarse-grained, i.e. each element of the array is a big array that contains the information for all four ways and is accessed at once. Thus, the first modification in the Rocket core i-cache that we needed to make was making the granularity finer, so as to access a single way. These changes implied modifications in both logics, reading and writing to the i-cache. Note that these modifications are needed to implement any of the way-prediction techniques since we have to either enable or disable access to separate ways.

Fig. 3.2 shows the overview of modifications for the i-cache reading access superimposed on the previously presented diagram (**Fig. 2.8**). The shaded parts display the added blocks that enhance the reading logic by the way-prediction mechanism. The MTU, the implementation of which is described in the next subsection, is checked to find a match between the tag+index from the virtual address and then the way. In the case of an MTU hit, only one way in the tag and data arrays is probed. Otherwise, all ways are accessed in parallel, and the read access proceeds as for a conventional cache. In this scheme, the tag and data arrays consist of *nWays* SRAMs, which can be accessed independently. **Fig. 3.3** depicts the array transformation and corresponding memory declaration Chisel codes.

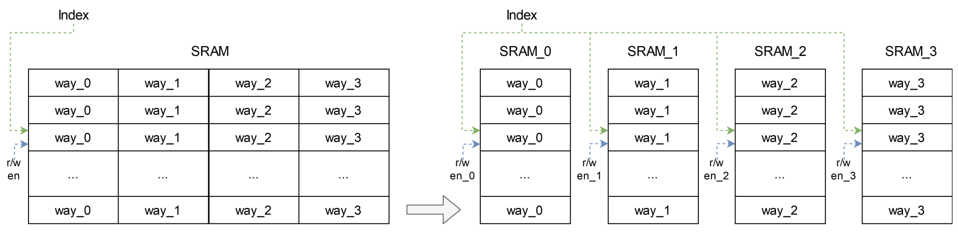


```
val tag_arrays = Seq.fill(nWays) (SeqMem(nSets, UInt(width = tECC.width(1 + tagBits))))
```

```
val vb_array = Reg(init=Bits(0, nSets*nWays))
```

```
val data_arrays = Seq.fill(tl_out.d.bits.data.getWidth / wordBits) {
Seq.fill(nWays)(SeqMem(nSets * refillCycles, UInt(width = dECC.width(wordBits)))) }
```

Figure 3.2: The Rocket core i-cache modified reading.



```
val tag_array = SeqMem(nSets, Vec(nWays, UInt(width = tagBits)))
```

```
val tag_arrays = Seq.fill(nWays) (SeqMem(nSets, UInt(width = tagBits)))
```

Figure 3.3: Transformation principle of the Rocket core tag array.

Since a read/write enable signal (r/w en) can be sent separately for each way, reducing energy consumption per tag and data arrays access may potentially range between 0 and

75%. However, the accuracy of the way-prediction in this case mostly depends on the number of entries in the MTU and replacement scheme. Both can be defined, for instance, with the help of simulation. The results and possible overheads of the tag and data arrays transformation are presented and evaluated in Chapter 5.

3.3 Matching table unit implementation

The main component of the VAM way-prediction technique is a matching table unit. Parameters of the MTU, like the number of entries and the width of a tag+index field, define the prediction accuracy and the cost of area and complexity overhead. In this project, the VAM was implemented in several steps, where the results of each step are compared against the results of the previous steps and conventional cache implementation in terms of performance degradation, area overhead and energy consumption reduction. The results are shown in Chapter 5.

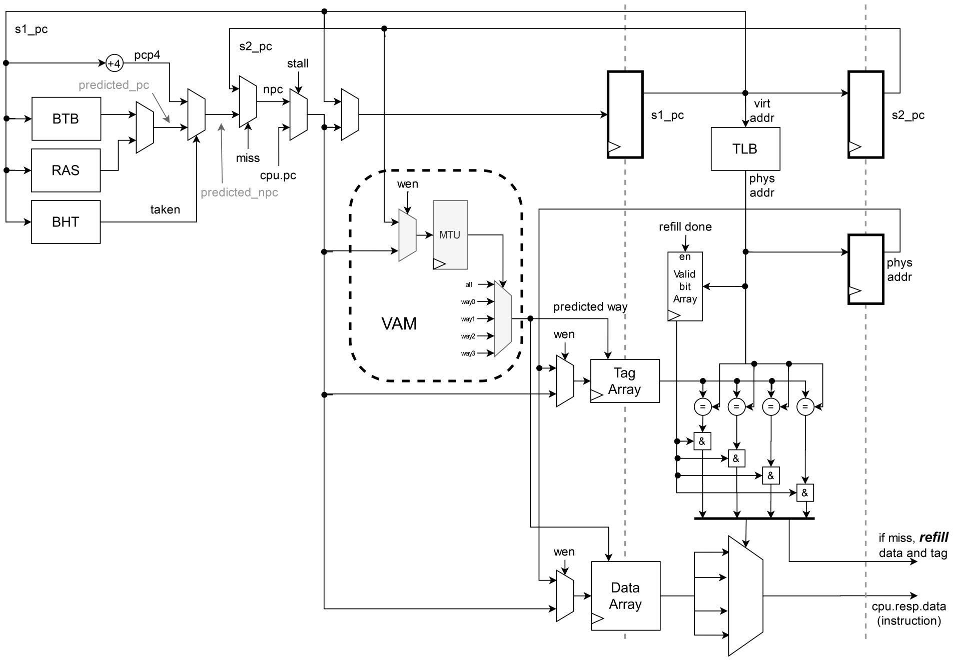


Figure 3.4: Embedding the MTU into the Fetch stage.

Fig. 3.4 shows the place of the MTU in the fetch stage of the Rocket core. When a cache miss occurs, the *refill* bit asserts and signals that the L1 i-cache line has to be filled by instructions brought from a lower-level memory. When the data array refill is completed, the i-cache writes the corresponding tag from the physical address to the tag array. On writing the tag to the tag array, the system is also aware of the corresponding virtual address which tag and index bits are used to create a record in the MTU mapping to the appropriate way. However, when the CPU generates an instruction request, the way search is initiated in the MTU based on the tag+index bits from the virtual address. In the case of an MTU miss, all the ways in the tag and data arrays are probed.

As we can see, the mapping mechanism is on the timing path of the data and tag arrays, which might affect the critical-path of the core. This influence will be also discussed in Chapter 5.

Furthermore, virtual memory does not always applies one-to-one mapping between virtual and physical memory. Several virtual addresses can point to the same physical location; this is called *synonym*. Conversely, if single virtual address reused by more than one process points to multiple physical addresses, it is known as *homonym*. These synonyms and homonyms can potentially affect the efficiency of the VAM. In the case of synonyms, two different VAs would point to the same set in the MTU since these VAs has the identical index bits. It means when the first VA is recorded to the MTU, the request with the second VA will not succeed in finding any match in the MTU, while actually the relevant data resides in the cache. To avoid unnecessary low-level memory access, we can use Population Count function that counts the number of set bits in the input signal. In the case of no matching found in the MTU, enable the read signal for all ways in the tag and data arrays. However, it is more difficult to detect and handle homonyms, which may occur in multiprocessing tasks. In the case of homonyms, we have only one VA, which maps to only one way in the MTU, while the requested data may reside in two different ways in the cache depending on the process's address space. It means that we need to use processes identifiers as a differentiator between their VAs or flush the MTU in each context switch. We have decided to keep the design simplicity in this project and leave the implementation of homonyms handling for future work.

3.3.1 Direct mapping

For the first stage, we decided to implement the mapping table unit as a register. It contains exactly the same number of entries, *nWays* x *nSets*, as the tag array, so that each set of tag+index bits in the MTU unambiguously maps to a single way from the tag array. When an entry in the tag and data arrays is refilled on an i-cache miss, the same entry in the MTU is also filled by bits from the corresponding virtual address which is conveyed from the previous clock-cycles. The processing of each instruction request in the fetch stage begins from translating the virtual address to the physical address by means of the TLB and simultaneously searching for a way that is mapped to the combination of tag and index bits from the virtual address in the MTU. The matching process in the MTU is much faster than that in the TLB since we have to check *nWays* number of entries because the index field of the VA defines the set. Thus, the quick comparison can detect whether there is a mapping in the MTU or not, providing a way or ways for access to the tags and data reading logic.

Fig. 3.5 shows how writing and reading processes in the MTU correspond to that of the SRAM. The mechanism is described for a four-way set-associative instruction cache. For the sake of simplicity, only the tag array present in the picture. The data SRAM uses the same signals for reading and writing. The VA index is used to define a set in the MTU, within which the way has to be predicted. The content of each of the four MTU entries is compared against the VA tag+index. At this stage, we have to detect whether there is any matching or not. A Population Count (PopCount) function is an in-build Chisel

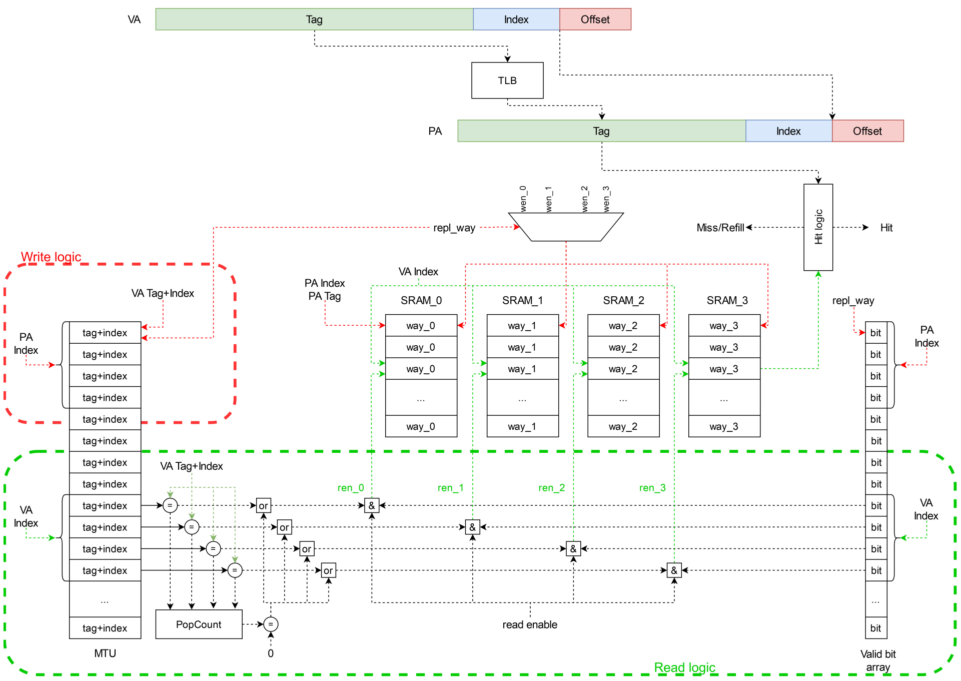


Figure 3.5: The MTU reading and writing logic.

function that returns the number of bits set in the input signal. Returning 0, this function indicates that there is an MTU miss and all four ways in the SRAM have to be probed. Furthermore, the valid bits of the corresponding ways are also ANDED with a read enable signal since it makes no sense to access a tag or data which is known as invalid. When a cache miss occurs, a replacement policy decides on which way (*repl_way*) to rewrite by a new tag and data in the SRAM, but the PA index is used to define the set to write[1]. The same replacement way and PA index choose the entry in the MTU to write, but the VA tag+index bits are what we need to write.

At this stage of implementation, the number of tag and data SRAM accesses is reduced but we still need to access all the ways in the MTU and compare a significant amount of tag+index bits, which negates the energy reduction.

3.3.2 Reducing entry bits

The next step of the implementation was to decrease the number of bits stored in the MTU. One effective way to do so is to map the VA bits to a smaller amount of bits using a hash function. A similar technique is used in AMD processors architectures (AMD, 2017) for way-prediction where the cache tags contain a virtual-address-based microtag (a hash

[1]The feature of the Rocket core i-cache implementation. Since it is virtually indexed - physically tagged, the VA and PA indies are the same, however the index is retrieved from the VA, which is given by the CPU at the first clock-cycle and from the PA at the next clock-cycle, when the PA is provided by the TLB

of the VA) that marks each cache line with the VA that was used to access this cache line initially. This hashed tag is used to determine which way of the cache to read.

In order to implement such a hash function, exclusive-OR-based (XOR-based) hash functions are usually used. There are several types of XOR-based hash functions, but in this project, we used a polynomial function since it is easy to implement, and it was proposed to use polynomial hash functions for caches due to their great performance in the presence of strides[2] (Vandierendonck and DeBosschere, 2005). Furthermore, researchers from the Graz University of Technology, Austria, exploring the vulnerability of AMD processors, have recovered the XOR function which is used to produce the microtag for the way-predictor (Lipp et al., 2020). The research has shown that AMD's way-predictor also uses a polynomial function to map the virtual addresses to the microtags, XORING two equal parts of the VA.

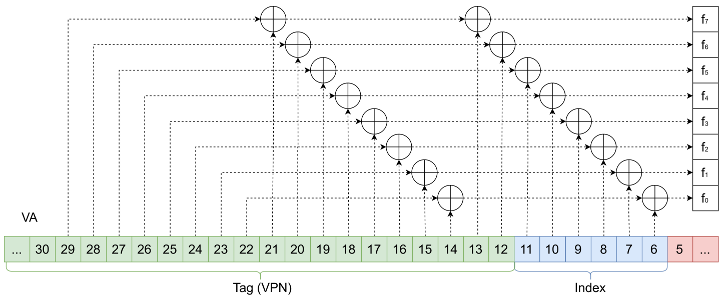


Figure 3.6: Polynomial XOR-based hash function uses bits 6 to 29 of the VA to compute the hashed tag.

Our hash function XORS three equal-length slices of the VA bits **Fig. 3.6**. It is assumed that the hash function maps $n = 24$ address bits to $m = 8$ bits of an MTU entry. Before passing the tag+index bits of the VA to the MTU for both read and write operations, the XOR function is performed on them. Thus, when we are seeking the way within the MTU we need to compare only 8 bits for each entry in the set. We have also experimented with others XOR mapping schemes such as 24:12 and 24:6, and the results are presented in Chapter 5.

The cache memory exploits the locality of application programs, enabling a small and fast cache to satisfy most memory requests issued by a processor. If the application programs exhibit memory access localities, then the most tag bits of successive CPU requests will be the same, except for a few differences in the least significant bits (Kwak and Jeon, 2010). This allowed us to exclude several most significant bits of the VA from the hash function computations.

Fig. 3.7 represents the MTU implementation with the hash function. The MTU entry stores the *hTag* which is the 8-bit output of the XOR-base function. As a result, the MTU size has reduced, and the number of bits for comparison is decreased from (*nWays* x 24) to (*nWays* x 8) bits. However, the amount of entries of the MTU still depends on the tag array size since the MTU searching process performs the same logic as for the tag SRAM, defining the set, then comparing the content of each way.

[2]The number of locations in memory between beginnings of successive array elements

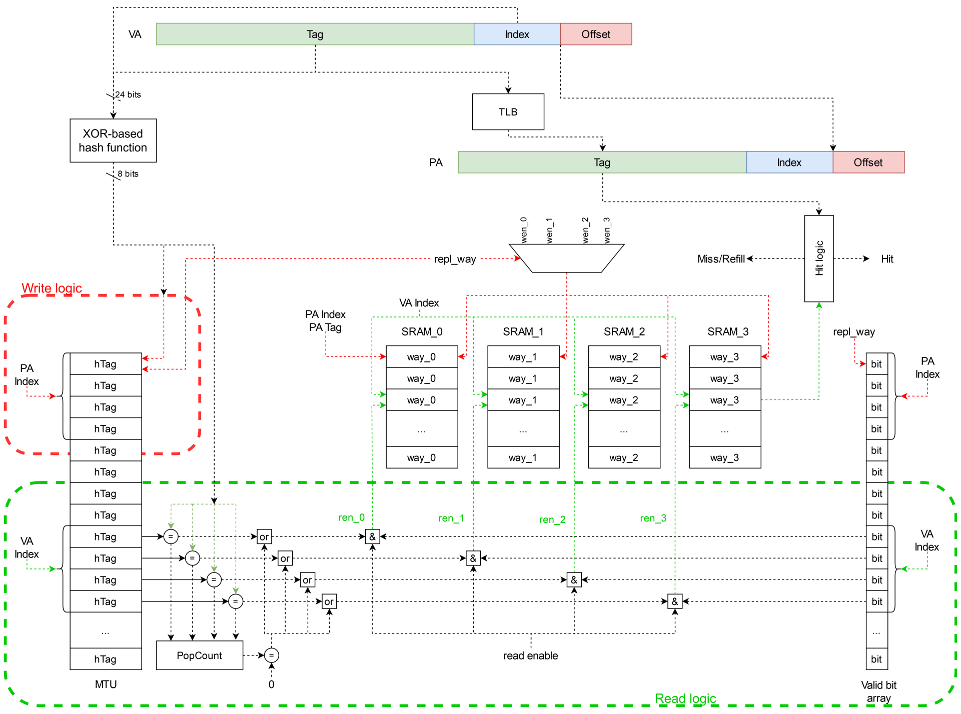


Figure 3.7: The hashed MTU reading and writing logic.

3.3.3 XOR-based mapped MTU

To reduce the area overhead that the big MTU may cause, we changed the MTU addressing mechanism by means of the same XOR-based hash function using an XOR-mapping scheme. The idea of using XOR functions to map memory address to a set of memory entries has been studied extensively, especially in the context of interleaved memories (Gonzalez et al., 1997).

In our case, the use of XOR-mapping schemes requires the computation of several XOR operations to derive the MTU index. Since all eight XOR operations within one XOR line (**Fig. 3.6**) can be done in parallel, the delay of this computation is just one XOR gate. However, the computation of these XOR operations starts to execute simultaneously with the TLB translation but performs much faster, so that this delay may not affect the whole i-cache performance.

The XOR-mapping scheme allows to accessing the MTU elements by their index, which is the result of the XOR-based hash function. To implement this mapping to the MTU, we utilized the XOR function that was described in the previous subsection and maps 24 to 8 bits dividing the tag+index bits set of the VA into three equal-length parts. It means that the MTU contains 2^8 entries. The width of each entry is *log(nWays)* bits since the content is the value of the way where the data related to this VA was written last time. For the 32-bit RISC-V instruction, each Rocket core i-cache line can contain 16

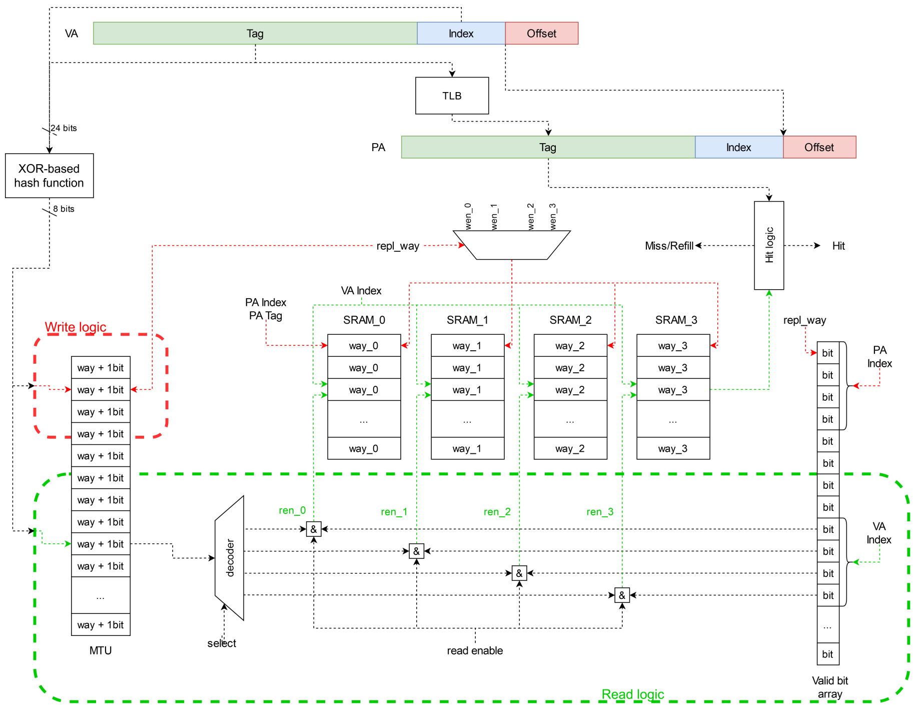


Figure 3.8: The XOR-mapped MTU implementation scheme.

instructions if the cache is 64 bytes per line. The MTU of 256 entries can cover 4096 (64 lines x 16 instructions per line) instructions.

Fig. 3.8 shows the schematic representation of the XOR-mapped MTU implementation. The hash function generates an 8-bit MTU index, which is used to define the corresponding entry in the MTU which is 256 (2^8) entries in size. Then, the content of the found entry is decoded to determine which way or ways of tag and data arrays should be enabled to read. We added one bit in each MTU entry in order to detect whether the record in the MTU is mapped to some way in the tag SRAM or not. For example, for four-way set-associative cache, the MTU is initiated by three bits of value '100'. The third bit is '1', means that this entry is not mapped yet to any way in the tag array. However, when a tag is written to the SRAM to the way '10' the corresponding record in the MTU becomes '010'. Thus, the selection logic for the decoder can assert all four outputs in case of '100' input signal to enable all four ways for reading.

Even if we use the same hash function as in the previous stage, this implementation utilizes the output of the XOR function in a different way, which distinguishes the approach of addressing the MTU. The potential benefit of this approach is that the size of the MTU does not depend on the tag array size, but it is adjusted by the XOR-based mapping scheme. That is, if we use a 24:6 scheme, the size of the MTU will be 64 (2^6) entries. Furthermore, we don't need to read all the ways from the MTU, but only one, which dramatically reduces the energy consumed by the MTU. However, compared to the XORED direct-mapped MTU, the XOR-based mapped MTU may suffer from reduced prediction accuracy since such a mapping mechanism may cause more address collisions in the MTU. In this case, choosing the right collision-free XOR function may help in solving this issue.

Chapter 4

Methodology

To evaluate the results, we utilized standard tools offered by the Rocket Chip generator. Chisel can generate code for three targets: (1) Verilog code for Very Large Scale Integration (VLSI), (2) a high-performance cycle-accurate Verilator, (3) Verilog code optimized for FPGAs. The Rocket Chip generator can target all three backends.

The Verilator is a free and open-source software tool that converts Verilog to a cycle-accurate behavioral model in C++ or System-C. Running *make -jN run* command from *rocket-chip/emulator*[1] directory, will generate C++ code for cycle-accurate emulator, compile the emulator, compile all RISC-V assembly tests and benchmarks[2], and run both tests and benchmarks on emulator. The output files of the executed assembly tests and benchmarks can be found at *rocket-chip/emulator/output/*.out*. Each file has a cycle-by-cycle dump of a write-back stage of the pipeline which is used to evaluate the performance. Additionally, vcd waveforms can be generated to observe the propagation of signals. The extended list of RISC-V Software Ecosystem[3] contains different types of simulators, debugging systems, toolchain, compilers, and libraries.

Since SiFive has already a manufactured version of the Rocket cores, its open-source repository has been used for this project. This repository[4] contains the RTL for SiFive's Freedom E300 and U500 platforms. The Freedom E310 Arty FPGA Dev Kit implements the Freedom E300 platform and is designed to be mapped onto an Arty FPGA Evaluation Kit[5]. For the purpose of this project, particularly for evaluating the area overhead of the proposed implementation, and delay that the integrated components add to the fetch stage, the Freedom was remapped to the Xilinx Zynq-7000 Series PYNQ-Z1 FPGA with 50 MHz clock frequency. This implementation is available on GitHub[6].

The parameters of the device, xa7z020clg400, which is chosen for this project as a target for synthesis and implementation are presented in **Table. 4.1** (Xilinx, 2018a,b).

[1] Available: https://github.com/chipsalliance/rocket-chip/tree/master/emulator
[2] Available: https://github.com/chipsalliance/rocket-tools
[3] Available: https://riscv.org/software-status/
[4] Available: https://github.com/sifive/freedom
[5] Available: https://www.xilinx.com/products/boards-and-kits/arty.html
[6] Available: https://github.com/hakimahunov/freedom/tree/pynq

Technology (nm)	28
Logic cells	85,000
LUTs	53,200
FFs	106,400
BRAM (KB)	630
DSP slices	220

Table 4.1: The target FPGA parameters.

The Rocket core caches are easily configurable. For this project, the configuration parameters for the i-cache have been modified for various cases in the process of evaluation. However, in most cases, the i-cache parameters are configured to the default values presented in **Table. 4.2**.

Number of sets	64
Number of ways	4
Word bits	32
Block bytes	64
TLB entries	32

Table 4.2: Default instruction cache configuration for the Rocket core.

The efficiency of the VAM implementations has been evaluated using the benchmarks which are pre-written by the Rocket Chip generator developers. The following five benchmarks were used for the tests:

- **Dhrystone:** a widely used integer benchmark that does not contain any floating-point operation
- **Qsort:** this test uses the quicksort algorithm to sort an array of integers into ascending order
- **SPMV:** this test executes double-precision sparse matrix-vector multiplication
- **MM:** this test for matrix multiplication. Both blocked and unblocked implementations
- **MT-VVADD:** this benchmark adds two vectors and writes the results to a third vector

All tests were performed on a computer running the Rocket Chip emulator. No testing on hardware was performed.

The implementation of all stages described in Chapter 3 was evaluated running the benchmarks individually on the Rocket core. This made it possible to compare results between different MTU implementations steps. Additional performance counters like hits and misses counters, MTU prediction counters were implemented in order to evaluate the output results. The energy savings presented in Chapter 5 were also evaluated with the aid of these performance counters. The performance of the implemented techniques have been computed using the equations (4.1) and (4.2):

$$Performance = 1 \,/\, Execution\, time \tag{4.1}$$

$$Execution\, time = I \times CPI \times T \tag{4.2}$$

where I - the number of instructions, CPI - cycles per instruction, T - cycle time, which is set to the critical path delay for a certain implementation.

The unit tests were applied to the added components such as the MTU, population counter, XOR-based hash function in order to check the correctness of their functionality.

4.1 Energy consumption evaluation

Equations (4.3) and (4.4) were used to calculate the energy consumption for the conventional Rocket core i-cache read/write access, while equations (4.5) and (4.6) have been used to calculate the i-cache read/write access with the implemented way-prediction mechanism.

$$E_{cacheR-init} = E_{tagR} \times N_{tagR} + E_{dataR} \times N_{dataR} \tag{4.3}$$

$$E_{cacheW-init} = E_{dataW} \times NW_{dataW} + E_{tagW} \tag{4.4}$$

$$E_{cacheR-pred} = E_{mtuR} \times N_{mtuR} + E_{tagR} \times N_{tagR} + E_{dataR} \times N_{dataR} \tag{4.5}$$

$$E_{cacheW-pred} = E_{dataW} \times NW_{dataW} + E_{tagW} + E_{mtuW} \tag{4.6}$$

where, for example, E_{tagR}/E_{tagW} is the average energy consumed when the cache tag of the cache way is accessed for a read/write, N - the number of ways to read, NW - the number of words in a cache block.

The values for tag and data arrays energy consumption are sourced from (Moreau et al., 2016) research paper and are presented in **Table. 4.3**. These numbers of access energy are retained by implementation of 16kB 4-way data and instruction caches in the context of a 5-stage in-order processor. The RTL implementation of the pipeline was synthesised based on a commercial 65-nm 1.2-V CMOS low-power process technology, with standard cells and mixed-V_T SRAM macros. All our estimations of the MTU accesses are done pessimistically. We assumed that access to the register consumes twice as much energy per bit as access to the tag SRAM and that the energy consumption for the XOR and the comparison operations are included in these numbers.

Operation	**Energy (pJ)**
Tag read	19.1
Data read	26.5
Tag write	17.6
Data write	27.2
MTU read (24 bits)	45.8
MTU read (8 bits)	15.3
MTU read (3 bits)	5.7
MTU write (24 bits)	41.3
MTU write (8 bits)	14.8
MTU write (3 bits)	5.3

Table 4.3: Energy consumption for different parts of the L1 cache.

4.2 Area overhead and timing evaluation

The area occupied by an integrated circuit is an important factor in its design since explicit and implicit physical constraints have to be always met. In this project, we estimated the area overhead caused by each stage of the VAM implementation using the Vivado utilization report (Xilinx, 2019). It breaks down the design utilization based on the resource type. We were focusing on three of them: LUTs, FFs, and BRAM.

The timing evaluation has been performed using dedicated Vivado timing report, which provides detailed information on the critical path.

Chapter 5

Results

In this section, all the execution results from the existing Rocket Chip benchmarks are presented. The results are displayed either in tabular form or as a graph. The first and the second subsections cover the results for the stage of implementation when the granularity of the SRAM is reduced, and the MTU is addressed by the index bit of the VA respectively. The third subsection covers the results, which are achieved by reducing the number of the storing and comparing bits in the MTU using the XOR-based hash function. The final subsection shows the results from the reduced XOR-mapped MTU.

5.1 Results for separated SRAM

Running synthesis and implementation on Vivado tools for both coarse-grained and fine-grained Rocket core SRAM implementations, show that the separated way access affect neither area nor performance for the selected FPGA. **Table. 5.1** shows the FPGA resources utilization report for the split four-way set-associative cache, and these figures are treated as a baseline for this work. The timing analysis, the results of which are shown in **Table. 5.2**, indicates that all specified timing constraints of the digital design are met, and it is presented here just for informational purposes.

Resource	Utilization	Available	Utilization %
LUT	31403	53200	59.03
FF	16342	106400	15.36
BRAM	24	140	17.14

Table 5.1: Report of the device resource utilization generated from the implemented design netlist.

The number of clock-cycles that each of the benchmarks executes are depicted in chart form in **Fig.5.1**. This performance indicator is also not affected by splitting the ways in the tag and data arrays.

Setup	Worst Negative Slack (WNS)	0.633ns
	Total Negative Slack (TNS)	0.000ns
	Total Number of Endpoints	44663
Hold	Worst Hold Slack (WHS)	0.011ns
	Total Hold Slack (THS)	0.000ns
	Total Number of Endpoints	44663

Table 5.2: Comprehensive sign-off quality timing report.

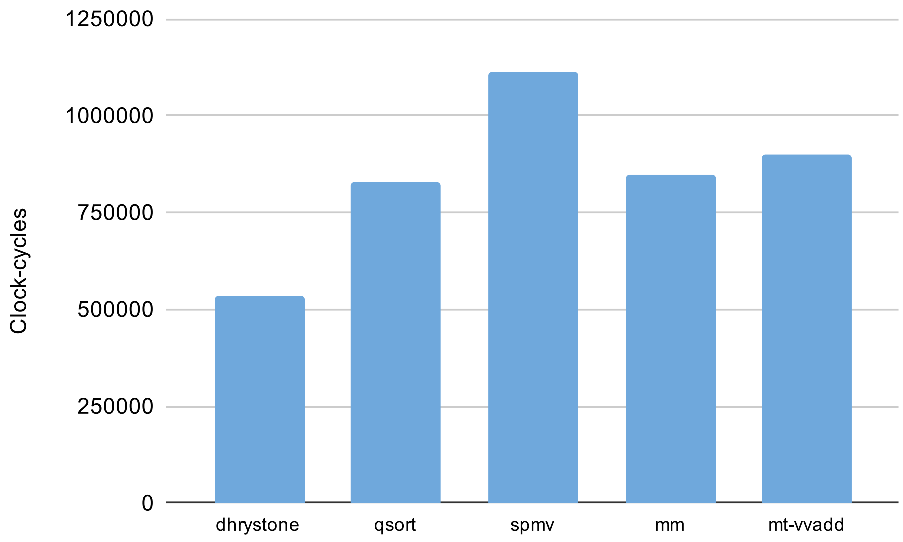


Figure 5.1: Benchmarks execution time in clock-cycles.

After splitting the SRAM access, but before adding the MTU, we have extended the SRAM reading logic of each way by the AND gate with two inputs: the read-enable signal and the valid bit value of the appropriate way from the valid bit array. Since this operation performs for all the ways in parallel, it adds just one AND gate delay. However, this modification affects the i-cache energy consumption reduction, since it allows to avoid reading tags and data on cold misses[1] or when the cache is invalidated for some reason.

5.2 Results for direct mapped MTU

Adding the MTU to the initial Rocket core i-cache design has not affected the number of execution clock-cycles of the benchmarks, hence miss and hit rates remain the same. **Fig.5.2** shows the miss rates for the baseline implementation cache, the cache with VAM

[1]The first reference to a block of memory, starting with an empty cache.

predictor and for the MTU accesses. The values for the modified and initial caches exactly match, while the MTU miss rate is insignificantly higher than that value for the cache by 0.2% on average.

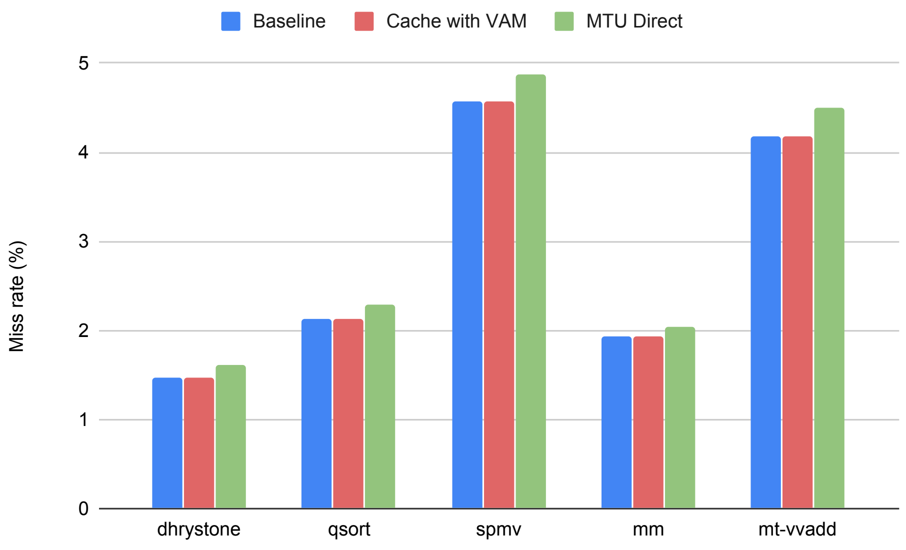


Figure 5.2: Miss rates.

Practically, at this stage, the MTU represents a duplicate of the tag array that can be read before starting reading the SRAM. However, this MTU is more expensive, since it is built on registers, and consumes thousands of additional FFs and LUTs increasing the area overhead. This implementation utilized almost 4% more LUTs and 6.4% more FFs. **Fig.5.3** represents the area chart for the cache with the VAM and the direct-mapped MTU compared to the conventional cache implementation.

Based on the implementation report from Vivado, the MTU match added a delay of 2.003 ns to the i-cache circuit. It was a critical delay that violated the timing constraints. Furthermore, the MTU implemented in this way was ineffective in terms of energy savings. Finding, reading and comparing the MTU entries of 24 bits increased the cache power consumption by 26% on average.

5.3 Results for XOR-based hashed MTU

In order to decrease the energy consumed by the MTU, we have reduced the number of bits stored in the MTU from 24 to 8 bits, thus eliminating the need to read and compare the large number of bits from the MTU. First of all, this modification reduced the additional i-cache delay to 0.931 ns, which allowed the cache to meet the timing constraints.

At this stage, we compared the results of resource utilization, the MTU miss rate and the misprediction rate for three types of virtual address XORING schemes. The VA was

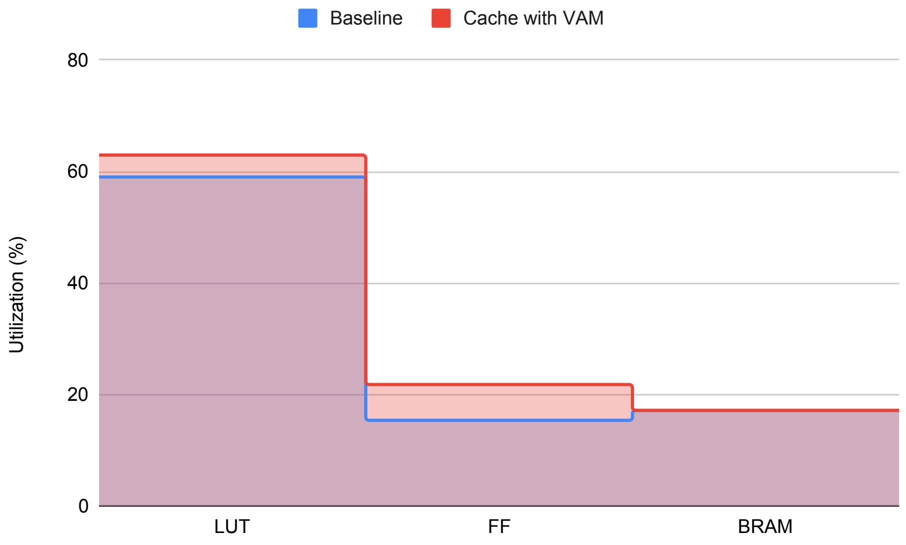


Figure 5.3: FPGA resources utilization for the cache with the VAM.

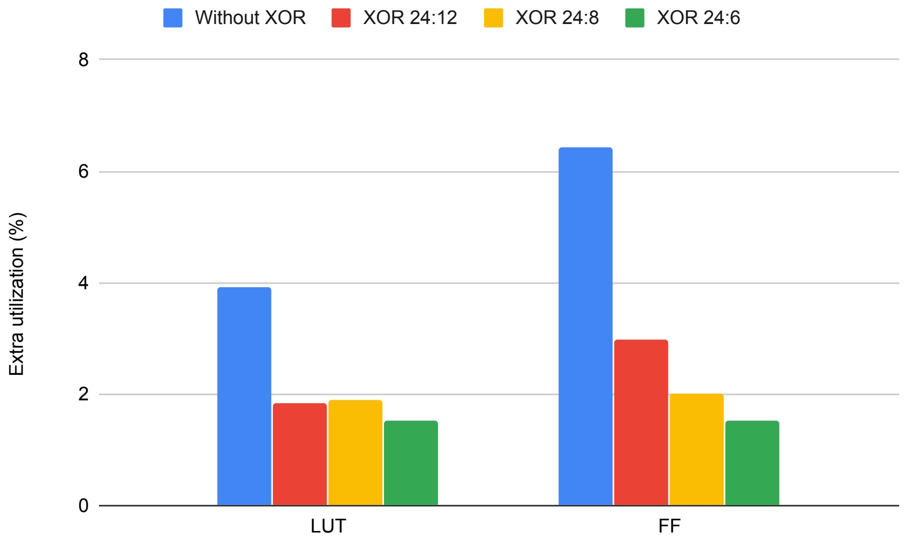


Figure 5.4: Extra resources utilization for the cache with different MTU XORING schemes.

divided into 2, 3 and 4 equal parts followed by bitwise XORING to implement 24:12, 24:8 and 24:6 mappings respectively. **Fig.5.4** shows that applying an XOR function to decrease the width of the MTU entries involved the less area overhead. Extra LUTs and FFs utilization reduced by more than half compared to the MTU implementation which contains all 24 bits of tag and index fields of the virtual address. Additionally, we can see that the varying the XOR function's output keeps the use of the LUTs almost at the same percentage while decreasing the use of the FFs proportionally.

However, despite the fact that the 24:12 XOR scheme lowered the area overhead noticeably, the MTU check still delayed the instruction fetch critically. Yet the next XOR function, 24:8, aided to meet timing constraints in Vivado.

The energy consumed by the cache with different XORING schemes was estimated based on the MTU hit and miss rates. The energy-saving efficiency of the implemented technique also depends on the XOR function. The fewer bits in the output, the more collisions in an XOR-based mapping scheme. **Fig.5.5** demonstrates that the prediction accuracy of the MTU declines sharply when we switching from a 24:8 to a 24:6 XORING scheme. It means that in more than 10% cases for the 24:6 scheme, the MTU chooses more than one way as predicted way, whereas only one prediction may be correct.

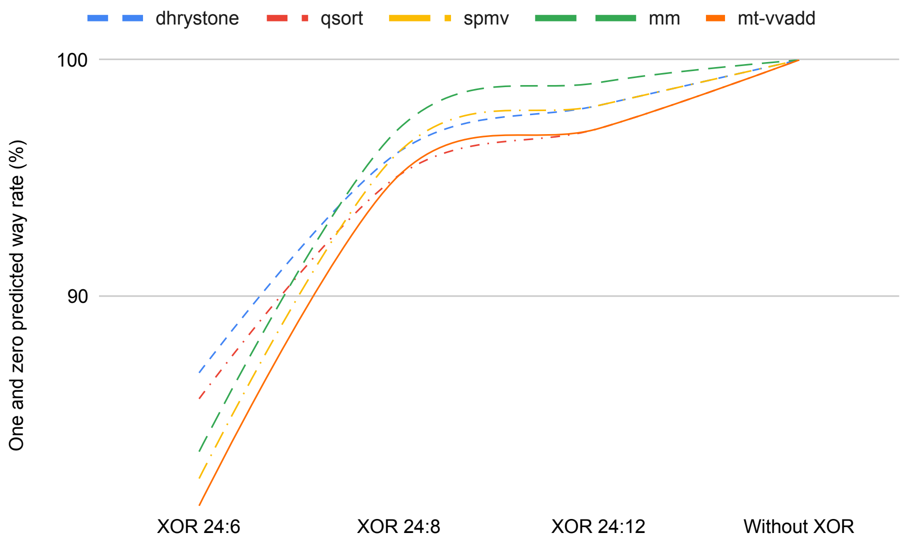


Figure 5.5: One and zero selected MTU ways depending on XORING scheme.

The MTU misprediction rate influences significantly the energy consumption, however, the overall performance of the cache is sparsely affected. For example, the scheme with the worst hit rate, 24:6, shows the performance degradation of 9.0% on average.

The percentages of the energy saved by three different XORING schemes are presented in **Fig.5.6**. The 24:12 scheme gives 17% energy reduction for dhrystone, which has the smallest number of lines of code and minimum branch instructions. For bigger benchmarks like spmv or mt-vvadd with lots of ramifications, this scheme could save only 13% of energy. The percentage of energy saved almost doubled when we applied the 24:8 scheme since it is spent less energy to read and compare 8 bits instead of 12 bits, while the hit rate and the one and zero predicted way rate are still at the high level (**Fig.5.5**). However, shortening the storing in the MTU bits suffers from more collisions in the XOR mapping leading to increasing the misprediction rate. For instance, in spmv, 18% of the MTU predictions gave from two to four matches for the four-way set-associative cache. Few of the predictions (0.4%) for the 24:6 scheme with one way match were also predicted wrong causing a false cache miss and subsequent low-level memory request. All this combined did not allow the latter to save more energy keeping this value at 31% on average. Our computations show that compared to the cache with the 24:8 scheme, the cache with the 24:6 scheme consumes more energy on writing suffering from the doubled miss rate. However, the latter almost completely compensates for this loss when reading. That is why the difference in energy saving between these two schemes is minimal for all tests.

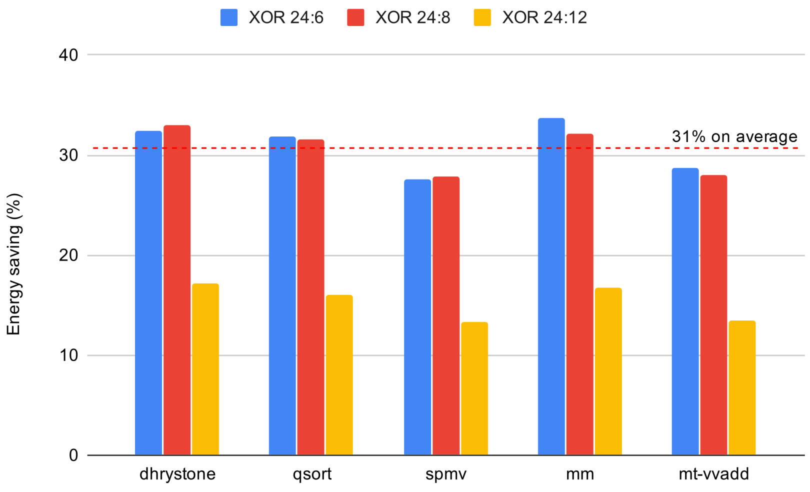


Figure 5.6: Energy saved by the MTU predictor.

5.4 Results for XOR-mapped MTU

At this stage of the VAM technique implementation, the idea was to use the output bits of the XOR-based hash function as an index to address the appropriate location in the MTU. That is the MTU for 24:8 mapping scheme consists of 2^8 entries of three bits each. The utilization results in **Fig.5.7** demonstrate that this modification in the VAM technique gives slightly better resource utilization. It compares the number of LUTs and FFs used by this implementation to the MTU implemented in the previous step, which is addressed by the index field of the VA (also 24:8). Along with reducing the number of bits in each MTU record, we also got rid of the need for comparison. We use the content of the MTU entry as a value of a predicted way. As a result, we consume 0,49% more LUTs, but 1,19% less flip-flops.

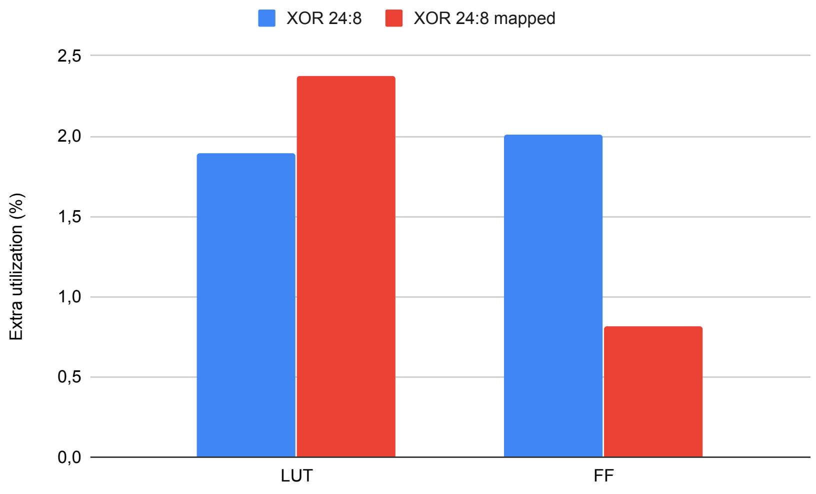


Figure 5.7: Extra resource utilization for the XOR-mapped MTU.

Even though we no longer need to check all the ways in the MTU, the delay time remains the same. **Fig.5.8** shows the delay time through the i-cache for different steps of the VAM implementation. We can see that the current implementation adds roughly 1 ns delay to the baseline solution, which is quite adequate to meet timing constraints.

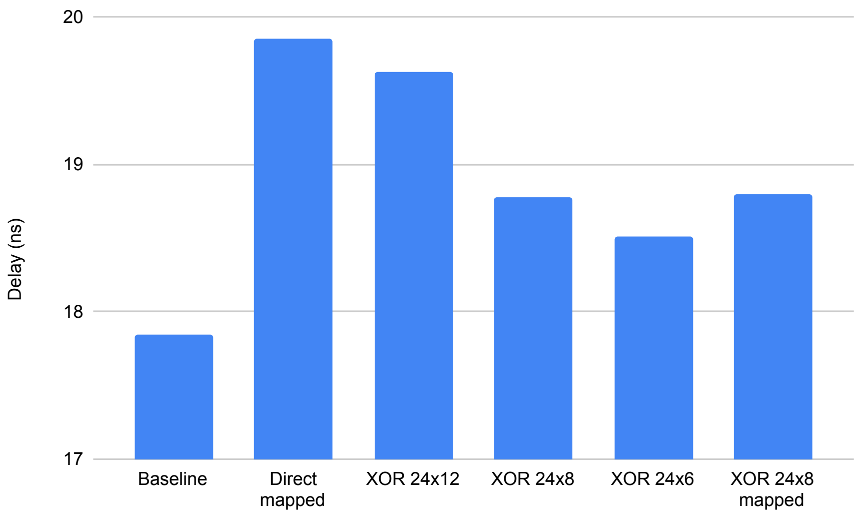


Figure 5.8: Delay time through the i-cache for different stages of implementation.

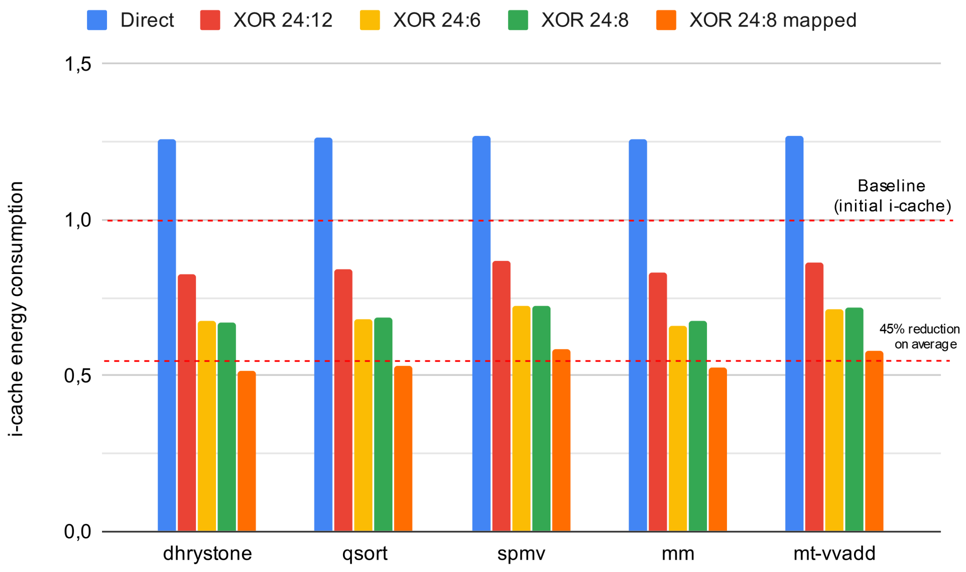


Figure 5.9: Normalized data of the energy consumption for different stages of implementation.

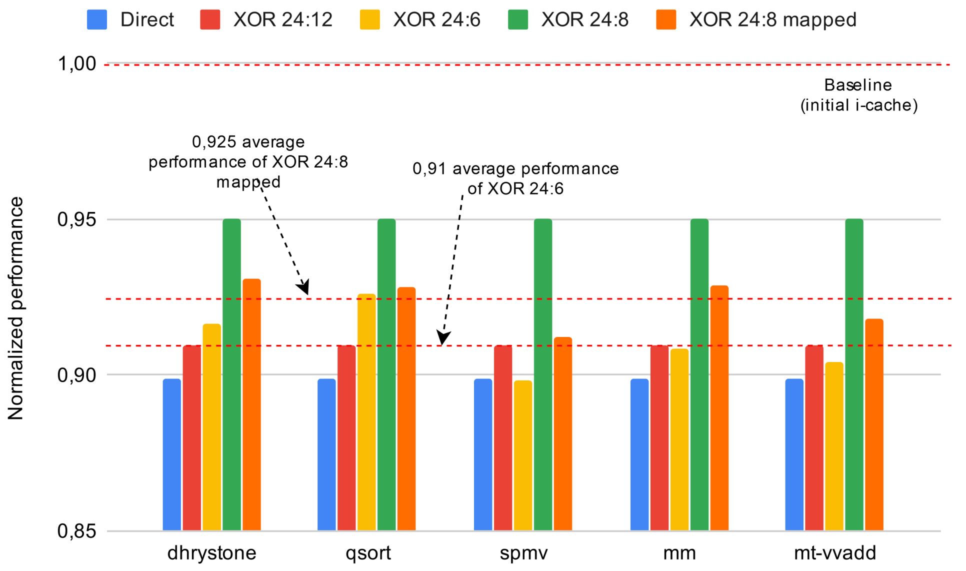


Figure 5.10: Normalized data of the performance for different stages of implementation.

However, the energy consumed by the MTU predictor has reduced since we need to read only one entry three bits width from that matching table. In **Fig.5.9** we have normalized the values of the energy consumed by the i-cache with different MTU types. The energy consumed by the direct-mapped MTU is above the baseline for all tests almost equally by 26%. This type of prediction did not have more than one MTU match in a request, that is why the growth is stable for all tests. The MTU with XOR-based mapping, contrary, has shown the best results of the energy-saving, 45% on average. The price for this energy efficiency is 7.5% of performance, however, this penalty is lower than that of the 24:6 XORING scheme, which is 9% on average (**Fig.5.10**). Though, there is a technique that reduces the performance only by 5% - 24:8 XORING scheme, but it consumes almost 15% more energy than the i-cache equipped by the MTU with 24:8 XOR-based mapping. Thus, the VAM technique at the last stage of implementation can be treated as a trade-off solution, which consumes the least amount of energy, shows less than 8% performance degradation, easy to implement, and requires insignificant additional area on the chip. **Table. 5.3** gives an overview of the results for different stages of the implementation in terms of four evaluation parameters (blue colour in bold - the best results, red colour - the worst results), where the baselines are the parameters for the conventional Rocket core i-cache implementation. We can see that the 24:8 XOR-based mapping (the rightmost column) shows the best results for three of the parameters and slightly loses in performance.

		Direct	XOR 24:12	XOR 24:6	XOR 24:8	XOR 24:8 m
Complexity		**Low**	**Low**	**Low**	**Low**	**Low**
Area+ (%)	LUT	3.93	1.83	**1.52**	1.89	2.38
	FF	6.42	2.97	1.53	2.01	**0.82**
	BRAM	**0**	**0**	**0**	**0**	**0**
Performance		0.90	0.91	0.91	**0.95**	0.925
Energy cons. (%)		126	85	69	69	**55**

Table 5.3: Comparison of different stages of implementation by four evaluation parameters.

The device layout for different stages of the VAM implementation with highlighted cache leaf cells is presented in appendix in **Fig.8.3**.

Chapter 6

Future work

In this chapter, some suggestions on the ways in which this work may be extended or improved are given.

First of all, to get more accurate results, more tests must be executed. To do so, additional RISC-V benchmarks have to be written, for example, tests like bitcount, dijkstra or stringsearch from MiBench benchmark suite (Guthaus et al., 2001) to cover more application categories. Additional tests will help in defining the algorithms for which this XOR mapping scheme doesn't work well and why. Then, this information can be used to research collision-free XOR-based mapping schemes, and apply one of the existing or create a new one that matches most of the applications. Finding a collision-free XOR function may also lead to reducing the number of entries of the MTU if the output of that function is less than eight, by this reducing the area used.

Another suggestion is to implement such a technique for the data-cache of the Rocket core. A prediction mechanism that works on a similar principle, way record buffering, will be mentioned in Chapter 7 and is proposed for a data-cache. The proposed technique gives less than 20% energy reduction, but it seems like applying some VAM tricks (for example, XOR-based mapping or storing the most recently reside way) could potentially save more energy. And then, try to apply these techniques to the BOOM core caches.

Different circumstances may cause the operating system to remap virtual page numbers to physical page numbers. This action leads to invalidating the TLB entry or even the whole TLB. In this situation, the content of the MTU also becomes irrelevant and may lead to unnecessary lower-level memory accesses. To avoid this, it makes sense to invalidate the MTU on the TLB invalidation.

The aliasing problem is an issue that is relevant for multiprocessing tasks. Different processes may map the identical virtual address to the different physical addresses. The process identity (PID) + tag access is a perfect approach to manage this problem.

Chapter 7

Related Work

Different techniques aimed at improving cache energy consumption exist. Some of them, which utilize the principle of avoiding reading unneeded data, are discussed in this chapter.

7.1 Phased cache

Phased cache proposed in (Megalingam et al., 2009) where the cache access process is divided into two parts. In the first stage, all the tags are probed in parallel. In the second stage, if there is a hit, then the data is only accessed for the hit way. Several proposals were made to increase the efficiency of such type of cache.

Another researchers in (Min et al., 2004) has showed that only a small part of the tag can be compared in the first phase, while the remaining bits of the tag are compared during the second phase in order to verify whether the result is valid or not. However, this modification does not address the performance loss.

The selected way may be predicted by exploiting one of the replacement policies' logic, namely Most Recently Used (MRU) bit or Least Recently Used (LRU) replacement strategy, since studies show that the memory accesses are intensively focused on the MRU region in the cache (So and Rechtschaffen, 1988). The main idea is that the replacement status of lines under an LRU policy can be represented by a stack. When a line is used by the processor, it becomes the MRU line and is placed at the top of the stack, respectively the LRU line is at the bottom of the stack. This information can be used to make a prediction.

Such a technique was implemented by Inoue et al. in (Inoue et al., 1999). Authors compare phased and way-predicting four-way set-associative caches with conventional one which accesses tag and data arrays all the ways in parallel outputting the data in one clock cycle. Their experimental analysis using a cache simulator shows that both phased and way-predicting caches reduce the average energy consumption by about 70%, but phased cache increases the average cache access time by about 100%. However, the implementation of this technique adds some complexity, since the current Rocket core i-cache implementation uses merely random replacement policy.

7.2 AMD way-predictor

Since the AMD Family 17h microarchitecture, AMD uses a way-predictor in the L1 data cache (AMD, 2017). Every cache line in the L1 data cache is tagged with a virtual-address-based $utag$ that was used to access the cache line initially. The $utag$ is used to determine which way of the cache to read using the virtual address given by the CPU, which is available before the load's physical address has been determined via the TLB. The $utag$ is a hash of the virtual address. This lookup enables a very accurate prediction of in which way the cache line is located prior to a read of the cache. This allows the cache to read just a single cache way, which saves power.

In this implementation of the way-prediction, the number of $utags$ directly depends on the size of the cache. Furthermore, it is possible for the $utag$ to be wrong in both directions: it can predict hit when the access will miss, and it can predict miss when the access can have hit. In either case, a fill request to the L2 cache is initiated and the $utag$ is updated when L2 responds to the fill request. Our implementation handles the second case by using PopCount function to enable all the ways to read in tag and data arrays in case of an MTU miss.

7.3 Sequential address way-predictor (SAWP)

The researchers in (Powell et al., 2001) propose to implement way-prediction by extending branch prediction concepts. The key thought here is that the fetch hardware performs branch prediction to determine the next PC while accessing the i-cache with the current PC. They can use the PC of the previous access to predict the way since i-cache accesses occur at the beginning of the pipeline. By the time the previous i-cache access is complete, the next predicted PC and the predicted way are ready, which does not add any delay to the i-cache access. The data from the BTB is used to provide a way-prediction for a taken branch. Not taken branches and non-branches reside in an extra table called Sequential Address Way-Predictor (SAWP), which is indexed by the current PC. The RAS also has to be modified in order to provide not only the return address but also the return address's way.

A correctly way-predicted fetch accesses the tag array and the predicted data way. On the other hand, way mispredicted fetches probe the matching data way a second time, suffering from extra energy and access time. This technique adds *log(nWays)* bits to each entry of the BTB, the SAWP, and the RAS. Powell et al. (2001) estimated that the proposed technique achieves an L1 i-cache energy-delay reduction of 64% with less than 3% performance degradation. In this case, the BTB and the RAS have to be extended by appropriate inner logic to output the needed signal. Additionally, the SAWP table (1024 entries in inventors experimental setup) should be added, which adds to area cost. Besides that, predicted way value has to be conveyed from the BTB to the cache.

7.4 Way record buffering (WRB)

The technique is described in (Wang and Wang, 2016). In this architecture, the tag needs to be saved into a separate register array called Tag Record Buffer (TRB). To predict the access way number, a Way Record Buffer (WRB) is also added. The WRB records the way number to which the corresponding tag in the TRB was written in the tag array. For a four-way set-associative cache, the width of the WRB will be respectively four bits. These registers have to be accessed before probing the tag and data arrays to define in which way or ways the required data may reside.

Increasing power consumption and area overhead, caused by the TRB and the WRB, are also considered by the authors. They propose to utilize a replacement scheme in order to decrease the number of entries for those two tables, which in their case, led to reducing this number up to three.

This technique implies less area overhead and energy consumption, while insignificant performance degradation, but less than 20% power reduction.

7.5 Way halting cache (WHC)

The idea is to halt one or more ways in a cache. This idea is not a novelty. In order to complete exactly this task, the WHC technique was proposed (Zhang et al., 2005). It performs a fully associative halt-tag check in parallel with the decoding of the L1 cache index. Nevertheless, the implementation of this technique may be impractical. Aiming to reduce word line length, memories are often banked, while the tags and data are stored apart. Halt-tag memory either needs to route its signals to all the different banks and memories or be replicated for each bank since the halt-tag cache is fully associative. Subsequently, this would either add delays or a higher energy and area overhead. The need to customize an SRAM implementation is another impracticality of this technique, which is not easily available and would be costly.

7.6 Speculative tag access (STA)

Bardizbanyan et al. (2013) proposed the STA technique: cache tag arrays are accessed during the address generation stage followed by a single data array access in the SRAM access stage if the speculation succeeds and there is a hit in the cache. This technique reduces the number of accessed data arrays but it will always access all tag arrays in parallel. The TLB is accessed in both SRAM access stage and address generation stage, which adds complexity to a conventional cache design since the input signal to the TLB has to be routed from various locations on the chip. Furthermore, the forwarding logic is explored to produce the input from the address generation stage, which can lead to additional delay.

7.7 Speculative halt-tag access (SHA)

The SHA technique (Moreau et al., 2016) is proposed to combine the advantage of the two previously discussed techniques, the WHC and the STA. The authors propose to use way halt tag array and access them earlier than the SRAM access stage - in the address generation stage. Thus, this speculative halt-tag approach defines which L1 tag and data arrays to access by the beginning of the SRAM access stage. In case the speculation fail, the cache is accessed conventionally the next clock-cycle. Since the halt-tag array is accessed before the TLB is accessed, the TLB is only accessed during the SRAM access stage. This technique has low complexity and performance degradation but mainly focuses on data caches.

7.8 Filtering

Many of reducing cache energy dissipation proposals are placing small energy-efficient buffers in front of the cache to filter incoming traffic. For example, the main idea of tag overflow buffering (Loghi et al., 2009) is to move a large number of tag bits from the cache into an external register, called a tag overflow buffer for identifying a current memory locality. This buffer is a kind of one entry L0 cache that detects the locality of application programs. Another example is a filter cache (Kin et al., 1997; Bardizbanyan et al., 2014), which is a also small and fast L0 cache. Because of its small size, the filter cache has a high miss rate, and using it leads to increasing program execution time due to increased load latency in the cache in case of a filter cache miss. However, the overall energy consumption is still decreased.

Chapter 8

Conclusion

These days, when data processing time becomes a critical factor, processor performance is a key aspect in the success of most applications. A few decades ago almost all the research and manufacture efforts were respectively directed to speed and to capacity due to division into microprocessor and memory fields. As a result, the gap between the processors and memory speeds is continuously growing. Trying to address this inequality, caches were proposed. That are small size memories of high speed and high cost, that accelerate other memories of high speed, high dimension, and reduced cost. In such a memory hierarchy, the L1 cache is a memory bank built into the CPU chip, and it is the fastest memory in the computer and closest to the processor.

Thus, an on-chip cache is one of the major components in contemporary high performance processors. However, it also becomes the main power consumer in a processor due to large area and high access frequency. Therefore, there have been increasing interests in designing low power on-chip caches especially for embedded systems, Internet-of-Things, mobile devices. Although there have been a number of techniques proposed to address this problem, all of them have a certain extent of improvements, overheads, and trade-off, and some of them are able to reduce the energy consumption in some cases up to 80%. One such technique is way-prediction, which attempts to avoid probing all the ways in a set and wasting energy for nothing while only one way contains the requested data.

In this thesis, we implemented the virtual-address-matching mechanism and applied it to reduce L1 instruction cache dynamic energy while maintaining high performance. We used this mechanism to predict the matching way number and provide the prediction prior to the cache access. The way-prediction technique reduces energy consumption because only the predicted way is accessed.

Several versions of this technique have been implemented and evaluated on the RISC-V ISA Rocket core. The effectiveness of the way-prediction mechanism in reducing L1 i-cache energy was evaluated in different stages and different configurations. The influence of the implementation on the instruction fetch performance and the i-cache occupied area was also considered. Relative to parallel access L1 i-cache, the implemented technique achieves the energy reduction of 45% with less than 8% of performance degradation.

Bibliography

Al-Zoubi, H., Milenkovic, A., Milenkovic, M., 2004. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite, ACM-SE 42 Proceedings of the 42nd annual Southeast regional conference, Alabama, US. pp. 267–272. doi:https://doi.org/10.1145/986537.986601.

AMD, 2017. Software optimization guide for amd family 17h processors. https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf. [Online; accessed 20-April-2020].

Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., 2016. The Rocket Chip Generator. No. UCB/EECS-2016-17. Technical Report. UCB. California, US.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanovic, A., 2012. Chisel: Constructing hardware in a scala embedded language, DAC Design Automation Conference 2012, California, US. pp. 1212–1221. doi:https://doi.org/10.1145/2228360.2228584.

Bardizbanyan, A., Själander, M., Whalley, D., Larsson-Edefors, P., 2013. Speculative tag access for reduced energydissipation in set-associative l1 data caches, 2013 IEEE 31st International Conference on Computer Design (ICCD). doi:https://doi.org/10.1109/ICCD.2013.6657057.

Bardizbanyan, A., Själander, M., Whalley, D., Larsson-Edefors, P., 2014. Designing a practical data filter cache to improve both energy efficiency and performance. ACM Transactions on Architecture and Code Optimization (TACO) 10, 1–25. doi:https://doi.org/10.1145/2541228.2555310.

Dinis, N., 2002. Cache: Why level it, 2002 3rd Internal Conference on Computer Architecture, Braga, Portugal. pp. 19–26.

Gonzalez, A., Valero, M., Topham, N., Parcerisa, J., 1997. Eliminating cache conflict misses through xor-based placement functions, 11th international conference on Supercomputing, Vienna, Austria. pp. 76–83. doi:https://doi.org/10.1145/263580.263599.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R., 2001. Mibench: A free, commercially representative embedded benchmark suite, Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4, TX, US. doi:https://doi.org/10.1109/WWC.2001.990739.

Hennessy, J., Patterson, D., 2007. Computer Architecture: A Quantitative Approach. 4 ed., Morgan Kaufmann, San Francisco, US.

Hennessy, J., Patterson, D., 2014. Computer Organization and Design. The hardware/software interface. 5 ed., Morgan Kaufmann, San Francisco, US.

Inoue, K., Ishihara, T., Murakami, K., 1999. Way-predicting set-associative cache for high performance and low energy consumption, Proceedings. 1999 International Symposium on Low Power Electronics and Design, California, US. pp. 273–275. doi:https://doi.org/10.1145/313817.313948.

Intensivate, 2018. Introduction to rocket chip code style. https://github.com/Intensivate/learning-journey/wiki/Introduction-to-Rocket-Chip-code-style/. [Online; accessed 20-April-2020].

Kenneth, J., 1997. Content-addressable memory core cells a survey. Integration, the VLSI Journal 23. doi:https://doi.org/10.1016/S0167-9260(97)00021-7.

Kin, J., Gupta, M., Mangione-Smith, W., 1997. The filter cache: an energy efficient memory structure, MICRO 30 Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, NC, US. pp. 184–193. doi:https://doi.org/10.1109/MICRO.1997.645809.

Kwak, J., Jeon, Y., 2010. Compressed tag architecture for low-power embedded cache systems. Journal of Systems Architecture 56, 419–428. doi:https://doi.org/10.1016/j.sysarc.2010.04.010.

Lee, Y., Watereman, A., Cook, H., Zimmer, B., Keller, B., Puggelli, A., Kwak, J., Jevtic, R., Bailey, S., Blagojevic, M., Chiu, P., Avizienis, R., Richards, B., Bachrach, J., Patterson, D., Alon, E., Nikolic, B., Asanovic, K., 2016. An agile approach to building risc-v microprocessors. IEEE Micro 36, 8–20. doi:https://doi.org/10.1109/MM.2016.11.

Lennon, P., Gahan, R., 2018. A comparative study of chisel for fpga design, 2018 29th Irish Signals and Systems Conference (ISSC), Dublin, Ireland. doi:https://doi.org/10.1109/ISSC.2018.8585292.

Lipp, M., Hadžić, V., Schwarz, M., Perais, A., Maurice, C., Gruss, D., 2020. Take a way: Exploring the security implications of amd's cache way predictors, 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), Taipei, Taiwan. doi:https://doi.org/10.1145/3320269.3384746.

Loghi, M., Azzoni, P., Poncino, M., 2009. Tag overflow buffering: Reducingtotal memory energy by reduced-tag matching. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 17, 728–732. doi:https://doi.org/10.1109/TVLSI.2009.2016720.

Megalingam, R., Deepu, K., Iype, P., Vikram, V., 2009. Phased set associative cache design for reduced power consumption, 2009 2nd IEEE International Conference on Computer Science and Information Technology, Los Alamitos, US. pp. 551–556. doi:https://doi.org/10.1109/ICCSIT.2009.5234663.

Min, R., Jone, W., Hu, Y., 2004. Phased tag cache: an efficient low power cache system, 2004 IEEE International Symposium on Circuits and Systems (ISCAS), Vancouver, Canada. pp. 805–808. doi:https://doi.org/10.1109/ISCAS.2004.1329394.

Moreau, D., Bardizbanyan, A., Själander, M., Whalley, D., Larsson-Edefors, P., 2016. Practical way halting by speculatively accessing halt tags, 2016 Design, Automation and Test in Europe Conference and Exhibition (DATE), Dresden, Germany. doi:https://doi.org/10.3850/9783981537079_0663.

Powell, M., Agarwal, A., Vijaykumar, T., Falsafi, B., Roy, K., 2001. Reducing set-associative cache energy via way-prediction and selective direct-mapping, MICRO 34 Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, TX, US. pp. 54–65. doi:https://doi.org/10.1109/MICRO.2001.991105.

Smith, A., 1982. Cache memories. ACM Computing Surveys (CSUR) 14, 473–530. doi:https://doi.org/10.1145/356887.356892.

So, K., Rechtschaffen, R., 1988. Cache operations by mru change. IEEE Transactions on Computers 37, 700–709. doi:https://doi.org/10.1109/12.2208.

UCB, 2019a. Berkeley architecture research. https://bar.eecs.berkeley.edu/projects/rocket_chip.html/. [Online; accessed 20-April-2020].

UCB, 2019b. Risc-v. https://riscv.org/members-at-a-glance/. [Online; accessed 20-April-2020].

Vandierendonck, H., DeBosschere, K., 2005. Xor-based hash functions. IEEE Transactions on Computers 54, 800–812. doi:https://doi.org/10.1109/TC.2005.122.

Wang, L., Wang, D., 2016. Way prediction set-associative data cache for low power digital signal processors, 2016 IEEE 13th International Conference on Signal Processing (ICSP), Chengdu, China. pp. 508–512. doi:https://doi.org/10.1109/ICSP.2016.7877886.

Waterman, A., Asanovic, K., 2019. The risc-v instruction set manual. https://riscv.org/specifications/. [Online; accessed 20-April-2020].

Willert, C., 1999. The evolution of programmable logic design technology. Xcell, Issue 32, Second Quarter .

Wulf, W., McKee, S., 1994. Hitting the Memory Wall: Implications of the Obvious. Technical Report. University of Virginia. Virginia, US.

Xilinx, 2014. A generation ahead for smarter systems: 9 reasons why the xilinx zynq-7000 all programmable soc platform is the smartest solution. `https://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf`. [Online; accessed 20-April-2020].

Xilinx, 2018a. Zynq-7000 soc data sheet: Overview. `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`. [Online; accessed 20-April-2020].

Xilinx, 2018b. Zynq-7000 soc technical reference manual, v1.12.2. `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. [Online; accessed 20-April-2020].

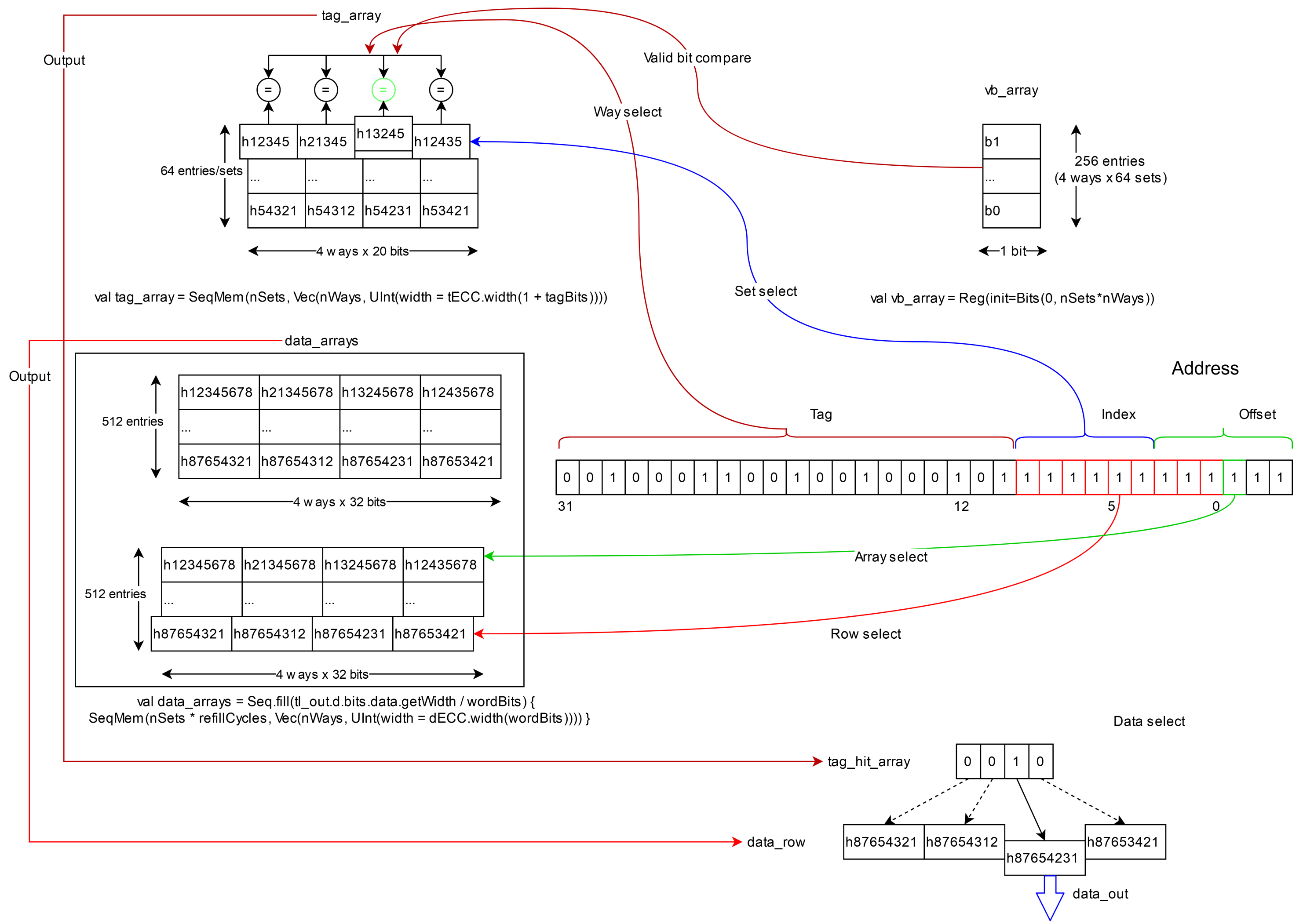
Xilinx, 2019. Vivado design suite userguide. design analysis and closure techniques, v2019.2. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug906-vivado-design-analysis.pdf`. [Online; accessed 20-April-2020].

Yang, Q., Li, H., 2010. A new virtual-address-mapping mechanism for low-energy i-cache, 2010 International Conference on Computational Intelligence and Software Engineering, Wuhan, China. pp. 1–4. doi:`https://doi.org/10.1109/CISE.2010.5677176`.

Zang, W., Gordon-Ross, A., 2013. A survey on cache tuning from a power/energy perspective. ACM Computing Surveys (CSUR) 45, 32:1–32:49. doi:`https://doi.org/10.1145/2480741.2480742`.

Zhang, C., Vahid, F., Yang, J., Najjar, W., 2005. A way-halting cache for low-energy high-performance systems, ACM Transactions on Architecture and Code Optimization, NY, US. pp. 34–54. doi:`https://doi.org/10.1109/LPE.2004.240851`.

Appendix



```
val tag_array = SeqMem(nSets, Vec(nWays, UInt(width = tECC.width(1 + tagBits))))
val vb_array = Reg(init=Bits(0, nSets*nWays))
val data_arrays = Seq.fill(tl_out.d.bits.data.getWidth / wordBits) {
SeqMem(nSets * refillCycles, Vec(nWays, UInt(width = dECC.width(wordBits)))) }
```

Figure 8.1: The Rocket core i-cache reading.

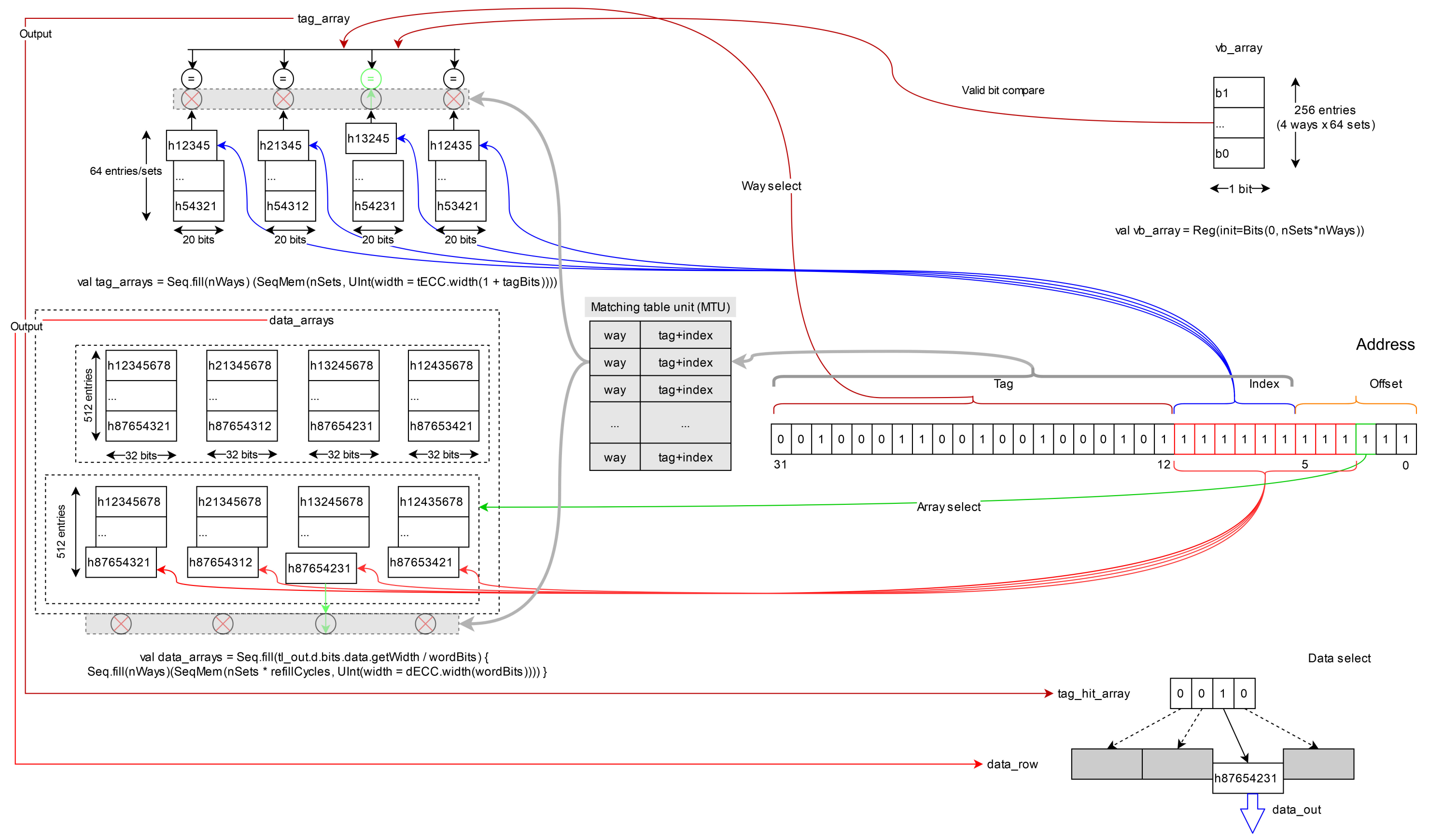
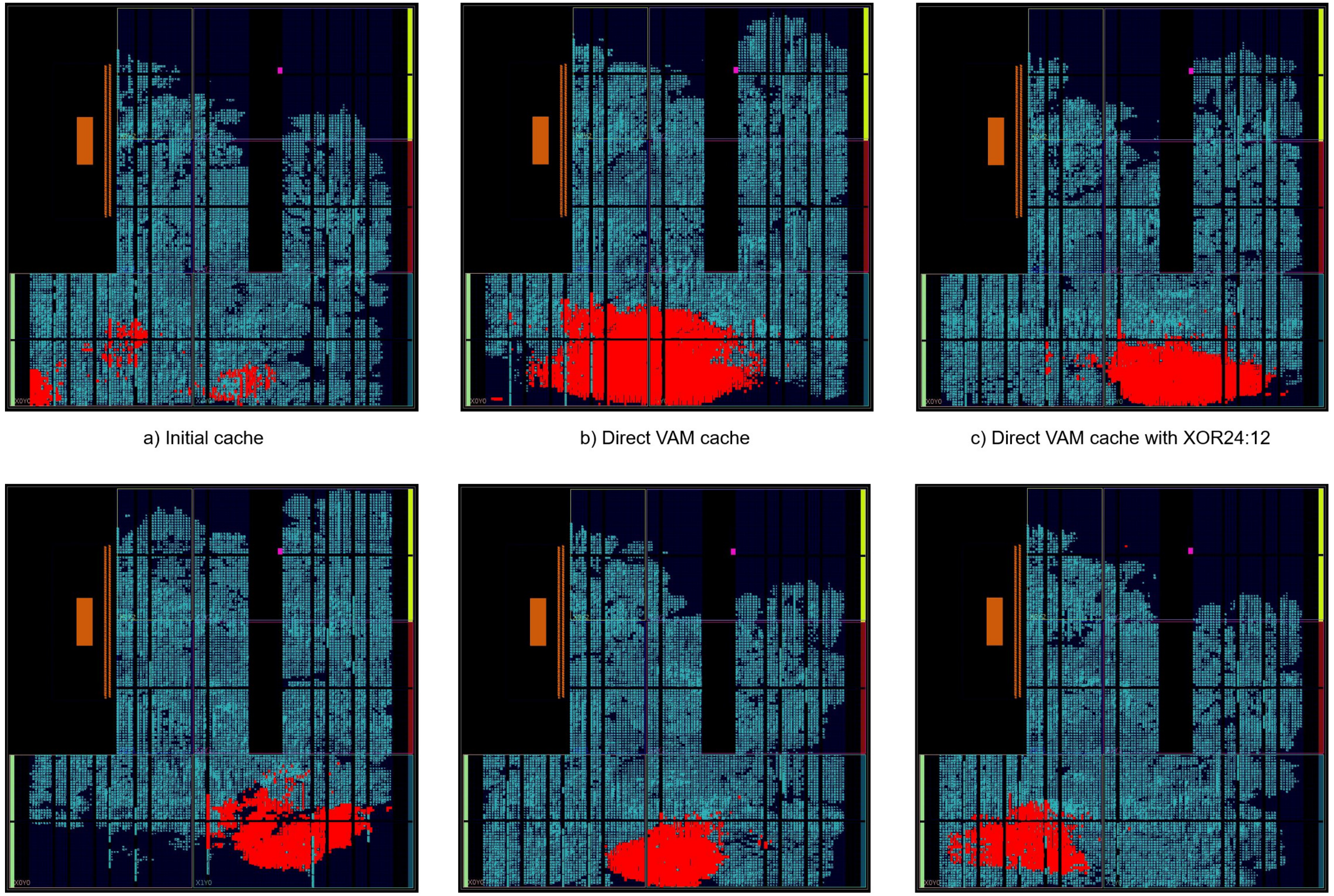


Figure 8.2: The Rocket core i-cache modified reading.



a) Initial cache

b) Direct VAM cache

c) Direct VAM cache with XOR24:12

d) Direct VAM cache with XOR24:8

e) Direct VAM cache with XOR24:6

f) XOR-based VAM cache with XOR24:8

Figure 8.3: Device layout with i-cache leaf cells highlighting for different stages of the VAM implementation.