Low Power Reduced-Tag Architecture for Set-Associative Caches for ARM-Core

Dissertation submitted in the partial fulfillment of requirement for the award of degree of

Master of Technology

In

VLSI Design

Submitted by:

INDERJIT SINGH

Roll No: 601161013

Under the guidance of:

Dr. SANJAY SHARMA

Professor

ECED, THAPAR UNIVERSITY

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

(Established under the section 3 of UGC Act, 1956)

PATIALA – 147004 (PUNJAB)

DECLARATION

I hereby declare that the work which is being presented in the dissertation entitled, **"Low Power Reduced-TAG Architecture for Set-Associative Caches in ARM-Core"** in partial fulfillment of the requirement for the award of degree of Master of Technology in VLSI Design & CAD submitted in Electronics and Communication Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Dr. Sanjay Sharma, Professor, ECED and refers other researcher's work which are duly listed in the reference section.

The matter presented in this dissertation has not been submitted in any other University/Institute for the award of degree.

Date: 15/7/13

(INDERJIT SINGH)
Roll No: 601161013

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

12.7.2013

(Dr. Sanjay Sharma)
Professor
ECED, Thapar University

Countersigned by:

Head
ECED, Thapar University
Patiala-147004

Dean of Academic Affairs
Thapar University
Patiala- 147004

ACKNOWLEDGEMENT

First of all, I would like to express my gratitude to **Dr. SANJAY SHARMA**, **Professor**, Electronics & Communication Engineering Department, Thapar University, Patiala for his guidance and support throughout this thesis work. I am really very fortunate to have the opportunity to work with him. I found his guidance to be extremely valuable.

I am thankful to the **Head of Department, Professor (Dr.) RAJESH KHANNA** and **PG Coordinator, Dr. KULBIR SINGH (Associate Professor)** of Electronics & Communication Engineering Department for their encouragement and inspiration for the execution of this thesis work.

I am also thankful to the entire faculty and staff of Electronics & Communication Engineering Department for the help and moral support which went along the way for the successful completion of this thesis work.

My greatest thanks are to all who wished me success especially my parents. Above all I render my gratitude to the Almighty who bestowed self-confidence, ability and strength in me to complete this work for not letting me down at the time of crisis and showing me the silver lining in the dark clouds. I do not find enough words with which I can express my feelings of thanks to my dear friends for their help, inspiration and moral support which went a long way in successful competition of the present study.

Inderjit Singh

Roll no. 601161013

ABSTRACT

Most of the embedded processors utilize cache memory in order to minimize the performance gap between memory systems and processor. In embedded systems caches are normally implemented along with processors in one IC. The power consumed by the cache system constitutes the major fraction of the power dissipated by the embedded processors.

With increasing computational demands on embedded processors, set-associative caches are being used. In larger caches the major portion of power consumption occurs in address decoding including tag comparisons. Set-associative caches consume larger energy as compared to the direct mapped caches as i) set-associative caches have greater tag bits, ii) they have parallel organization of tag arrays, and hence parallel tag comparison dissipates more energy. It is further analyzed that not all the tag bits are necessary for a cache configuration to achieve a normal performance in terms of hit rate. Hence, architecture with reduced but optimum number of tag bits is possible, which would consume lesser energy.

Novel reduced tag architecture for set-associative caches is proposed, which uses lesser number of tag bits in the tag array to minimize power consumption; with minimum hardware modifications. The proposed architecture is inspired from compressed tag architecture for Direct-Mapped caches, proposed by Kwak and Jeon. New modified Way selection methods called *MASKED FIFO Way Selection* is presented for the proposed architecture that makes the design at par with the conventional set-associative design in terms of performance.

An average Tag Reduction of 40% is achieved on different cache configurations. Thereby an energy savings of 10 – 63% for different cache configurations have been achieved. Nonetheless an embedded system architect can chose specific values of optimum tag-length for the specific application programs for specific cache size/associativity.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER

INTRODUCTION

1.1 Overview

Ideally one would desire an indefinitely large memory capacity such that any particular . . . word would be immediately available. . . .

We are . . . forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine, and J. von Neumann

The increasing demands of higher computational capabilities of computer systems have led to immense technological improvements in the last half century. This rapid rate of improvement has come both from advances in the technology used to build computers (IC technology) and from innovation in computer design.

In 1980 microprocessors were often designed without caches. Microprocessor performance improved 55% per year since 1987, and 35% per year until 1986 [7]. The main focus had been to improve the computational performance of the processor in terms of number of instructions executed per second.

This put a great demand on performance of memory, to take it at par with the increasing performance of CPU; hence there is a processor-memory performance gap that computer architects must try to close. An economical solution to that is a *memory hierarchy*, which takes advantage of locality and cost/performance of memory technologies [7]. Since fast memory is expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next lower level. The goal is to provide a memory system with cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level [7].

Cache is the name given to the first level of the memory hierarchy encountered once the address leaves the CPU. Since the principle of locality applies at many levels, and taking advantage of locality to improve performance is popular, the term cache is now applied whenever buffering is employed to reuse commonly occurring items. Examples include file caches, name caches, and so on.

1.2 MEMORY HEIRARCHY

The speed at which a processor can execute instructions and interpret data is far higher than the time taken to access a memory location. Thus, the presence of slower memory devices can indeed be a bottleneck to the performance of the processor and the system as a whole.

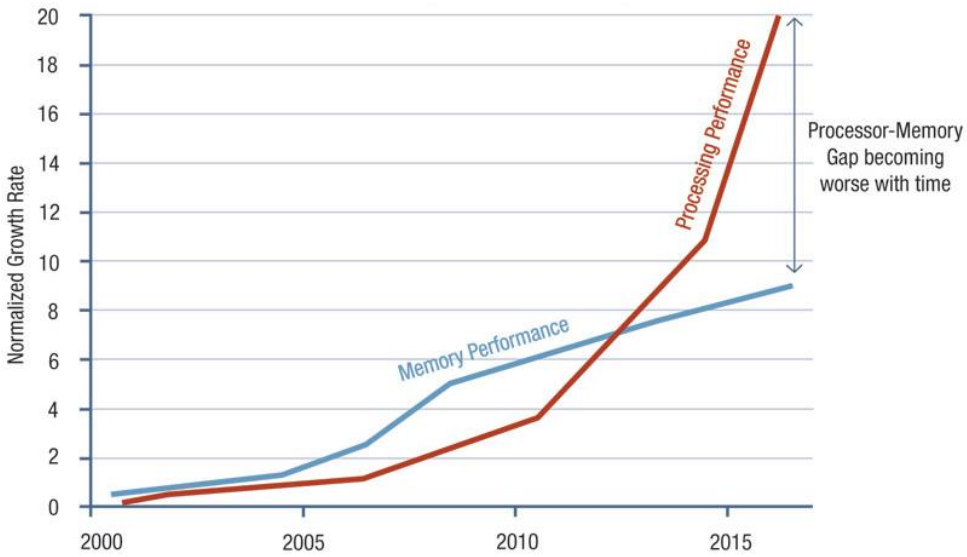


Figure 1.1 Gap in the performance of CPU and Memory [7]

A well designed computer system should perform as if all the memory were fast. The goal is to provide a memory system with cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level.

An economical solution to this is a memory hierarchy, which takes advantage of locality and cost/performance of memory technologies. The principle of locality (to be presented shortly), says that most programs do not access all code or data uniformly. This principle, plus the guideline that smaller hardware is faster, led to hierarchies based on memories of different speeds and sizes.

1.2.1 Memory Hierarchy Structure

A typical Computer Organisation consists of several different types of memory. As we now understand, different types of memory differ in terms of access speed and cost. Typically, high speed memory devices cost more and thus can be used only sparsely, to provide specialized functionality. On the other hand, cheaper and slower memory devices are used for generic purposes, in larger quantities. This situation is expressed as the memory hierarchy.

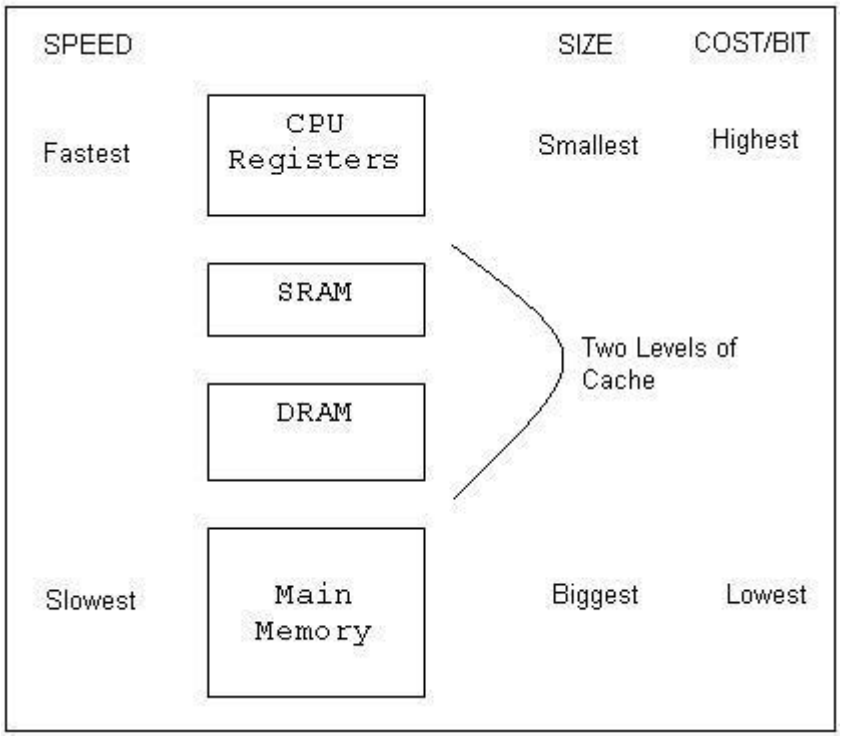


Figure 1.2 Memory Hierarchy

1.3 CACHE MEMORY

The speed at which a processor can execute instructions and interpret data is far higher than the time taken to access a memory location. Thus, the presence of slower memory devices can indeed be a bottleneck to the performance of the processor and the system as a whole. A well designed computer system should perform as if all the memory were fast. To create such an illusion, specialized pieces of *fast memory*, called caches are used. A cache sits in between a processor and typically a primary memory device and stores frequently accessed data and instructions.

With good prediction schemes and a large cache, the performance of the system can be increased enormously. As shown in the coming figure are different abstraction layers, most advanced microprocessors today have several *physical* layers of memory, making up the cache memory, embedded on the chip. The closest one is called layer 1 (L1) cache and

usually has direct contact with the Central Processing Units (CPU) pipeline. This gives an extremely short access time, and therefore provides the highest performance. This cache is usually run at the same clock frequency as the CPU. The strict requirements of this L1 cache and the fact that it has to access the CPU pipeline means that it is very expensive in terms of area. The relative large area per cell of this high performance memory makes it very difficult to place large L1 caches close to the CPU. As an example the Intel® Pentium® 4 processors have only 8KB of L1 cache. To make up for the relatively small L1 cache, a larger level 2 (L2) cache is often put on-chip. This cache is placed slightly further away from the CPU, and is connected to it through an internal bus. This results in a larger latency. It is also often run at a lower frequency making it possible with smaller, less performance optimized, cells. As a comparison most Intel® Pentium® 4 processors have 512KB of L2 cache. Sometimes a third cache on chip is used. It is referred to as level 3 (L3) cache and is, following the same convention as above, the furthest from the CPU. It is in most cases quite comparable in performance to L2 cache or L1 cache.

1.4 CACHE: WORKING PRINCIPLE

The theory that explains this performance improvement is called "Locality of Reference." The concept is that at any given time the processor will be accessing memory in a small or localized region of memory. The cache loads this region allowing the processor to access the memory region faster. (How well does this work? In a typical application, the internal 16K-byte cache of a Pentium® processor contains over 90% of the addresses requested by the processor. This means that over 90% of the memory accesses occur out of the high speed cache [7].

Caches work on the concept of *Memory Locality*, which states that future memory accesses are *near* past accesses. This is facilitated by the structure of programs, where large portions of instructions are just loops acting on the same set of data again and again.

Two types of locality can be leveraged by a cache:

1. **Temporal Locality or Nearness in Time**
   This translates to the axiom that the same data will be accessed again very soon. As for example, the loop counter will need to be accessed at the end of every iteration of the loop.

2. **Spatial Locality or Nearness in Space**

   This translates to the axiom that the next access will often be very close (physically near memory location) to the last access. As for example, if the instructions of a small sub-routine are to be accessed, thus fetching not only first but certain subsequent memory contents (instructions) to the cache, predicting they will be required in near future will make subsequent accesses faster by virtue of locality in space.

1.5 CACHE ARCHITECTURE

Cache memory unit looks something like in the figure 1.3. It receives an address from the CPU as input, and provides data with HIT/MISS signal as output. Cache memory system is composed of decoders, wordline and bitline signals, multiplexers, comparators, encoders and some multi-input gates.

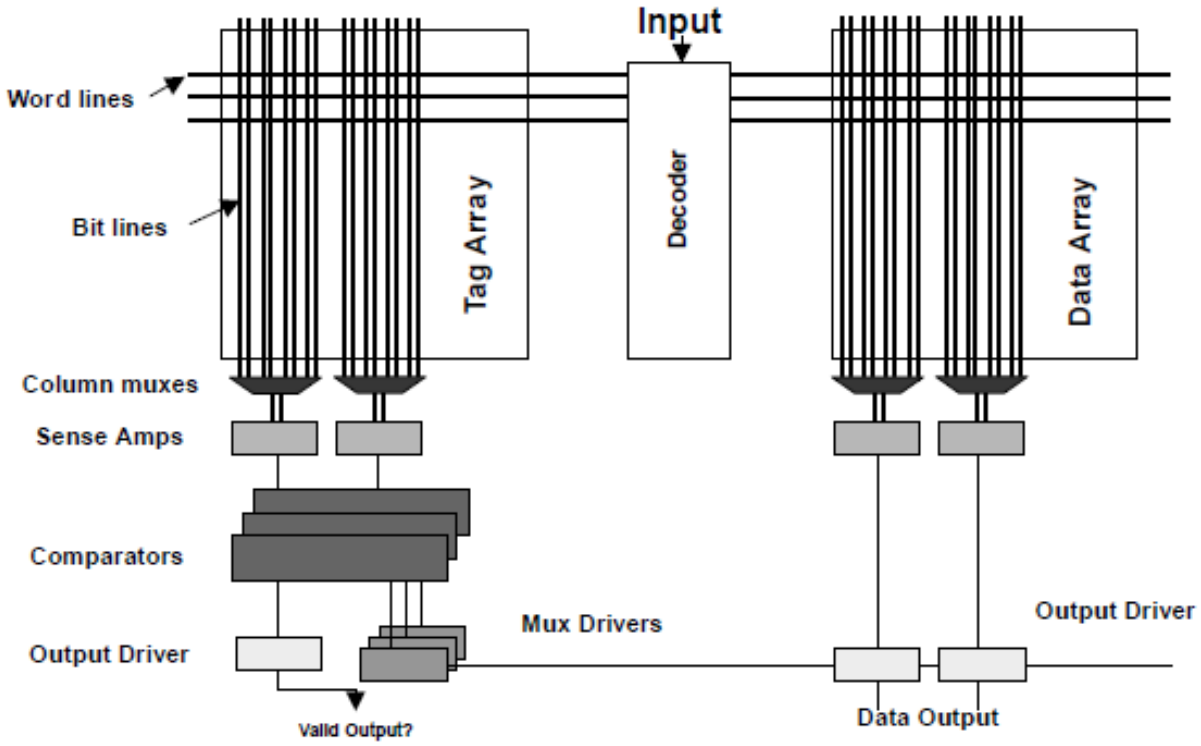


Figure 1.3 Cache Structure [4]

The decoder first decodes the address and selects the appropriate row by driving one wordline in the data array and one wordline in the tag array. Each array contains as many wordlines as there are rows in the array, but only one wordline in each array can go high at a time. Each memory cell along the selected row is associated with a pair of bitlines; each bitline is initially precharged high. When a wordline goes high, each memory cell

in that row pulls down one of its two bitlines; the value stored in the Memory cell determines which bitline goes low.

Each sense amplifier monitors a pair of bitlines and detects when one changes. By detecting which line goes low, the sense amplifier can determine what is in the memory cell. It is possible for one sense amplifier to be shared among several pairs of bitlines. In this case, a multiplexer is inserted before the sense amps; the select lines of the multiplexer are driven by the decoder. The number of bitlines that share a sense amplifier depends on the layout parameters described in the next section.

Cache components' sizes depend on following Cache Parameters:

- *C*: Cache size in bytes
- *B*: Block size in bytes
- *A*: Associativity
- *bo*: Output width in bits
- A_{bits}: Address width in bits

1.6 Cache Access Mechanisms

Scenario I: Block Size = 1 Word, Direct-Mapped Cache

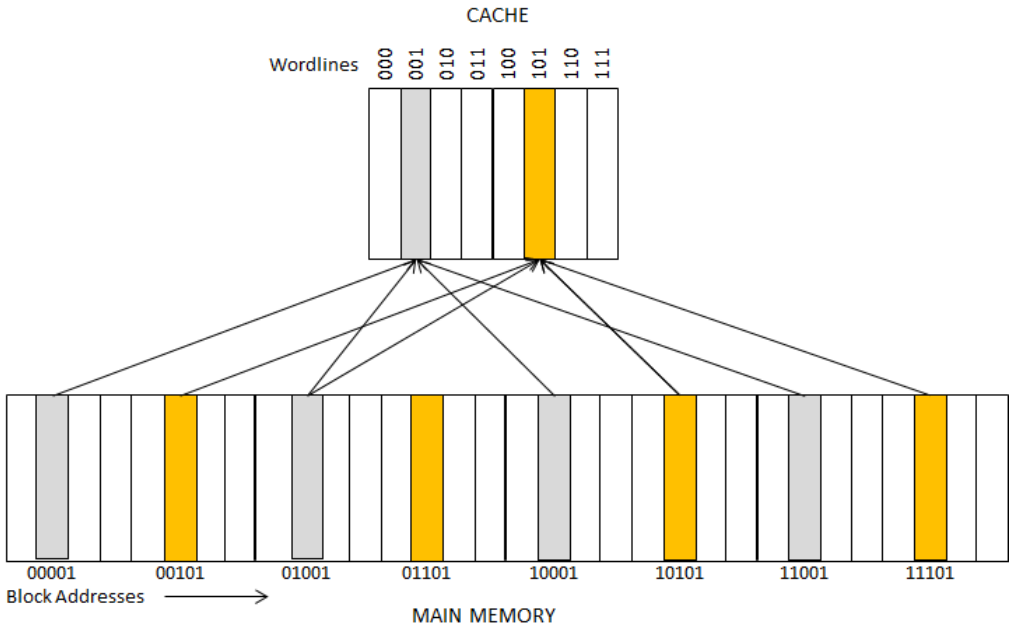


Figure 1.4 Direct Map Cache

The main memory can be thought to be divided into cache pages. The size of each page is equal to the size of the cache. Unlike the fully associative cache, the direct map cache may only store a specific line of memory within the same line of cache. For example, Line 1 of any page in memory must be stored in Line 1 of cache memory (indicated by grey colour in figure 1.4). Therefore if Line 1 of Page 0 is stored within the cache and Line 1 of page 1 is requested, then Line 1 of Page 0 will be replaced with Line 1 of Page 1. This scheme directly maps a memory line into an equivalent cache line, hence the name Direct Mapped cache.

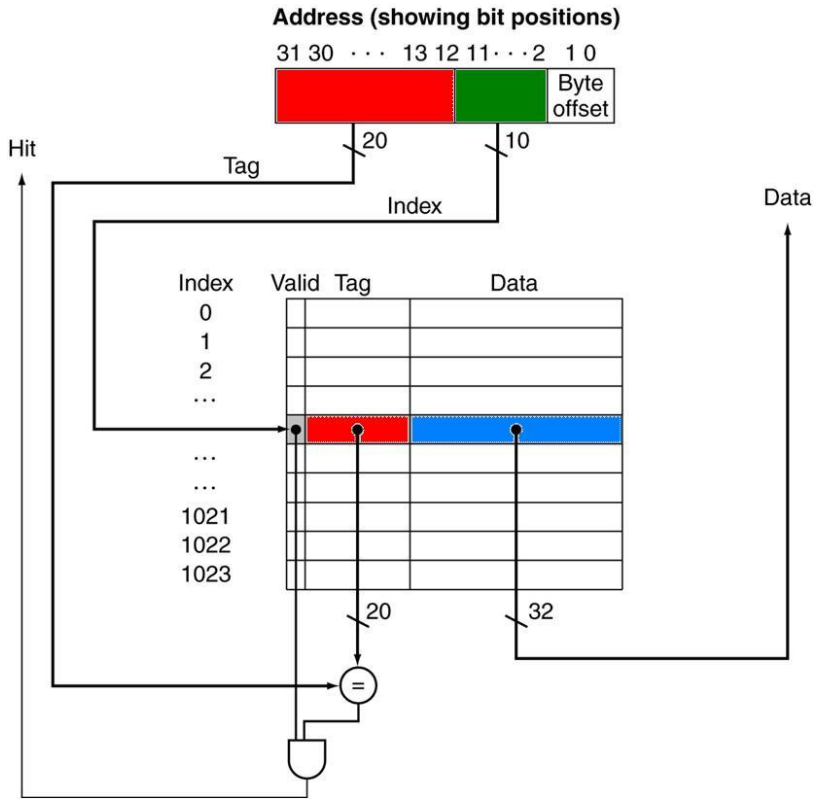


Figure 1.5 Direct Cache Access Mechanism

Assume Cache size 1024 words i.e. 4096 bytes, and main memory to be N times the cache size. Now as we see from figure 1.5, a word at an index of 2 in the cache may be mapped to location 2 or 1026 or 2050 and so on in the main memory. The mechanism successfully establishes the HIT signal by identifying the correct word correctly mapped to the main memory.

Lower 2 bits select byte within a Block (here one word). Next 10 bits select a Block or a Cache Line in the Cache. Each Block has additional 20-bit tag which is matched to identify to which page of the memory that word belongs. Further, valid-bit is checked if at all contents are present in the Block.

Scenario II: Block Size = 4 Words, Direct-Mapped Cache :: Locality of Space

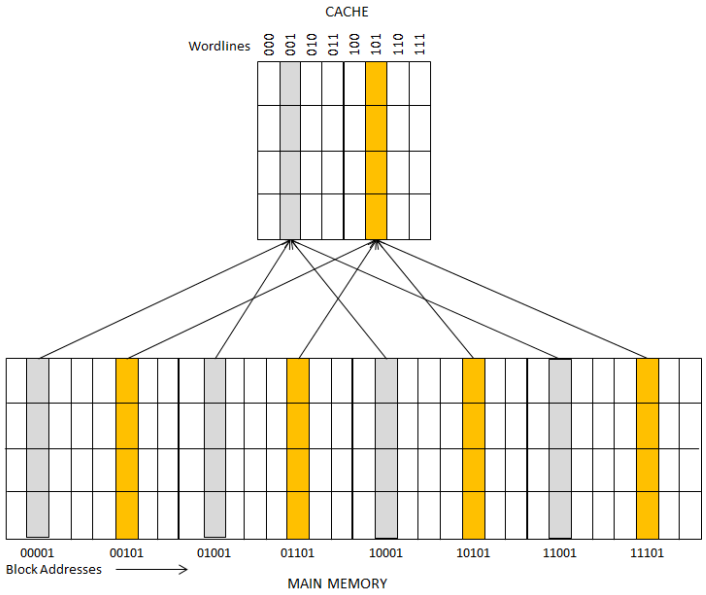


Figure 1.6 Direct Map Cache (Block Size 4)

Unlike above case now cache has block size of 4 words each. Notice if we have a cache of size 4096 words, we will have 1096 Blocks or Cache Lines each consisting of 4 words.

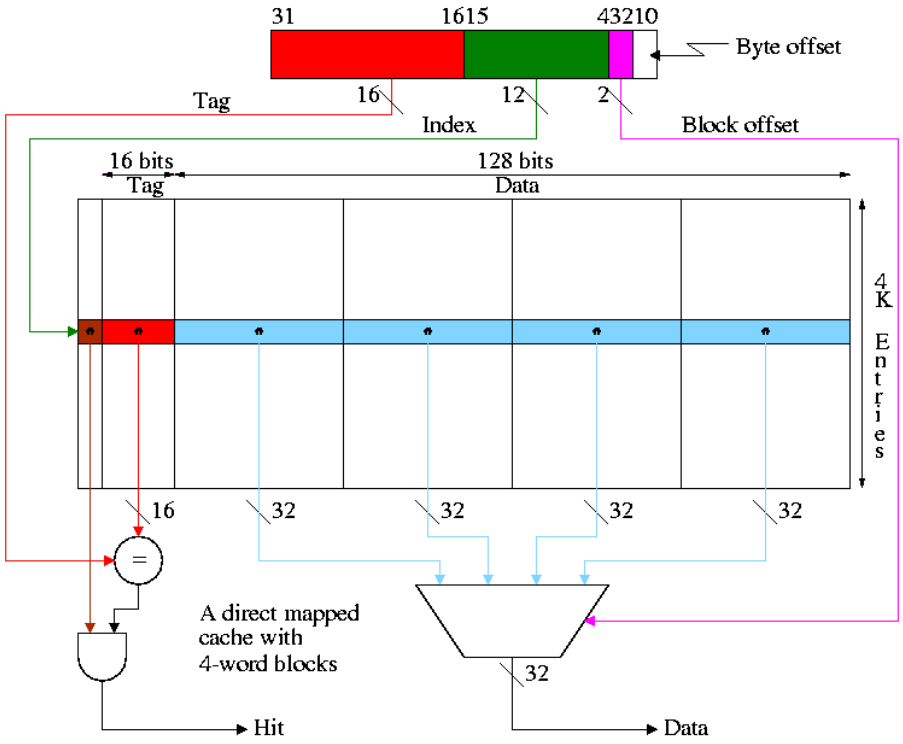


Figure 1.7 Cache with 4-word Block

Notice that the words within the block are addressed by the address lines split from the index which now provide the block offset.

Advantage: Note that this architecture of cache picks up 4 words from memory which will have single tag. Such architecture incorporates spatial locality: locality of space, by picking up three additional words instead of only one predicting that they may be required soon.

Scenario III: Block Size = 4 Words, 4-way Associative Cache:: Temporal Locality

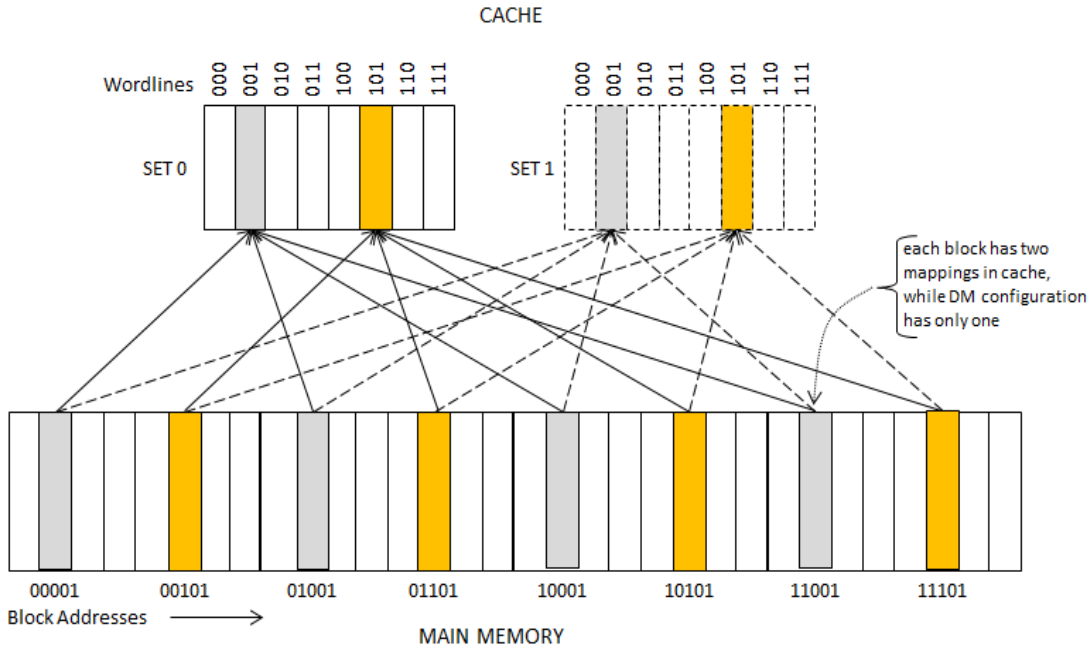


Figure 1.8 Set-Associative Cache Memory

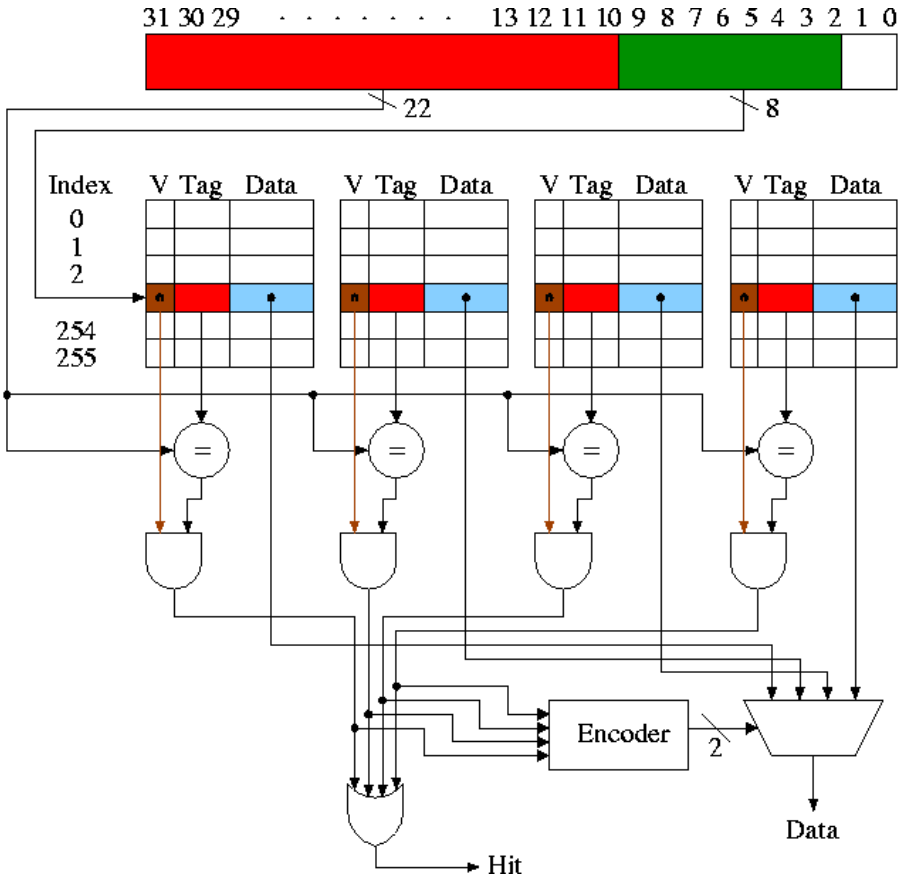


Figure 1.9 4-Way, 4-Word Block, Set Associative Cache

Here the index field as usually selects one row or the Cache Line. Then the tag matching happens in parallel fashion generating local Hit if any, across the 4 ways. If any of the way generates a Hit, appropriate Data Line is multiplexed to retrieve the data to CPU.

Advantage: This technique improves the temporal locality by organising Cache into a number of sets, thereby reducing conflict between often used blocks which otherwise map to the same set in the cache. But there is an overhead of increased hardware to implement the same.

CHAPTER 2

LITERATURE SURVEY

In previous chapter it has been shown that there are various considerations while designing the cache architecture. Currently lot of research is going on to find newer architectures for Cache designs so as to give better performance. Previously, performance and area were the two main factors involved in cache design, but recently power has become an increasingly more important design consideration. Largely, power consumption in cache occurs in data bit-lines, sense amplifiers and comparators. As the associativity is increased, the address decoding including tag comparison consume larger fraction of power [4]. Because larger associativity increases the number of parallel sets and hence increases the parallel tag comparisons in associative caches.

Conventional caches use full tag array holding tag value of each cache line in the cache. In associative caches, when degree of associativity increases the numbers of sets increase and hence the numbers of parallel tag arrays. In a typical case of 16KB 8Way Qword cache, there are 21 tag bits in one set, and a single memory access triggers comparison of 168bits (21x8=168) of 8 parallel 21-bit tag values. Such an operation consumes power and introduces delay.

Although a lot of research is also going on for better cell structures, but many different successful architectural proposals has been given providing better performances.

EXISTING POWER REDUCTION AND PERFORMANCE ENHANCEMENT SCHEMES IN CACHE ARCHITECTURE DESIGN:

2.1 Partial address directory for Cache Access

Operation

This technique utilizes approach to partially compare the tag array (address bits) to speed up the cache operation. It uses a separate partial tag array called Partial Address Directory (PAD), as partial address. The PAD can speed up most cache array access by accurately predicting cache locations without having to wait for results from conventional cache directory (Full Tag) lookups. In each memory access operation, the partial tag (PAD) comparison enables the selection bits to select data from data array using data multiplexer. The figure 2.1 shows 4 way set associative design [9].

In a normal operation, comparison in PAD late-selects one of the data values from different sets, without waiting Cache Directory lookups; thereby making the operation faster and power efficient.

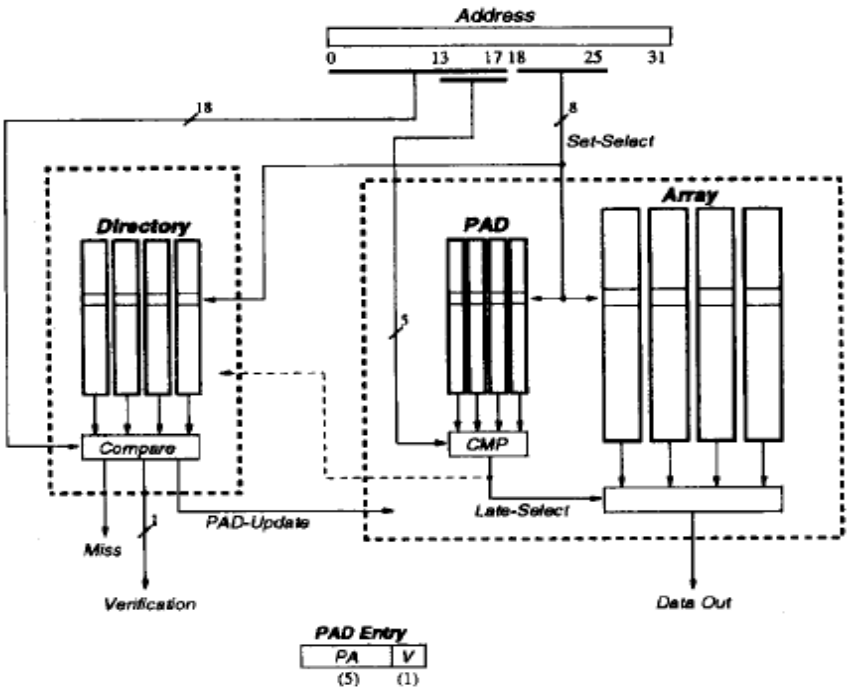


Figure 2.1 Partial Address Directory Architecture [9]

Using partial address comparison makes out a possibility of more than one match along the sets, which may trigger a fallacy situation where more than one selection bits of multiplexer are high. This situation (*called ghost hit*) has been rescued by incorporating the conventional

tag-array called *Cache Directory* too, into the architecture. When such a situation occurs, the conventional cache directory comes into play, giving correct select signals and updates PAD. As this operation is a read, altogether once again; it consumes additional machine cycle.

Gaps

- Design uses Cache directory (Full Tag) array and PAD (Partial Tag) array, i.e. two separate directory lookups. This will lead to consumption of larger area as compared to conventional design.
- Two tag arrays will use two set of comparators, consuming larger power.
- Also author proposes the Cache Directory to be implemented separately, only PAD to be imbedded into cache structure. It may result into larger access times when accessing a distant Full Tag Directory upon *ghost hit*.

Motivation

- The design achieves greater speeds for greater address bits, hence for the designs with greater associativity.
- Also as the number of sets increase, the number of distinct partial addresses (distinct PAD entries) per set increase.
- Although author proposes possibility of reducing Full Tag array thereby modifying the design with two arrays containing two portions of the full tag address i.e. two arrays having separate 16 and 5 bits of full 21 bit tag.

Summary

- The design uses PAD as faster portion of Cache, and it achieves better speeds of operation consuming larger area and lesser flexibility.
- The author hasn't clearly reported comparison of power consumption.

2.2 Partial Tag resolution in Data value predictors

This technique uses partial tag for data value prediction, resolving data dependences thereby increasing the Instruction Level Parallelism (ILP) for processing at considerable low hardware costs.

As clear in the figure 2.2, the conventional Full Resolution tag implementation stores full tag value into the Data/Tag array. The partial resolution figure 2.3, implementation stores only a

part of tag bits. Owing to address sequences being linear in nature, only small portion of tag is able to uniquely predict the correct data values.

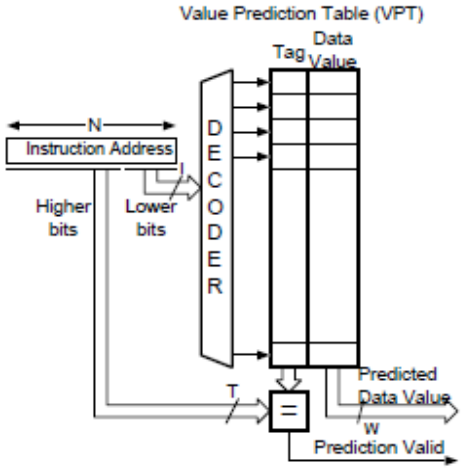


Figure 2.2 Full Resolution VPT [11]

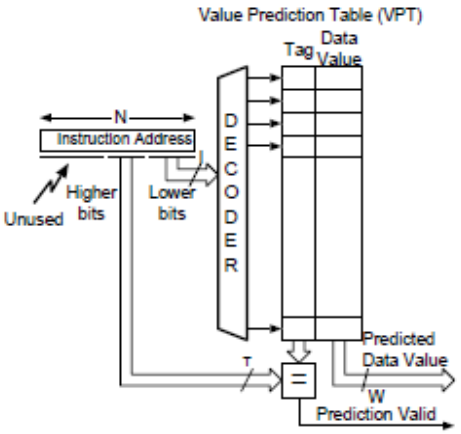


Figure 2.3 Partial Resolution VPT [11]

Operation

Lower address bits select the decoded line, and a portion of higher order address bits (tag) is then matched with the stored tag bits from the decoded line [11].

For an instruction address of N bits and VPT indexed by I bits, the tag address is $T = N - I - 2$ bits. But using partial resolution tag address is $T < N - I - 2$.

Gaps

This work introduces various partial tag resolution policies to reduce hardware costs, not the reduction of dynamic power.

Motivation

- The Value predictors for Data Speculation don't require full resolution tag values.
- Destructive aliasing if introduced by partial resolution can be avoided by deciding enough bit length of tag address.
- The contribution of this work is the introduction of various partial tag resolution policies, such as stride, last-value, two-level, etc. However, the main focus of this work is reducing hardware costs, not the reduction of dynamic power; thus, the results of power management are not provided in their work.

2.3 Cost-effective value prediction using partial tag

This used above technique of reduced tag comparison in the caches. It also introduces value prediction policies. The partial tag address is generated by hash function in this technique [12].

Motivation

- It also shows the possibility of partial address generation and provided the hit ratios for each partial address generation method in Direct-Mapped caches as suggested by previous technique.
- The technique proposes the policies for Direct Mapped Caches not set associative caches.

2.4 Partial Tag structure: Early switching of sense amplifiers

This technique adds a very small tag dedicated for the amplifiers. This small tag array is only several bits wide. It is organized and accessed as a regular tag.

The architecture of this partial comparison cache is given Figure 2.4(a), with the corresponding cache access paths shown in Figure 2.4(b). The shadowed path is the new SE path.

Operation

The small tag array keeps a copy of the least significant bits of the original tag. For each way, a few significant bits are compared. Only when there is a match, the sense amplifiers attach to the data array bit-lines will be enabled. Because of its small size, the new hardware can complete the comparison operation using only 60-80% of the original comparison time [10].

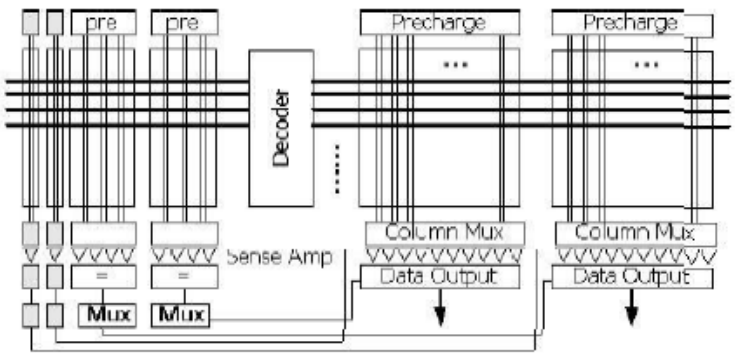


Figure 2.4(a) Early Switching of Sense Amplifiers [10]

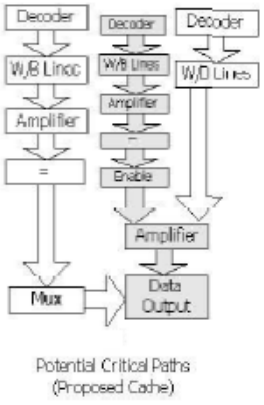


Figure 2.4(b) Critical Path [10]

Gaps

- The technique uses additional small tag array in addition to the regular tag array. Not area effective.
- False-hits are possible in partial tag comparisons, in which the results of comparisons seem to be cache hits but are actually false hits.
- Technique only emphasizes the power reduction in the multiple sense amplifiers on the data-array side, neglecting power consumption in comparators, which compare a large number of tag bits in the set associative caches.

Motivation

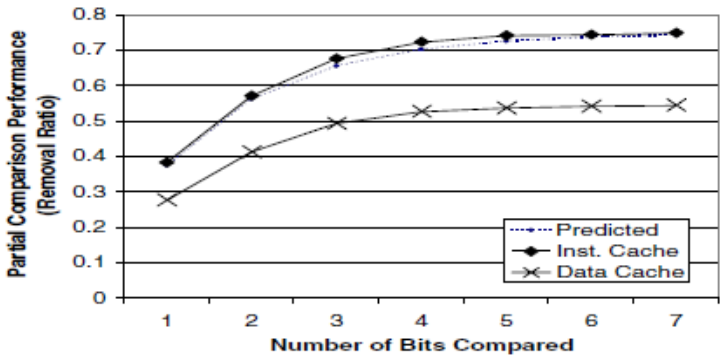


Figure 2.5 Partial Comparison vs. Number of Bits Compared [10]

- Technique successfully uses partial tag comparison to enable the sense amplifiers well before the data reaches them, thereby enabling only one sense amplifier depending on the way selected.
- The technique is well designed for the set-associative caches, and reports 25-60% power efficiency over conventional technique.

2.5 Tag Overflow Buffering

Operation

Tag is broken into two parts. The LSB-side (*called TagL*) is stored normally in the cache as regular structure. The MSB-side (*called TagH*) of tag is stored in a separately into an external register called Tag Overflow Buffer. Similar to conventional cache operation, the TagL corresponding to the decoded wordline is compared with TagL address field generating TagL Hit/Miss. For the TagH side, the input TagH is compared with the existing contents of TOB,

if it is also Hit cache operation is completed successfully. Upon miss, the Locality Change Detection mechanism predicts for the degree of change of locality, if requested address corresponds to a different locality plus the number of requests exceeds threshold, then the TOB is updated with the candidate TagH address [13].

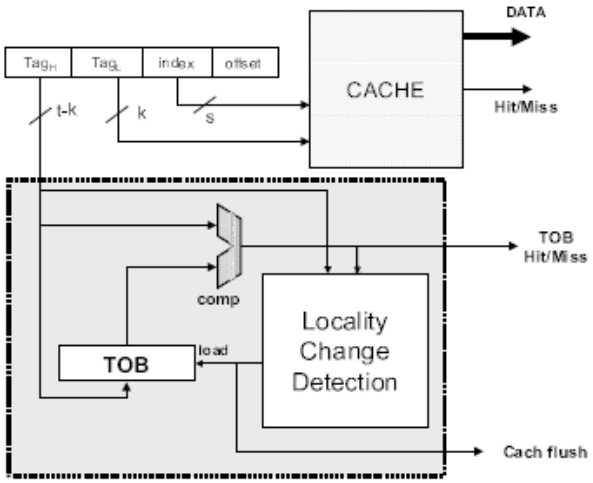


Figure 2.6 Dynamic TOB-Based Architecture [13]

Gaps

- The architecture becomes very fragile when application program frequently change memory access localities i.e. the hit ratio can dip.
- Change of associativity has no impact on the power savings, the power reduction in these cases is no better than Direct Mapped Caches.

Motivation

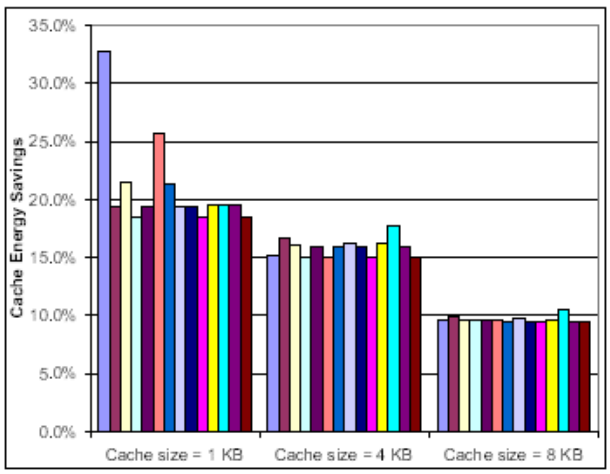


Figure 2.7 Percentage Energy Savings [13]

- The architecture is helpful for reducing false hits.
- In this policy, the average energy savings on the tag array is about 48%, corresponding to a savings of about 20% on the total cache energy.

2.6 Compressed Tag Architecture

This technique uses a Locality Buffer called *LoB* like *TOB* but has more than one entry in it. Like in the previous techniques it also utilizes partial tag comparison in the cache structure, enable lower power consumption. Similar to techniques discussed earlier LSB-side tag called *TagL* is stored normally as a part of cache structure. The *TagH* is stored in accordance with the policies in the locality buffer containing a few entries. Also to select between the *LoB* entries *LCB* bits are also allocated along with the *TagL* bits.

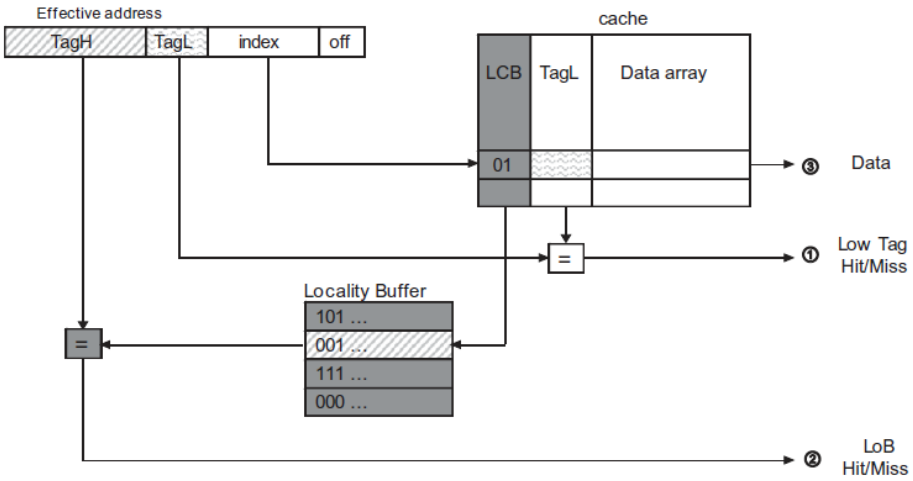


Figure 2.8 Compressed Tag Architecture of Cache [14]

Operation

Similar to conventional cache operation, lower tag bits i.e. *TagL* are compared resulting in Low Tag Hit/Miss. Also *LCB* bits corresponding to the selected wordline are used to select the *LoB* entry, which is compared with the *TagH* to generate *LoB* Hit/Miss.

Low Tag Hit along with *LoB* Hit convey an overall Hit operation. On the other hand, a *LoB* miss will trigger the other design policies [14].

Gaps

The design policies limit it for Direct Mapped Caches only i.e. a similar design when applied to set-associative caches give no better results.

Motivation

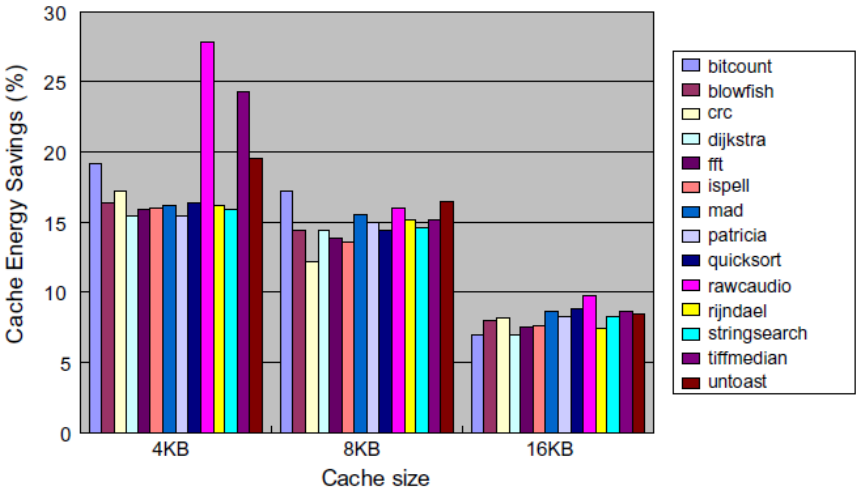


Figure 2.9 Percentage Energy Savings [14]

- The proposed architecture is free from false hits.
- The design establishes quite lesser degree of fragility and can handle frequent program locality changes.
- The design provides a flexible sensitivity to the frequency of change of program localities.

CHAPTER 3

PROPOSED REDUCED TAG ARCHITECTURE

3.1 OVERVIEW

As discussed in the previous chapter, there has been several reduced tag architectures designed for low power and/or smaller access times in different cache configurations. Most of these designs have been developed in simulation environments such as SimpleScalar Toolset, CACTI, PowerStone Suite, Platune Simulation Platform, SystemC and also VHDL; along with benchmark suites such as SPEC CPU2000, PowerStone, MiBench, CacheBench etc.

The conventional cache design with simple hardware and architectural policies is the dominating candidate till date; it has been shown that this design is still providing quite bearable power and access times.

We intend to develop an architecture consuming a considerable lower power, with negligible to zero performance degradation. Our design is inspired from the design proposed by Kwak and Jeon. They proposed the basic reduced tag architecture, with simple cache policies for low power in Direct Mapped Caches. We take their design further to set associative caches with little hardware modifications but several cache policy enhancements to make a considerable low power cache architecture.

3.2 PROPOSED REDUCED TAG CACHE ARCHITECTURE

In most reduced tag direct map architectures, a small part of tag bits is stored conventionally in the cache structure and the bigger portion of tag field (*TagH*) is stored separately not in regular cache structure. The basic idea was and is that program localities normally do not change much frequently, and most of the entries in the tag array have a large part of the tag bits common, therefore one can accumulate that common larger fraction of tag bits separately reducing the effective tag bits in the cache structure. The similar design approach can be

utilized with no additional hardware but with crucial changes in cache replacement policies to make a suitable design for set-associative caches.

The block diagram of the proposed design is shown in figure-3.1. Basic architecture of the cache is similar to the conventional architecture, except the larger fraction of tag bits (*TagH*) are stored separately called *Locality Buffer (LoB)*.

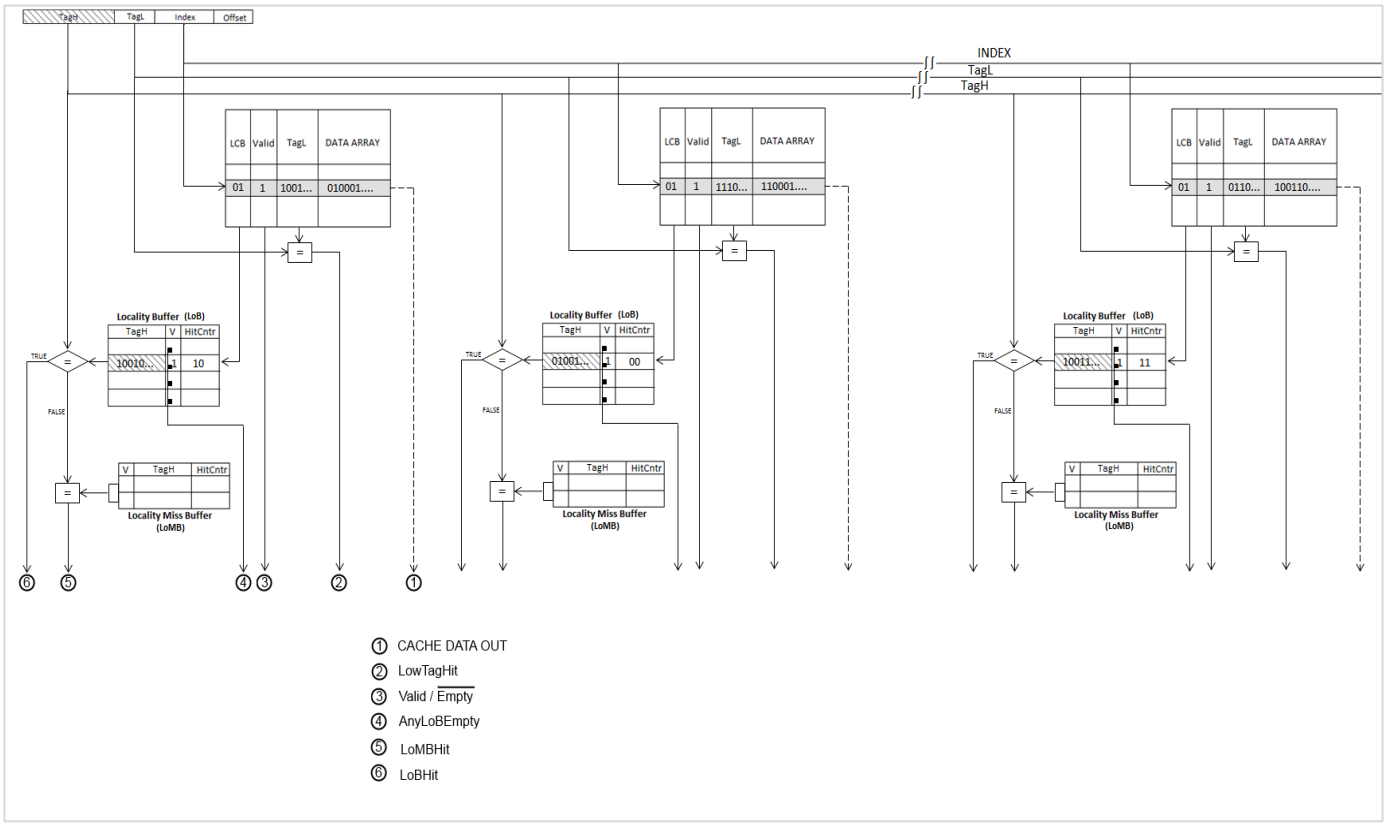


Figure 3.1 Proposed Low Power Architecture for Set-Associative Caches

The fundamental idea is to reduce the number of tag bits in parallel sets, thereby reducing the total no. of bits compared. The simulation results depict that one can achieve a power dissipation improvement from 10-60% in different set-associative cache configurations.

The architecture relies on modified *Masked FIFO replacement policies* (discussed shortly) to achieve the performance equivalent to the conventional architecture. A step by step description is as follows:

3.3 STRUCTURAL DETAILS OF ARCHITECTURE

- **Address Field**

The address field is decomposed into *TagH, TagL,* Index and the Offset. The Offset field selects the byte from the data read from Cache. Index selects the wordline along all parallel sets in the Cache. Tag is decomposed into TagH and TagL, where TagL is stored in the Cache.

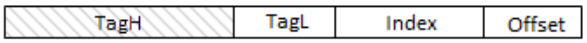


Figure 3.2 Address Field Decomposition

- **SRAM Cache**

The cache structure is similar to conventional cache. The difference is that the complete Tag is not stored, rather small portion TagL is stored (design feature). LCB are a few bits stored to map each wordline with corresponding TagH entry.

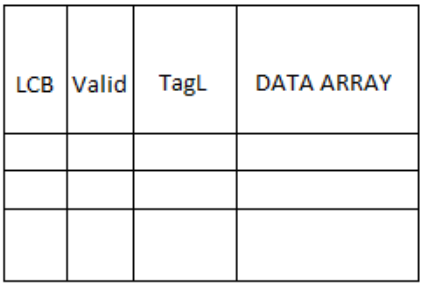


Figure 3.3 SRAM Cache

- **LCB Bits**

LCB are stored in the cache structure along with TagL bits. The number of LCB bits determines the number of LoB entries with a relation:

$$LoB\ rows = 2^{LCB\ bits} \qquad \ldots(1)$$

For example if the number of LCB bits is 3 the LoB entries are 8.

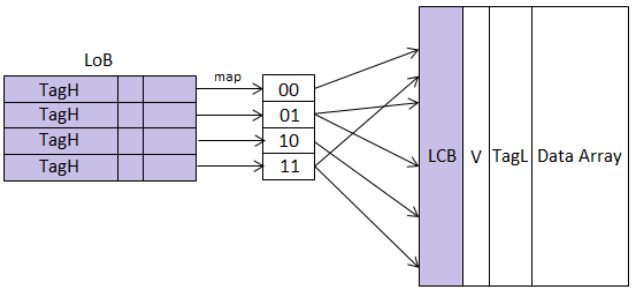


Figure 3.4 Locality Buffer and LCB Bits [14]

As already discussed, LoB keeps the TagH values of the requested addresses, the more the program locality changes the more the number of distinct TagH values. The program

execution can change the locations frequently and repeatedly, requiring different memory blocks to be accessed in frequently and repeatedly in the near vicinity of execution.

Table 3.1 Test Case: Choosing number of LCB Bits

Sr. No.	Address 32-Bit	Tag 24-Bit	TagH 20-Bit	TagL 4-Bit	Index 4-Bit
1	0x02021d40	0x02021d	0x02021	0xd	0x4
2	0x02021d48	0x02021d	0x02021	0xd	0x4
3	0x0201bd68	0x0201bd	0x0201b	0xd	0x6
4	0x0201bd60	0x0201bd	0x0201b	0xd	0x6
5	0x02021d44	0x02021d	0x02021	0xd	0x4
6	0x0201bd64	0x0201bd	0x0201b	0xd	0x6

As an example, consider single set of cache for the time, a situation as in the table 3.1can come, where two different memory blocks (differentiated by TagH) are accessed frequently and repeatedly by the processor, that map to two different set of memory addresses differentiated by TagH. Requested set of addresses have different Tag and Index. When the first address is accessed the TagH (0x02021) is stored in a LoB entry, giving a *cold miss*. Address request 2 results into a hit as TagH (0x02021) is already stored in the LoB. 3rd address request has a different TagH (0x0201b) at a different index. If the LoB had only one entry like in *Tag Overflow Buffering* (TOB) technique, the operation would be to replace the existing TagH (0x02021) with new TagH (0x0201b). There are two different TagH values, which are accessed alternatively and repeatedly. Therefore alternate requests would result into subsequent misses.

Therefore it is clear that there must be sufficient LoB entries so that the required number of different and frequently occurring TagH values can be stored. The sufficient number of LoB entries as dictated by Kwak and Jeon is 4. Therefore optimum number of LCB bits from relation (1) is 2.

It is to be re-emphasized that LoB entries are to keep alternatingly occurring different TagH values, i.e. to cover alternating program execution localities; not to keep all different TagH values that occur during program execution as a whole.

- **Locality Buffer**

It stores the program's current memory address locality. In other words, the larger portion of MSB-side of Tag bits is stored in the LoB. Locality Buffer holds a valid/$\overline{\text{empty}}$ bit to indicate whether the LoB entry is valid. Another field stores the Hit-Counter, accounting for the number of times the LoB entry had been requested.

TagH	V	HitCntr

Locality Buffer

- **Locality Miss Buffer**

It is used to store the yet another memory address locality just in case all the LoB entries are filled. The working operation of the LoB and LoMB will be clear shortly when we discuss the operation in detail.

TagH	V	HitCntr

Locality Miss Buffer

3.4 READ OPERATION

The address field is decomposed into *TagH*, *TagL*, Index and the Offset. Decoded Index as usually selects the wordline in all sets/ways in parallel. The fields – Data (①), TagL and Valid bit (③) are accessed. For tag matching, TagL read from the each way in the cache is compared to the requested TagL conventionally, generating LowTagHit signal (②.

Figure 3.5 - Address Field Decomposition

As already discussed, TagL is the reduced LSB portion of the complete tag field. This reduced tag comparison is the key design feature leading to lower power in set-associative caches.

But to maintain correct functionality complete tag matching has to happen. Therefore, LCB (Locality Compressed Bits) bits (of very small bit length) are introduced. LCB select the TagH entry in the LoB. TagH entry read from LoB is compared with the requested TagH generating LoBHit signal (⑥). Read operation is a HIT when both LowTagHit and LoBHit are HIGH in any of the set/way.

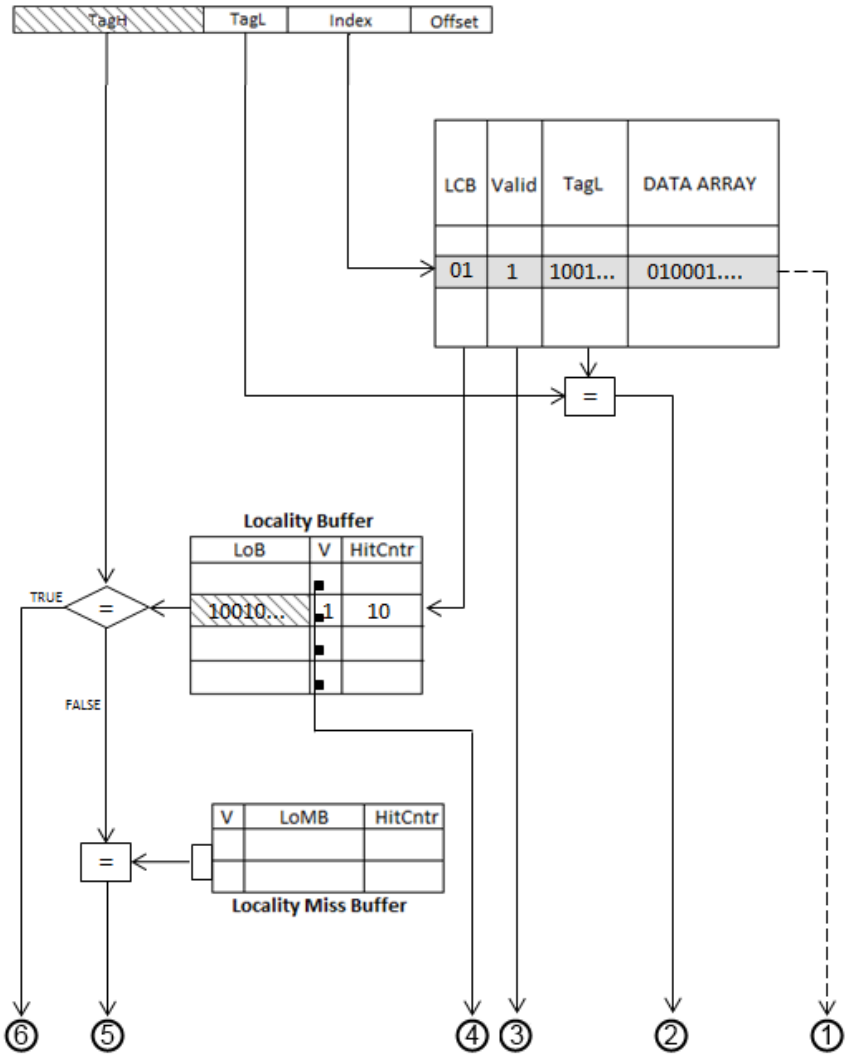


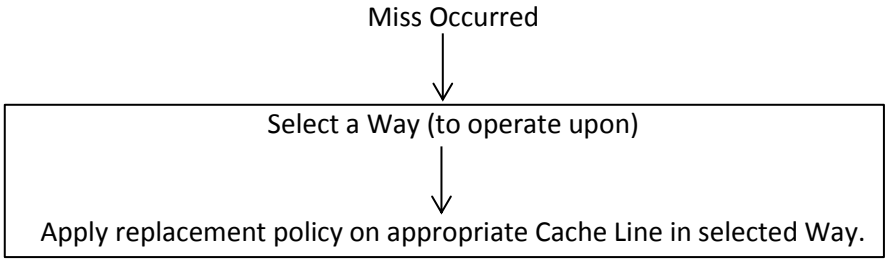
Figure 3.6 - Read Operation (Showing one Way only)

A successful TagL match is called *LowTagHit*. Successful LoB match is called a *LoBHit*. When both *LowTagHit* and *LoBHit* occur in any of the set/way, the final **Hit** is said to be occurred.

However if any of mismatch occurs in TagL or TagH, a final **Miss** occurs. Always when a miss occurs, some mechanisms are triggered called replacement policies or replacement mechanisms, because then some or the other Cache Lines are to be replaced.

3.5 HANDLING A CACHE MISS – *REPLACEMENT POLICIES*

A cache miss can occur due to two reasons in the current design – LowTagMiss or LoBMiss or both. Upon a cache miss in Direct Mapped Caches, a cache line is replaced following replacement policies. In set-associative caches a way has to be selected prior to perform replacement mechanism. Therefore the mishandling operation takes two steps:



These two steps are explained in detail as follows:

3.5.1 Way Selection

Conventional Method

In conventional set-associative caches the ways are selected using policies like Random, LRU, FIFO or MRU. FIFO has been the optimum design choice, as it is easier to implement and provides reasonable performance in terms of hit rate.

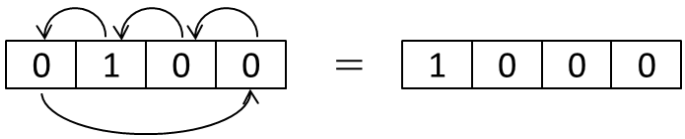


Figure 3.7 - Conventional FIFO based Way Selection

But the conventional method doesn't hold for the reduced tag architecture design.

3.5.1.1 Conventional FIFO unsuitable for current design

The conventional FIFO as discussed above is not suitable for the current design. To understand this, consider the following test case sheet. It is the simulation log of sample address inputs given to cache configuration – 16KB 4Way 4Word Cache with TagL size 4 bits.

	Address 32- bit	Tag 20-Bit	TagH 16-Bit	TagL 4-Bit	Index 8-Bit	HIT	Modified FIFO	Way4 LoB	Way4 LCB	Way4 TagL	V	HIT	Way3 LoB	Way3 LCB	Way3 TagL	V	HIT	Way2 LoB	Way2 LCB	Way2 TagL	V	HIT	Way1 LoB	Way1 LCB	Way1 TagL	V	HIT
1	0x02000eb0	0x02000	0x0200	0	eb	0	0b00001	00000,00000,00000,00000	0x0	0x00	0	0	00000,00000,00000,00000	0x0	0x00	0	0	00000,00000,00000,00000	0x0	0x00	0	0	00000,00000,00000,00000	0x0	0x00	0	0
2	0x02010eb4	0x02010	0x0201	0	eb	0	0b00010	00000,00000,00000,00000	0x0	0x00	0	0	00000,00000,00000,00000	0x0	0x00	0	0	00000,00000,00000,00000	0x0	0x00	0	0	00200,00000,00000,00000	0x0	0x00	1	0
3	0x02001eb0	0x02001	0x0200	1	eb	0	0b00100	00000,00000,00000,00000	0x0	0x00	0	0	00000,00000,00000,00000	0x0	0x00	0	0	00201,00000,00000,00000	0x0	0x00	1	0	00200,00000,00000,00000	0x0	0x00	1	0
4	0x02011eb4	0x02011	0x0201	1	eb	0	0b01000	00000,00000,00000,00000	0x0	0x00	0	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00000,00000,00000	0x0	0x00	1	0	00200,00000,00000,00000	0x0	0x00	1	0
5	0x02020eb0	0x02020	0x0202	0	eb	0	0b00001	00201,00000,00000,00000	0x0	0x01	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00000,00000,00000	0x0	0x00	1	0	00200,00000,00000,00000	0x0	0x00	1	0
6	0x02020eb4	0x02020	0x0202	0	eb	1		00201,00000,00000,00000	0x0	0x01	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00000,00000,00000	0x0	0x00	1	0	00200,00202,00000,00000	0x1	0x00	1	1
7	0x02041eb0	0x02041	0x0204	1	eb	0	0b00001	00201,00000,00000,00000	0x0	0x01	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00000,00000,00000	0x0	0x00	1	0	00200,00202,00000,00000	0x1	0x00	1	0
8	0x02041eb4	0x02041	0x0204	1	eb	1		00201,00000,00000,00000	0x0	0x01	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00000,00000,00000	0x0	0x00	1	0	00200,00202,00204,00000	0x2	0x01	1	1
9	0x02002eb0	0x02002	0x0200	2	eb	0	0b00010	00201,00000,00000,00000	0x0	0x01	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00000,00000,00000	0x0	0x00	1	0	00200,00202,00204,00000	0x2	0x01	1	0
10	0x02012eb4	0x02012	0x0201	2	eb	0	0b00001	00201,00000,00000,00000	0x0	0x01	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00200,00000,00000	0x1	0x02	1	0	00200,00202,00204,00000	0x2	0x01	1	0
11	0x02003eb0	0x02003	0x0200	3	eb	0	0b01000	00201,00000,00000,00000	0x0	0x01	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00200,00000,00000	0x1	0x02	1	0	00200,00202,00204,00201	0x3	0x02	1	0
12	0x02013eb4	0x02013	0x0201	3	eb	0	0b00100	00201,00200,00000,00000	0x1	0x03	1	0	00200,00000,00000,00000	0x0	0x01	1	0	00201,00200,00000,00000	0x1	0x02	1	0	00200,00202,00204,00201	0x3	0x02	1	0
13	0x02022eb0	0x02022	0x0202	2	eb	0	0b00010	00201,00200,00000,00000	0x1	0x03	1	0	00200,00201,00000,00000	0x1	0x03	1	0	00201,00200,00000,00000	0x1	0x02	1	0	00200,00202,00204,00201	0x3	0x02	1	0
14	0x02022eb4	0x02022	0x0202	2	eb	1		00201,00200,00000,00000	0x1	0x03	1	0	00200,00201,00000,00000	0x1	0x03	1	0	00201,00200,00202,00000	0x2	0x02	1	1	00200,00202,00204,00201	0x3	0x02	1	0
15	0x02043eb0	0x02043	0x0204	3	eb	0	0b00010	00201,00200,00000,00000	0x1	0x03	1	0	00200,00201,00000,00000	0x1	0x03	1	0	00201,00200,00202,00000	0x2	0x02	1	0	00200,00202,00204,00201	0x3	0x02	1	0
16	0x02043eb4	0x02043	0x0204	3	eb	1		00201,00200,00000,00000	0x1	0x03	1	0	00200,00201,00000,00000	0x1	0x03	1	0	00201,00200,00202,00204	0x3	0x03	1	1	00200,00202,00204,00201	0x3	0x02	1	0

Having a clear look into the above log, one can see that there are situations in which conventional FIFO is not followed. A distributed summary with all possible issues is given below:

a) A Way has an empty LoB entry available

Sr. no. 10 in the above log table states the situation. Consider a figure 3.8 below. It is possible that all the ways have their cache lines filled.

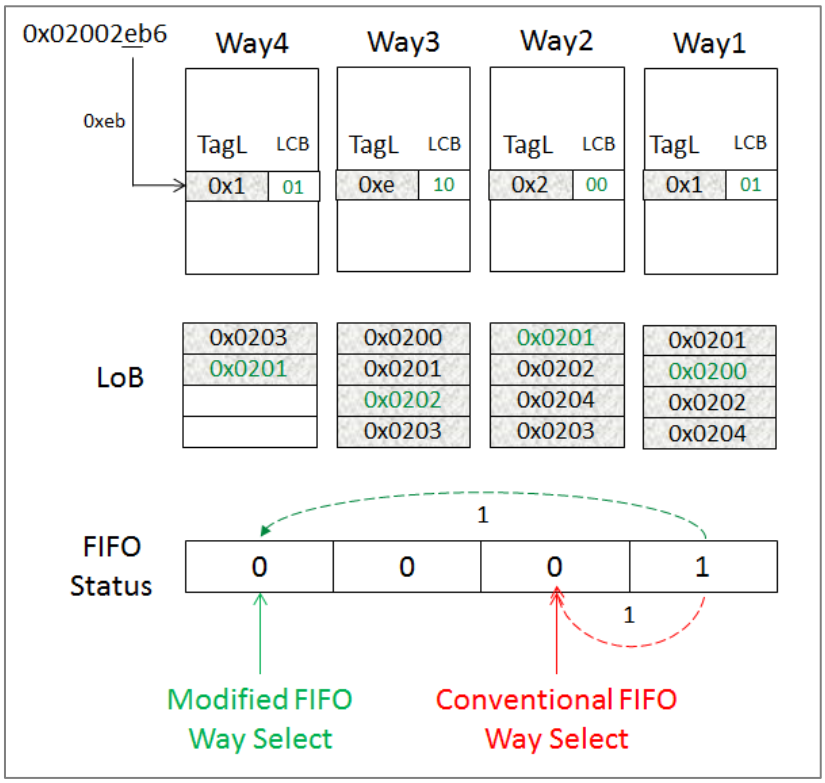


Figure 3.8 - FIFO Issue1

A new request of address 0x02002eb6 occurs, in which tag field contains 0x02002. While all the ways for the requested line; i.e. line corresponding to index field (0xeb); are already filled with different tags. As summarized in the table below, in none way are the TagL and TagH both matched i.e. in none of the ways complete tag matches. Therefore the requested address is not found in the cache and a MISS is notified.

Table 3.2 - Tag Matching*

Requested Tag	Way4 Tag	Way3 Tag	Way2 Tag	Way1 Tag
0x02002	0x02011	0x0202e	0x02012	0x02001

*WayN Tag = TagH + TagL

As it is clear in the figure-3.8, FIFO status is 0001; meaning that last replacement mechanism happened for way number 1. A conventional FIFO would trigger the next way for replacement, i.e. way number 2.

Here is the issue. It is clear that a cache line has to be replaced for sure, but there comes a difference in way selection. Suppose conventionally Way2 is selected to apply replacement. Along with cache line, LoB entry with TagH 0x0201 is replaced leading to a possible subsequent misses. This is because in this particular architecture, LoB carries all the possible TagH values for the corresponding Way. And if a LoB entry is replaced all the address requests corresponding to entry containing TagH, with different index are affected. In other words, treating conventional FIFO, a single conflict miss can trigger several capacity misses in the current architecture.

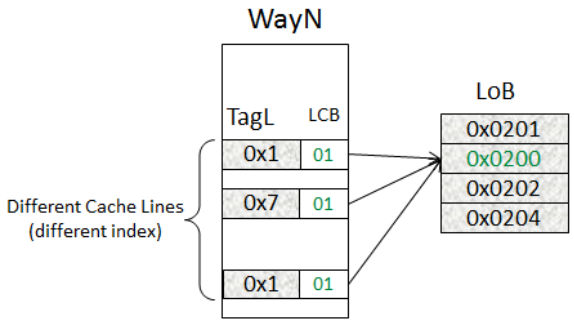


Figure 3.9 - Importance of one LoB Entry

This is pictorially presented in the figure – 3.9, one LoB entry holds the TagH value for many cache lines in a Way. Therefore if a Way is available with an empty LoB entry in a situation where LoBHit among all Ways are FALSE, it must be chosen preferably overriding the FIFO operation.

The modified FIFO algorithm/operation will be discussed shortly.

b) A Way has matching LoB Entry available

A similar situation happens when a different way other than conventionally target Way has a matching LoB entry. This is pictorially presented in the figure – 3.9. In the figure – 3.9 (above) it is already shown how crucial is one LoB entry.

Requested address with Tag field (0x02032) with TagH (0x0203) is not found in any of the Ways in the Cache. Conventional FIFO directs Way2 to be operated for replacement, inspite of the fact that there is a LoB entry in Way3 that contains TagH (0x0203) but *non-mapped*.

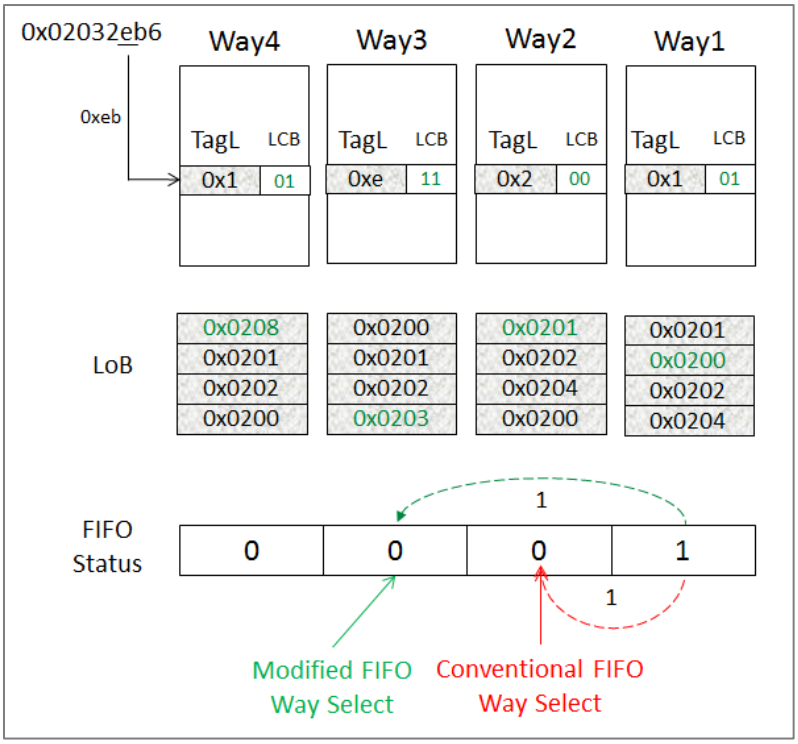


Figure 3.10 - FIFO Issue2

Thus if Way3 rather than Way2 is selected a valuable entry from Way2 is saved from getting replaced. Therefore preferably that Way should be selected which have at least a LoB HIT even though have LoWTag MISS.

Selecting a Way using conventional FIFO can replace a valuable LoB entry in a Way. Therefore in current architecture, as one LoB entry is much important there must be an equally good replacement mechanism which could intelligently select the appropriate Way for replacement. A modified FIFO rather a masked FIFO is proposed as below for the current design.

NOTE: As will be seen in subsequent sections that not only LoBHit rather AnyLoBHit can be a better design parameter while doing the Way Selection. It will be clearer in the next section.

3.5.1.2 Masked FIFO for Reduced Tag Architecture

Conventional FIFO needs to be overridden by some preferable situations as described in the previous section. Therefore a modified rather overridden FIFO is proposed here. To take into consideration the different situations, one has to receive the current status of the cache and the LoB Buffer as the situation parameters.

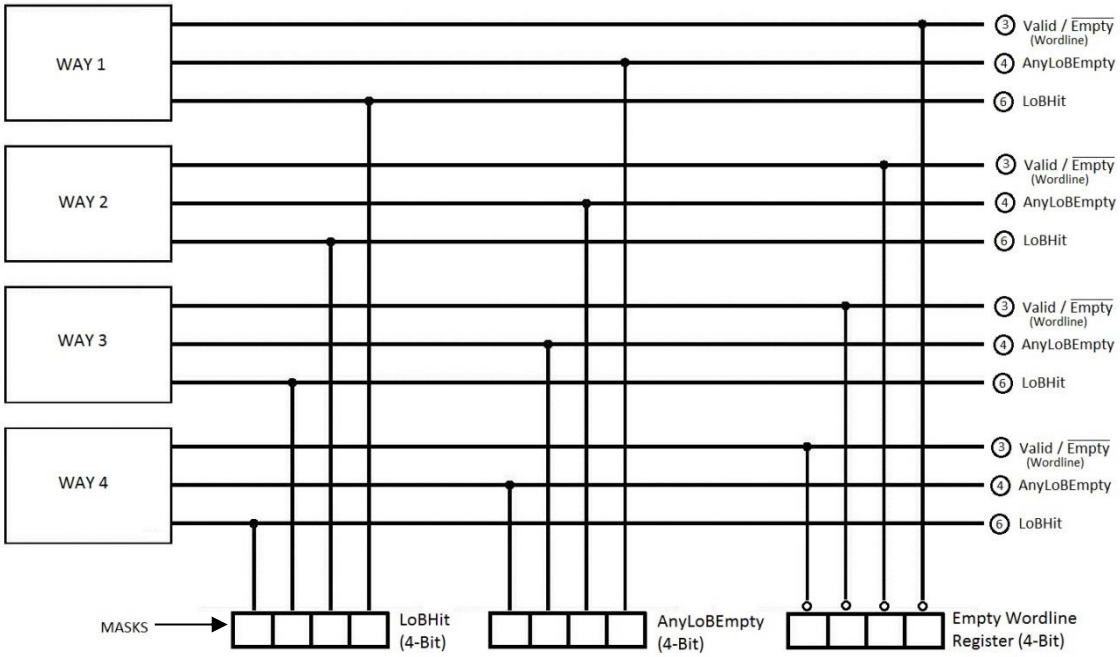


Figure 3.11 - Generating Masks for FIFO

In figure 3.11 it is shown that how the status of cache is used to derive the FIFO operation, by generating masks. Three different signal values for each Way are loaded: Valid / Empty, AnyLoBEmpty, LoBHit. These masks are collectively used to generate a FIFO Mask register. The following simple illustrative flowchart makes it easier to understand the FIFO Mask generation process.

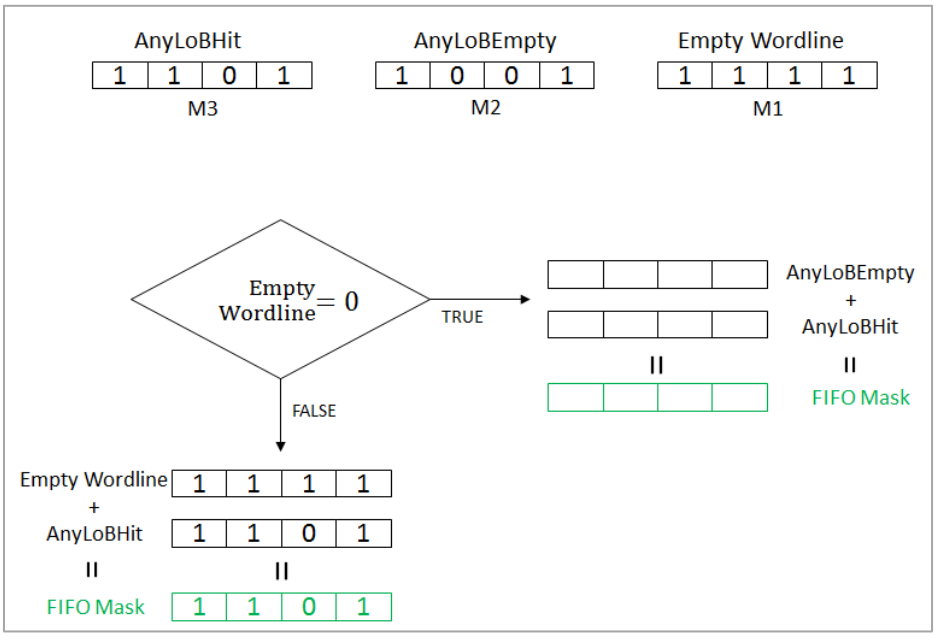


Figure 3.12 - FIFO Mask Generation

After the FIFO mask is generated, FIFO register in conjunction with FIFO Mask are used to derive the Way Selection process. It will be clearer with the help of example in figure 3.13. Suppose a FIFO Mask generated has a value 1101, meaning Way number 1,3 and 4 are the best possible Ways to be selected; the FIFO eliminates the Way number 2 and selects Way number 3. Whereas the conventional Way selection would just select Way number 2.

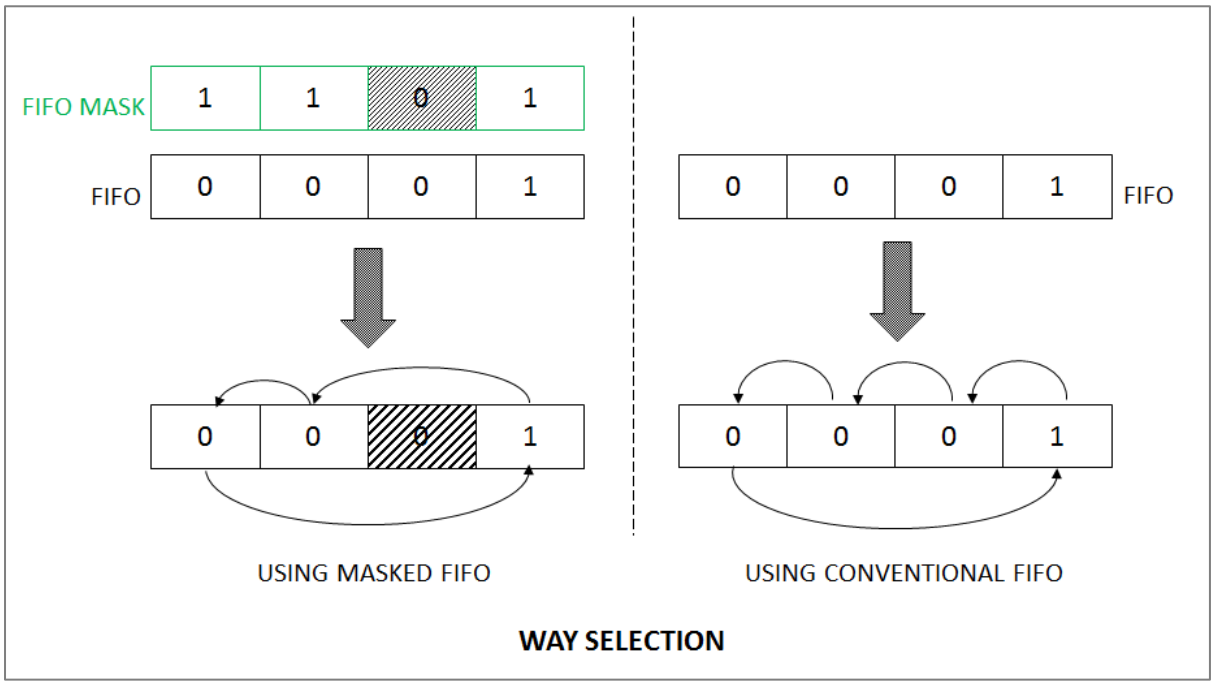


Figure 3.13 - WAY SELECTION MECHANISM

To conclude, in the proposed reduced tag architecture, there are limited TagH entries available for all the wordlines in a Way, therefore before replacing the LoB entry (i.e. TagH) one has to consider if other better selections are possible irrespective of conventional FIFO selection. In this architecture there are certain such possible situations. Therefore as explained above, a Masked FIFO selection is proposed for the current architecture.

3.5.2 Replacement policies for selected way

Once the way is selected, there are replacement mechanisms to be followed inside the selected way. These replacement mechanisms are independent of the degree of associativity; well it obviously means that these are applicable for Direct Mapped caches as well. Both the conventional replacement policies and proposed replacement policies are discussed in detail as follows:

a) Conventional Replacement Policies

In conventional replacement mechanism once the way is selected, the cache line corresponding to index field is replaced from the lower memory hierarchy directly as described in the figure 3.14.

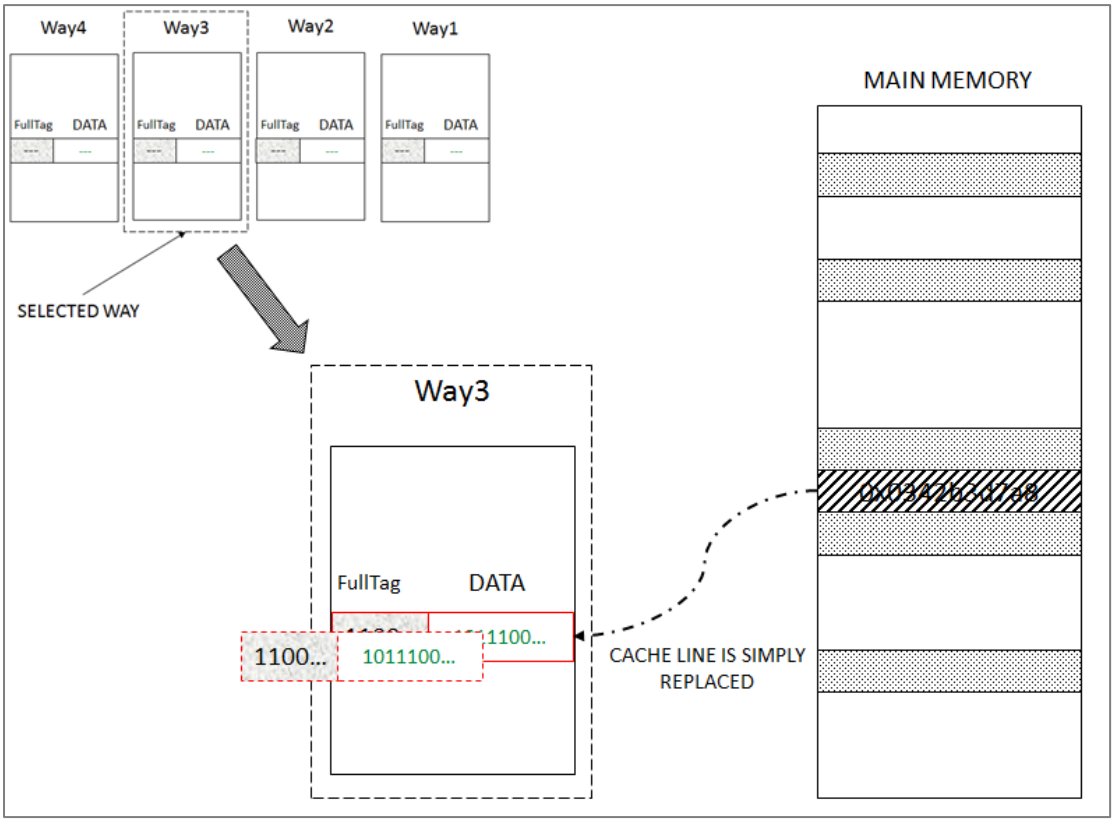


Figure 3.14 - Conventional Replacement Mechanism

As all the wordlines / cache-lines have their own set of DATA and associated full tag, a cache line is simply replaced from lower memory hierarchy.

b) Proposed Replacement Policies

In proposed architecture, conventional method is simply not applicable as each cache line doesn't possess its associated full tag; rather have a small portion TagL. Also as already discussed TagH entries in LoB are very crucial, as they are the only available complementary tags for all cache lines in a Way.

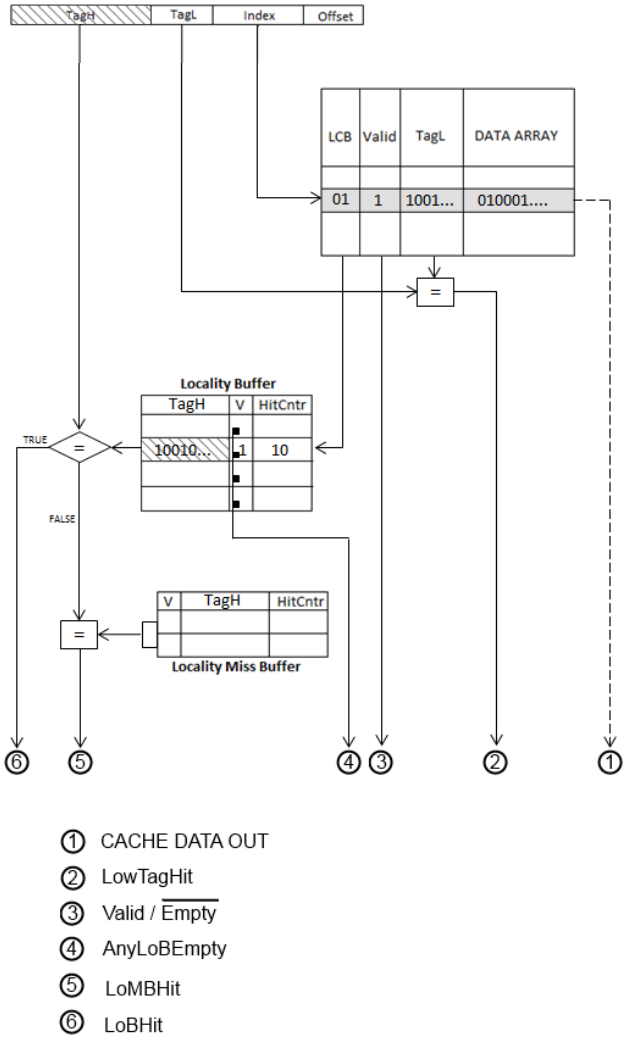


Figure 3.15 Replacement Policies [14]

The replacement policies applied in the proposed architecture are almost the same as proposed by Kwak and Jeon in their architecture.

Consider the figure 3.15, during handling a CACHE MISS a cache line is almost simply replaced but that doesn't happen as simply in LoB. Treating every LoB entry equally important, methods had been proposed by Kwak and Jeon analyze the LoB and then act.

For example, for a certain request, suppose TagL matches while TagH selected by the LCB doesn't match; In other words LowTagHit is a TRUE but LoBHit is FALSE for a certain request.

And let us suppose some other LoB entry (other than steered by LCB) matches with the requested TagH. In such a situation mere replacement of LoB would be disaster.

Therefore some modifications are to be applied on the TagH-side. As dictated by Kwak and Jeon, replacement policies in a selected Way are summarized as follows:

Table 3.3 Reduced Tag Operations [14]

Low Tag (TagL)	Locality Buffer (LoB)	Locality Miss Buffer LoMB	OPERATION
HIT	HIT	---	▪ No Action. CACHE HIT
	MISS	HIT	▪ Fetch from Lower Level Memory ▪ LoMB Hit Counter ↑ ▪ If(LoMB Hit Counter > Threshold) Swap LoB and LoMB Cache in L1 Cache Else Directly feed to CPU – No Caching
	MISS	MISS	▪ Fetch from Lower Level Memory ▪ If LoMB available, Insert. Else select a candidate LoMB, replace. ▪ If LoB Cold Miss insert into LoB Cache in L1 Cache ▪ Directly feed to CPU – No Caching
MISS	HIT	---	▪ (Conventional Operation) ▪ Fetch from Lower Level Memory, Cache in L1 Cache ▪ Adjust corresponding LCB ▪ LoB Hit Counter ↑
MISS	MISS	HIT	▪ Fetch from Lower Level Memory ▪ LoMB Hit Counter ↑ ▪ If(LoMB Hit Counter > Threshold) Swap LoB and LoMB Cache in L1 Cache Else Directly feed to CPU – No Caching
MISS	MISS	MISS	▪ Fetch from Lower Level Memory ▪ If LoMB available, Insert. Else select a candidate LoMB, replace. ▪ If LoB Cold Miss insert into LoB Cache in L1 Cache ▪ Directly feed to CPU – No Caching

NOTE: All actions when LoB is a HIT are exactly same as conventional actions, with LoB counter up.

Table 3.4 – Cache Operations Summary

LowTagHit	LoB	LoMB	LoMB.HitCounter >Threshold	LoBEmpty	LoMBEmpty	CacheHit	RAMAccess	LoB Modified	LoMB Modified	Tagh-Side Modifications	Cache Modified	Tagl-Side Modifications
1	1					1	0	0	0	NIL	0	NIL
0	1					0	1	0	0	LoB.HitCounter++	1	All from RAM & LCB=LCB_in (corresponding Current LoB)
x	0	1	1			0	1	1	1	Swap LoB with LoMB at same LCB, LoB.HitCounter=0	1	All from RAM, but LCB unchanged
x	0	1	0			0	0	0	1	LoMB.HitCounter++	0	NIL
x	0	0		1		0	1	1	0	New LoB=tagh at available LCB.	1	All from RAM & LCB=LCB_in (corresponding New LoB)
x	0	0		0	1	0	0	0	1	New LoMB=tagh at available row.	0	NIL
x	0	0		0	0	0	0	0	0*	* Select LoMB candidate & replace. (LoMB.HitCounter<=1)	0	NIL

CHAPTER 4

SIMULATION SETUP AND IMPLEMENTATION

4.1 DESIGN REQUIREMENTS

Developing a simulation model of cache with different configurations and analyzing the performance with varying design parameters requires a platform for which:

- Test benches can be easily written and modified.
- User friendly inputs and outputs can be handled.
- A controlled and detailed simulation log can be generated.
- Simulation model can be ported to different machines, with least dependencies.
- Run-time flexibility + debugging facilities available.
- Can be batch processed/recompiled/initiated/terminated/generated (for bulk simulation).

There are several platforms that have been used by many researchers for architectural modeling like SimpleScalar Toolset, CACTI, PowerStone Suite, Platune Simulation Platform, SystemC and also VHDL. Different simulators/environments are used for different analysis.

Further there have to be standard benchmark programs that can be used to test the simulation cache model, so that a comparative study can be made. There are several benchmark programs provided by several agencies. Some of them are SPEC CPU2000, PowerStone, MiBench, CacheBench etc.

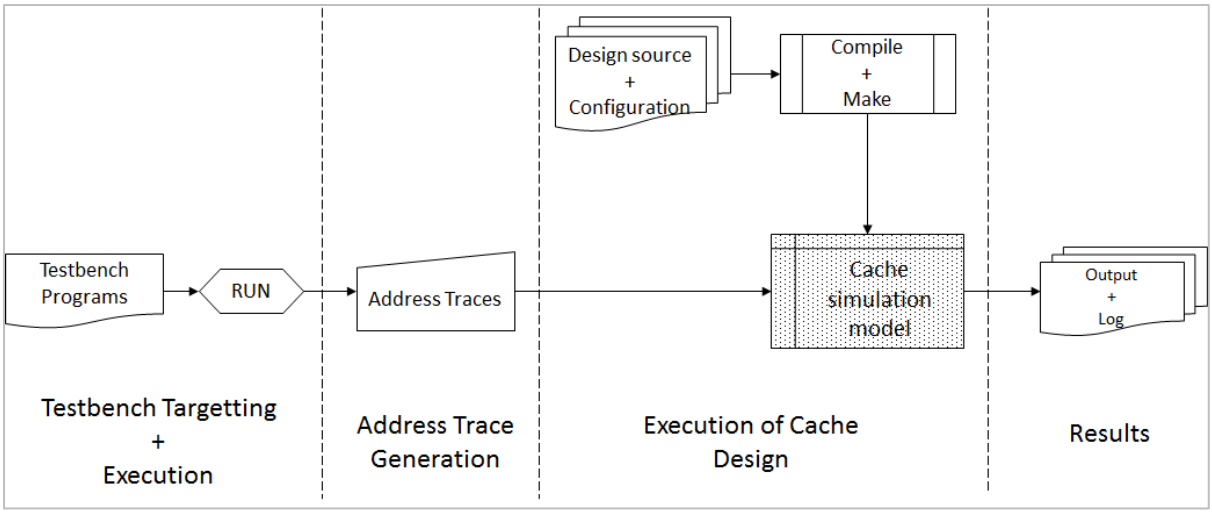


Figure 4.1 - Design Flow: General Distribution of Implementation

Still further, there are several address trace generation tools available depending upon the platforms and nature of operation. To name some – PIN tool from Intel, Simplescalar, Keil Debugger are a few. PIN tool generate the memory access traces of running programs on-the-fly. Simplescalar is a toolset which generate memory address traces of different programs for different targets.

As it is clear from the figure 4.1 the implementation is distributed in various sub-implementations. These can further be implemented separately in different tools available.

A brief detail of the tools used for different sections of implementations are as follows:

Table 4.1 - List of Tools Used

IMPLEMENTATION	TOOL USED	PLATFORM
TestBench Programs	MiBench Programs	Linux (Ubuntu 12.04) x86
Target for TestBench	ARM	---
Testbench Compiler	ARM/Simplescalar	Linux (Ubuntu 12.04) x86
Address Trace Generation	ARM/Simplescalar	Linux (Ubuntu 12.04) x86
Cache Design & Execution	SystemC v 2.2.0	Windows 7 Ultimate x64
Power Consumption Analysis	CACTI v 4.0	Windows 7 Ultimate x64

4.2 DISCUSSION ON TOOLS USED:

4.3 MiBench

It is commercially representative embedded benchmark suite. It is freely available to researchers. It is a set of 35 embedded applications for benchmarking purposes. These benchmarks are divided into six suites with each suite targeting a specific area of the embedded market. The six categories are Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. All the programs are available as standard C source code. Since many past embedded applications have been written directly in assembly language, it has been difficult to collect a portable set of benchmarks for the embedded domain [19].

Each benchmark comes with small and large data sets. The small data set represents a light-weight, useful embedded application of the benchmark, while the large data set provides a more stressful, real-world application.

Some of the used benchmark applications are described as below:

4.3.1 Automotive and Industrial Control

- *basicmath*: The basic math test performs simple mathematical calculations that often don't have dedicated hardware support in embedded processors. For example, cubic function solving, integer square root and angle conversions from degrees to radians are all necessary calculations for calculating road speed or other vector values. The input data is a fixed set of constants.
- *bitcount*: The bit count algorithm tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers. It does this using five methods including an optimized 1-bit per loop counter, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, nonrecursive bit count by bytes using a table look-up and shift and count bits. The input data is an array of integers with equal numbers of 1's and 0's.
- *qsort*: The qsort test sorts a large array of strings into ascending order using the well-known quick sort algorithm. Sorting of information is important for systems so that priorities can be made, output can be better interpreted, data can be organized and the overall run-time of programs reduced. The small data set is a list of words; the large data set is a set of three-tuples representing points of data.

- *susan*: Susan is an image recognition package. It was developed for recognizing corners and edges in Magnetic Resonance Images of the brain. It is typical of a real world program that would be employed for a vision based quality assurance application. It can smooth an image and has adjustments for threshold, brightness, and spatial control. The small input data is a black and white image of a rectangle while the large input data is a complex picture.

4.3.2 Network

- *dijkstra*: The Dijkstra benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm. Dijkstra's algorithm is a well-known solution to the shortest path problem and completes in O(n2) time.
- *patricia*: A Patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the trie to reduce traversal time at the expense of code complexity. Often, Patricia tries are used to represent routing tables in network applications. The input data for this benchmark is a list of IP traffic from a highly active web server for a 2 hour period. The IP numbers are disguised.

4.3.3 Security

- *rijndael encrypt/decrypt*: Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks. The input data sets are the same as the ones used by blowfish.

4.3.4 Consumer Devices

- *jpeg encode/decode*: JPEG is a standard, lossy compression image format. It is included in MiBench because it is a representative algorithm for image compression and decompression and is commonly used to view images embedded in documents. The input data are a large and small color image.
- *tiffmedian*: Tiffmedian converts an image to a reduced color palette by taking several medians of the current color palette.
- *mad*: Mad is a high-quality MPEG audio decoder. It currently supports MPEG-1 and the MPEG-2 extension to Lower Sampling Frequencies, as well as the socalled MPEG 2.5 format. All three audio layers (Layer I, Layer II, and Layer III a.k.a. MP3) are fully implemented. It uses small and large MP3s for its data inputs.

4.3.5 Office Automation

- *stringsearch*: This benchmark searches for given words in phrases using a case insensitive comparison algorithm.

4.3.6 Telecommunications

- *FFT*: This benchmark performs a Fast Fourier Transform on an array of data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal. The input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components.
- *GSM encode/decode*: The Global Standard for Mobile (GSM) communications [18] is the standard for voice encoding/decoding in Europe and many countries. It uses a combination of Time- and Frequency-Division Multiple Access (TDMA/FDMA) to encode/decode data streams. The input data is small and large speech samples.
- *CRC32*: This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission. The data input is the sound files from the ADPCM benchmark.

4.4 Simplescalar Tool-Set

The SimpleScalar tool set (release 2.0) performs fast, flexible, and accurate simulation of modern processors that implement the SimpleScalar architecture (a close derivative of the MIPS architecture). The tool set takes binaries compiled for the SimpleScalar architecture and simulates their execution on one of several provided processor simulators [15].

The advantages of the SimpleScalar tools are high flexibility, portability, extensibility, and performance. It includes five execution-driven processor simulators in the release. They range from an extremely fast functional simulator to a detailed, out-of-order issue. The simulators have been aggressively tuned for performance, and can run codes approaching "real" sizes in tractable amounts of time. On a 200-MHz Pentium Pro, the fastest, least detailed simulator simulates about four million machine cycles per second, whereas the most detailed processor simulator simulates about 150,000 per second.

4.4.1 Simplescalar Flow

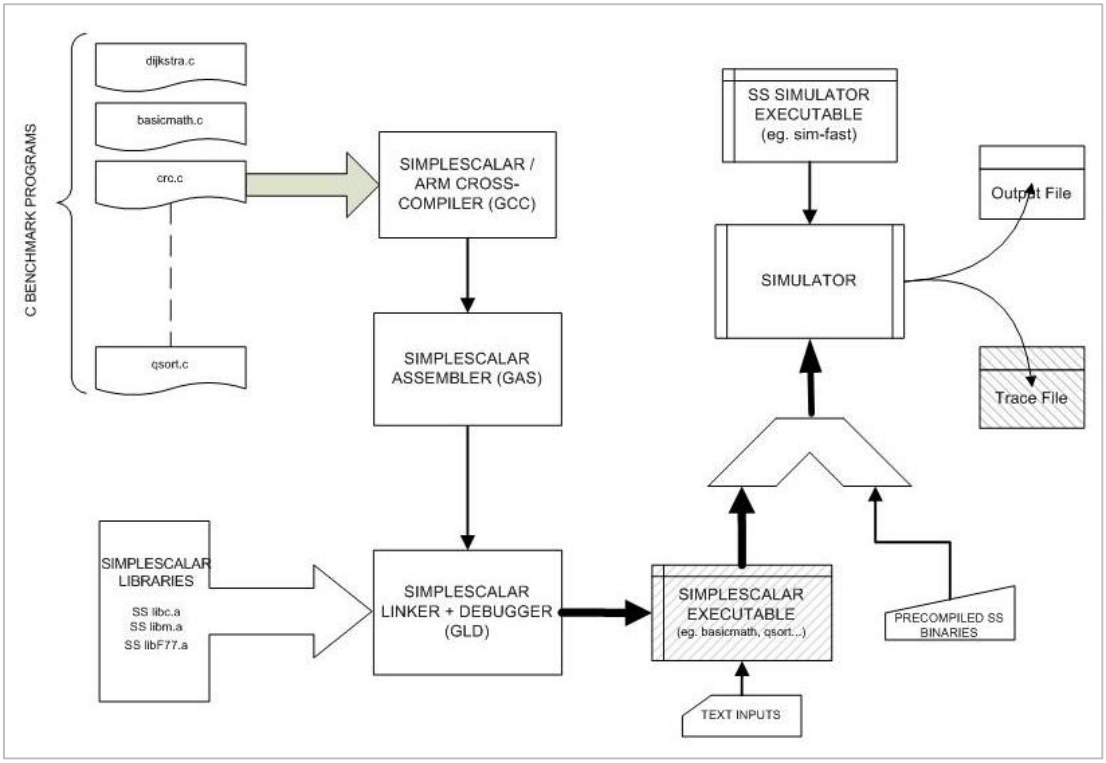


Figure 4.2 - SimpleScalar Toolset Overview

MiBench benchmarks written in C are compiled using the SimpleScalar version of GCC, which generates SimpleScalar assembly. The SimpleScalar assembler and loader, along with the necessary ported libraries, produce SimpleScalar executables that can then be fed directly into one of the provided simulators. (The simulators themselves are compiled with the host platform's native compiler; any ANSI C compiler will do).

An illustration of the execution of MiBench program for ARM target and generating address trace all using simplescalar is as follows:

i. Compile the MiBench C program using Simplescalar/ARM cross-compiler and generate Simplescalar binary file; executable for ARM

Syntax: `$ arm-linux-gcc <source.c> -o <binary.o>`

Command: `$ arm-linux-gcc patricia.c -o patricia.arm`

```
where
qsort.c –          MiBench C application
arm-linux-gcc –    is the GNU Cross-Compiler for ARM
```

```
qsort.arm –        executable binary.
```

ii. Run the ARM executable using Simplescalar simulators (eg. sim-safe)

Syntax:

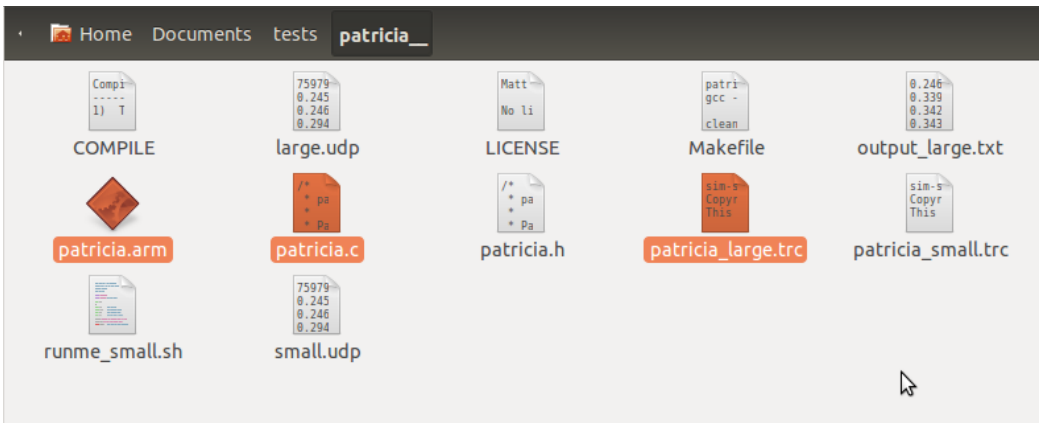
```
$ sim-safe <simulator args.> <executable> <executable args.>
```

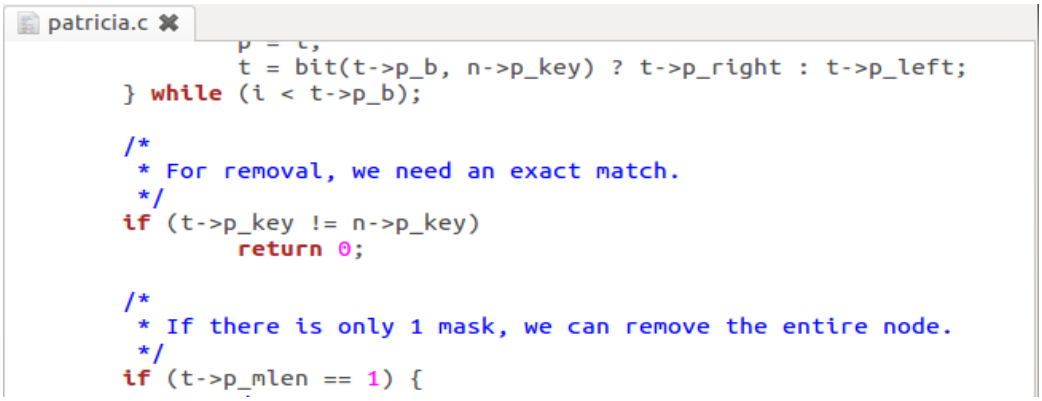
Command:

```
$ sim-safe -redir:sim patricia_large.trc -v -max:inst 100000
patricia.arm large.udp
```

```
where
sim-safe:          Simplescalar Functional Simulator
-redir:sim <file>  Redirects simulation results into <file>
-v                 Verbose (detailed logging)
-max:inst <N>      Run maximum of N instruction of executable
patricia.arm       Simplescalar/ARM Executable
large.udp          (Filename) Argument to patricia.arm
```

Illustrative screenshots are as follows:





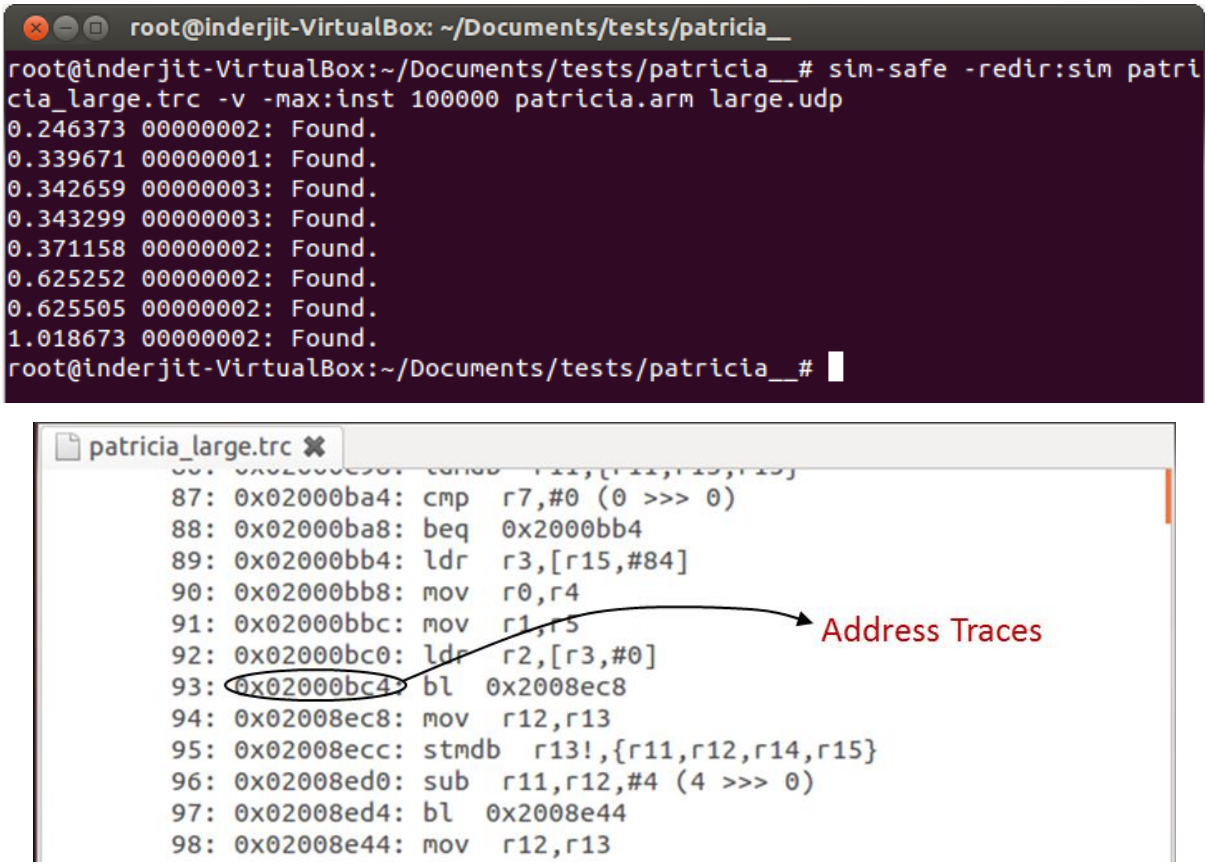


Figure 4.3 - Illustrative Screenshots: Simplescalar Address Trace Generation

4.5 SYSTEMC

4.5.1 Motivation

The motivation starts with the changing nature of "systems" under design. To specify, design, and implement complex systems, incorporating functionality implemented in both hardware and software forms, one is compelled to move on from the HDLs of old. One must also move beyond the RT level of abstraction used with these HDLs. One needs to move to what has been termed the "system-level" of design. And one needs a modeling language that can support this level [21].

4.5.2 Requirements for a Language to Model Systems

- Specification and design at various levels of abstraction
- Incorporation of embedded software portions of a complex system, both models and implementation-level code;
- Creation of executable specifications of design intent;

- Creation of executable platform models, representing possible implementation architectures on which the design intent will be mapped;
- Fast simulation speed to enable design-space exploration, of both functional specification and architectural implementation alternatives; and
- Constructs allowing the separation of system function from system communications, in order to allow flexible adaptation and reuse of both models and implementations.

4.5.3 SystemC

Therefore SystemC was developed to enable system-level modeling; that is, modeling of systems above the register-transfer level of abstraction, including systems that might be implemented in software, hardware, or some combination of the two. Of course RTL modeling can also be performed in SystemC, and, in addition, one can develop models above the RT level and refine them down to RTL within a single language and environment. SystemC provides a common language for software and hardware, C++.

4.5.4 Language Comparison

SystemC is not a language, but rather a class library for a well-established language, C++. Several languages have emerged to address the various aspects of system design. Although Ada and Java have proven their value, C/C++ is predominately used today for embedded system software. The hardware description languages (HDLs), VHDL and Verilog, are used for simulating and synthesizing digital circuits. Vera and e are the languages of choice for functional verification of complex application-specific integrated circuits (ASICs). SystemVerilog is a new language that evolves the Verilog language to address many hardware-oriented system design issues.

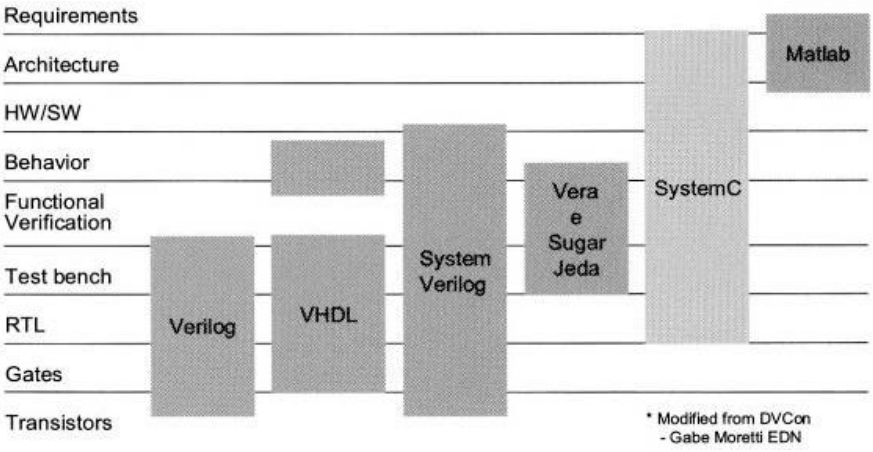


Figure 4.4 - Language comparison for SystemC [21]

4.5.5 SystemC Language Architecture

A large system may be composed of communicating subsystems that are modeled heterogeneously, and such a system needs to be simulated efficiently. To address this challenge, SystemC uses a layered approach that allows for the flexibility of introducing new, higher-level constructs that share an efficient simulation engine.

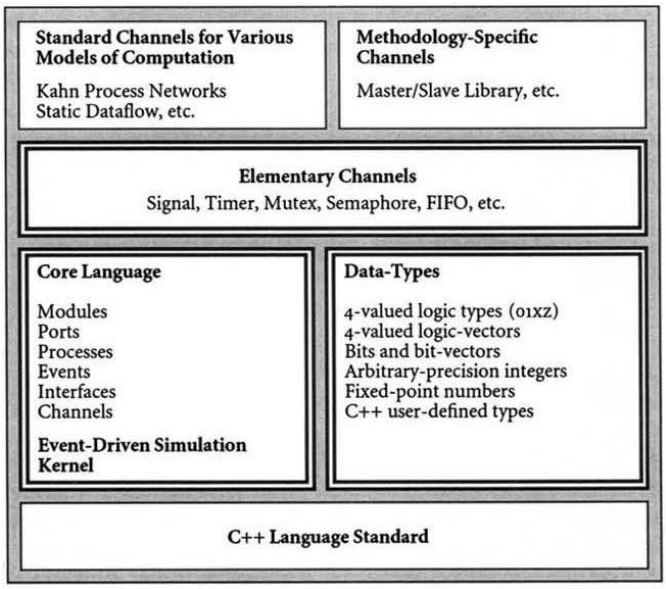


Figure 4.5 - SystemC Language Architecture [21]

The bottommost layer highlights the fact that SystemC is built entirely on standard C++. This means that any program written in SystemC may be compiled with a C++ compiler to produce an executable program. Alongside the core language is a set of data-types that are useful for hardware modeling and for certain kinds of software programming (e.g., fixed-point types for DSP software, and for hardware implementations of DSP-type functions).

4.5.6 SystemC Components

SystemC implements the structures necessary for hardware modeling by providing constructs that enable concepts of time, hardware data types, hierarchy and structure, communications, and concurrency. Let's look at what SystemC offers for hardware modeling in brief.

- Modules and Hierarchy
- Hardware Data Types
- Methods and Threads
- Events, Sensitivity
- Interfaces and Channels

Modules and Hierarchy

Modules are building blocks of SystemC designs; they are like modules in Verilog, class in C++. Modules can instantiated in another Module, Each of the lower level modules instance can be referred through the complete hierarchy. This is same as in Verilog or C++.

Example:

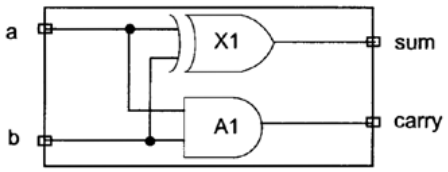


Figure 4.6 - A module with 4 ports

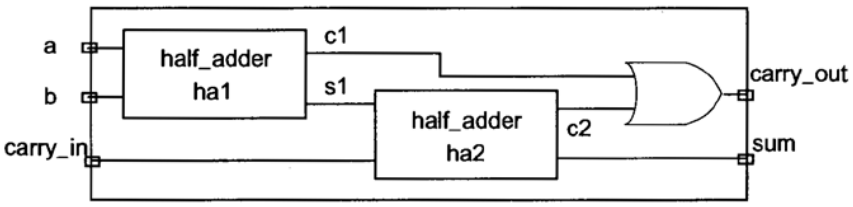


Figure 4.7 - Module with sub-modules [Hierarchy]

Methods and Threads

SC_METHOD and SC_THREADS are backbone for modeling hardware. SC_METHODS are like functions in Verilog, which does not consume time when they execute, whereas SC_THREADS consume time, like waiting for event (like positive-edge of clock).

Events, Sensitivity

Events and Sensitivity give SystemC power to emulate hardware (concurrency). Events are implemented by the SystemC sc_event class. Events are caused or triggered through the sc_event member function.

SystemC has two types of sensitivity: static and dynamic. Static sensitivity is implemented by applying the SystemC sensitive command to an SC_METHOD, SC_THREAD, or SC_CTHREAD at elaboration time (within the constructor). Dynamic sensitivity lets a simulation process change its sensitivity on the fly. The SC_METHOD implements dynamic sensitivity with a next_trigger command. The SC_THREAD implements dynamic sensitivity with a next command.

Interfaces and Channels

In real hardware pins (ports) are used for communicating with external world. In SystemC, modules are interconnected using either primitive channels or hierarchical channels. Both types of channels connect to modules via ports.

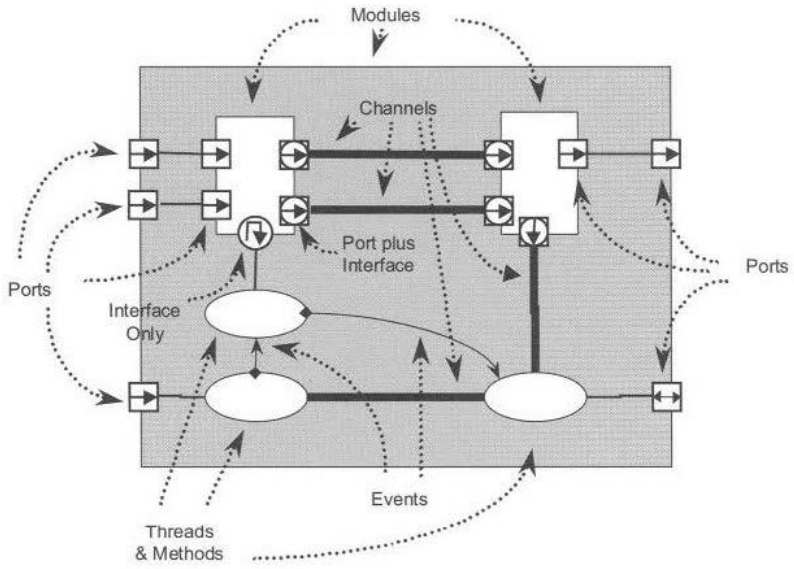


Figure 4.8 - SystemC Components [21]

4.5.7 SystemC Cache Design Flow

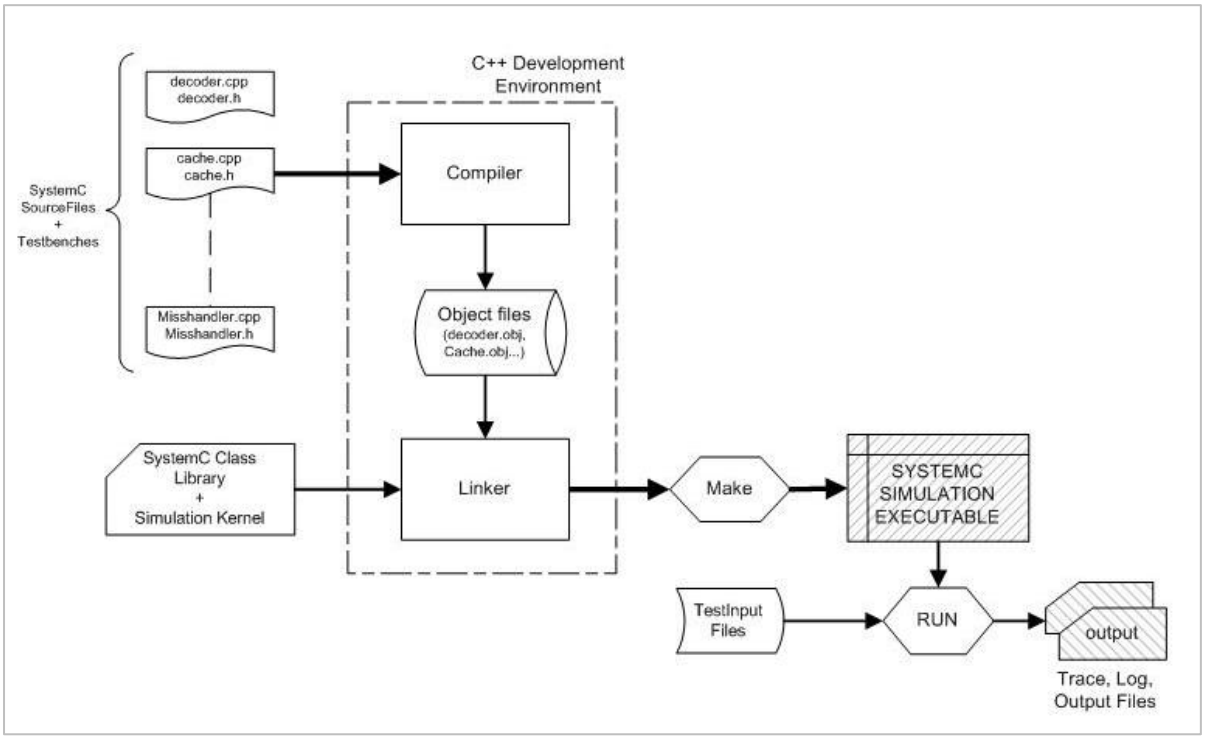


Figure 4.9 - SystemC Design Flow

As clear from the figure 4.9, cache model is developed in SystemC. Complete system is modeled by developing separate sub-modules like decoder, cache memory, cache controller, memory hierarchy sub-system and Testbenches etc.

SystemC allows modules of different abstraction levels to be simulated together; this is called *Hardware/Software Co-Simulation* [21]. All the sub-module plus testbench source code are compiled and build together to generate a SystemC simulation executable.

The simulation executable includes the functionality of the designed system and testbenches for the system as one complete package. This is a simple win32 console application (.exe) that can run on any machine with windows operating system. This is the major advantage of SystemC development environment.

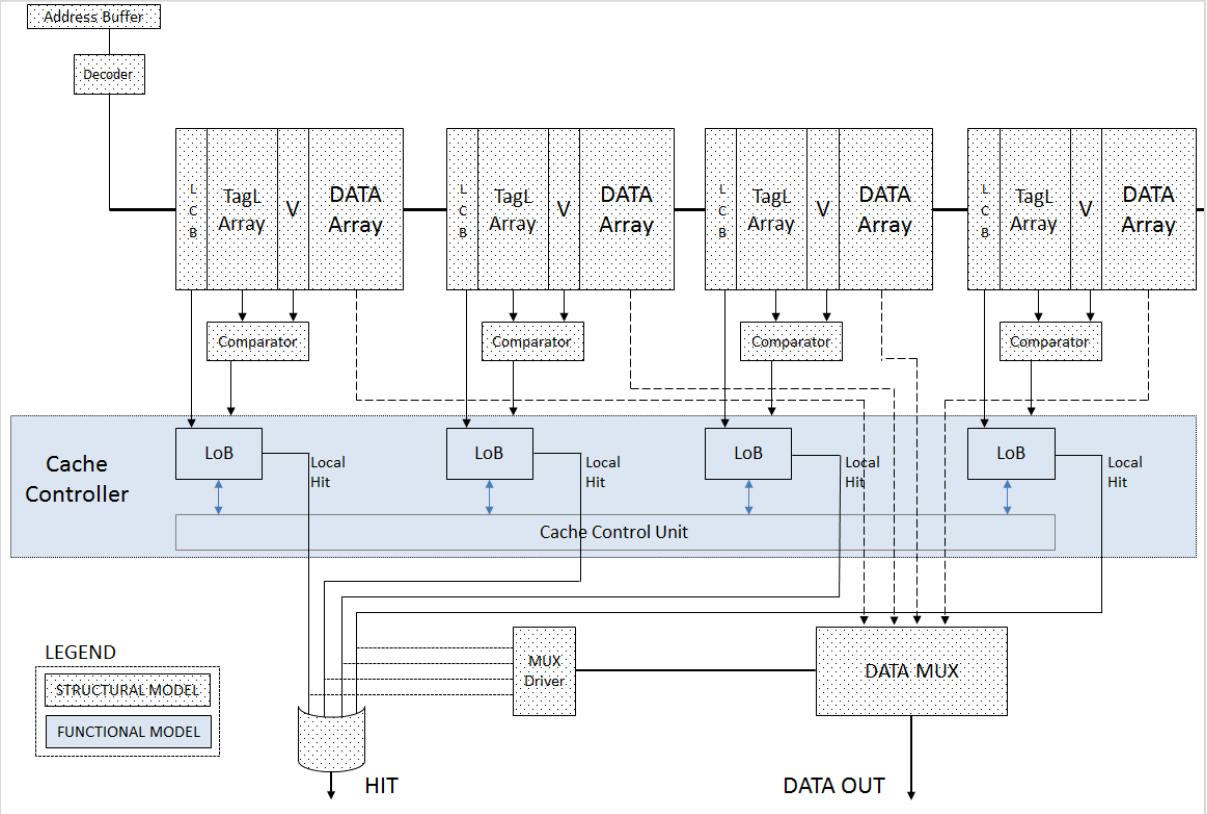


Figure 4.10 - SystemC Cache Model: HW/SW Co-Simulation

SystemC development environment also gives you full featured access to file-system to read and write files natively in C++/SystemC. Therefore the complete system designed is a TestBench–DUT system, which is a mix of structurally and functionally modeled modules. The Device Under Test (DUT) here is the Cache Memory, and rest of the modules are the part of TestBench as shown in the figure–4.11.

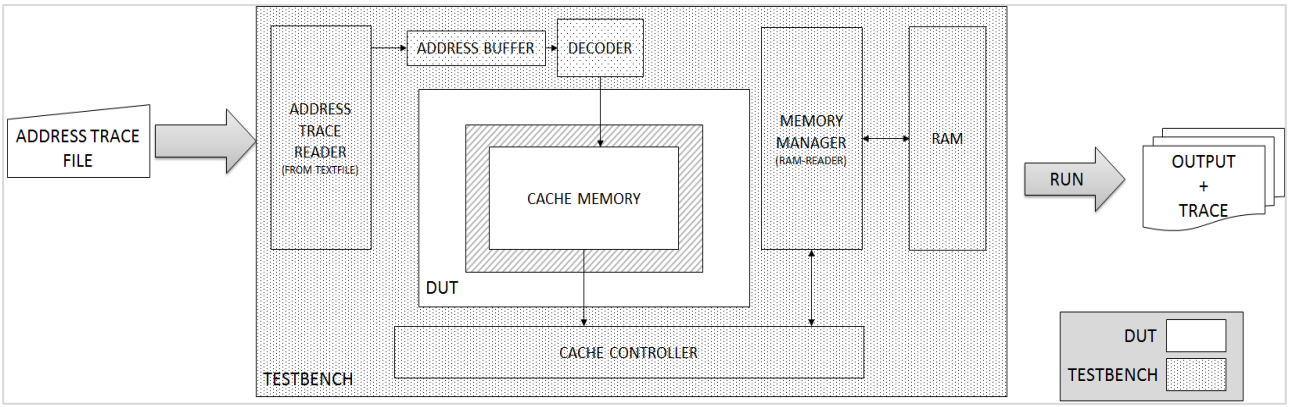


Figure 4.11 - SystemC Simulation EXE: DUT–TestBench Partition

4.5.8 Illustrative Screen Shots

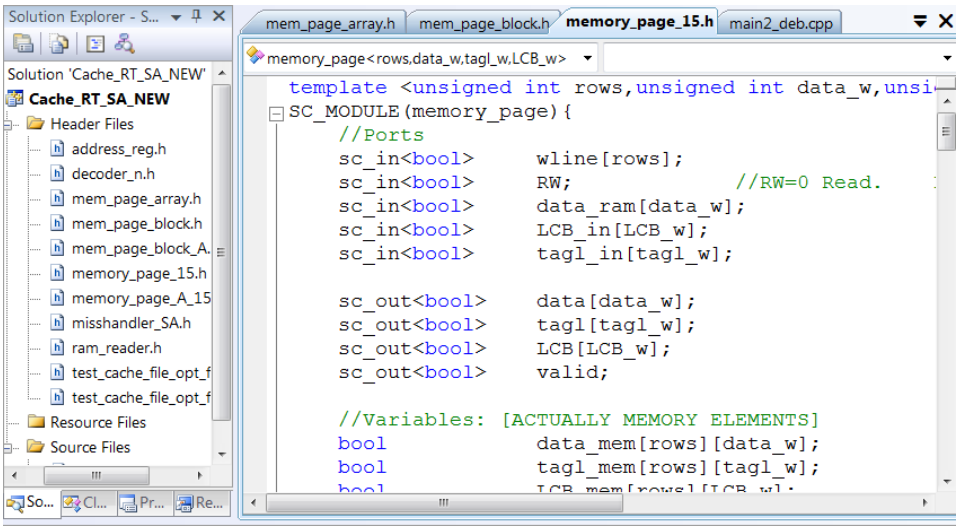


Figure 4.12 - SystemC Cache Design Source Files Snapshot

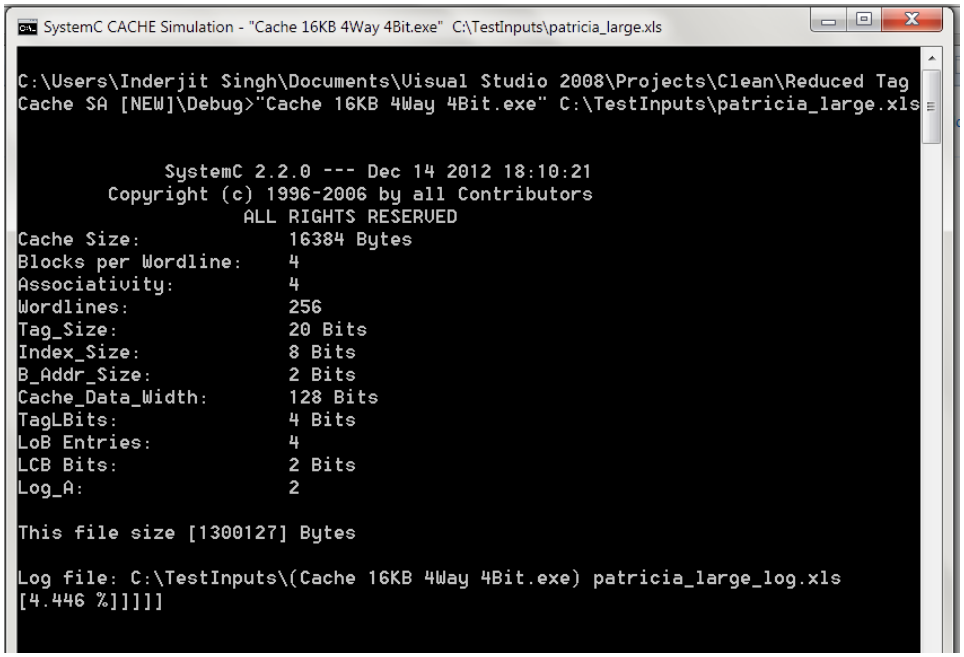


Figure 4.13 - SystemC Simulation EXE snapshot

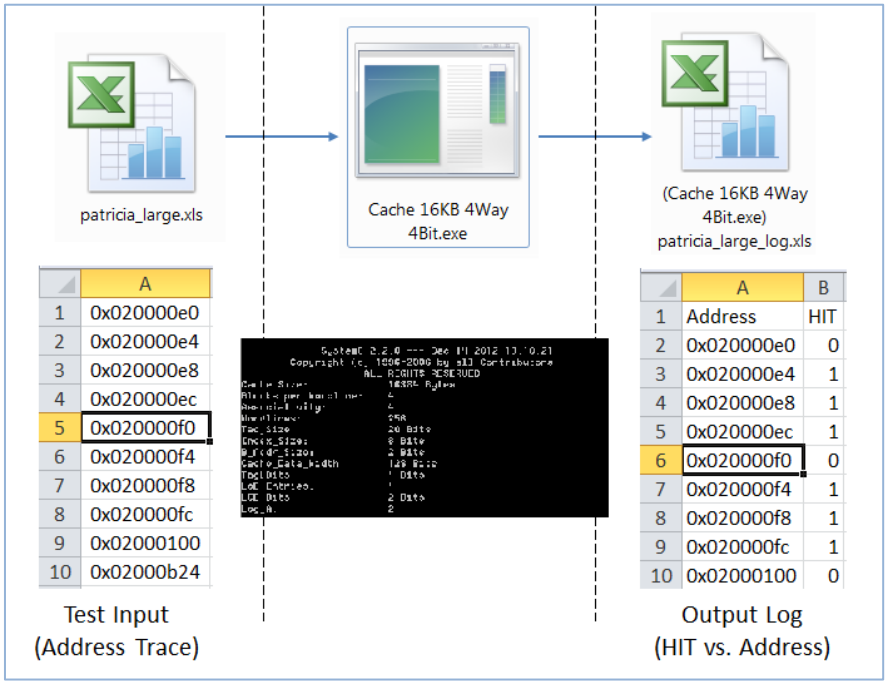


Figure 4.14 - Implemented Cache Simulation Model Execution Flow Snapshot

4.6 IMPLEMENTATION SUMMARY

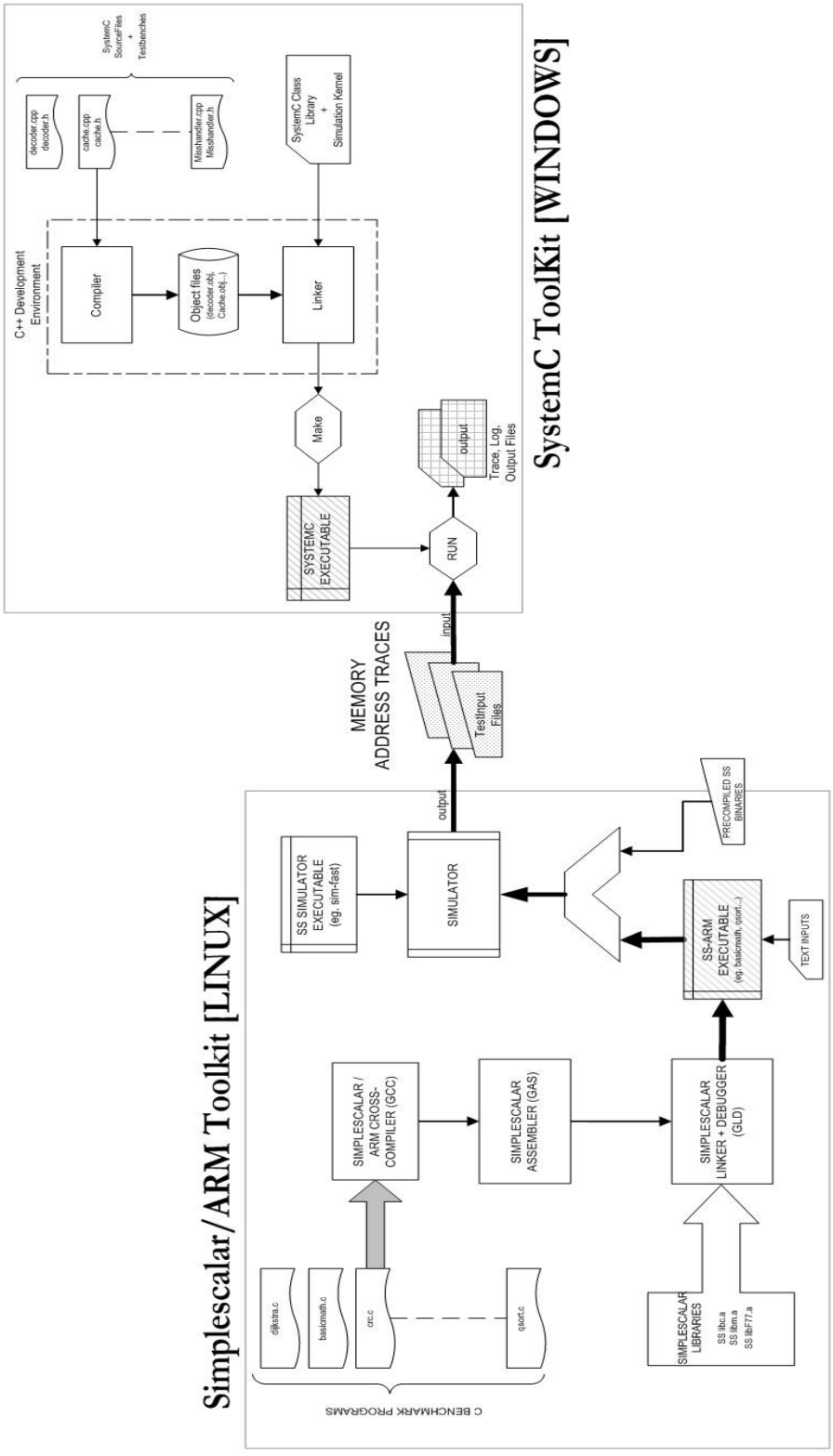


Figure 4.15 Combined Implementation Flow

As discussed in detail in preceding sections, a flexible, portable, synthesizable model of Cache Memory is built in SystemC.

CHAPTER 5

SIMULATION RESULTS

A very fundamental observation which inspired this design is as follows:

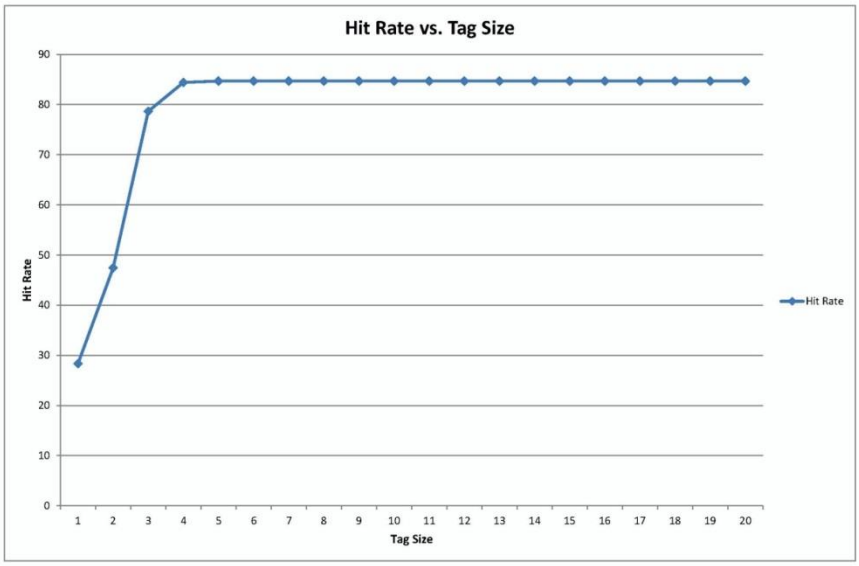


Figure 5.1 - Typical Hit Rate vs. Tag Size

It follows from figure 5.1 that using Reduced-Tag cache, not all the tag-bits are necessary to achieve the regular performance.

5.1 Measure of Optimum Tag-Length

Therefore in the first set of simulations, a detailed hit ratio variation (%) that depends on the number of tag bits used is evaluated. Table 5.1 summaries the results in a 16 KB 4-Way Set-Associative cache memory. In Table 5.1, *fulltag* denotes a conventional full tag cache mechanism and the number of partial tag bits varies from 0 to 8. The bold-faced numbers in Table 3 indicate the points at which the actual miss ratios of the partial tag + MASK FIFO policy become the same as those of the conventional full tag policy. As shown in the simulation, 1–6 tag bits are enough to provide a comparable hit ratio with the full tag policy. On average, 6 tag bits are enough to obtain the same level as the full tag policy.

Table 5.1: Number of TagL bits vs. Hit Ratio [16KB 4Way]

	1	2	3	4	5	6	7	8	20 FullTag
basicmath	25.54	57.09	82.57	97.06	97.75	98.09	**98.10**	98.10	**98.10**
bitcount	98.34	99.33	99.38	99.39	**99.40**	99.40	99.40	99.40	**99.40**
blowfish	99.52	**99.53**	99.53	99.53	99.53	99.53	99.53	99.53	**99.53**
crc	83.03	**99.50**	99.50	99.50	99.50	99.50	99.50	99.50	**99.50**
dijkstra	61.68	86.53	99.24	99.15	**99.28**	99.28	99.28	99.28	**99.28**
fft	69.81	**99.42**	99.42	99.42	99.42	99.42	99.42	99.42	**99.42**
gsm	69.52	90.26	92.75	93.00	**93.02**	93.02	93.02	93.02	**93.02**
jpeg	41.38	47.78	98.34	98.46	**98.42**	98.42	98.42	98.42	**98.42**
mad	89.37	98.90	98.96	99.25	99.25	**99.26**	99.26	99.26	**99.26**
patricia	42.88	62.80	78.44	94.20	93.72	93.77	**93.79**	93.79	**93.79**
pgp	9.33	9.35	50.57	99.03	**99.07**	99.07	99.07	99.07	**99.07**
qsort	50.37	41.22	65.65	98.68	**99.24**	99.24	99.24	99.24	**99.24**
rijndael	74.70	93.31	93.89	**93.93**	93.93	93.93	93.93	93.93	**93.93**
stringsearch	63.00	63.25	98.74	**99.34**	99.34	99.34	99.34	99.34	**99.34**
susan	35.20	96.00	96.81	97.22	97.12	**97.42**	97.42	97.42	**97.42**
tiffmedian	59.35	61.51	98.34	**98.84**	98.84	98.83	98.83	98.83	**98.83**
Average	60.8	75.4	90.8	97.9	97.9	**98**	98	98	**97.97**

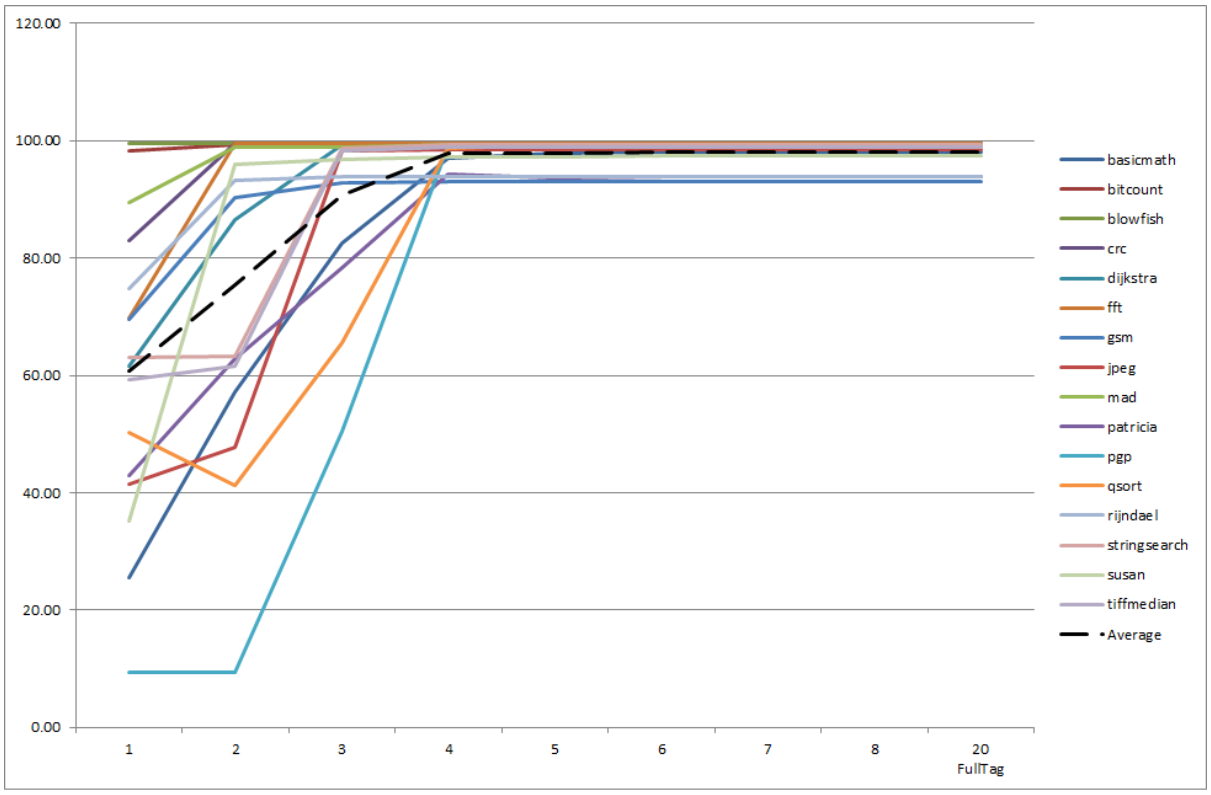


Figure 5.2: Number of TagL bits vs. Hit Ratio [16KB 4Way]

Table 5.2: Number of TagL bits vs. Hit Ratio [16KB 1Way]

	1	2	3	4	5	6	7	8	18 FullTag
basicmath	95.37	95.78	**95.81**	95.81	95.81	95.81	95.81	95.81	**95.81**
bitcount	98.40	99.36	**99.37**	99.37	99.37	99.37	99.37	99.37	**99.37**
blowfish	99.51	**99.52**	99.52	99.52	99.52	99.52	99.52	99.52	**99.52**
crc	99.47	**99.48**	99.49	99.49	99.49	99.49	99.49	99.49	**99.48**
dijkstra	97.24	97.38	**97.39**	97.39	97.39	97.39	97.39	97.39	**97.39**
fft	99.36	**99.40**	99.40	99.40	99.40	99.40	99.40	99.40	**99.40**
gsm	92.81	92.71	**92.75**	92.75	92.75	92.75	92.75	92.75	**92.74**
jpeg	98.31	98.31	**98.32**	98.32	98.32	98.32	98.32	98.32	**98.32**
mad	98.39	98.62	**98.68**	98.68	98.68	98.68	98.68	98.68	**98.68**
patricia	85.68	91.54	**91.84**	91.84	91.84	91.84	91.84	91.84	**91.84**
pgp	50.53	64.35	95.07	**98.97**	98.97	98.97	98.97	98.97	**98.97**
qsort	97.42	97.88	**98.08**	98.08	98.08	98.08	98.08	98.08	**98.08**
rijndael	93.28	**93.60**	93.58	93.58	93.58	93.58	93.58	93.58	**93.57**
stringsearch	98.16	**98.23**	98.23	98.23	98.23	98.23	98.23	98.23	**98.23**
susan	94.40	**96.74**	96.74	96.74	96.74	96.74	96.74	96.74	**96.74**
tiffmedian	98.26	98.73	**98.74**	98.74	98.74	98.74	98.74	98.74	**98.74**
Average	93.5	95.1	97.1	**97.3**	97.3	97.3	97.3	97.3	**97.3**

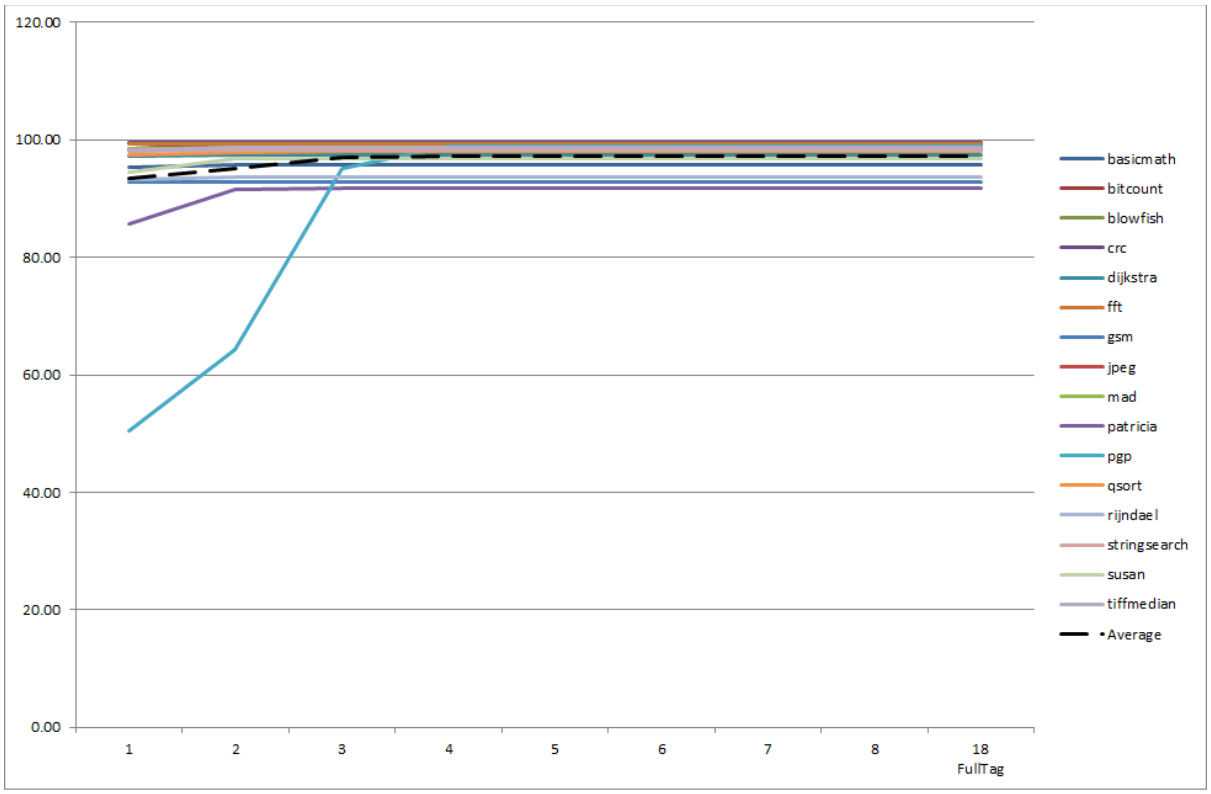


Figure 5.3: Number of TagL bits vs. Hit Ratio [16KB 1Way]

Table 5.3: Number of TagL bits vs. Hit Ratio [16KB 2Way]

	1	2	3	4	5	6	7	8	19 FullTag
basicmath	56.88	81.45	95.31	96.78	96.95	**96.96**	96.96	96.96	**96.96**
bitcount	97.80	97.85	**99.40**	99.40	99.40	99.40	99.40	99.40	**99.40**
blowfish	98.89	99.51	**99.53**	99.53	99.53	99.53	99.53	99.53	**99.53**
crc	98.95	98.96	**99.50**	99.50	99.50	99.50	99.50	99.50	**99.50**
dijkstra	31.66	73.16	99.01	**99.27**	99.27	99.27	99.27	99.27	**99.27**
fft	98.14	98.13	99.41	**99.42**	99.42	99.42	99.42	99.42	**99.42**
gsm	84.50	92.81	**92.97**	92.97	92.97	92.97	92.97	92.97	**92.97**
jpeg	47.74	98.36	98.39	98.38	**98.40**	98.40	98.40	98.40	**98.40**
mad	98.72	98.97	**99.25**	99.24	99.25	99.25	99.25	99.25	**99.25**
patricia	46.99	64.79	82.68	**93.04**	92.89	92.89	92.89	92.89	**92.89**
pgp	9.32	50.56	64.38	**99.06**	99.05	99.06	99.06	99.06	**99.06**
qsort	39.35	50.12	84.28	**99.19**	99.19	99.19	99.19	99.19	**99.19**
rijndael	88.75	89.23	**93.91**	93.91	93.91	93.91	93.91	93.91	**93.90**
stringsearch	63.06	63.29	**98.92**	98.92	98.92	98.92	98.92	98.92	**98.92**
susan	70.29	93.59	96.59	96.87	**97.34**	97.34	97.34	97.34	**97.34**
tiffmedian	59.48	96.25	98.73	**98.79**	98.79	98.81	98.81	98.81	**98.81**
Average	68.2	84.2	93.9	**97.8**	97.8	97.8	97.8	97.8	**97.8**

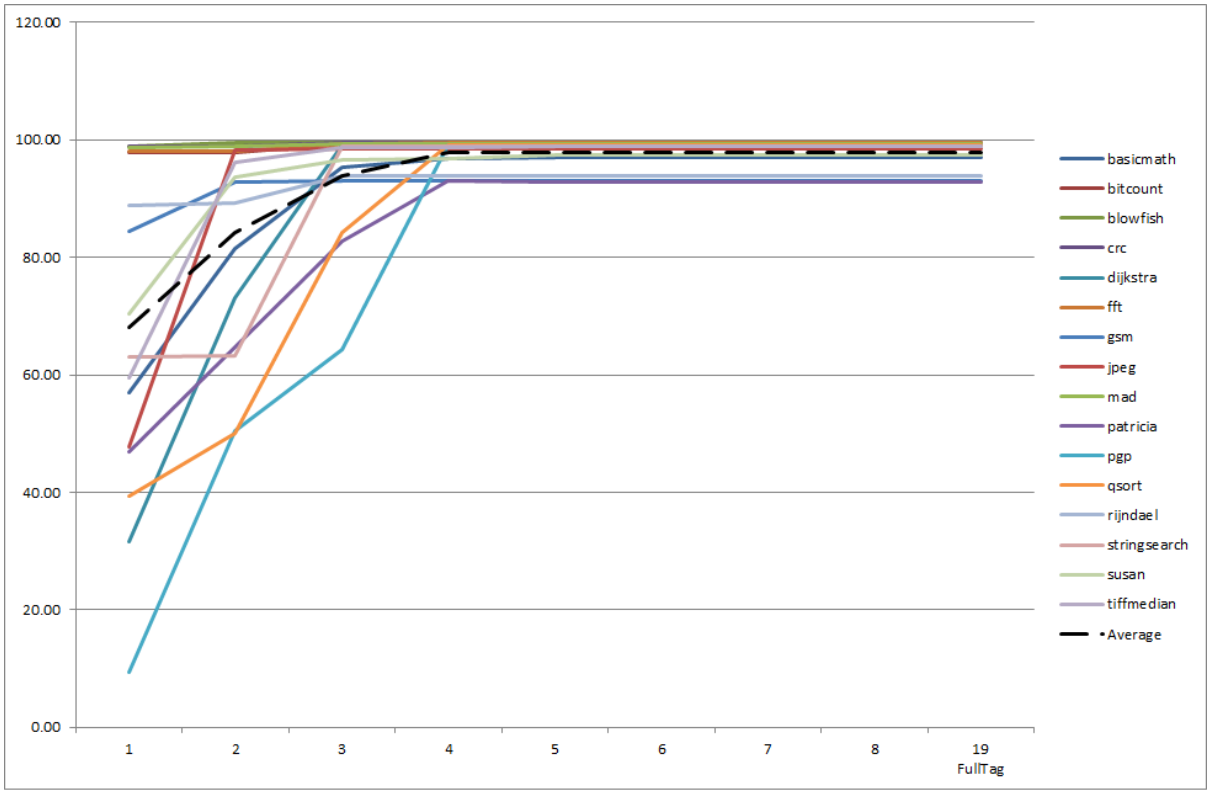


Figure 5.4: Number of TagL bits vs. Hit Ratio [16KB 2Way]

Table 5.4: Number of TagL bits vs. Hit Ratio [16KB 8Way]

	1	2	3	4	5	6	7	8	21 FullTag
basicmath	32.62	63.41	82.58	82.87	97.73	98.16	**98.25**	98.25	**98.25**
bitcount	99.24	99.36	99.39	99.39	**99.40**	99.40	99.40	99.40	**99.40**
blowfish	99.52	**99.53**	99.53	99.53	99.53	99.53	99.53	99.53	**99.53**
crc	99.46	99.48	**99.50**	99.50	99.50	99.50	99.50	99.50	**99.50**
dijkstra	49.22	83.82	98.84	**99.28**	99.28	99.28	99.28	99.28	**99.28**
fft	99.33	99.38	**99.42**	99.42	99.42	99.42	99.42	99.42	**99.42**
gsm	67.18	69.76	90.34	92.88	**93.03**	93.01	93.01	93.01	**93.01**
jpeg	51.34	58.67	70.98	**98.45**	98.46	98.43	98.43	98.43	**98.43**
mad	89.36	53.68	99.15	**99.25**	99.25	99.25	99.25	99.25	**99.25**
patricia	52.44	58.42	82.33	91.26	**94.38**	94.39	94.00	94.03	**94.03**
pgp	9.28	9.40	10.07	85.33	99.04	**99.08**	99.08	99.08	**99.08**
qsort	56.17	53.01	98.24	99.08	98.54	**99.24**	99.24	99.24	**99.24**
rijndael	92.77	93.26	93.45	93.88	**93.94**	93.94	93.94	93.94	**93.93**
stringsearch	63.15	88.37	88.33	98.98	**99.34**	99.34	99.34	99.34	**99.34**
susan	35.14	35.16	96.73	97.18	**97.42**	97.42	97.42	97.42	**97.42**
tiffmedian	58.92	59.89	98.65	98.70	98.85	**98.87**	98.86	98.86	**98.86**
Average	65.9	70.3	88	95.9	97.9	**98**	98	98	**98**

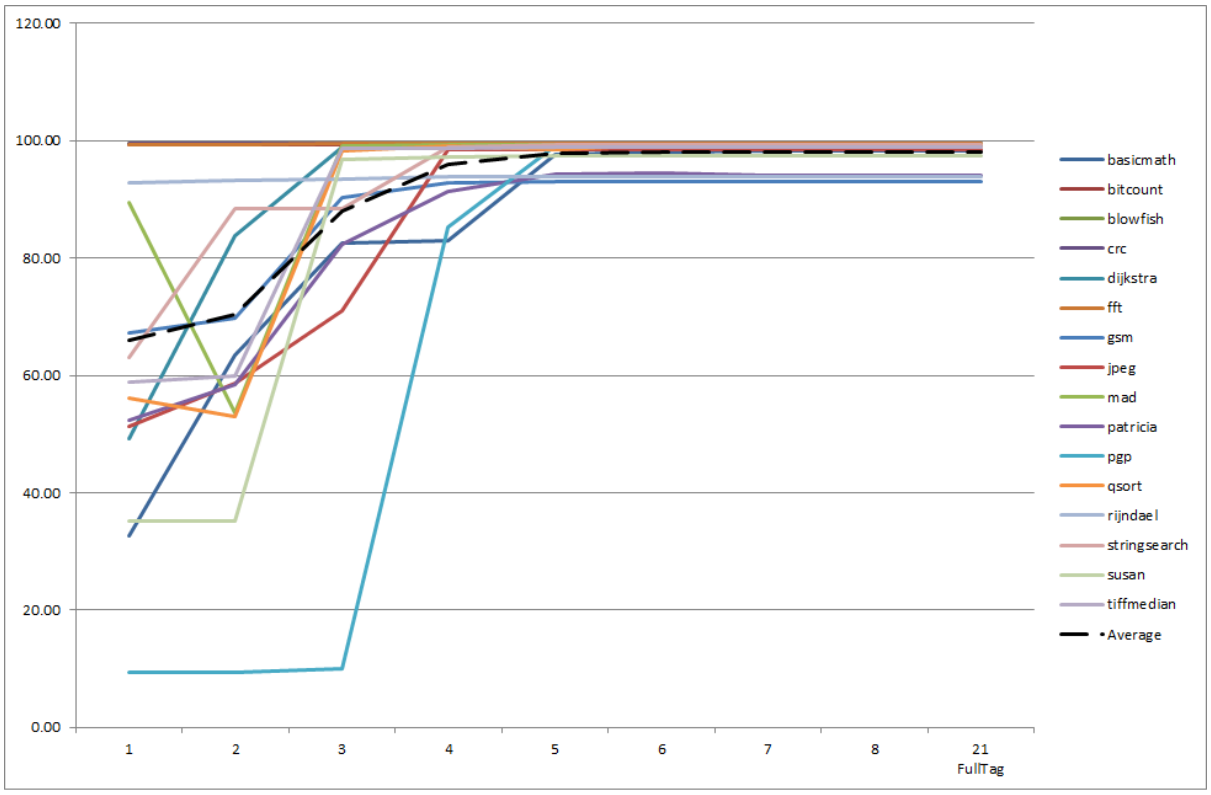


Figure 5.5: Number of TagL bits vs. Hit Ratio [16KB 8Way]

Table 5.5: Number of TagL bits vs. Hit Ratio [8KB 1Way]

	1	2	3	4	5	6	7	8	19 FullTag
basicmath	46.55	92.26	92.42	**92.44**	92.44	92.44	92.44	92.44	**92.44**
bitcount	97.80	98.39	99.35	**99.36**	99.36	99.36	99.36	99.36	**99.36**
blowfish	99.42	**99.51**	99.52	99.52	99.52	99.52	99.52	99.52	**99.52**
crc	99.42	**99.47**	99.47	99.48	99.48	99.48	99.48	99.48	**99.48**
dijkstra	80.23	95.62	95.76	**95.77**	95.77	95.77	95.77	95.77	**95.77**
fft	98.12	99.36	**99.39**	99.39	99.39	99.39	99.39	99.39	**99.39**
gsm	84.18	**92.13**	92.13	92.14	92.14	92.14	92.14	92.14	**92.13**
jpeg	47.64	98.03	98.05	**98.06**	98.06	98.06	98.06	98.06	**98.06**
mad	92.92	98.38	98.61	**98.66**	98.66	98.66	98.66	98.66	**98.66**
patricia	49.83	81.81	88.14	**87.99**	87.99	87.99	87.99	87.99	**87.99**
pgp	7.70	50.53	57.52	94.39	**98.97**	98.97	98.97	98.97	**98.97**
qsort	39.78	93.70	94.46	**94.62**	94.62	94.62	94.62	94.62	**94.62**
rijndael	91.99	92.96	**93.25**	93.24	93.24	93.24	93.24	93.24	**93.23**
stringsearch	63.20	98.04	**98.12**	98.12	98.12	98.12	98.12	98.12	**98.12**
susan	70.09	93.48	**95.94**	95.94	95.94	95.94	95.94	95.94	**95.94**
tiffmedian	59.51	98.10	98.49	**98.55**	98.55	98.55	98.55	98.55	**98.55**
Average	70.5	92.6	93.8	96.1	**96.4**	96.4	96.4	96.4	**96.39**

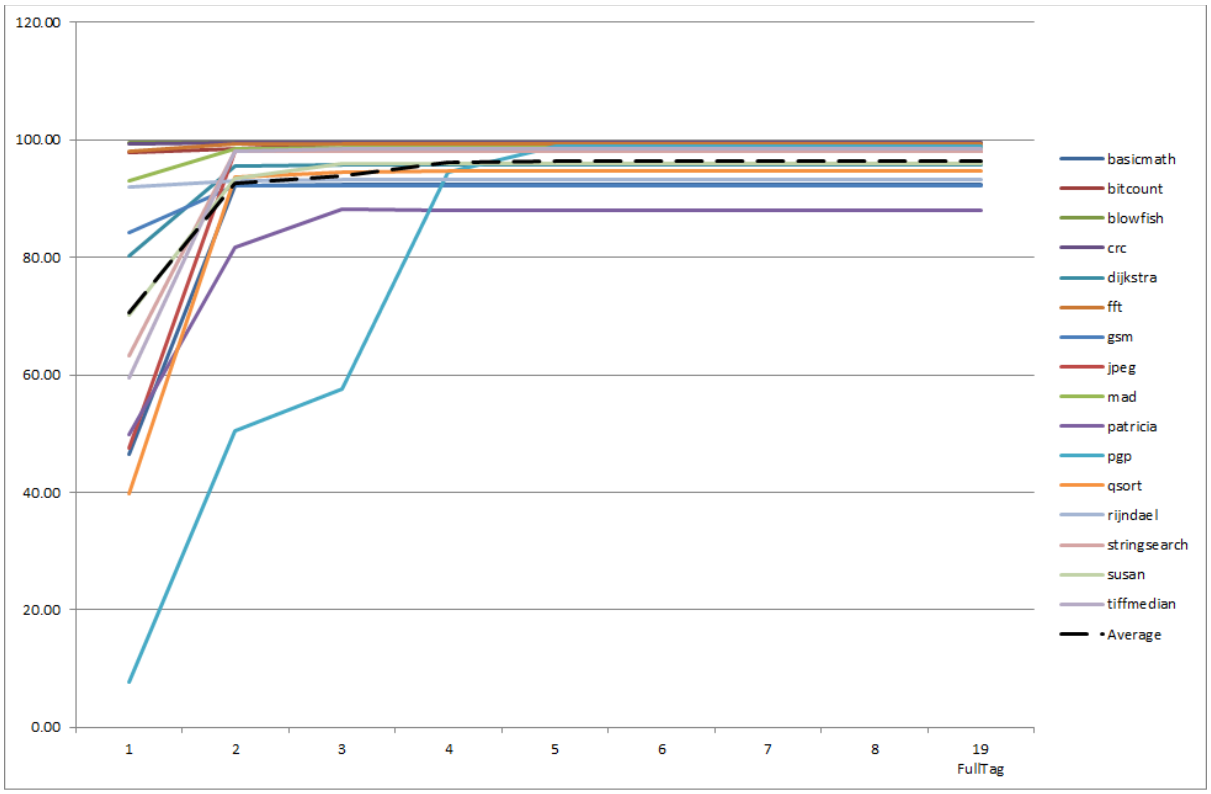


Figure 5.6: Number of TagL bits vs. Hit Ratio [8KB 1Way]

Table 5.6: Number of TagL bits vs. Hit Ratio [8KB 2Way]

	1	2	3	4	5	6	7	8	20 FullTag
basicmath	24.45	39.50	78.52	79.33	93.35	93.46	**93.47**	93.47	**93.47**
bitcount	94.88	97.81	97.85	98.44	99.39	**99.40**	99.40	99.40	**99.40**
blowfish	98.89	98.89	98.89	**99.53**	99.53	99.53	99.53	99.53	**99.53**
crc	81.30	98.95	99.47	**99.50**	99.50	99.50	99.50	99.50	**99.50**
dijkstra	35.25	31.68	70.57	95.60	96.49	**97.23**	97.23	97.23	**97.23**
fft	67.59	98.07	98.13	99.37	**99.40**	99.40	99.40	99.40	**99.40**
gsm	34.14	83.72	92.37	92.43	92.57	**92.77**	92.77	92.77	**92.77**
jpeg	36.89	47.49	98.00	98.16	98.09	**98.15**	98.15	98.15	**98.15**
mad	28.57	92.72	92.99	98.80	98.94	**99.19**	99.19	99.19	**99.19**
patricia	29.00	43.22	62.12	75.52	88.13	**87.99**	87.99	87.99	**87.99**
pgp	2.69	9.29	50.52	64.35	**99.01**	99.01	99.02	99.02	**99.02**
qsort	19.82	33.27	47.69	96.32	**97.55**	97.55	97.55	97.55	**97.55**
rijndael	70.18	88.73	89.15	92.80	93.77	**93.82**	93.82	93.82	**93.81**
stringsearch	62.63	62.48	62.87	97.98	**98.54**	98.54	98.54	98.54	**98.54**
susan	15.45	70.23	55.02	96.43	96.72	**96.99**	96.99	96.99	**96.99**
tiffmedian	52.56	56.82	95.92	98.45	98.61	**98.67**	98.67	98.68	**98.68**
Average	47.1	65.8	80.6	92.7	96.8	**96.9**	97	97	**96.95**

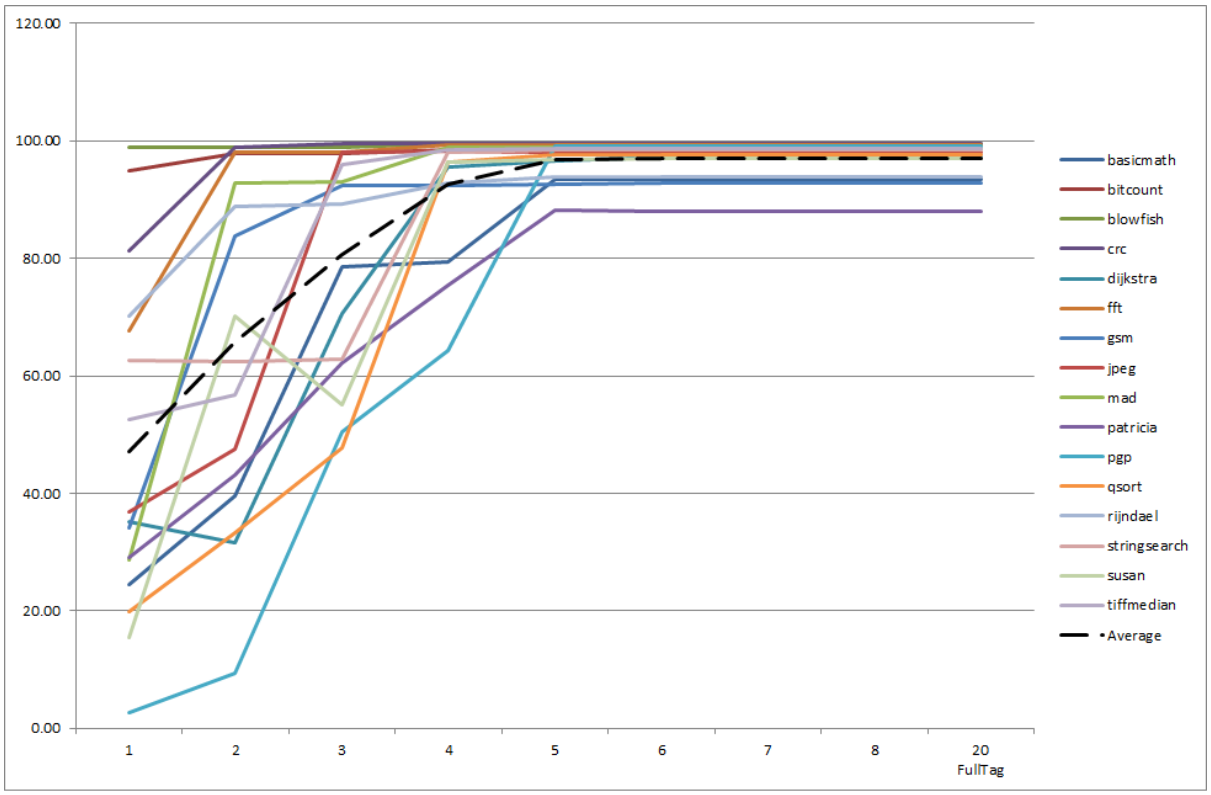


Figure 5.7: Number of TagL bits vs. Hit Ratio [8KB 2Way]

Table 5.7: Number of TagL bits vs. Hit Ratio [8KB 4Way]

	1	2	3	4	5	6	7	8	21 FullTag
basicmath	16.39	30.11	46.89	78.68	93.67	93.86	94.12	**94.14**	**94.14**
bitcount	95.99	97.79	98.35	99.38	99.39	**99.40**	99.40	99.40	**99.40**
blowfish	98.88	99.42	99.43	98.47	**99.53**	99.53	99.53	99.53	**99.53**
crc	98.75	98.91	98.94	**99.50**	99.50	99.50	99.50	99.50	**99.50**
dijkstra	32.35	35.71	64.51	96.38	97.89	98.25	98.55	**98.75**	**98.75**
fft	69.08	68.93	97.19	97.20	97.97	**99.41**	99.41	99.41	**99.41**
gsm	57.29	69.55	89.82	92.35	92.60	**92.83**	92.84	92.84	**92.84**
jpeg	39.25	41.20	47.54	98.03	98.21	98.20	**98.24**	98.24	**98.24**
mad	87.33	87.52	98.85	98.53	98.92	99.14	**99.21**	99.21	**99.21**
patricia	25.24	33.16	63.90	81.48	88.32	88.56	**87.38**	87.39	**87.39**
pgp	8.20	9.23	9.29	50.50	64.38	99.02	99.03	**99.04**	**99.04**
qsort	29.24	31.80	58.76	64.74	96.00	98.96	**99.20**	99.20	**99.20**
rijndael	65.63	70.35	92.27	93.58	93.75	**93.82**	93.82	93.82	**93.82**
stringsearch	61.09	62.54	62.94	63.22	98.35	98.92	98.98	**99.04**	**99.04**
susan	31.10	32.62	93.61	94.27	97.27	97.20	**97.38**	97.38	**97.38**
tiffmedian	58.42	59.29	61.49	97.88	98.39	98.62	98.71	**98.73**	**98.74**
Average	54.6	58	74	87.8	94.6	**97.2**	97.2	97.2	**97.23**

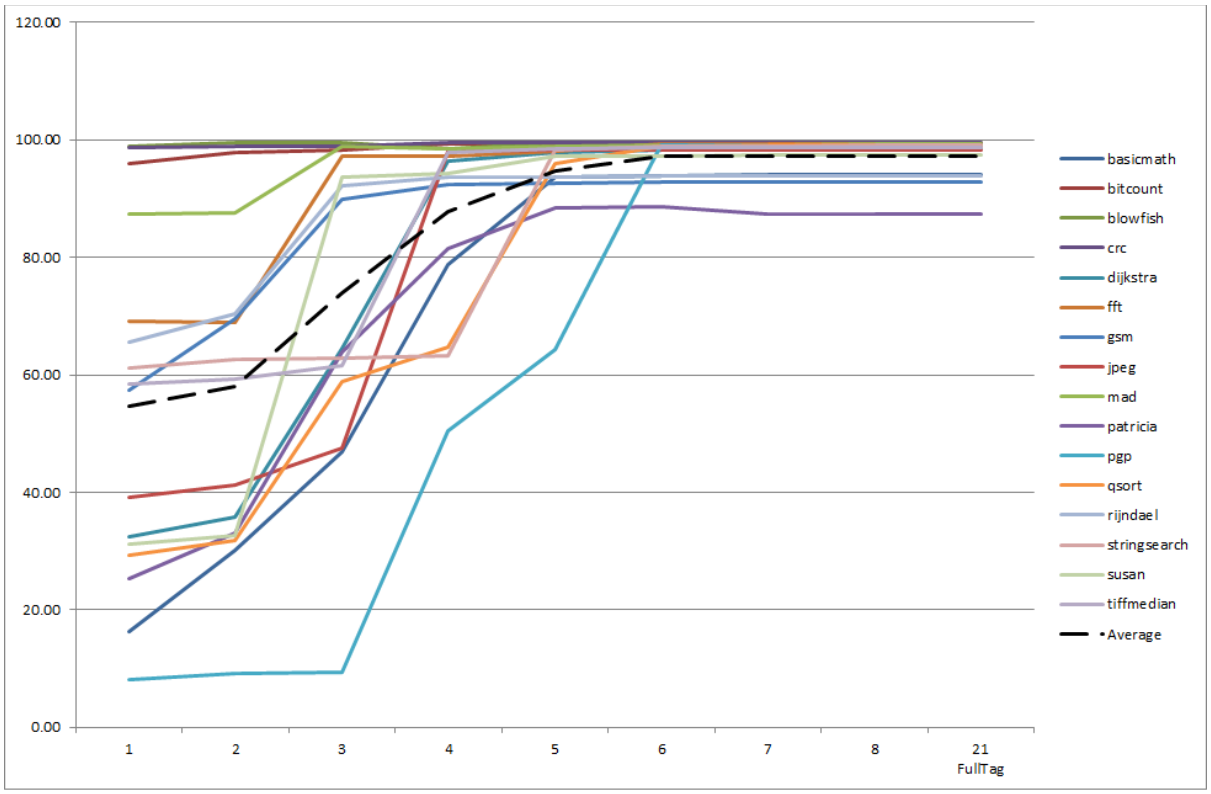


Figure 5.8: Number of TagL bits vs. Hit Ratio [8KB 4Way]

Table 5.8: Number of TagL bits vs. Hit Ratio [8KB 8Way]

	1	2	3	4	5	6	7	8	22 FullTag
basicmath	9.49	23.43	33.80	57.42	79.87	93.12	93.66	93.99	**94.06**
bitcount	96.10	97.41	99.27	99.33	99.38	99.40	**99.40**	99.40	**99.40**
blowfish	59.56	99.51	99.51	99.52	99.52	**99.53**	99.53	99.53	**99.53**
crc	49.97	99.24	99.45	**99.50**	99.50	99.50	99.50	99.50	**99.50**
dijkstra	36.98	58.81	61.65	86.20	95.28	98.56	98.74	**98.97**	**98.97**
fft	34.54	70.47	71.07	99.40	99.41	99.41	**99.42**	99.42	**99.42**
gsm	56.22	58.51	69.16	90.03	92.29	92.81	**92.91**	92.91	**92.92**
jpeg	38.77	39.42	41.26	47.72	98.11	98.26	98.23	**98.25**	**98.25**
mad	77.20	89.13	89.35	98.86	98.88	99.02	99.06	**99.22**	**99.23**
patricia	28.82	38.90	46.30	68.45	80.84	**88.07**	88.60	87.10	**87.11**
pgp	8.94	9.17	9.24	9.34	50.59	99.00	**99.05**	99.05	**99.05**
qsort	24.16	47.72	50.32	58.67	97.75	96.44	99.12	**99.20**	**99.21**
rijndael	65.37	70.19	74.76	93.40	93.81	**93.89**	93.86	93.84	**93.83**
stringsearch	55.69	61.35	63.19	88.16	98.02	**99.34**	99.34	99.34	**99.34**
susan	33.35	35.06	35.24	95.47	97.11	97.40	97.39	**97.42**	**97.42**
tiffmedian	12.94	58.46	59.33	61.45	98.11	98.50	98.51	**98.73**	**98.74**
Average	43	59.8	62.7	78.3	92.4	97	97.3	**97.2**	**97.25**

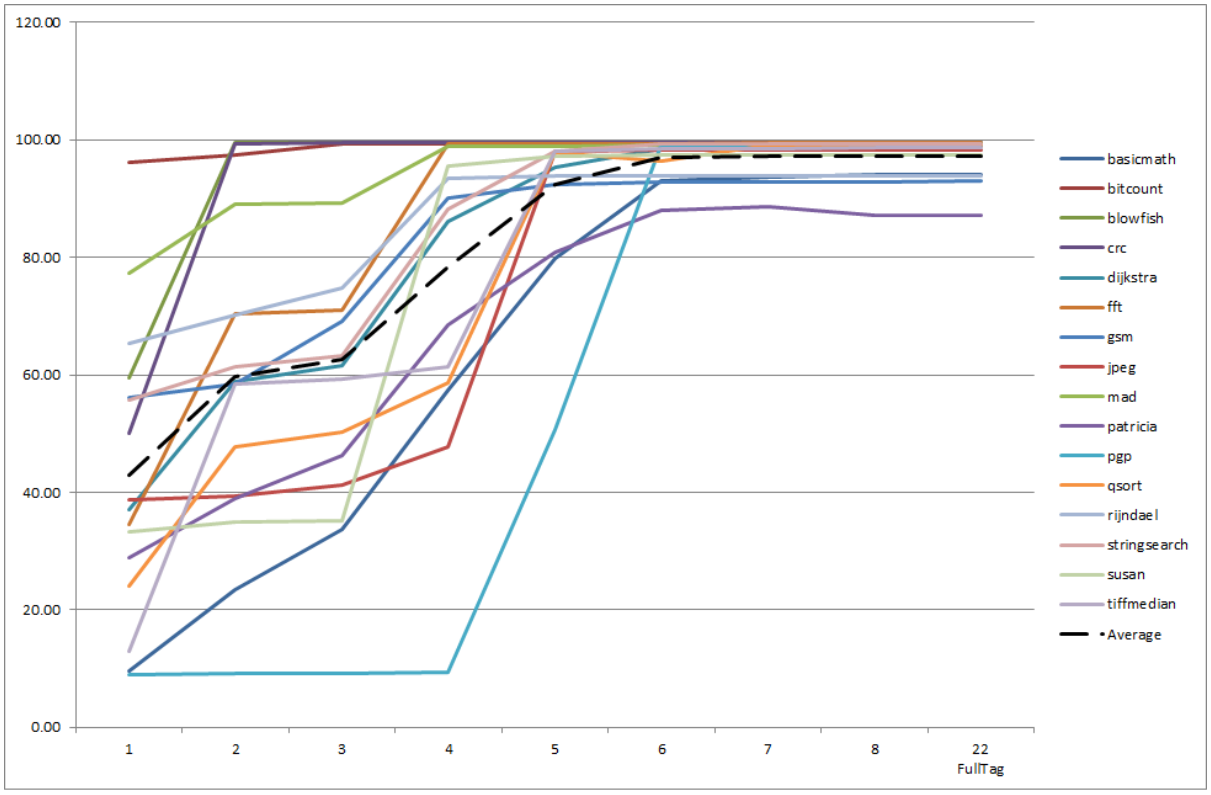


Figure 5.9: Number of TagL bits vs. Hit Ratio [8KB 8Way]

A wide range of simulation had been carried out to analyze the optimum Tag-Length for different cache configurations. This involved changing cache size, associativity and tag length for a vast range of values. A summary is provided by the table below:

Table 5.9 - Average Hit Ratio of various Cache configurations.

	1-bit	2-bit	3-bit	4-bit	5-bit	6-bit	7-bit	8-bit	FullTag
8KB 1Way	70.52	92.61	93.79	96.1	**96.39**	96.39	96.39	96.39	**96.39**
8KB 2Way	47.14	65.8	80.63	92.69	96.85	**96.95**	96.95	96.95	**96.95**
8KB 4Way	54.64	58.01	73.99	87.76	94.63	97.2	97.21	**97.23**	**97.23**
8KB 8Way	43.01	59.8	62.68	78.31	92.4	97.01	97.27	**97.24**	**97.25**
AVERAGE 8KB	53.83	69.06	77.77	88.72	95.07	96.89	**96.96**	96.95	**96.96**
16KB 1Way	93.54	95.1	97.06	**97.31**	97.31	97.31	97.31	97.31	**97.3**
16KB 2Way	68.16	84.19	93.89	97.77	**97.8**	97.8	97.8	97.8	**97.8**
16KB 4Way	60.81	75.36	90.76	97.88	97.93	**97.97**	97.97	97.97	**97.97**
16KB 8Way	65.95	70.29	87.97	95.94	97.94	**98.02**	98	98	**98**
AVERAGE 16KB	72.12	81.24	92.42	97.23	97.75	**97.78**	97.77	97.77	**97.77**
32KB 1Way	95.65	**97.87**	97.87	97.87	97.87	97.87	97.87	97.87	**97.87**
32KB 2Way	86.51	95.02	98.15	**98.17**	98.17	98.17	98.16	98.17	**98.17**
32KB 4Way	83.68	94.91	98.25	98.26	**98.27**	98.27	98.27	98.27	**98.27**
32KB 8Way	90.95	96.33	98.12	**98.3**	98.3	98.3	98.3	98.3	**98.3**
AVERAGE 32KB	89.2	96.03	98.1	**98.15**	98.15	98.15	98.15	98.15	**98.15**

The bold-faced numbers in Table 5.9 also indicate the points at which the actual hits ratios of the partial tag + MASK FIFO policy become the same as those of the conventional full tag policy. The above table helps us to summarize the optimum tag length for various cache configurations.

Additionally, one thing to be noted from table 5.9 is that, the average hit ratios have a similar pattern along different cache sizes. Therefore one can calculate Optimum Tag Length averaged along the Ways for different cache sizes, which can be called Averaged Optimum Tag Length. The Averaged Optimum Tag Length values corresponding to bold grey average hit ratio values. Calculated averaged optimum tag length is plotted along with optimum tag bits in the figure 5.10.

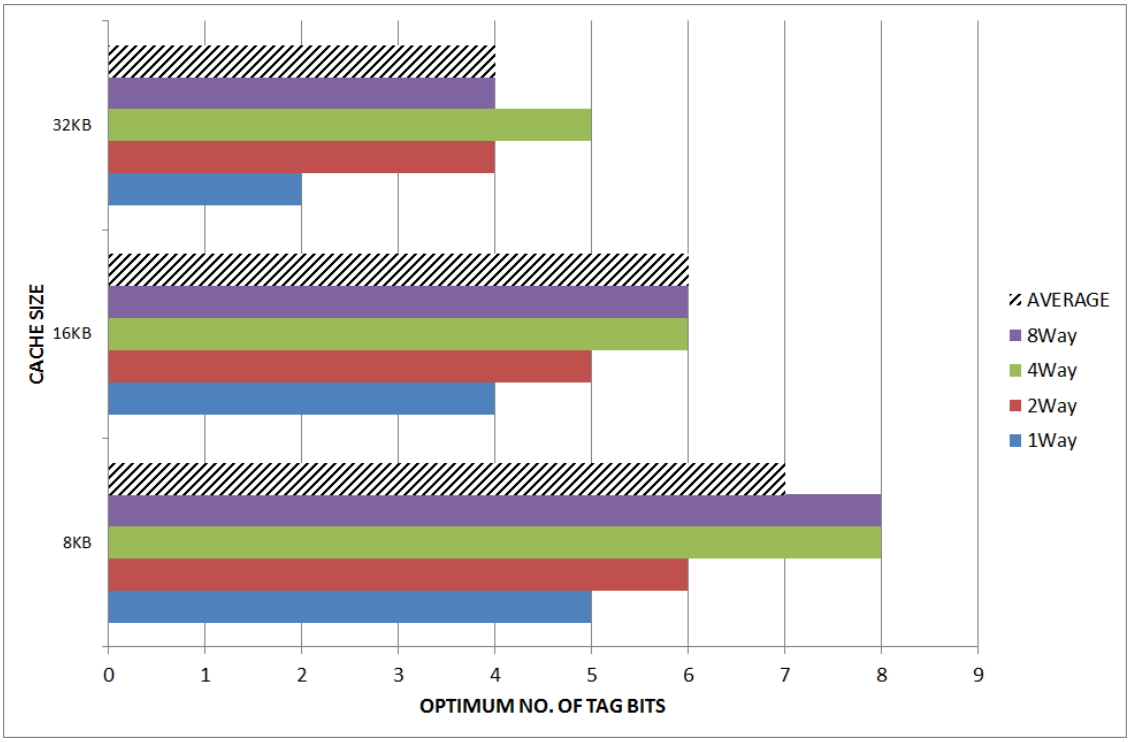


Figure 5.10 - Optimum no. of Tag-Bits with no Performance Degradation

5.2 ESTIMATING POWER DISSIPATION

For analyzing power dissipation in reduced tag set-associative caches, CACTI v5.3 is used. CACTI [4] [16] [17] is an analytical tool that takes a set of cache/memory parameters as input and calculates its access time, power, cycle time, and area. CACTI takes various parameters as input such as Cache Size (bytes), Line Size (bytes), Associativity, No. of Banks, Technology Node (nm), Read/Write Ports , Read Ports, Write Ports, No. of Bits Read Out, Temperature (300-400 K, steps of 10) etc.

As discussion in previous chapters, in associative caches, the energy consumption can be minimized with the reduced no. of tag comparisons. As the associativity increases the number of tag bits increase and the no. of parallel comparisons also increase.

A rigorous power simulation is done for all the cache configurations for different cache sizes with different associativity and with different tag sizes. A typical energy consumption variation for a 16 KB cache with increasing Tag-size is displayed in the following figure.

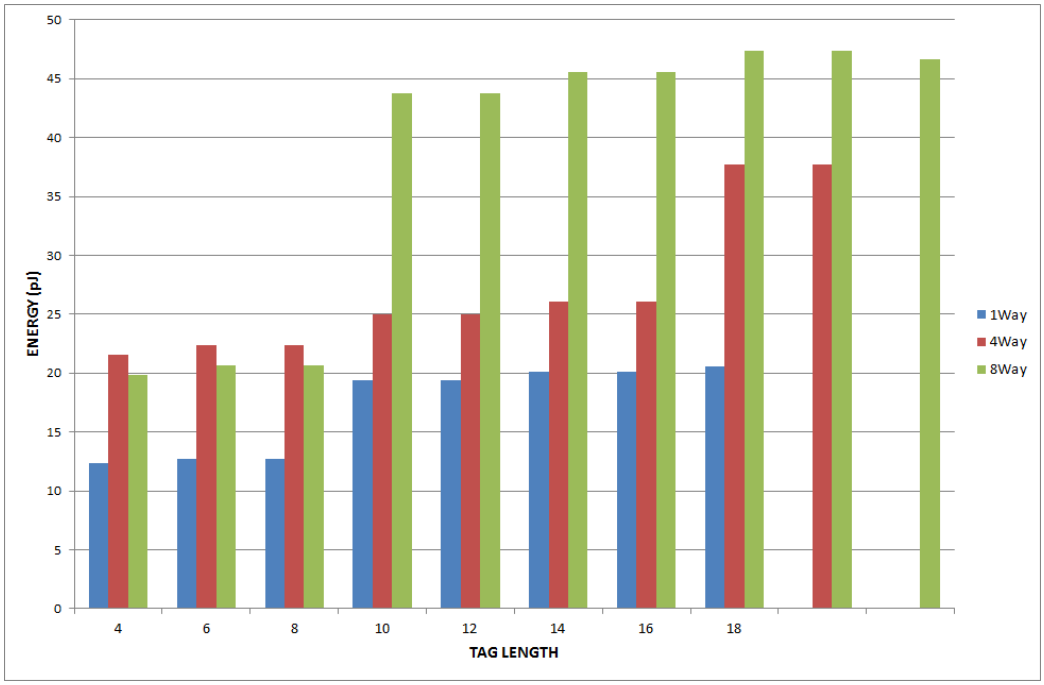


Figure 5.11 - Typical Energy Consumption with varying Tag-Size

As it is expected the energy consumption grows with the increase in number of tag bits. Also the energy consumption is larger for greater associativity. Such set of simulation is carried out for different cache configurations and compared to the energy consumption of a full-tag conventional cache counterpart. As a typical case an energy savings for 16KB plot has been presented in the figure 5.11.

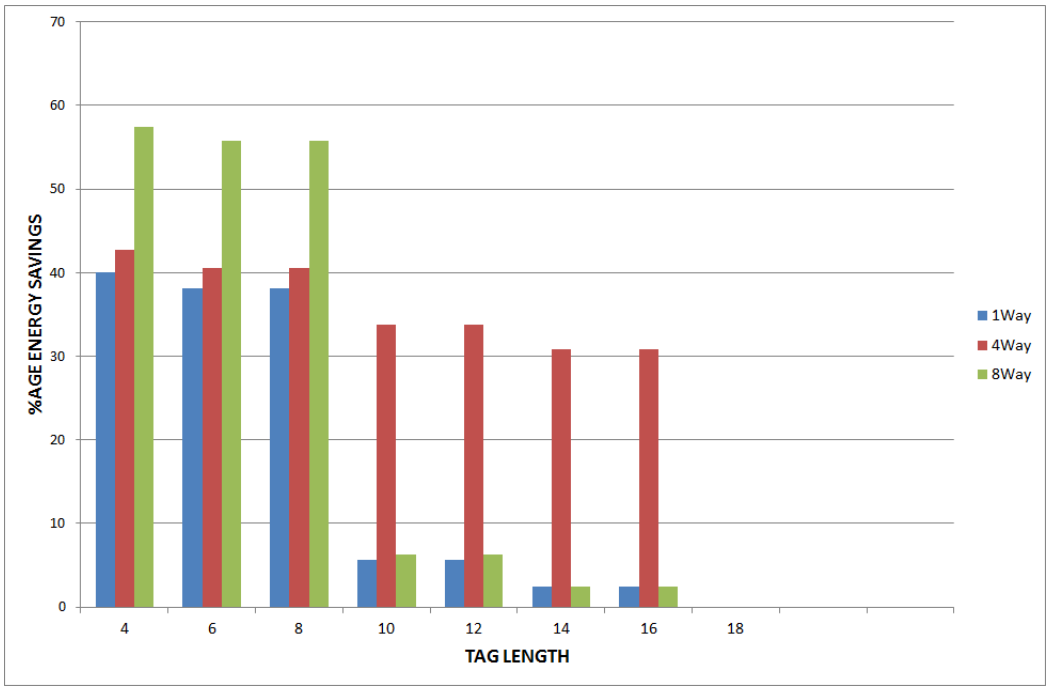


Figure 5.12 - Typical Power Consumption Improvement %age with varying Tag Size

Clearly lesser the number of tag-bits for the greatest associativity is the most energy efficient configuration.

As already discussed (from table 5.9), cache memory with different size and associativity have different optimum tag length. Therefore power simulation results for all the different configurations of cache with their respective optimum tag sizes are plotted. And to summarize, power dissipation results for different sizes of cache with their respective optimum tag-length (ref. Table 5.9) are compared with that of the full tag length to make a comparison in figure 5.12.

Table 5.10 Energy Consumption (in pJ) in different Cache Configurations

	1Way		2Way		4Way		8Way	
	Opt. Tag	Full Tag	Opt. Tag	Full Tag	Opt. Tag	Full Tag	Opt. Tag	Full Tag
8KB	9.71308	10.40924	14.18198	25.86214	15.19472	17.58556	14.8944	40.87615
16KB	12.72054	20.57462	27.00008	28.87946	22.404	40.52272	20.61994	55.79919
32KB	37.56363	40.13541	24.5056	34.58016	25.56752	42.90919	44.28562	64.54164

Even further, referring back to table 5.9 again; 'An Average Optimum Tag-Length' along different Ways for different cache sizes has been calculated. For the average optimum tag length for different sizes of cache has been re-plotted in figure 5.14. Clearly from figure 5.14, it can be seen that there is not much difference in the two graphs, hence one can treat the average optimum tag length for each cache size with minor difference in energy savings.

Table 5.11 % Age Energy Savings (in pJ) in different Cache Configurations

	1Way	2Way	4Way	8Way
8KB (Respective Opt. Tag)	6.687877	45.16315	13.59547	63.56213
16KB (Respective Opt. Tag)	40.03061	6.507683	40.52272	55.79919
32KB (Respective Opt. Tag)	6.407766	29.13393	35.80531	31.38443

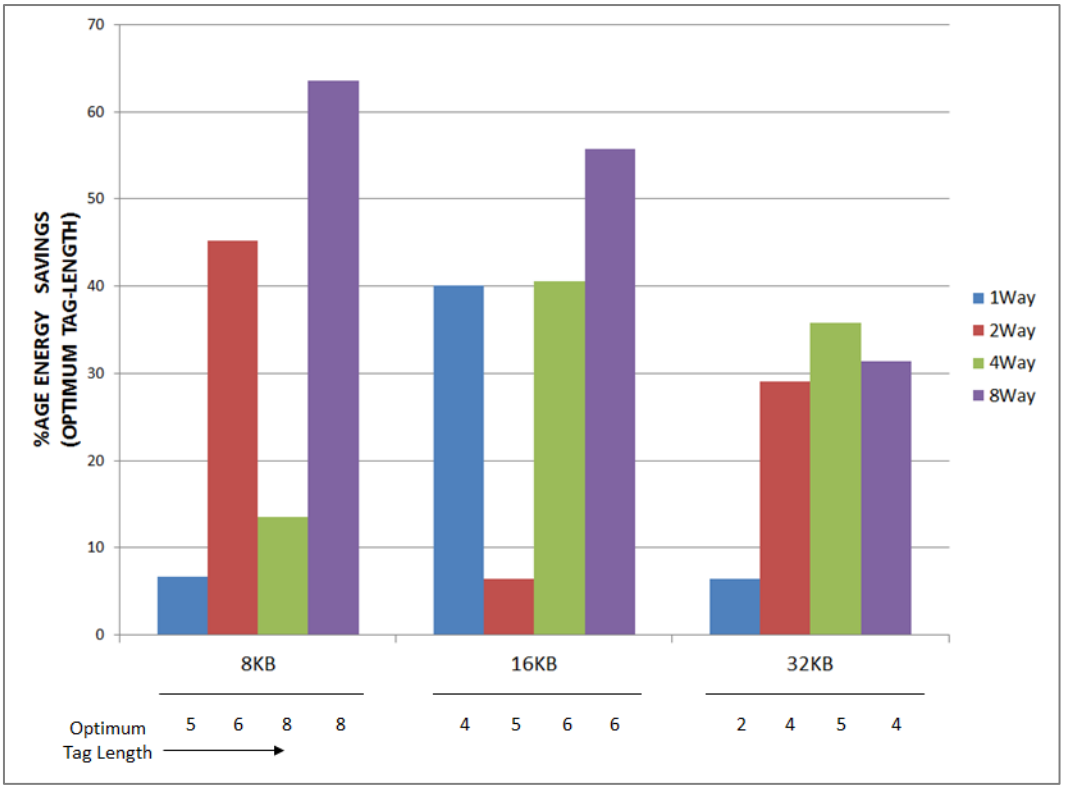


Figure 5.13 - %AGE Energy Savings for Optimum Tag-Length

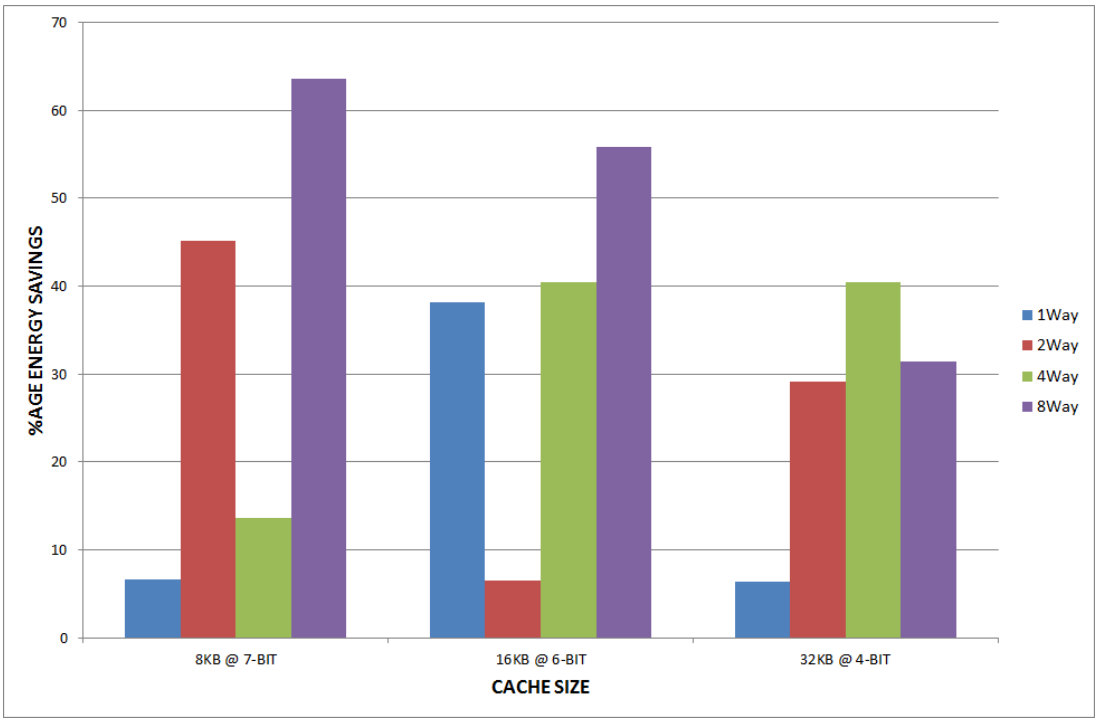


Figure 5.14 - %AGE Energy Improvement for Averaged Optimum Tag-Length

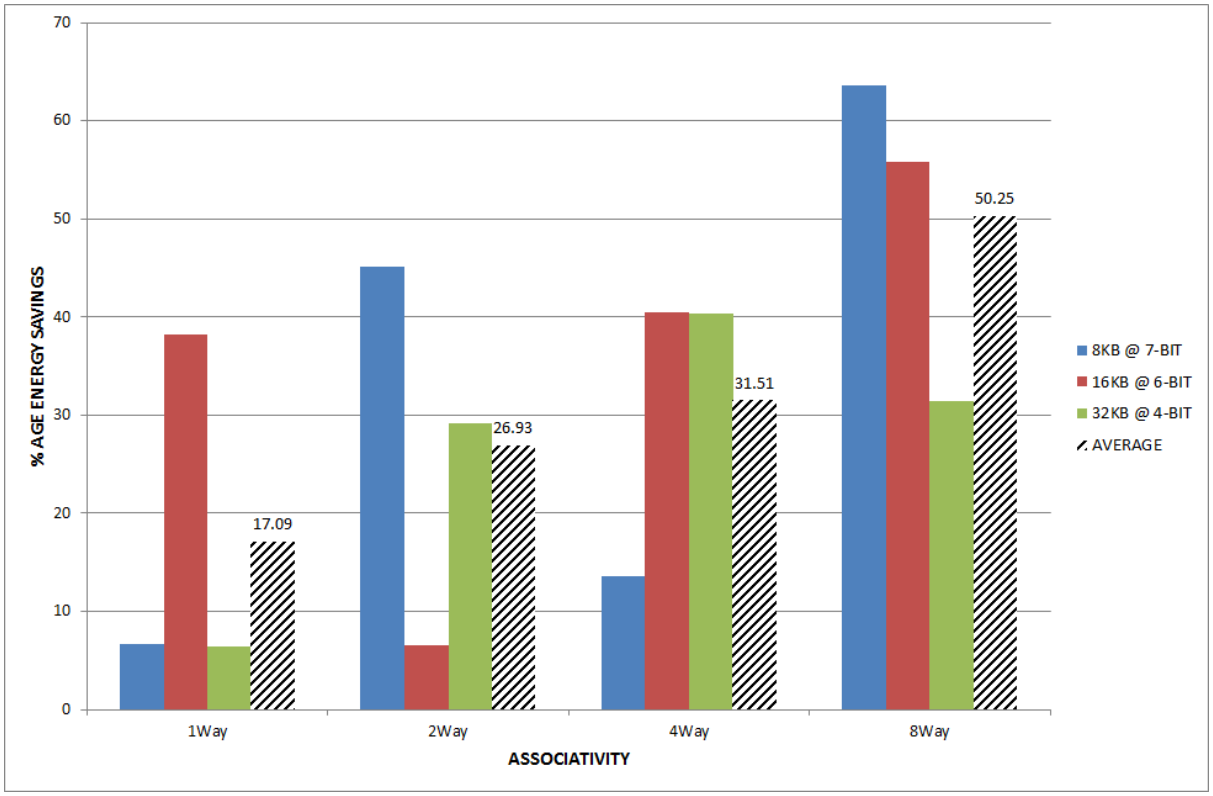


Figure 5.15 - %AGE Energy Savings vs. Associativity

An additional energy savings vs. associativity plot has been derived, that clearly depicts that as the associativity grows the tag bits reduction saves more energy.

CHAPTER 6

CONCLUSION

Now it can be said that, Tag bits of the cache can be reduced to a considerable value to obtain energy efficient caches in embedded systems.

A novel architecture that enables reduced tag structure with modified replacement policies and Way selection mechanism and with least hardware modifications has been proposed. Simulation results confirm the proposal stating that one can obtain optimized tag length for different cache configurations without any performance degradation in terms of hit ratio. Further an average optimum tag-length is computed for different caches sizes.

Energy consumption variation with the optimum tag-size for different cache configurations is computed from standardized simulations. Results show that an energy savings from 9 – 63% can be achieved on different cache configurations, with 8KB-8Way with 8bits of tag achieving maximum energy savings among others.
Further, percentage energy savings were calculated for averaged optimize tag-length for each cache size; were found to be almost consistent with the values calculated using specific tag-length values. Percent energy saving variation along increasing associativity states that caches with higher associativity are more energy efficient when designed using proposed architecture.

Although an average values for the tag bits for a wide variety of application programs has been computed, but an embedded system designer can undoubtedly choose a specific minimum possible tag length for specific application thereby achieving maximum energy savings.

CHAPTER 7

FUTURE SCOPE

Present work established a good framework to investigate and analyze multiple architectural modifications in the cache system architecture.

These are the future prospects for the proposed architecture –

- Architecture can be tested for general microprocessors in addition to embedded processors (ARM in this case).
- SystemC cache model can be refined for synthesis on silicon or FPGA and real-time performance can be evaluated.
- Nonetheless SystemC cache model can be extended to include timing and latency details, thus comparing the performance of processors and cache memory viz a viz.

REFERENCES

[1] Intel Corporation, “An Overview of Cache”, article at http://www.intel.com/design/intarch/papers/cache6.htm

[2] ARM Architecture Reference Model, ARM DDI 0100l, 2005.

[3] Alipour, M.; Salehi, M.E.; Moshari, K., "Cache power and performance tradeoffs for embedded applications," Computer Applications and Industrial Electronics (ICCAIE), 2011 IEEE International Conference on , vol., no., pp.26,31, 4-7 Dec. 2011

[4] Glen Reinman and Norman P. Jouppi; “CACTI 2.0: An Integrated Cache Timing and Power Model”, HP Labs, WRL Research Report 2000/7

[5] SystemC. IEEE-1666 standard http://www.systemc.org

[6] IBM Application note, "Understanding SRAM operation", Mar. 1997.

[7] John L. Hennessy & David A. Patterson, “Computer Architecture: A Quantitative Approach”, Prentice Hall, third edition, Morgan Kaufmann Publishers, 2003.

[8] A. Malik, B. Moyer, D. Cermak, A lower power unified cache architecture providing power and performance flexibility, in: ISLPED’00: International Symposium on Low Power Electronics and Design, pp. 241–243, 2000.

[9] L. Liu, Partial address directory for cache access, in: IEEE Transactions on VLSI Systems, vol. 2, No. 2, pp. 226–240, June 1994.

[10] R. Min, Z. Xu, Y. Hu, W.-B. Jone, Partial tag comparison: a new technology for power-efficient set-associative cache designs, in: VLSID’04: 17th International Conference on VLSI Design, pp. 183–188, Jan. 2004.

[11] T. Sato, I. Artia, Partial resolution in data value predictors, in: Proceedings of International Conference of Parallel Processing, pp. 69–76, 2000.

[12] B-S. Choi, D-I. Lee, Cost-effective value prediction micro-operation using partial tag and narrow-width operands, in: PACRIM’01: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp. 319–322, Aug. 2001.

[13] M. Loghi, P. Azzoni, M. Poncino, Tag overflow buffering: an energy-efficient cache architecture, in: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pp. 520–525, March 2005.

[14] Jong Wook Kwak, Young Tae Jeon, Compressed tag architecture for low-power embedded cache systems, ELSEVIER, Journal of Systems Architecture, Volume 56, Issue 9, pp. 419–428, September 2010.

[15] SimpleScalar LLC to serve and project, Available from: <http://www.simplescalar.com>.

[16] Naveen Muralimanohar, Rajeev Balasubramonian, Norman P. Jouppi, ‘‘CACTI 6.0: A Tool to Model Large Caches”, HPL-2009-85, Tech. Report, Compaq Western Research Lab, Palo Alto, CA, 2009.

[17] Shyamkumar Thoziyoor, Naveen Muralimanohar, and Norman P. Jouppi; “CACTI 5.0”, HP Labs, October 19, 2007

[18] CACTI, An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model, Available from: <http://www.hpl.hp.com/research/cacti>.

[19] D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: a free commercially representative embedded benchmark suite, in: Proc. Fourth IEEE Int. Workshop on Workload Characterization, pp. 3–14, Dec. 2001.

[20] EDN Embedded Microprocessor Benchmark Consortium, Available from: <http://www.eembc.org>.

[21] David C. Black and Jack Donovan, SYSTEMC: FROM THE GROUND UP, Kluwer Academic Publishers.

[22] Hyunsun Park; Sungjoo Yoo; Sunggu Lee, "A Multistep Tag Comparison Method for a Low-Power L2 Cache," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.31, no.4, pp.559,572, April 2012.

[23] Quanquan Li; Lidan Bao; Tiejun Zhang; Chaohuan Hou, "Low power optimization of instruction cache based on tag check reduction," Solid-State and Integrated Circuit Technology (ICSICT), 2012 IEEE 11th International Conference on , vol. no., pp.1,3, Oct. 29 2012-Nov. 1 2012.

[24] A. Agarwal, J. Hennesy, and M. Horowits, “Cache performance of operating systems and multiprogramming,” in ACM Transactions on Computer Systems, pp. 393–431, November 1988.

[25] M. Powell, A. Agrawal, T. Vijaykumar, B. Falsafi, and K. Roy, “Reducing set-associative cache energy via way-prediction and selective direct-mapping,” in 34th Annual International Symposium on Microarchitecture (MICRO’01), pp. 54–65, December 2001.

[26] Alves, M.A.Z.; Khubaib, K.; Ebrahimi, E.; Narasiman, V.T.; Villavieja, C.; Navaux, P.O.A.; Patt, Y.N., "Energy Savings via Dead Sub-Block Prediction," Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th

International Symposium on , vol., no., pp.51,58, 24-26 Oct. 2012.

[27] P. Panda, N. Dutt, Memory Issues in Embedded SoC Optimization and Exploration, Kluwer, 1999.

[28] A. Macii, L. Benini, M. Poncino, Memory Design Techniques for Low-Energy Embedded Systems, Kluwer Academic Publishers, 2002.

[29] Jongmin Lee; Seokin Hong; Soontae Kim, "TLB index-based tagging for cache energy reduction," Low Power Electronics and Design (ISLPED) 2011 International Symposium on , vol., no., pp.85,90, 1-3 Aug 2011.

[30] Mohammad, Baker; Saleh, Hani, "Energy efficient and hign bandwidth embedded memory implementation," Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2013 8th International Conference on , vol., no., pp.117,121, 26-28 March 2013.

[31] Jianwei Dai; Lei Wang, "An Energy-Efficient L2 Cache Architecture Using Way Tag Information Under Write-Through Policy," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.21, no.1, pp.102,112, Jan. 2013.

[32] Mittal, S.; Zhao Zhang; Yanan Cao, "CASHIER: A Cache Energy Saving Technique for QoS Systems," VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on , vol., no., pp.43,48, 5-10 Jan. 2013.

APPENDIX

APPENDIX A: WINDOWS BATCH SCRIPTING

Running multiple console applications/batch files that take multiple inputs and generate multiple outputs is a tedious task. Because one has to selectively run and manage the outputs being generated.

The best option is to use Windows Batch Scripting. Some of the batch programs used in the current work are listed below:

- Generating Simulation EXE Files for different configurations

```
::--------------------GenerateEXE.bat-----------------------------
@echo off
for /f "tokens=* " %%i in (cases.cfg) do (
      cd "C:\Users\Inderjit Singh\Documents\Visual Studio
2008\myinclude"
      geninclude_CATagl %%i
      cd "C:\Users\Inderjit Singh\Documents\Visual Studio
2008\Projects\Clean\Reduced Tag Cache SA [NEW]\Debug"
      devenv ..\Cache_RT_SA_NEW.sln /rebuild
      devenv ..\Cache_RT_SA_NEW.sln /build
      ren Cache_RT_SA_NEW.exe "Cache_RT_SA_NEW_%%i%.exe"
      xcopy *.exe ALL_Threshold\
      del *.exe /f
      @echo CASE  %%i   [DONE]
      pause
      cls
)
::---------------geninclude_CATagl-------------------------------
@echo off
set pwd=%cd%
C:
cd "C:\Users\Inderjit Singh\Documents\Visual Studio 2008\myinclude"
if "%4" == "" (
      @echo 4th argument=NULL
      if "%3"=="" (
            @echo WRONG INPUT
      ) else (
            geninclude_CATagl %1 %2 %3
      )
) else (
      geninclude_CA_TaglLcb %1 %2 %3 %4
)

cd %pwd%
::-----------------------------------------------------------------
```

- Renaming multiple files in current directory

  The following batch program replaces all underscores (_) with the asterisk (*) in the exe filename in current directory

```
::----------------------RENAME.BAT--------------------------------
setlocal enableDelayedExpansion
for /f "usebackq tokens=*" %%i in (`dir /s /b "*.exe"`) do (
    set S=%%~nxi
    set "T=!S:_=*!" :: ← rename string manupulation
    ren "%%~nxi" "!T!"
)
Endlocal
::----------------------------------------------------------------
```

- Running multiple Simulation EXE with inputs and outputs

  As described below, RunMultipleEXE runs a list of executables (EXEList.lst) for each input (inputtestfiles.lst) and keeps a record in batch.log.

  NOTE: Both the list files must have full path of the exe or testfile as their contents.

```
Command: RunMultipleEXE inputtestfiles.lst EXEList.lst batch.log

::--------------------------RunMultipleEXE.bat----------------------
@echo off
for /f "tokens=* " %%i in (%1) do (
      for /f "tokens=* " %%j in (%2) do (
            find /c "%%j %%i" %3
            if ERRORLEVEL 1 (
                  title [%%~nj] %%~nxi
                  cd "%%~dpj"
                  %%~nj %%i
                  cd ..\..
                  @echo %%j %%i>> %3
            ) else (
                  @echo %%i %%j [-Skipped No Action-]
            )
      )
)
::--------------batch.log (sample)-------------------------
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_1.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_2.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_3.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_4.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_5.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
```

```
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_6.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_7.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_2_8.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_4_1.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_4_2.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_4_3.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_4_4.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_4_5.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_4_6.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
J:\thesis2013\TestInput\__SA\NEW\New_RT_SA_4_4_7.exe
J:\thesis2013\TestInput\XLS_original\jpeg__\cjpeg_large.xls
::--------------------------------------------------------------------
```

APPENDIX B: SAMPLE SYSTEMC SOURCE CODE

//Memarray.h

```
template<unsigned int rows,unsigned int data_w,unsigned int tag_w,
            unsigned int A=2,unsigned int log_A=1>
SC_MODULE(memarray){
      sc_in<bool>       tag_input[tag_w]; //Input Port Array
      sc_in<bool>       wline[rows];      //Input Port Array

      sc_out<bool>      data[data_w];     //Output Port
      sc_out<bool>      hit;              //Output Port

      sc_signal<bool>   sig_data[A][data_w];
      sc_signal<bool>   sig_local_hit[A];
      sc_signal<bool>   sig_mem_A_sel[log_A];

      unsigned int      i;
      unsigned int      j;

      memblock_A<rows,data_w,tag_w> mem[A];
      encoder<A,log_A>              e;
      mux_xn_x1<log_A,A,data_w>     mux;
      or_n_1<A>                     or;

      SC_CTOR(memarray):mem(),mux("mux"),e("encoder"),or("or_gate"){
            //PortBindings for "memory-pages"
            for(i=0;i<A;i++){
                  portbind(mem[i].tag_input,tag_input,tag_w);
                  portbind(mem[i].wline,wline,rows);
                  portbind(mem[i].data,sig_data[i],data_w);
                  mem[i].hit(sig_local_hit[i]);
            }

            //PortBindings for "Encoder"
            portbind(e.in,sig_local_hit,A);
            portbind(e.out,sig_mem_A_sel,log_A);

            //PortBindings for "mux"
            portbind(mux.sel,sig_mem_A_sel,log_A);
            for(i=0;i<A;i++)
                  portbind(mux.bus_in[i],sig_data[i],data_w);
            portbind(mux.out,data,data_w);

            //PortBindings for "or"
            portbind(or.a,sig_local_hit,A);
            or.b(hit);
      }
};
```

//PortBind.h

```
template <class type1,class type2>
void portbind(type1 *x,type2 *y,unsigned int len,unsigned int start=0){
      static unsigned int i;
      for(i=0;start<len;i++,start++){
            x[i](y[i]);
      }
}
```

//Mux_n_1.h

```
template<unsigned int sel_count=1,unsigned int in_count=2>
SC_MODULE(mux_n_1){
	//Ports
	sc_in<bool>		sel[sel_count];
	sc_in<bool>		in[in_count];
	sc_out<bool>		out;

	//Signals
	sc_signal<bool>	sig_sel_b[sel_count];
	sc_signal<bool>	sig_and_out[in_count];

	//Variables
	sc_biguint<sel_count>	din;

	//Sub-Modules
	not_n_n<sel_count>			not;
	and_n_1_A<sel_count+1>	and[in_count];
	or_n_1<in_count>		or;

	//Loop variables
	unsigned int i;
	unsigned int j;

	SC_CTOR(mux_n_1):not("Not_gate"),and(),or("Or_gate"){

		portbind(not.a,sel,sel_count);
		portbind(not.b,sig_sel_b,sel_count);

		for(i=0,din=0;i<in_count;i++,din++){
			//and[i]=and_n_1<i_size>("And_Gates");

			for(j=0;j<sel_count;j++){
				if(din[j]==0)
					and[i].a[j](sig_sel_b[j]);
				else
					and[i].a[j](sel[j]);
			}
			and[i].a[j](in[i]);
			and[i].b(sig_and_out[i]);
		}
		portbind(or.a,sig_and_out,in_count);
		or.b(out);
	}
};
```

//TagMatcher.h

```
template<unsigned int width>
SC_MODULE(tagmatcher){
	//PORTS
	sc_in<bool>		in1[width];
	sc_in<bool>		in2[width];
	sc_in<bool>		valid;
	sc_out<bool>		hit;

	//Signals
	sc_signal<bool>	xnor_out[width];
```

```
        sc_signal<bool>    and1_out;

        //SUB-MODULES
        xnor_n_1_A<2>      xnor[width];
        and_n_1<width>     and1;
        and_n_1<2>         and2;

        //Loop Variables
        unsigned int       i;

        SC_CTOR(tagmatcher)

        :xnor(),and1("and_gate1(in_tagmatcher)"),and2("and_gate2(in_tagmatche
r)"){
                for(i=0;i<width;i++){
                        xnor[i].a[0](in1[i]);
                        xnor[i].a[1](in2[i]);
                        xnor[i].b(xnor_out[i]);
                }

                portbind(and1.a,xnor_out,width);
                and1.b(and1_out);

                and2.a[0](and1_out);
                and2.a[1](valid);
                and2.b(hit);
        }
};
```

//Main.h

```
/*Predefined Header Files*/
#include <systemc.h>
#include <conio.h>

/* User Defined Header Files */
#include <portbind.h>
#include <assigntoport.h>
#include <test_cache_file_ask_opt_final.h>
#include <monitor_xyz.h>
#include <init_bus.h>
#include <not_n_n.h>
#include <and_n_1_A.h>
#include <or_n_1.h>
#include <xnor_n_1_A.h>
#include <decoder_n.h>
#include <encoder_n.h>
#include <mux_xn_x1.h>
#include <tagmatcher.h>
#include <portexpander.h>
#include <mem_page_array.h>
#include <address_reg.h>
#include <misshandler.h>
#include <ram_reader.h>
#include <parameters_FT_SA.h>

sc_event    NEW_ADDRESS,
                DECODER_DONE,
                CACHE_MISS,
                CACHE_DONE,
                CACHE_W_DONE;
```

```
SC_MODULE(cache_miss_indicator){
	sc_in<bool>		hit_in;
	void generate(){
		if(!hit_in.read())
			CACHE_MISS.notify();
	SC_CTOR(cache_miss_indicator){
		SC_METHOD(generate);
		sensitive<<CACHE_DONE;
	}
};

int sc_main(int argc,char*argv[]){
	cout<<"Cache Size:\t\t"<<C<<" Bytes"
		<<"\nBlocks per Wordline:\t"<<B
		<<"\nAssociativity:\t\t"<<A
		<<"\nWordlines:\t\t"<<wordlines;

	cout<<"\nTag_Size:\t\t"<<tag_w<<" Bits"
		<<"\nIndex_Size:\t\t"<<index_size<<" Bits"
		<<"\nB_Addr_Size:\t\t"<<b_addr_size<<" Bits"
		<<"\nCache_Data_Width:\t"<<data_w<<" Bits"
		<<"\nLog_A:\t\t\t"<<log_A;

	unsigned int i,j;
	//=========================================
	sc_signal<bool>		address[A_bits],
				add_tag[tag_w],
				dec_in[index_size],
				block_addr[b_addr_size],
				byte_offset[2],
				RW[A],

				wline[wordlines],
				data[data_w],
				data_ram[data_w],
				word[32],
				byte[8];

	sc_signal<bool>		hit;
	hit.write(true);

	testcache<A_bits,wordlines,data_w,32,8,10,prmtr>
	test1("test",argv[0],argv[1]);
	portbind(test1.address,address,A_bits);
	portbind(test1.in1,wline,wordlines);
	portbind(test1.in2,data,data_w);
	portbind(test1.in3,word,32);
	portbind(test1.in4,byte,8);
	test1.out(hit);


	address_reg<A_bits,tag_w,index_size,b_addr_size>
reg("Address_Register");
	portbind(reg.address,address,A_bits);
	portbind(reg.tag,add_tag,tag_w);
	portbind(reg.index,dec_in,index_size);
	portbind(reg.block_addr,block_addr,b_addr_size);
	portbind(reg.byte_addr,byte_offset,2);

	decoder<index_size,wordlines> d("Decoder");
```

```
	portbind(d.in,dec_in,index_size);
	portbind(d.out,wline,wordlines);

	mem_page_array<wordlines,data_w,tag_w,A,log_A>
	cache("Cache_Memory");
	portbind(cache.tag_input,add_tag,tag_w);
	portbind(cache.wline,wline,wordlines);
	portbind(cache.data_ram,data_ram,data_w);
	portbind(cache.RW,RW,A);
	portbind(cache.data,data,data_w);
	cache.hit(hit);

	cache_miss_indicator miss_ind("Miss_Indicator");
	miss_ind.hit_in(hit);

	mux_xn_x1<b_addr_size,B,32> mux_block("mux_block_blockoffset");
	portbind(mux_block.sel,block_addr,b_addr_size);
	for(i=0;i<B;i++){
		for(j=0;j<32;j++)
			mux_block.bus_in[i][j](data[32*i+j]);
	}
	portbind(mux_block.out,word,32);

	mux_xn_x1<2,4,8> mux_byte("mux_byteoffset");
	portbind(mux_byte.sel,byte_offset,2);
	for(i=0;i<4;i++){
		for(j=0;j<8;j++)
			mux_byte.bus_in[i][j](word[8*i+j]);
	}
	portbind(mux_byte.out,byte,8);


	misshandler<wordlines,A>	miss("Misshandler");
	portbind(miss.wline,wline,wordlines);
	portbind(miss.RW,RW,A);
	miss.hit(hit);

	ram_reader<A_bits,data_w>	rread("Ramreader");
	portbind(rread.address,address,A_bits);
	rread.trigger(hit);
	portbind(rread.data_ram,data_ram,data_w);

	sc_start();
	return 0;
}
```