VARIABLE LENGTH PATTERN CODING FOR POWER REDUCTION IN OFF-CHIP DATA BUSES

A Thesis

by

JAYAKRISHNAN VENKITASUBRAMANIAN IYER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2008

Major Subject: Computer Engineering

VARIABLE LENGTH PATTERN CODING FOR POWER REDUCTION IN OFF-CHIP DATA BUSES

A Thesis

by

JAYAKRISHNAN VENKITASUBRAMANIAN IYER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2008

Major Subject: Computer Engineering

ABSTRACT

Variable Length Pattern Coding for Power Reduction in Off-chip Data Buses.

(May 2008)

Jayakrishnan Venkitasubramanian Iyer, B.E., National Institute of Technology Karnataka, Surathkal, India

Chair of Advisory Committee: Dr. Eun Jung Kim

Off-chip buses consume a huge fraction (20%-40%) of the system power. Hence, techniques such as increasing bus widths, transition encoding etc. have been used for power reduction on off-chip data buses. Since capacitances at the I/O pads and interwire capacitances contribute significantly to increase in power, encoding/decoding schemes have been developed to reduce switching activity of the off-chip bus lines, thus reducing power. Frequent-Value Encoding(FVE) [1], Frequent Value Encoding with Xor (FVExor) [1] and VALVE [2] are some of the better known encoding schemes but they still have scope for improvement.

This thesis addresses the problem of power reduction in off-chip data buses by encoding variable number (1 to 4) of fixed-size (32-bit) data values (variable length patterns) which exhibit temporal locality. This characteristic enables us to cache these patterns using 64-entry CAM at the encoder and 64-entry SRAM at the decoder. Whenever a pattern match occurs a 2-bit code indicating the index of the match is sent. If a variable length pattern match occurs then the code and unmatched portion of data is sent.

We implemented our scheme, Variable Length Pattern Coding (VLPC) for various integer and floating point benchmarks and have seen 6% to 49% encodable patterns in these benchmarks. Based on the experiments on simplescalar and our analysis in MATLAB, we obtained 4.88% to 40.11% reduction in transition activity for

SPEC2000 benchmarks such as crafty, swim, mcf, applu, ammp etc. over unencoded data. This is 0.3% to 38.9% higher than that obtained using FVE, FVExor [1] and VALVE [2] encoding schemes. Finally, we have designed a low-power custom CAM and SRAM using 45nm BSIM4 technology models which has been used to verify lower latency of data matching and storing.

DEDICATION

To my loving parents, Mr. K. S. Venkitasubramanian and Mrs. Sundarambal K. Iyer, my loving brother, Jayasuryan and sister-in-law, Sandhya for their blessings and constant encouragement.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep gratitude for my research advisor, Dr. Eun Jung Kim, for her guidance and encouragement throughout my research. I would like to extend my gratitude towards my committee members, Dr. Duncan M. Walker and Dr. Gwan Choi, for their valuable suggestions and guidance. Also, I would like to thank my research group for their feedback on my research.

Finally and most importantly, I would like to thank my family and friends for their endless support throughout the course of this work. I would especially like to thank my brother, Jayasuryan for his infinite belief in me and endless hours of motivation. Without him and my family this would have been impossible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Power optimization is very critical due to its applications to varied domains such as embedded systems, mobile computing environment and other general computing devices. Most of the devices dissipate power dynamically due to charging and discharging of capacitances at the I/O pins. The interwire capacitances between adjacent wires also contribute to dynamic power. Due to 20-40% of the overall system power being consumed by off-chip buses, we see huge scope for power savings at that level. Reduction of switching activity on bus lines reduces dynamic power dissipation at the processor I/O pads, bus drivers, as well as individual bus lines. Hence, previous studies have focused on reducing the switching activity of adjacent wires using transition encoding/decoding. This is achieved by reducing the hamming distance[1] between consecutive values sent over the off-chip buses. The trade-off is a slight increase in capacitances of internal nodes which is much less than that of off-chip bus capacitances, thus increasing power savings.

Encoding/Decoding methods to reduce power at bus-level can be mainly classified as data bus encoding and address bus encoding. These schemes can either be general in nature and hence applicable to both domains [3][4][5] or specific such as [1][6][7][2][8][9] for data bus encoding and [10][11][12][13] for address bus encoding. Addresses sent over the bus being incremental in nature as well as previous studies on address bus encoding do not leave much room for improvement. In this thesis work we mainly concentrate on data bus encoding due to much higher randomness (both

The journal model is *IEEE Transactions on Automatic Control.*

[1]The hamming distance between 2 n-bit binary sequences is the number of bit positions k in which they differ.

spatial and temporal) found in data bus traces rather than address bus traces.

Among these schemes we primarily look at the popular schemes, Frequent Value Encoding (FVE) [1], Frequent Value Encoding with Xor (FVExor) [1] and Variable Length Value Encoding (VALVE) [2]. These schemes have certain drawbacks which we discuss here. Access to the same memory location in most cases do not occur consecutively, hence hamming distance between consecutively coded data is 2. Also, size of the Value Cache (VC) at both encoder/decoder ends is too small viz. 8, 16, 32-entry and is constrained by maximum bus width leading to ineffective utilization of frequent values in the data stream. Frequent value encoding with Xor (FVExor) [1] reduces bit switching for consecutively coded values (from 2 to 1), but in certain cases of non-consecutively coded values it decreases performance. In [2], Yang et. al. focus on coding partially matched data in a given 32-bit value. Fixed partial matching (16, 24, 32-bit) is not very useful due to, excessive data comparison in CAM at encoder and decoder.

Additionally, we extract data traces for different SPEC2000 benchmarks using the simplescalar out-of-order processor simulator. Analysis of these data traces show a large number of consecutive 32-bit data values (1 to 4) exhibiting the property of temporal locality. We call these variable length (1 to 4) consecutive 32-bit data values as *encodable patterns*. The frequent observation of these *encodable patterns* motivates us to cache them and encode them. We keep a sliding window of size $k \in [1, 4]$ which we use to sample the data from the memory buffer at the encoder and the bus buffer at the decoder. Preliminary experiments with a window size of 4 show a high percentage of encodable patterns ranging from 6% to 49% for various SPEC2000 benchmarks.

Therefore, in this thesis we focus on reduction of switching activity by encoding variable length data patterns, spaced out over time. We keep a Variable length

Pattern CAM/cache at both ends(encoder/decoder) to keep track of frequently observed consecutive 32-bit data values (1 to 4 sized data patterns) and send a simple 2-bit code instead of the original code. At the other end of the bus it is decoded before forwarding it to the Unified L2 cache. Using this scheme we get a 4.88% to 40.11% reduction in power over a purely unencoded scheme for various SPEC2000 benchmarks.

Chapter II gives an insight into the related literature on reducing power on off-chip data buses using various encoding techniques, details of problem statement are sketched out in chapter III, our algorithm and explanation follow in chapter IV with experimental results in chapter V and lastly we conclude with comments and future enhancements in chapter VI.

CHAPTER II

BACKGROUND AND LITERATURE

A. Bus Energy Model

The first step is to define the bus energy model which will serve as the backbone to understand the relevant research done in the past. In this thesis we use the following energy model, suggested by [14]. Energy consumption for the bus per cycle can be given as,

$$\begin{aligned} E_{eff} &= \alpha_{switch} \cdot C_{eff} \cdot V_{dd} \cdot V_{swing} && (2.1) \\ &= \alpha_{switch} \cdot C_{eff} \cdot V_{dd}^2 && (2.2) \end{aligned}$$

where α_{switch} is the number of bitlines that switch on the bus for a single cycle, V_{dd} is the source voltage and C_{eff} is the effective bitline capacitance. Also, in the second equation V_{dd} is an upper-bound on V_{swing}, as for buses, $V_{swing} < V_{dd}$. Now as all the bitlines do not switch, we have for K cycles, according to [8]

$$E_{eff} = \left(\sum_{i=1}^{K} (\alpha_{switch})_i \right) \cdot C_{eff} \cdot V_{dd}^2 \qquad (2.3)$$

where $(\alpha_{switch})_i$ is the number of bitline switches at the i^{th} cycle. This is the dynamic power consumed by the bus over a period of K cycles. Bus capacitance mainly depends upon the following factors namely, switching activity, supply voltage, effective bus capacitance (which can be a combination of diffusion capacitance, interwire crosstalk capacitance, driver capacitance, capacitances at the I/O pads etc.). From the above equations it can be seen that the reducing the switching activity on the adjacent bitlines can lead to a significant reduction in the off-chip data bus energy consumption, primarily due to a direct impact on interwire capacitances and capacitances at the I/O pads.

B. Power Reduction Strategies for Off-chip Buses

After a brief introduction to the subject of power dissipation at the bus level in the previous section, it is important to look at the previous work that has been done in this subject. Most of the work done to reduce power dissipation on bus lines is generally classified into the following approaches:

- Changing the bus topologies or widening buses but at the cost of increased cross sectional areas.
- Applying optimal algorithms to Place and Route tools so as to achieve minimal cross talk. The main problem associated with this approach is scalability and its application to off-chip buses.
- Bus encoding techniques so as to reduce bitline switching between adjacent bitlines as well as among consecutive words communicated over the bus lines.

In this thesis we employ an encoding algorithm to reduce power, hence it is important to look at the various bus encoding schemes. Some of the important contributions to reduce power of the off-chip data bus using encoding/decoding techniques are detailed in the next section.

C. Literature Survey

In this section we discuss encoding in both the domains, data bus encoding and address bus encoding.

1. Data bus encoding

- **Bus Invert Coding**

  In its nascent stage there were advances made by Stan/Burleson in [3] to reduce the transition activity on the bus lines by introducing a technique called Bus Invert Coding. The authors sent inverted bit sequence whenever Hamming distance of consecutive words to be sent over the bus was high. They indicated this inverted word using an additional bitline. This was a seminal work with respect to bus encoding techniques. This was improved upon by [4] which is a variant of bus-invert coding. Of significance are the schemes, partial bus-invert coding and multiway partial bus-invert coding [4]. In this scheme, Shin et. al. address the issue of redundancy of bus-invert coding for bus lines which are relatively inactive for most of the data transfer or are uncorrelated. They eliminate this redundancy by identifying highly correlated bus lines and clustering them as groups of subbuses. Each such subbus is encoded independently after clustering. The main disadvantage of bus-invert coding and its variants is their generic nature. Hence these schemes are not able to take advantage of temporal properties of different applications thus leading to modest power savings

- **Bus coding considering inter-wire capacitances**

  Some of the bus encoding techniques [15][16] specifically model buses according to the dynamic power consumed due to inter-wire capacitances while transmitting data over the bus. In this scheme the value ($n = m + a$ bits) sent over the bus at time t, $L(t)$ consists of the data part, $L^D(t)$ (m bits) and the control part, $L^C(t)$ (a bits). Thus control part (a bits) is calculated using an energy function $E = f(D(t) \bigoplus P_r, J_r, L(t-1))$, where $D(t)$ is the original data to be encoded at time t, $L(t-1)$ is the the coded value sent over the bus at time

$t-1$ and $r = 1, 2, .., 2^a - 1$. The authors use a linear combination of predefined set of basis vectors $Y = Y_1, Y_,..., Y_a$ to construct vectors $P = P_1, P_2, ..., P_{2^a-1}$. Also, J_r is r in binary form. At the decoder end the multiplexer uses the control bits ($|L^C(t)| = a$) to select one of the P_rs which eventually aids in getting the decoded data $D(t)$ using $L^D(t) \bigoplus P_r$. [16] also builds on this previous scheme to take a more theoretical approach to bus coding for deep submicron bus technologies.

- **Dictionary-based and Frequent Value Encoding schemes**

  In [17], Tiehan et. al. introduce an adaptive dictionary based encoding/decoding scheme so as to exploit temporal locality of the data transmitted over the bus. They use an adaptive updating mechanism to keep track of transition patterns over time. A similar approach is applied in [1] where Yang et. al. keep a Value Cache (VC) at each end of the bus to keep track of frequent values that have been transmitted over the bus over time. If a frequently occurring value is encountered then the index of the value is sent using a one-hot code, else the unencoded data is sent. They used two approaches to keeping track of frequent values, that of keeping the set of frequent values fixed and that of run-time updation of the value cache. They also use correlator-decorrelator pair which uses the XOR'ing function to reduce switching among consecutive codewords. Suresh et. al. [7], improve over FVE by removing external control signals as well as to have independent MSB-LSB tables for exploiting partial data matches. Some other papers which solve the problem using the same fundamental concept of using a Value Caches (VCs) for encoding frequent values are Variable Length Value Encoder (VALVE) [2] and Power Protocol [6]. Variable Length Value Encoder VALVE [2] is important encoding scheme that we compare against in this work. Yang et. al. [2] concentrate on coding partially matched data in a given 32-bit value. The partial matches are fixed to be 32, 24 or 16 bit with individual caches for each of the partial matched data. Every data value is compared against all these three CAMs and code is generated according to the maximally matched value (32, 24 or 16-bit). Some disadvantages to these schemes are constrained size of Value Cache in FVE [1], which leads to ineffective utilization of frequent values in a working set while VALVE [2] suffers from excessive data comparisons in the 3 CAMs.

- **Encoding schemes based on Frequent value encoding with conceptual and architectural-level modifications**

  Suresh et. al [18] propose a scheme wherein they observe consecutive words to identify hot bits (more transitions) and silent bits (less transitions). These bits which have been identified are encoded independently, for which the authors use an off-line profiling mechanism. They use an M-hot code to encode the hot and silent bits. Two control signals are used, external for indicating whether data(0) or code(1) has been sent on the entire bus, and internal for indicating if the code is available on the upper segment(0) or both segments(1), upper and lower. Through [9], the authors introduce a modified architecture for the the Value Cache (VC) used for Frequent Value Encoding. They have proposed a Hierarchical Unified VC (HUVC), which has multiple cache levels each storing 32-bit data values, while the Hierarchical Combinational VC (HCVC) has multiple cache levels, with monotonically increasing number of smaller sized caches at each level. The HUVC has the cache levels ordered as a binary tree. Thus [9] tries to reduce power consumption through architectural modifications to the Value Cache used in FV encoding [1]. Due to the fundamental encoding scheme used in these enchanced schemes is FVE [1] it suffers from inherent problems of that scheme.

- **Theoretical approach to find optimal codewords for a non-adaptive memoryless encoding scheme**

  In [8], Chee et. al. show a more theoretical approach to minimizing the bit transitions in off-chip data buses. Also, this work is of significance due to its use of a memoryless approach to encoding rather than a conventional approach of adaptive encoding schemes which store information observed in the past. They have developed optimal and explicit codes which can be constructed in polynomial time over the input data set, $S \subset H(n)$ where $H(n)$ is Hamming n-space defined as the set $\{0, 1\}^n$ where for each $u, v \in H(n)$, hamming distance d_H is the number of bit positions in which u and v differ. The authors prove that the codes obtained are optimal and reduce power significantly at the bus-level. The codes have been shown to be theoretically optimal while being memoryless but the performance trade-off associated with the coding scheme has not been considered. Thus the scheme is practicable only if it power/performance trade-off is low for different applications.

- **Prediction-based approach to encoding for power reduction** Yatish et. al. [19] use a prediction based encoding approach to reducing transitions on buses (also called transcoding). Although, the proposed approach is mainly geared towards on-chip data buses, the coding schemes are general in nature and use classical data prediction approaches such as *stride-predictor*, *LAST-value predictor*, *context-based predictor*. Also they explore encoders like the *inversion encoder* and *spatial encoder*. As this method is mainly developed for on-chip buses it would difficult to adapt it for off-chip buses due to increase in latency whenever any data value is mispredicted. Also, mispredicted data has to be sent again leading to reduction in power savings if measured over n cycles of operation.

2. Address bus encoding techniques

In this subsection, we introduce the address bus encoding techniques. Address bus encoding is different from data bus encoding due to much less randomness found in the addresses generated by the processor and are generally found to be consecutive in nature, thus leaving a huge scope to encode them. Physically, the codes can be classified into redundant and irredundant codes. irredundant codes are those which transmit fixed k-bit patterns to encode 2^k data words, while redundant codes are those which transmit additional redundant bits for better performance. The most important motivation behind address bus encoding is that of the following:

- reduced hamming distance leads to reduction in transition activity thereby yielding power savings
- the previous goal can be attained by exploiting heavy spatial locality exhibited by addresses transmitted over the address bus.

[11][20][10] have made important contributions to developing schemes to reduce switching activity over the address bus. [10] uses irredundant gray code to encode stream of consecutive addresses, while [11] improves on this scheme by using an extra bitline *INC* and raising it high to indicate consecutive addresses. Here, the bitlines are frozen to reduce bitline activity when consecutive addresses are found. *INC* remains low for non-consecutive addresses and normal transaction occurs over the address bus. [20] introduced the T0-XOR scheme which reduced switching activity of address buses by almost 74%. The encoder is given by the expression, $B(t) =$

$b(t) \bigoplus (b(t-1)+S) \bigoplus B(t-1)$ where $B(t)$ is the codeword at time t and $b(t)$ is the sourceword at time t. They also proposed the Offset-XOR code which is given by $B(t) = (b(t) - b(t-1)) \bigoplus B(t-1)$. Most of the schemes for address bus encoding are on the same lines as that of those described in the previous section for data buses.

D. Summary

In this chapter we laid the groundwork by describing in brief the low-level details of the influential factors such as interwire capacitances and substrate capacitances on dynamic power dissipation with energy models. Also, we delved into the previous works done on energy reduction using encoding schemes. We categorized the different techniques used (data/address bus encoding) and put each relevant work into perspective. Our work falls into the category of *Dictionary-based encoding schemes* where data patterns are tabulated adaptively to obtain power savings. In this approach we do not use an off-line profiling scheme to set initial parameters, but it can be done in order to achieve customized results depending on the application in consideration. The next chapter explains the details of the problem, while laying out the foundation for our algorithm which is explained in detail in chapter IV.

CHAPTER III

PROBLEM DETAILS

A. Insight into the Problem

An architecture-level block diagram which serves as an example for our data bus encoding scheme (Variable Length Pattern Coding, VLPC) is given in Figure 1. Here, we have a data pattern (3 consecutive 32-bit words) which is encoded using a 2-bit code at the encoder module and sent over the off-chip bus. This 2-bit code is the index into a Content Addressable Memory. The decoder module uses this 2-bit code to retrieve the original data from its cache.

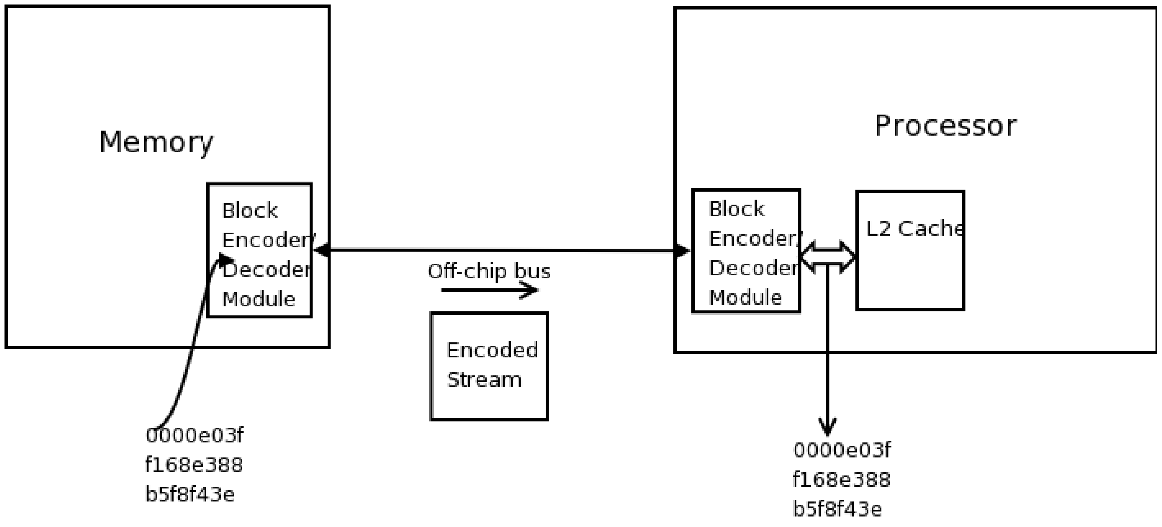

Fig. 1. Encoding variable number (3, here) fixed-length (32-bit) data streams

Data bus encoding techniques in literature have been discussed in the previous section. Some solutions to the drawbacks in those schemes could be summarized here.

- In FVE [1] Hamming distance between consecutively coded data is 2 $\Rightarrow$ disadvantageous for related data. Hence, to overcome this problem, we adaptively encode variable number of fixed-size (say, 32-bit) streams (depending on the bandwidth of the bus) to effectively utilize repeating patterns.

- In FVE [1] size of the Value Cache (VC) at both encoder and decoder ends is too small viz. 8, 16, 32-entry, constrained by the coding scheme. Hence, we use increased number of entries in the CAM/cache for effective utilization of frequently observed data patterns.

- In VALVE [2], excessive data comparison in CAM at encoder and decoder leads to increased power consumption. We overcome this by using a bank-precharging strategy where we shut off $k - i$ CAM banks if there are misses in the i^{th} CAM bank, thus reducing power. Here $k \in [1, 4]$ and $i \in [1, k)$.

In addition to overcoming the disadvantages of the previous schemes, we take advantage of the variable length data patterns which are read from the memory and are observed frequently over time. Considering a 64-byte block is read from the memory, there are consecutive 32-bit data values which are repeated over a period of time. An example of 3 consecutive words that are repeated periodically during the run of the *ammp* SPEC2000 benchmark on simplescalar processor simulator is seen in Figure 1.

As our scheme keeps track of the data patterns observed in the past, it is important to consider the following two approaches to storing data values,

- *Fixed Values*: This is done by doing an off-line profile of the entire data transfer between chip and memory and tabulating frequently occurring values, which in our case would be variable length 32-bit patterns. These values once fixed are not changed over another run of the particular application.

- *Adaptive Updation*: In this approach we have a control circuit which samples the data from the memory buffer and checks the CAM for partial or full matches. It accordingly updates the CAM banks or sends out the code, in the event of a miss or a match, respectively.

The approach to keep the values in the CAM/cache fixed is a simple but rather inflexible solution. The advantages of having a fixed CAM/cache are, non-requirement of the control circuitry for updation and maintenance of data values, elimination of the need for implementing a replacement policy such as LRU/LFU and effectiveness for individual applications in the case of a small set of frequent values. The main drawbacks include, a lack of flexibility across multiple applications, need for off-line profile before running the actual application and if the CAM/cache size (number of entries) is small then it might not be able to fully utilize the frequent values for various working sets[1] of an application.

In our case the cons outweigh the pros mainly due to the fact that for most applications the frequent values do not remain constant over a period of the entire application lifetime. Also, doing an off-line profile for each application is cumbersome and time-consuming. Hence, we decide to go with the on-line updation mechanism for the CAM/cache.

B. Summary

In this section we took a look at the the problem at hand and listed the drawbacks of the popular encoding/decoding methods for reducing transition activity. Then we detailed certain areas which could be improved upon, especially the variable number of cache entries that could be used which is not tied to the encoding algorithm. In

[1]the collection of information referenced by a process during the process time interval $(t-\tau, t)$. This gives an approximation of the information that the application will access in τ time units

addition, we try to exploit the temporal locality of data values alongwith minimal spatial locality due to random distributions of data over space. As the patterns we look at are variable in length, our prefix word (32-bit) matching algorithm matches and stores consecutive words ranging from length 1 to 4, details of which are explained in the next chapter. Also, we explained our rationale for choosing an adaptive updation mechanism over that of a static CAM/cache. The algorithm skeleton and details of each component are explained in the next chapter.

CHAPTER IV

VARIABLE LENGTH PATTERN CODING

In this chapter we look at our variable length pattern coding algorithm in detail. In the following sections, we sketch the outline of the encoding and decoding algorithm with a detailed example in section A.2 and flow charts in section A.3. The algorithmic/pseudo-code representation of the workings of the encoder and decoder are also given in section A.4.

The main components of our encoding algorithm are initialization, Data extraction, Tag comparison, Code generation/Populating cache. A similar albeit simpler structure exists for our decoding algorithm. We expand the skeleton structure by detailing the work of each component in section B. Also we correlate the algorithm with the architectural-level details in section B. The hardware details are handled in sections C and we summarize in section D.

A. Details of the Algorithm

1. Assumptions

- The block size transferred at any point in time from the memory to the cache and cache to memory is same.
- Each block say 64-byte (cache block size) is transferred in one point-to-point burst sequentially as 32-bit consecutive streams (as is handled in simplescalar [21] processor simulator)
- Experiments were conducted assuming a memory bus width of 32-bit in order to make a fair comparison against FVE and FVExor schemes. The algorithm can be scaled to 64-bit buses also, but will need subtle changes in the architecture.

- **Disambiguation**: At the encoder end, the storage table used is a Content Addressable Memory (CAM) [22] for data comparison and storage while at the decoder end we have an SRAM circuit of equivalent size which is used to extract the original data from code.

2. Encoder and decoder operation - example scenarios

In this section we explain how the encoder and decoder works, with an example. We formalize this explantion in the following sections.

a. Encoder operation

Consider the snapshot of an 8-entry CAM which has 4 banks each storing 32-bit values for data matching shown in table 1.

The first scenario of operation of an encoder for a stream of consecutive 32-bit data values from the memory buffer is shown in Figure 2. In this example stream of data in the memory buffer is shown. We take disjoint samples using a sliding window of size 4. As seen in Figure 2 if there is a *full match* a max 2-bit code corresponding to index of the match is sent over the off-chip bus with bitline $encoded = 1$. If there is a miss in the CAM then the original data is sent with $encoded = 0$ and the data is stored in the CAM. Now, in case of a prefix match where prefix is present in the CAM, the index corresponding to the prefix is sent using a max 2-bit code followed by the unmatched data. Here, $encoded = 1$ for the code and $encoded = 0$ for the unmatched data.

The second scenario is shown in Figure 3. The first two cases are similar to scenario 1. Now, in case of a prefix match with prefix not present in the CAM, we insert the prefix into the CAM. To indicate to the decoder that the prefix has to be inserted into it's cache, a padding of size $wind - x$ is added to the prefix, where x

is the prefix match size. The prefix is sent over the bus with *encoded* $= 0$ and the padding is sent with *encoded* $= 1$. The unmatched data is sent with *encoded* $= 0$.

After the scenario 1, the updated CAM is shown in table 2. A similar table exists for scenario 2.

Table 1. Sample snapshot for a 8-entry CAM

index	bank1 data	bank2 data	bank3 data	bank4 data
1	a1072f40	8d042244	1004eb47	1204e947
2	a1f32240	cc042644	a4076c40	400080f4
3	1204e947	00000000	00000000	00000000
4	90dc0120	01000000	a0450120	01000000
5	10410120	01000000	509d0120	01000000
6	1f04ff47	00000000	00000000	00000000
7	080010a4	300050a4	20014040	020000f8
8	020000f8	080050b4	0180fa6b	000050a0

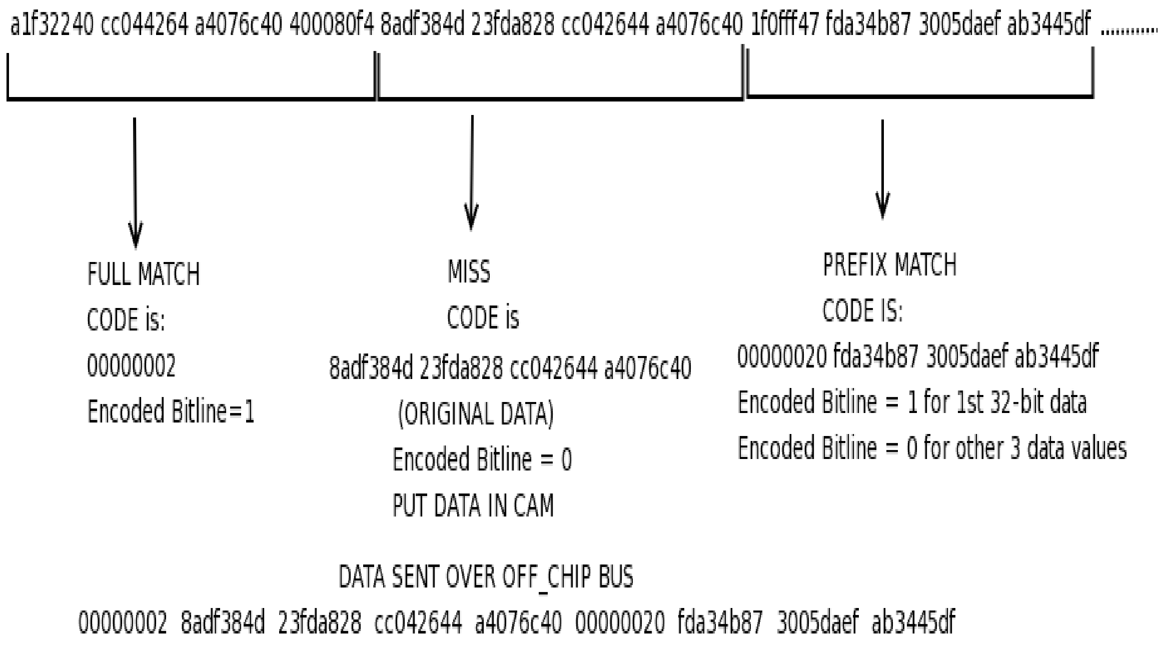
a1f32240 cc044264 a4076c40 400080f4 8adf384d 23fda828 cc042644 a4076c40 1f0fff47 fda34b87 3005daef ab3445df ....
FULL MATCH
CODE is:
00000002
Encoded Bitline=1
MISS
CODE is
8adf384d 23fda828 cc042644 a4076c40
(ORIGINAL DATA)
Encoded Bitline = 0
PUT DATA IN CAM
PREFIX MATCH
CODE IS:
00000020 fda34b87 3005daef ab3445df
Encoded Bitline = 1 for 1st 32-bit data
Encoded Bitline = 0 for other 3 data values
DATA SENT OVER OFF_CHIP BUS
00000002 8adf384d 23fda828 cc042644 a4076c40 00000020 fda34b87 3005daef ab3445df

Fig. 2. Scenario 1 showing cases of full match, miss and prefix matched data found in cache.

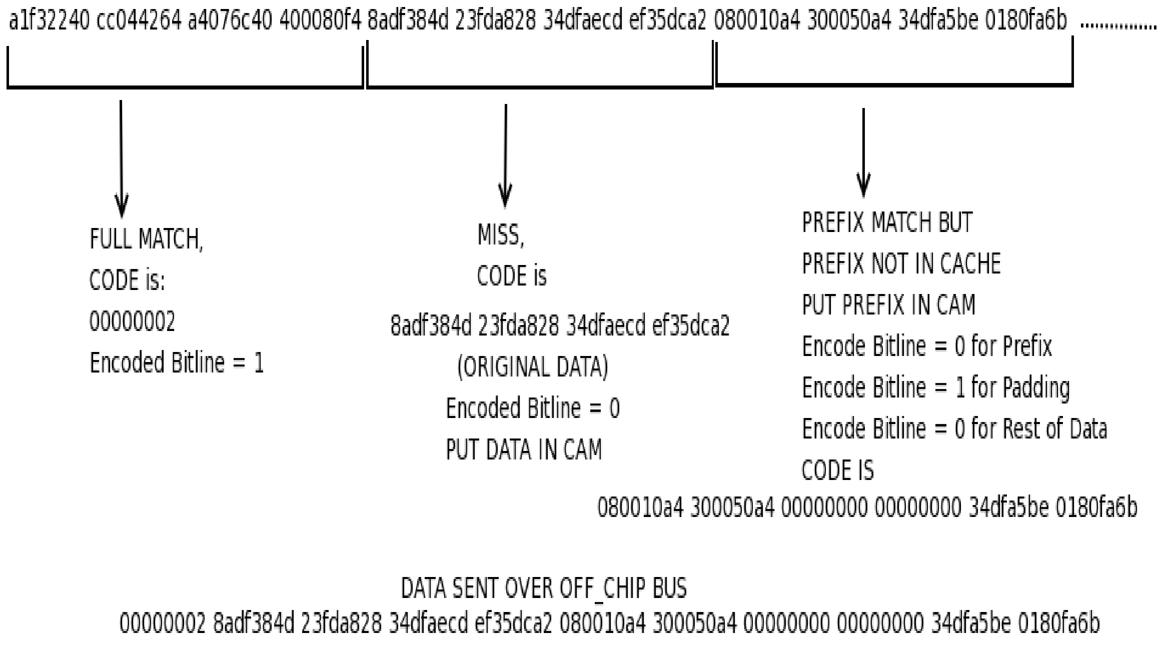
a1f32240 cc044264 a4076c40 400080f4 8adf384d 23fda828 34dfaecd ef35dca2 080010a4 300050a4 34dfa5be 0180fa6b ....
FULL MATCH,
CODE is:
00000002
Encoded Bitline = 1
MISS,
CODE is
8adf384d 23fda828 34dfaecd ef35dca2
(ORIGINAL DATA)
Encoded Bitline = 0
PUT DATA IN CAM
PREFIX MATCH BUT
PREFIX NOT IN CACHE
PUT PREFIX IN CAM
Encode Bitline = 0 for Prefix
Encode Bitline = 1 for Padding
Encode Bitline = 0 for Rest of Data
CODE IS
080010a4 300050a4 00000000 00000000 34dfa5be 0180fa6b
DATA SENT OVER OFF_CHIP BUS
00000002 8adf384d 23fda828 34dfaecd ef35dca2 080010a4 300050a4 00000000 00000000 34dfa5be 0180fa6b

Fig. 3. Scenario 2 showing cases of full match, miss and prefix matched data not found in cache.

Table 2. The 8-entry CAM after inserting the matched prefix and unmatched data into the *LFU_index*.

index	bank1 data	bank2 data	bank3 data	bank4 data
1	a1072f40	8d042244	1004eb47	1204e947
2	a1f32240	cc042644	a4076c40	400080f4
3	1204e947	00000000	00000000	00000000
4	90dc0120	01000000	a0450120	01000000
5	080010a4	300050a4	00000000	00000000
6	1f04ff47	00000000	00000000	00000000
7	8adf384d	23fda828	34dfaecd	ef35dca2
8	020000f8	080050b4	0180fa6b	000050a0

b. Working of the decoder

The decoder cache snapshot is similar to the encoder snapshot in table 1 taken at time $t + k$ where k denotes the operational cycles completed.

Similar to the encoder scenarios depicted above, we have a stream of consecutive 32-bit words taken from the off-chip bus. The first scenario is shown in Figure 4.

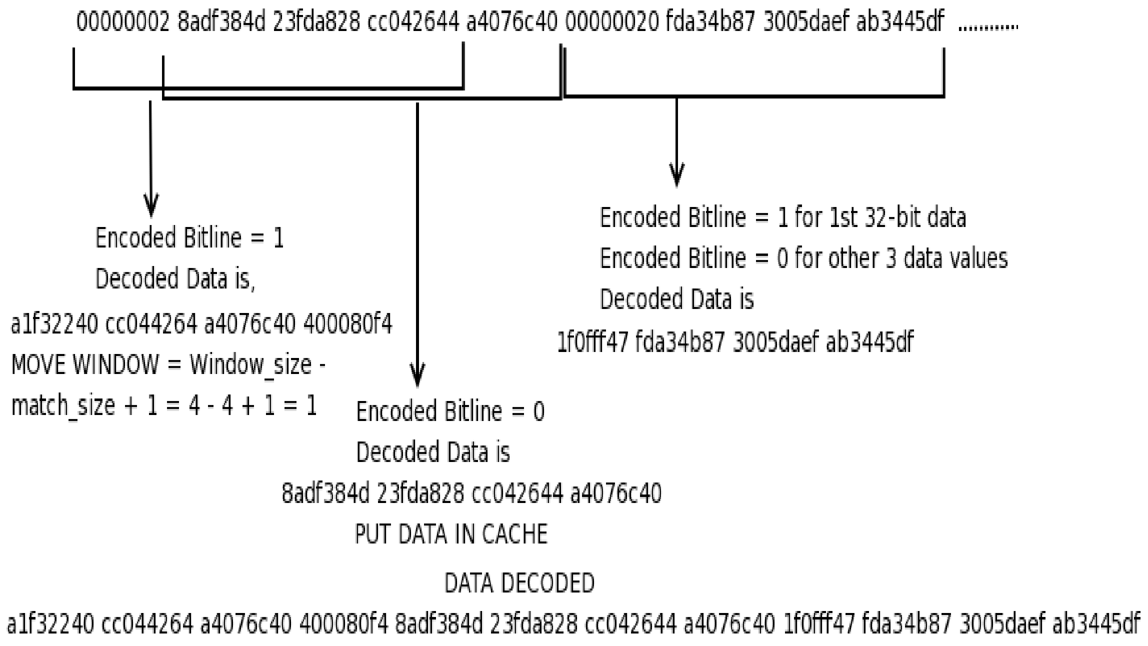
00000002 8adf384d 23fda828 cc042644 a4076c40 00000020 fda34b87 3005daef ab3445df ....
Encoded Bitline = 1
Decoded Data is,
a1f32240 cc044264 a4076c40 400080f4
MOVE WINDOW = Window_size - match_size + 1 = 4 - 4 + 1 = 1
Encoded Bitline = 0
Decoded Data is
8adf384d 23fda828 cc042644 a4076c40
PUT DATA IN CACHE
Encoded Bitline = 1 for 1st 32-bit data
Encoded Bitline = 0 for other 3 data values
Decoded Data is
1f0fff47 fda34b87 3005daef ab3445df
DATA DECODED
a1f32240 cc044264 a4076c40 400080f4 8adf384d 23fda828 cc042644 a4076c40 1f0fff47 fda34b87 3005daef ab3445df

Fig. 4. Scenario 1 showing cases of coded, unencoded and prefix coded data.

Here, if bitline *encoded* = 1 the original data is extracted from the cache corresponding to the max 2-bit code. In case of *encoded* = 0 for the whole window, the data from the bus is the original data is forwarded to UL2 cache and it is also inserted into the cache. Now, if *encoded* = 1 for beginnning 32-bit value of the window and 0 for the remaining $wind - x$ data where x is prefix data size corresponding to the 2-bit code. The data is extracted from cache corresponding to the 2-bit code and forwarded followed by $wind - x$ words of unencoded data.

The second scenario is shown in Figure 5.

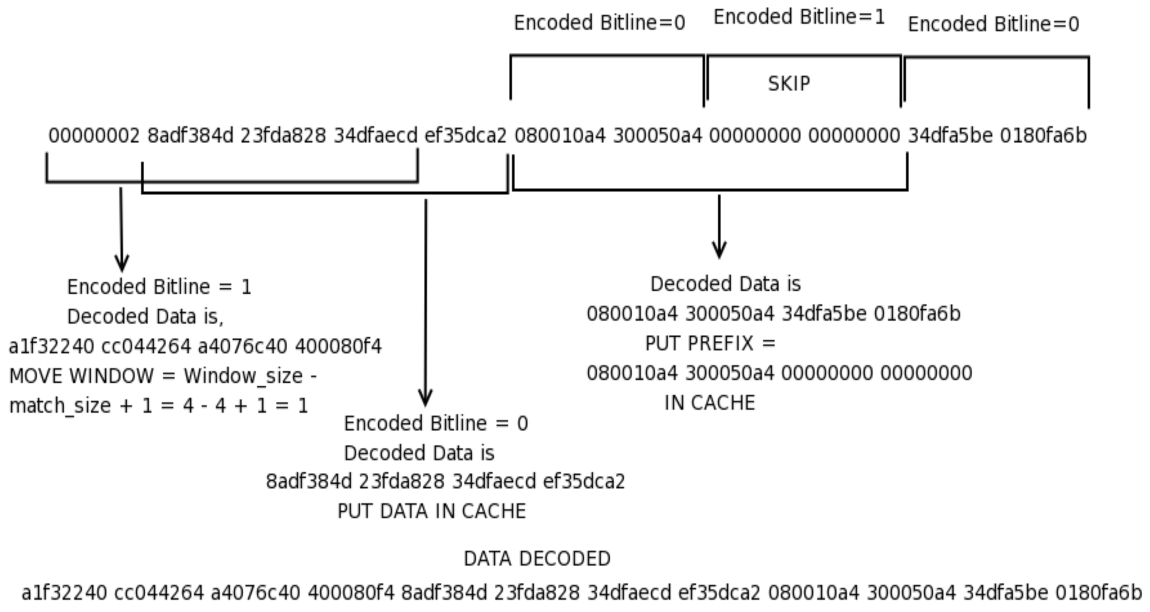
Encoded Bitline=0
Encoded Bitline=1
Encoded Bitline=0
SKIP
00000002 8adf384d 23fda828 34dfaecd ef35dca2 080010a4 300050a4 00000000 00000000 34dfa5be 0180fa6b
Encoded Bitline = 1
Decoded Data is,
a1f32240 cc044264 a4076c40 400080f4
MOVE WINDOW = Window_size - match_size + 1 = 4 - 4 + 1 = 1
Encoded Bitline = 0
Decoded Data is
8adf384d 23fda828 34dfaecd ef35dca2
PUT DATA IN CACHE
Decoded Data is
080010a4 300050a4 34dfa5be 0180fa6b
PUT PREFIX =
080010a4 300050a4 00000000 00000000
IN CACHE
DATA DECODED
a1f32240 cc044264 a4076c40 400080f4 8adf384d 23fda828 34dfaecd ef35dca2 080010a4 300050a4 34dfa5be 0180fa6b

Fig. 5. Scenario 2 showing cases of coded, unencoded data and prefix with padding to be added to the cache.

The first two cases are similar to scenario 2. Now, for a sampled data with padding, we insert the prefix with padding in the cache and remove the padding. Then we extract the following unencoded data with $size(padding)$. This consolidated data is sent to the UL2 cache.

An updated CACHE after scenario1 is complete is shown in table 3. A similar table exists for scenario 2.

Table 3. Updated 8-entry CACHE at the decoder after insertion.

index	bank1 data	bank2 data	bank3 data	bank4 data
1	a1072f40	8d042244	1004eb47	1204e947
2	a1f32240	cc042644	a4076c40	400080f4
3	1204e947	00000000	00000000	00000000
4	90dc0120	01000000	a0450120	01000000
5	080010a4	300050a4	00000000	00000000
6	1f04ff47	00000000	00000000	00000000
7	8adf384d	23fda828	34dfaecd	ef35dca2
8	020000f8	080050b4	0180fa6b	000050a0

3. Flow chart

In Figure 6 we have the flow chart for the encoder. We can clearly mark out the stages: initialization, tag comparison and prefix matching, code generation/populating the cache. With certain marked differences to the encoder in Figure 6 we have the flow chart diagram for the decoder in Figure 7. Here too, we have clearly separated stages of initialization, recovering original data/populating the cache. The stages are explained in Section B.

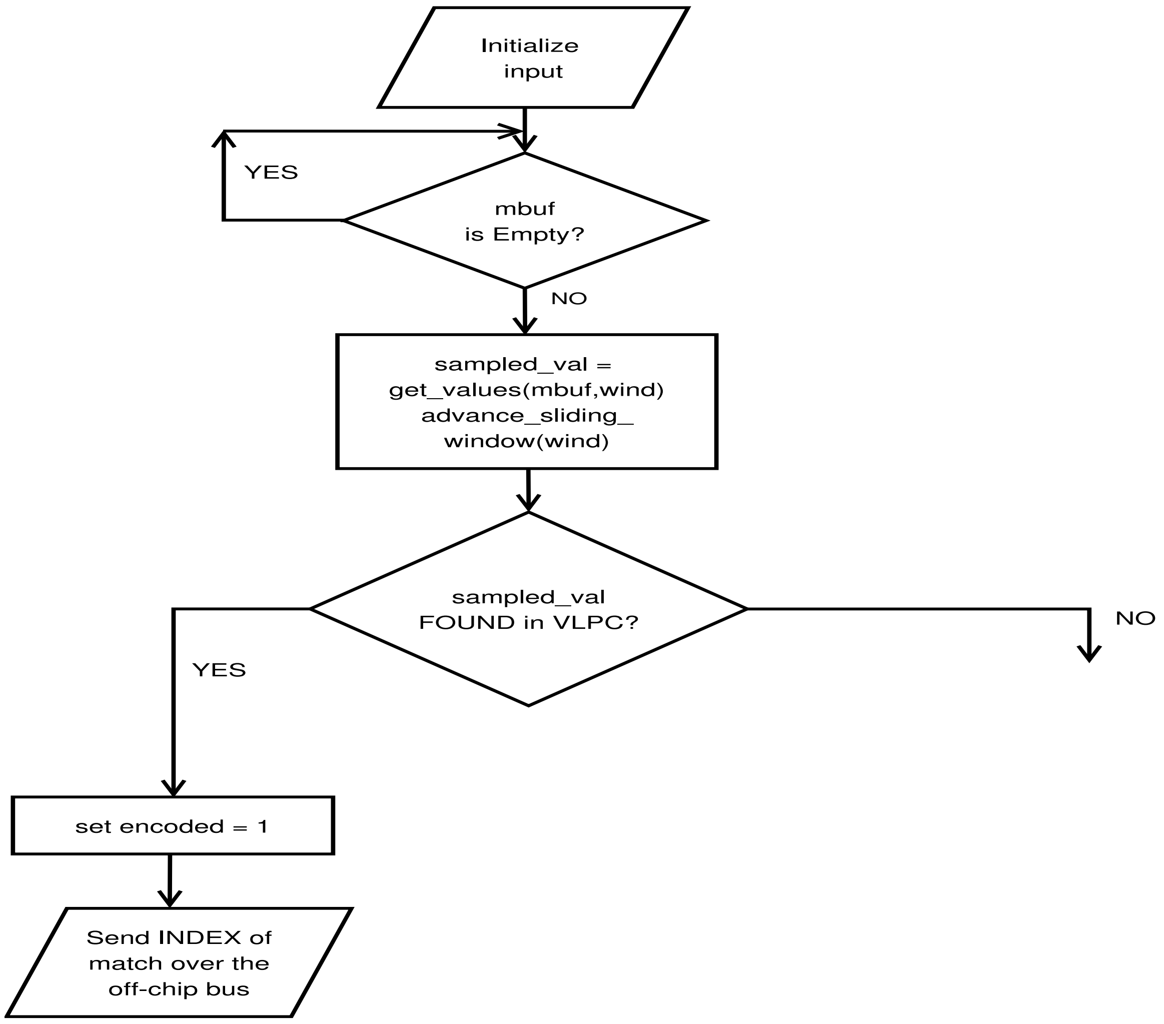
Initialize input
YES
mbuf is Empty?
NO
sampled_val = get_values(mbuf,wind) advance_sliding_ window(wind)
sampled_val FOUND in VLPC?
YES
NO
set encoded = 1
Send INDEX of match over the off-chip bus

Fig. 6. Flow chart description of working of the encoder

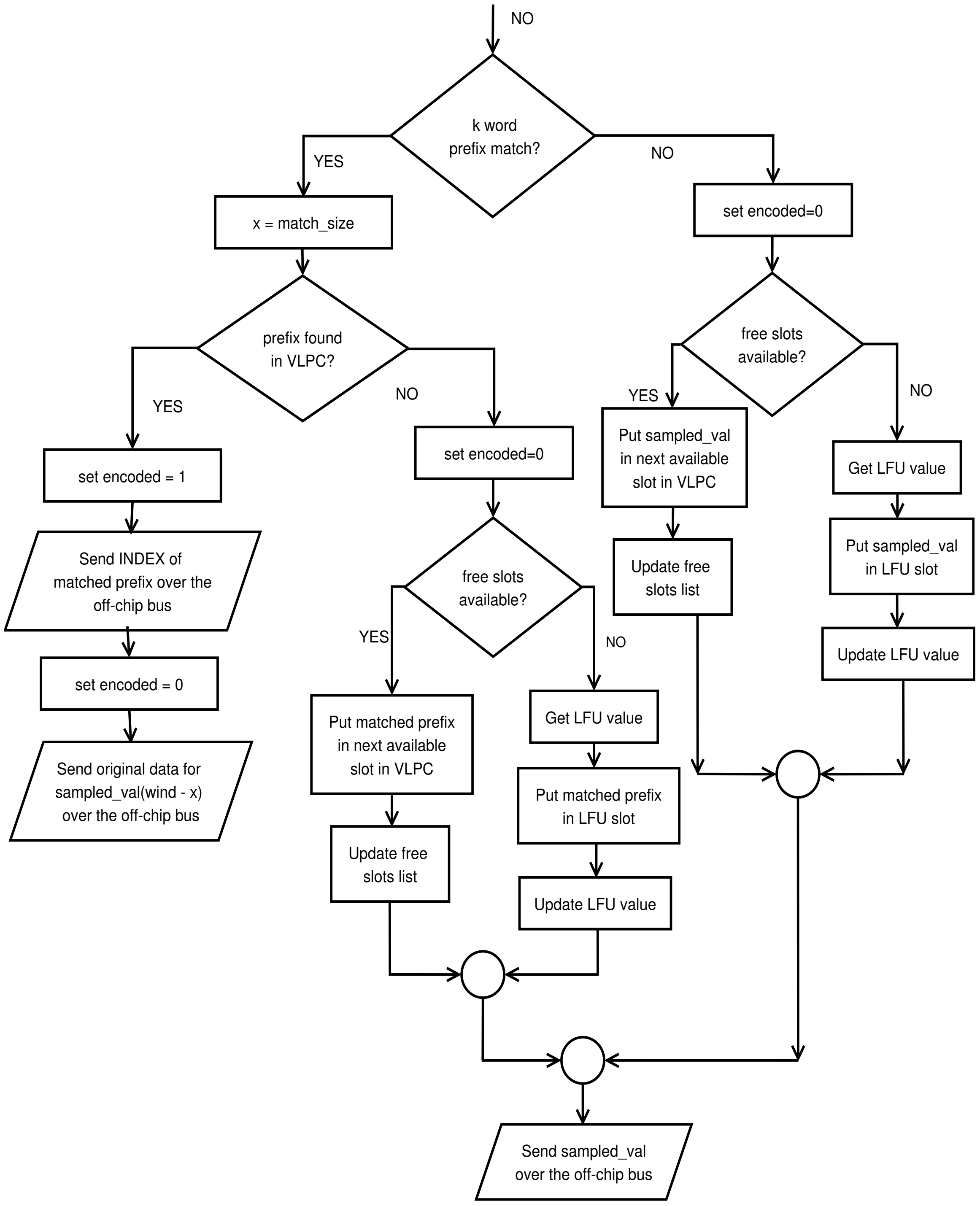
NO
k word prefix match?
YES
NO
x = match_size
set encoded=0
prefix found in VLPC?
YES
NO
free slots available?
YES
NO
set encoded = 1
set encoded=0
Put sampled_val in next available slot in VLPC
Get LFU value
Send INDEX of matched prefix over the off-chip bus
free slots available?
Update free slots list
Put sampled_val in LFU slot
YES
NO
Update LFU value
set encoded = 0
Get LFU value
Send original data for sampled_val(wind - x) over the off-chip bus
Put matched prefix in next available slot in VLPC
Put matched prefix in LFU slot
Update free slots list
Update LFU value
Send sampled_val over the off-chip bus

Figure 6 continued.

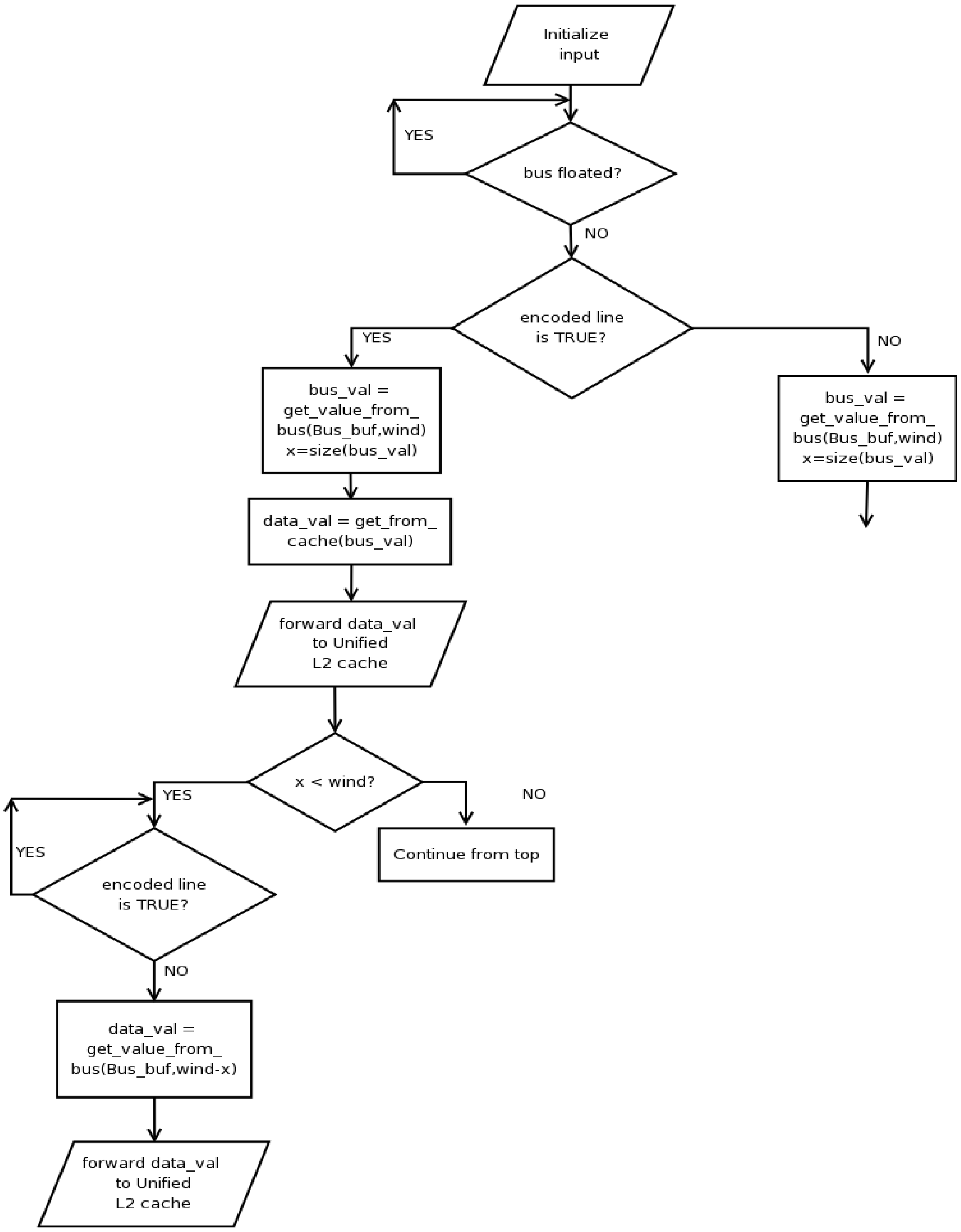

Fig. 7. Flow chart description of working of the decoder

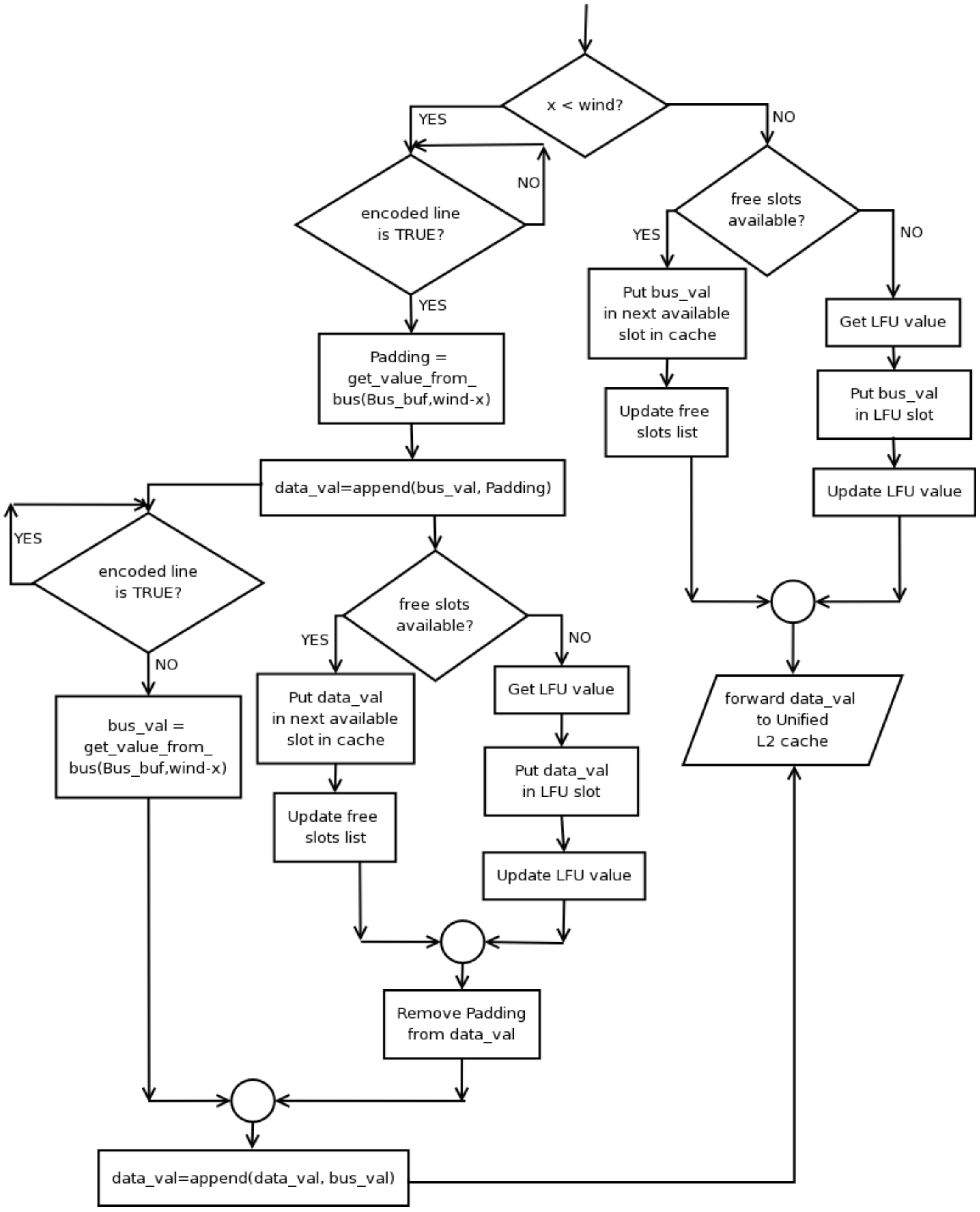
x < wind?
YES
NO
encoded line is TRUE?
NO
YES
Padding = get_value_from_bus(Bus_buf,wind-x)
data_val=append(bus_val, Padding)
YES
encoded line is TRUE?
NO
bus_val = get_value_from_bus(Bus_buf,wind-x)
free slots available?
YES
NO
Put data_val in next available slot in cache
Update free slots list
Get LFU value
Put data_val in LFU slot
Update LFU value
Remove Padding from data_val
data_val=append(data_val, bus_val)
free slots available?
YES
NO
Put bus_val in next available slot in cache
Update free slots list
Get LFU value
Put bus_val in LFU slot
Update LFU value
forward data_val to Unified L2 cache

Figure 7 continued.

4. Algorithm pseudo-code

The algorithm to encode the data to be transferred from the memory buffer to the processor via the off-chip data bus is given in Alg:1

Algorithm 1 Encoding Algorithm for coding Variable length 32-bit data patterns

```
1: $mbuf \Leftarrow$ addr_memory_buff {Address of the head of the memory buffer is given
   by $mbuf$}
2: $wind \in [1, 4]$ {$wind$ is the size of the sliding window which determines the size of
   data sampled at any given time $t$}
3: $free\_slots = size\_of\_CAM()$ {$free\_slots$ gives the number of free slots available
   at any time $t$. When free_slots_available is called, the value of $free\_slots$ is checked
   against 0}
4: while TRUE do
5:   while $mbuf = NULL$ do
6:     idle
7:   end while
8:   if POSEDGE(CLK1) then
9:     sampled_value $\Leftarrow$ copy_values($mbuf + wind * next\_offset$)
10:    advancewindow(wind)
11:    if comparevalue(sampled_value, VLPC) = FOUND then
12:      set encoded_control_line $\Leftarrow$ 1
13:      send(index_of_the_match)
14:    else
15:      if 32bit_prefix_match = TRUE then
16:        $x \Leftarrow$ prefix_match_size
17:        matched_prefix = sampled_value($x$)
```

```
18:     **if** NotIn(VLCP,prefix) = TRUE **then**
19:       **if** free_slots_available() = TRUE **then**
20:         $slt \Leftarrow$ get_next_free_slot
21:         matched_prefix $\Rightarrow$ VLPC($slt$)
22:         update_free_slots();
23:       **else**
24:         $slt \Leftarrow$ get_LFU_index()
25:         matched_prefix $\Rightarrow$ VLPC($slt$)
26:         update_LFU()
27:       **end if**
28:       set encoded_control_line $\Leftarrow 0$
29:       send(matched_prefix)
30:       padding $\Leftarrow$ zero_words($wind - x$) {For sending to the decoder for
         inserting in cache at that end}
31:       set encoded_control_line $\Leftarrow 1$ {Indicator for the decoder so as to insert
         in cache and skip forwarding to UL2 cache}
32:       send(padding)
33:       set encoded_control_line $\Leftarrow 0$
34:       send(sampled_value($wind - x$)) {Sending unmatched suffix data to
         be forwarded as is by decoder}
35:     **else**
36:       set encoded_control_line $\Leftarrow 1$
37:       send(index_of_prefix_match)
38:       set encoded_control_line $\Leftarrow 0$
39:       send(sampled_value($wind - x$))
40:     **end if**
```

```
41:         else
42:           if free_slots_available() = TRUE then
43:             $slt$ ⇐ get_next_free_slot
44:             sampled_value ⇒ VLPC($slt$)
45:             update_free_slots()
46:           else
47:             $slt$ ⇐ get_LFU_index()
48:             sampled_value ⇒ VLPC($slt$)
49:             update_LFU()
50:           end if
51:         end if
52:       end if
53:     end if
54: end while
```

The decoding algorithm which is used to decode the data received at either end of the bus follows the algorithmic structure below in Alg:2

Algorithm 2 Decoding Algorithm for coding Variable length 32-bit data patterns

```
1: $bus\_buf$ ⇐ bus_data_buff {Address of the head of the bus data buffer is given by $bus\_buf$}
2: $wind \in [1,4]$ {$wind$ is the size of the sliding window which determines the size of data sampled at any given time $t$}
3: $free\_slots = size\_of\_cache()$ {$free\_slots$ gives the number of free slots available at any time $t$. When free_slots_available is called, the value of $free\_slots$ is checked against 0}
4: initialize
```

```
5: while TRUE do
6:   while bus_floated() = TRUE do
7:     idle
8:   end while
9:   if encoded_control_line = 1 then
10:     $bus\_value \Leftarrow$ get_data_from_bus($bus\_buf$,$wind$) {Function fills data value
       buffer till encoded_control_line is in same state(0/1), else returns}
11:     $x \Leftarrow$ size($bus\_value$)
12:     $data\_value \Leftarrow$ get_data_from_cache($bus\_value$)
13:     forward_to_UL2($data\_value$) {Assuming the next component is a Unified L2
       Cache}
14:     if $x < wind$ then
15:       $data\_value \Leftarrow$ get_data_from_bus($bus\_buf$,$wind - x$)
16:       forward_to_UL2($data\_value$)
17:     end if
18:   else
19:     $data\_value =$ get_data_from_bus($bus\_buf$,$wind$) {Function fills data value
       buffer till encoded_control_line is in same state(0/1), else returns}
20:     $x \Leftarrow$ size($data\_value$)
21:     if $x < wind$ and encoded_control_line = 1 then
22:       $padding \Leftarrow$ get_data_from_bus($bus\_buf$,$wind - x$)
23:       $data\_value \Leftarrow$ append($padding$,$wind - x$) {Append into temporary regis-
         ter file at $wind - x$ index}
24:     end if
25:     if free_slots_available() = TRUE then
26:       $slt \Leftarrow$ get_next_free_slot
```

```
27:         $data\_value \Rightarrow$ VLPC($slt$)
28:         update_free_slots()
29:       else
30:         $slt \Leftarrow$ get_LFU_index()
31:         $data\_value \Rightarrow$ VLPC($slt$)
32:         update_LFU()
33:       end if
34:       remove($data\_value$,$wind-x$) {Remove padding from temporary register file
          at ($wind-x$) index}
35:       if encoded_control_line = 0 then
36:         $data\_value \Leftarrow$ get_data_from_bus($bus\_buf$,$wind-x$) {Add to register orig-
            inal data of same size after removing padding}
37:       end if
38:       forward_to_UL2($data\_value$)
39:     end if
40:   end while
```

B. Explanation of the Algorithm (break-down) - Architectural-level

1. Encoder algorithm

For the sake of brevity and modularity we subdivide our discussion into four parts, initialization, data extraction, tag comparison and code generation/populating CAM. They are explained in detail as follows:

a. Initialization

When the system is reset or the reset interrupt is triggered, all the scratchpad registers are zeroed and the CAM is cleared at both the ends. After the reset, the window size for the sliding window ($wind$ in the above algorithm) can be set. Here we set the window size $wind \in [1, 4]$ due to the following reasons,

- To customize sampling interval for different applications.
- Also we cater to pattern sizes from 1 to 4 so as to take care of single 32-bit frequent values as well as frequent data patterns (here, consecutive 32-bit words) of size 2 to 4.
- Exceeding a window size of 4 (i.e. $wind > 4$) is not realistic for most applications.
- Also, for $wind > 4$, CAM sizes will be very large.
- It also allows us the liberty to control the shutting-off of some of the CAM banks which leads to reduced power usage of the CAM lines.

Also, we initialize the scratchpad registers which temporarily hold the data from the memory buffer as the window slides by $wind$ after each processing step. In the algorithm Alg:1, it is referred to as $mbuf$. Also we keep track of the next free slot available in a yet non-full CAM using $free\_slots$ while for a full CAM we keep of the next pattern to be replaced using LFU_index, for which we use a modified LFU strategy which is detailed in Alg. 3.

b. Data extraction

Each processing step begins with copying the data of size $wind * size(word)$ from the memory buffer into the scratchpad register. We also store the address of the head of the memory buffer so as to slide the window according to $mbuf + wind * size(word)$.

c. Tag comparison

After obtaining the current value of the memory buffer, the control unit on the positive clock edge raises the RE line, at which point the tag matching is done on the 1st bank, which is a 64-entry 32-bit CAM, as shown in Figure 8. After the 1st bank comparison is done, if there is a match on any line (i.e. one of the match-lines is high), then only the next PC_bar and RE signal (precharge) will be asserted, so as to not precharge all the banks if there is miss in the first bank itself. Thus the matchline ML in $bank_{i-1}$ acts as trigger for PC_bar (active low) to begin precharging $bank_i$ for $i \in [2, 4]$. We can call this *bank-precharging* strategy. The drawback of this strategy is an increase in delay between the banks but this is only in the case of a full match. In case of prefix matches the delay is much less.

d. Code generation/populating CAM

When code has to be generated for incoming data we have two cases, CAM miss and match respectively. Each case is explained below:

- *Code generation*:Whenever there is match we generate a 2-bit code corresponding to index of the match and encoded line is raised to 1. The encoded line set to

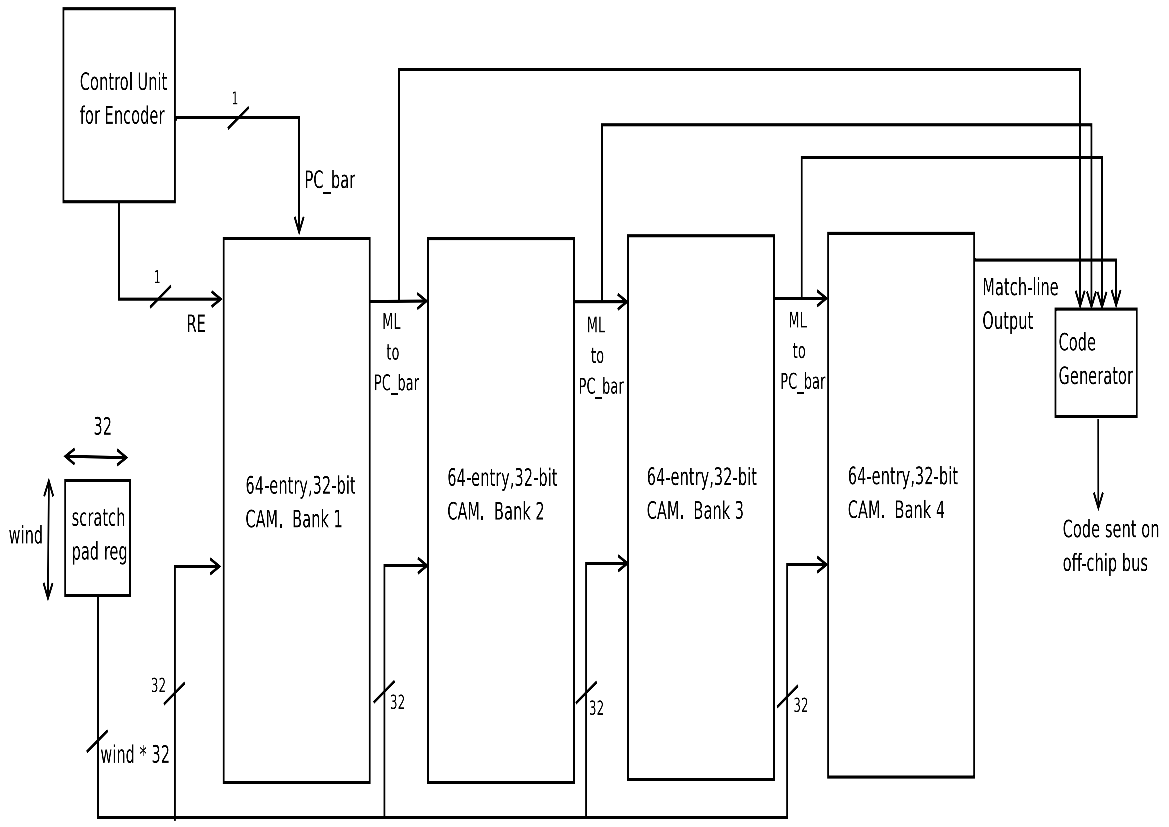

Fig. 8. Architectural-level representation of the encoder

1 or 0 suggests if the data on the bus is coded or unencoded, respectively. The code is generated by the code generator block as shown in Figure 8. In case of a prefix match, if the prefix is found in the CAM then we send the index of the prefix match, with encoded line raised to 1. Here we generate a 2-bit code using positional notation so as to indicate the index of the 64-entry CAM. Also for the rest of unmatched data of size $wind-x$ where x is the size of the matched prefix, we send the data as is with the encoded line lowered to 0. If matched prefix is not found in the CAM then we send the prefix with encoded line as 0. Also, we send padding (zero-valued) of size, $x = wind - size(matched\_prefix)$ with encoded line as 1. This is an indication to the decoder to insert the matched

prefix in its cache. Rest of the unmatched data is sent as is with encoded line as 0. We use a zero-valued padding as we never insert zeroed original data into our CAM/cache.

- *Populating CAM*: In case of a full miss we send the original data as is with encoded line lowered to 0. Also we populate the CAM by direct write to the CAM in case of a non-full CAM using $free\_slots$ as seen in Alg:1 while in the event of a full CAM we replace the least frequent used slot, LFU_index. In case of prefix match where the prefix itself is not present in the CAM, we place the prefix in the CAM, similar to that described previously.

The modified LFU strategy is given below:

Algorithm 3 modified LFU strategy for CAM slot replacement

```
INPUT: CAM_count, previous_index
OUTPUT: $LFU_index$
index ⇐ previous_index
len ⇐ size(CAM_count)
i ⇐ modulo(index,len)+1
count ⇐ 1
while count ≤ len do
    if CAM_count(i) ¡ CAM_count(index) then
        index ⇐ i
    end if
    i ⇐ modulo(i,len) + 1
    count ⇐ count+1
end while
```

```
RETURN index
```

2. Decoding algorithm

The decoding algorithm is explained in the following two stages, initalization and data decoding/populating cache:

a. Initialization

Similar to the encoding algorithm, initialize the scratchpad registers which temporarily hold the data from the bus as the window slides by *wind* after each processing step. In the algorithm Alg:2, it is referred to as bus_buf. Also we keep track of the next free slot available in a yet non-full CAM using $free\_slots$ while for a full CAM we keep of the next pattern to be replaced using LFU_index, for which we use the modified LFU strategy given in Alg:3.

b. Data decoding/populating cache

During the decoding phase when the data is received over the bus there can be two cases corresponding to whether the encoded line is 1 or 0.

- *Unencoded*: In case of a 0 on the encoded line we populate the cache as discussed in the previous subsection. If the encoded line is 0 and then transitions to 1 for the sampled window of data, then we insert the data in the cache. This data consists of the matched prefix at the encoder alongwith the padding. Now, we remove the padding and resample data of size $x = size(padding)$ from the bus and place in scratchpad register. Forward this data to the L2 cache.
- *Coded*: In case of a 1 we use the data on the bus, as an index into the cache and get the corresponding original data value which is forwarded to the Unified L2

cache. In case of a prefix match we still get the corresponding original data value from the cache, but for size $wind - x$ where x is the size of the data obtained from cache, forward the original data from the bus scratchpad register.

The architectural-level representation of the decoder is give in Figure 9.

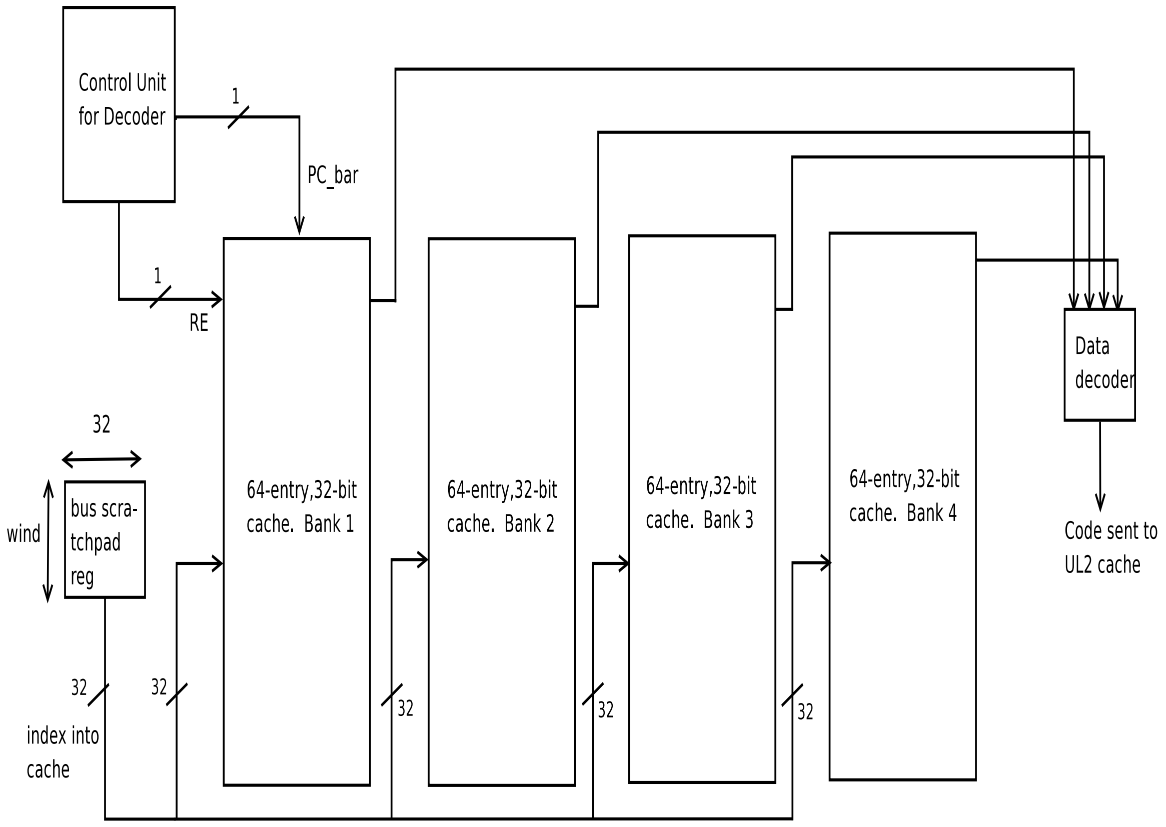

Fig. 9. Architectural-level representation of the decoder

C. Low-level Design

The encoder end uses a CAM to store the incoming data patterns. Matching the incoming data pattern value with the CAM banks can be pipelined with the data extraction from the memory buffer. Also, in case of a miss, storage of the value in the CAM can be done in parallel with sending the value on the bus. Parallelism can also be achieved during the match, in the case of sending the encoded value on the bus as well as updation of the LFU counters.

Our encoder uses 4 banks where each is an 64-entry 32-bit CAM bank shown in Figure 10. Here each building block is a 4-entry 4-bit CAM block which is show in Figure 11.

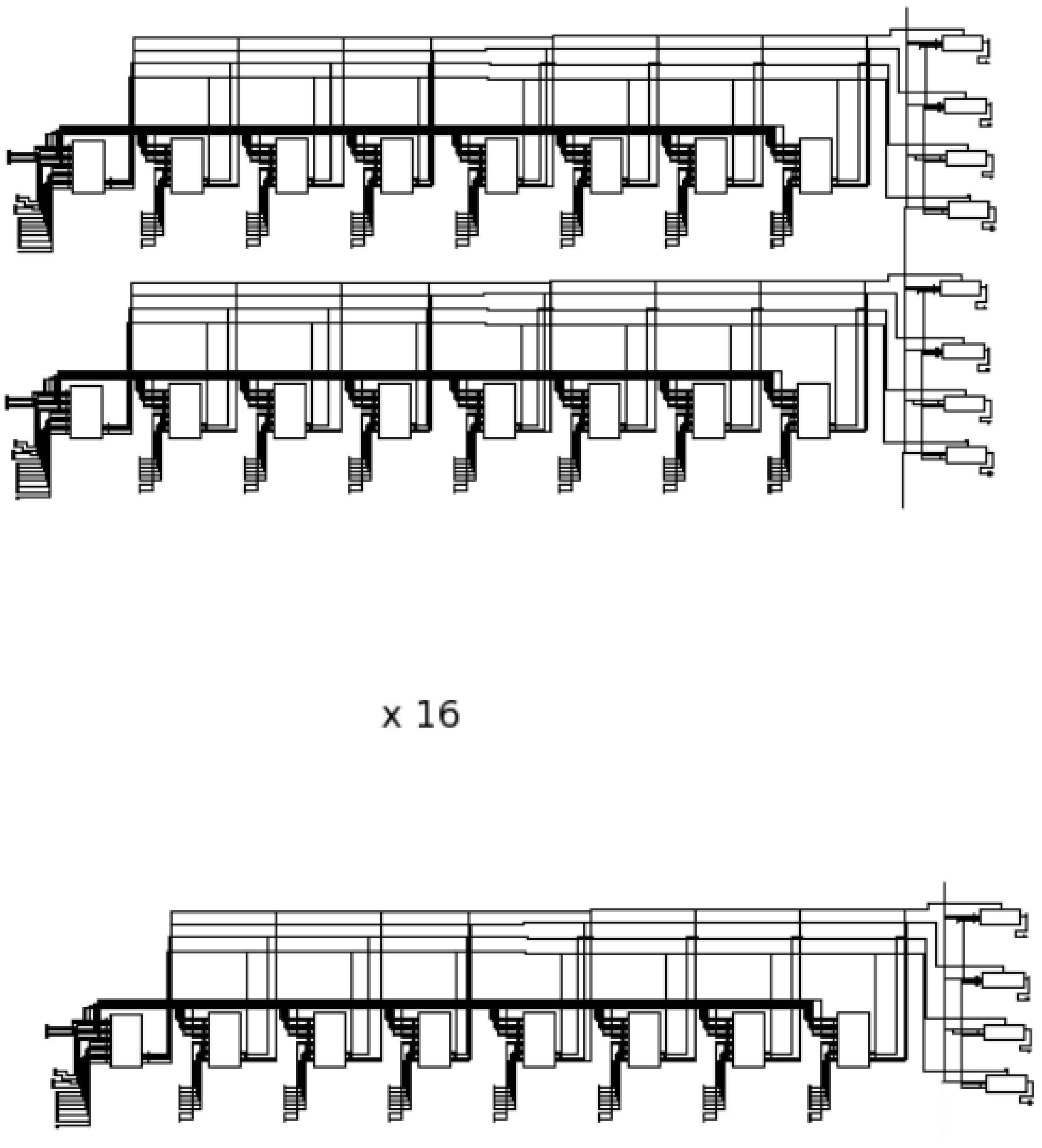

Fig. 10. One 64-entry one word(32-bit) CAM array, with each small block being a 4-entry 4-bit CAM building block.

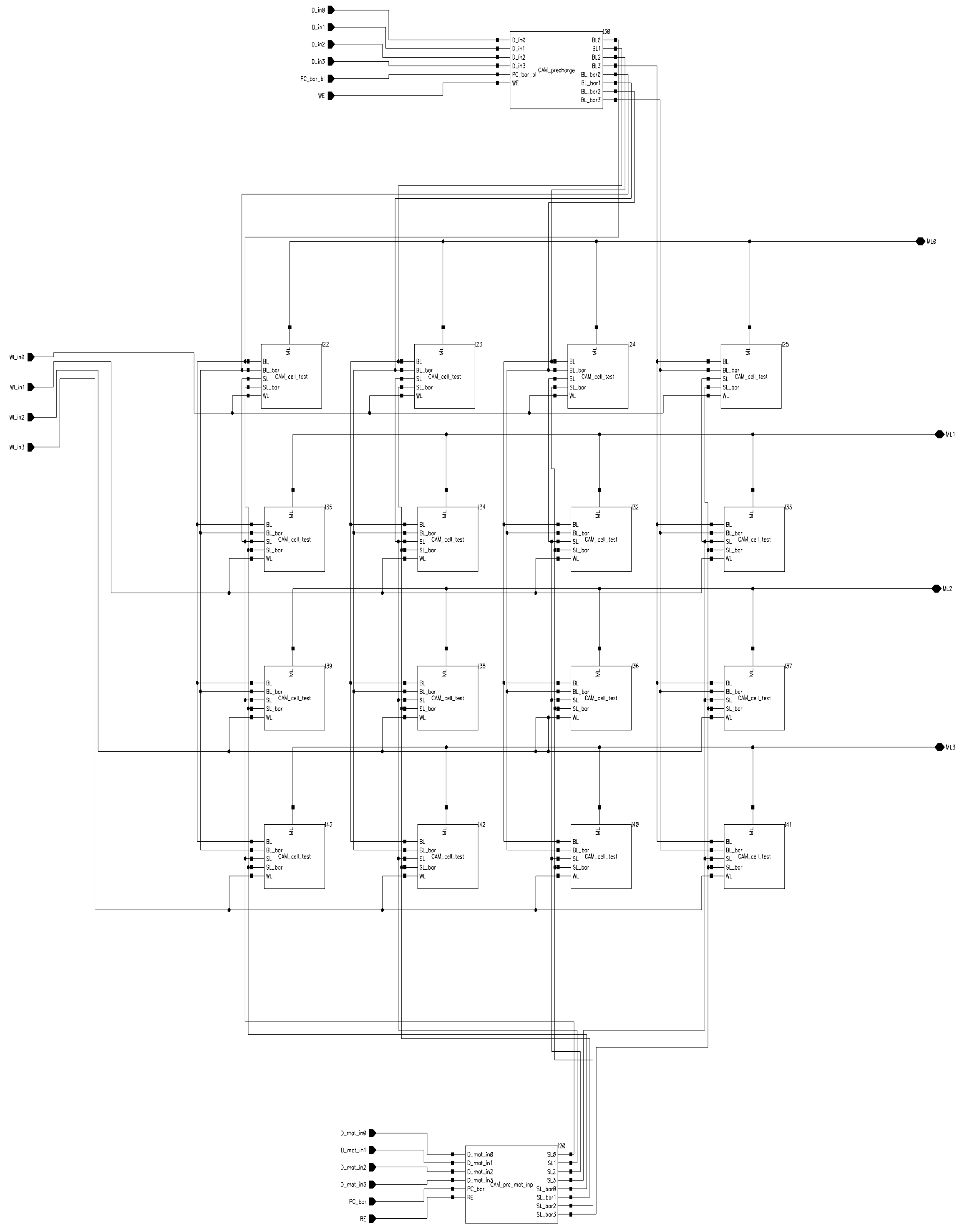

Fig. 11. A 4-entry 4-bit CAM array building block.

The main component of a 4-entry 4-bit CAM is a 9 transistor CAM cell shown in Figure 12. In a 9-T CAM cell write process is similar to that in a 6-T SRAM cell. During the matching process, SL and SL_bar lines are precharged high. When RE line is asserted, if data match input D_mat_in0 in Figure 13 is 0, SL is driven low, else if D_mat_in0 is 1, SL_bar is driven low. If data in a CAM cell is 1 and SL is 1, then there is a match and ML remains high during evaluation. For more detailed explanation of content addressable memories please refer [23][22][24][25][26].

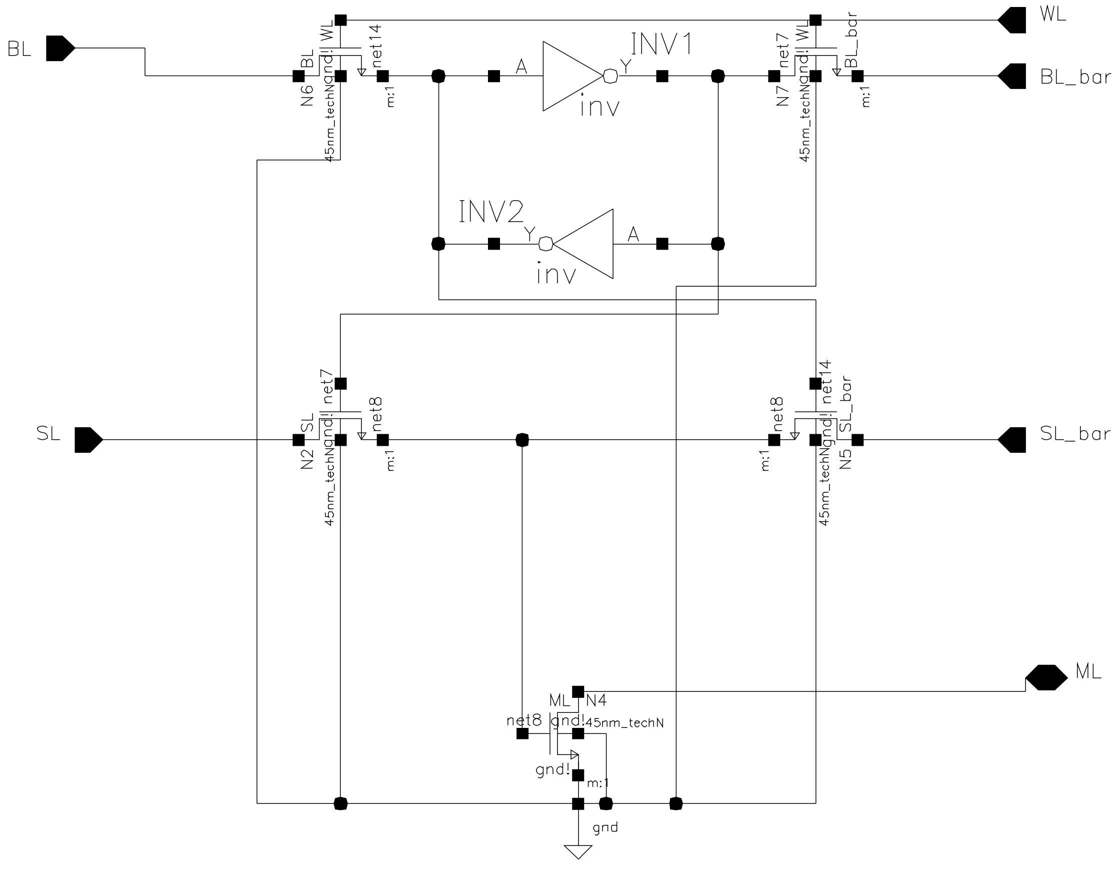

Fig. 12. A single 9-T CAM cell.

Other components that make up a 4-entry 4-bit CAM are a data match input circuit shown in Figure 13 and data write input circuit shown in Figure 14. The data match input circuit uses data inputs for matching, D_mat_in, to drive SL or SL_bar low whenever a RE signal is asserted. On the other hand, the data write input circuit uses data inputs for storing, D_in, to drive BL or BL_bar low whenever a WE signal is asserted.

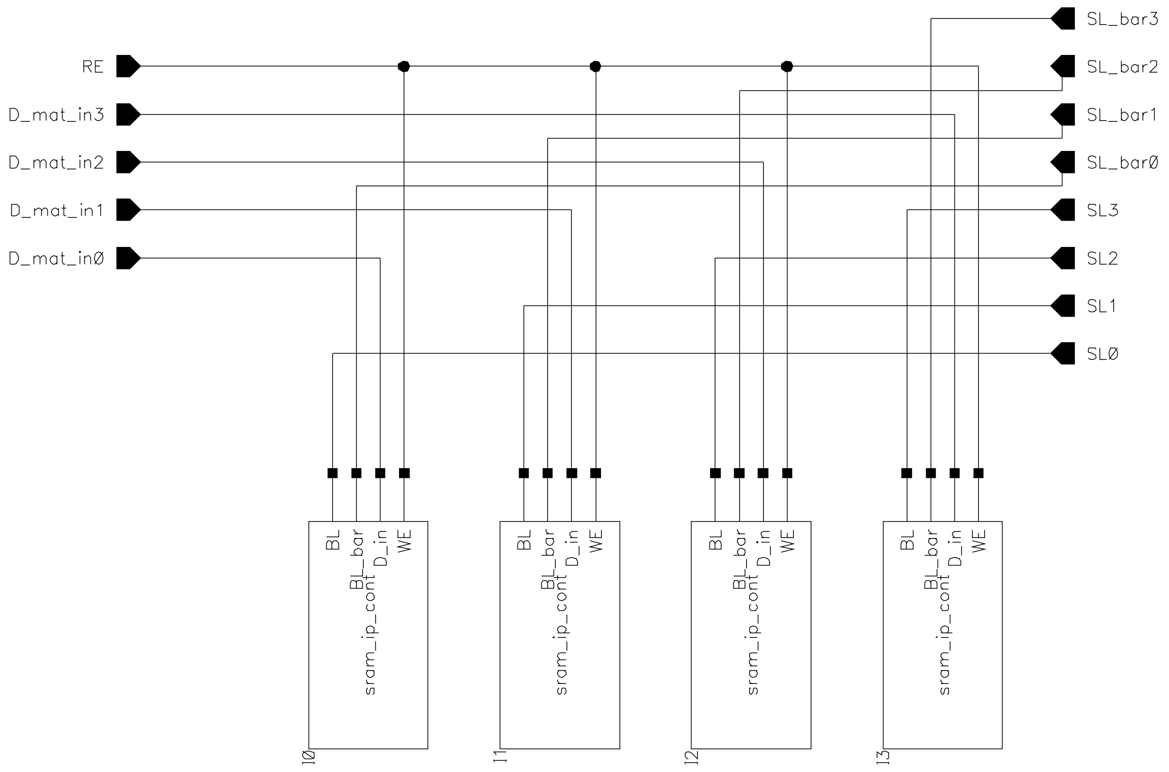

Fig. 13. Input control for data used for CAM tag matching.

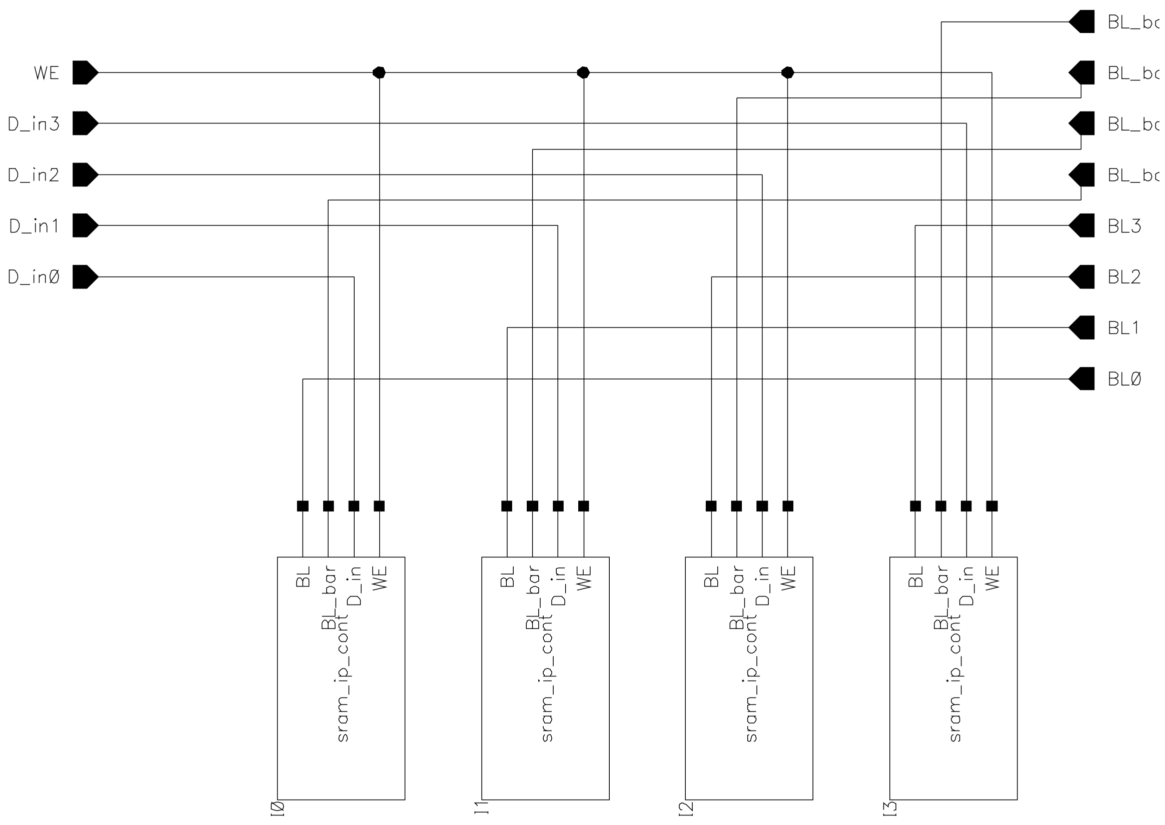

Fig. 14. Input control for writing data to the CAM array.

The input control for CAM and SRAM write/match circuits is shown in Figure 15. For write if WE signal is asserted, depending on whether D_in is 0 or 1, BL or BL_bar is driven low respectively. In case of matching, if RE signal is asserted, depending on whether D_mat_in is 0 or 1, SL or SL_bar is driven low respectively.

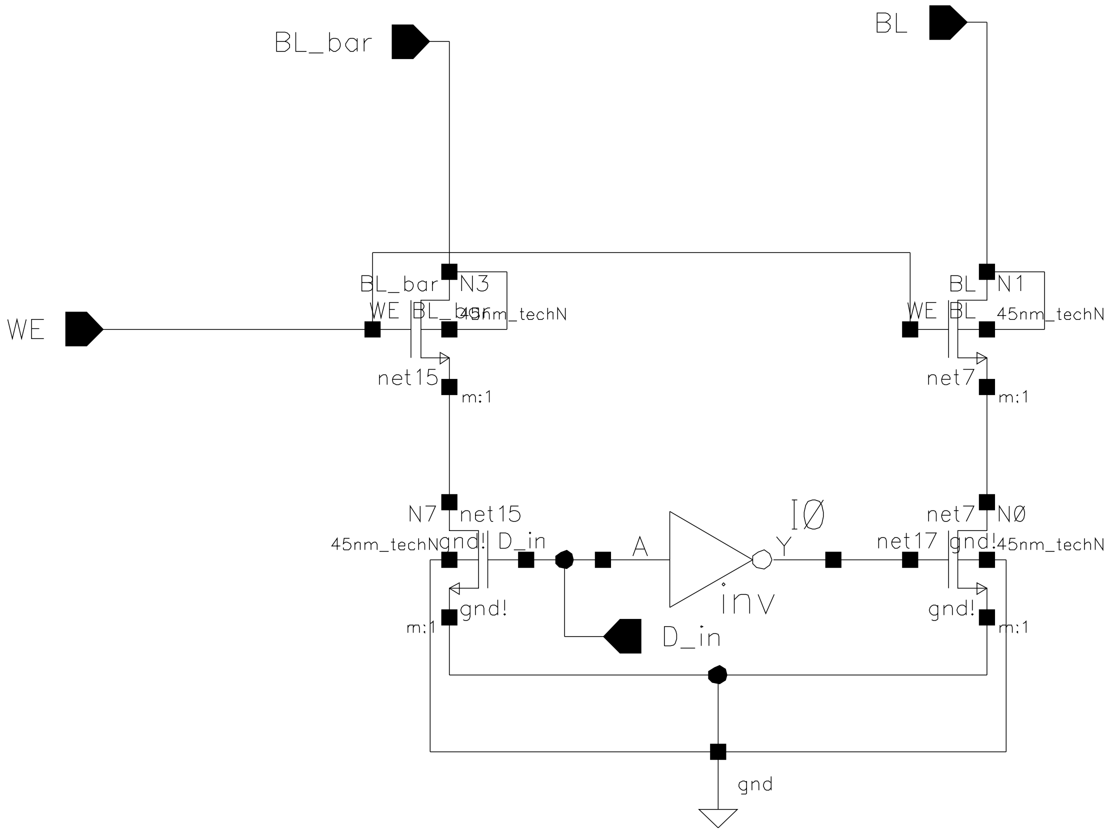

Fig. 15. Input control which serves as a building block for CAM write/match circuits and also SRAM read/write circuits.

Also we have the matchline sense amplifier (MLSA) in Figure 16 which senses the matchlines to see if there is a match on any of the lines and amplifies them. This has to be done due to attenuation of the match signal over the array. Depending on the attenuation and noise margins we are able to sense the match line signal and amplify it correspondingly. In the MLSA circuit, initially the output ML_out is set to zero by driving ML line to the ground. This is done by setting ML_pre to high. During matching ML_pre is set to low and ML is precharged by driving cur_en_bar to low. If there is a match ML remains high thus driving the input A to the inverter I3 low. ML_out hence remains high in case of a match. Similarly in case of a miss, ML_out remains low.

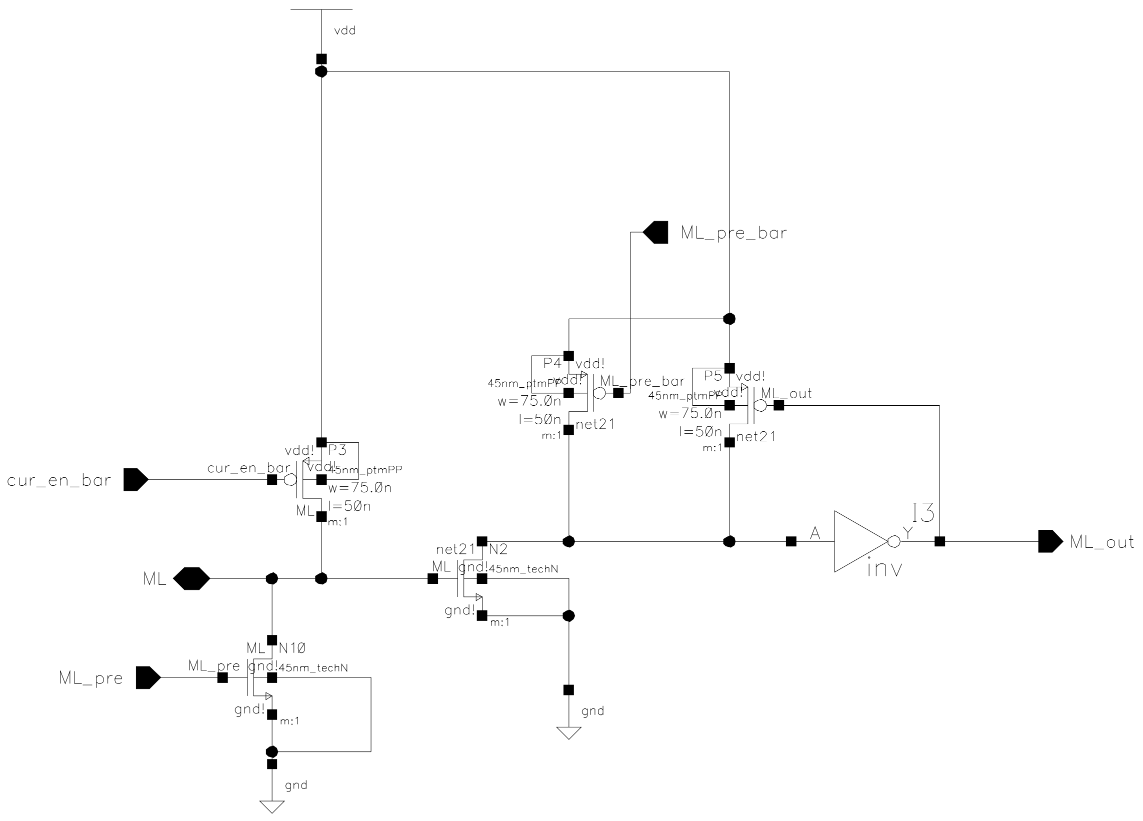

Fig. 16. Circuit for the match-line sense amplifier

Next we have the SRAM array shown in Figure 17, and similar to the description for CAM, we have the 6 transistor SRAM cell shown in Figure 18. The sense amplifier for the sram is different from that of a matchline sense amplifier as the sram sense amplifier is used for column sensing and amplification while that for a CAM is used for row matchline sensing. Hence, given an address we read the value from the SRAM, while in a CAM we perform a parallel search for a given data value. The SRAM sense amplifier is shown in Figure 19. The input control for the CAM and the SRAM is shown in Figure 15. It handles the precharging of each line BL and BL_bar in case of an SRAM and also the SL and SL_bar lines for precharging them for the purpose of matching in case of a CAM.

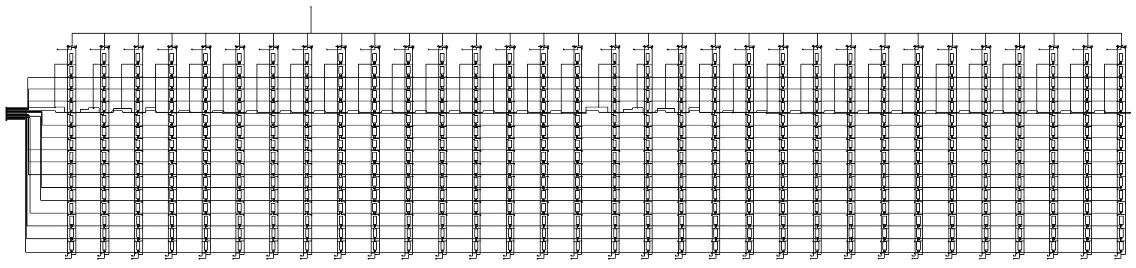

Fig. 17. One 64-entry one word(32-bit) SRAM array, with each block being a 4-entry 1-bit column array

D. Summary

In this chapter we looked into at the most important part of this thesis, the *variable length pattern encoding* algorithm with a detailed explanation of the various components of the algorithm using comprehensive set of flow charts, pseudo-code and diagrams (both architectural-level and circuit-level). With flow-charts and algorithms in sections A, we clearly differentiate our scheme from those of [1][2]. The architectural-level and circuit-level diagrams of the Encoder and Decoder with their corresponding

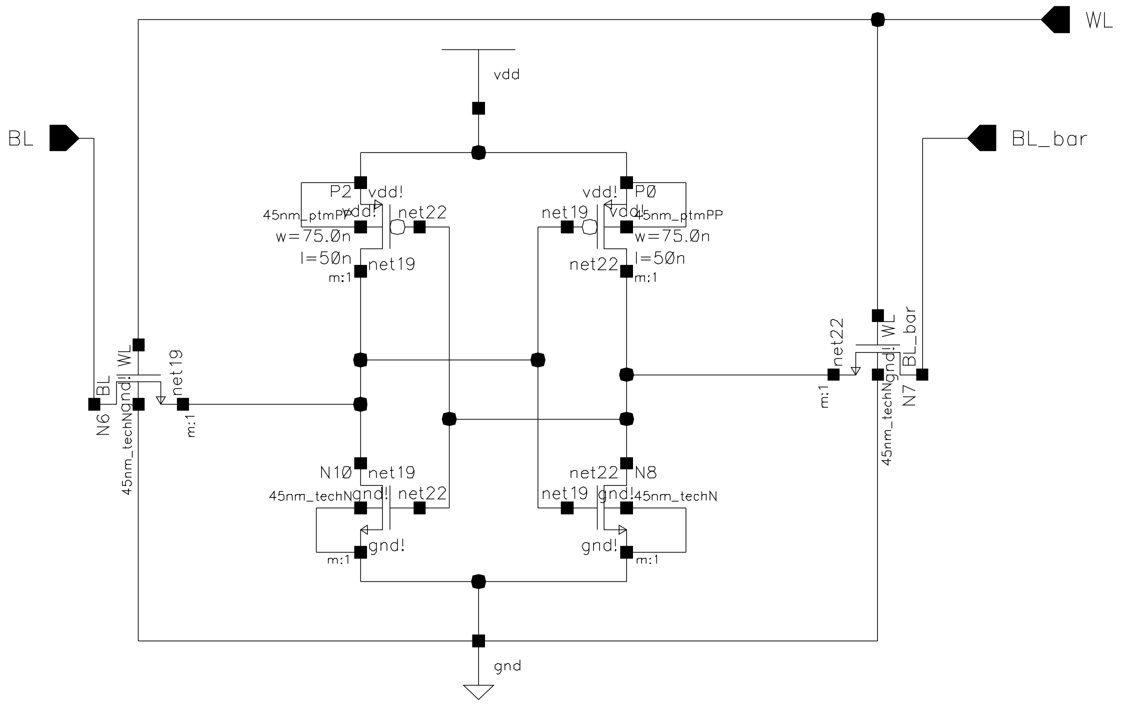

Fig. 18. A single 6-T SRAM cell

explanations in section B and section C respectively, give an insight into the inner workings of algorithm thus serving to be the backbone of our scheme. We obtain the power and latency for the CAM and SRAM cache from Spectre simulations. These and other comparison results are discussed at length in the next chapter.

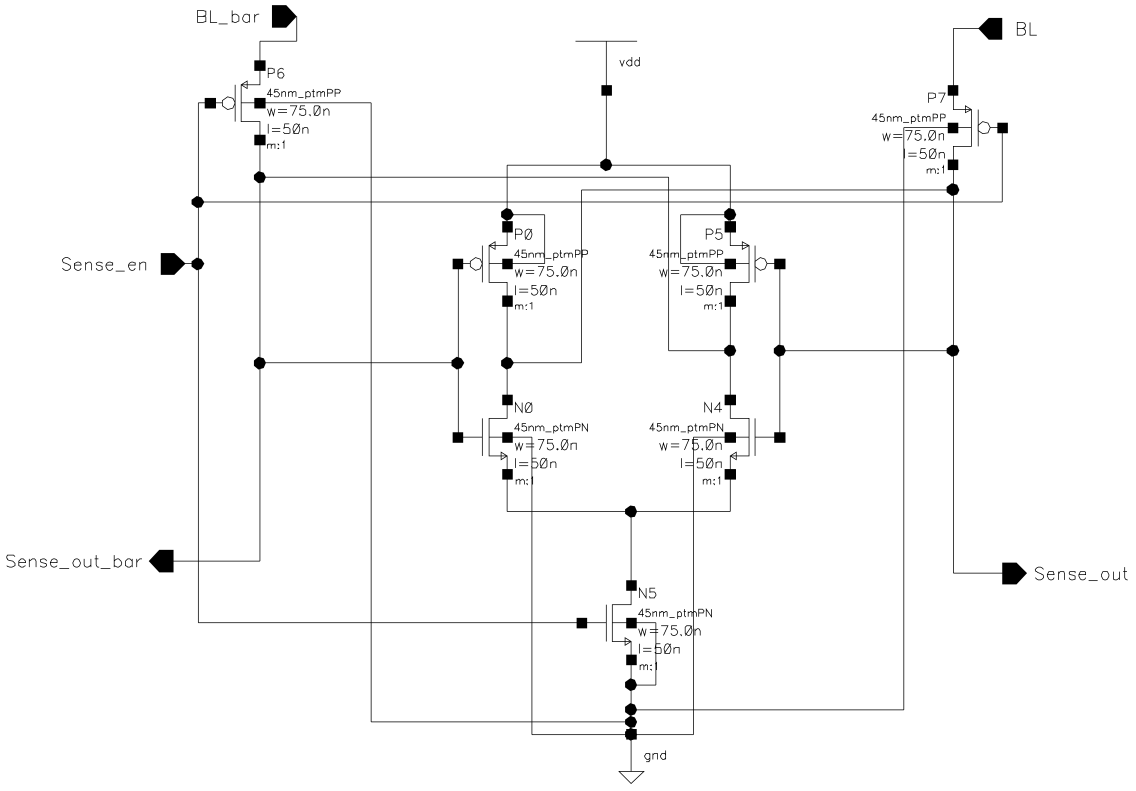

Fig. 19. Circuit for the SRAM sense amplifier

CHAPTER V

EXPERIMENTS AND RESULTS

In this chapter we present a thorough comparison of VLPC with the popular methods, FVE, FVexor [1] encoding and the VALVE [2] encoding schemes, which have been discussed in detail in Chapter II. We divide this chapter into two major sections, A) experiments and B) results & discussion, where we discuss the experimental details including the simulation parameters and configurations with the summary to follow in section C.

A. Experiments

We modified the Simplescalar out-of-order simulator, sim-outorder to obtain data traces for the various SPEC2000 benchmarks. The simulator was also modified to incorporate various encoding schemes in consideration and obtain data traces for each different scheme. The schemes that we compare against are Frequent Value encoding (FVE) [1], FVE with Xor'ing scheme applied [1] and the VALVE [2] encoding scheme . The flow chart in Figure 20 shows the experimental flow for the work performed. As shown in the table 4, we run the simulations for 10M instructions and record the data traces for each of the encoding schemes that we have incorporated into simplescalar [21]. These data traces are then given as inputs to a MATLAB m-file for analysis to calculate overall bit transitions for 10M instructions and percentage reduction in bus energy consumed from the unencoded.

The basic experimental configuration for the simplescalar [21] out-of-order simulator is given in the table 4.

Here we have performed simulations for eight different SPEC CPU2000 [27] benchmarks whose descriptions have been given in Table 5.

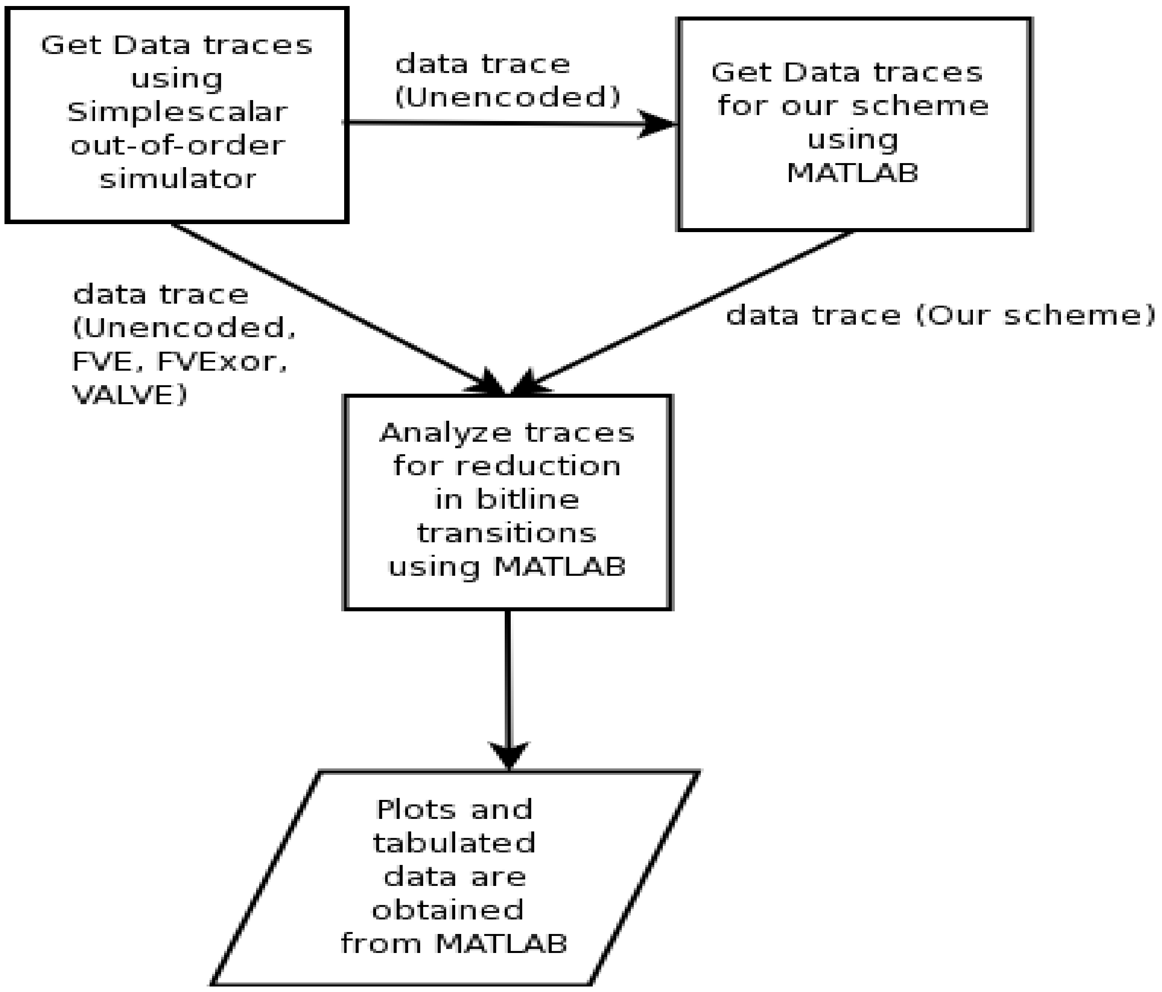
Get Data traces using Simplescalar out-of-order simulator
data trace (Unencoded)
Get Data traces for our scheme using MATLAB
data trace (Unencoded, FVE, FVExor, VALVE)
data trace (Our scheme)
Analyze traces for reduction in bitline transitions using MATLAB
Plots and tabulated data are obtained from MATLAB

Fig. 20. Experimental flow

B. Results and Discussion

1. Architectural-level simulations

The simplescalar [21] simulation parameters have been discussed in detail in the previous section. Here we answer the question as to how observing the patterns of data values spaced out over time is helpful in reducing the transitions between

Table 4. Experimental configuration

L1 Data Cache	128KB
L2 Unified Cache	1MB
L1 Instruction Cache	128KB
Instruction TLB	1MB
Data TLB	1MB
Memory-CPU bus width	4 bytes
Maximum instructions to Execute	10M

consecutively sent values. As shown in the Figure 21, with a window size of 4 we obtain a high percentage of patterns which can be encoded. The results for the benchmarks, ammp, gap, bzip2, crafty and mcf are of significance here.

In the figure Figure 22 it can be seen that our results clearly agree with that of the Figure 21, thus verifying our assumptions. It can also be seen that in most of the cases, namely crafty, swim, mcf, applu, ammp, and gap our algorithm performs better than all the other algorithms, FVE, FVExor [1] and VALVE [2] compared here. The exceptional cases are for benchmarks lucas and bzip2. As seen in Figure 21 lucas has very low encodable fraction of the overall data, hence the behavior is anticipated. For bzip2, although the fraction of encodable data is almost 32%, but for the 10M instructions executed in simplescalar [21] the cache miss rate is 0.0008 for the dl1 cache and 0.5 for the ul2 cache, which shows that the off-chip data bus transactions are very low. Also, dtlb miss rate is 0.0002, which is very low, thus substantiating the above statements. Here we also explain why our algorithm is better than the algorithms we compare against:

Table 5. Benchmarks and their descriptions

Benchmarks	Description
bzip2	A standard compression/decompression algorithm found on most UNIX-based machines. To isolate work done to the CPU and Memory, the compression/decompression entirely takes place in memory.
crafty	A high performance computer chess program, designed around a 64-bit word. It is primarily an integer benchmark which uses significant number of logical operations.
mcf	A combinatorial optimization program written in C for single-depot vehicle scheduling in mass transportation. Uses integer arithmetic.
gap	Implements a language and library to do group computation such as finite fields, permutation groups, lattice computation, in C.
ammp	A computational chemistry program which models molecular dynamics on a protein-inhibitor complex embedded in water. It is a floating-point benchmark written in C.
applu	It is program to find solution of five coupled non-linear Partial Differential Equations on a 3-d logical grid structure. It is written in Fortran and is a floating-point benchmark.
swim	This program does weather prediction by using shallow-water equation models. Highly computationally-intensive floating-point benchmark written in fortran.
lucas	This program is designed to perform primality testing of Mersenne numbers, using arbitrary-precision arithmetic. A floating-point benchmark written in fortran 90

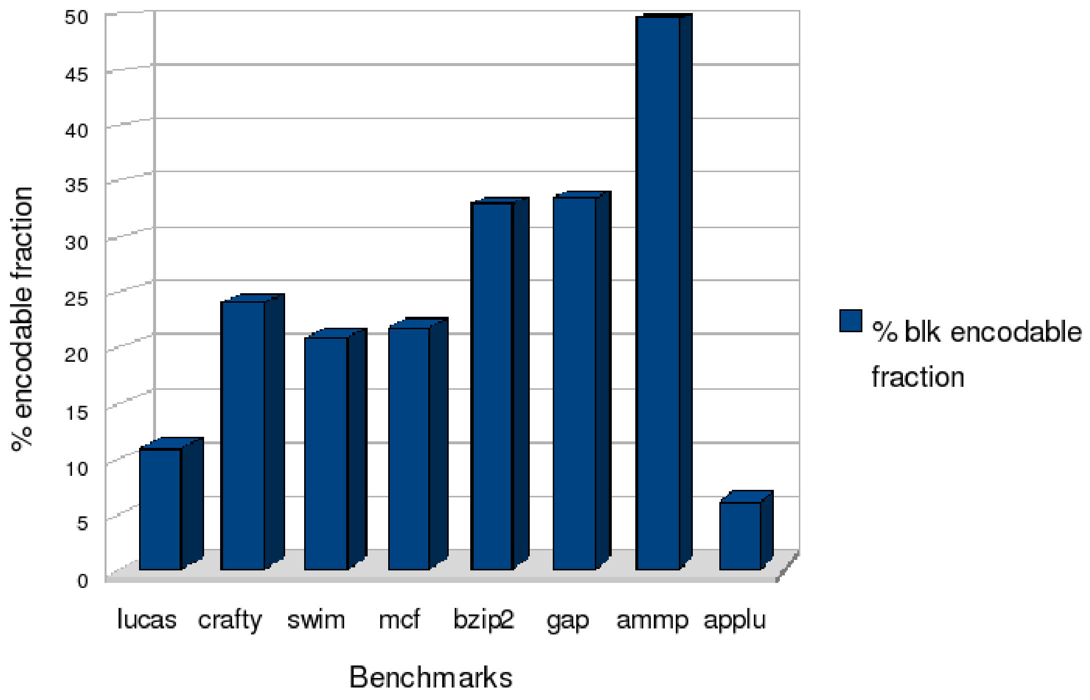

Fig. 21. Fraction of *encodable patterns* obtained from data trace

- the encoding schemes FVE, FVExor and VALVE are not able to take advantage of m*32-bit consecutive patterns that occur over time. Here $m \in [1, 4]$, as explained in the previous chapter.

- In VALVE [2], encoding is done either per 32-bit sequence [1] or for partial data in 32-bit value (24, 16-bit) [2].

- Due to the above stated drawbacks there is inevitable bit switching in either cases.

- Also, as we implement the CAM at the encoder as bank-precharged one wherein precharging for $(i + 1)^{th}$ 64-entry 32-bit bank occurs only if there is a match at the i^{th} bank. This leads to reduced power consumption in case of prefix pattern matches. Also, as we use a window size of 4 with a maximum overlap of

1 between consecutive sliding windows, we reduce redundant tag comparisons thus saving power.

- The encoding scheme in [2] suffers from the fact that for every 32-bit data value there is a comparison done in all the tables (32-bit table, 24-bit table, 16-bit table), thus increasing energy consumption.

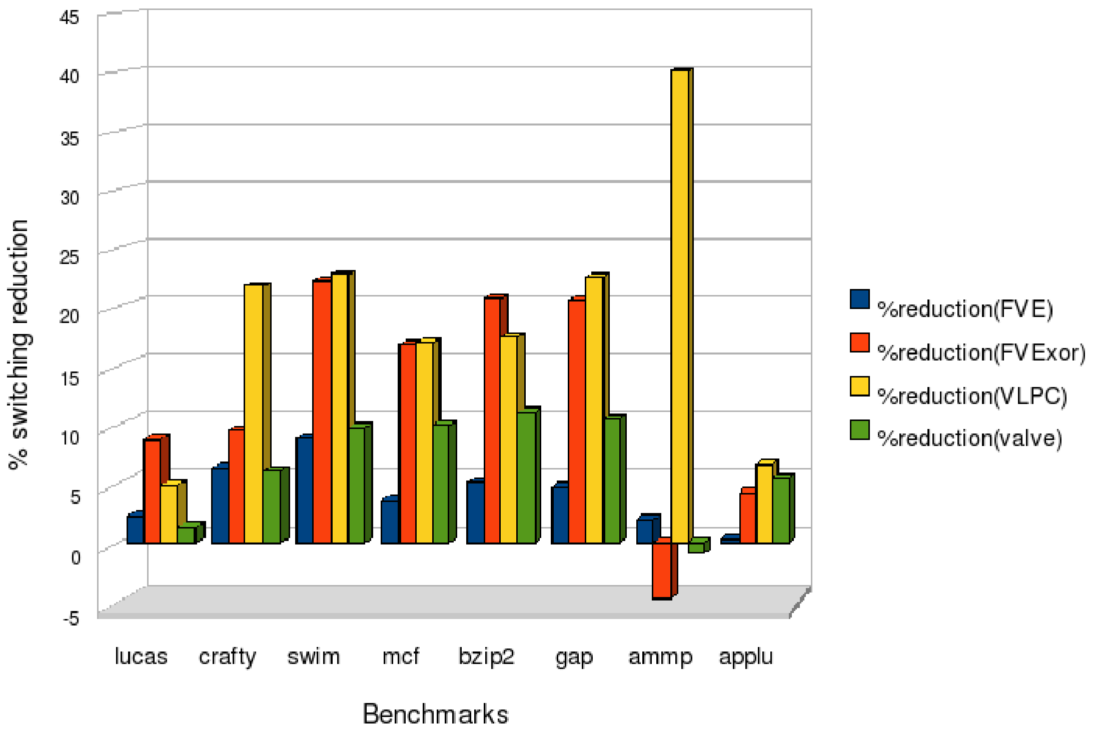

Fig. 22. Energy reduction(percent) comparison of FVE, FVExor [1], VALVE encoder [2] and our VLPC encoder

The exact figures from the analysis of the data traces in MATLAB is given in tables 6 and 7. These tables depict the number of observed encodable data values and percentage (%) reduction in Energy for different benchmarks over 10M instructions executed in the out-of-order simplescalar [21] simulator (sim-outorder).

Table 6. Number of encodable patterns (1 to 4 consecutive 32-bit words) obtained from analysis of data traces in MATLAB

Benchmark	Overall	blks_2_encode	percentage blk encodable fraction
lucas	30146	3299	10.94
crafty	20387	4927	24.17
swim	63759	13353	20.94
mcf	7683	1680	21.87
bzip2	3778	1244	32.93
gap	9138	3066	33.55
ammp	1833106	911864	49.74
applu	237239	14433	6.08

Table 7. Percentage reduction in bit transitions obtained from analysis of data traces in MATLAB

Benchmark	percentage reduction (FVE)	percentage reduction (FVExor)	percentage reduction (ours)	percentage reduction (valve)
lucas	2.3	8.79	4.88	1.24
crafty	6.3	9.52	21.81	6.07
swim	8.82	22.2	22.79	9.74
mcf	3.54	16.86	17.07	10.01
bzip2	5.12	20.81	17.53	11.08
gap	4.66	20.6	22.57	10.52
ammp	1.92	-4.68	40.11	-0.91
applu	0.26	4.17	6.64	5.41

It is important to measure the performance trade-off alongwith power. Cache misses are the main source of traffic on off-chip buses. Data transfer occurs mainly as 64-byte blocks which is the size of a cache block. Hence latency for transferring 32-bit data is mainly divided into 2 components, latency for tranferring the first 32-bit data value (chunk) and latency for transferring subsequent chunks. Thus, latency for our approach can be given by:

$$Lat_{first\_chunk} = Lat_{mem} + Lat_{bus} + Lat_{enc} + Lat_{dec}$$

$$Lat_{next\_chunk} = Lat_{bus} + max(Lat_{enc}, Lat_{dec})$$

Here, $num\_chunks = cache\_blk\_size/mem\_bus\_width$. This latency is measured for all SPEC2000 benchmarks for executing 10M instructions. Latency comparison with the approaches in consideration (FVE, FVExor and VALVE) is shown in Figure 23. It clearly shows that our performance suffers in benchmarks such as lucas, bzip2, swim due to lack of sufficient encodable patterns found in the data trace as confirmed by Figure 21. Lack of patterns causes large fraction of data being sent unencoded thus increasing latency in our case due to the waiting time associated with each window of data. If there is higher percentage of encodable patterns then there is decrease in latency as seen for benchmarks, crafty, ammp. FVE encoding has a better performance due to simplicity of operation while FVExor has a higher latency due correlator and decorrelator circuit added to the encoder and decoder respectively.

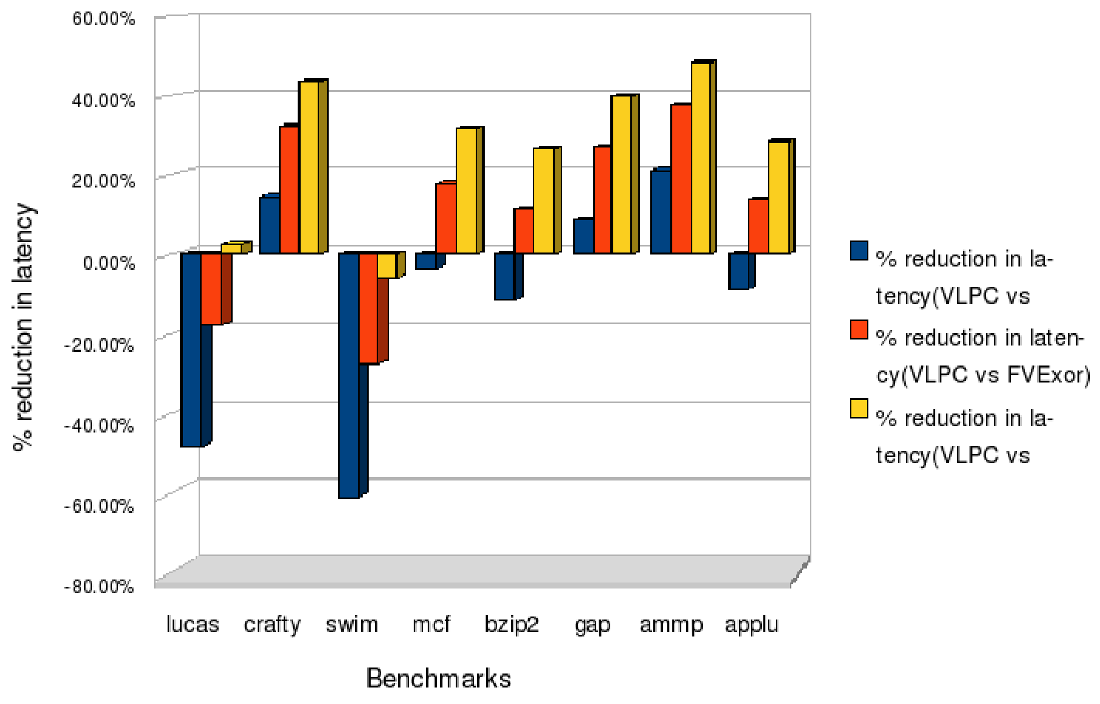

Fig. 23. Latency reduction(percent) comparison of FVE, FVExor [1], VALVE encoder [2] and our VLPC encoder

2. Low-level simulations

Some of the plots that show correctness of the operation of the Content Addressable Memory designed by us are given in Figure 24, 25 26. The first plot 24, shows the matchline output ML_out_i where $i \in [0, 3]$. Here for a match on the 4th line the matchline ML_out_3 has been asserted. This triggers the precharge of the next CAM bank. Rest of the matchlines 0 to 2 are zero, which means there is a miss on those lines. If there is a miss in the first bank then the precharge for the next bank PC_bar_{i+1} is not pulled down to active low. In this case the rest of banks are not precharged, thus saving power.

Data inputs for some of the 32-bit data lines are shown in Figure 25. Also, the corresponding plot for some 32-bit data lines to be matched are shown in Figure 26.

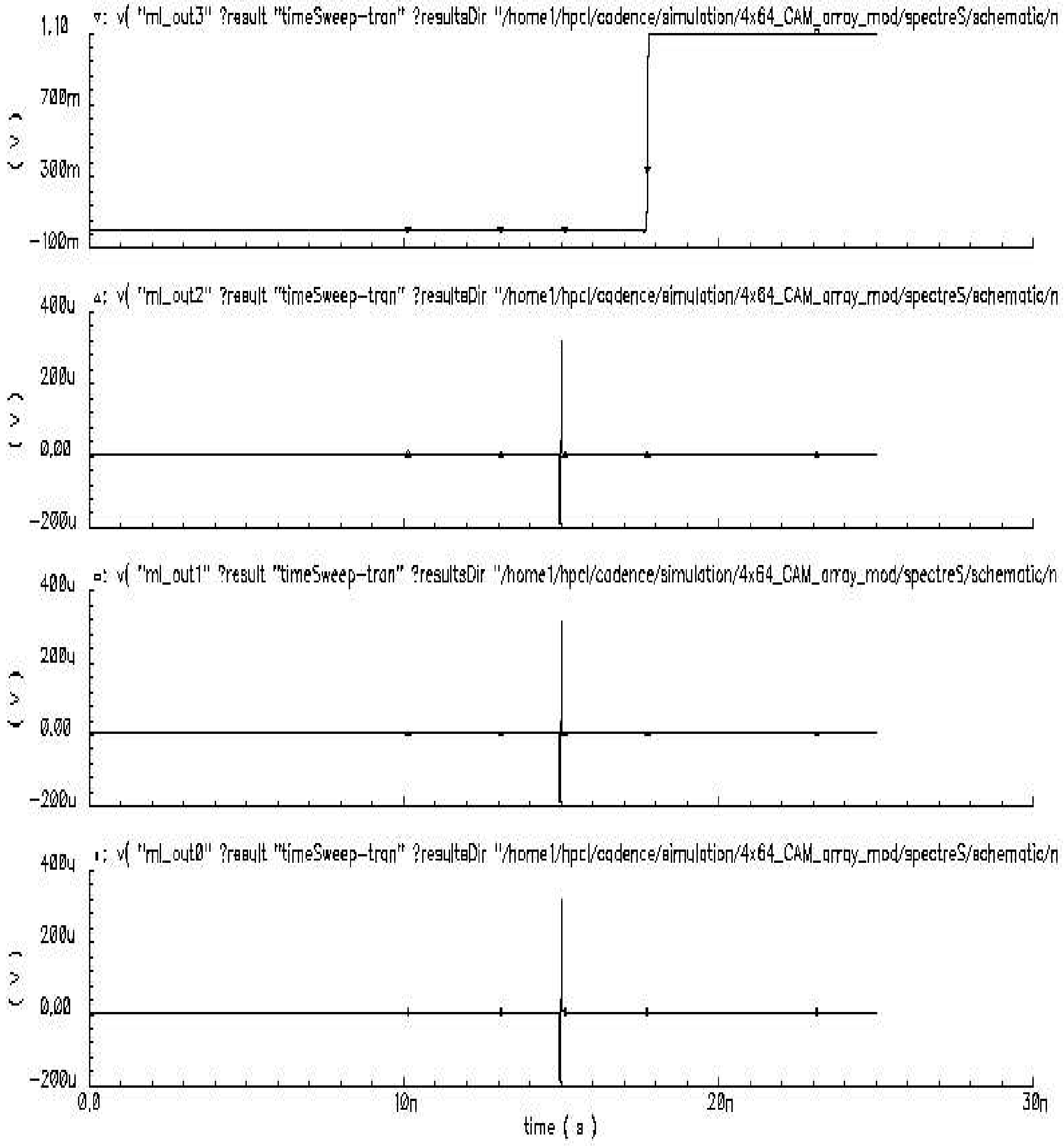

Fig. 24. The matchline output when the data matches with 4th CAM line

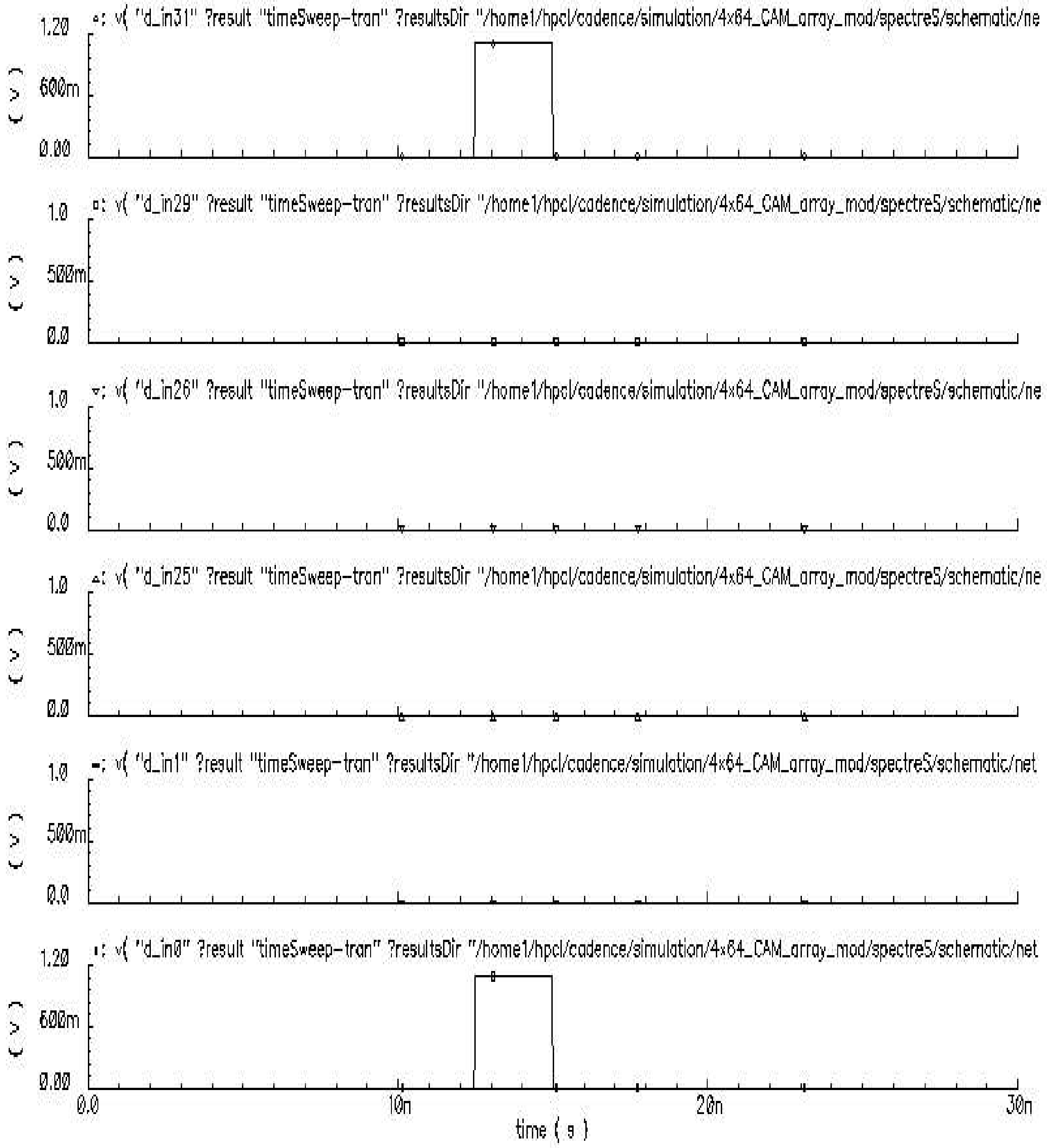

Fig. 25. The data written to the 64-entry CAM

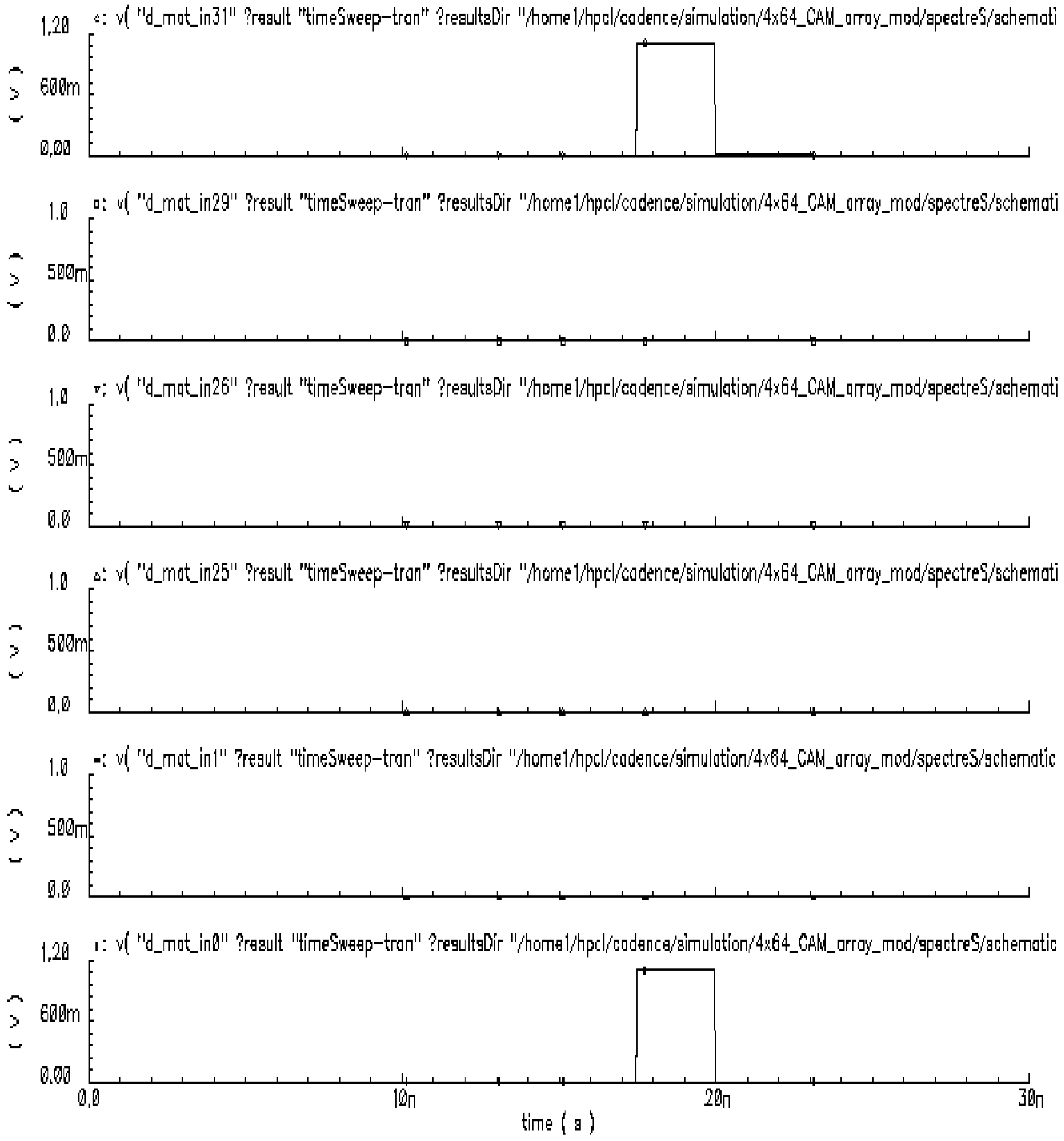

Fig. 26. The data to be matched with that of the 64-entry CAM

C. Summary

In this section we have discussed the experiments that we have performed concerning our VLPC algorithm that we discussed in detail in the previous chapter. We performed simulations in simplescalar [21] for different benchmarks and incorporated the algorithms, FVE, FVExor [1] and VALVE [2] so as to obtain the corresponding data traces. The data traces were analyzed in MATLAB to obtain the comparison results with respect to the overall bit transitions and corresponding reduction in energy consumption of the off-chip data bus. The results have been detailed along with their explanations as to why our algorithm performs better than the other schemes in light of the given configuration parameters.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

A. Conclusions

It is imperative to reduce power consumption at the off-chip level due to its impact on overall system power. In this research we have developed a new encoding/decoding strategy (VLPC) to reduce power in off-chip data buses. We have demonstrated that this method is superior to existing popular schemes such as FVE [1], FVExor [1] and VALVE [2].

To summarize, we believe we have made some significant contributions to the problem of power reduction in off-chip data buses. We highlighted significant drawbacks in the previous schemes FVE, FVExor and VALVE, such as, small size of the Value Cache (VC) at both encoder/decoder ends, ineffective utilization of frequent values in the data stream, decreased power savings in case of non-consecutively coded values, and excessive data comparison at the encoder due to fixed partial matching (16, 24, 32-bit) [2]. To alleviate these drawbacks we developed a VLPC scheme which reduces bit transitions by 4.88% to 40.11% when compared to the Unencoded scheme for various SPEC2000 benchmarks. Also, bit transitions are 0.3% to 38.9% lesser than other popular schemes, FVE, FVExor and VALVE. Additionally, we demonstrated the effective usage of bank-precharged CAM circuits for our maximal prefix matching strategy.

B. Future Work

There is scope for improvement in our proposed scheme of encoding. For example, experimenting with multimedia benchmarks would further ensure that our scheme is highly effective in reducing power exclusively for these applications. The bank-precharging strategy could be changed to reduce latency in case of large percentage of full matches. Performing RTL synthesis of the encoder circuits in verilog would enable measurement of area constraints of the encoder circuits and intrinsic power dissipation of the controller logic. Additionally, we are working on a new technique, called *Signature-value* encoding, where we generate signature along with code at the encoder, and the signature is used to obtain original data from code at the decoder. The functionalities of this encoding scheme are:

- Uses intrinsic properties of the patterns to generate the code.
- Removes the need for cache at the decoder, as it has to only apply the decoding function to obtain original data. As no storage of values is required, it consumes less power.
- The computation at the encoder end is pipelined to reduce latency.

REFERENCES

[1] J. Yang and R. Gupta, "FV Encoding for Low-power Data I/O," in *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, Huntington Beach, California, 2001, pp. 84–87.

[2] D. C. Suresh, B. Agrawal, W. A. Najjar, and J. Yang, "VALVE: Variable Length Value Encoder for Off-chip Data Buses," in *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, Washington, DC, 2005, pp. 631–633.

[3] M. R. Stan and W. P. Burleson, "Bus-invert Coding for Low-power I/O," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 3, no. 1, pp. 49–58, 1995.

[4] Y. Shin, S.-I. Chae, and K. Choi, "Partial Bus-invert Coding for Power Optimization of System Level Bus," in *ISLPED '98: Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, Monterey, California, 1998, pp. 127–129.

[5] Y. Zhang, J. Lach, K. Skadron, and M. R. Stan, "Odd/even Bus Invert with Two-phase Transfer for Buses with Coupling," in *ISLPED '02: Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, New York, 2002, pp. 80–83.

[6] K. Basu, A. Choudhary, J. Pisharath, and M. Kandemir, "Power Protocol: Reducing Power Dissipation on Off-chip Data Buses," in *MICRO 35: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, Los Alamitos, CA, 2002, pp. 345–355.

[7] D. C. Suresh, B. Agrawal, J. Yang, W. Najjar, and L. Bhuyan, “Power Efficient Encoding Techniques for Off-chip Data Buses,” in *CASES ’03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, California, 2003, pp. 267 – 275.

[8] Y. M. Chee, C. J. Colbourn, and A. C. H. Ling, “Optimal Memoryless Encoding for Low Power Off-chip Data Buses,” in *ICCAD ’06: Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, San Jose, California, 2006, pp. 369 – 374.

[9] C. H. Lin, C. L. Yang, and K. J. King, “Hierarchical Value Cache Encoding for Off-chip Data Bus,” in *ISLPED ’06: Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, Tegernsee, Bavaria, Germany, 2006, pp. 143–146.

[10] P. R. Panda and N. D. Dutt, “Reducing Address Bus Transitions for Low Power Memory Mapping,” in *EDTC ’96: Proceedings of the 1996 European Conference on Design and Test*, Washington, DC, 1996, p. 63.

[11] L. Benini, G. D. Micheli, E. Macii, D. Sciuto, and C. Silvano, “Address Bus Encoding Techniques for System-level Power Optimization,” in *DATE ’98: Proceedings of the Conference on Design, Automation and Test in Europe*, Washington, DC, 1998, pp. 861–867.

[12] Y. Aghaghiri, F. Fallah, and M. Pedram, “Irredundant Address Bus Encoding for Low Power,” in *ISLPED ’01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, New York, 2001, pp. 182–187.

[13] “ALBORZ: Address Level Bus Power Optimization,” in *ISQED ’02: Proceedings*

of the 3rd International Symposium on Quality Electronic Design, Washington, DC, 2002, p. 470.

[14] F. Catthoor, E. de Greef, and S. Suytack, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Norwell, MA: Kluwer Academic Publishers, 1998.

[15] P. Sotiriadis and A. Chandrakasan, "Low Power Bus Coding Techniques Considering Inter-wire Capacitances," in *Proceedings of the IEEE 2000 Custom Integrated Circuits Conference*, Orlando, FL, 2000, pp. 507–510.

[16] ——, "A Bus Energy Model for Deep Submicron Technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 3, pp. 341–350, June 2002.

[17] T. Lv, W. Wolf, J. Henkel, and H. Lekatsas, "An Adaptive Dictionary Encoding Scheme for Soc Data Buses," in *DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe*, Washington, DC, 2002, p. 1059.

[18] D. C. Suresh, B. Agrawal, J. Yang, and W. Najjar, "A Tunable Bus Encoder for Off-chip Data Buses," in *ISLPED '05: Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, New York, 2005, pp. 319–322.

[19] V. Wen, M. Whitney, Y. Patel, and J. D. Kubiatowicz, "Exploiting Prediction to Reduce Power on Buses," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, Washington, DC, 2004, p. 2.

[20] W. Fornaciari, M. Polentarutti, D. Sciuto, and C. Silvano, "Power Optimization of System-level Address Buses based on Software Profiling," in *CODES '00: Pro-*

ceedings of the Eighth International Workshop on Hardware/Software Codesign, New York, 2000, pp. 29–33.

[21] D. C. Burger and T. M. Austin, “The Simplescalar Tool Set, Version 2.0, Tech. Rep. CS-TR-1997-1342, 1997. [Online]. Available: citeseer.ist.psu.edu/burger97simplescalar.html

[22] L. Chisvin and R. J. Duckworth, “Content-addressable and Associative Memory: Alternatives to the Ubiquitous RAM,” *IEEE Computer Society*, vol. 22, no. 7, pp. 51–64, 1989.

[23] T. Kohonen, *Content Addressable Memories.* New York: Springer-Verlag, Inc., 1987.

[24] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey,” *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, March 2006.

[25] C. Zukowski and S. Y. Wang, “Use of Selective Precharge for Low-power Content-addressable Memories,” in *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, vol. 3, June 1997, pp. 1788–1791.

[26] I. Arsovski and A. Sheikholeslami, “A Mismatch-dependent Power Allocation Technique for Match-line Sensing in Content-addressable Memories,” *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1958–1966, Nov. 2003.

[27] SPEC, “SPEC CPU2000 V1.3,” Accessed on 03/20/08. [Online]. Available: http://www.spec.org/cpu2000

VITA

Jayakrishnan Venkitasubramanian Iyer obtained his B.E. degree in computer engineering from National Institute of Technology Karnataka, Surathkal (formerly, Regional Engineering College, Surathkal) in 2005. He received an M.S. degree in May 2008 in computer engineering at Texas A&M University. His research interests include computer architecture, vlsi design and wireless sensor networks.

He can be contacted at jviyer@gmail.com