Mixed-Signal Implementation of Low-Density Parity-Check Decoder

SANJOY BASAK

A Thesis

In

The Department

Of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Applied Science (Electrical and Computer Engieering)

At

Concordia University

Montréal, Québec, Canada

December 2017

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: Sanjoy Basak

Entitled: Mixed-Signal Implementation of Low-Density Parity-Check Decoder

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Rabin Raut

_____ External Examiner
Dr. Lata Narayanan

_____ Internal Examiner
Dr. Chunyan Wang

_____ Supervisor
Dr. Glenn Cowan

_____ Co-supervisor
Dr. Warren J. Gross

Approved by: _____
Dr. William E. Lynch, Chair
Department of Electrical and Computer Engineering

_____20____ _____
Dr. Amir Asif, Dean,
Faculty of Engineering and Computer Science

ABSTRACT

Mixed-Signal Implementation of Low-Density Parity-Check Decoder

Sanjoy Basak

The receiver side of many communication systems incorporates an error-correction decoder to achieve good bit-error rate (BER) performance. While good BER is a metric of reliable communication, high throughput and energy-efficiency are also desired. Low-density parity-check (LDPC) decoders are able to perform well in term of these metrics. In this thesis, the Modified Differential Decoding Binary Message Passing (MDD-BMP) algorithm of LDPC codes has been chosen to implement in mixed-signal domain. The goal of this research is to achieve energy-efficiency in LDPC decoding while maintaining high-throughput in an implemented design of reasonable effective area.

The re-design of some digital parts of the LDPC decoder in analog domain is expected to offer energy-efficiency and high throughput. However, these benefits come at a cost of analog impairments, such as, different random mismatch between similar inverters arising from process variation during fabrication. The comparative contribution of these impairments on the BER performance of the decoder has been investigated. During the design of the decoder, an on-chip calibration scheme has been arranged and global routing of the tuning signals has been maintained to address these random mismatches. Furthermore, modulation of the decoding speed by off-chip tuning has been made possible. For the purpose of high-speed testing of the decoding process, enough on-chip memory has been placed to store 10 codewords and feed them to the decoder through a binary-weighted capacitor-based digital to analog converter. Design and placement of analog MUXes enable us to debug sensitive analog nodes inside the decoder from off-chip. Finally, the full process of the physical design of the decoder in TSMC 65nm has been almost fully automated in Cadence SKILL code. Over 100 simulations including parasitic capacitance of long wires in physical design yield an average decoding speed of approximately

2.04 ns in moderate speed mode, therefore, providing a high throughput of 134 Gb/s. Taking into account the average current drawn by the circuits during both the pre-charge phase and the decoding phase, the calculated average energy per bit consumed by the decoder is 1.267 pJ/bit.

ACKNOWLEDGEMENT

I am very grateful to my supervisor Professor Glenn Cowan for his helpful guidance and the time and effort he has spent for me. Over the past two years of my master's program, we had a lot of insightful discussions on practical challenges to circuit design, which gradually increased my interest in this field. His thoughtful suggestions have always been instrumental in leading me through the ebb and flow of research and will continue to be an essential component in shaping my future career.

I would like to thank Mr. Ted Obuchowicz for his help in troubleshooting issues with CAD tools. It undoubtedly helped improve my productivity in research.

My colleagues in the group not only helped me with their design experience but also maintained an excellent environment conducive to research. I would like to thank Abdullah, Xiangdong, Chris, Diaa, Marjan, Weihao, and Marc for helping me achieve my goals.

All through this academic journey, my parents have been a source of endless encouragement and love. My mother's caring suggestions and my father's practical advice helped me make this journey smoother. I also want to thank my brother Joyanta for his encouragement. I am forever indebted to my girlfriend Sushmita for her love and support.

CONTENTS

List of Figures

List of Tables

List of Acronyms

BER	Bit-error rate
DFF	D flip flop
I/O	Input / output
SNR	Signal-to-noise ratio
VN	Variable node
CN	Check node
LDPC	Low-density parity-check
SRAM	Static random-access memory
MUX	Multiplexer
DAC	Digital-to-analog converter
FG	Finite geometry
ESD	Electro-static discharge
SPA	Sum-product algorithm
MS	Min-sum
DD-BMP	Differential decoding binary message passing
MDD-BMP	Modified differential decoding binary message passing
LLR	Log-likelihood ratio

Chapter 1
Introduction

1.1 Motivation

A low-density parity-check (LDPC) error-correction decoder can potentially increase the reliability of a communication system by improving bit-error rate (BER) performance close to the Shannon limit [1][2][3]. The encoding of the LDPC code is done in a straightforward way in the transmitter side of the communication system by adding redundant parity-check bits appending the information bits [4]. However, the receiver side decoding has to follow one of the different algorithms to determine the appropriate transmitted codeword. Initially, the computational complexity of real-world implementation was overwhelming. Over time, there have been a number of less-complex algorithms proposed and also the VLSI technology went through improvements. Therefore, nowadays the incorporation of an LDPC decoder can be found in many commercially successful communication systems, such as, IEEE 802.an 10 Gbit/s Ethernet, DVB-S2 Digital video broadcasting, IEEE 802.11n-2009 Wi-Fi standard, etc. Therefore, both the improvement of the decoder in the algorithmic level and also from the VLSI implementation point of view are relevant to the development of communication systems.

Figure 1 shows a simplified flow diagram of a communication system. The LDPC encoded digital data is modulated, converted and transmitted over a communication channel that adds uncorrelated Gaussian noise to it. The received data is first converted to digital data where there are multiple bits per received symbol. Then, this digital data is de-modulated and finally sent to the LDPC decoder for error-correction. If any of the bits transmitted is received as a different bit, there is an error. The total number of bit errors divided by the total number of received bits, determines the bit-error rate. Introduction of LDPC encoder and decoder can offer significant advantage in BER improvement at the cost of transmitting additional redundant bits from encoding, and the energy and resources spent for encoder and decoder. Since the advantages of including the LDPC coder-decoder far outweigh its small resource demands, there is a growing use of it.

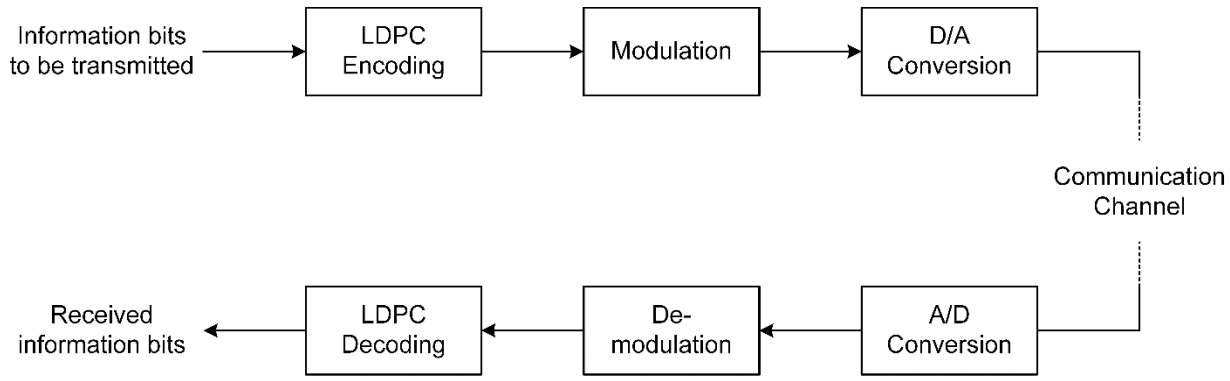


Figure 1: Simplified flow diagram of a communication system.

Since the LDPC decoding is done once the received digital bits are available to us, it is not unintuitive that the initial effort to implement the decoding process will be made in a digital domain. The first decoding algorithm was suggested by the proponent of LDPC decoding, Gallager himself in [5]. In any LDPC decoding algorithm, each data bit received from the channel is assigned an estimate of probability of being zero or one. These probabilities, being unbounded, are then clipped, mapped into different voltage levels between the two rail voltages, and quantized. Based on how these quantized values will be used in the decoder initialization process and also the particular way of implementation, LDPC decoding algorithms are categorized into two types – hard decision decoding and soft decision decoding. While in soft decision decoding, these quantized values are used to initialize the decoder, hard decision decoding rounds these quantized voltage levels to one of the rail voltages depending on their position above or below the midpoint between the two rails. For example, if the supply voltages are VDD and zero, all the quantized levels are rounded to either VDD or zero depending on their values above or below VDD/2 respectively. Gallager's decoding algorithm was hard-decision algorithm. Although the initialization of the hard decision decoding algorithms looks simple with only VDD or zero, their bit-error-rate (BER) performance is poor compared to that of the soft-decision decoding algorithms. However, the superior BER performance of the soft decision

decoding algorithms also comes along with higher computational complexity. Once the decoder is initialized, binary messages are passed iteratively between two types of nodes—variable nodes and check nodes until some decoding termination criterion is reached. The bit flipping algorithm, for example, works by flipping the variable node bits whenever a majority of check nodes connected to that variable node are not parity-check satisfied.

At the time of Gallager, the implementation of LDPC decoding algorithm was computationally intensive and did not attract much research effort. With the advance of VLSI technology, researchers focused back to the implementation of LDPC decoding [6]. Consequently, more and more efforts were made by the information theorists to simplify the computationally complex but better performing soft decision decoding algorithms so that the routing complexity gets lower. Such simplification, although at a cost of small BER performance degradation, offers far more implementation advantages in return. The best known of all the proposed soft-decision LDPC decoding algorithm in terms of BER performance is the relaxed sum-product decoding algorithm [7][8]. In the sum-product algorithm, the variable nodes are initialized with estimates of probability of being zero or one based on the channel characteristics. Commonly used estimates of probability are the likelihood ratio (LR) or the log-likelihood ratio (LLR). After that, decoding is performed by iterative message passing between the two types of nodes and by calculating their values until some decoding termination criterion is reached. While the BER performance of the sum-product algorithm is good, the implementations of the variable node and check node calculation equations are complex. This motivated researchers to find out a simplified variant of the sum-product algorithm. One such example is the min-sum algorithm. The initialization and the iterative procedure in the min-sum algorithm are the same as those of the sum-product algorithm except for the fact that the check node value-calculation equations are much simpler in the min-sum algorithm. Thus, in the circuit-level implementation, the min-sum algorithm offers reduced complexity in the check node architectures at the expense of a small BER performance degradation.

The idea of analog decoding of LDPC codes was first proposed by Hemati et al [8]. They investigated the application of successive relaxation method [9][10] to the iterative decoding algorithms, such as sum-product and min-sum algorithm. The continuous-time analog decoding in an ideal analog mismatch-free scenario was shown to significantly outperform synchronous

digital decoding in terms of the BER performance. Inspired by the impressive performance of successive relaxation in decoding algorithms [8][11], Mobini implemented this idea in his differential binary message-passing decoder [12]. Unlike Hemati's analog decoder, Mobini's decoder was still synchronous digital in nature but employed small discrete-time update of the memory. In other words, in her decoder, soft information was used not only to initialize the decoder but also to store the updated values of the memory at each iteration. In [13], the differential decoding algorithm was further simplified by replacing check node specific variable node broadcasting with the same variable node broadcasting to all connected check nodes. This simplified variant was named the Modified Differential Decoding Binary Message Passing (MDD-BMP) algorithm. Therefore, MDD-BMP algorithm significantly reduces the wiring complexity from the previous differential binary message passing decoding.

While the reduction in the wire routing complexity is mostly up to the algorithm itself, there is an expectation that replacing some digital signal processing circuitry with its analog counterpart may prove useful. For example, a large area consuming clock-synchronized digital adder can be replaced by continuous-time current-mode addition of incoming signals. The analog replacement of some parts does not come without analog impairments. For example, the random mismatch between current-sources, need for delay matching in asynchronous decoding, etc. On the other hand, successfully addressing these challenges can put us in a very advantageous position of claiming low-power high-throughput mixed-signal LDPC decoding. This achievement will be particularly interesting due to the fact that battery-operated portable communication devices will continue to see power-saving architectures in near and foreseeable future.

Considering all the above opportunities, we venture to design and implement the mixed-signal LDPC decoder with high performance and to resolve the limitations associated with the analog replacements.

1.2 Research Objective and Contribution

The aim of this research is to implement the MDD-BMP algorithm for LDPC decoding in the mixed-signal VLSI domain. The implemented decoder has to be able to minimize the effects of the analog impairments predicted in the prior work. In addition to that, the decoder has to show energy-efficiency and maintain high-throughput. Area compression strategy of the physical design of the decoder has to be sought as well.

Keeping the objectives in mind, the following contributions were made in this thesis.

- Investigating comparative contribution of the analog impairments on the performance of the decoder through modified Python simulation.
- Schematic design of different parts of the decoder, which also address the impairments. Design of memory array for the storage of codewords, DAC for analog-level construction, the core decoder, Analog MUX for debugging, termination-check circuit, and the customized I/O shift registers.
- Formulation of a compression strategy for compact decoder physical design and almost full automation of the decoder physical design in TSMC 65nm CMOS.

1.3 Thesis Organization

This thesis consists of six chapters, including this Chapter 1 as the introduction. Chapter 2 discusses the evolution and performance of different LDPC decoding algorithms, citing relevant publications in the literature. This chapter also presents the prior coding and simulations done, which establishes the basis of the thesis. From Chapter 3, starts the actual contribution of the writer towards the thesis. Chapter 3 describes the comparative contribution of different analog impairments of the mixed-signal decoder and proposed solutions to address them. Chapter 4 contains detailed discussion on the schematic design of different parts of the decoder and also shows the simulation results for the decoder. Chapter 5 is about the physical design of all parts of the chip, the compression techniques in area, and wiring strategy for complex

interconnects. In Chapter 6, the test plan for the fabricated chip is presented. Chapter 7 concludes the thesis and delineates potential future areas of research. There are two appendices for this thesis. Appendix A contains the MATLAB code for decoder area compression in physical design. Appendix B includes the Cadence SKILL code used to almost fully automate the design of complex decoder build up in physical design.

Chapter 2

Background and Literature Review

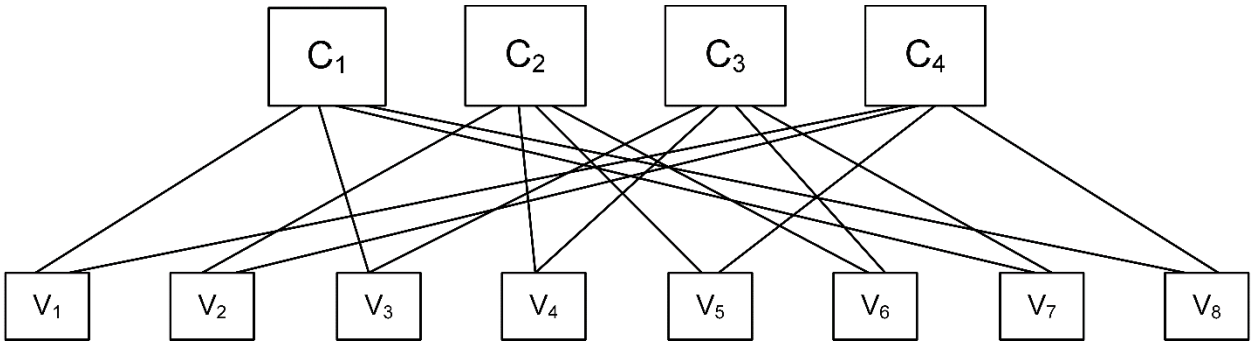
2.1 Low-Density Parity-Check (LDPC) Codes

Low-density parity-check (LDPC) codes are one of the most well-known error-correcting codes with the potential to achieve error correction performance close to the Shannon limit [2][5]. Although introduced by Gallager more than half a century ago, their phenomenal strength has been realized much later as high-end computational tools became available. There have been a lot of efforts to reduce the complexity of the LDPC decoders while trying to maintain their superior performance.

Similar to other block codes, an LDPC code can be described by a parity-check matrix, H. While the alphabet consisting of the entries of the H matrix can be a wide selection of real numbers, binary representation is the most common form used for simplicity. The H matrix is a sparse M × N matrix for which an LDPC codeword $\underline{c} = (C_1, C_2, \ldots, C_N)$ becomes,

$$\underline{c}H^T = 0 \tag{2.1}$$

The LDPC can alternatively be represented by a Tanner graph [14], as shown in Figure 2.1. This figure shows the Tanner graph representation of the H matrix following it. Each row of the matrix represents a check node (CN) and each column represents a variable node (VN). Therefore, the H-matrix in Figure 2.1 has four CNs and eight VNs. The non-zero entries of the H-matrix form an edge or a connection between a particular CN and a particular VN. In other words, a non-zero entry $d_{i,j}$ makes a connection between the i-th CN and the j-th VN on the tanner graph. In practice, the H matrix is much larger in both dimensions and has very few non-zero elements compared to the zeros, making H a sparse matrix.



$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.1: The Tanner Graph and the H matrix

The encoding of the LDPC code is done using a generator matrix G of size K × N. Here, K is the length of the information bits, u = (U_1, U_2, …., U_K) and (N-K) additional bits are inserted to form the codeword c.

$$\underline{c} = \underline{u}G \tag{2.2}$$

From equation (2.1), we have, $\underline{c}H^T=0 \Rightarrow \underline{u}GH^T=0 \Rightarrow GH^T=0$.

Therefore, the generator matrix G can be found from the parity-check matrix H. The H matrix can be rearranged by Gaussian elimination to form $H = [P|I_{N-K}]$, where I_{N-K} is the identity matrix of size N-K by N-K and the P matrix is of size M=N-K by K. Now, the generator matrix, $G = [I_K|P^T]$.

The encoding of the LDPC code is done at the transmitter side of the communication system and the decoding has to be done at the receiver side. At first, the binary codeword $\underline{c}$ is mapped into a bipolar signal $\underline{b}$, for which each entry is a map of the corresponding entry of $\underline{c}$, i.e., $b_i = (-1)^{ci}$. As the bipolar codeword $\underline{b}$ passes through a binary-symmetric input additive white Gaussian noise (AWGN) channel, noise is added to the codeword. So, at the receiver side

of the channel, we get $\underline{r} = \underline{b} + \underline{n}$, where n is a zero-mean Gaussian random variable independent of the transmitted codeword $\underline{b}$.

The decoding of this noise-added transmitted codeword is done by a number of algorithms. Many of these algorithms works by iterative message passing between the two nodes of the LDPC decoder until convergence is achieved. The set of all these algorithms are called message passing algorithms. Since a binary alphabet is being considered here, the binary versions of these message passing algorithms will be discussed. In the next sub-section, different hard-decision and soft-decision LDPC decoding algorithms will be discussed and their merits will be compared in terms of the bit-error-rate (BER) performance and suitability to implement in a low-complexity energy-efficient design.

2.2 LDPC Decoding Algorithms

There is a wide spectrum of message passing LDPC decoding algorithms available, each offering a different performance/complexity trade-off. Among them, hard decision algorithms are the simplest in implementation complexity. Some hard decision algorithms are Gallager A and B algorithms [5][15], bit flipping algorithm [16], and majority-based algorithm [17]. Although the parity-check matrix is sparse in nature, there are a lot of edges between variable nodes and check nodes. This large amount of wiring makes the VLSI implementation of LDPC decoder a challenging one. Hard decision algorithms offer the lowest wiring complexity, which, in turn, necessitates smaller area and low power. In spite of these attractive features, the major drawback of the hard decision algorithm is its poor bit-error rate (BER) performance compared to that of soft-decision algorithms. Therefore, most researchers in recent years have put effort to find out simplified variants of the soft decision algorithms, which offer significant reduction in complexity while trading off as little error-correction performance as possible.

In this thesis, a particular simplified variant of the soft-decision decoding algorithm, i.e., Modified Differential Decoding Binary Message Passing (MDD-BMP) algorithm has been

chosen to implement. However, the discussion of the decoding algorithms' evolution from hard decision to gradual simplification of soft decision algorithms is relevant.

2.2.1 Bit-Flipping Algorithm and the Analogy between Hard Decision and Soft Decision Algorithms

Bit-flipping algorithm is a hard-decision decoding algorithm. Its initialization requires calculation of the estimates of probability of a symbol being 0 or 1, and these estimates are then rounded to one of the rail voltages and assigned to each variable node. Now, the check nodes calculate the modulo-2 sum of the VNs connected with the edges of the tanner graph. The modulo-2 sum at each CN is called the parity-check sum. If the sum is zero at a particular CN, the CN is satisfied. On the other hand, a CN with check sum non-zero means the CN is unsatisfied. At this point, the CNs, based on their satisfaction, make judgement about the VNs connected to them. If all the CNs connected to a VN are satisfied, then the CNs judge the VN's position to be correct and reinforces its position. On the contrary, the unsatisfied CNs connected to a VN wants to flip the VN's binary position. In a divided jury, how many unsatisfied CNs will have to be there to flip the binary VN bit, depends on the threshold for the bit flipping algorithm selected. For now, suppose, a majority of unsatisfied CNs over satisfied CNs is capable of flipping the corresponding VN bit. Once all the VNs are either flipped or reinforced by the corresponding CNs, VNs again pass their messages to the connected CNs for new modulo-2 sum. Again, the judgement by the CNs repeats. This process continues until all the CNs are satisfied or some maximum number of iterations is reached. When all the CNs are satisfied, the successfully decoded bits are obtained as the values of the VNs.

The Bit-flipping algorithm provides a good starting point to understand the LDPC decoding process. Furthermore, the process discussed above is highly analogous to the soft-decision decoding. In hard decision decoding, each VN is initialized by just the received bits since the probability of them being '0' or '1' is not available to us. On the other hand, in soft decision algorithms, VNs are initialized with an estimation of likelihood of a symbol being a one

or zero. Such probabilistic estimations are called 'belief'. The beliefs often come in the form of log-likelihood ratios (LLRs). The LLR is defined by the following equation.

$$LLR_v = \ln\left[\frac{P(x_v=0|y_v)}{P(x_v=1|y_v)}\right] \quad (2.3)$$

where x_v is the v-th transmitted bit and y_v is the v-th value received from the channel.

Interestingly, the LLRs calculated for the initialization of the variable nodes are unbounded. Therefore, they need to be clipped at a maximum and a minimum point, and then they are quantized into voltage levels between this clipping range. The variable node is initialized with these quantized voltages.

In addition to the initialization process, many modified soft-decision algorithms contain memory inside the variable node, which enables discrete-time update of the variable node at each iteration instead of flipping altogether if a majority of connected check nodes are unsatisfied.

Initialization of the VNs with LLRs is a very intuitive idea in the sense that the VNs initialized as being close to zero or one are more likely to be zero or one respectively in the finally decoded form. In contrary to it, the VNs initialized with the LLRs close to the middle of zero and one, are more likely to be in error and are more likely to cross the threshold to move close to the opposite rail. This phenomenon is shown in Figure 2.2. In the hard-decision decoding, the VNs are initialized as either VDD or zero. On the contrary, the soft-decision decoding requires the VNs to be initialized as different quantized voltage levels between VDD and 0.

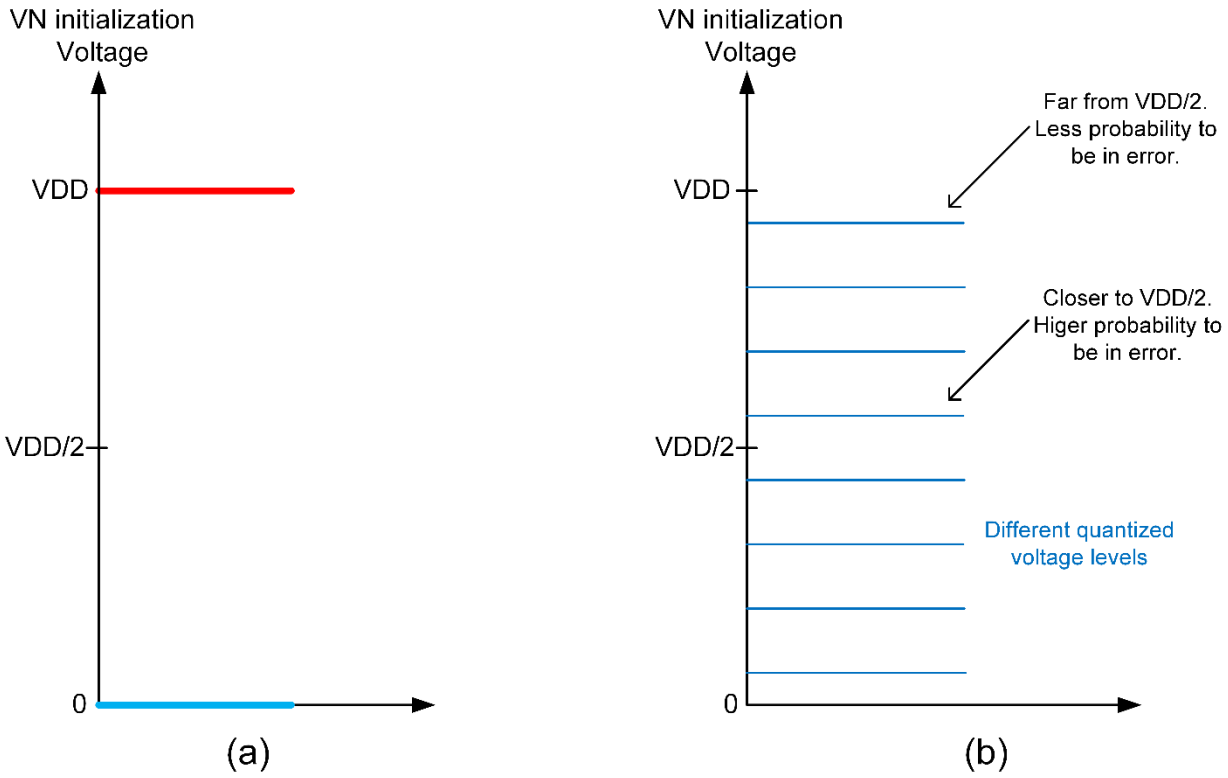


Figure 2.2: (a) Initialization of VN as either VDD or 0 in hard-decision decoding. (b) Initialization of VN as different quantized levels between VDD and 0 in soft-decision decoding.

Although soft-decision decoding is capable of providing significantly better bit-error rate (BER) performance, the implementation of memory-based VNs cost space and energy. Before the digital implementation of soft decision decoding is discussed, the evolution of soft-decision decoding to make it more amenable to VLSI implementation will be highlighted in brief.

2.2.2 Sum-Product Algorithm

The sum-product algorithm is one of the soft-decision decoding algorithms [7]. The difference between different variants of the soft decision decoding algorithms is the calculation inside the CN. The BER performance of the sum-product algorithm is better compared to other soft decision decoding algorithms. However, the implementation complexity of sum-product is much higher.

Suppose that a binary codeword $\underline{c} = (C_1, C_2, \ldots, C_N)$ is transmitted over an AWGN communication channel. At the receiver end, a noise added codeword $\underline{b} = (B_1, B_2, \ldots, B_N)$ is received. Let $Y_{v\to c}$ and $Z_{c\to v}$ be the messages passed from VN_v to CN_c and from CN_c to VN_v respectively. The sum-product algorithm follows the following steps.

1. Calculate the a posteriori log-likelihood ratio (LLR) for all VNs.

   LLR for the k-th VN, $\lambda_k = \ln\frac{P(c_k=0|b_k)}{P(c_k=1|b_k)}$, where c_k is the k-th transmitted bit, and b_k is the k-th value received from the channel. Now, the iteration count begins, i=1.

2. Compute the check nodes from the newly assigned variable nodes' values.

   The check node computes, $b_{c\to v}^{(i)} = 2\tanh^{-1}\left(\prod_{v\prime\epsilon V\setminus v}\tanh(b_{v\prime\to c}^{(i)}/2)\right)$.

   Here, $b_{v'\to c}$ represents the message passed from variable node *v* to check node *c*. *V* is the set of variable nodes connected to the check node *c*. Therefore, *V\v* is the set of variable nodes connected to check node *c* with variable node *v* removed.

3. Now, update the variable nodes according to the new values of the check nodes.

   $b_{v\to c}^{(i+1)} = \lambda_v + \sum_{c\prime\epsilon C\setminus c} b_{c\prime\to v}^{(i)}$

4. Make a hard decision $\underline{\hat{u}} = (\hat{U}_1, \hat{U}_2, \ldots, \hat{U}_N)$ from the updated variable node values.

$$\hat{U}_v = \begin{cases} 0 & if\ \lambda_v + \sum_{c\prime \epsilon C} b_{c\prime \to v}^{(i)} \geq 0 \\ 1 & otherwise \end{cases}$$

If the syndrome, $\underline{\hat{u}}H^T = 0$, $\hat{u}$ is the decoder output. Otherwise, increment iteration, i = i+1 and go to step 2.

From the steps of the sum-product algorithm, it is obvious that the product inside the check node is computationally complex. Therefore, other simpler variant emerged, such as, the Min-Sum algorithm.

2.2.3 Min-Sum Algorithm

Min-Sum algorithm, being an approximation of the sum-product algorithm, has similar steps except for the check node computation. MS algorithm has made CN computation significantly simpler compared to the sum-product algorithm and therefore, much more suitable for hardware implementation. The steps are sequentially described below. Let $b_{v\to c}$ and $b_{c\to v}$ be the messages passed from VN_v to CN_c and from CN_c to VN_v respectively.

1. Assign calculated LLR data to the VNs at the first iteration, i=1.
2. Compute check nodes. $b_{c\to v}^{(i)} = min_{v\prime \epsilon V\setminus v}|b_{v\prime \to c}^{(i)}| \ . \ \prod_{v\prime \epsilon V\setminus v} \mathrm{sgn}(b_{v\prime \to c}^{(i)})$.
3. Update the variable nodes. $b_{v\to c}^{(i+1)} = b_{v\to c}^{(i)} + \sum_{c\prime \epsilon C\setminus c} b_{c\prime \to v}^{(i)}$.
4. Make the hard decision from updated VN values. The hard decision of the n-th VN is

$$\hat{U}_v = \begin{cases} 0 & if\ \lambda_v + \sum_{c\prime \epsilon C} b_{c\prime \to v}^{(i)} \geq 0 \\ 1 & otherwise \end{cases}$$

If the syndrome, $\underline{\hat{u}}H^T = 0$, the hard decision for all the VNs constitute the decoder output. Otherwise, go to the step 2.

While the VN update happens as the sum of the CN decisions are accumulated on top of the VN value from previous iteration, the CN computation is just the product of all the signs of the messages passed from VNs multiplied by the minimum absolute value. The implementation of the MS algorithm is straightforward. The VN requires a number of separate memories and adders. The CN implements the product of the VN signs, implemented through an XOR gate, and multiplies the product with the output of the minimizer block, which finds the minimum absolute value of the incoming VN signs. MS algorithm's error-correction performance is a little less than that of the sum-product algorithm. Nevertheless, MS algorithm's significantly reduced complexity makes it a preferred choice for implementation.

2.2.4 DD-BMP Algorithm and Its Modified Form

The differential decoding binary message passing (DD-BMP) algorithm is the modified binary version of the Min-Sum algorithm with a memory inside the variable node [12]. Now, unlike the hard decision decoding algorithms or the sum-product or min-sum algorithms, DD-BMP algorithm offers VN memory for storing and updating soft information. The initialization of the VN is done with LLR data which are themselves soft information. In MS algorithm, the summation done inside the VN is memoryless. However, in DD-BMP, VN memories are differentially updated from their previous quantized levels at each iteration based on the binary messages from the check nodes. Furthermore, in DD-BMP algorithm, the CN calculation has been further simplified by omitting the multiplication of the minimum of all the incoming VN output signals with the product of their signs inside the CN block, which the min-sum algorithm requires.

In the highly interconnected Tanner graph, at the initial period of decoding, a few VN beliefs can potentially enable connected CNs to make calculations leading to dissatisfaction. Therefore, a VN connected to majority unsatisfied CNs is not necessarily storing a bit that is an error. As more iterations continue, the particular VN might stay in the side it initially was. In case of bit flipping, the VN will be switched multiple times back and forth until the decoding terminates. However, if the unsatisfied CNs make the connected VNs only move a little at each

iteration instead of flipping altogether, the occurrences of switching actions will be much lower. This will happen because in this case, the VNs crossing little beyond the threshold will be detected for a possible error and corrected if they are in error compared to the bit flipping where the VNs go all the way to the rail voltage if flipped.

The steps of the DD-BMP algorithm will not be discussed since it resembles the steps of the min-sum algorithm except for the update expression of the VN and the CN. The update expression of the VN is described as follows.

$$b_{v\to c}^{(i+1)} = b_{v\to c}^{(i)} + \sum_{c\prime\epsilon C\backslash c} b_{c\prime\to v}^{(i)} \tag{2.4}$$

where, $b_{v\to c}$ is the edge from the v-th VN to the c-th CN, and $b_{c'\to v}$ is the edge from the c-th CN to the v-th VN. C is the set of all check nodes connected to v-th VN. So, C\c is the set of check nodes connected to the v-th VN with the c-th check node removed. Interestingly, in DD-BMP algorithm, the initial channel data is not used in VN computations except during initialization.

Similarly, the update of the check node is little different from that in the Min-Sum algorithm. In DD-BMP, the CN output is the binary signal representing the product of the signs of incoming signals. The update expression of the CN in DD-BMP is as follows.

$$b_{c\to v}^{(i)} = \prod_{v\prime\epsilon V\backslash v} \mathrm{sgn}(b_{v\prime\to c}^{(i)}) \tag{2.5}$$

where, $b_{v'\to c}$ is the message from the v-th VN to the c-th CN. $b_{c\to v}$ is, therefore, a binary signal.

In [12], it has been shown that the performance of the DD-BMP algorithm is significantly higher than the hard-decision algorithms and also, its complexity is lower than the other soft decision algorithms. In spite of the simplification of the VNs and CNs, the wire routing complexity is still high. The reason behind is the fact that each CN has to broadcast a VN-specific message and each VN also has to have separate memories for CN specific updates. Mitigating these two issues makes the modified DD-BMP algorithm (MDD-BMP) a very powerful one.

In MDD-BMP [13], two major changes have been made compared to the original DD-BMP algorithm. First, the check node specific VN memories have been replaced by a single memory per VN. Therefore, the VN memories are no longer associated with the edges of the tanner graph. Instead, all the connected CNs' decisions affect the single VN memory in a

combined way. While this step reduces the area and energy consumption significantly, it has been shown that the performance degradation is small. Second, the variable node specific message passing from the CN has been replaced by a single message passing from a CN to all connected VNs. This change has been made possible by simultaneous addition of an XOR de-embedding stage for each incoming CN message inside VN. This de-embedding stage filters a VN's own contribution to the connected check node. This single message passing by a CN to all its VNs is called broadcasting. This step also greatly reduces the wiring complexity. Figure 2.3 from [13] shows the difference between DD-BMP variable node and the MDD-BMP variable node. The reduction in the size of CN in DD-BMP non-broadcast version compared to the original DD-BMP broadcast version is illustrated in Figure 2.4, which has also been discussed in [13].

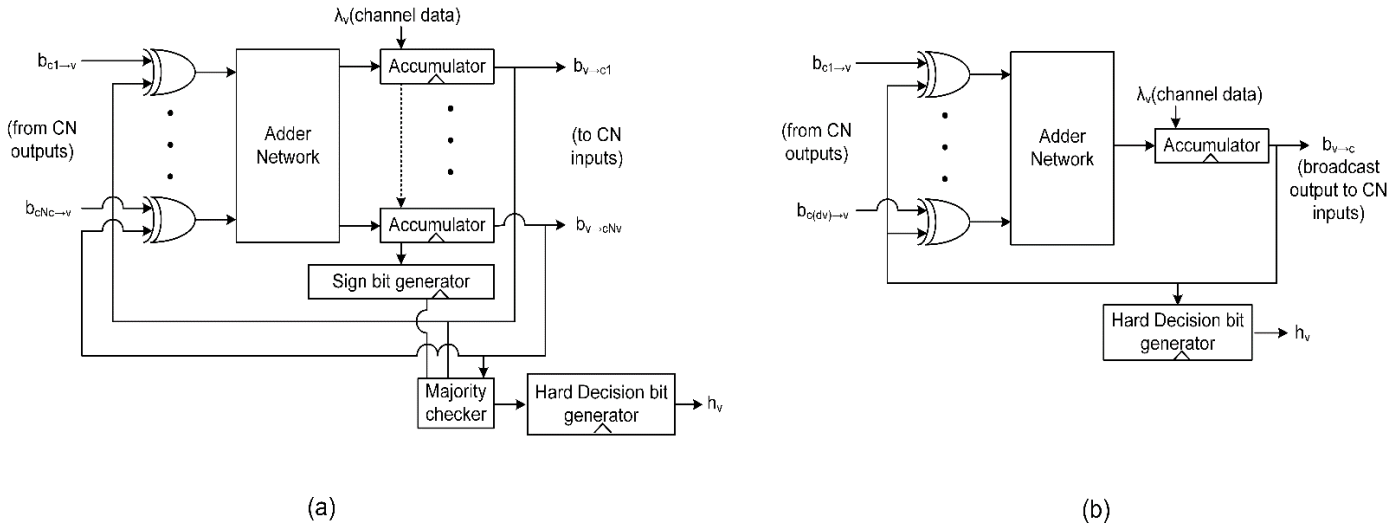


Figure 2.3: (a) VN schematic in DD-BMP. Each input and output is connected to the corresponding CNs, i.e., c1 to c(d_v).
(b) VN schematic in MDD-BMP. A single output $b_{v\rightarrow c}$ is broadcast to all connected CNs [13].

To sum up, the broadcasting approach used in MDD-BMP greatly reduced the wiring complexity compared to that in DD-BMP algorithm. Therefore, MDD-BMP becomes better suited to VLSI implementation.

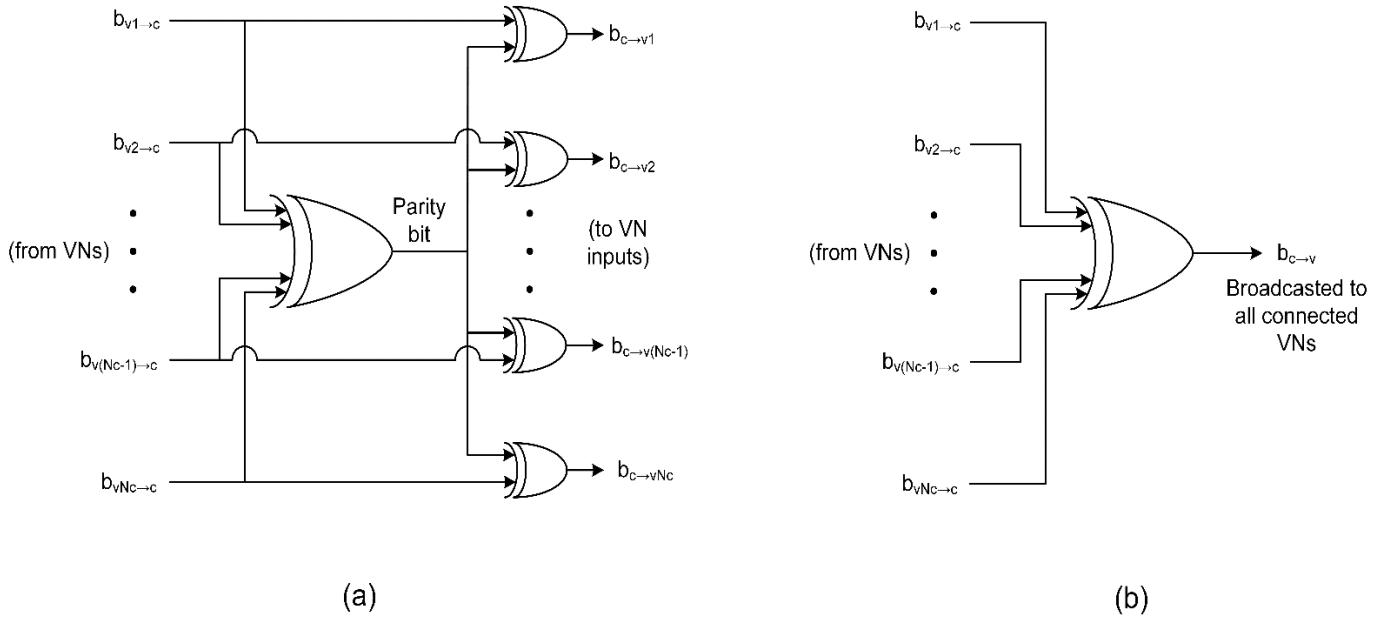


Figure 2.4: (a) DD-BMP CN schematic for non-broadcast version.
(b) DD-BMP CN schematic for broadcast version. Figures taken from [13] and re-drawn.

Let us revisit our past discussion about the bit flipping algorithm and see how that compares to the simplified MDD-BMP version. It has been observed in the bit flipping algorithm that the majority unsatisfied CNs can flip the corresponding VN. In MDD-BMP, the unsatisfied CNs would simply push the VN trajectory to cross the median line between one and zero to the opposite side so that the VN is now in the correct binary side. Let us investigate this situation with an example. Suppose, in an LDPC parity-check matrix H, the degree of each VN is 17, i.e., each VN is connected to 17 CNs. If the bit flipping algorithm is considered first, it takes 9 CNs to be unsatisfied to flip the binary VN bit. Otherwise, if the number of unsatisfied CNs is equal to or less than eight, the corresponding binary VN bit remains unflipped. Interestingly, the bit flipping algorithm does not differentiate between its actions whether the number of unsatisfied CNs is 0 or 8, or 9 or 17. In both scenarios, the unsatisfied CNs either flip the VN bit or not with equal intensity. Therefore, the MDD-BMP algorithm makes better sense since each unsatisfied CN adds its own strength in pushing the VN trajectory to the other side of the threshold line while each satisfied CN pulls the VN to stay in the current side of the threshold. The fight among

the satisfied and the unsatisfied CNs decides the movement of the VN. Figure 2.5 (a) and (b) shows the movement of an example VN trajectory at each iteration in bit flipping and in MDD-BMP decoding respectively. In Figure 2.5(a), the vertical axis represents the binary VN state, which can be either +1 or -1. In Figure 2.5(b), the vertical axis represents the LLR values as the VN states. In both figures, the VN trajectory moves rightward along the horizontal axis as number of iteration increases. Let us assume that in this example, the total number of CNs connected to this VN is 17. In bit flipping, at least 9 unsatisfied check nodes are necessary to flip the bit from +1 to -1 as depicted in the Figure 2.5(a). On the other hand, each unsatisfied CN has its contribution towards forcing the VN to cross the LLR=0 threshold. A higher number of unsatisfied CNs, therefore, causes the VN to make a larger change of its LLR level.

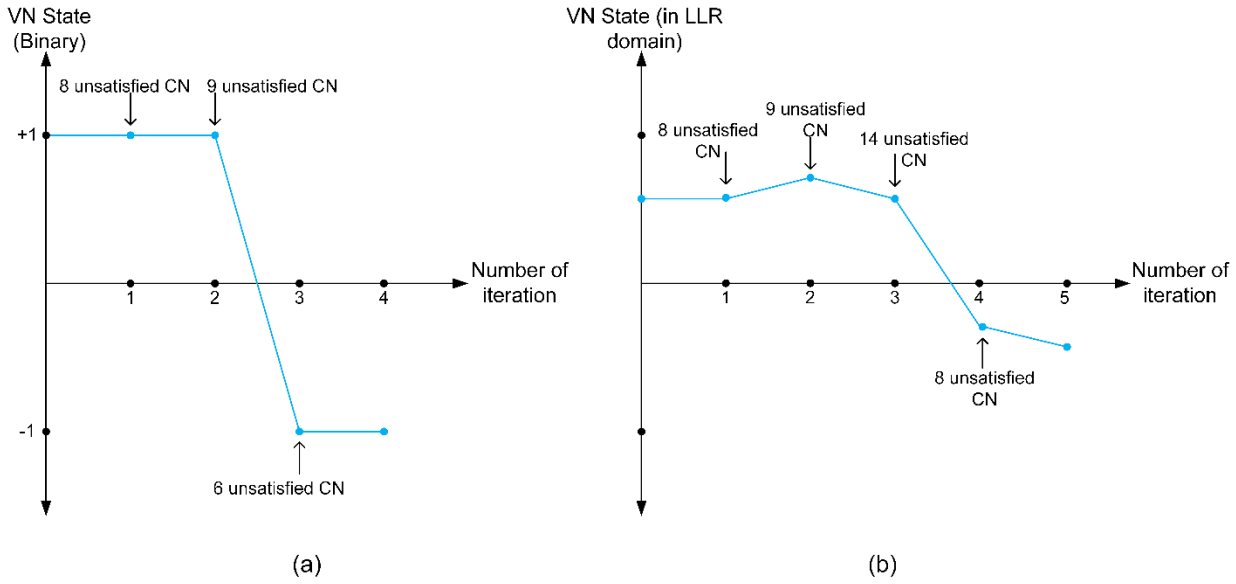


Figure 2.5: (a) VN trajectory in bit flipping decoding. (b) VN trajectory in MDD-BMP decoding.

2.3 Digital Implementation of MDD-BMP and Possibility of Its Analog Reorganization

MDD-BMP is the most simplified version of all the soft-decision LDPC decoding algorithms. In VLSI implementation, there are two major blocks that have to be designed – the VN and the CN. Their structures have already been depicted in Figure 2.3(b) and 2.4(b) respectively. The VN has a single memory that is initialized with the channel LLR data at the beginning. The update of the VN in subsequent iterations is just the addition of the scaled CN messages to this particular VN.

$$M_v^{(k)} = M_v^{(k-1)} + \left\lfloor s.\ \sum_{c\prime \epsilon C} M_{c\prime \to v}^{(k-1)} \right\rfloor \tag{2.6}$$

Here, k is the iteration index and C is the set of all check nodes connected to the variable node v. The parameter 's' scales the sum of all the incoming CN messages to the VN.

From equation (2.6), it is obvious that the VN updates in a step-by-step fashion. However, the message a VN transmits to all the connected CNs is the sign of its value. This sign is, in other words, called the hard decision.

$$b_{v\to c}^{(k)} = \mathrm{sgn_r}\,(M_v^{(k)}) \tag{2.7}$$

Here, $\mathrm{sgn_r}$ is a modified sign function for which $\mathrm{sgn_r}\,(0) = 1$. $b_{v\to c}^{(k)}$ is the hard decision message the VN *v* passes to any connected CN *c* during the k-th iteration.

While the VN updates itself, the CN has to simultaneously update as well. The calculation inside the CN is simpler – the product of all the incoming signs from the VNs.

$$b_{c\to v}^{(k)} = \prod_{v\prime \epsilon V} sgn_r(b_{v\prime \to c}^{(k)}) \tag{2.8}$$

Where, V is the set of all VNs connected to the CN c. The CN is able to broadcast its value to all its VNs since the de-embedding is being done inside the VN schematic.

The iterations continue until the parity-check satisfaction is reached or the process is terminated by a pre-specified maximum number of iterations.

The analog implementation of the LDPC decoder in MDD-BMP algorithm argues that the complexity of the VN block can be reduced with the proposed analog counterpart while keeping the CN block as it is [18]. In digital implementation, there is an adder to sum the incoming CN messages to the VN. After summation and scaling, the updated value will be stored in the memory, replacing the value stored from the previous iteration. The analog implementation proposes to use a capacitor as a memory while incoming signals will be added in current-mode on top of the capacitor. This replaces the large energy-consuming digital adder. The concept has been illustrated in Figure 2.6. In this figure, all the incoming CN output messages are first XORed by the delayed sign of the VN memory itself. The XORed signals then drive the current-source inverters, which combinedly charge up or down the VN capacitor. However, the analog implementation has its own drawbacks. The circuit implementation capable to address the drawbacks is the purpose of this thesis.

2.4 Overview of the Prior Work on Analog Decoder Implementation

The prior work on the analog implementation of the LDPC decoder and the estimation of potential limitations of this scheme have been addressed in [18] by Prof. Glenn Cowan, the principal investigator in this project. In this section, the changes made to the digital decoder to implement the analog counterpart will be presented and the challenges that affect the BER performance of the decoder will be discussed.

In the digital decoder, inside the VN, all the incoming CN messages are added in an adder and the scaled version of this addition is accumulated with the value from the previous iteration in the accumulator. On the other hand, in the analog decoder, VN memory is implemented through a capacitor. The signal addition is done in current-mode where some inverters are used as the current-suppliers. The process is shown in Figure 2.6.

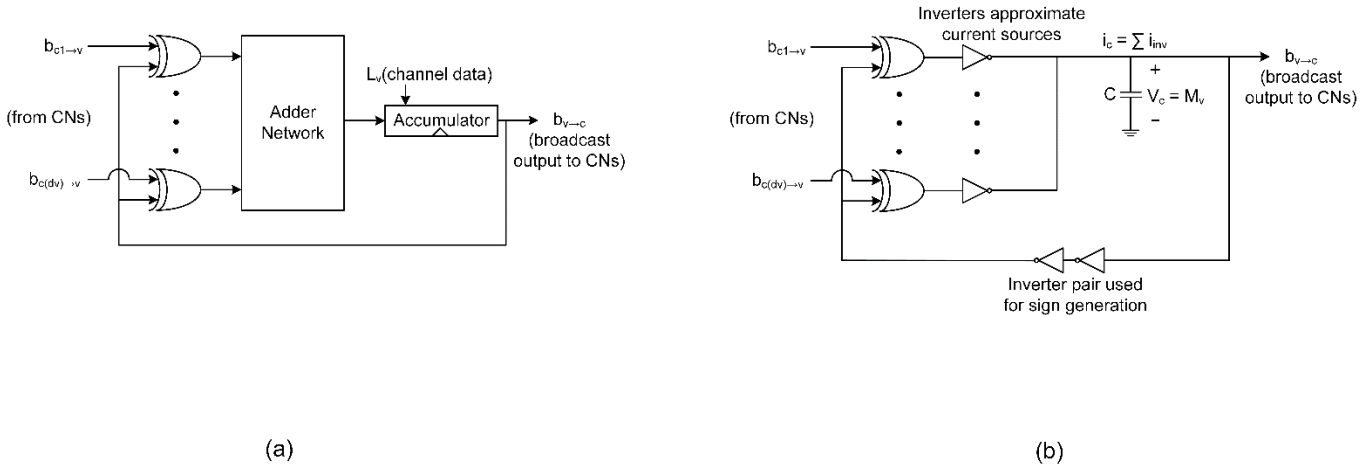


Figure 2.6: (a) VN in Digital MDD-BMP Decoder. (b) VN in Analog MDD-BMP Decoder.

The analog implementation makes the digital adder unnecessary saving area and power. However, it also suffers from a few drawbacks, as described below [18].

(i) The current-source inverters have finite output resistance. Ideally, the output conductance of these inverters should be zero. In other words, the current flowing out of these inverters should be independent of the voltage level at the output node, therefore, the current supplied should be a function of the input to the inverter only. However, in practice, the output current is a function of the capacitor voltage.

To include this behavior in simulating the decoder dynamics in python, a single 4^{th} order polynomial, mapping VN state (on capacitor) to message weight was used. However, the use of this model does not produce any significant deterioration in the decoding performance.

(ii) Systematic offset between PMOS and NMOS transistors in current-source inverters due to process variation. This offset can lead to different sinking and sourcing currents by different current-source inverters. Let us illustrate the effect of this systematic offset with an interesting example. Suppose, a VN is connected to 17 CNs. 9 of the current-sourcing inverters are supplying currents to the VN capacitor while the other 8 inverters are drawing current to discharge the capacitor. If each of the current-source inverters is assumed to have equal current supplying or sinking

capability, the VN capacitor will gradually charge up. However, in case the current-sourcing and current-sinking inverters are not of the same current supplying capability, 8 sinking inverters may be able to draw more current than the total current the other 9 sourcing inverters provide. This will lead the VN capacitor to discharge and the VN voltage to gradually fall compared to a gradual rise in the ideal situation. Occurrence of such situation can have adverse effect on decoding performance.

During the design and simulation, the current-source inverters can be calibrated so that a current-sinking and a current-sourcing inverter provides the same amount of current when their output is connected to a VN state of VDD/2. However, in the fabricated chip, due to process variation, the sinking and sourcing currents can be different. Corner simulation shows this offset to be as much as 20% for some process corners. Figure 2.7 shows the effect of different magnitudes of systematic skew in current-source inverters on the BER performance. This systematic skew needs to be addressed to keep it much smaller. The proposed solution to eliminate this skew has been discussed in detail in chapter 4 section 4.4 of this thesis.

(iii) Random mismatch between current-source inverters feeding variable nodes. In addition to the offset between a sinking and a sourcing inverter, there can also be random mismatch between different inverters. [18] reports that after running 100 monte-carlo simulations, the relative standard deviation in output current across V_{DS} voltage (VN state) is in the range of 5%. The effect of this random mismatch on BER is also depicted in Figure 2.7.

(iv) Mismatch between threshold inverters used for sign in VNs relative to those used in CNs. In MDD-BMP, VN has a single memory. Its sign is broadcasted to all connected CNs. Since these broadcasted binary signals can lead to significant power dissipation on the highly capacitive long wires, the use of the capacitance of the long wires has been proposed to hold the VN state. In this way, the analog VN state will be broadcasted and the sign will be generated locally at the inputs inside the CNs. There is still the scope of mismatch between the sign inverters in one CN to those in another. In [18], the mismatch in inverter threshold for minimum sized inverters has been reported to have a standard deviation of approximately 13mV. Before the

decoding starts, the VNs are initialized with the voltage levels corresponding to the channel LLR data. The LLR data, being unbounded are first clipped at a maximum and a minimum value, and then quantized into discrete voltage levels ranging from 0V to 1V. In [18], the LLR data found from the channel have been clipped at +10.5 and -10.5 for maximum and minimum levels respectively. Therefore, 13mV standard deviation in 0V to 1V range will translate into 13m * [10.5-(-10.5)] = 0.27 in LLR domain. This deviation can lead to improper parity-check satisfaction or decoding failure. Figure 2.7 also suggestes that the increase in random threshold inverter mismatch can be the most significant contributor to BER degradation.

Finally, a transistor level design was done in TSMC 65nm CMOS for FG(273,191) code. The simulation, which includes the above analog impairments, reports energy consumption of 0.56 pJ/bit and a throughput of 137 Gb/s.

A number of changes were made in the python code, originally written by Kevin Cushon [13]—to simulate the performance of the analog decoder. The quantitative and relative inclusion of current skew, threshold mismatch, and inverter mismatch leads to different BER characteristics that can be obtained from the modified python code [18]. The following Figure 2.7, reported in [18], shows simulated bit-error rate resulting from different magnitude of non-ideal behavior.

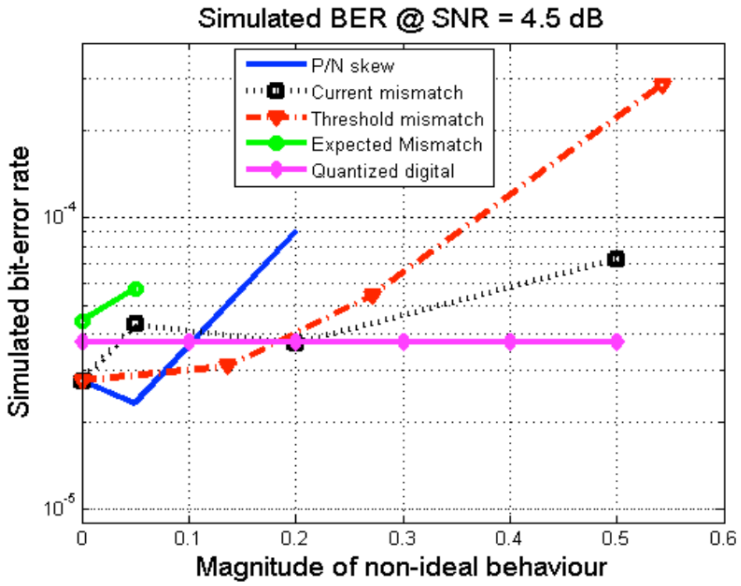


Figure 2.7: Simulated bit-error rate at 4.5 dB SNR for different analog mismatch [18].

Chapter 3

Effects of Analog Impairments and the Proposed Solutions

This chapter starts with the revision of the modified python code to analyze and document the comparative contribution of the different analog impairments. Later in the chapter, the proposed solution to these drawbacks will be discussed. It is also noteworthy that from this section starts the contribution of the writer towards the thesis.

In the modified python code, the BER performance of the FG(273,191) LDPC code will be observed. The codeword is generated from a pre-specified generator matrix, G, whose size is 191 × 273. A random binary word of length 191 is generated. Then the word is multiplied with G^T to obtain an LDPC codeword, c of length 273. In this codeword, 191 bits are the information bits and the remaining is the redundant parity-check bits. As soon as the codeword is generated a syndrome check is performed with the parity-check matrix to ensure that the codeword is valid, i.e., $c.H^T = 0$. Once there is the codeword, a Gaussian random noise corresponding to a particular SNR is added to the codeword to imitate the effect of an AWGN channel on the transmitted codeword. The VNs and the CNs are already constructed. So, now the binary message passing between the nodes starts until the codeword is decoded.

The simulation as described above predicts performance of a digital decoder. To simulate the analog behavior out of it, some changes have to be made in python simulation to enable the VN state make fine-grain update of its discrete-time values. From equation (2.4), it is obvious that the variable node, once initialized by LLRs at the start of decoding, will continue to update itself in the subsequent iterations. The LLR values fed to the digital decoder are not just the raw channel log-likelihood ratios. It is, rather, in a quantized form. The amount of update in VN memory at each successive iteration depends on two aspects: (i) the distribution of the incoming CN messages. The more a particular message (0 or 1) gains majority, the more the summation in equation (2.4) will be. For the constant scaling parameter ‘s’, the maximum change to the VN

memory occurs when all the connected CNs broadcast either 0 or 1. (ii) The value of the scaling parameter ‘s’. This parameter can scale down the changes in VN memory. While a large value of s ($0<s\leq1$) can enable the update of VN memory to take long strides, s as a small fraction can lead to fine-grain changes in VN memory. These fine changes in VN memory update imitate the continuous-time dynamics of the analog decoder.

The termination criterion, both in the digital [13] and the analog [18] implementations of MDD-BMP, has been selected to be the parity-check satisfaction instead of the syndrome check. It made us initially cautious that the parity-check satisfaction may lead to satisfaction with wrong codewords in some cases when simulations are run with a lot of (~10k) codewords. So, some changes were made in the code to implement a full termination by comparing the initial transmitted data to the decoded codeword. This required us to transfer initial data from another file to the one where the decoded output is obtained through function argument. The comparison between the initial and the decoded codeword gives us the error rates. Some addition in codes were also made to figure out the error indices.

A lot of cases have been seen with very small ‘s’ parameter, where there are only one or two errors. To investigate this sort of errors, a generated code was written down in a file and the code was read from the file when simulating. Otherwise, since codewords generated at each simulation are random, there is no way to further look into a specific error VN trajectory for different values of ‘s’ parameter. This is why the code has been further modified to be able to write a generated codeword during one random test and to read from it in the other times during the fixed tests.

Now, different steps taken to reduce the drawbacks stemming from the analog implementation limitations will be discussed. Before going into it, let us look into very closely-stepped discrete-time dynamics of the analog decoder from python simulation. As the scaling parameter becomes smaller, the decoder behavior tends to be very closely-spaced discrete in nature or the decoder tends to show almost continuous analog dynamics. To understand how the analog impairments impact the analog decoder dynamics, some particular values of the impairments are taken for the Python simulation. The standard deviation of random mismatch in current-source inverter currents has been taken to be 0.05, which is the maximum value observed for this inverter from monte-carlo simulation. Also, the threshold inverter mismatch obtained

from the monte-carlo simulation has a standard deviation of 13 mV. Since the calculated LLRs are clipped at -10.5 and +10.5, this range corresponds to a voltage range from 0 to 1 V. Therefore, in LLR domain, the standard deviation of 13 mV corresponds to 0.8. Using the above mentioned values of impairments, the BER has been calculated at 4.5 dB SNR for different values of scaling parameters and the results have been put in Figures 3.1, 3.2, and 3.3.

From Figure 3.1, it can be seen that the almost continuous decoder has a very high BER overall. When the contributions of the impairments behind it are separately shown, it is obvious that the mismatch between the threshold inverters is the most dominant contributor to the overall BER. The other two contributors are much less significant since the graph of their contribution almost coincides with the plot in case of no mismatch at all.

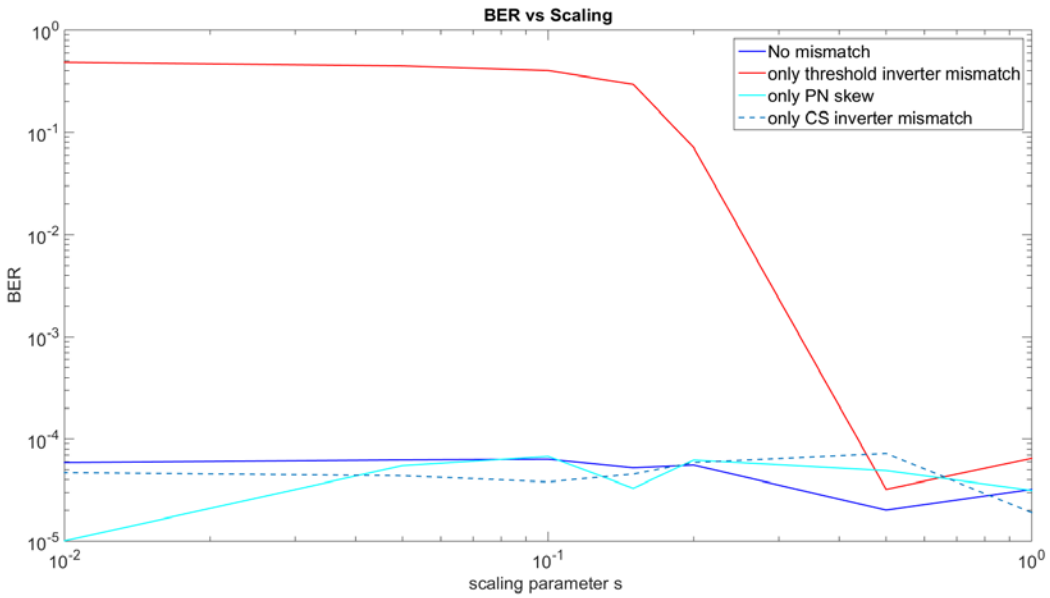


Figure 3.1: BER vs scaling parameter s.

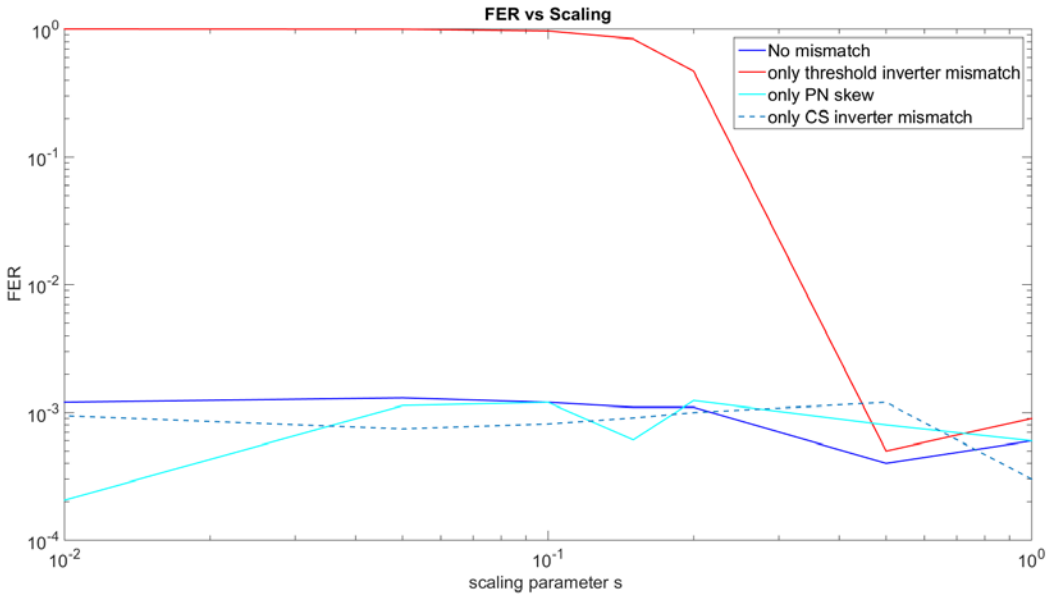


Figure 3.2: FER vs scaling parameter s.

Similarly, Figure 3.2 shows the frame-error rate (FER) vs the scaling parameter s. Each frame is a codeword of length 273. Therefore, the FER states how many codewords out of the total codewords simulated could be decoded error free. Again, in Figure 3.2, the random mismatch between threshold inverters comes out as the most significant source of FER degradation. In Figure 3.1, 3.2, and 3.3, the full termination criterion considering parity-check satisfaction was used.

Since a small scaling parameter means heavy scaling down of the sum of the incoming CN messages to the VN, the update of the VN will be much slower per iteration. This, in turn, requires more iterations to reach the decoded output. Figure 3.3 depicts the required number of iterations vs scaling parameter s. It is noteworthy that to save the tremendous time demanded by the simulation of 10,000 codewords in Figure 3.1, 3.2, and 3.3, the maximum iterations of the simulations have been set to be 1000.

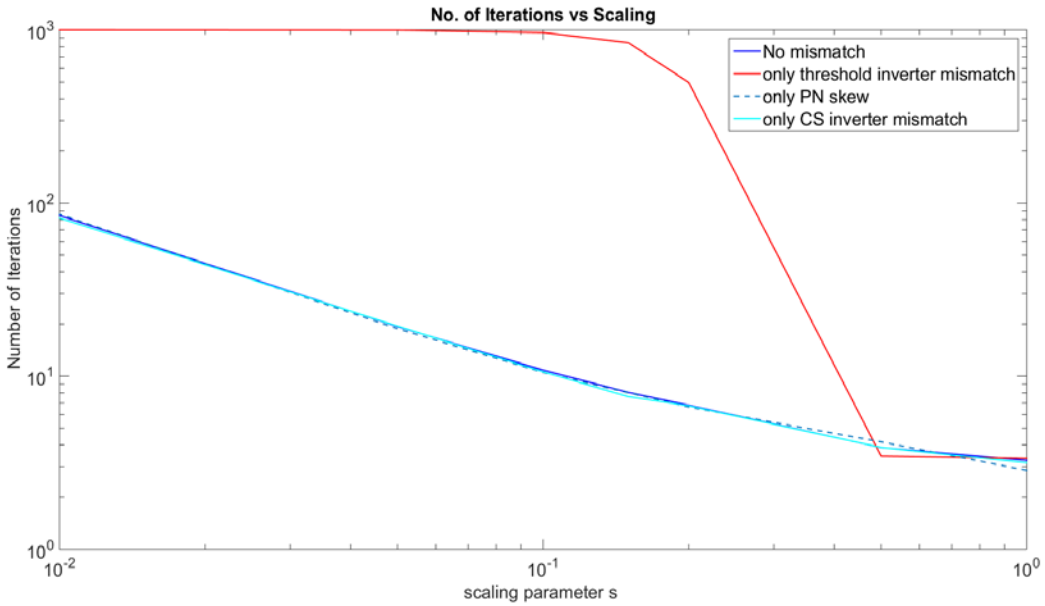


Figure 3.3: Number of iterations vs scaling parameter s.

It has already been discussed in the previous chapter that finite output resistance of the current-source inverter, leading to 4^{th} order polynomial mapping VN state to message weight, does not have any discernible effect on the decoding performance. So, there is no need of further investigation into this.

The systematic skew between the PMOS and the NMOS inside the current-source inverter is a problem that a well-designed current-source inverter cannot fix. The reason behind is that the skew is attributed to the process variation during chip fabrication. Therefore, in addition to the design of well-calibrated inverters, there has to be the calibration circuitry in the chip as well so that it can be calibrated during the testing phase. Design of the current source inverters for on-chip testing requires the change of their simple inverter structure so that the current supplied by the inverters can be tuned from off-chip. These tunable inverters have to be calibrated on-chip for equal strength in current sourcing and current sinking. Most importantly, the tuning voltages have to be routed to reach all the current-source inverters present in the whole decoder chip. In this way, it will be possible to escape any systematic skew during the process variation.

Now, any systematic skew between threshold inverters used for sign generation due to process variation can be fixed in the same way – on-chip calibration and routing of the calibrated

tuning signal. The calibration setup has been discussed in detail in the next chapter. In short, the rail PMOS gate signal from off-chip is supplied and a high gain OpAmp is used to generate the gate voltage for the rail NMOS of the threshold current-starved inverter structure. This tuning is precisely done so that for an input of VDD/2, the output of the inverter is also at VDD/2 (=500 mV). The tuned gate voltages of the rail transistors are then routed to all the places where there are threshold inverters on the chip.

Another limitation in analog design is the problem of delay matching. This problem is not evident unless the schematic design of the analog decoder is investigated. In the digital implementation, the update of the VN is done in synchronization with the clock. As the VNs are updated, the CNs calculate their values before the clock ticks for the next iteration. However, the analog implementation is fully asynchronous. So, the time taken for a VN output to move into the VN's own de-embedding XOR gate is much less than the time taken for the broadcasted VN output to move through the CN and come back to the de-embedding input. This delay mismatch creates undesired oscillation when the VN trajectory is crossing the VDD/2 level from either side. Suppose, a VN state is crossing the VDD/2 level from below. As it crosses past the VDD/2 level, the input of the de-embedding XORs get this update almost instantly, just delayed by a pair of sign inverters. On the other hand, the propagation of the VN output broadcast through the CN is still far in terms of time. Previously, the de-embedded message was that the VN was in the wrong side and so there was a push to move it to the correct side. As the VN trajectory is getting just past the VDD/2 level, de-embedded output will just flip and instead of reinforcing the VN in its now correct side, they will indicate that the VN, after crossing VDD/2, is still on the wrong side and needs to move back below. As the VN gets pushed below VDD/2, the updated CN messages now come to pull it high again. This is how some VN states gets stuck close to VDD/2 region and may or may not be able to recover themselves. If they are able to eventually move to a side, they have delayed the decoding time, and caused lots of CNs to flip a few times leading to more power consumption. In the next chapter, this phenomenon is described in detail. In short, to alleviate this trouble, delay tunable pair of current-starved inverters is needed to be inserted between the output of the VN memory and the input to the sign inverter pair. Then, delay is tuned to match both the VN output to de-embed input delay, to the delay from VN output through CN to VN input back. Simulation in Cadence shows that the decoder is able to tolerate

little mismatch in delay while good matching can potentially reduce latency and energy consumption.

Finally, it is evident from the Figure 3.1 that the random mismatch between threshold inverters has the most adverse impact on the BER performance of LDPC decoder. Interestingly, in the implemented design, the effect of the random mismatch can be minimized due to the matched delay in the de-embedding path. Since the python simulation does not include this delay in its calculation, the actual dynamics of the decoder performace is not fully evident from the Python simulations. While simulation in Python, it is assumed that anytime the VN trajectory crosses VDD/2 voltage, the CNs instantly re-calculate their decision based on the updated VN's sign. However, in practice, there is a certain amount of delay between VN's broadcasting of its analog value and CNs' outputting the updated decision. In the previous paragraph, this delay and the need for delay matching has already been discussed. Due to this delay, a VN that crosses the VDD/2 line will not be able to instantly get the updated decision from the connected CNs. Therefore, it will continue to move in the original direction crossing VDD/2 for some time until other CNs either re-inforce its direction or oppose it. During this time, the VN trajectory has moved significantly past the VDD/2 line. Therefore, it is very unlikely that the little amount of random mismatch between threshold inverters will be able to come into effect, which it would do in case of little or no loop delay. From this discussion, the random mismatch is expected to have much less effect compared to what has been forecasted by the no-delay Python simulations.

As different structures were used for the threshold inverter and the current-source inverter in our decoder design, it is important to determine the amount of analog impairments in this design. After running monte-carlo simulation in Cadence Spectre, the standard deviation of random threshold mismatch in threshold inverters are found to be 12.8797 mV, which is almost equal to the 13 mV reported in [18]. Furthermore, the random mismatch between PMOS and NMOS in current-source inverters has a standard deviation of 0.0509 compared to 0.05 reported in [18]. These almost equal amounts of impairments in both this design and the previous design is not surprising due to the fact that both designs used almost minimum-sized transistors. Now that the amounts of impairments are almost exactly equal, the comparative contribution from Python simulation is also almost equal. It is important to note that the proposed solutions in the new design are expected to eliminate the systematic offsets and improve bit-error rate

performance post-fabrication. Due to the nature of analog dynamics, it cannot be simulated in Python taking all the delay matching issues into consideration. However, in the next chapter, it will be shown that the monte-carlo simulation of the full decoder in Cadence is able to provide successful decoding taking all these mismatches and skews into account.

Chapter 4

Schematic Design of the Decoder

The designed chip includes not only the core analog decoder but also a number of circuit blocks for testing purpose. The core decoder is made up of two important building blocks, namely the variable node and the check node. The design of the check node is comparatively simpler; it includes the design of a 17-input XOR. The variable node has 17 pairs of threshold inverters to generate the sign of the analog VN wire at different locations, 17 delay-insertion inverter blocks, 17 de-embedding XOR gates, 17 current-source inverters, and 17 transmission-gate switches connecting the current-source inverter outputs to the analog VN wire. The pair of threshold inverters inside VN is also used by the corresponding CNs that enables us to save some space. The wires for global tuning of the threshold inverters, delay-tuning inverters, and the current-source inverters run vertically encompassing the full height of the decoder and are connected horizontally at regular intervals to reduce the resistance of the network as signals propagate from the bondpads into the middle of the decoder. The decoder is also full of a lot of wiring connections between the VN and the CN since each VN is connected to 17 CNs, and similarly, each VN is connected to 17 VNs.

The circuits needed for testing includes shift registers from data input and output, a memory array made up of SRAM cells to store data bits, digital to analog converters (DAC) to generate quantized voltage levels, termination-check circuits to indicate parity-check satisfaction, and an analog MUX to investigate the analog voltages of the VNs for the debugging purpose. There are also a few shift registers to supply control signals for memory block selection and the analog MUX. Before the design requirements and procedures of individual circuit blocks are discussed in detail, it is important to understand their placement and connections to the neighboring blocks. The following Figure 4.1 illustrates the placement of all the blocks inside the chip.

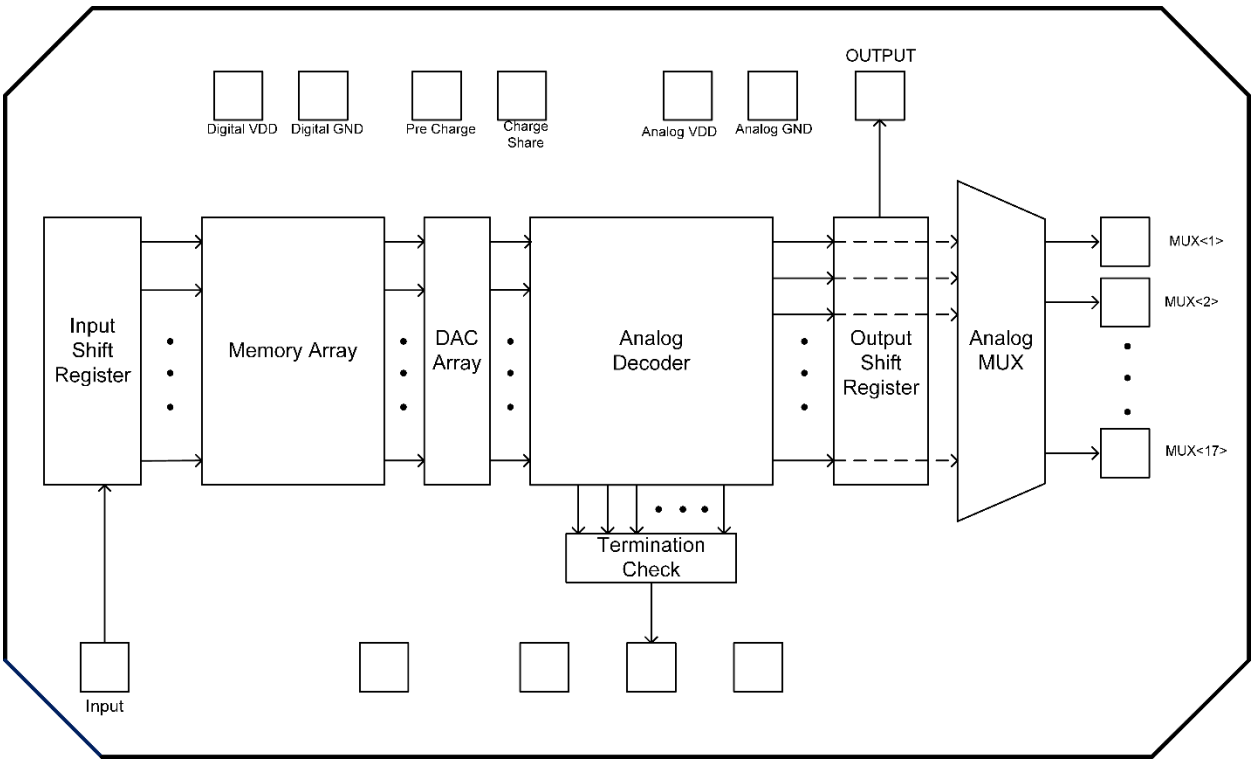


Figure 4.1: A simplified view of the circuit blocks placement and connections inside the designed chip.

The schematic design and simulations of the different modules of the chip and the whole decoder and testing circuits were done in TSMC 65nm CMOS technology. In this technology, the minimum length for transistors can be 60nm and there are 9 metal layers. The reservation of these metal layers will be discussed in detail in Chapter 5 of this thesis, which discusses the physical design.

4.1 Check Node (CN) Schematic: Delay Matching Through All Possible Paths inside CN

Check nodes (CNs) are one of the two major building blocks of the core analog decoder; the other being the variable node (VN). Unlike the VN, the structure of the CN is more straightforward – a large 17-input XOR gate and some inverters preceding and following the inputs and the output of the gate respectively.

Figure 4.2 depicts a simpler schematic of the check node. A large 17-input XOR gate is at the center of it. All these inputs come from the signs of the 17 corresponding VNs. Since the VNs in the decoder represent different analog levels, their sign will be generated through a pair of tuned current-starved inverters for thresholding purpose. So, each of the pairs of inverters are properly tuned so that an input of any voltage above VDD/2 or below VDD/2 will yield VDD or GND respectively at the output of the inverter pair. On the other hand, the single inverter connected to the output of the XOR gate is not a threshold inverter. Rather, it is a simple inverter with multi-finger PMOS and NMOS to provide high drain current for buffering purpose. Furthermore, it will be shown later that the long-running vertical CN wire also requires a good amount of current to charge up its parasitic capacitance in the physical design. Therefore, a high-current-providing buffering inverter has a role in the faster propagation of the CN output signal.

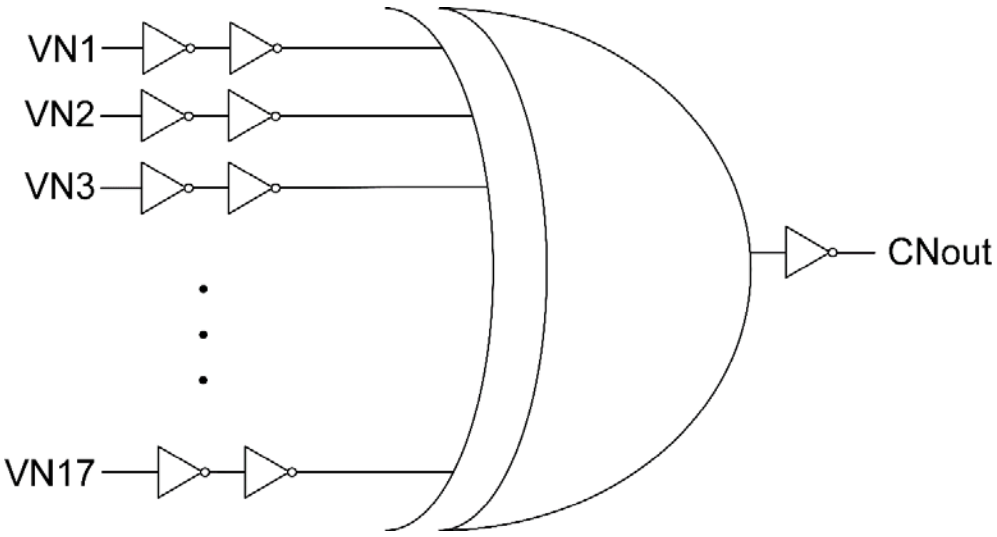


Figure 4.2: Check node Schematic.

Now, how the 17-input XOR gate was built up by choosing the proper structure from a few possible ones will be discussed. Let us start building the XOR with smaller XOR gates, as shown in Figure 4.3.

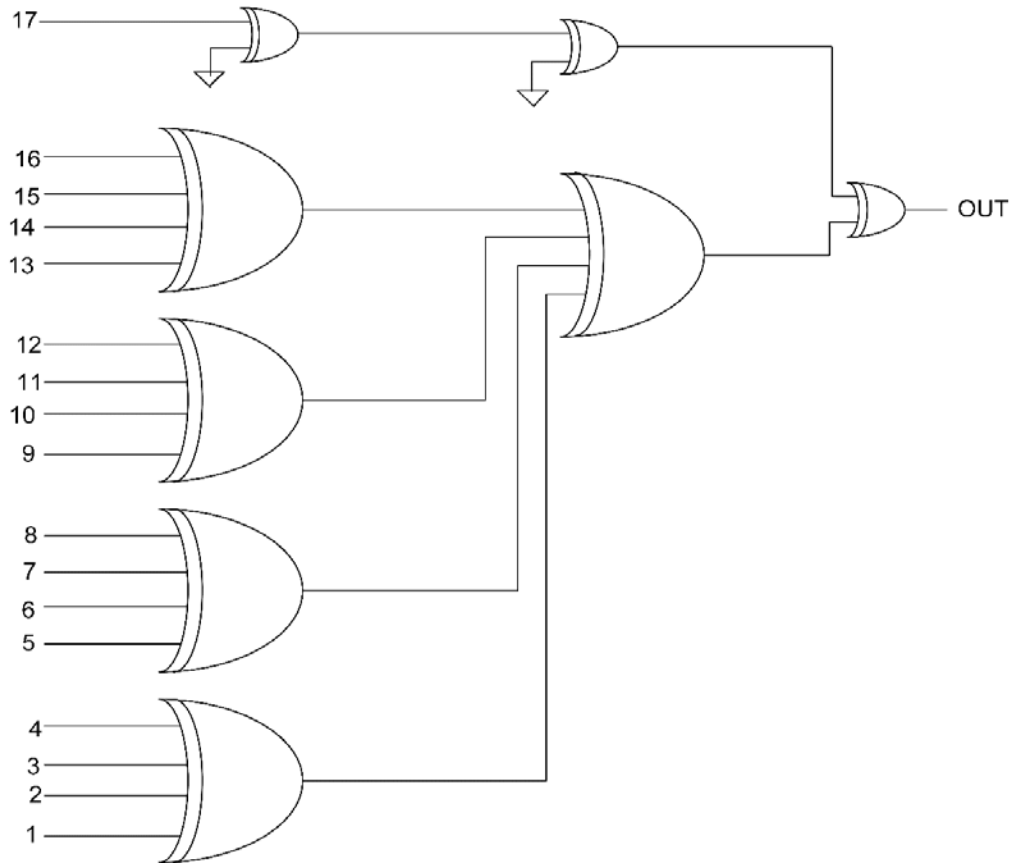


Figure 4.3: A possible implementation of 17-input XOR gate (Not the implementation in our design)

Here, both 4-input and 2-input XOR gates have been used to build up the 17-input gate. Two extra 2-input XORs have been used in the 17th input path to insert delay, since all other input signals are also propagating through three gates to reach the output. While using this structure gives proper decoding in simulation, there are a couple of issues to consider. First, the

signal propagation delay through the 2-input gate and 4-input gate are not exactly equal. Although the 4-input gate is not made up of cascaded 2-input gates, different nodes in the signal path have more capacitance. In an ideal case, it is not desired to have different delays in different signal paths. Second and more importantly, this structure in Figure 4.3 is difficult to implement in the physical design. In the layout, signal paths encompass much longer distances due to the sparse nature of the LDPC description matrix. Therefore, a symmetric structure is desired where each signal path will be propagating through the same length of metal wires in the implemented design. Furthermore, a 4-input XOR gate is difficult to accommodate in an area-optimized design of the large decoder. Therefore, the schematic shown in Figure 4.4 has been used for our design.

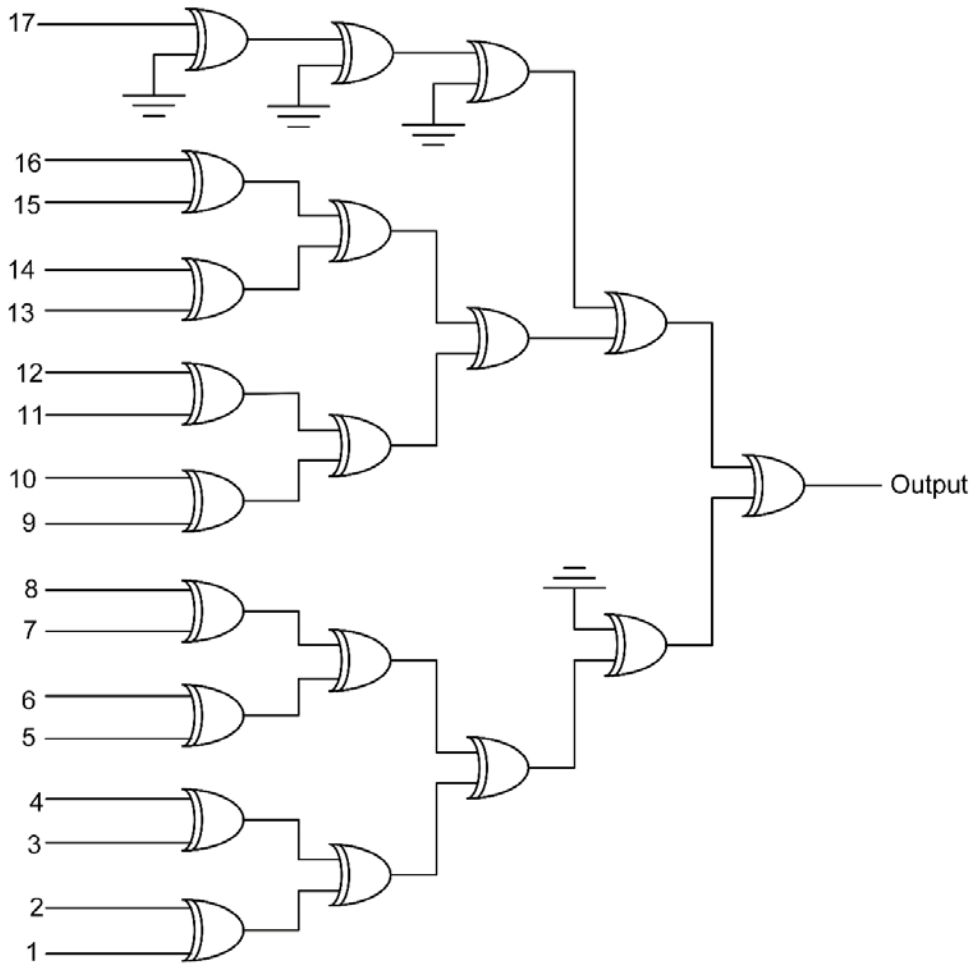


Figure 4.4: Schematic of 17-input XOR gate.

As can be seen in Figure 4.4, CN XOR gate in our design consists of small 2-input XOR gates cascaded together. A signal from any input propagates through five 2-input XORs to reach the output. So, in this structure, the latency through any path is exactly equal. On the other hand, vertical distribution of these small 2-input XOR blocks is symmetric around the vertical mid-point in the physical design. Therefore, the intermediate metal wiring for each path is almost equal.

The delay of the 4-input XOR gate is more than that of the 2-input XOR gate. However, unless the 4-input XOR is made up of cascaded 2-input XORs, the compact design of 4-input XOR has less delay than twice the 2-input XOR delay. Even after taking this into account, the delays through all the 17 input to output propagation paths in Figure 4.4 will translate to slightly more delay compared to the structure shown in Figure 4.3. This small additional delay will not significantly slow down the decoder. Also, delay matching through different paths is a more pressing issue than a slight increase in the delay for successful decoding performance.

The 2-input XOR gate used in our design has the structure as shown in Figure 4.5. The inverters and the transmission gates first implement an XNOR gate and finally an inverter is placed to turn this XNOR output into an XOR output. The reason behind placing inverter at the last step of the XOR design is to increase the 2-input XOR's drive strength. In physical design, this structure is suitable for speedy propagation of signals through cascaded XOR gates due to good drive strength.

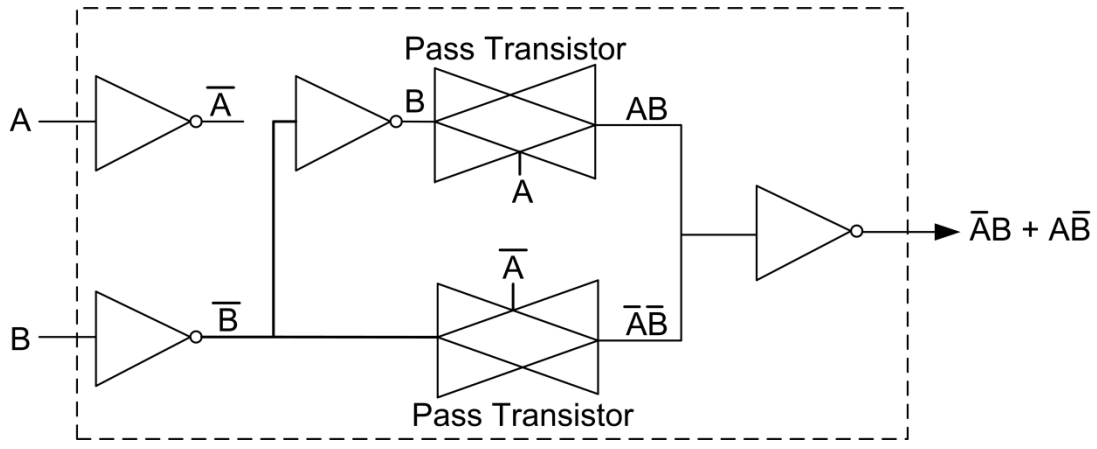


Figure 4.5: 2-input XOR gate schematic that has been used in this design.

4.2 Tuning and Calibration of Threshold Inverter

The contribution of different sorts of mismatch on the bit-error-rate (BER) performance of the analog decoder has been shown. This was observed with the help of Python code by looking into the discrete-time behavior of the decoder at very small time steps. The different types of analog impairments include threshold mismatch between the thresholding inverters, the PMOS and NMOS drain current skew in the current-source inverters, and the random mismatch between current-source inverters.

Now that the concern of systematic offset between threshold inverters has to be fixed, the tunability of current-starved inverter structure was utilized. The difference between this structure and a simple inverter is that there is a flexibility to calibrate this structure by controlling the voltage applied at the gates of rail PMOS and NMOS. On the other hand, a simple inverter has no parameter to control it from outside the chip, therefore, leaving no way to control the systematic skew post-fabrication. This skew will be potentially fatal to successful decoding. Figure 4.6 shows a current-starved inverter structure.

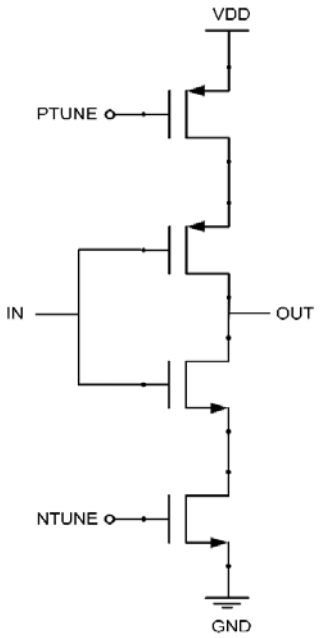


Figure 4.6: Schematic of current-starved inverter.

After this structure has been chosen as the threshold inverter structure, this will be calibrated on-chip to address the systematic skew. There are several parameters at hand to make this calibration. They include the tuning voltages PTUNE and NTUNE, and the dimensions of rail PMOS and NMOS. First, the dimensions of all the MOS are chosen. All of them are of minimum length, which is 60nm. The PMOS and NMOS widths have been chosen as 260nm and 195nm respectively. As can be seen, the dimensions chosen are small. The places where the threshold inverters have been used, there is no need any high load buffering requirement or the inverter does not need to drive a high capacitance node at the output. Therefore, small dimensions are okay and also convenient for minimizing physical design area.

The random mismatch of this threshold inverter of the parcular chosen dimensions has been obtained from monte-carlo simulation in Cadence. Figure 4.7 shows the schematic used for obtaining the random mismatch. 1000 monte-carlo simulations provide a mean of 500.961 mV and the standard deviation of 12.8797 mV. All the deviations from monte-carlo simulation are plotted in Figure 4.8, where maximum and minimum deviations are also indicated. Figure 2.7 from chapter 2 shows that this quantity of random mismatch will not have significant effect on the bit-error rate.

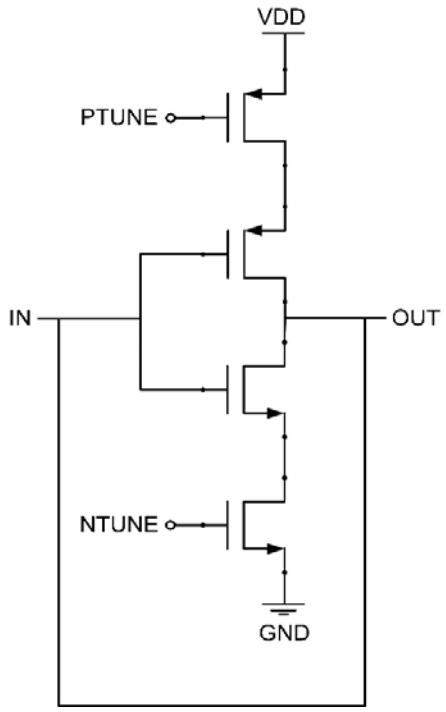


Figure 4.7: Schematic for determining random threshold inverter mismatch.

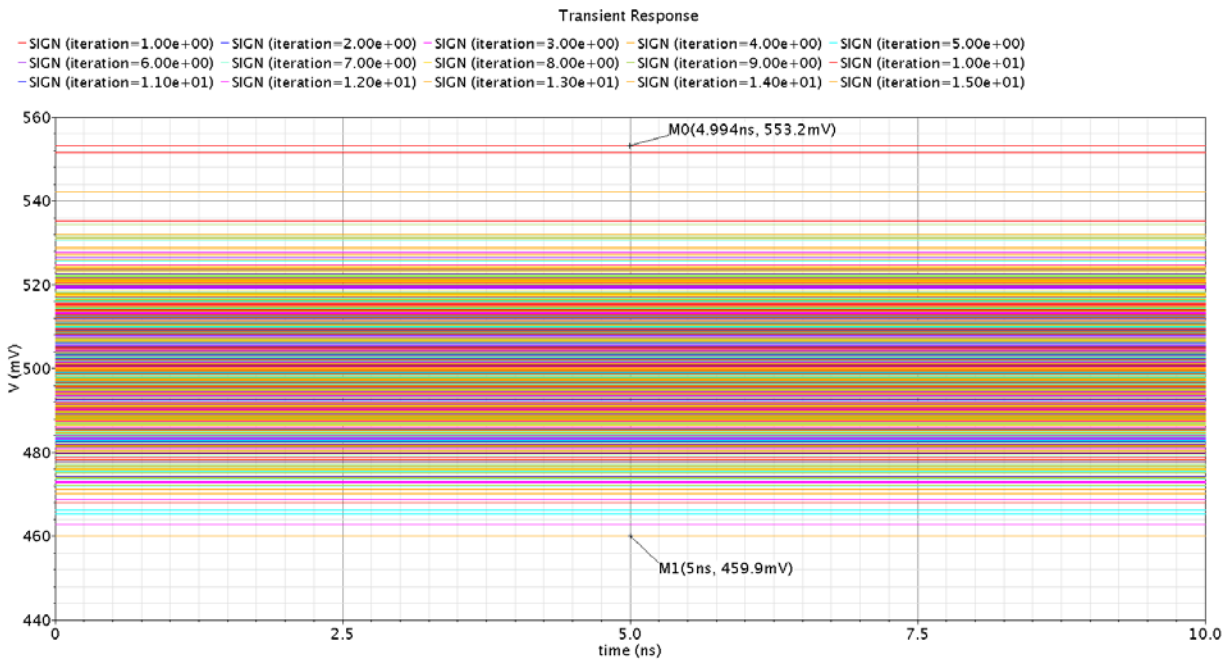


Figure 4.8: Monte-carlo simulation results showing deviation from ideal threshold VDD/2.

Now, the threshold inverter will be calibrated so that for an input voltage of VDD/2, the output voltage is also VDD/2. A dimension has already been selected for the MOSs inside the threshold inverter. So, the parameters left are the PTUNE and the NTUNE voltages, i.e., the gate voltages of the rail MOSs of the threshold inverter. The need for tuning these voltages to calibrate the threshold inverter in the fabricated chip to alleviate the systematic skew in the real world situation has already been discussed. In our design, one of these two voltages will be provided off-chip and another one will be generated utilizing a high gain voltage-controlled voltage source (VCVS). The VCVS has been implemented with a high gain OpAmp, which needs to be stable but does not need to be fast. This is because the PTUNE will be supplied off-chip and calibrate and generate the NTUNE before starting decoding in the core decoder.

Figure 4.9 illustrates the calibration setup of the threshold inverter. The input of the inverter is connected to VDD/2 independent DC voltage source. The output of the inverter is compared to VDD/2 at the inputs of VCVS. Vempirical is an estimation of NTUNE. However, with a sufficiently high gain of the VCVS, Vempirical can be far from the actual value of

NTUNE. In simulation, output voltage is compared to VDD/2 and once it is there, the generated NTUNE voltage can be obtained.

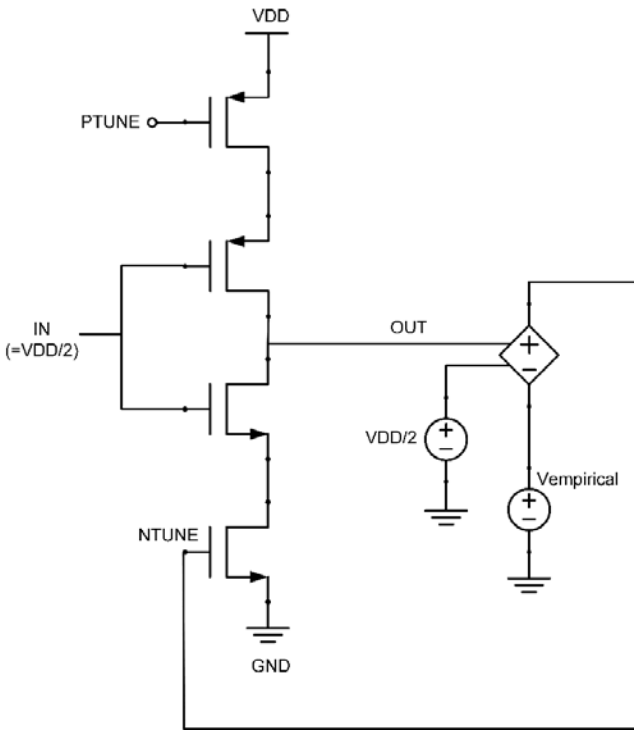


Figure 4.9: Calibration scheme of the threshold inverter.

In our design, there is a calibration circuit on chip, where VDD/2 as input and PTUNE are fed off-chip. Then both the supplied PTUNE and the generated NTUNE are routed to all the positions where there is a cascaded pair of threshold inverters.

At this point, a high-gain OpAmp will be designed and placed replacing VCVS. Figure 4.10 shows the threshold inverter calibration scheme with OpAmp. After that, Figure 4.11 shows the schematic of the folded-cascode OpAmp that has been used.

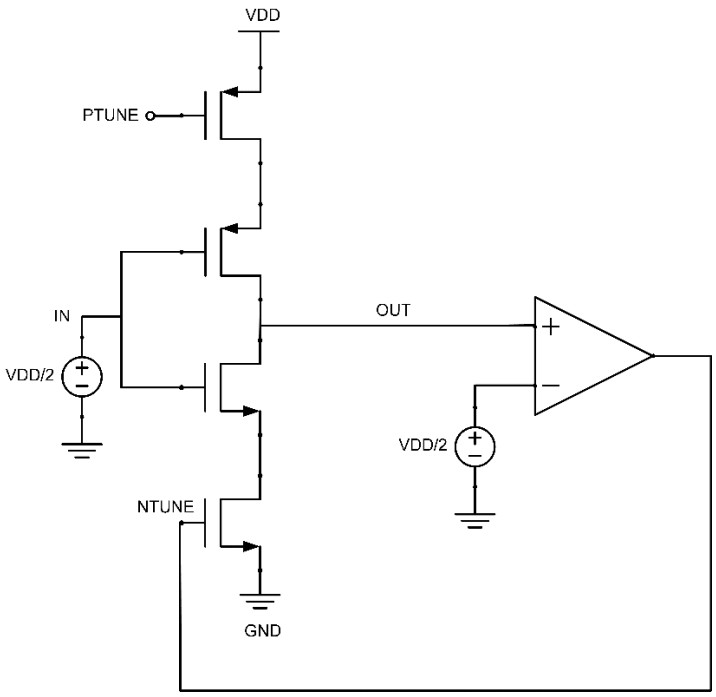


Figure 4.10: Calibration scheme of the threshold inverter using OpAmp.

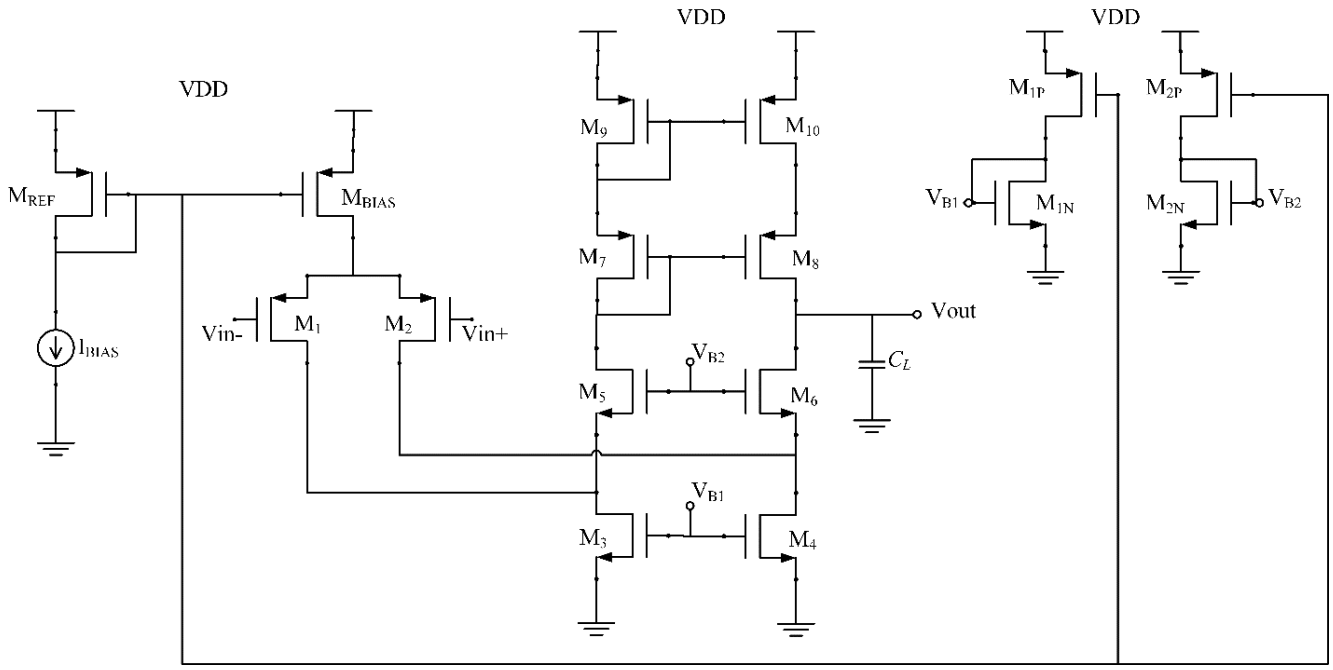


Figure 4.11: Folded cascade OpAmp schematic. This OpAmp has been used as a Voltage-controlled Voltage Source (VCVS). The same OpAmp has been used as a buffer in Analog MUX. (C_L=200fF, I_{BIAS}=20µA)

TABLE 4.1

Transistor dimensions and small signal parameters

Transistors	W(µm)	L(nm)	g_m(µA/V)	g_{ds}(µA/V)
M_{REF}	25	500	340.8	5.745
M_{BIAS}	27	500	342.3	13.04
M1	20	500	190.6	1.891
M2	20	500	190.6	1.891
M3	6.5	500	319.2	12.96
M4	6.5	500	318.9	13.14
M5	3	500	156	5.086

M6	3	500	153.9	7.734
M7	20	400	195.4	3.579
M8	20	400	195.4	3.384
M9	20	400	194.8	3.856
M10	20	400	194.8	3.853
M_{1N}	1.51	100	252.5	10.17
M_{1P}	4	100	230.1	11.84
M_{2N}	0.200	500	40.01	1.09
M_{2P}	2.01	100	102.9	6.105

The addition of 200fF capacitor at the output node of the OpAmp is to increase the stability only. Since high speed is not needed in either case of using OpAmp as VCVS or using OpAmp in analog buffer, addition of this capacitance is justified in terms of speed.

High open loop gain is essential for precise generation of the NTUNE voltage. Figure 4.12 shows the open loop magnitude response of the OpAmp. At low frequency, the open loop gain is 48.47 dB. In other words, the open loop gain is 265 V/V. More gain can be achieved by adding a second stage to this OpAmp. However, this amount of gain is capable to generate the NTUNE voltage upto one decimal point in precision. As an example, if PTUNE=0, the generated NTUNE from the scheme in Figure 4.10 is 493.7 mV, which is the same as that in a high gain VCVS. It is, however, more important to find out the achieved threshold in this calibration scheme where the target threshold is 500mV. From simulation, the achieved threshold using this OpAmp is 499.6 mV. This PTUNE=0 V and NTUNE=493.7 V have been used to tune the threshold inverters in the simulation. During the testing of the chip, PTUNE=0V will be suypplied from off-chip and the NTUNE voltage will be generated from the calibration circuitry and then it will be routed globally.

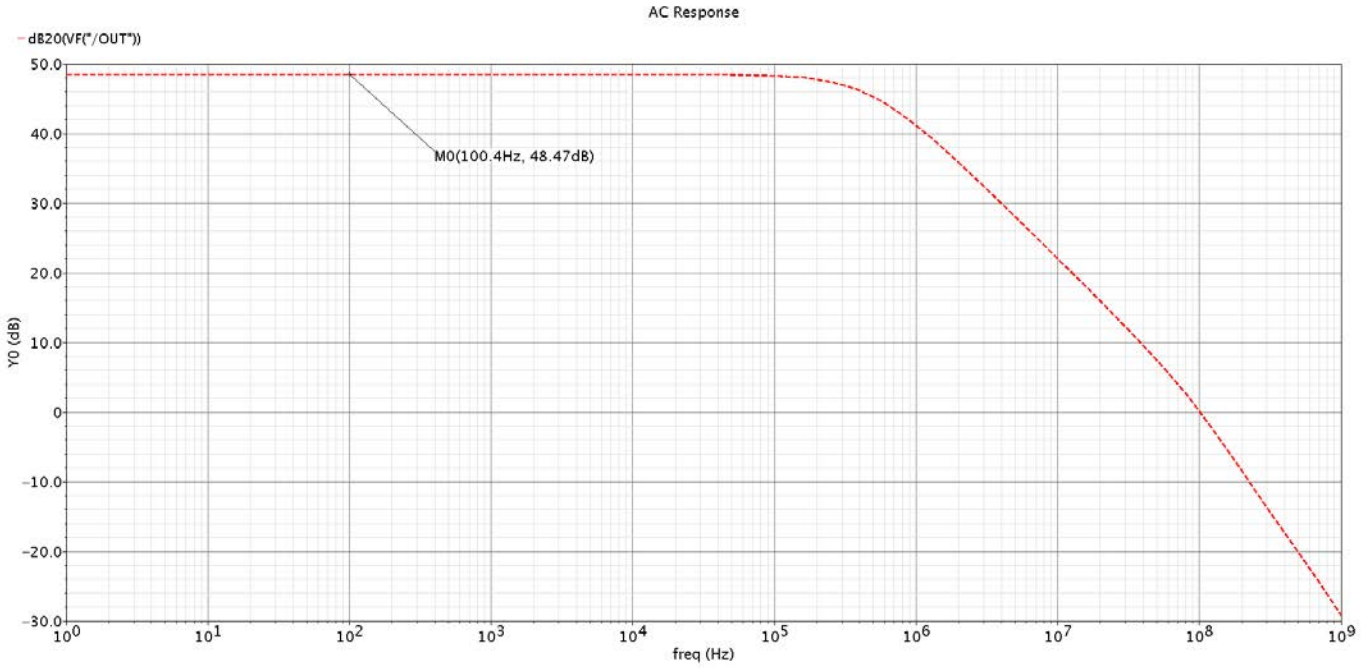


Figure 4.12: Open-loop magnitude response of the OpAmp. At low frequency, the gain is 48.47 dB.

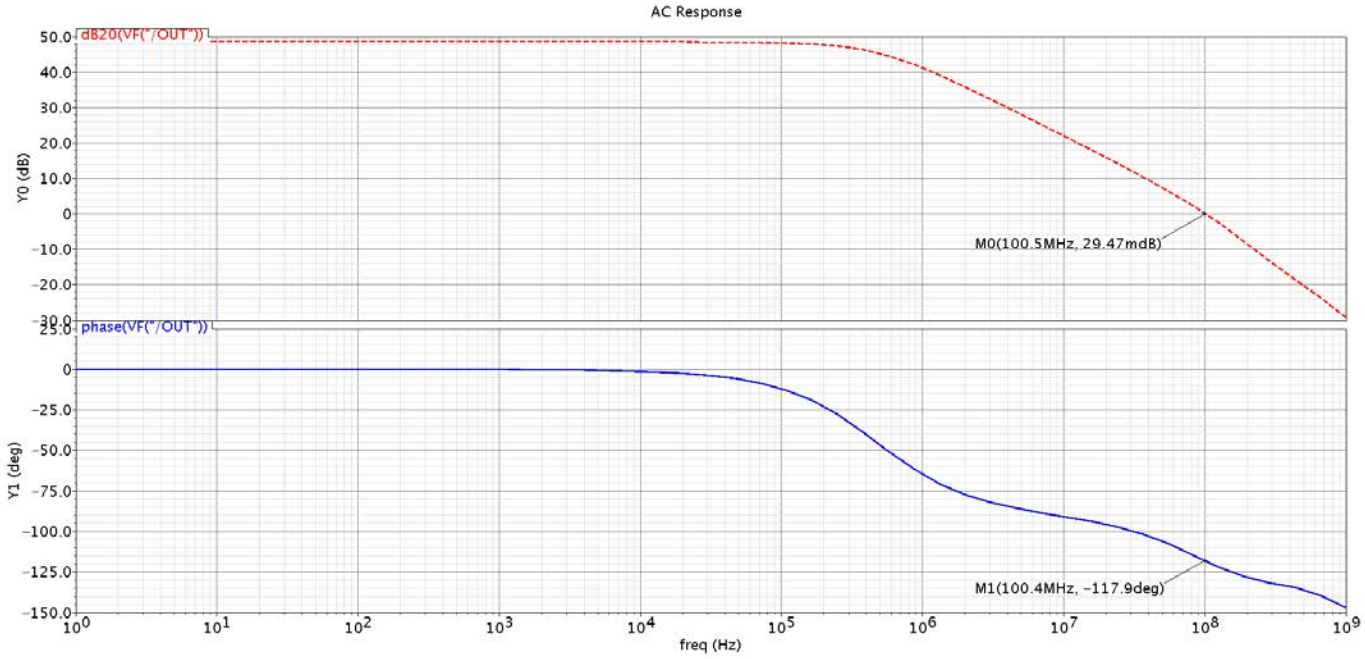


Figure 4.13: Open-loop magnitude and phase response of the OpAmp. At unity gain, phase = -118 degree. Therefore, Phase Margin = -118-(-180) = 62 degree.

4.3 Insertion of De-embedding Delay

The Variable Node (VN) wire acts as a parasitic capacitance and holds an analog voltage level. This analog level is broadcasted to the corresponding check nodes (CN) the VN is connected to. In our particular design, any VN is connected to 17 CNs. Therefore, each VN's sign is generated by a pair of tuned threshold inverters inside each of the 17 corresponding CNs. Now, each CN calculates whether they are parity-check satisfied. If satisfied, the output of a CN will be '0', and if not, it will be '1'. The digital output of these 17 CNs is now fed to the VN. So, it can be seen that there is a loop starting from a VN output to the CN input, going through the CN, and finally from the CN output to the VN input. This loop has a certain amount of delay due to the node capacitances through the path and the path consisting of multiple gates inside the CN.

On the other hand, inside each VN, the VN's own output is thresholded using a pair of tuned threshold inverters and the sign is compared to the CN output. The circuitry that does this comparison is called the de-embedding circuitry. So, the VN output to the de-embed input has little delay – only the delay due to the pair of threshold inverters. However, in theory, there is a need to compare the sign of a VN to the broadcasted VN-based CN outputs. In other words, the delay from the VN output through the CN to the VN input has to be equal to the delay from the VN output through the threshold inverter pair to the VN input. This essential condition will not naturally take place unless additional delay is inserted inside the VN and match both the delays. Figure 4.14 illustrates these two paths through which the delays have to be matched. The shorter delay path is shown in red while the longer one is shown in blue. Figure 4.15 depicts the location of additional delay insertion inside the VN.

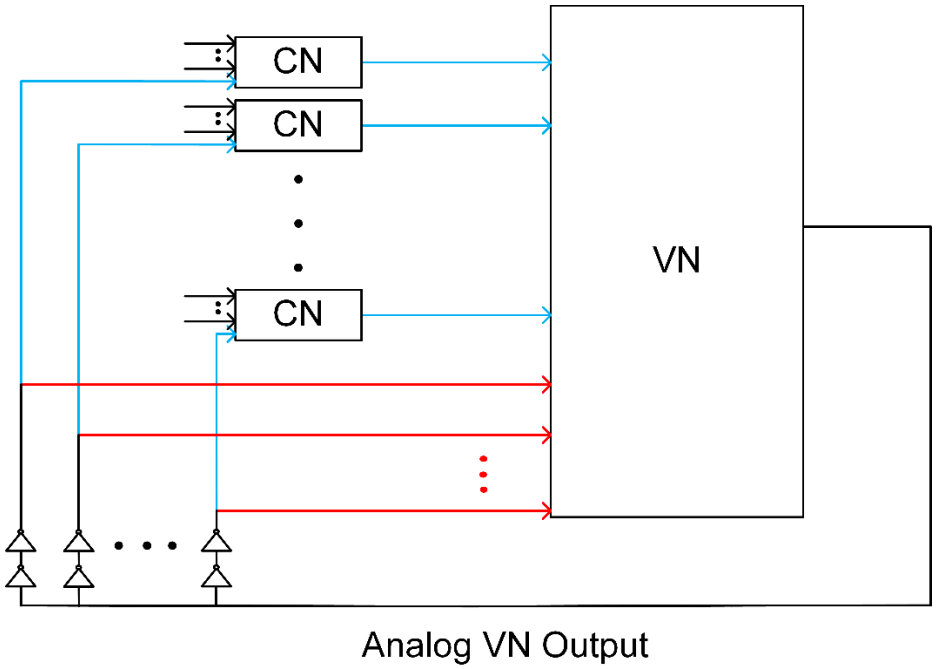


Figure 4.14: Delays through the red path and the blue path have to be matched.

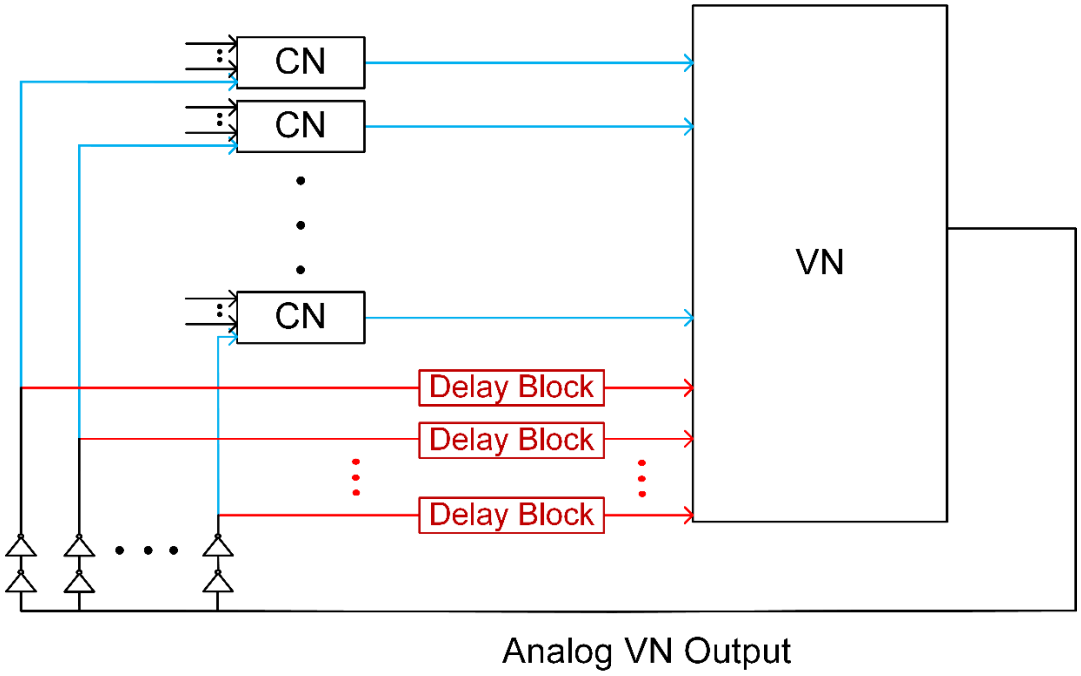


Figure 4.15: Variable Node (VN). Insertion of de-embed delay after sign generation of VN.

Now that the necessity of inserting additional delay has been established, the circuit behavior in the absence of proper matching of both delays will be discussed. Also, Figure 4.16 will graphically depict the situation in this example. Let us consider for now that there is less delay in the path inside VN compared to the loop that goes through the CN and comes back. Also, let us think of a situation where the VN memory voltage is gradually rising and just crosses the VDD/2 line to the correct side. Therefore, the sign of VN changes and it forces the current-source to start discharging instead of charging. All of these is happening while the CN output will take some more time to update itself. Now, due to discharging by current source, the VN, just after rising above VDD/2, will again fall below VDD/2. Interestingly, the time VN has returned below VDD/2, the updated CN output will come. Since it is assumed that the side above VDD/2 to be the correct side, the updated CN output will say the current VN status (below VDD/2) to be correct, therefore, pushing it further down until the sign of VN again starts to pull it high. This example clearly shows that an unequal delay will cause undesired oscillations the time some VNs will cross the VDD/2 line. Due to these undesired oscillations, the VN may take an unusually long time going little above and below the VDD/2 and eventually resulting in high decoding time. A long decoding time and frequent change in CN output also means a significant amount of energy waste.

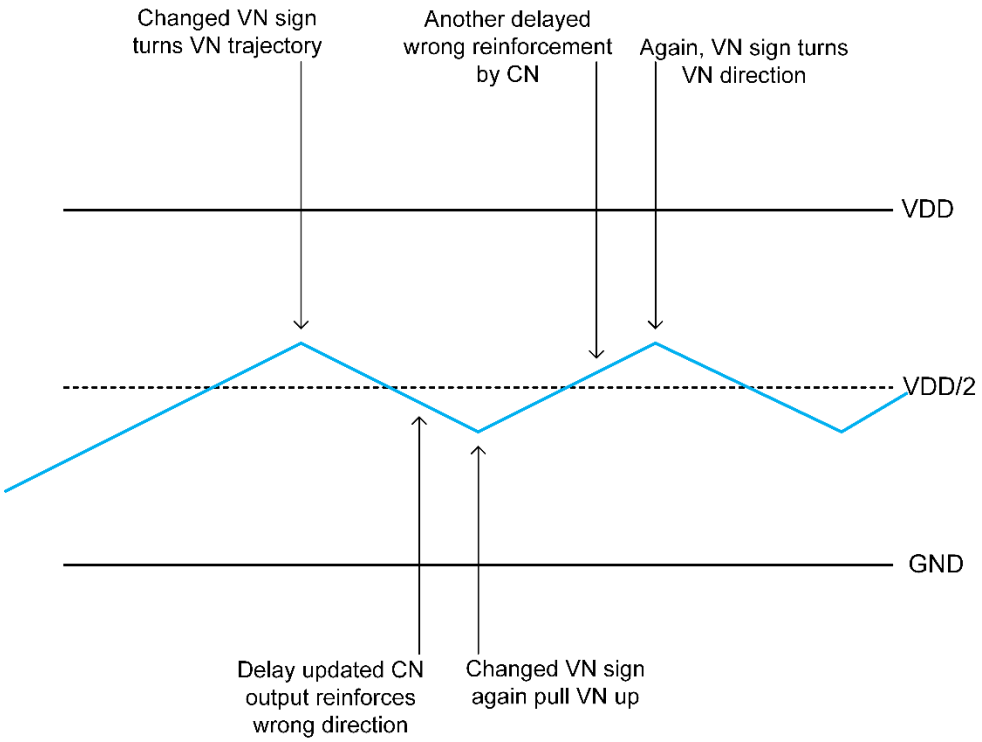


Figure 4.16: VN trajectory stuck dwindling above and below the VDD/2 line, due to lack of delay matching.

Now, addition of delay circuits and the way to tune the delay for proper matching will be discussed. In our design, the de-embed delay block consists of a current starved inverter followed by three simple inverters in a cascaded chain. The current-starved inverter is the key circuit for tuning the delay, which will be performed by tuning the gate voltages of rail PMOS and NMOS. However, in case it is tuned for a good amount of delay, the output of this inverter is much less steep. Since the de-embed delay block will only process a digital signal and a delayed output needs to be compared to the input, an additional inverter chain is needed to make the output steep. Figure 4.17 shows the delay block. The voltages PTUNEDELAY and NTUNEDELAY will supplied from off-chip to tune the delay of this block.

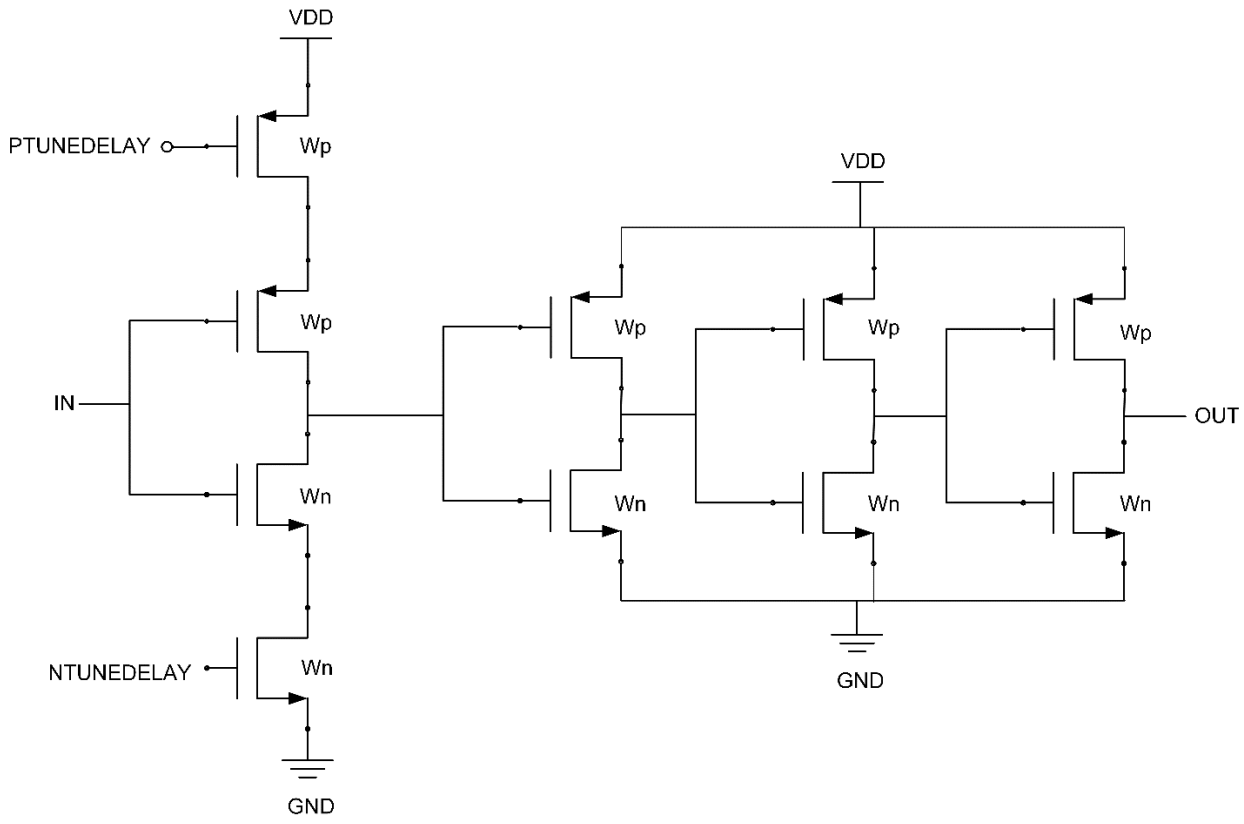


Figure 4.17: The de-embed delay Block. All MOS has minimum length, i.e., 60nm. Widths are – Wp=260nm, Wn = 195nm.

Let us see the simulation of this delay block to observe the delay it provides. The waveform output of the simulation is plotted in Figure 4.18. From the figure, it is obvious that most of the delay of this de-embed delay block comes from the delay through the current-starved inverter. The three cascaded inverter chain following it inserts very little delay. However, they make the gradual change in the output of the current-starved inverter very steep. The rising and falling edge delay through this de-embed delay block is almost 225ps. Signals propagating through the delay block can be both rising and falling and irrespective of their orientation, they have to incur the same amount of delay. Therefore, in addition to increasing or decreasing the delay, another careful delay tuning aspect is the matching of rising and falling edge delay. By some trial and error, the PTUNE and NTUNE voltages have been set so that they give us (1) the required amount of delay so as to match the delay through VN output to threshold inverters to

VN input, to the delay from VN output through CN to VN input, and (2) equal rising edge and falling edge delay through the block.

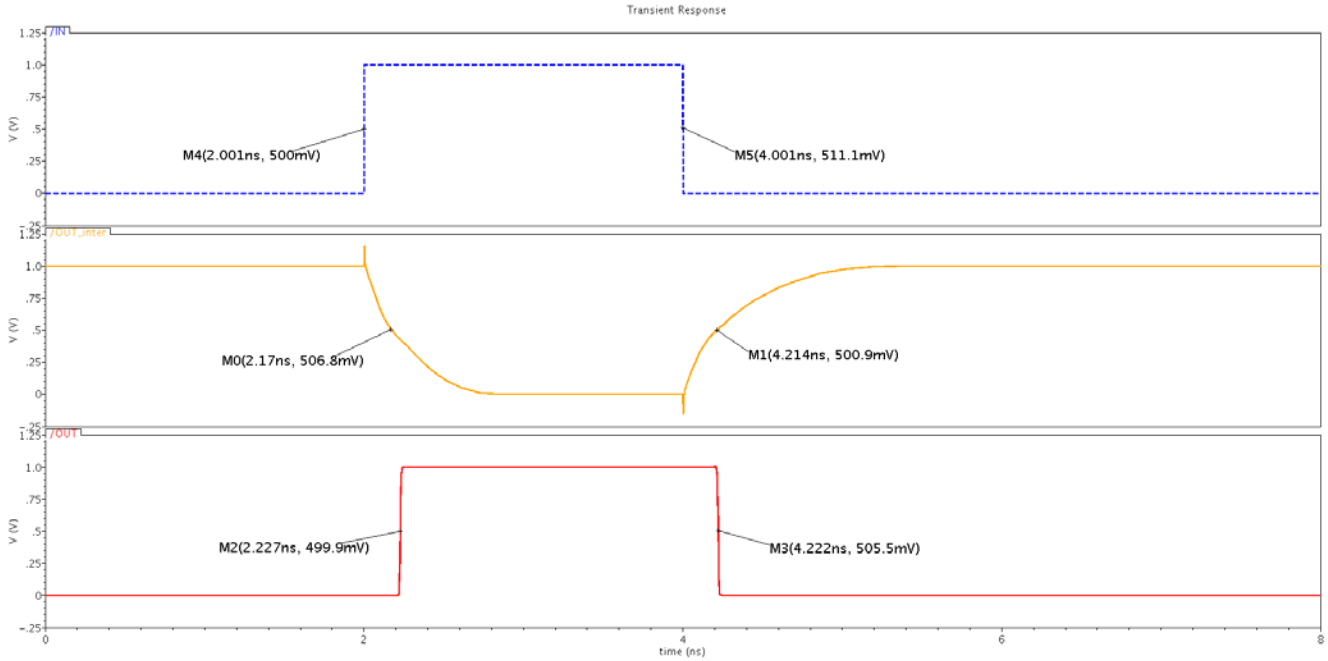


Figure 4.18: Simulation of de-embed Delay block. Blue – Input; Yellow – Output of current-starved inverter; Red – Output of de-embed Delay block.

4.4 Tuning and Calibration of Current-source Inverter

A major difference between the digital implementation of the decoder and its analog counterpart is the current-mode addition of de-embedded check node messages and the feeding of this current to the variable node capacitor. This requires a voltage-to-current converter, which can be implemented by an inverter. However, while implementing the current source with an inverter, there are a few things to be careful about. First, an inverter inverts the input logic, i.e., an input voltage of 1(=VDD) means discharging capacitor charge by a drain current at the output and vice versa. Since this inversion of logic to take place is undesired, another small inverter will be put preceding the current-source inverter. A combination of both of them will ensure that a de-embedded voltage of 1 will charge the VN capacitor up and the de-embedded voltage of 0 will discharge the capacitor down. Second, the current-source inverter has to be calibrated, which means that the current charging and discharging capabilities of two inverters with opposite input logic levels have to be equal. Finally, to be able to modulate the decoding speed of the decoder, the current-source inverters have to be tunable in terms of their current-supplying strength. In other words, a faster charging or discharging by the inverter will enable a reduction in decoding time. This phenomenon will be discussed in this section

Figure 4.19 shows a current-source inverter structure that is tunable to provide different amounts of charging and discharging current. The lengths of all the transistors are selected to be minimum, i.e., 60nm. The widths of both the PMOS and the NMOS are 400nm. It is entirely possible to select a different set of widths as long as the inverter is properly calibrated. However, our choice of this set of widths stems from the fact that they are not too large for the layout area and at the same time this inverter is capable of supplying good amount of drain current to speed up the decoder. The relationship of the current-supplying capability of the current source inverter to the speed of the decoder will be discussed later in this chapter.

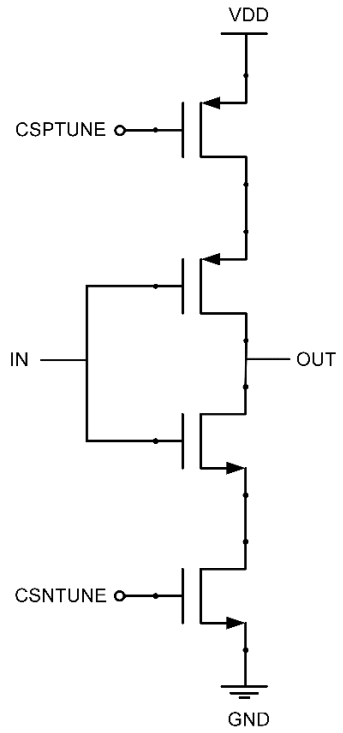


Figure 4.19: Current-Source Inverter (PMOS width=NMOS width=400nm, Length of all MOS = 60nm). Different sets of calibrated CSPTUNE and CSNTUNE voltage pairs were used to provide different amount of drain current through the structure.

Figure 4.20 shows one of the seventeen de-embedding paths inside the VN, which fixes the first issue discussed above. The inverting logic of the current-source inverter has been fixed with the XNOR gate preceding it. In fact, this XNOR gate followed by an inverter is actually an XOR gate followed by a current-source that does not invert the logic.

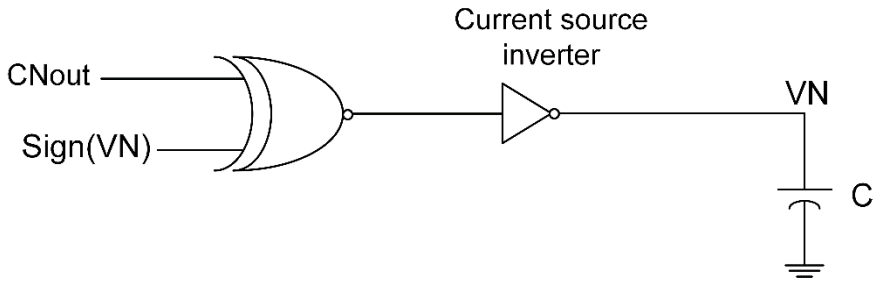


Fig 4.20: A slice of VN showing the current-source inverter following the XNOR gate.

Now comes the calibration of this current-source inverter. The purpose of the calibration is to ensure that all the current sources are of equal strength irrespective of the input to it (0 or 1). In other words, the current being supplied by a CS inverter with input '0' has to be equal to the current being drawn by a CS inverter with input '1'. There are two methods to calibrate our current-source inverter. In the first method, two CS inverters are taken separately and 0V is placed at one inverter's input and VDD is placed at the other inverter's input. The output of both inverters are separately connected to an ideal VDD/2 independent source or a very low-impedance VDD/2 source. In DC simulation, the CSPTUNE and CSNTUNE signals have to be tuned to achieve equal supplied or drawn current. Figure 4.21 depicts the setup of this method.

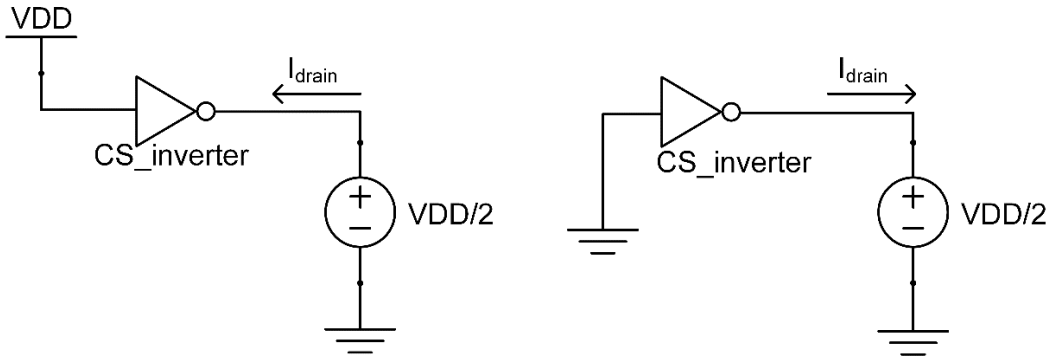


Figure 4.21: Method I for calibration of current-source inverter. Both inverters have same CSPTUNE and CSNTUNE pair (not shown in figure for simplicity).

In the other method, shown in Figure 4.22, the two CS inverters' outputs are connected while their input voltages are 0V and VDD respectively. Now, tuning the CSPTUNE and CSNTUNE voltages, a point is reached where the common output point has a voltage of VDD/2. Thus, the calibrated pair of CSPTUNE and CSNTUNE voltages has been obtained. Again, another pair can be obtained by trying a different set so that there will be more drain current supplied. In my opinion, method II is more interesting because it utilizes the fact that two CS inverters of the same dimensions and equal current supplied or drawn also have equal impedance when seen from their common output node. Therefore, this common output approaches VDD/2 as a result of voltage division when the inverters are properly tuned. The inputs of the current-

source inverters will always be at 0V or VDD irrespective of the inverters' positions in the calibration loop or the decoder schematic. Figure 4.22 only shows two current-source inverters used for calibration.

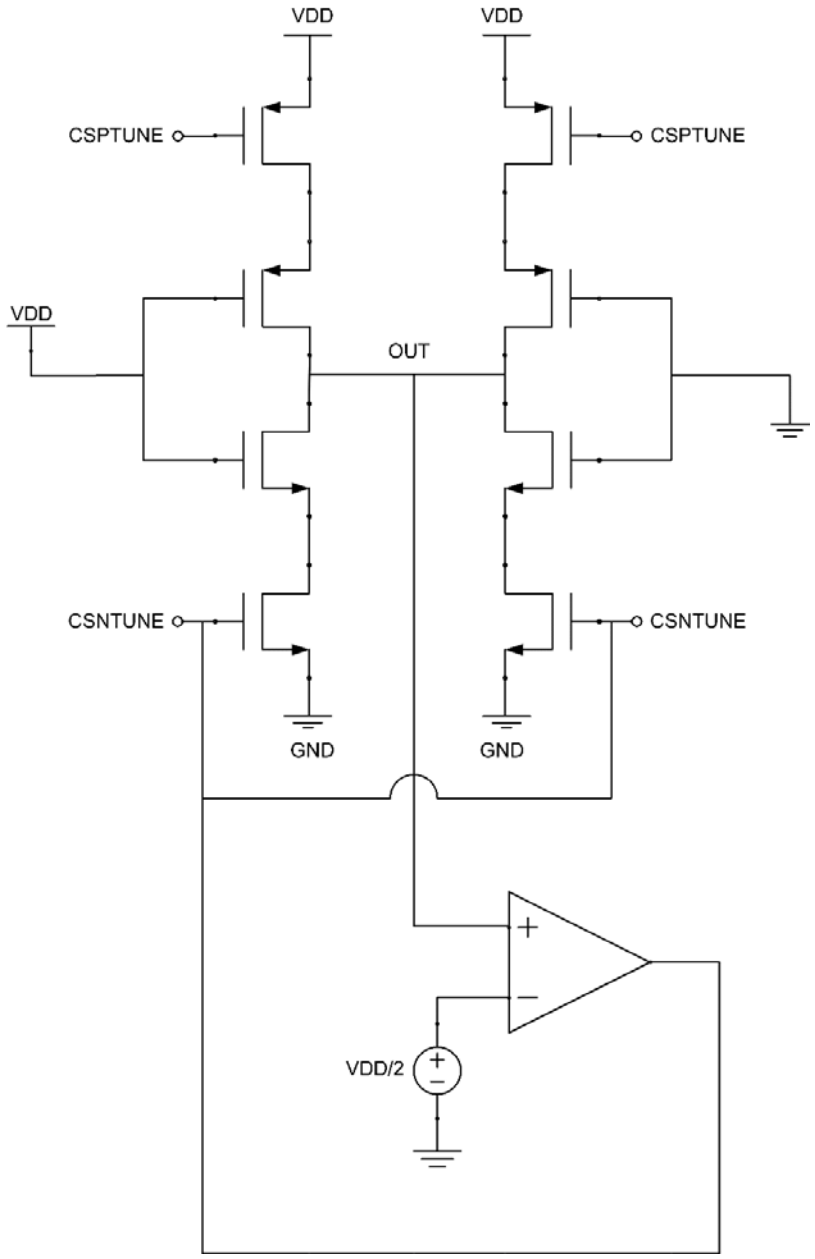


Figure 4.22: Method II for current-source inverter calibration. With proper CSPTUNE, comparison with the ouput and the VDD/2 by OpAmp will generate appropriate CSNTUNE.

Finally, let us relate the CSPTUNE and CSNTUNE voltage pairs to the drain current supplied or drawn in quantity. The more the drain current, the faster the decoding speed, up to a ceiling. In this chapter, different current ranges will be seen in quantity. Their effects on the decoding speed will be discussed later in the chapter dealing with the full decoder simulations.

Table 4.2

A few sets of tuning voltages for current-source inverter

CSPTUNE(mV)	CSNTUNE(mV)	Supplied or Drawn Current(μA)
0	571.4	72.02
397.2	462.5	36.00
521.5	392	18.02
600	339	8.95

While the current-source inverter's on-chip calibration is able to address the systematic offset, some measurement of the random mismatch between different current-source inverters is also important. The schematic to measure this particular random mismatch is depicted in Figure 4.23. The input of the current-source inverter is connected to either VDD or 0 V. In this figure, the input has been shorted to ground. The tuning voltages CSPTUNE and CSNTUNE are supplied to the rail PMOS and rail NMOS gate volatges respectively. The output of the inverter is connected to a voltage source. The voltage V_0 supplied by the voltage source is swept from 0 to VDD and the current drawn I_0 is measured at different supplied voltages. As the input of the inverter is connected to 0V, I_0 is maximum when the voltage V_0 is at 0V. Increasing V_0 will decrease the current until the inverter PMOS enters triode. From monte-carlo simulation, standard deviation of this current for the entire range of supplied voltage is obtained. The scaled standard deviation (s.d./I_0) at a point where V_0 is close to 0V, i.e., the PMOSes of the current-source inverter are in the saturation mode, gives the quantitative expression of random mismatch between current-source inverters. From the monte-carlo simulation, the amount of scaled standard deviation in supplied current has been obtained as 0.0509. Re-iterating the experiment

with supplying VDD at the input of the current-source inverter provides equal quantity of scaled standard deviation. In this case, only the NMOSes conduct while the output of the inverter stays close to VDD.

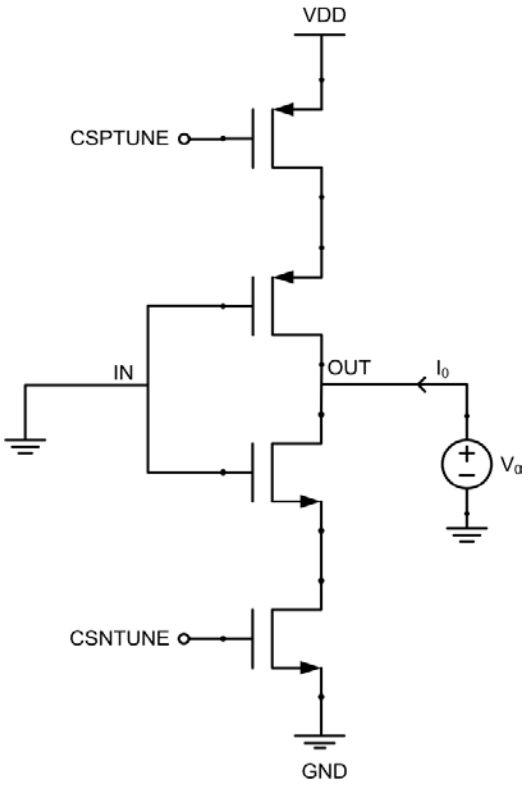


Figure 4.23: Schematic for determining random current-source inverter mismatch.

4.5 Memory Organization

Supplying a lot of bits from off-chip to generate the quantized voltage levels and to feed to the decoder cannot be done quickly. Hence, the memory array is useful in not only storing the bits of the codewords but also providing parallel 273 paths for loading into the 273 DACs. Without the incorporation of the on-chip storage, it would not be possible to quickly generate and feed the DACs from an off-chip serial data supply. Furthermore, decoding multiple codewords one after another helps in estimating the average decoding speed, which, with only one codeword, would be difficult to accurately measure. However, one codeword requires the

storage of 273*7 bits on chip. Therefore, the area occupied for storing several codewords is large. So, storing ten codewords seems to be a fairly good optimization between the quantity of codewords and the area consumed to store them. In that case, the memory array has to store 10*7*273 bits in total.

Storage of 10*7*273 bits will require the memory array to have 10*7*273 storing cells. So, several possible ways have been checked to implement this. While D flip-flops could be an option, a 6 transistor SRAM is much smaller in size. So, it has been decided to store the bits into an SRAM array.

Figure 4.24 shows an SRAM cell, which consists of two back-to-back inverters and two access NMOS transistors. The access transistors work as a switch that is turned on during writing in or reading out. In other times, the switch will stay off. The BIT and BITbar nodes are the wires where the incoming message to be written into the SRAM comes first. Then, the access switch is turned on and one bit gets written. Then, the switch is turned off. Now, when it comes to the reading phase, the access MOSs are turned on and the SRAM yields its stored bit into the BIT and BITbar wires.

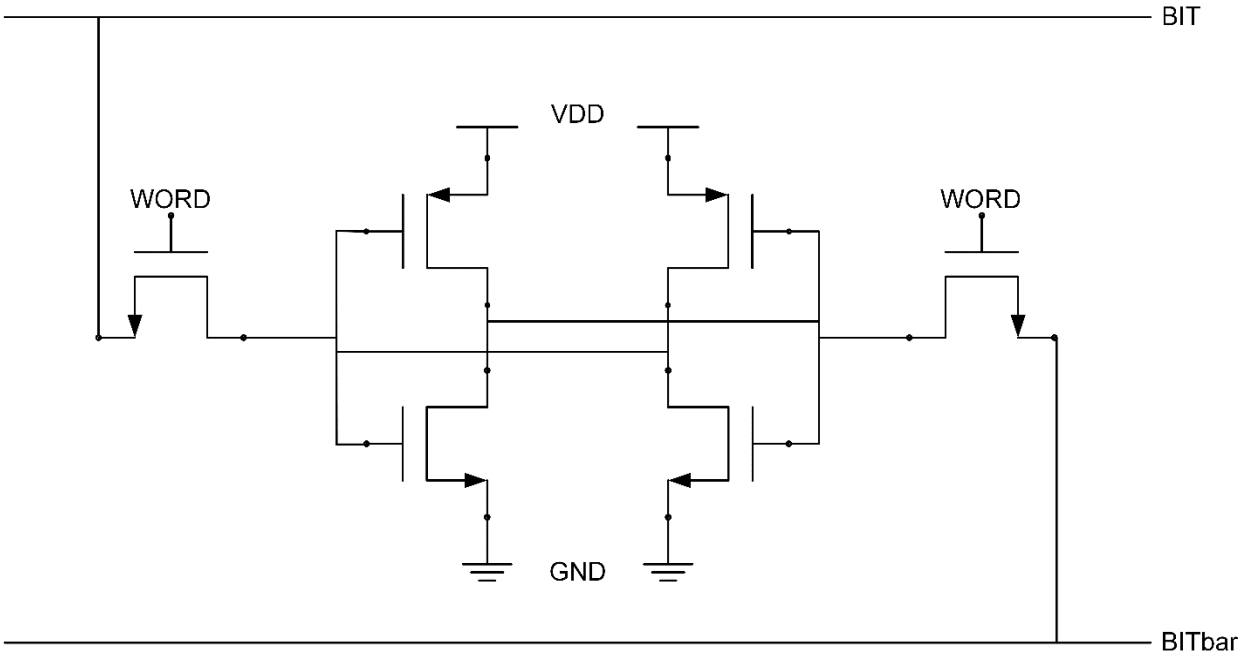


Figure 4.24: Schematic of an SRAM cell.

To generate an analog voltage level at the output of the DAC, the DAC has to be fed with 7 bits corresponding to the analog voltage at a time. This, in turn, means that there must be seven parallel BIT lines and seven parallel BITbar lines carrying 7 bits and their inverts simultaneously. 273 such qantized levels have to be generated through 273 DACs at a time.

Figure 4.25 illustrates the structure that can write in and store 7 bits simultaneously from its bit lines. It can also read out the stored 7 bits for the generation of analog levels in DAC. The full memory structure is shown in Figure 4.26. Here, the cluster of 7 SRAMs from Figure 4.25 is marked as SRAM7. There are 10 WORD lines to choose the 10 codewords independently for either writing in or reading out.

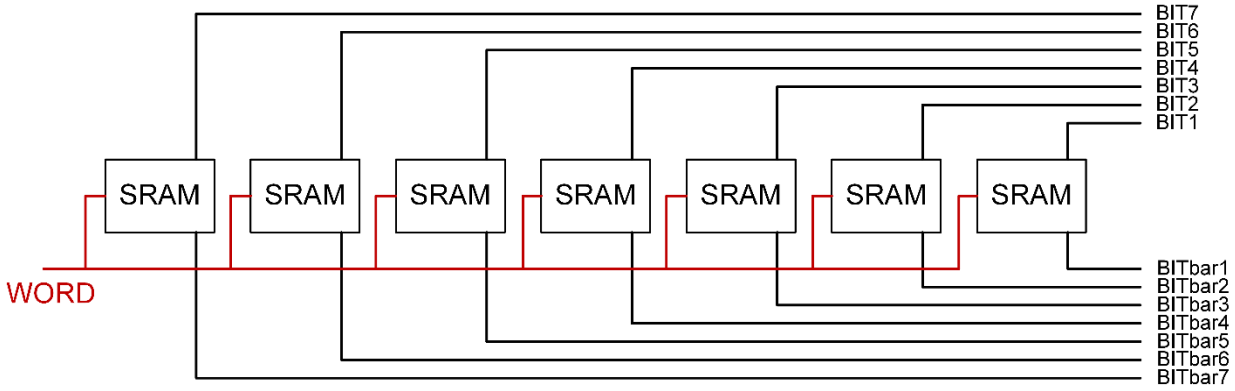


Figure 4.25: 7 SRAMs providing 7 bits in parallel to generate one analog level.

In Figure 4.26, ten WORD lines individually selects which portion of the memory array will be selected for writing in or reading out. For example, the bits corresponding to the first codeword is written in when WORD1 is raised high. Similarly, all other codewords are stored. When the time comes to read them out, WORD1 is again chosen high to read out the stored bits to the common BIT lines of the memory array. These BIT lines then carries this read-out bits to the DAC inputs. It is important to note that no more than one WORD line has to be raised high at a time and once all the bits of ten codewords are stored, the switches connecting the memory array to the input shift register are turned off.

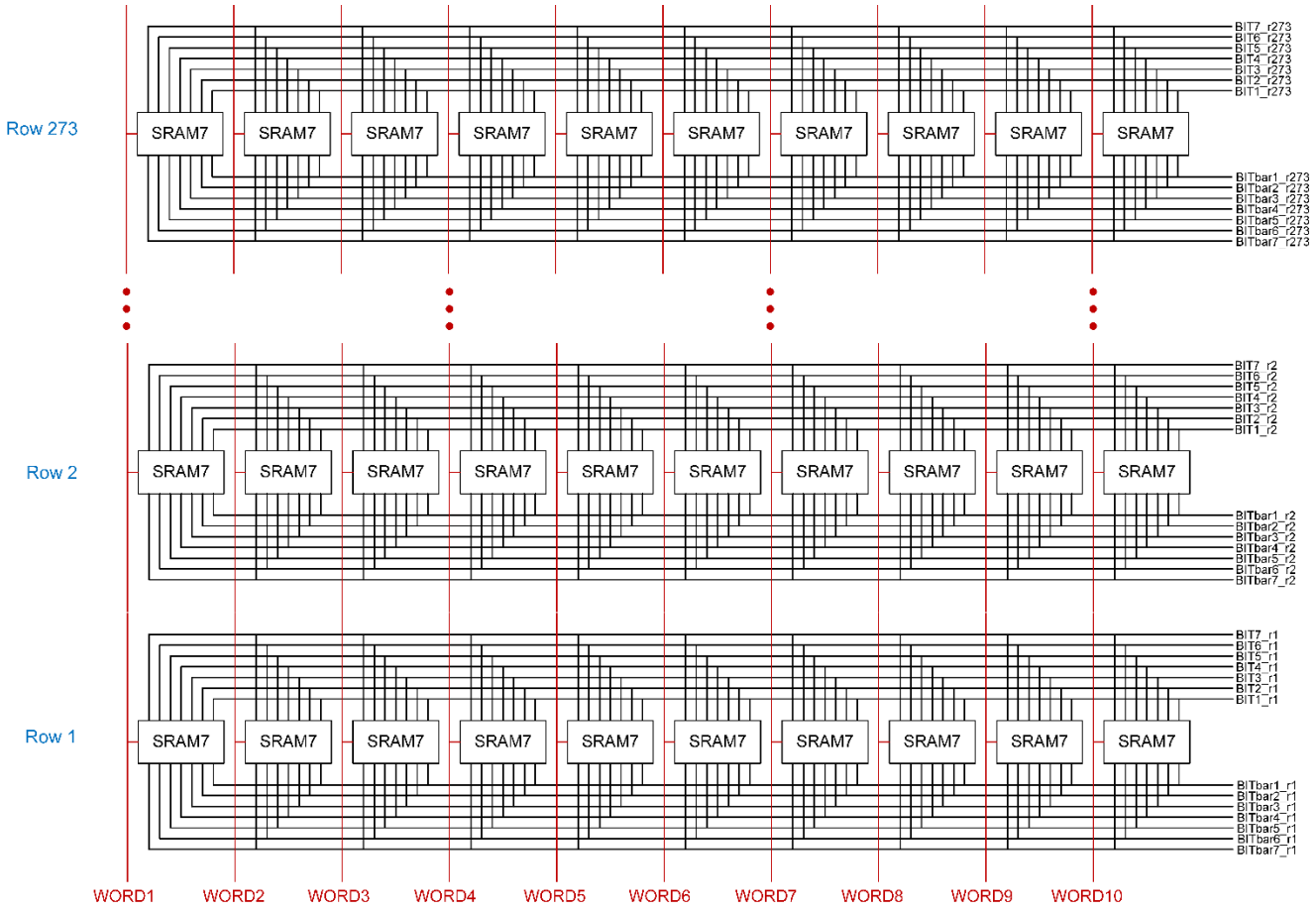


Figure 4.26: The full memory array.

4.6 Digital to Analog Converter (DAC)

The binary-weighted capacitor DAC is one of the key blocks in the design of our decoder IC. Its inputs are connected to the memory (SRAM array) and its outputs drive the VN capacitors in the core of the analog decoder. It is also connected to a few control signals from off-chip to regulate different modes and phases the DAC has to perform in. Its function is to generate analog voltage levels corresponding to the log-likelihood-ratio (LLR) out of the digital bits stored in the

memory and to feed the generated voltage levels to the variable node (VN) capacitance in the decoder.

Before the structure of the DAC is developed, let us discuss the general requirements of the DAC. First, the DAC has to be precise in generating analog voltage levels. If there is a lot of switching inside the DAC structure, which is true in our design, charge injection can affect the precision adversely. Therefore, the design of the switches has been carefully done so as to eliminate or greatly reduce charge injection. Second, the speed of the DAC is another requirement. DAC has to be fast enough so that the output can reach and settle to precise analog value before another set of digital input bits is loaded in. The average speed of the analog decoder in slow speed mode is approximately 1.5ns and that in moderate speed mode is approximately 0.9ns. The DAC operation time will be added to this time to give average decoding time when rapid feed in and decoding are being done. Now that the requirements are in mind, let us develop the structure of the DAC and then tune the transistor dimensions to nullify charge injection and to achieve good speed.

The DAC designed for our decoder is a 7-bit DAC. It is easier to analyze in an example a DAC that has a smaller number of bits. So, let us take a 4-bit DAC as an example. In a 4-bit DAC, the MSB decides if the constructed output voltage will be above or below VDD/2. Other bits can affect the output level according to the significance of the bit position. In a binary word like this, each bit can affect (i.e., raise or lower) the output with double strength compared to the next less significant bit. That is why, a charge-sharing based approach for DAC design can be implemented if capacitors are put in binary-weighted capacitance values. Each capacitor will have values according to the position of corresponding control bit in the input binary word. Charging up these capacitors and then sharing their charges will generate an analog level out of their binary-weighted charge distribution. A drawing of a 4-bit DAC is shown in Figure 4.27.

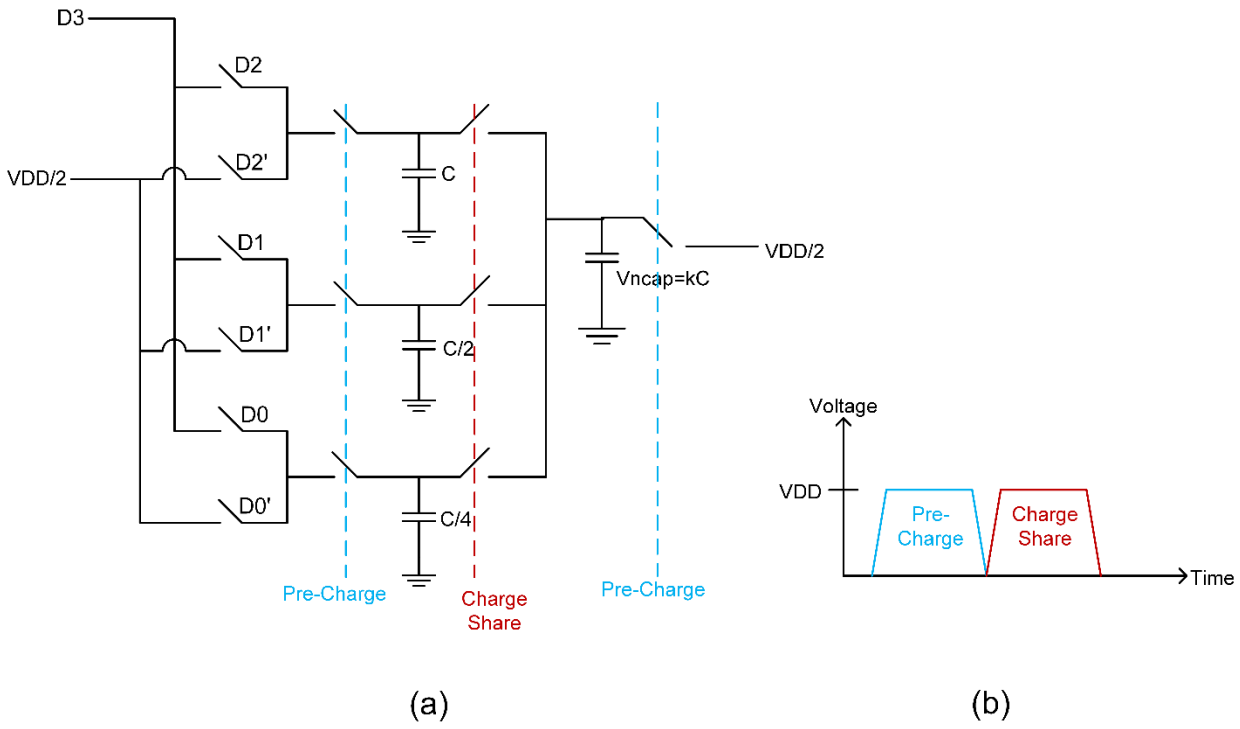


Figure 4.27: (a) A 4-bit DAC structure. Input $D_3D_2D_1D_0$, (b) Pre-charge and charge share pulses control DAC operation.

As can be seen in Figure 4.27, input binary word $D_3D_2D_1D_0$ is translated to corresponding quantized level by this DAC. The capacitors in the DAC are binary-weighted. The capacitor corresponding to the least-significant bit D_0 is the least of all capacitors, C/4. On the other hand, D_1 and D_2 correspond to capacitors C/2 and C respectively. The MSB D_3 is utilized in the pre-charging of the capacitors before charge-sharing takes place.

The variable node capacitance is kC. While the choice of 'k' is dependent on the parasitic capacitance of the VN wire, its value has significance in the construction of output voltage levels. The higher the value of 'k', the less spread the adjacent levels from each other. This happens because with higher value of k, charge redistribution takes place among larger amount of shared capacitance now. Although the larger capacitor kC is now being pre-charged to VDD/2 to hold more charge than it would with a smaller value of 'k', the value of other capacitances do not change. Therefore, re-distributed charge has less ability to raise output level than before. On

the other hand, k, if small, means that other capacitors in the DAC is large relative to the VN capacitor kC. This is a waste of space in the DAC design. Capacitors take larger space in physical design compared to a lot other circuit elements. So, smaller capacitors should be used, which are good enough to create sparse adjacent output voltage levels so that these levels are immune to little aberrations like charge injection. In our original 7-bit DAC deign, k=1 has been used, i.e., the largest capacitor in the designed DAC is equivalent to the VN capacitor. In practice, the VN capacitor represent capacitance obtained from shielded long horizontal metal wire in physical design of our IC.

The DAC works in two non-overlapping phases – the pre-charge phase and the charge-sharing phase. During the pre-charge phase, all the DAC capacitors and the VN capacitor are pre-charged. While the DAC capacitors can be pre-charged to either VDD/2 or the MSB (D3 here) depending on the sign of the MSB, the VN capacitor is always pre-charged to VDD/2. It is important to note that there is a big concern about the adjacent levels of the VDD/2. Any small deviation from the ideal value of the DAC capacitances or little charge injection from the series switches can most adversely affect the levels close to DAC. If, due to these effects, a VN voltage level close to and below VDD/2 is produced by the DAC to be slightly above VDD/2, the decoding will start with the wrong sign of the particular VN. Therefore, the DAC designed has to produce symmetric voltage levels around VDD/2. This is why, the DAC design requires pre-charge of the VN capacitance kC to VDD/2 during the pre-charge phase. After the capacitors have quickly risen to their pre-charge levels, pre-charge switches turn off and the charge sharing switch is turned on. Charge re-distribution takes place and proper voltage level at the VN capacitor is generated. An important fact about these two timing phases is that the time encompassing these two phases combined has to be equal to or less than the decoding time of the core analog decoder if the DAC latency needs to be removed from the decoding latency bottleneck.

Now, the pre-charge table will be shown describing the relationship between the DAC input bits and the pre-charge voltage levels. If the MSB (D3 in this case) is 1, the input bits will stay as they are. However, if the MSB is 0, the 1's complement of all the input bits will be taken except for the MSB. This conversion will have to be done prior to feeding into the IC from off-chip. Now that the appropriately converted bits are coming into the DAC, C, C/2 and C/4 will be

pre-charged according to the values of D2, D1, D0 and also the MSB. The pre-charge condition is as follows. If the MSB is 0, charge the capacitor to 0 if the corresponding input bit is 1. If the MSB is 1, charge the capacitor to VDD if the corresponding input bit is 1. Finally, whatever the MSB is, always charge the capacitor to VDD/2 if the corresponding input bit is 0. A flow diagram as in Figure 4.28 more conveniently puts the conditions.

As an example, let us take the input word of the DAC to be $D_3D_2D_1D_0 = 1101$. So, total charge accumulated in the capacitors during pre-charge, $\sum Q = C.VDD + \frac{C}{2}.\frac{VDD}{2} + \frac{C}{4}.VDD + kC\frac{VDD}{2} = \frac{2k+6}{4} C.VDD$

Total capacitance, $\sum C = C + \frac{C}{2} + \frac{C}{4} + kC = \frac{4k+7}{4} C$

Therefore, The DAC output after all the capacitors are connected for charge re-distribution will be

$= \sum Q/\sum C = \frac{2k+6}{4k+7} .VDD.$

Table 4.3 shows the generated levels at the output of the DAC for all possible inputs. In addition to that, the expressions for the case k=1 has been placed in the last column. Interestingly, rising each adjacent level in the DAC input is $\Delta Q = \frac{C}{4}.VDD$ more charge. Therefore, the difference between each successive level is $\Delta Q/C = \frac{1}{2(4k+7)}$ VDD. This is the resolution of the 4-bit DAC. From this expression of resolution, our previous discussion about the effect of the value of 'k' on the resolution is obvious. There is only one exception of the input word $D_3D_2D_1D_0 = 0111$, which will produce exactly VDD/2 at the DAC output, and so, this level has been intentionally omitted from the table and will not be fed to the DAC.

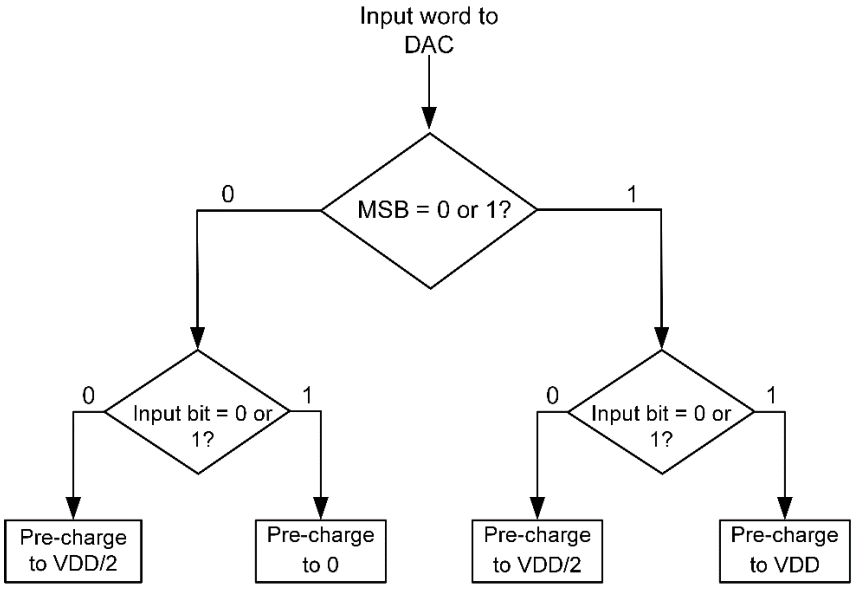


Figure 4.28: Flow Diagram showing the pre-charge voltages of the DAC.

Table 4.3

4-bit DAC inputs and pre-charge voltages

D4	D3	D2	D1	Input Word (1's complement if MSB=0)	Pre-charge Voltage at C	Pre-charge Voltage at C/2	Pre-charge Voltage at C/4	Quantized Voltage Level after Charge Share	Quantized Voltage Level if k=1
1	1	1	1	1111	VDD	VDD	VDD	$\frac{4k+14}{2(4k+7)}VDD$	$\frac{18}{22}VDD$
1	1	1	0	1110	VDD	VDD	VDD/2	$\frac{4k+13}{2(4k+7)}VDD$	$\frac{17}{22}VDD$
1	1	0	1	1101	VDD	VDD/2	VDD	$\frac{4k+12}{2(4k+7)}VDD$	$\frac{16}{22}VDD$

1	1	0	0	1100	VDD	VDD/2	VDD/2	$\frac{4k+11}{2(4k+7)}VDD$	$\frac{15}{22}VDD$
1	0	1	1	1011	VDD/2	VDD	VDD	$\frac{4k+10}{2(4k+7)}VDD$	$\frac{14}{22}VDD$
1	0	1	0	1010	VDD/2	VDD	VDD/2	$\frac{4k+9}{2(4k+7)}VDD$	$\frac{13}{22}VDD$
1	0	0	1	1001	VDD/2	VDD/2	VDD	$\frac{4k+8}{2(4k+7)}VDD$	$\frac{12}{22}VDD$
0	1	1	0	0001	VDD/2	VDD/2	0	$\frac{4k+6}{2(4k+7)}VDD$	$\frac{10}{22}VDD$
0	1	0	1	0010	VDD/2	0	VDD/2	$\frac{4k+5}{2(4k+7)}VDD$	$\frac{9}{22}VDD$
0	1	0	0	0011	VDD/2	0	0	$\frac{4k+4}{2(4k+7)}VDD$	$\frac{8}{22}VDD$
0	0	1	1	0100	0	VDD/2	VDD/2	$\frac{4k+3}{2(4k+7)}VDD$	$\frac{7}{22}VDD$
0	0	1	0	0101	0	VDD/2	0	$\frac{4k+2}{2(4k+7)}VDD$	$\frac{6}{22}VDD$
0	0	0	1	0110	0	0	VDD/2	$\frac{4k+1}{2(4k+7)}VDD$	$\frac{5}{22}VDD$
0	0	0	0	0111	0	0	0	$\frac{2k}{4k+7}VDD$	$\frac{4}{22}VDD$

Interestingly, if the MSB is 1, the pre-charge voltage for input bit 1 is VDD (logic 1). Also, if the MSB is 0, the pre-charge voltage for the input bit 1 is 0. In other words, an input bit of 1 always pre-charges the capacitor to the MSB. This behavior can be utilized in the DAC schematic. That is why, in Figure 4.27, the DAC switches are connected to D3 when the single-pole double-throw (SPDT) switch is controlled by 1. A more practical approach will be to insert a pair of inverters followed by the MSB to meet the high load requirement claimed by all the capacitors in the DAC whose input bits are all 1. The 4-bit DAC transfer characteristic for the case k=1 is depicted in Figure 4.29.

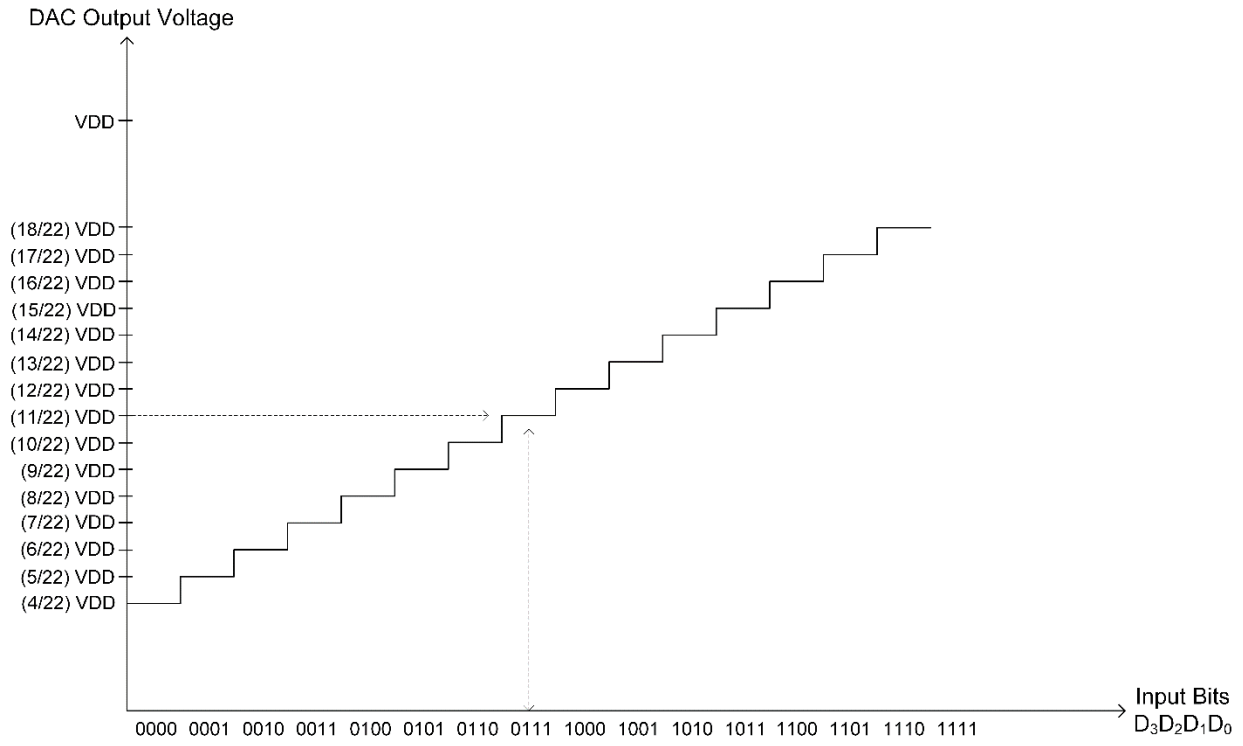


Figure 4.29: 4-bit DAC transfer curve.

With the above example and detailed description, all the requirements and performance metrics of the DAC in general have been established. Now, the complete schematic of the 7-bit DAC used in our design will be shown. The 7-bit binary-weighted capacitor DAC used in our design appears in Figure 4.30. The value of C is the capacitance of the VN. Its value of 256fF has been calculated from the parasitic extraction of the physical design of the decoder. However, the value of the VN capacitance from the parasitic extraction can change after chip fabrication. Therefore, arrangements have to be made to be able to measure the VN capacitance and also if the VN capacitance deviates from the expected value, the quantized levels at the DAC output have to be scaled accordingly. First, the measurement of the VN capacitance can be done with the analog MUX. The DAC has to be fed with a certain input word. The generated output voltage corresponding to that input word can be calculated. If the VN voltage inspected by the analog MUX is less than expected from the calculation, the actual VN capacitance is more than expected. On the other hand, if the generated VN voltage level is greater than expected from

calculation, the VN capacitance is actually less than expected from parasitic extraction in Cadence. Since the analog MUX does not fetch analog values close to the rail, this testing for VN capacitance should be done feeding the DAC with input words corresponding to VDD/2 or levels closer to VDD/2. Second, if the VN capacitance shows significant deviation from the expected value, there has to be some provision to scale the generated DAC output. This function can be performed using a TOGGLE signal. Suppose, the VN capacitance is greater than expected, therefore, the generated VN voltage is less than calculated. In this case, toggling the bottom plate of the largest DAC capacitor, C from 0 to 1 can provide the charge-shared capacitors an equivalent amount of C.VDD extra charge, therefore, pulling the VN voltage higher. On the other hand, if the VN voltage crosses the calculated value and becomes higher than it, toggling the bottom plate from 1 to 0 can pull out C.VDD amount of charge from the capacitor arrangement. This, effectively, makes the TOGGLE signal behave as the 7-th bit of the DAC structure. From Table 4.3, the VN voltage for the input word 1111 is $\frac{4k+14}{2(4k+7)}VDD$. Now, injecting C. VDD charge into a total shared capacitance of $\frac{4k+7}{4}$ C, raises the VN voltage by $\Delta V = \Delta Q/\sum C = \frac{8}{2(4k+7)}$ VDD, which is eight levels above the highest level achieved without toggling. However, the VN capacitance being greater than expected, the final VN voltage after toggling is smaller than calculated here.

A noteworthy point here is that if the VN capacitance is larger than expected, the toggling is done. In our design approach, the DAC output voltage level is automatically estimated to be greater than VDD/2 or less than that by looking into the MSB of the DAC input word. Then, if the DAC output is supposed to be greater than VDD/2, the bottom plate will be toggled from 0 to 1 to make the DAC output higher. Otherwise, the toggling is done from 1 to 0 to pull the DAC output, below VDD/2, further lower. The TOGGLE signal, if turned on, activates this action. TOGGLE=0 means no toggling will be performed. On the other hand, if the VN capacitance in the fabricated chip is less than expected, no toggling is needed in that case. The input words supplied to the DAC have to be scaled down to make smaller before storing them into the memory.

At the bottom left corner of the Figure 4.30(a), the ‘toggle’ signal is used to push the DAC output voltage to the appropriate direction. An output supposed to be less than VDD/2 is

pulled lower and the output supposed to be higher than VDD/2 is pushed higher. If TOGGLE is set to be zero, its effect is not felt.

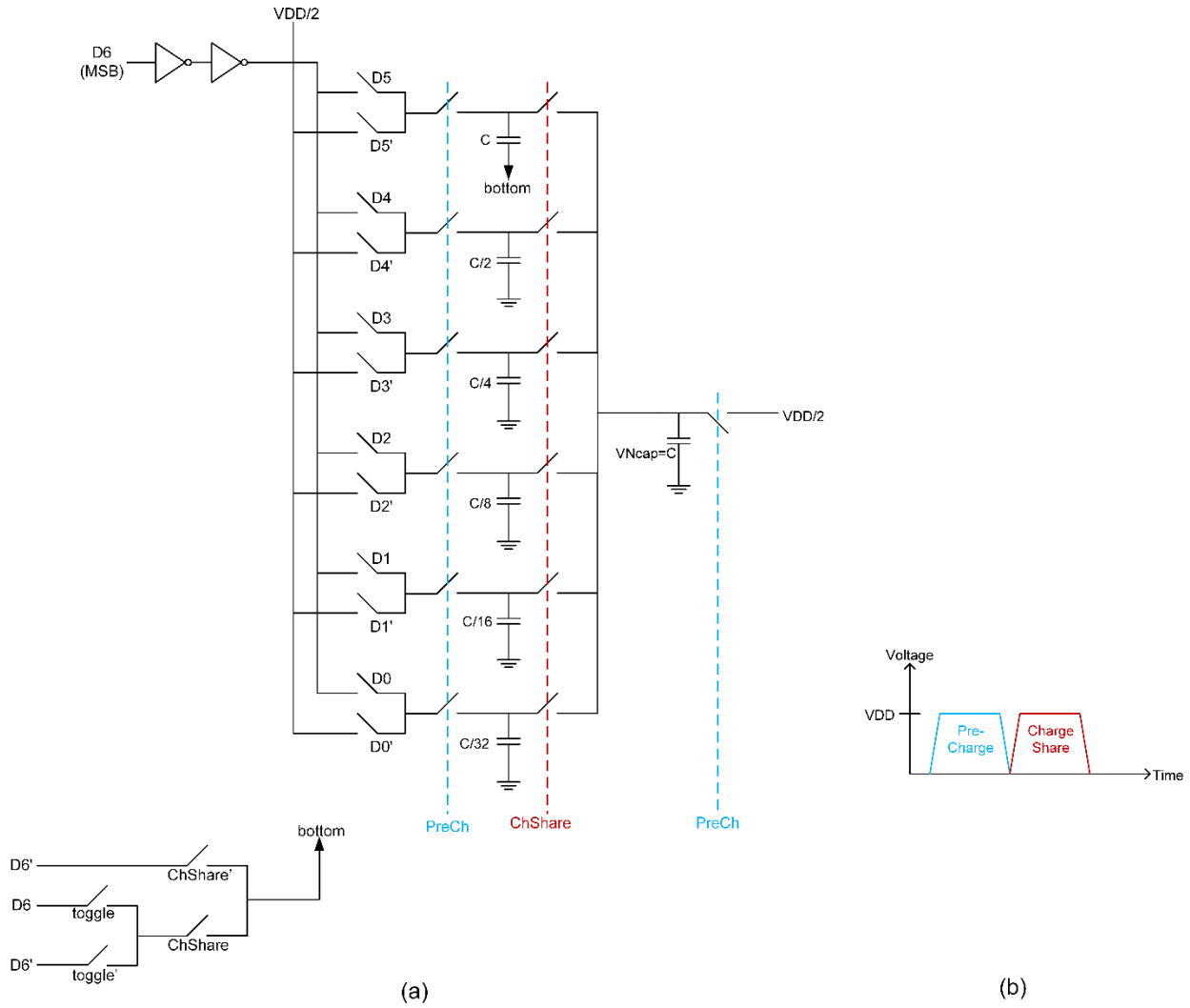


Figure 4.30: (a) 7-bit DAC used in our design. Control signals: PreCh, ChShare, toggle. C=256fF, (b) Pre-charge and charge share pulses control DAC operation.

Let us do an example based on equal total charge to understand the analysis of charge re-distribution. Here, the decoder is fed with an input word 1111100 while toggle=0. Since the MSB=1, the 1 bit capacitors will be pre-charged to VDD while the 0 bit capacitors will be pre-charged to VDD/2. Also, the VN capacitor C will be pre-charged to VDD/2.

Total charge accumulated in the capacitors before charge sharing, $\sum Q = C.VDD + \frac{C}{2}.VDD + \frac{C}{4}.VDD + \frac{C}{8}.VDD + \frac{C}{16}.\frac{VDD}{2} + \frac{C}{32}.\frac{VDD}{2} + C.\frac{VDD}{2} = \frac{155}{64} C.VDD$

Total capacitance, $\sum C = C + \frac{C}{2} + \frac{C}{4} + \frac{C}{8} + \frac{C}{16} + \frac{C}{32} + C = \frac{95}{32} C$

Therefore, The DAC output after all the capacitors are connected for charge re-distribution will be

$= \sum Q/\sum C = \frac{155}{190} .VDD$

There are 273 such DACs in our IC to generate 273 analog levels for 273 VNs. The inputs D0, D1, D2,……, D7 and D0', D1', D2',……., D7' come from the bit lines of the SRAM array preceding the DAC. All these input signals except for D7 are connected to the gate of the transmission gate switches for pre-charging capacitors. Therefore, these signals have low load requirement and hence does not need a pair of buffer inverters before they come to the input of the DAC. The SRAM is able to bear these loads at its bit lines. On the other hand, D7 signal, supplying pre-charge current to capacitors, requires buffering through a pair of inverters to help meet the high load demand.

During the two DAC phases, such as, pre-charge and charge share, the VN capacitor is held either at VDD/2 or at the analog voltage level produced by the DAC output. However, successful decoding means change in VN voltage trajectory over time so as to enable incorrect VN states to cross VDD/2 line to the proper side and to move to VDD or GROUND rail. This means that the decoding cycle has to accommodate a VN-free movement time duration in addition to the pre-charge and charge-share phases. So, a quick pre-charge and charge share phase can potentially reduce overall decoding time.

Now, the latency in the pre-charge phase and the charge sharing phase will be seen. Figure 4.31 graphically puts the time taken by the DAC output signal (in red) to rise to its charge-redistributed voltage after the charge-sharing control signal (in blue) goes high. There can be different amounts of timing required for different voltage levels risen or fallen from the VDD/2=500mV state. The farther away the desired output level is from the VDD/2 line, the more time it will take the charge sharing to reach the level final. Therefore, if the time required to rise to the maximum level of the DAC from the VDD/2 line is found out, that will be the

worst-case time. This very situation is depicted in Figure 4.31. So, the maximum time required for the charge sharing to take place is 11ps as shown in the graph.

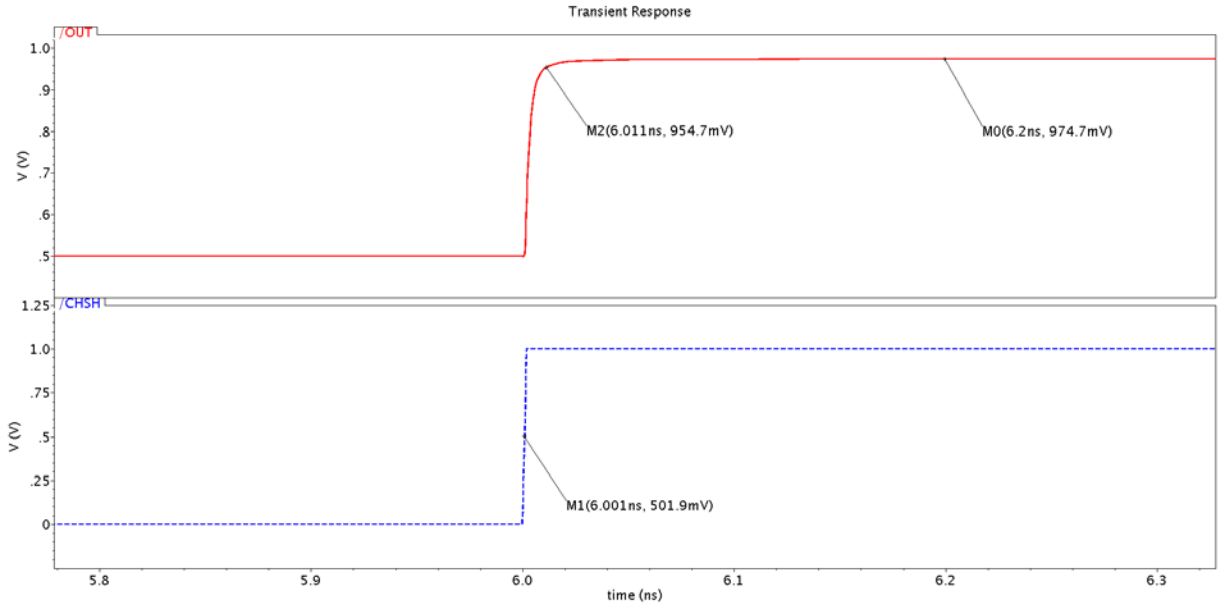


Figure 4.31: Charge Share control signal in blue. DAC output on VN capacitance in red. Time taken by charge sharing is 11ps to reach 98% of the final voltage.

Similarly, if the maximum voltage produced at the output of the DAC falls down to VDD/2 due to the pre-charge signal, the time taken for that will be a good estimate of worst-case pre-charge time. Figure 4.32 clearly illustrates this situation. While the pre-charge signal (in blue) turns on, it takes 20ps for the DAC output to move to the VDD/2 level. If the DAC output was previously in a level close to VDD/2, it takes less time to move to VDD/2. Therefore, considering the worst-case time is the most pragmatic way.

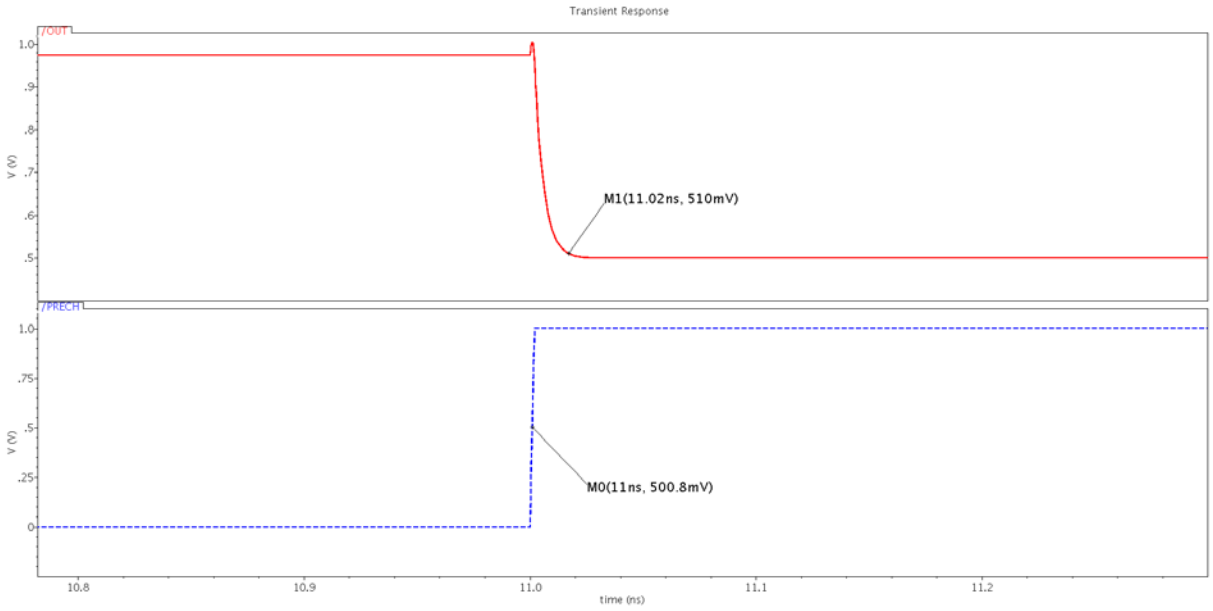


Figure 4.32: Pre-charge control signal in blue. DAC output on VN capacitance in red. Time taken for pre-charging is 20ps to reach within 2% of VDD/2.

The control signal ‘toggle’ in the above discussion was kept 0. It will be turned ‘1’ depending on the value of the VN capacitance. In the physical design, any actual capacitance are not being placed for VN, rather, the parasitic capacitance of long horizontal wire shielded above, below and by the sides is being used. Therefore, depending on the process variation, the VN capacitance can be smaller or larger than expected. In case the VN capacitance is smaller, input will be scaled and binary word will be reduced at the input. However, if the VN capacitance is larger, the DAC output levels will be closer together and the maximum and minimum output voltage levels will be far from the rails. To cope with the situation, toggle signal will be used to toggle the bottom plate of the largest DAC capacitor to inject more charge into the distribution to achieve the voltage levels, which would be lowered if VN parasitic capacitance is higher.

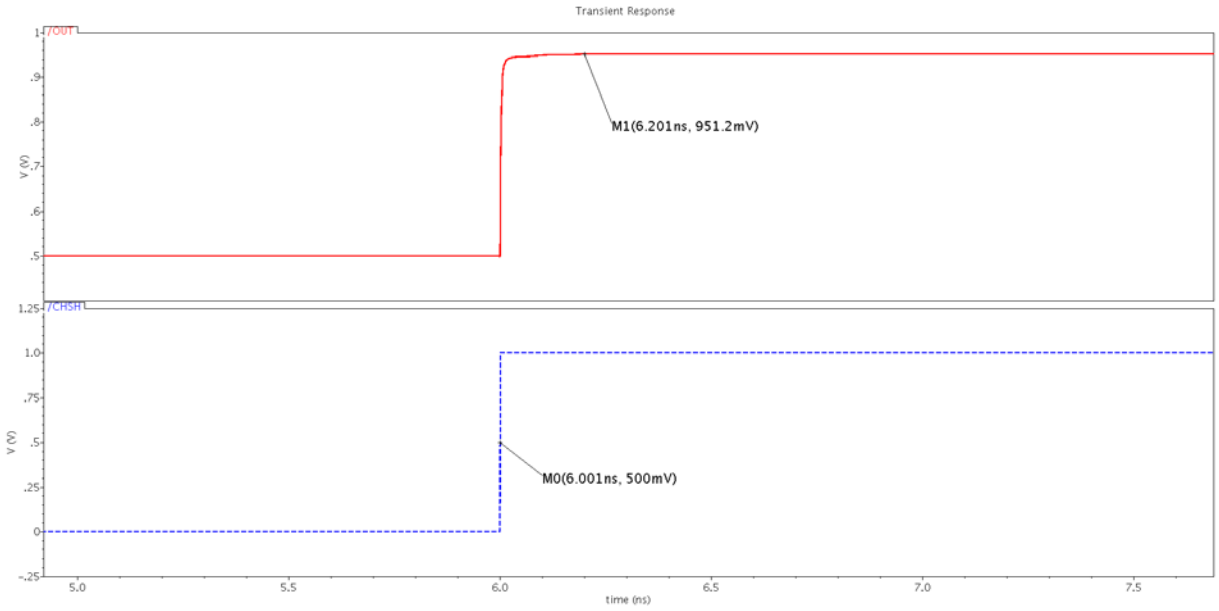


Figure 4.33: DAC output in red for input word 1111100 (toggle=0).

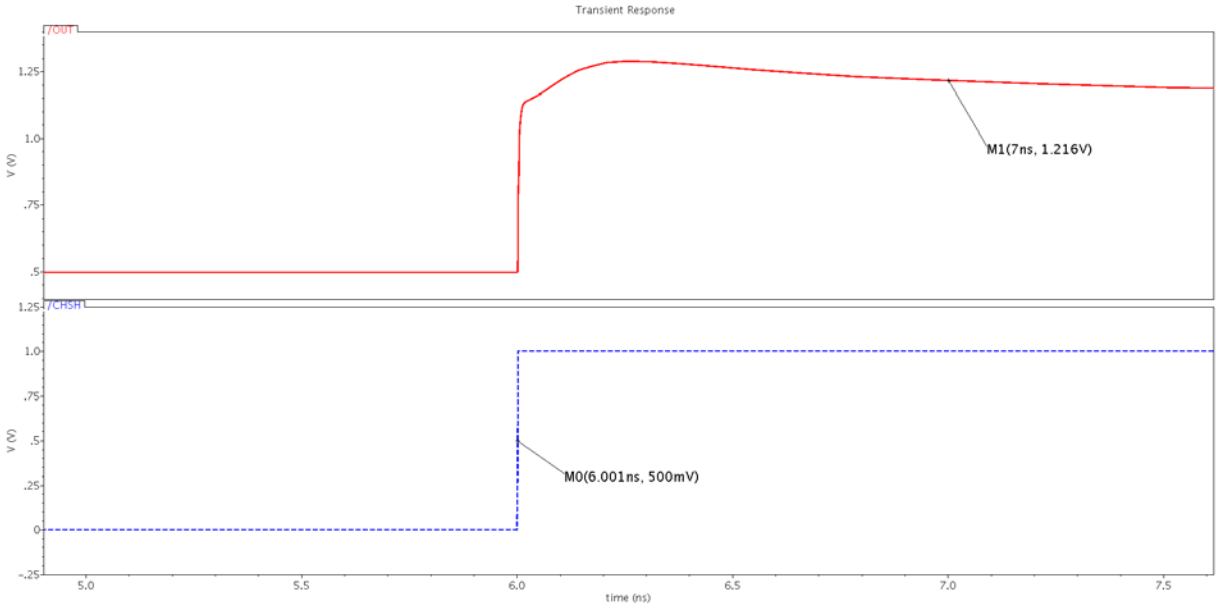


Figure 4.34: DAC output in red for input word 1111100 (toggle=1). Toggled from 0 to 1 to get higher level values. VN capacitance kept same in Figure 6 and 7.

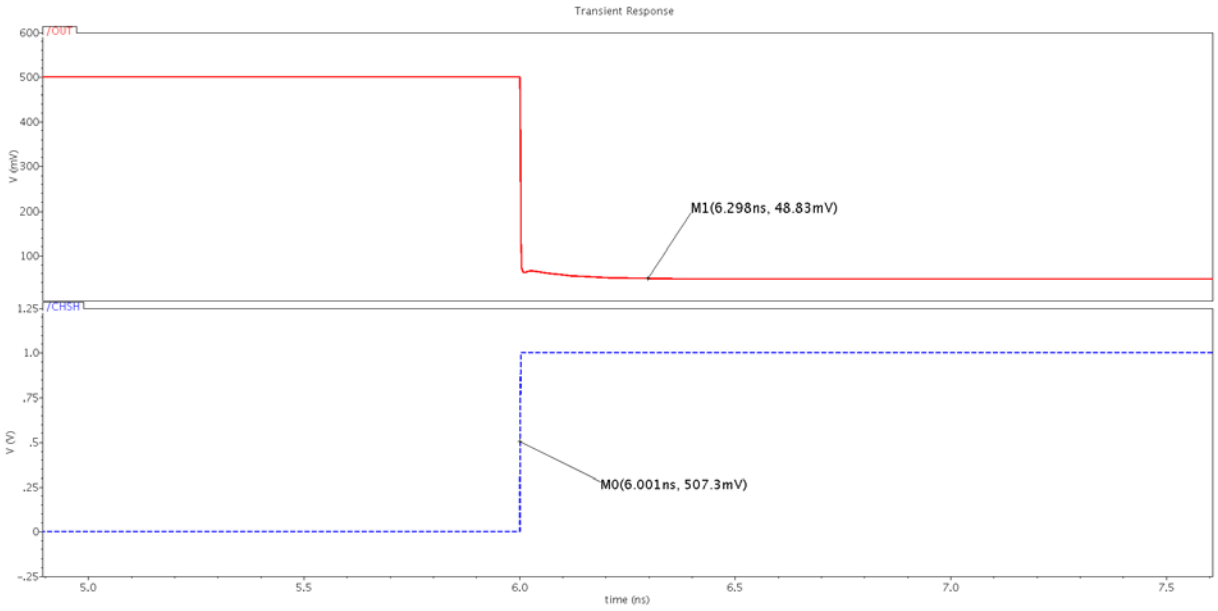


Figure 4.35: DAC output for input word 0111100 (toggle=0).

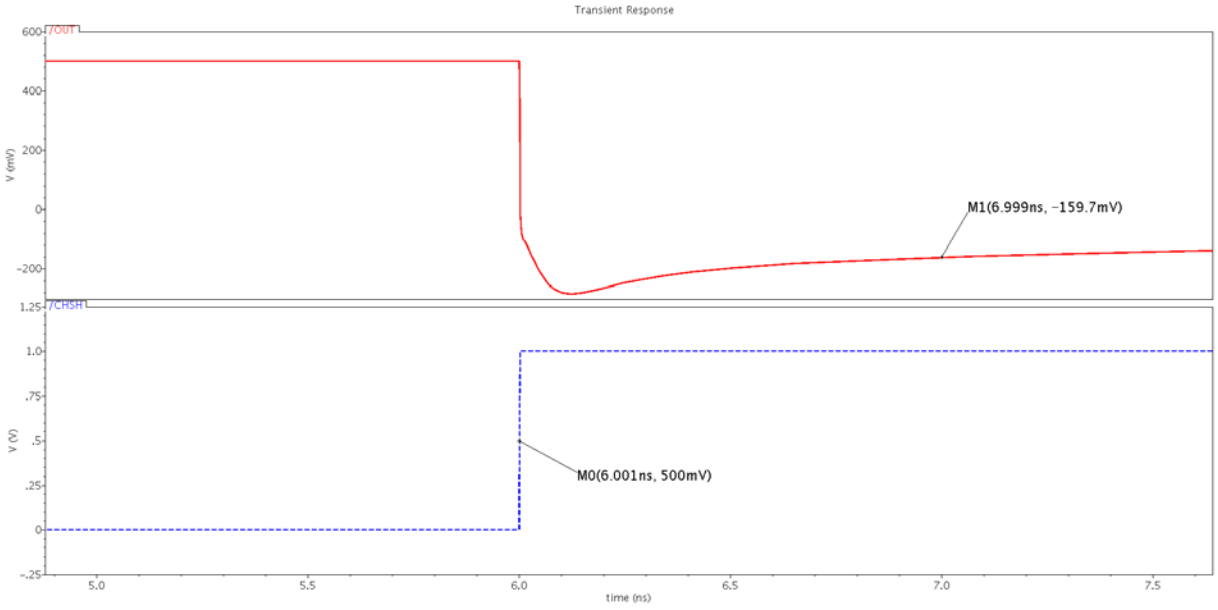


Figure 4.36: DAC output for input word 0111100 (toggle=1). Toggled from 1 to 0 to get lower level values. VN capacitance kept same in Figure 8 and 9.

4.7 Shift Registers

4.7.1 Input Shift Register

In our design, several shift registers of different structures have been included for different purposes. The shift register (SR) for buffering in the input digital bits from off-chip is one of them. Another SR is used to bring decoded digital bits out of the chip. In addition to these two SRs, there are a couple of small SRs for the selection of the WORD line in the SRAM array and for the selection of the SELECT pins of analog MUX. In this section, the bottom-up design approach of these SRs has been followed.

Let us begin with the input SR. Each codeword to be corrected by the decoder consists of 273 initial LLR values. The 7-bit data corresponding to each LLR equivalent voltage level have to be fetched from the memory and the decoder has to be fed 273 voltage levels at the same time. In the memory, 10 codewords are stored so that the codewords can be fed to the decoder in fast mode one after another. The input shift register fetches data bits serially in a slow mode and when it has loaded in all the 273*7 bits required for one codeword, it writes the bits into the SRAM array in parallel. This requires the input SR to be 273*7 DFF long – 7 in each row and 273 such rows.

Figure 4.37 show the schematic of the input shift register. There are 273*7 DFFs in total – 7 DFFs in each row. The output (Q pin) of each DFF is connected to a pair of cascaded small simple inverters. These inverter pairs insert delay into the path from the preceding DFF output to the following DFF input. It will be discussed shortly why these inverter pairs are essential while both the output and the opposite to the output can be obtained from the DFF. Each row in the above design also supplies signals to 14 horizontal wires – 7 of them are bit lines of the next SRAM array stage and the other 7 are the bitbar lines. There are TG switches to connect the input shift register outputs to the SRAM bit lines. It is important since the DFFs cannot be clocked in the shift register while the SRAMs will be writing in the data. The last DFF in each row is connected to the first DFF input of the next row. The 273*7 positive edges have to be clocked to let the data reach the very last DFF of the SR.

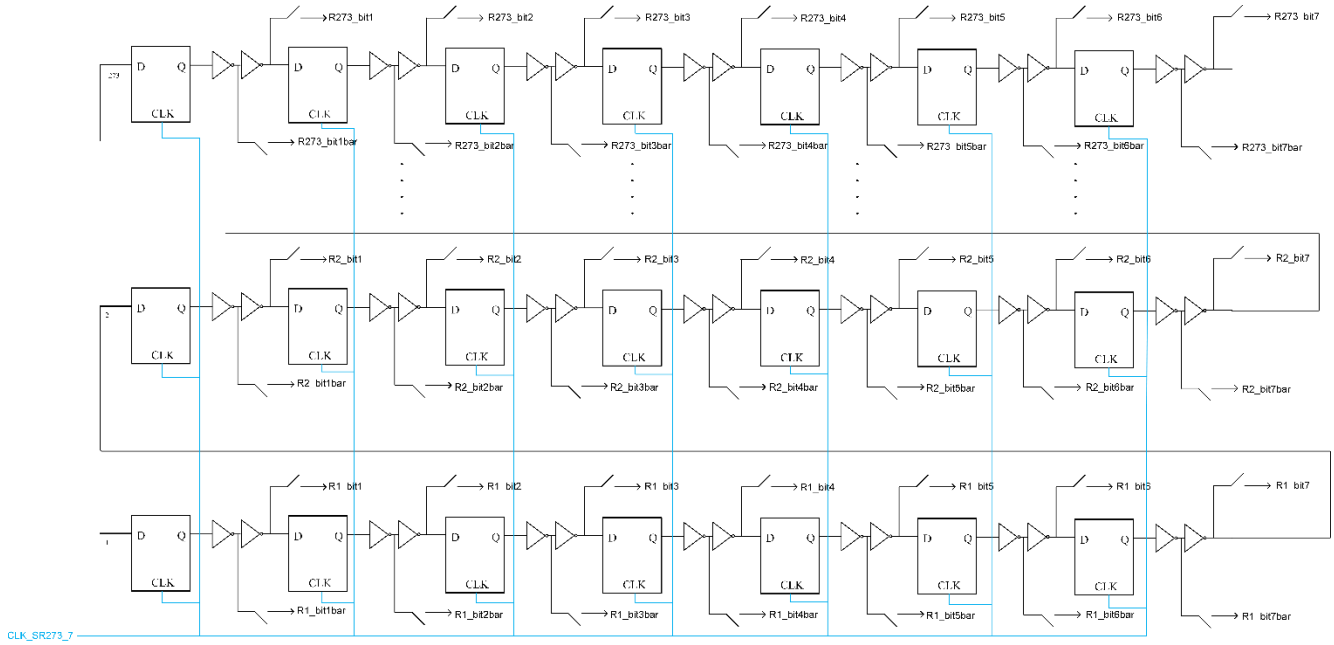


Figure 4.37: Schematic of Input Shift Register

While supplying input to the chip, the synchronization of the clock and data is important. This demands special attention since the data is being sampled by the DFF at the clock edge. Figure 4.38 shows the clock signal sampling the incoming data bits. The setup time T_S and hold time T_H are also marked on the figure. If, due to the large distribution of the clock signal, the clock reaches after a delayed time to some DFFs located far from the data input pin, this may lead to diminishing hold time. In other words, the long clock-to-Q delay may reduce T_H to such an extent that there is a risk of sampling the next data bit at a particular sampling clock edge. Therefore, delaying the data propagation as more DFFs are added to this large serial shift register can check undesired reduction in the hold time. A way to implement this is to insert a pair of inverters following the DFF output, which were done in our SR design.

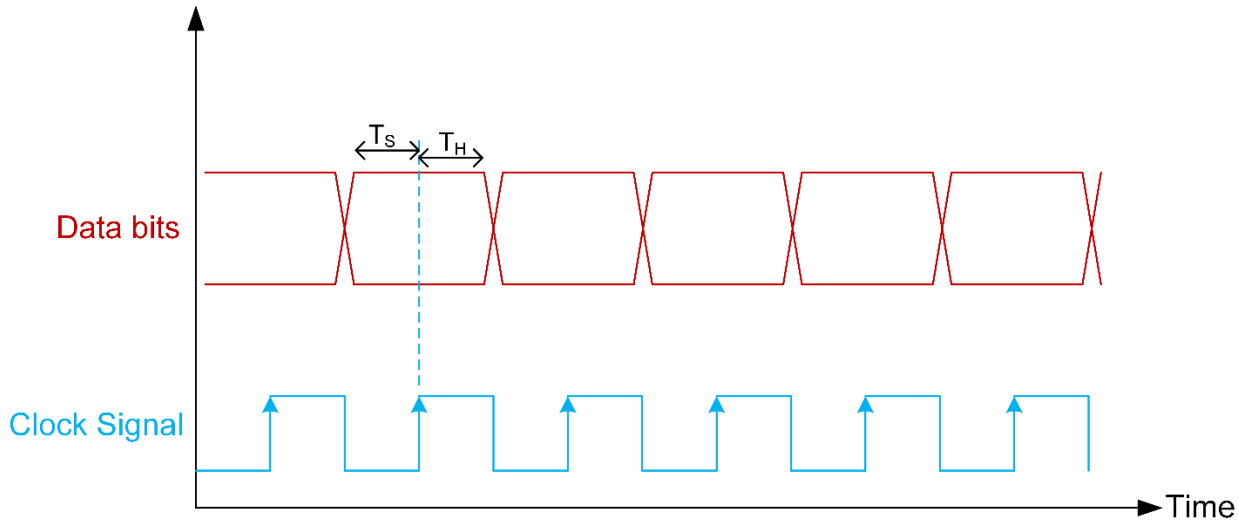


Figure 4.38: Clock signal sampling incoming data bits.

4.7.2 Output Shift Register

The output shift register performs the task of bringing the decoded digital bits out of the chip. Once the decoded bits are out, they can be compared with the correct bits and the bit-error-rate can be calculated. Now, the output shift register consists of a parallel-input parallel-output (PIPO) register and a parallel-input serial-output (PISO) register. The job of the PIPO register is to fetch the decoded digital data from the core decoder and pass it to the PISO register which then brings the data out serially. If the PISO register would have been directly connected to the output of the decoder, clocking the DFFs to bring the data out serially would be a problem. Hence, there is the need for a separation provided by the PIPO register.

In each row of the PIPO register, there is a DFF, a tuned threshold inverter preceding the DFF and a switch to connect or disconnect the DFF output to the input of the PISO register. The PISO register, on the other hand, has a DFF and a TG switch following it in each row. Obviously, the output of one row of the PISO register is connected to the input of the subsequent row of the PISO register. Similar to the input SR, the output SR has inverter pairs following the DFFs (included inside the DFF unit in schematic). Figure 4.39 depicts the output shift register.

Once the decoding is complete, just one clock cycle can bring in parallel the thresholded VN states (which are the decoded digital bits too) to the output of the PIPO register. Afterwards, a separate clock for PISO register serially brings all the 273 bits out of the chip. The time the PISO clock is running the PIPO output switch will be turned off to disconnect PIPO from PISO.

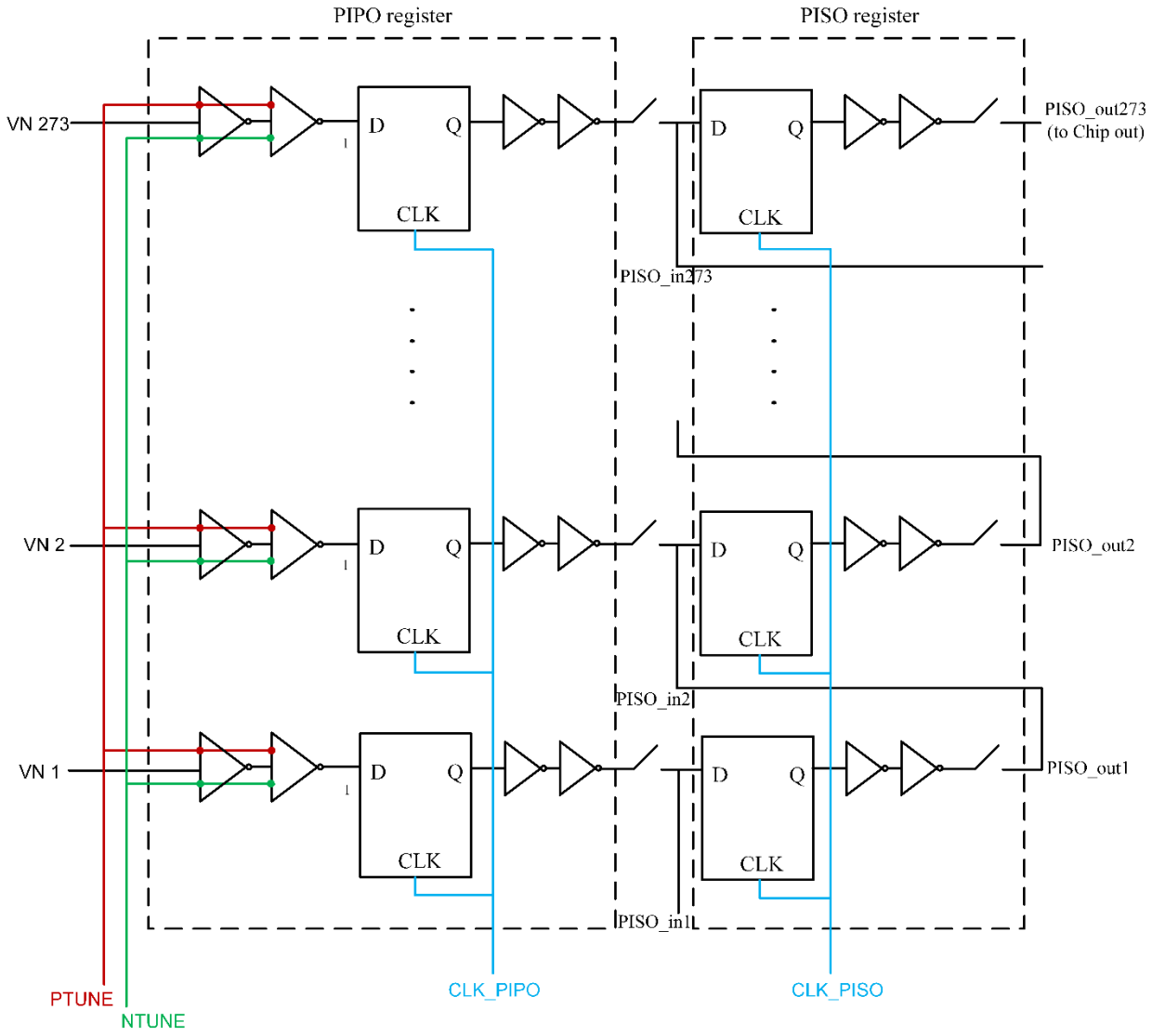


Figure 4.39: The output Shift Register. The PIPO and the PISO registers are shown in dashed line boxes. The leftmost side blocks are tuned threshold inverter pairs. Separate CLK signal for PIPO and PISO.

4.7.3 Shift Register for Memory

A 10 DFF long shift register has been used to select the WORD line of the SRAM array. The array stores 10 codewords. One out of the 10 WORD lines can be selected to choose which one-tenth of the memory array will be used to write the data in. This SR has its own clock and own input fed from off-chip. Usually the first codeword will be written to the part of the memory array to be made active by making the SR output 10000 00000. When it is needed to write the next codeword, the SR is just clocked once and the next address is obtained out of the SR, i.e., 01000 00000. Figure 4.40 presents the memory shift register. The switching connection is interesting in the sense that it gives us the ability to turn the SR to WORD line completely off and clock in any desired 10 bit word. However, when the SR is not connected to the SRAM array, all the 10 WORD lines are to stay off so that the memory block selection is not in a floating state.

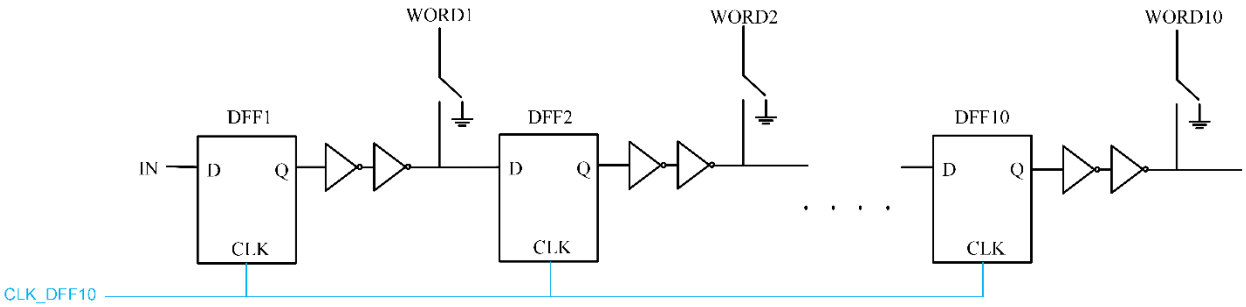


Figure 4.40: Shift register for selecting WORD line of the SRAM array.

4.7.4 Shift Register for Analog MUX

This is another shift register for generating the control signal for the Analog MUX. This MUX will be used for the debugging purpose. It will help bringing out the analog value of any VN from inside the chip. Each analog MUX can buffer 16 VN levels. That means, $\log_2 16 = 4$ selector pins are needed. Accommodating 4 bondpads for this is a sheer waste of resource.

Therefore, similar to what was done for the memory SR, a small 4 DFF long SR is made for selecting the selector pins of the MUX. The input and the clock of this shift register have also got separate wirebonding pads.

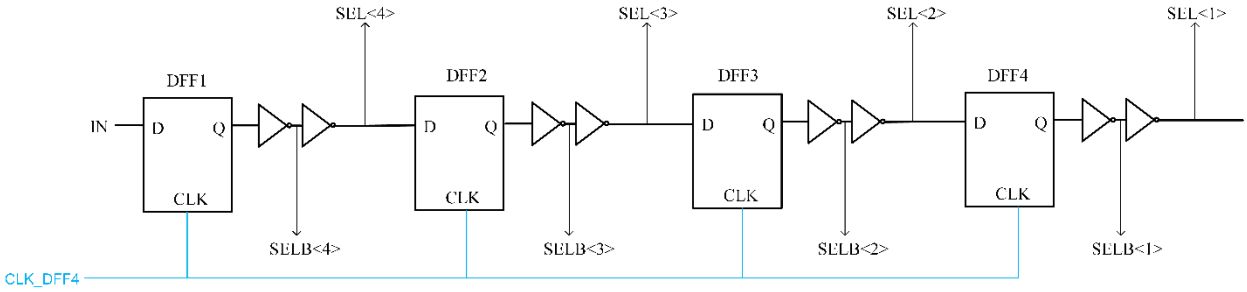


Figure 4.41: Shift Register for Analog MUX SELECT Pins.

4.8 Analog MUX

The chip is fed with data bits and the decoded bits are brought out of it. The analog VNs are not generally accessible from off-chip. An analog MUX is useful in the sense that the access to the VNs will enable us to debug the analog voltages of the VNs. The schematic of the analog MUX consists of a switching tree to choose a particular VN out of many the MUX is connected to and a high gain OpAmp following the switches. To access 273 VNs, different sizes of MUXes can be chosen. To keep the number of selector pins low and also to avoid occupying large area with a lot of MUXes, 16:1 MUXes are chosen. Any MUX will be able to obtain the analog voltage level of any of the 16 VNs. 18 such MUXes are needed in total. Figure 4.42 shows the MUX schematic.

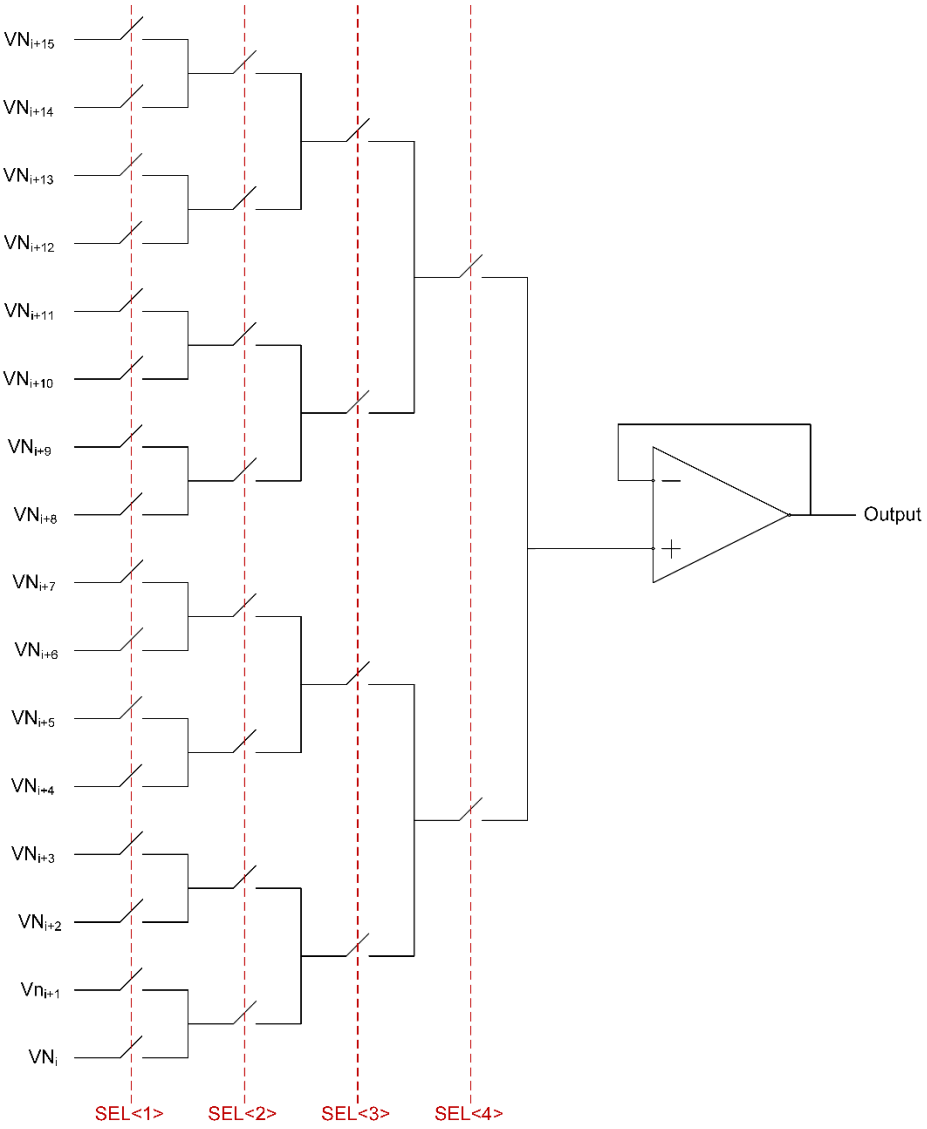


Figure 4.42: 16-input Analog MUX. SEL<4> is the MSB.

The reason there is an OpAmp buffer in addition to the switch tree is primarily to be able to safeguard the sensitive VNs from the off-chip connection parasitics and also, to be able to withstand the off-chip parasitics, such as, bondwire capacitance and inductances, while rendering the output value.

The major limitation of the analog MUX is that it has not been designed as a rail-to-rail analog MUX for simplicity. The MUX can precisely deliver input values from 300mV to 850mV. The VN voltage levels above or below this range shows an output at the margin of this range. However, this satisfies our investigation about which side of the VDD/2 the particular VN is. Also, the analog input levels located very close to the VDD/2 line can be precisely obtained.

The schematic of the folded-cascode OpAmp has already been discussed in detail in the Threshold Inverter Tuning and Calibration section of this thesis. Here, the precision in rendering analog levels will be discussed.

The open-loop gain of the OpAmp is 48.47 dB, i.e., 265 V/V. Therefore, a voltage level of 500mV will be obtained with a maximum deviation of $500 - \frac{265}{265+1} \times 500 = 1.88$ mV, which is much less than the resolution of the quantized analog LLR voltages. In practice, the levels close to the VDD/2 are much more accurate than the one just discussed and those far away from VDD/2 will see little deviations.

A higher gain OpAmp can be used by adding a second stage to its structure or by ramping up its dimensions. Both of these approaches will cost us more area. Therefore, our choice has provided us with reasonably high gain at an expense of moderate area occupation.

Figure 4.43 shows an example where the selected input of the analog MUX is 500mV and the output the MUX provides is 499.8mV. The output of the analog MUX, which is also the OpAmp output, is connected with a 200fF capacitor as part of the OpAmp design process. The addition of new bondpad and wiring capacitances will not affect the precision although these parasitics might affect the speed.

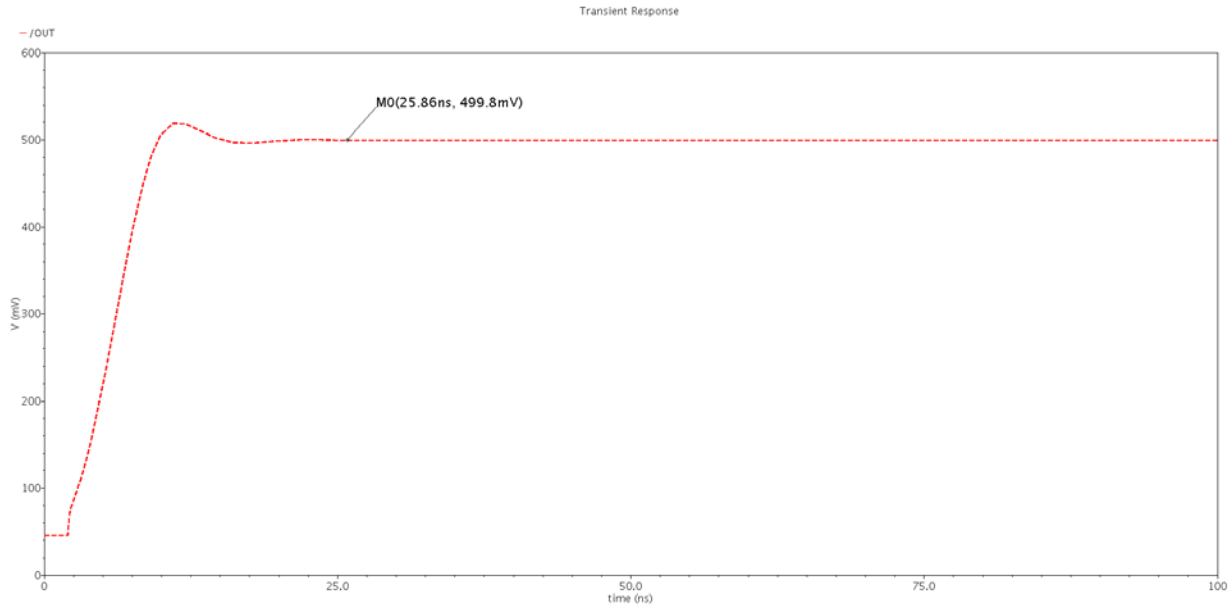


Figure 4.43: Analog MUX providing 499.8mV at the output. The input voltage selected is 500mV in this example.

The latency of the MUX is a few tens of nanoseconds, as shown in Figure 4.43. Interestingly, its speed is not a matter of concern since its use is only for manual debugging and it is not a part of high speed decoding process. This is why, a large capacitance was connected to the OpAmp output at the first place for better stability trading off higher speed.

4.9 Termination Check Circuitry

The end of a decoding cycle is signaled by the parity-check satisfaction of all the check nodes (CNs). Since all the satisfied CNs cannot be seen from off-chip, a signal has to be brought out that indicates the termination of decoding. This signal has been named as the termination check signal.

Figure 4.44 illustrates the circuits that generate the termination check signal. This circuit is essentially a large NOR gate. There are 273 NMOS transistors. The gates of all of them are

connected to the CN outputs. The drains of all NMOSes are tied together and connected to a large resistor. This common drain's sign is generated by a pair of tuned threshold inverters and then a pair of high-load buffering digital inverters bring out off-chip as the output. This output signals the end of a decoding cycle.

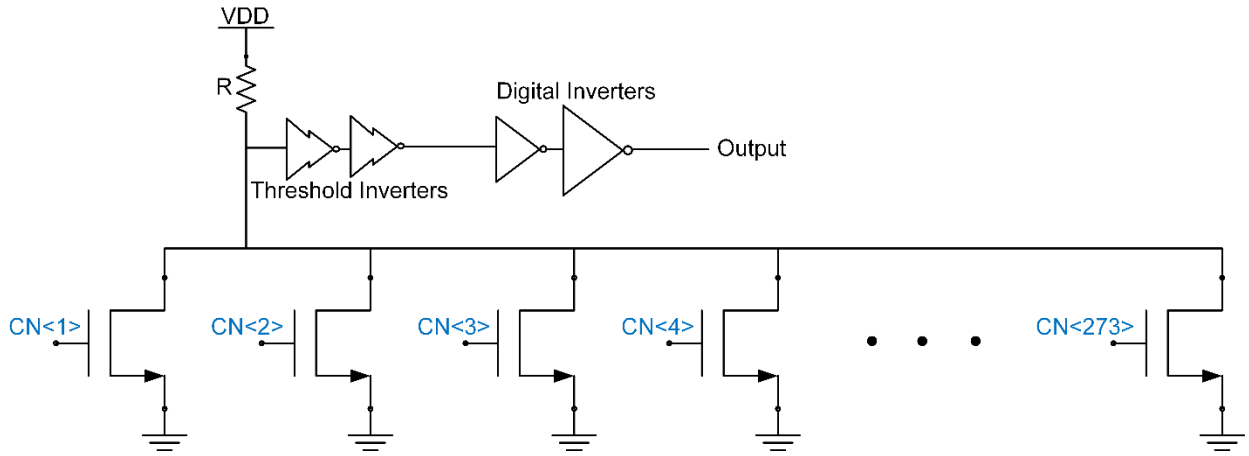


Figure 4.44: Schematic of Termination check circuit. Output is connected to a Bondpad.

The choice of resistance R has both upper and lower limits. At the start of the decoding process, some CNs are not satisfied. Therefore, the termination check signal is lowered by the unsatisfied CNs with value '1'. As the decoding process continues, some CNs flip their binary values once or multiple times. Eventually when all the CNs become '0' meaning, all of them are satisfied, the decoding is done. At this point, all the NMOS are off and so the termination check signal becomes high. The upper limit of R is set by the fact that the leakage current of the 273 off NMOS can pull the termination signal down if R is too large. On the contrary, if R is too small, the voltage drop across it during unsatisfied state will be too low to signal low at the output. Therefore, the choice of R has to be empirically verified in accordance with the NMOS dimensions. In this design, R=5.139 kΩ. NMOS width=200nm and length=60nm.

At the drain node of the NMOSes, the voltage will not be fully binary, hence the need for thresholding the signal using a pair of tuned threshold inverters connected to it. These threshold

inverters are good at providing digital values but not buffering large bondpad capacitances off-chip. Therefore, a pair of digital inverters are connected – the first one of medium size and the later one of large size. Simulation has been done by connecting the output of the termination check circuit to 1 pF capacitance and 2 nH inductance. The plot of the termination check signal is shown in Figure 4.45.

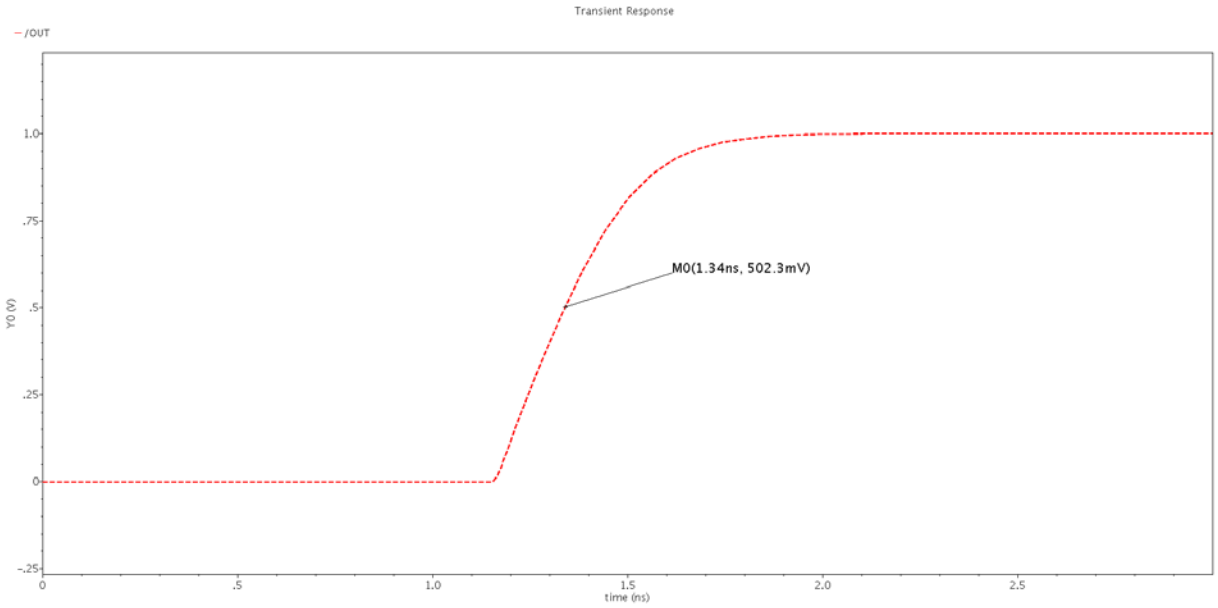


Figure 4.45: Plot showing the termination check signal going high at 1.34ns while all CNs are satisfied at 1.0ns.

4.10 Decoder Simulation

The core analog decoder simulation consists of several phases in time, such as, the pre-charge to VDD/2, the initialization of the voltages corresponding to the LLR, and finally the decoding. In the schematic design of the DAC, the variable node capacitance has been included for ease of calculation and understanding. However, there's no separate VN capacitance placed in the physical design. It is the capacitance of the shielded wire that acts as the VN memory capacitance and broadcasts the analog value to all connected CNs. It has been already discussed in the DAC design section that before the charge-sharing takes place between all the binary-weighted capacitors inside the DAC and the VN capacitance, there is a pre-charge phase where all these caps are pre-charged to VDD/2. As part of that rule, the VN capacitances are pre-charged to VDD/2.

Now, as all the 273 VN memories hold the VDD/2 voltage, the analog levels are ready to be fed in. Due to the very high amount of time taken for the core decoder simulation, i.e., around 35 hours for starting to simulate only one codeword in transient mode, the decoder simulation have been done separately from the initialization circuitry consisting of the input shift register, SRAM array, and the DACs. From the analog design library in Cadence, the pulsating DC source Vpwl has been used to write in several bits corresponding to several codewords and to feed them as the decoding starts. It is important to note that the decoder simulation takes into account the parasitic capacitance of the long vertical CN output wire, which has been found to be approximately 150fF using parasitic extraction in Cadence.

4.10.1 Decoder Throughput

Extensive decoding has been done on the decoder to be confident about the success of the decoding procedure. Furthermore, this decoder can be made to run at different speed modes. The speed modulation is done by tuning the charging/discharging currents by the calibrated current-source inverters. In the Table 4.4, the gate voltages at the rail PMOS and rail NMOS of the

current source inverter have been put as CS_PTUNE and CS_NTUNE respectively, and the corresponding charging current it provides in different speed modes.

Table 4.4

Different decoding speed modes and corresponding current-source inverter properties

Decoding Speed	Decoding time in one Example Decoding (ns)	CS_PTUNE (mV)	CS_NTUNE (mV)	Supplied Charging Current (μA)
Fast	0.91	0	571.4	72.02
Moderate	1.04	521.5	392	18.02
Slow	1.42	600	339	8.95

Table 4.4 shows, how tuning the charging current supplied from the current-source inverter can change the decoding speed of the analog decoder. Figures 4.42, 4.43, 4.44 illustrate the decoding speed further in slow, moderate and fast mode respectively.

From the Table 4.4, Let us take the average decoding speed in moderate speed mode as 1.04 ns. This corresponds to a theoretical maximum throughput of 0.961 x 10^9 codewords per second x 273 bits per codeword = 262 Gb/s. This represents an unachievable upperbound if termination checking and loading of the next codeword were instantaneous. However, the decoding cycle should also include the time needed for the pre-charge of VN to VDD/2 and the time needed for decoder initialization with these quantized voltage levels. In this simulation, 0.5 ns was allocated for the pre-charge phase and another 0.5 ns was allocated for the initialization. Therefore, if a decoding takes only 1.04 ns to complete, the overall cycle time will be 1.04 + 0.5 +0.5 = 2.04 ns, resulting in a throughput of 134 Gb/s.

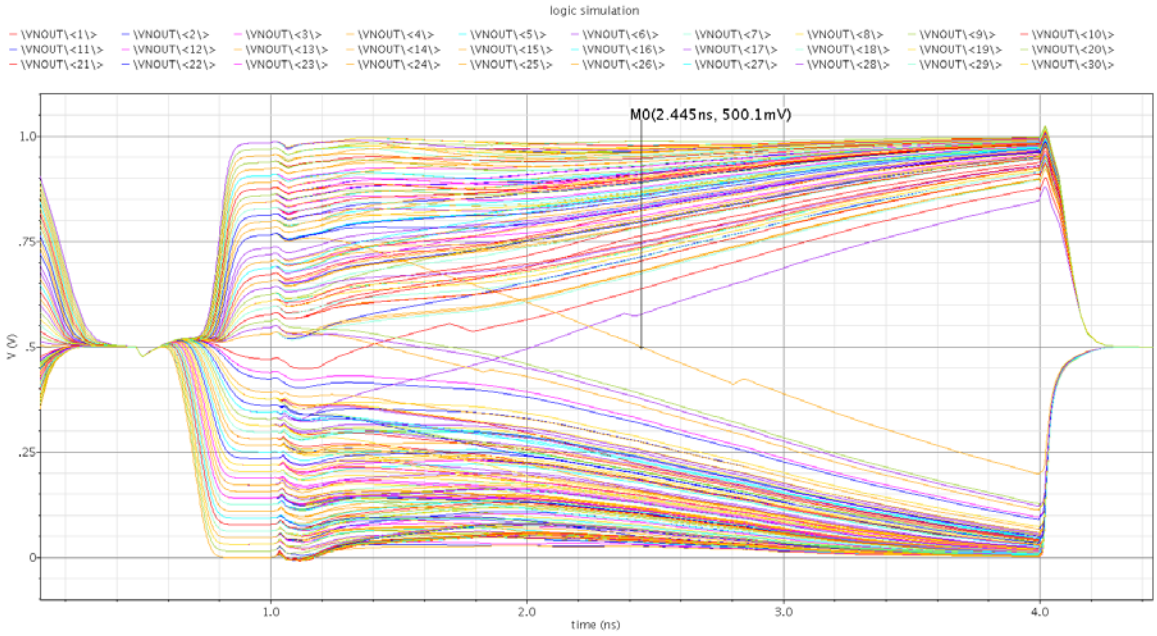


Figure 4.46: Decoding in slow mode. First 0.5 ns was the pre-charge to VDD/2 time. 0.5ns to 1.0 ns was the LLR voltage loading time. The decoding starts at 1.0 ns and ends at 2.445 ns.

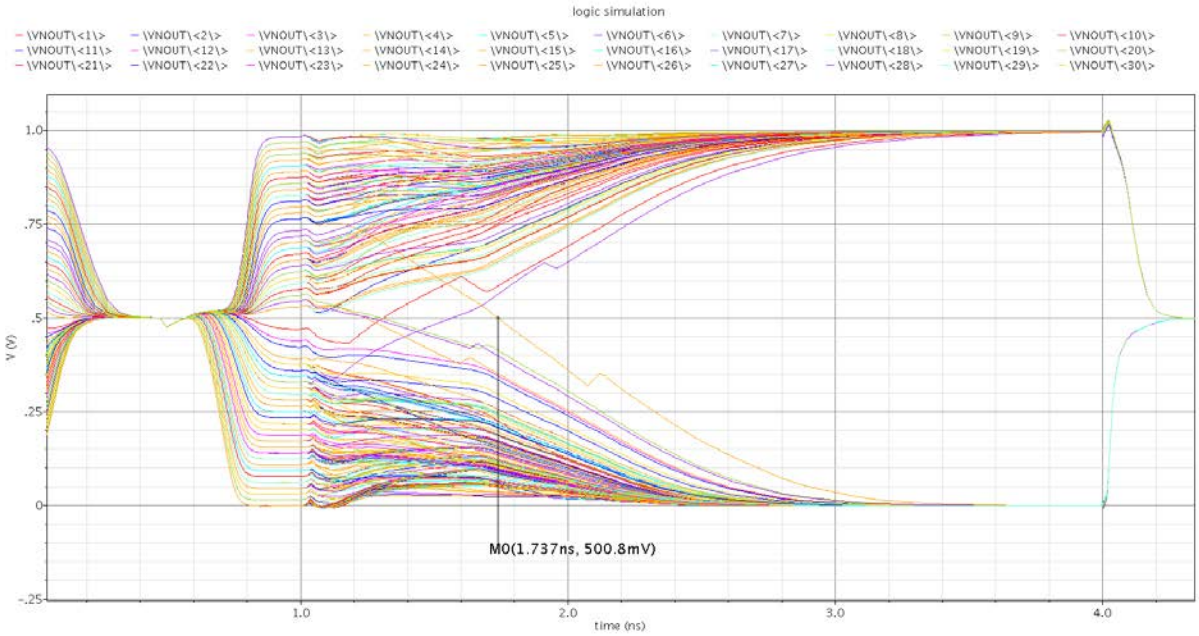


Figure 4.47: Decoding of the same codeword as in Figure 4.46 in moderate speed mode. First 0.5 ns was the pre-charge to VDD/2 time. 0.5ns to 1.0 ns was the LLR voltage loading time. The decoding starts at 1.0 ns and ends at 1.737 ns.

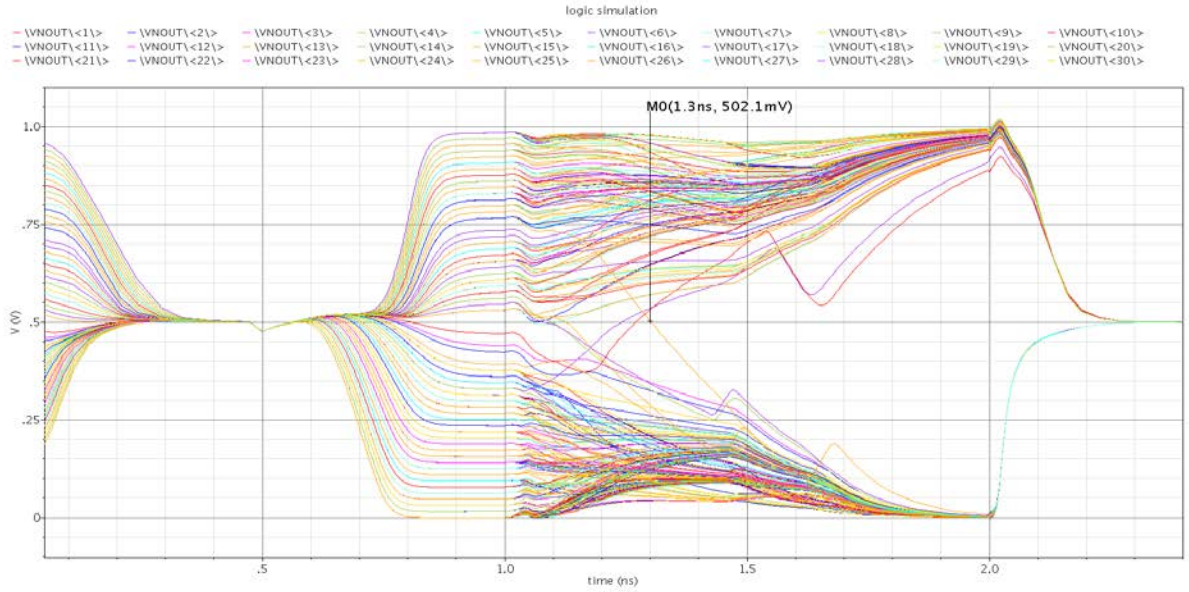


Figure 4.48: Decoding of the same codeword as in Figure 4.46 and 4.47 in fast speed. First 0.5 ns was the pre-charge to VDD/2 time. 0.5ns to 1.0 ns was the LLR voltage loading time. Decoding starts at 1.0 ns, ends at 1.3ns.

As it has already been discussed, the end of decoding is indicated by the parity-check satisfaction of all the check nodes. So, let us see in Figure 4.49 shows the end of decoding in CN output trajectory.

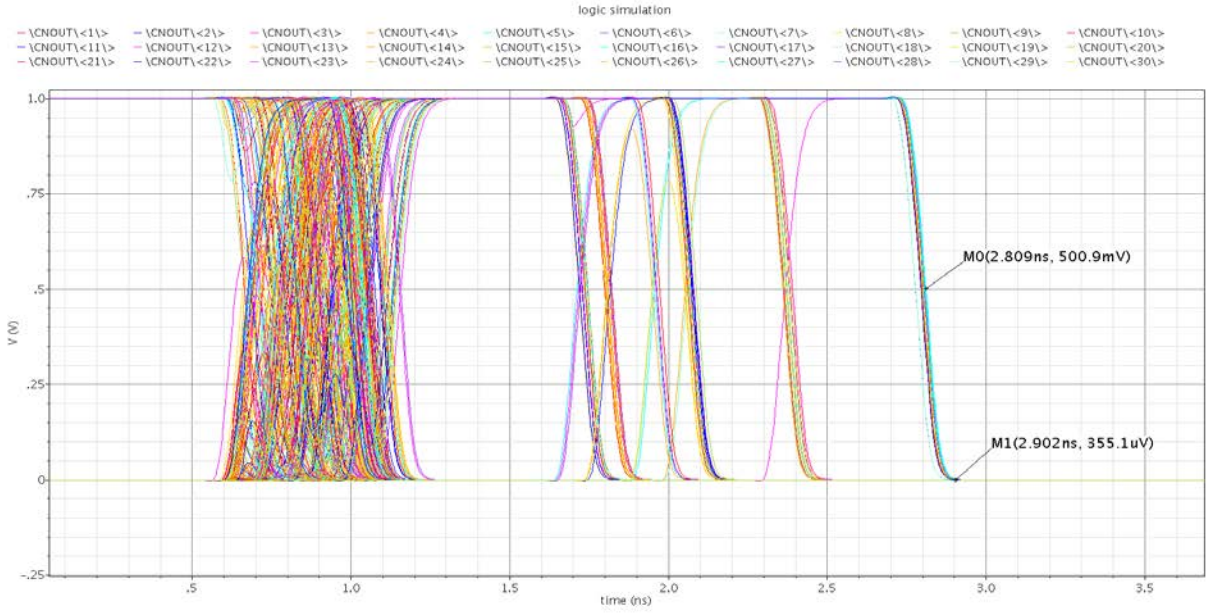


Figure 4.49: The return of all the CNs to zero indicates the end of decoding. Decoding starts at 1.0 ns. This figure correspond to Figure 4.46 of slow mode decoding, where it took VNs 1.45 ns to decode but here in CNs, it takes almost 1.9 ns to decode. CNs indicate the actual termination of decoding by parity-check.

Before submission of the designed decoder IC for fabrication, successful simulation of 100 codewords have been completed, taking into account the parasitic capacitance of the CN output wire. Figure 4.50 depicts a fraction of these simulations in only 10 of them.

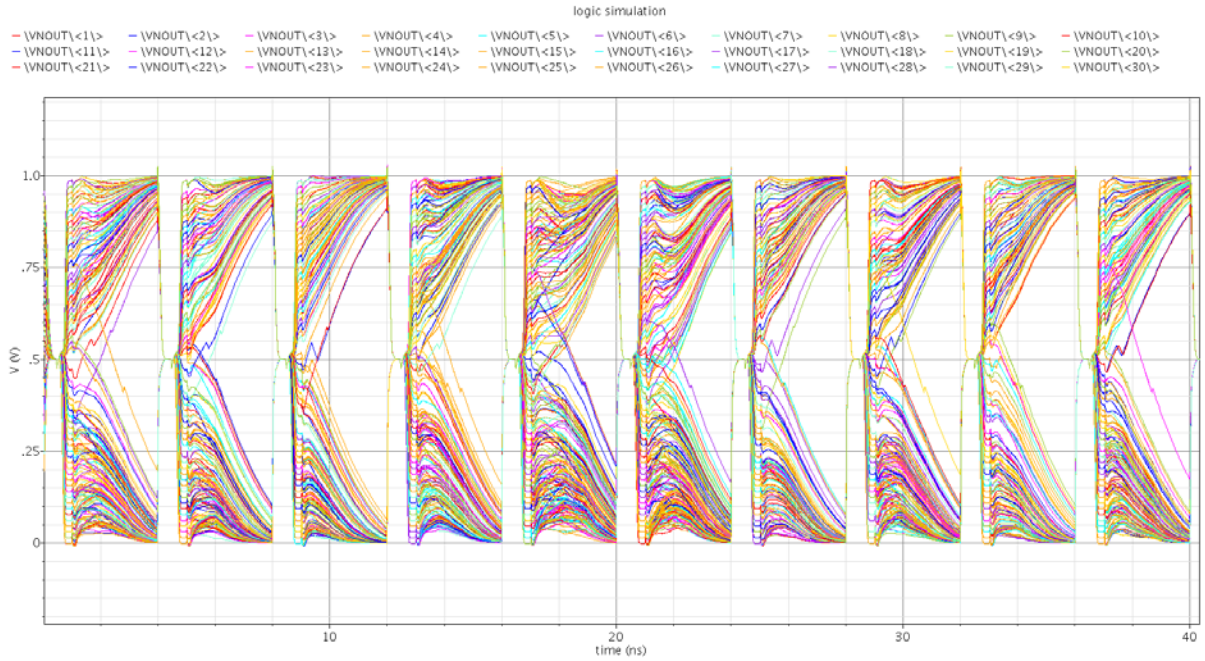


Figure 4.50: VN trajectories showing successful decoding of 10 codewords.

4.10.2 Power consumption

At this point, the energy consumed by the core decoder needs to be calculated. It is important to note that the complete chip is made up of a number of testing circuitry in addition to the core decoder. For example, the memory array storing the data bits, the DAC generating quantized VN levels, shift registers passing in data bits or bringing out the decoded bits, etc. They, however, do not directly take part in the decoding process. Therefore, only the energy consumed by the core decoder is of interest and will be reported here.

The VDD and ground voltages used in this decoder are 1.0V and 0V respectively. In the schematic with which the simulation is being done, all the VDD nodes of the core decoder are tied together to conveniently obtain the current drawn by the circuitry inside the decoder. In addition to this, it is also necessary to include the current drawn by the VNs during the pre-charge phase. This current pre-charges the VNs to VDD/2. Interestingly, while some VNs, initially staying below VDD/2, need to charge up to reach VDD/2, some other VNs, initially

lying above VDD/2, get discharged to reach VDD/2. Therefore, there is good amount of local re-distribution of current on the VDD/2 voltage wire and the current drawn for pre-charging is less compared to what one would expect if all the VNs were to be pre-charged to VDD/2 starting from 0V initially.

Figure 4.51 shows the current drawn by the core decoder at voltage VDD and also the current drawn at voltage VDD/2 during pre-charge phase. The purple and the orange pulses represent the pre-charge and the START signals respectively. The red and the blue curves indicate the current drawn by the decoder at VDD and at VDD/2 respectively. Careful inspection shows that there is a time gap between the pre-charge signal being turned off and the START signal turning on. In this gap, there is the charge-share signal turned on, which is not shown in the figure. The charge-share signal re-distributes the voltages inside the DAC to generate appropriate quantized voltage level on VN capacitor.

At the start of the pre-charge phase, there is a steep dip as shown on the blue curve. During this short interval, most of the currents drawn for pre-charging take place. As all the VNs quickly settle to the voltage of VDD/2, there is no more current drawn from VDD/2 wire except for little leakage current. Although the decoding does not start until the START signal is turned on, the threshold inverters, de-embedding circuitry and the check nodes remain active during this pre-charge period. Therefore, a good amount of current is drawn during this duration. As soon as the pre-charge signal is turned off, the charge-sharing signal turns on and stays on until the START signal turns on. During the charge re-distribution, the quantized VN levels are formed. This enables all the CNs and de-embedding circuitry to re-calculate their values and in doing so, they draw a good amount of current as can be seen in the red curve. After the START signal turns on, some CNs change decision and they soon reach parity-check satisfaction. After all the CNs are satisfied, there is still some steady state current drawn by the decoder – the value of which is around 6.7 mA as shown in the figure. This current is drawn because in the steady-state situation, lots of circuitry are biased on and they draw drain current to maintain their position. For example, the threshold inverters are always turned on and continue to drain current. By this time, the flipping of CNs, which is also a major source of power consumption, comes to an end as they reach pari-check satisfaction.

To understand more about the large amount of current drawn by the decoder from VDD node during the pre-charge phase, another simulation has been done with long duration of the pre-charge phase. The currents drawn over different phases in the simulation are plotted in Figure 4.52. The red curve showing current drawn from VDD node is of interest in this figure. At the start of the pre-charge phase, current drawn from the VDD node increases and as the VNs get pre-charged to VDD/2, this currents settles down to a static value of around 305 mA. Interestingly, the blue curve showing energy consumption by the VNs during pre-charge stays the same as that in Figure 4.50. Therefore, long pre-charge phase clearly illustrates the static power consumption by the threshold inverters during this time. Interestingly, as the pre-charge signal turns off and the charge-sharing signal turns on, the VNs rise to their quantized voltage levels. For a short duration, there is some energy consumption due to the re-calculation by the XORs inside the CN. Then, the current drawn from VDD quickly approaches close to zero amperes. As the START signal turns on following the charge share signal turning off, there is little energy consumption since the CNs switch position for some time until parity-check satisfaction is achieved.

The red and blue plot data from Figure 4.51 have been exported to MATLAB. Their time steps being unequal, piecewise integration of the current drawn has been done. Multiplying these integrated values to the corresponding voltage (VDD or VDD/2) gives energy supplied to the decoder from VDD and VDD/2 respectively. The average energies consumed over a period of 4ns at VDD and VDD/2 are 0.3537 nJ and 0.2904 pJ respectively. If one codeword is taken to be decoded in 4ns including the pre-charge and charge-sharing time, the energy per bit is the total energy consumed by the core decoder during this time divided by 273. The calculation yields an average power consumtion of 88.51 mW and energy per bit of 1.297 pJ/bit. Similarly, calculations for simulations done in 2.04 ns cycle time were made. The average power consumption is 169.75 mW and energy per bit is 1.267 pJ/bit. Interestingly, the latter case has almost double throughput but also almost the double power consumption. Therefore, the energy per bit is almost same in both cases. Inspection provides us the reason behind this. In almost all of the 100 codeword decoding, the decoding is complete within around 2.04 ns. This is why it is the average decoding time requirement. After all the codewords are decoded, the VN levels are very close to the rail voltages. This puts an end to the current drained by the threshold inverters

connected to the VNs except for small leakage current. In other words, after around 2.04 ns, there is almost no further energy dissipated by the decoder.

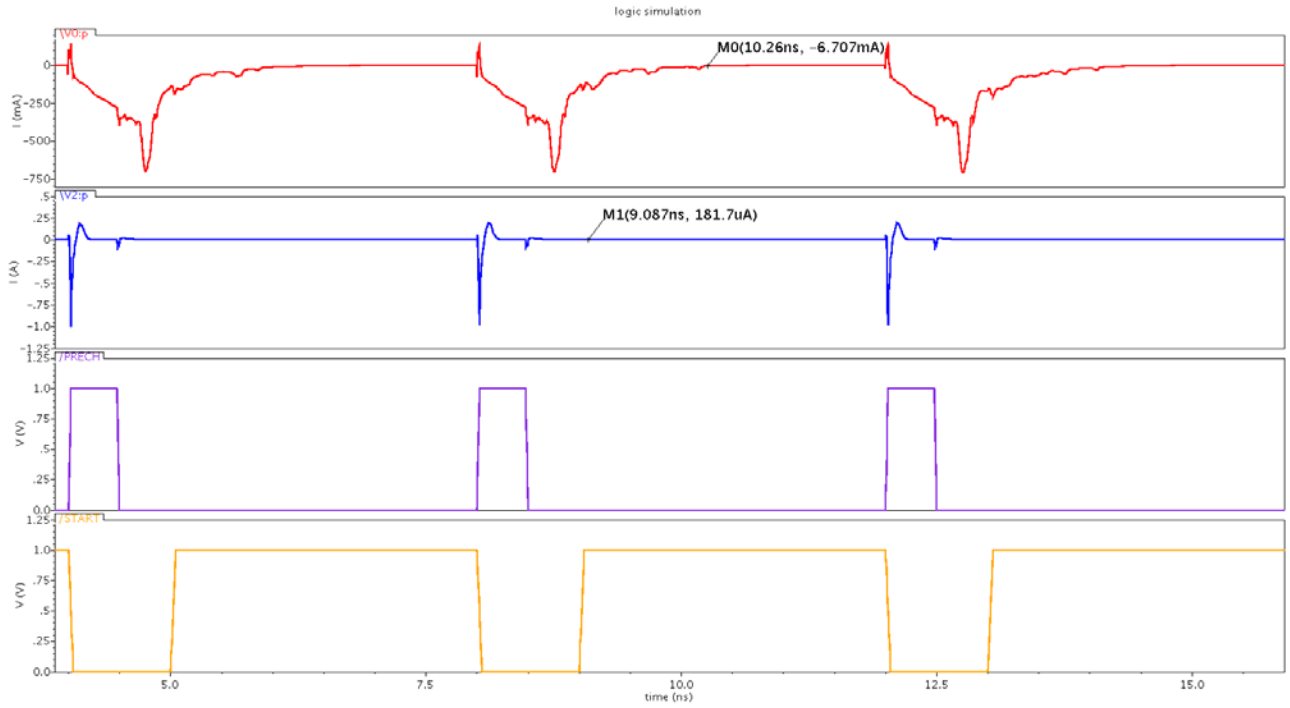


Figure 4.51: Top (red) graph shows current drawn by the decoder at VDD. Second from top (blue) graph shows the current drawn at VDD/2. Third from top (purple) graph indicates the pre-charge signal while the bottom (orange) graph represenst the START signal turned on during decoding.

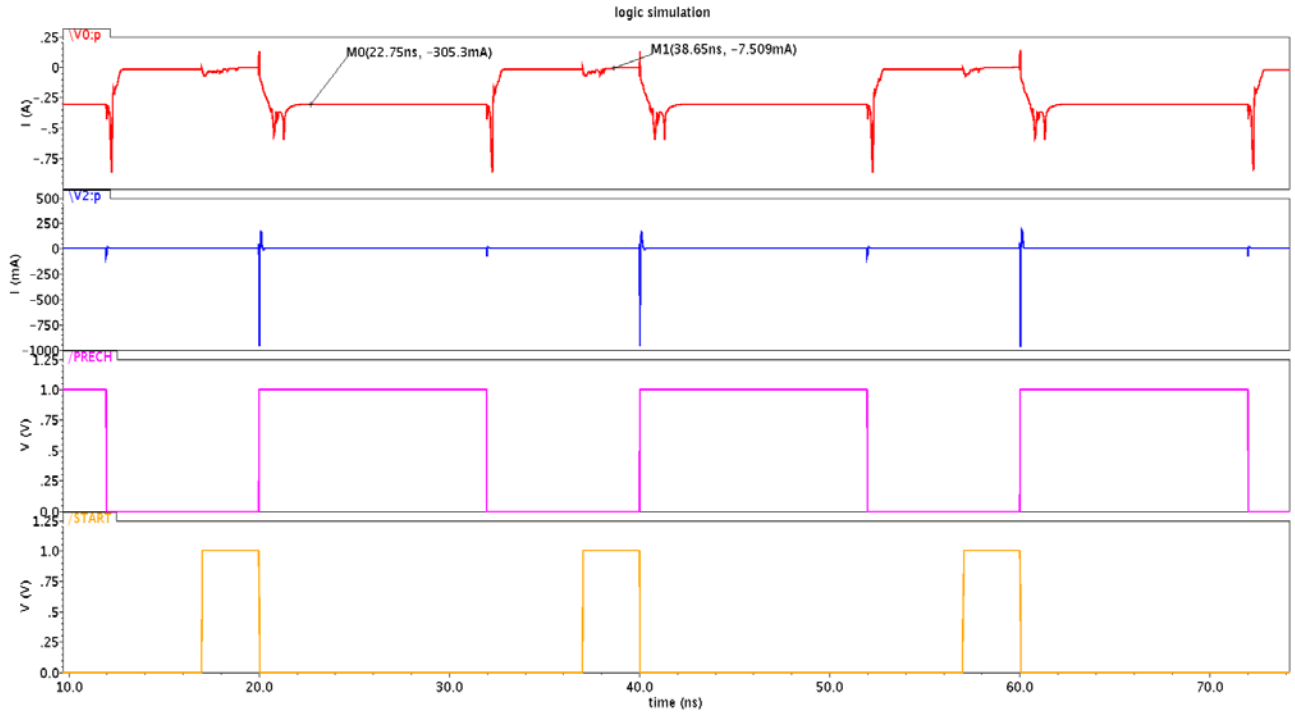


Figure 4.52: Top (red) graph shows current drawn by the decoder at VDD. Second from top (blue) graph shows the current drawn at VDD/2. Third from top (purple) graph indicates the long pre-charge signal while the bottom (orange) graph represenst the START signal turned on during decoding.

4.10.3 Monte-Carlo Simulation

Although we were able to decode a good number of noise-added codewords in different speed modes and calculate the throughputs, the simulations did not previously take into account the random analog impairments, such as the random threshold inverter mismatch and the random mismatch in current supply of the current-source inverters. Unlike systematic skews, these random mismatches cannot be eliminated by on-chip calibration and global routing scheme. Therefore, it is important to show that the designed decoder is resilient to the random mismatches. To show the decoder behavior in the presence of random threshold inverter mismatch of standard deviation 12.88 mV and random current-source inverter mismatch of scaled standard deviation of 5%, monte-carlo simulation has been run in Cadence Spectre and VN and CN trajectories showing successful decoding have been obtained. In simulation, each decoding cycle of 4ns takes around four days to start transient monte-carlo simulation, and additional five hours to complete decoding each codeword. Figure 4.53 illustrates VN trajectories for one decoding cycle in transient monte-carlo simulation.

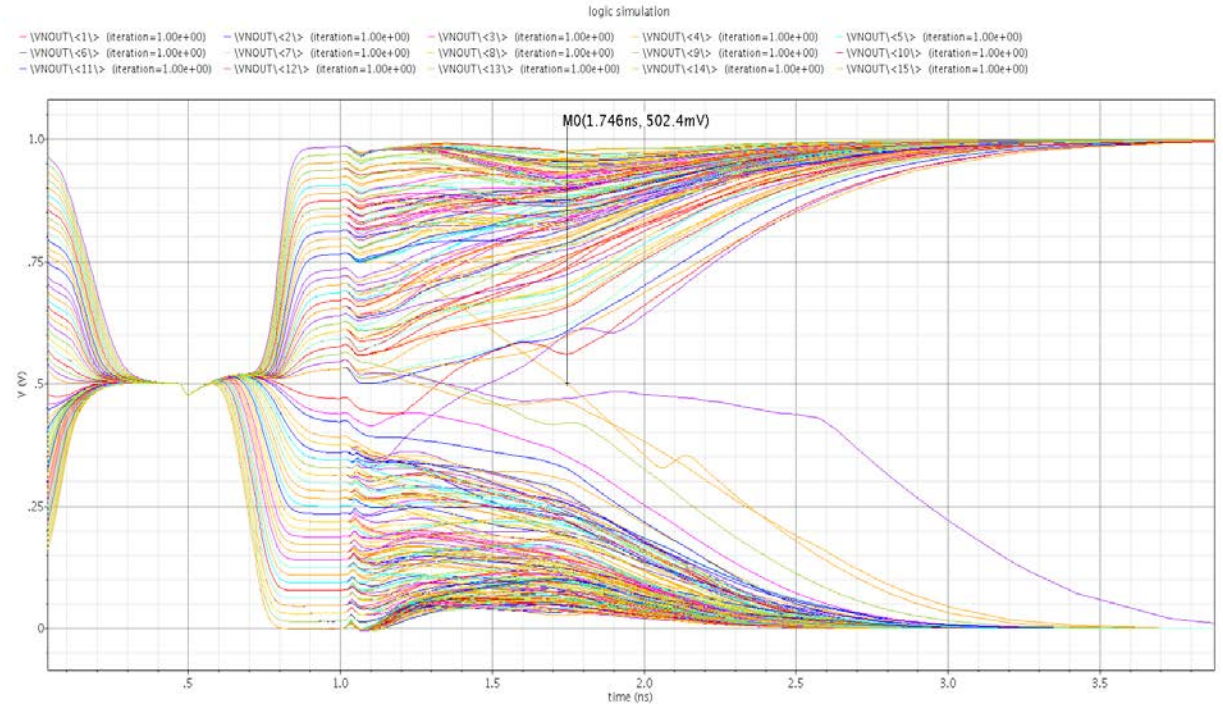


Figure 4.53: Monte-Carlo simulation in Cadence showing VN trajectories for one decoding cycle.

In Figure 4.53, the first 0.5 ns was allocated for pre-charging VNs to VDD/2 and the next 0.5 ns was allocated for decoder initialization. At 1.0 ns, decoding starts and the VN trajectories move to the correct sides. In the example shown in this figure, it takes 0.746 ns for the VNs to completely move to the correct sides of VDD/2 from the start of decoding. However, the termination in decoding is indicated by the satisfaction of all the CNs. In Figure 4.54, we have put the CN trajectories corresponding to this example. Although the VNs took only 0.746 ns to move to the correct side of VDD/2, it took 1.696 ns for the CNs to be completely parity-check satisfied. Figure 4.55 depicts VN trajectories of ten codewords indicating successful decoding of all of them.

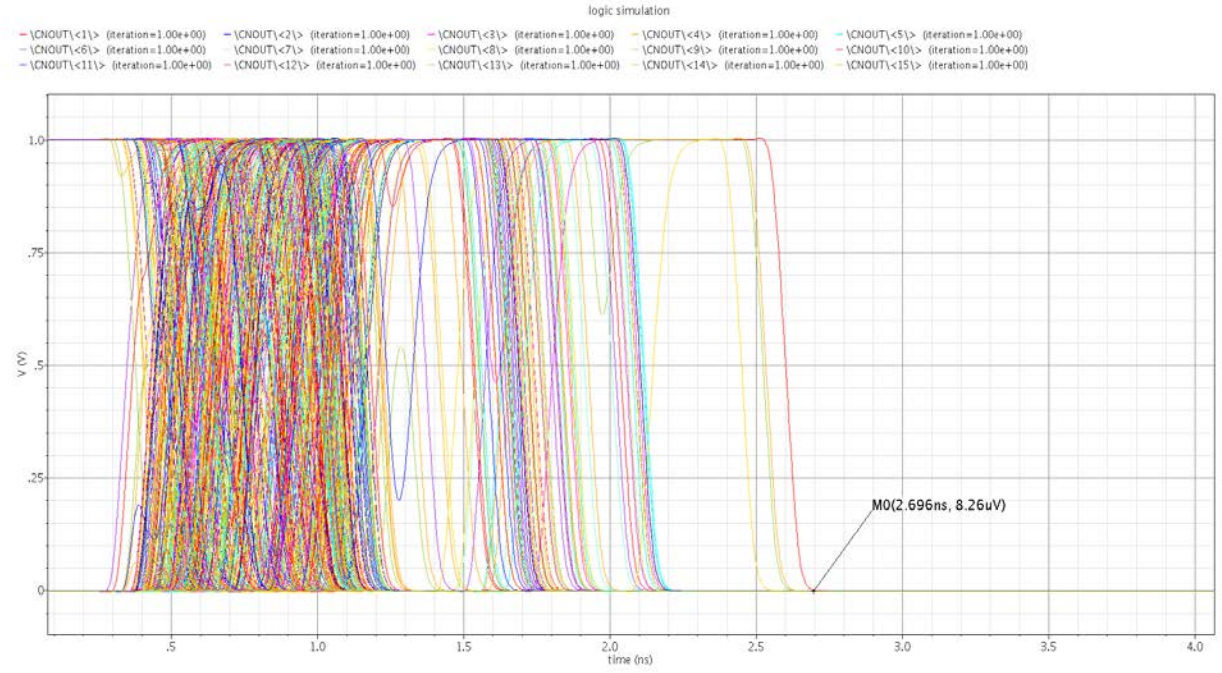


Figure 4.54: Monte-Carlo simulation in Cadence showing CN trajectories for one decoding cycle.

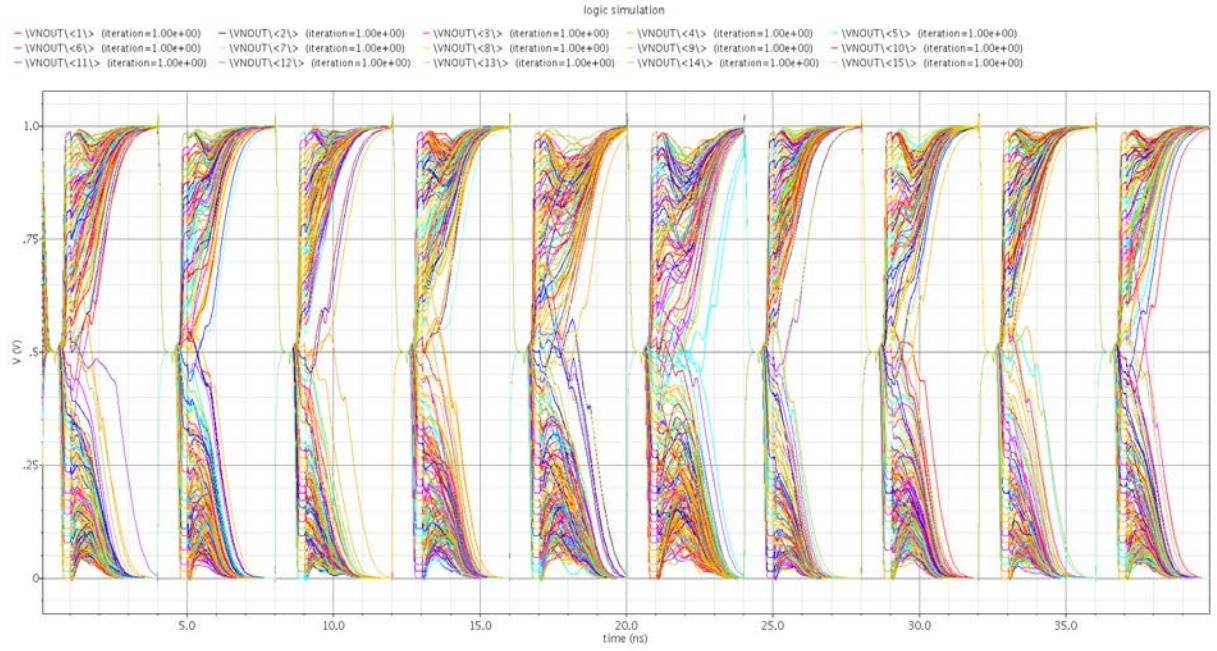


Figure 4.55: Monte-Carlo simulation in Cadence showing ten codewords being successfully decoded.

4.10.4 Performance Comparison with State-of-the-art

The throughput and power consumption obtained from simulations in Cadence Spectre have been used to compare against the performance metrics reported in literature. First, the designed decoder's performance will be reported and compared with other designs in the same technology TSMC 65nm using MDD-BMP algorithm. Table 4.5 illustrates this comparison. After that, comparison will be done with other LDPC decoders in a broader range, i.e., recently published designs of LDPC decoders employing different decoding algorithms and implemented in different technology node. This will give us a bigger picture to understand where our decoder fits in terms of different performance/complexity trade-off.

Table 4.5

Peformance comparison for FG (273, 191) MDD-BMP decoders in TSMC 65nm Technology

	This work		MDD-BMP[18]	MDD-BMP[13]	
Supply voltage (V)	1.0		1.0	1.0	0.8
Average throughput (Gb/s)	68.25	134	137	37.4	25.8
Average power (mW)	88.51	169.75	76	183	79.3
Average energy (pJ/bit)	1.297	1.267	0.56	4.88	3.07

The performance metrics listed in Table 4.5 compare this decoder's performance with those reported in [13] and [18]. First, the throughput calculation assuming instantaneous reset and re-program of the VNs as soon as the previous decoding finishes is not possible in practice. In real world situation, the decoder has to go through a pre-charge phase to pre-charge VDDs to VDD/2 and then a charge share phase to load VNs with quantized voltages. After these two phases, the decoding can actually start. Therefore, each decoding cycle should include the pre-charge and charge share time durations in addition to the decoding time. If 0.5ns is allocated for pre-charge and another 0.5 ns for charge sharing, the decoding cycle will be 1.04+ 0.5 +0.5 =2.04ns. This correspond to a throughput of 0.4902 x 10^9 codewords per second x 273 bits per codeword = 134 Gb/s. This value is almost same as the throughput reported in [18]. In addition to that, a slower decoding taking overall cycle time of about 4ns will result in a reduced throughput of 68.25 Gb/s.

The power consumption by this decoder at higher throughput mode as reported in Table 4.5 is little higher than the one in [18]. It is important to note that this work is a complete work on the decoder which takes into account the problems of delay matching and systematic offsets and applies circuit-level solutions to address them. For example, the delay-insertion inverter chain is always on and drawing some amount of drain current. More importantly, there are DACs in this design, which consume energy during pre-charging capacitances to VDD/2, including the pre-charge of analog VN wire to VDD/2 as well. This pre-charge phase is absent in [18], where

all VNs are directly loaded their quantized values, and no method of generation of quantized voltages was proposed. Consequently, the energy per bit for this work is little higher than that reported in [18] but is still significantly lower than the work in [13]. However, Figure 4.52 showed the good amount of static current which is being dissipated during the pre-charge phase. The reduction in this current by modifying the de-embedding loop can be a nice option for future work.

Now that we have compared this decoder with some other decoders using MDD-BMP algorithm and implemented in the same technology, let us put another table showing a broader range of comparisons with designs published in recent literature. Table 4.6 illustrates these comparisons. In terms of area, throughput, and energy efficiency, our design provides competitive performance compared to decoder designs published in recent literature. Apart from these metrics, bit-error-rate performance is another important metric for comparing any decoder performance. For this design, the bit-error-rate will measured after the fabricated and packaged chip comes back and testing is performed. At present, 100 monte-carlo simulations have been done in the presence of random mismatch and the decoding has been shown to be robust aginst these variations. Therefore, there is an expectation to obtain good BER performance from the implemented design.

Table 4.6

Comparison of this work with state-of-the-art

	This work	Hemati et al. [19]	Darabiha et al. [20]	Chen et al. [21]	Rabbani et al. [22]	Cheng et al. [23]	Toriyama et al. [24]
Decoding algorithm	MDD-BMP LDPC (273,191)	Min-Sum LDPC (32,8)	Modified Min-Sum LDPC (2048,1723)	LDPC-CC N=491, R=1/2	TS-LDPC (120,75)	NP Min-Sum LDPC (2048,1723)	Non-binary LDPC
CMOS technology	65nm	180nm	130nm	90nm	90nm	90nm	40nm
Supply voltage (V)	1.0	1.8	1.2, 0.6	1.2	1.2	0.9	1.2 V
Core area (mm^2)	1.40	0.57	7.3	2.24	1.38	9.6	4.73
Throughput (Gb/s)	134	0.006	3.3, 0.648	2.37	0.75	45.42	2.267
Average power (mW)	169.75	5	518 @ 4dB	284	13	1110	212.4
Energy efficiency (pJ/bit)	1.267	830	10.4 @ 4dB	24	17	9.4 @ 4.4 dB	93.7
Bit-error rate	–	4×10^{-4} at 5 dB SNR	10^{-5} at 4.7 dB SNR	10^{-5} at 2.5 dB SNR	10^{-5} at 5.2 dB SNR	10^{-7} at 4.4 dB SNR	–

Chapter 5

Physical Design of the Decoder

5.1 Layout of the Core Analog Decoder

The low-density parity-check decoder description matrix H has 273 rows and 273 columns. Rows indicate parity-check equations, with 1s representing connections from variable nodes. In other words, each row is a CN with 17 1s as its corresponding VNs and each column is a VN with its 17 1s as corresponding CNs. There are 17 ones in each row and also 17 ones in each column in the description matrix that have been used. Therefore, the H matrix is sparse in nature. Direct uncompressed implementation of this matrix with hand-drawn wirings and connections will occupy a very large area most of which is empty. It is noteworthy that the digital decoder is implemented by placing smaller layout blocks in an irregular fashion where place and route tools are utilized for wirings and connections. Therefore, the area consumed is much less with little blank space compared to what the manual hand-drawn layout would occupy. On the other hand, physical design of the analog decoder requires a regular implementation so that the connections between blocks can be manually drawn. This motivates us to find out a regular compression of the sparse H matrix.

Although the H matrix is sparse in nature, its compression poses a number of challenges. First, irregularity of the location of 1s makes compression difficult. Suppose, one column space has been decided to be allocated in layout for four successive merged columns because in most rows there are only 0s or one or two 1s in successive columns. However, there are always a few rows with all 1s in four successive columns. This phenomenon happens for compression of rows as well. This makes compression of any number of rows or columns into one difficult. Second, even if several rows can be compressed into one somehow, each CN represented by each row has to stay distinct from other CNs. This, in turn, means that the wirings belonging to CNs will not

compress. Finally, the compression strategy has to be regular in the sense that the same amount of compression has to be continued all through the H matrix. So, a cluster of neighboring 1s in a certain area will be compressed the same way as other blank spaces.

Before our compression strategy is discussed, it is important to mention that in our physical design the transpose of the H-matrix, H^T have been used. This use has no significance from the compression point of view. Instead, it has been done for convenience. In the transpose form of the H matrix, the rows now represent the VN. Since the DAC outputs come from the left side of the core decoder, it is simply more convenient to feed the 273 DAC outputs directly to the corresponding 273 VN rows of the decoder instead of routing them to the corresponding columns. The columns in our design are now CNs.

After a lot of experiments on this H^T-matrix in MATLAB, it has been found that in any particular row of the H^T matrix, there are never more than two 1s in successive eight entries. Figure 5.1 illustrates a 16 by 16 submatrix taken from H^T. It can be seen from this figure that within successive eight entries in a row, there are maximum two 1s. Therefore, if 2 column spaces are allocated for eight successive columns, it is completely doable. This finding allows us to compress the H^T matrix fourfold horizontally. To understand how many eight successive entries contain two 1s or one 1, or no 1 at all, let us calculate a compressed matrix by adding the entries in eight successive columns into one column. The size of the compressed matrix is now 273 by 35. This addition leads to three unique numbers in the compressed matrix – 2, 1, and 0. The distribution of these numbers are displayed in Figure 5.2. A little calculation states that 38.6% entries of the compressed matrix is non-zero while only 6.2% elements in the H^T matrix were non-zero.

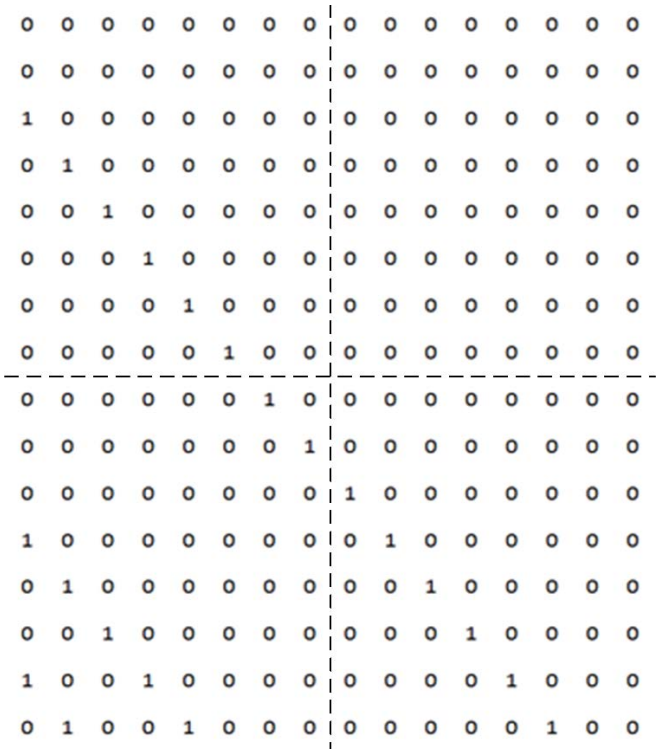


Figure 5.1: 16 by 16 submatrix taken from H^T.

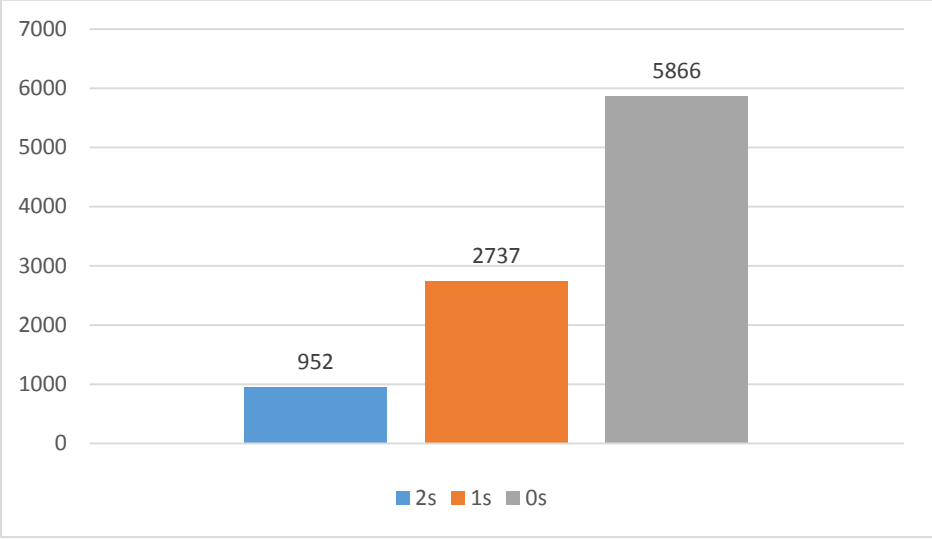


Figure 5.2: Bar chart shows the distribution of number of 2s, 1s, and 0s in the compressed matrix.

It is important to note that the compressed matrix that has been used for our physical design is not the one whose distribution is shown in Figure 5.2. The compressed matrix discussed in the previous paragraph just shows a quantitative calculation of how this compression leads to more efficient use of area in physical design. On the other hand, the new mapped matrix that will be used for the physical design has two columns allocated for successive eight entries, effectively leading to fourfold horizontal compression. The mapped matrix has 272/4+1= 68+1 = 69 columns. The number of rows, 273, stays the same. Interestingly, similar compression is possible in the vertical direction as well, leaving the horizontal dimension uncompressed. This would be problematic in the sense that matching DAC row pitch to the decoder row pitch would be another issue to address. So, horizontal compression of H^T matrix seemed most convenient to us. The MATLAB codes used for compression are attached in Appendix A.

Let us remind ourselves the layout blocks that need to be placed inside the core decoder. The layout placement of these blocks is more complicated in the physical design compared to the schematic where there is no compression issues to take care of. There are two major blocks – CN and VN. CN is made up of a few 2-input XOR gates cascaded to build a 17-input large XOR gate, which is the CN itself. While laying out the XOR gate is straightforward, the placement of these blocks is the trickiest part. The structure of the 17-input XOR gate has been discussed in detail in the schematic design part of this thesis. Here, our discussion will be only how the locations of placement were chosen and the wiring connections between the 2-input XOR gates.

In each column of the 69 column and 273 row H^T matrix array, there are 17 1s, meaning that there has to be 17 VN circuits in these locations. The VN circuitry includes threshold inverters, delay-tuning inverters, a de-embedding XOR gate, current-source inverter, and a decoder start TG switch at the end. Figure 5.3 shows the part of the VN circuitry placed at the locations of 1 in the H matrix. The thresholding inverter pair is common between the VN and the CN, both in the schematic and the physical design. Inside the CN, the outputs of these thresholding inverter pair are fed to the 2-input XOR gates. Each of these XOR gates has two inputs but each row can provide only one thresholded VN state out of a thresholding inverter pair. Therefore, one XOR gate has to be placed for two 1s in a column of the mapped matrix. If we go from the bottom of the chip to the top and consider the indexes starting from the bottom

towards the top, the XOR placement for two consecutive 1s in a column goes to the second location. Figure 5.4 revisits the 17-input XOR gate broken down into 20 2-input XOR gates.

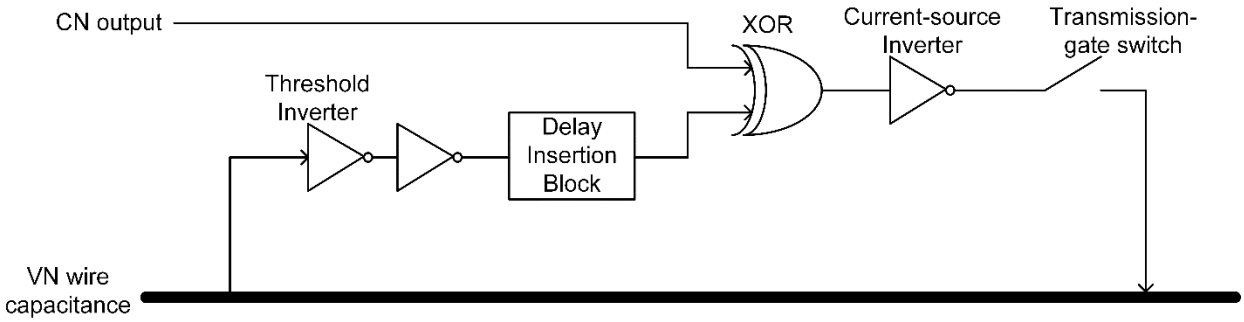


Figure 5.3: Diagram showing part of the variable node. The circuits shown in this figure is put in the places where there is a 1 in the H matrix

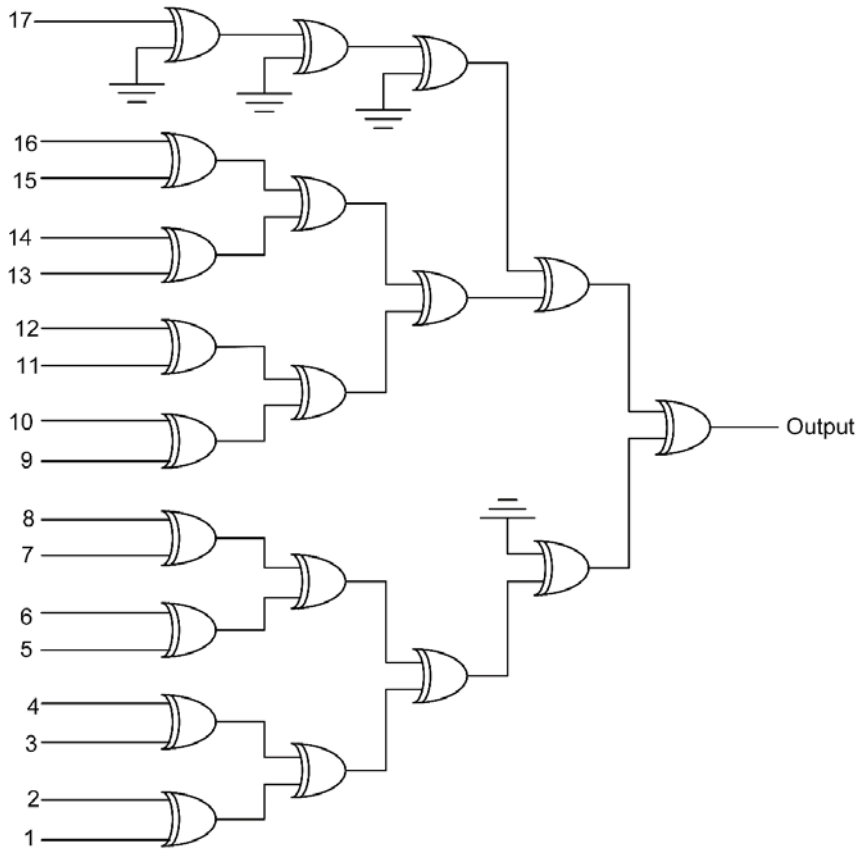


Figure 5.4: Schematic of CN 17-input XOR gate.

The distribution of the 20 2-input XOR blocks has been done in a way to best use the space. 15 of them have been placed into the space at the left of the VN circuitry in the locations of 1s in the column of the mapped matrix. 2 of them have been placed above the 273 rows and 3 of them in the middle of the 273 rows – above 136th row and below 137th row. The placement locations and wiring connections of 20 2-input XOR gates for one CN is shown in Figure 5.5.

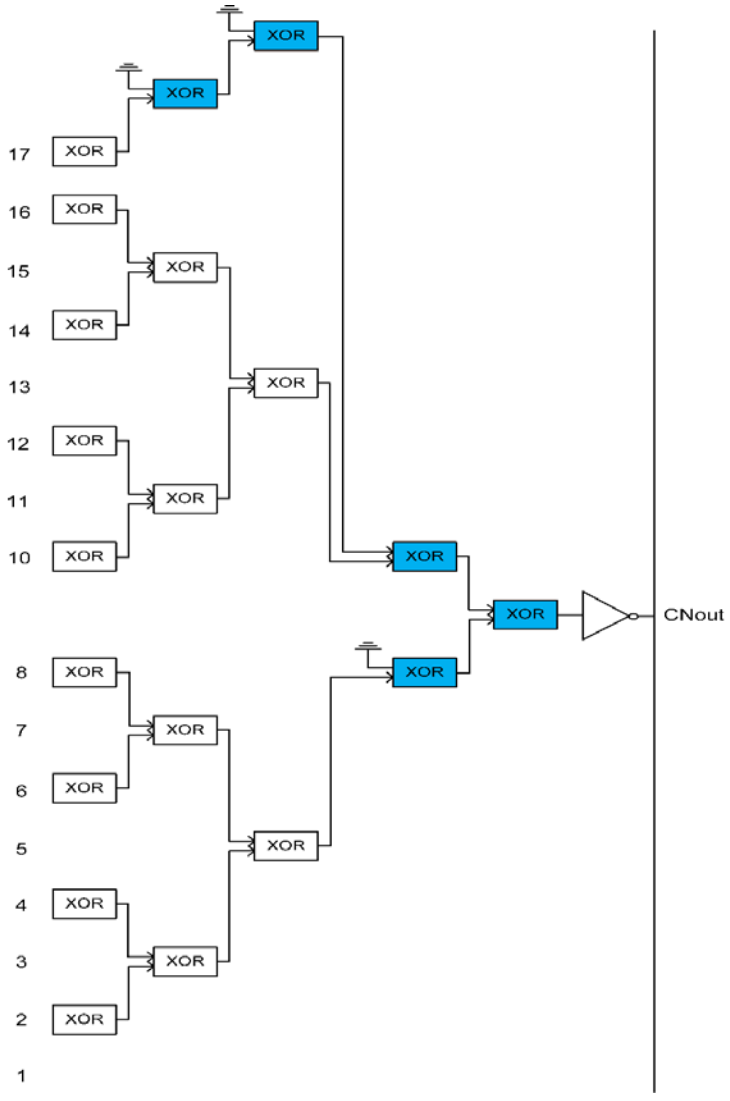


Figure 5.5: Placement of 2-input XOR blocks inside one CN. The number shows the indexes of the location of 1s in the column of mapped H^T matrix. Blue XORs do not correspond to these 1s' locations.

In addition to the MATLAB-generated file that enumerates the indices of the rows where there are 1s in each column, there is another MATLAB-generated file that indicates which 1 is the second 1 in a certain row and a block of 8 successive entries. If the 1 is the second 1, the XOR placement is shifted by a column. As it has already been said, there can be maximum two 1s in successive eight entries in a certain row. If there is a second 1 in any row, the XOR in Figure 5.7 is placed there, otherwise, the XOR in Figure 5.6 is placed.

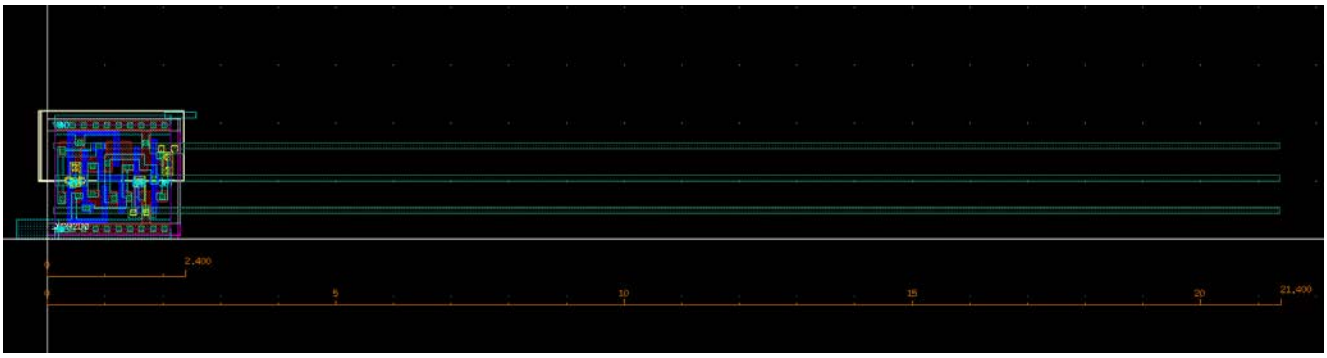


Figure 5.6: 2-input XOR. It is placed in the index of first 1 of eight column block.

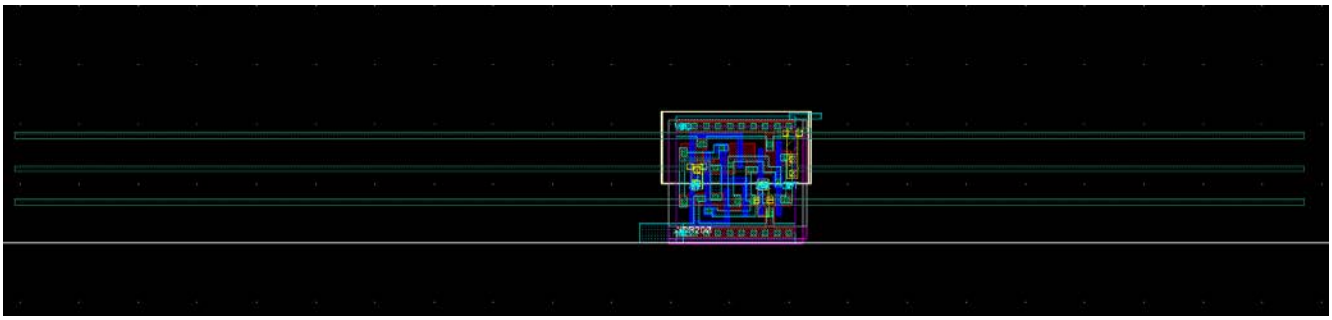


Figure 5.7: 2-input XOR. It is placed in the index of second 1 of eight column block.

Now, the placement of the blue XOR blocks in Figure 5.5 is different. There are 5 such blue XOR blocks for each CN, and so, 20 such for a group of eight. Since they are not being placed according to the indices of the 1s in the mapped matrix, their placement is well-defined.

For a block of eight CNs, above the 273 rows, there are 2 blank columns and also 8 extra rows – 274 to 281 are taken. These eight places will be filled up by 2 blue XORs from each of the eight CNs. Similarly, above the 136^{th} row, there comes the blue XOR blocks in the middle. Again, there are two blank columns left and 12 rows have to be taken to accommodate 3 blue XORs from each of the 8 CNs. The reason the blue XOR blocks have been placed in the middle is to obtain almost equal delay in all the paths from the input of the 17-input XOR gate to its output. The interesting way of placing XOR blocks in a location and placing the XOR for the second 1 is a rightward shifted location has also been followed for placement of the other circuitry inside VN.

The placement of all the core decoder circuits, including the gates, switches, metal wiring and contacts, has been fully automated using Cadence SKILL code. This gives us a tremendous capability to bring changes in placement, wiring, and distances between blocks to meet the DRC and LVS requirement during the physical design. Even the placement of the other more regular blocks, such as, 273 DACs, SRAM memory array, shift register, etc., has been done using the SKILL code. Manual hand placement of these blocks would otherwise take days if not more and be time consuming to modify in the event of a design change. The SKILL codes used in this design have been attached in the Appendix B of this thesis.

Each CN output runs all the way from the bottom to the top of the core analog decoder encompassing 273 rows. While it could vertically stretch up to the bottom-most place it is used by the connected VNs, this would have created unequal vertical lengths for different CN outputs. In order to have equal parasitic capacitance at each CN outputs, the CN output wires were drawn until the bottom of the analog decoder. In the physical design, the CN output will have good amount of capacitance. This capacitance, valued 150 fF as obtained from the parasitic extraction in Cadence, has been included in the simulation of the core decoder. Furthermore, the rightmost XOR block in Figure 5.5 is followed by a high current digital inverter for buffering of the CN output capacitance.

Unlike the CN output, VN outputs are analog nodes. Since its parasitic capacitance is being used as the VN capacitance, all the VNs have to be made to retain same amount of capacitance. To do this, the VN has been shielded from above and below and by the sides. While designing the core decoder, three metal layers have been reserved for the VN – M5, M6, M7.

The VN wire, made of M6 metal layer, stretches from the very left to the very right of the core decoder and has a width of 0.10 µm. It is connected to the corresponding DAC output to the left and to the input of the thresholding inverter pair of the output shift register to the right. A layer of M5 metal and a layer of M7 metal covers it from below and top respectively. By either side of VN output, there are two parallel running M6 metal wires. Figure 5.8 shows the shielding of the VN output. The red filled horizontal line is the VN output in metal M6. There are M6 wires in parallel on the top and the bottom. There are M7 and M6 shields in blue and green respectively. M7-M6 contacts connect the M7 to the top parallel M6 wire, which is in turn connected to VDD. M5-M6 contact connects M5 metal to the bottom running M6 wire, which is also connected to the GND. So, the shielding of the VN wire is not floating, rather, it supplies VDD and GND. The power grid, which supplies VDD and GND with the two highest levels of metal and has vertical and horizontal connections at regular intervals to minimize the resistance across it, is also appropriately shorted with the shielding wires.

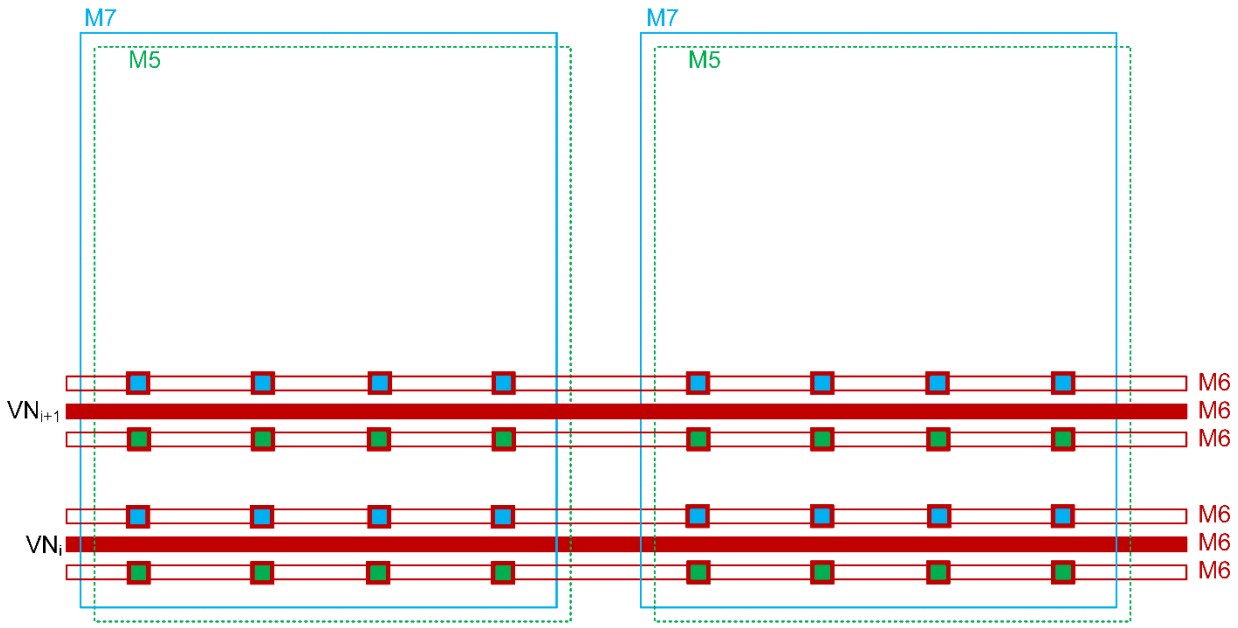


Figure 5.8: Shielding of the VN output. Blue vontacts are between metal 6 and 7 while green contact are between metal 5 and 6.

The VN includes the thresholding inverter pair (common between VN and CN), a pair of delay-inserting inverters, a de-embedding XOR gate, current-source inverter, and a TG switch. All of these have to be placed in the location corresponding to the index of 1 in the mapped H^T matrix. For convenience, all these have been packed into a unit shown in Figure 5.9. The choice of dimensions of these different types of inverters has been discussed in the schematic design section of this thesis. From the figure, three pairs of yellow M2 vertical wires can be seen. They are the control signal routing. In the full core decoder design, these control signals are connected to bondpads for the ability to tune them. They run vertically all the height of the core decoder. They are also horizontally connected at regular distances in a power grid-like structure. The leftmost yellow wire pair is the control signals PTUNEDELAY and NTUNEDELAY, which are used to insert additional delay inside the VN to match the delay another de-embedding input is going through inside CN. The middle yellow wire pair is the control signals CSPTUNE and CSNTUNE. They control the current-supply from the current-source inverter. If this inverter supplies high current, the decoder will decode faster. The rightmost pair of yellow wires are the switches START and STARTBAR of the TG switch. This switch, when turned on, connects the current-source inverter output to the shielded VN wire. This TG switch effectively starts the decoder. When the DAC output loads the VN with appropriate analog voltages corresponding to the LLR data, the VN states stays at their voltages since no charge is supplied to or drawn from them. Although the de-embedding is still performed, it is only when the TG switch is turned on that the decoder starts changing VN trajectory towards successful decoding.

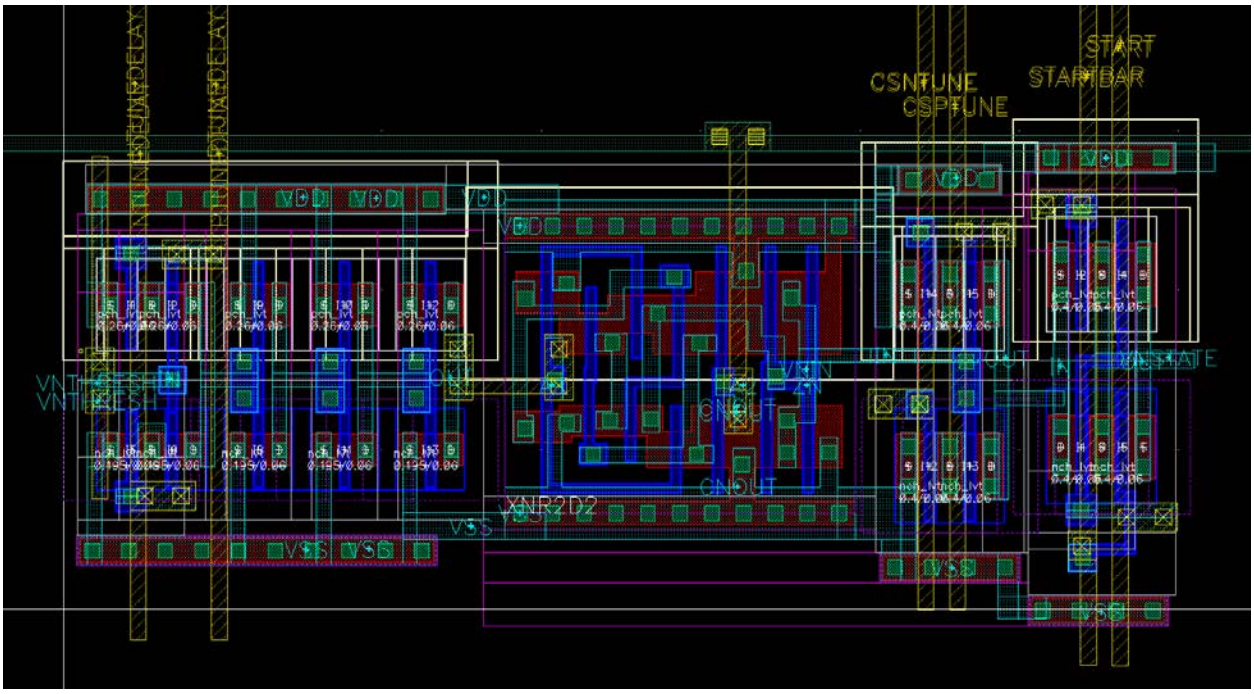


Figure 5.9: VN block including threshold inverter pair, delay insertion inverter pair, XOR gate, current-source inverter, and a TG switch.

Except for the core decoder, all other circuits placed for the testing purpose, which will be discussed in the following sections, are fairly densely packed maintaining minimum TSMC design rules for metal width and spacing. Due to the sparse nature of the LDPC description matrix, even after employing the compression strategy, there is still a good amount of blank space. The use of automatic place and route tools could have made the design of the core decoder much more compact with a very high area utilization factor. In this design, Cadence SKILL code has been used to place different layout blocks in the core decoder at the appropriate coordinates. The wiring connections between them have also been automated using SKILL code. Earlier in this section, the importance of a regular compression technique for this sort of automation has been discussed. Therefore, in this design, a fully-compacted placement of layout modules inside the core decoder were not possible. The core decoder takes an area of 0.77 x 1.85 mm^2 = 1.4245 mm^2. Now, calculating the area of each individual block used in the decoder, multiplying them with their respective quantity, and then adding these areas gives a total area of 0.21 mm^2.

Therefore, the area utilization factor is 0.21/1.4245 = 14.74%. Table 5.1 enlists different areas taken by different sub-blocks of the core decoder.

Table 5.1

Area taken by different sub-blocks of the core decoder

Name of sub-block	Quantity of the sub-block	Area taken by each sub-block (μm^2)	Total area taken by all similar sub-blocks (μm^2)
Threshold inverters	273 x 17	5.3 x 1.5	36,896
De-embedding block (delay-insertion inverter, de-embedding XOR, current-source inverter, TG switch)	273 x 17	8.5 x 3.2	126,235
XOR (type I)	273 x 17	3 x 2.4	33,415
XOR (type II) located at higher level of the CN XOR tree	68 x 8 +1 + 68 x 8 + 1	3 x 2.4	7,848
XOR (type III) driving CN output wire	68 x 4 +1	6.5 x 2.5	4,436

This calculation is a very crude estimate and also misleading. The reason behind is the fact that it does not include a tremendous amount of wirings used to connect blocks in the core decoder. The wirings are not only large in quantity, they have a minimum width set by TSMC 65nm DRC (design rules check) rules. Also, the metal-to-metal distance has to follow DRC rules. If these wiring constraints could be ignored, further horizontal compression just by placing the circuit layout blocks would have been possible. Furthermore, this calculation does not include the VN wire area, the parallel shielding wires, etc., since this metal layers pass above the metals used in designing the layout of other circuits. Finally, the contacts connecting VDD and GND wires in the circuits to the power-supplying grids of higher-level metals take some space at every row of the decoder design. With the selected approach to VN wire shielding and choice of metal layers for other signals, minimum metal-to-metal spacing has been used. Therefore, it is unlikely that further compression can be achieved, while still using a regular layout.

5.2 Memory Organization

The unit of the memory array is an SRAM cell. There are 10*7*273 SRAMs suitably and compactly arranged in the SRAM array. The SRAM cell is made of 6 transistors – 2 cross-coupled inverters and two access NMOS transistors. The layout of the SRAM cell is depicted in Figure 5.10. The vertical yellow M2 wire is the WORD line and the other two yellow lines are BIT and BITbar lines that will be connected to the horizontally running wires in the memory organization.

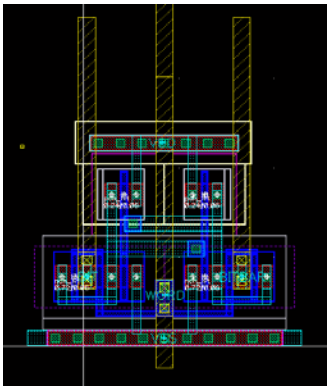


Figure 5.10: Layout of an SRAM cell.

In the SRAM array, there are 273 rows each containing 70 SRAM cells. Among 70 cells in a row, every 7 cells contain 7 bits corresponding to one quantized level. The vertical WORD wire selects which 7 cells in each row will be activated at a time for writing in or reading out.

5.3 Digital to Analog Converter (DAC)

The digital to analog converters (DACs) are used to precisely generate quantized voltage levels and feed them to the variable nodes inside the core decoder. There are 273 DACs in this chip who work parallel to each other and are able to simultaneously feed the decoder VNs. Each

DAC has six binary-weighted capacitors. The largest one is 256fF and the smallest one is 8fF. In addition to these six capacitors, which were implemented with MIMCAPs in layout, there are switches, as discussed in the schematic design. The size of the switches increases as the capacitors connected in series to them increase to allow more charging or discharging current to flow through. Figure 5.11 shows the layout of one DAC.

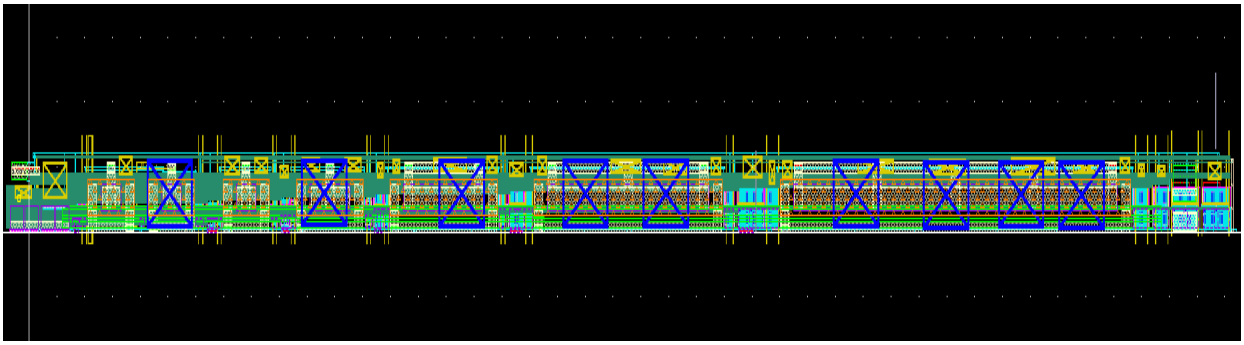


Figure 5.11: DAC Layout. The red boxes correspond to the MIMcaps. The switches are placed between the MIMcaps. Some dummy metals are also there to meet local density requirement across the chip.

It is noteworthy that the MIMcap height used here is the minimum height that could be set for a MIMcap. Furthermore, there is a design rule check (DRC) requirement for the distance between capacitor bottom plate metal M8 to the plate in the neighboring row. This height sets the bottleneck for the compression of the pitch distance from one row to the successive row. In other words, in the input shift register, there are 273 rows to deliver the 273 codewords of 7 bits length. After that comes the SRAM array with 273 rows, who again holds 10 codewords, each with 273*7 bits. Then, there are 273 DACs in one column. Finally, the core decoder has 273 horizontal variable node wires. For all these sections, the row to row distance has to be equal to make the design more straightforward to implement. So, when all the block placements and inter-block connections have been made with the Cadence SKILL code, the maximum compression in the row-to-row vertical pitch that could be achieved is 6.25 μm per row. This

row-to-row vertical pitch has also been matched with the VN-to-VN vertical pitch of the decoder. Therefore, the DACs and the VNs are joined when they are placed adjacent to each other.

The DAC is sensitive because it must generate precise output levels. So, the dummy filling at the last stage of the chip design was done carefully. The dummy metal fill below one capacitor was almost doubled in area for a capacitor of the double value. The DAC draws a lot of current, especially in the pre-charge phase of its operation. To address this, two bondpads, one on top and one on bottom, were reserved to supply voltage (and also current) to a vertical wide wire of higher level metal. Local demands are met with wide multiple contacts with the vertically running higher level metal.

5.4 Analog MUX

The analog MUX is the rightmost block in the chip. After the core decoder, come the output shift register. The analog MUXes comes just after the output shift register. Although the horizontally spread VN wires are thresholded to feed into the output shift register, the VN wires have to be directly fed into the analog MUX input. Each analog MUX is capable of having 16 inputs. Therefore, 18 MUXes are needed in total. Each MUX has its own output through the bondpad and also will be selected through 4 selector pins, which are common for all the MUXes. An analog MUX is composed of a transmission gate switch tree, and an OpAmp for buffering the output. The OpAmp, in addition to precise rendering of the analog VN level off-chip, also creates virtual isolation between the VN and off-chip when the TG switches are ON. The layout of one analog MUX is put in Figure 5.12.

5.5 Input Shift Register

The input shift register is made up of 7*273 DFFs – 7 DFFs in each row. Effectively, the input shift register is a large shift register that loads in the digital bits serially. Each DFF is

followed by a pair of simple inverters to insert a little delay in signal propagation so that the distributed clock signals can sample the input data bits at the input of the DFFs correctly. It will be too clumsy and inconvenient to see the whole input shift register. Instead, the layout of seven DFFs are placed in a row in Figure 5.13. This block, placed vertically with the SKILL code, will automatically connect the metals appropriately to form a large serial shift register.

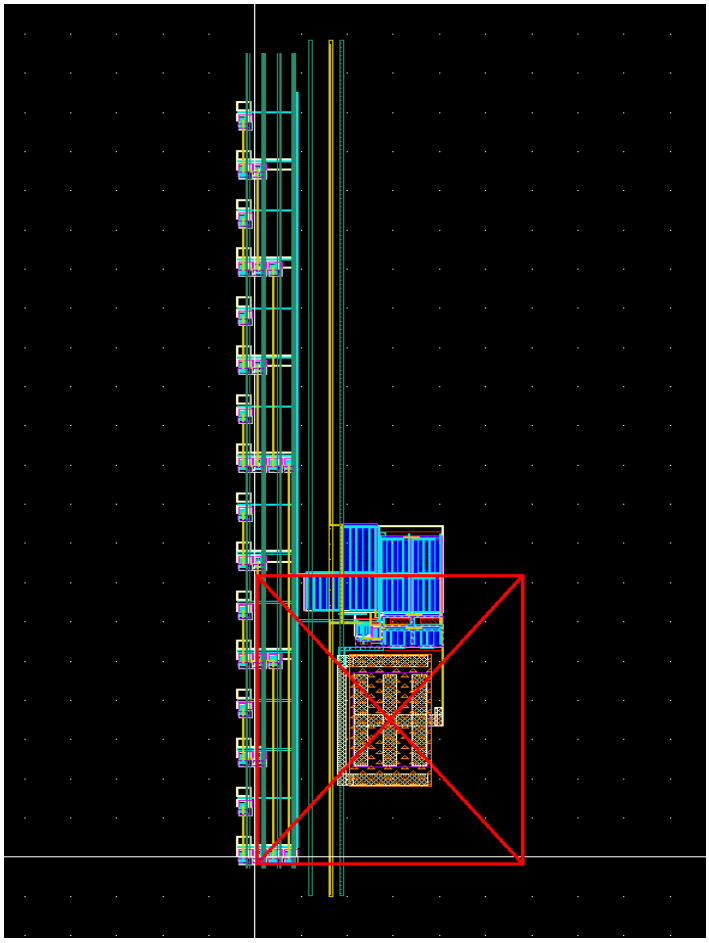


Figure 5.12: Layout of Analog MUX. The switches are on the left and the OpAmp is on the right.

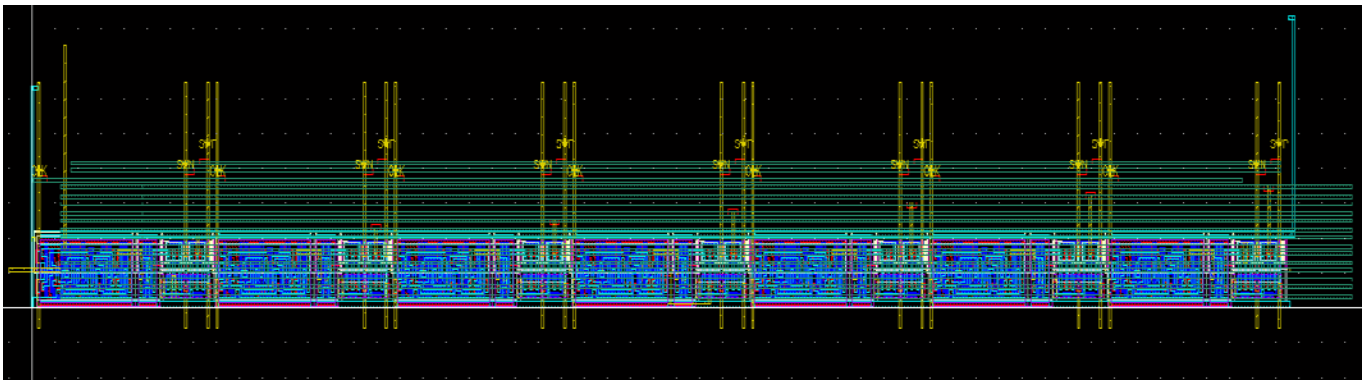


Figure 5.13: Layout of seven DFFs in each row of the input shift register. The Green horizontal M3 wires are bit lines to feed the SRAMs.

5.6 Output Shift Register

In the schematic design section, the structure of the output shift register has been discussed in detail. The output shift register is made up of parallel-in-parallel-out (PIPO) and parallel-in-serial-out (PISO) register. For convenience of layout placement using SKILL code, the DFF of PIPO and the DFF of PISO have first been integrated in one unit layout, called UNIT_PIPO_PISO. Then, with the help of the code, 273 units were placed vertically by maintaining the same row-to-row pitch. The layout of the unit has been designed such that the units placed vertically will be automatically connected to protruding metal wires from neighboring units to form the output shift register. Figure 5.14 illustrates the physical design of UNIT_PIPO_PISO.

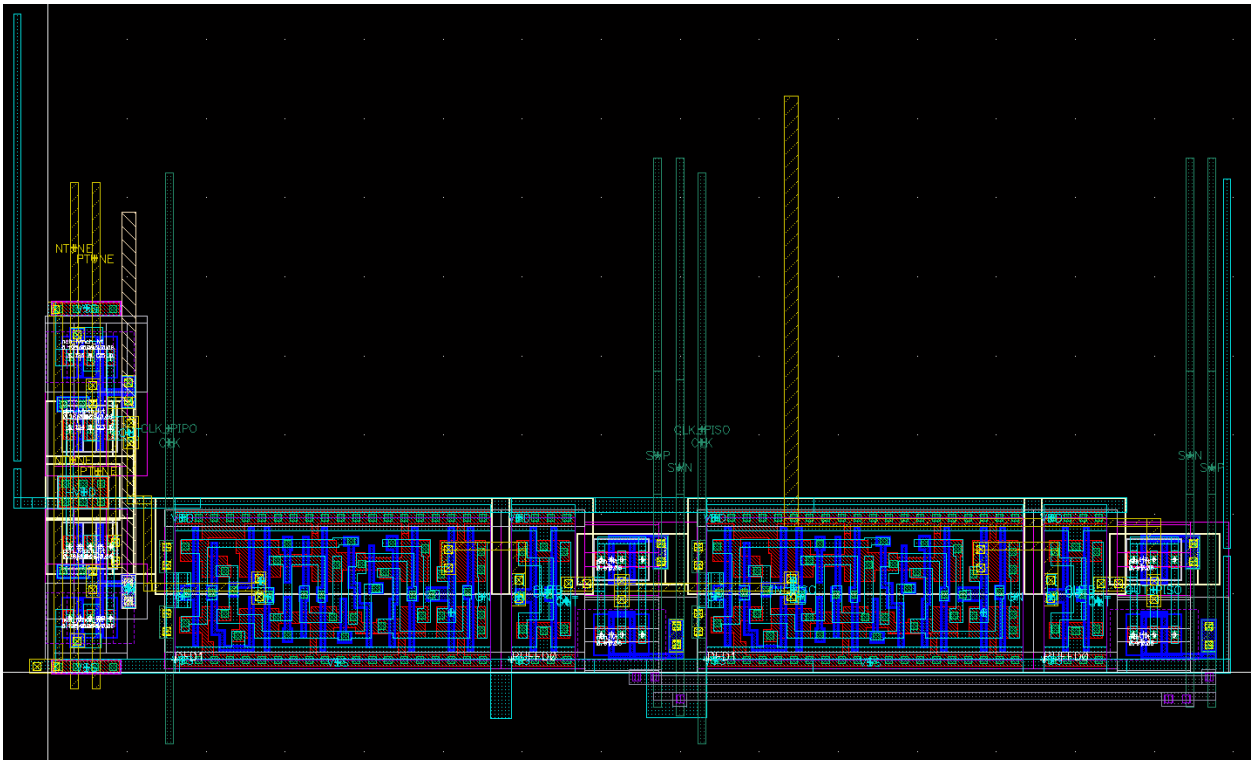


Figure 5.14: Layout of UNIT_PIPO_PISO. The vertically running protruding wires are visible in the figure.

5.7 The Final Design

When all the layout blocks are drawn, they need to be placed side by side. The care has been taken when building the unit blocks for each part so that when large blocks are placed side by side they will get automatically connected appropriately. One good example is the case of the connection between the input shift register and the SRAM array. As it has been shown in Figure 5.15, the seven DFFs in each row of the input shift register are also connected to the horizontally stretching bit lines of the SRAM. Therefore, the SRAM cluster placed by the side of the input shift register cluster will be correctly joined when the blocks are placed adjacent to each other. This connection between only a few higher level clusters has been done manually. Otherwise the

whole connections inside this chip has been done with the SKILL code, attached in Appendix B of this thesis.

Before the final physical design is presented, let us illustrate a simplified version of the block placement plan on the chip in Figure 5.16. Then in the Figure 5.16 next page, the final design will be shown that has been submitted for fabrication.

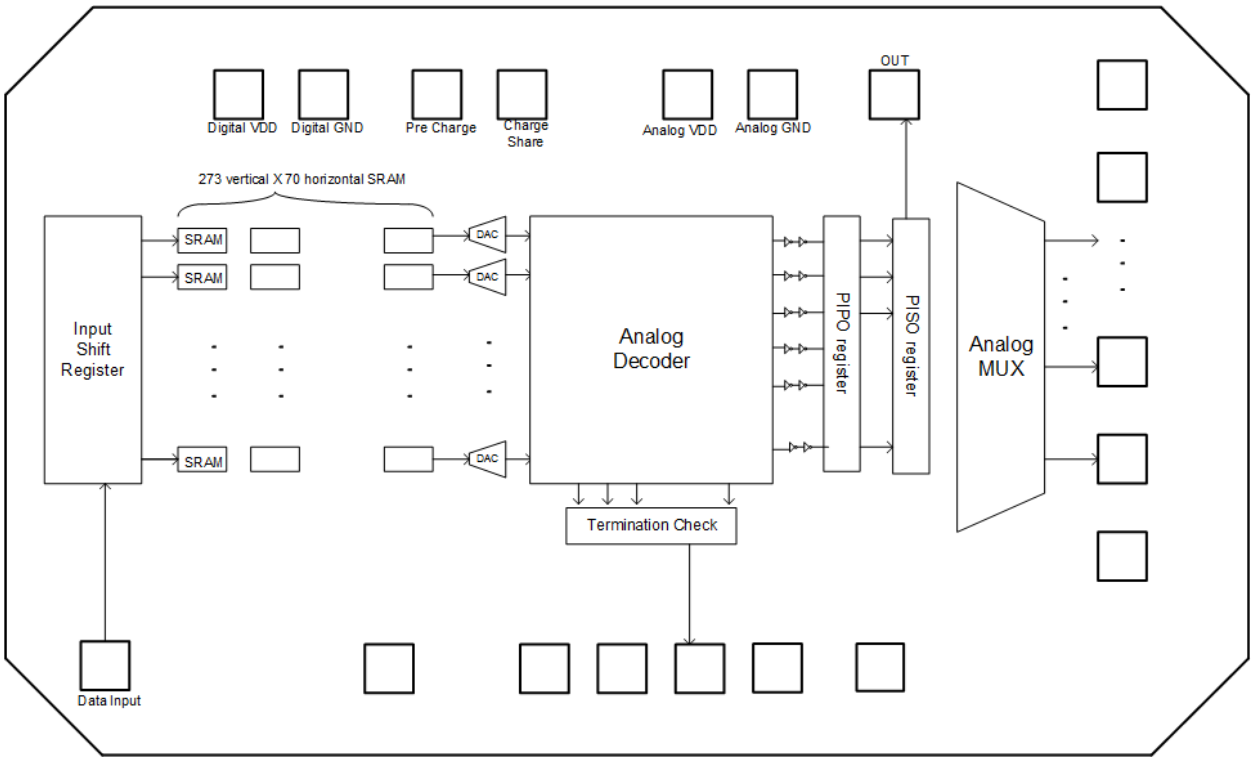


Figure 5.15: Simplified placement plan of higher level blocks inside the chip. A lot of bondpads are omitted for simplified view.

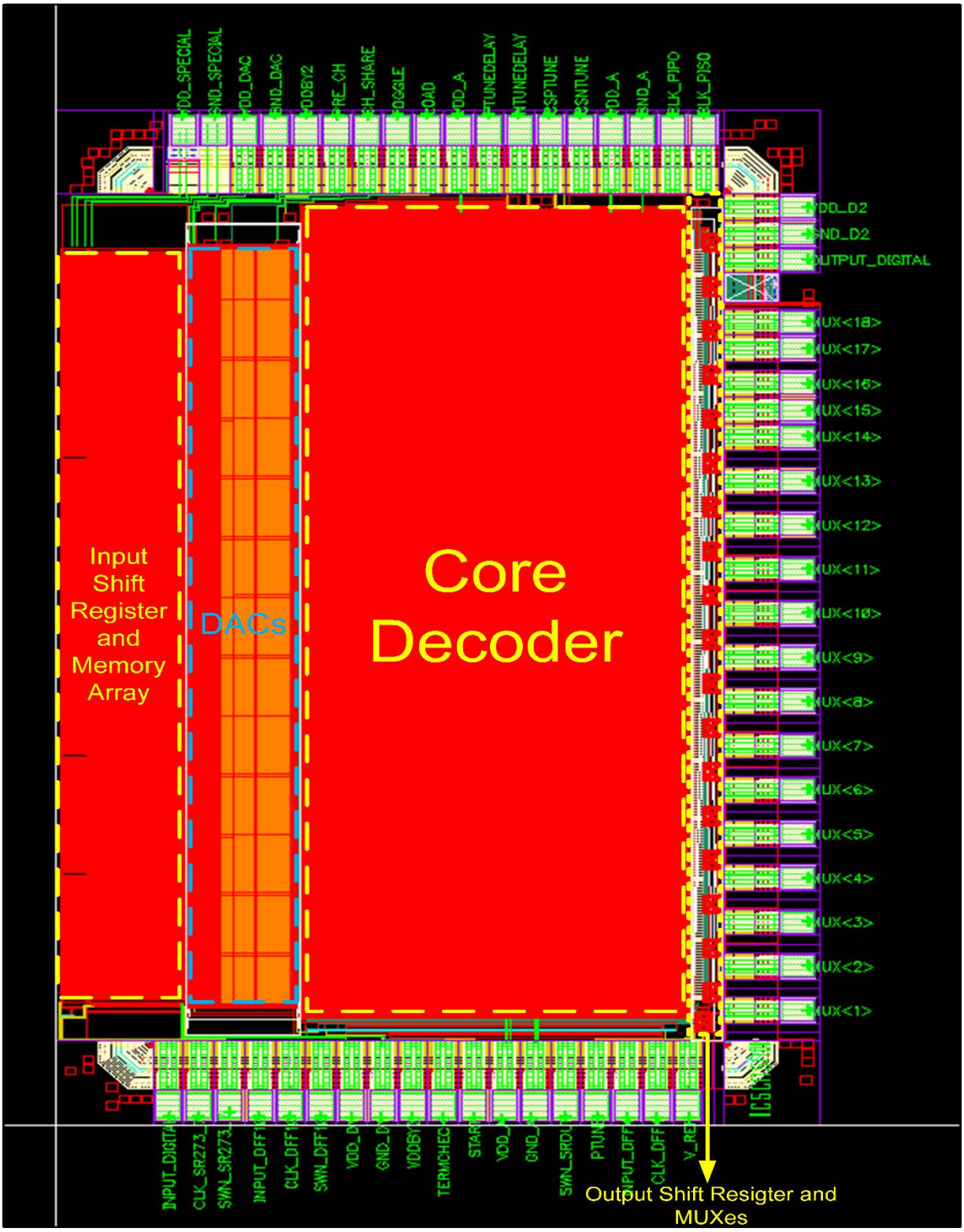


Figure 5.16: Final Layout design of the whole chip. Submitted for fabrication on Sept. 13, 2017.

Chapter 6

Test Plan

The prototype chip in TSMC 65nm has been submitted for fabrication to the Canadian Microelectronics Corporation (CMC) on September 13, 2017. The fabricated chip is supposed to come back around February 2018. After that, the measurements will be performed to obtain the power consumption, throughput, and the bit-error-rate of the implemented decoder. For now, the step by step workflow during the measuring process will be described.

Before the measurement plan is described, it is convenient to enlist the data and control signal I/O pins the IC has. Table 6.1 enlists the chip I/O pins.

Table 6.1

Data and control signal I/O pins

Name of the Pin	**Type**	**Description**
VDD_SPECIAL, GND_SPECIAL	Power	Supplies power to the ESDs, and the input shift register.
VDD_D1, GND_D1	Power	Supplies power to the memory array
VDD_DAC, GND_DAC	Power	Supplies power to the DAC.
VDD_A, GND_A	Power	There are two VDD_A and two GND_A pins. The pins with identical names are shorted by a power grid inside the chip. They supply power to the core analog decoder
VDD_D2, GND_D2	Power	Supplies power to the output shift register and the analog MUX.

INPUT_DIGITAL	Input	Used to input data bits into the chip.
OUTPUT_DIGITAL	Output	Used to bring decoded bits out of the chip.
MUX<1:18>	Output	18 pins connected to the output of analog MUXes. Buffers the analog voltages of the variable nodes.
CLK_SR273_7	Clock	Slow-speed clock for the input shift register.
CLK_DFF10	Clock	Slow-speed clock to clock the 10 DFF long shift register that selects WORD lines of the memory array.
CLK_DFF4	Clock	Slow-speed clock to run 4 DFF long shift register that selects the analog MUX input.
CLK_PIPO	Clock	Slow-speed clock for the parallel-input parallel-output (PIPO) shift register
CLK_PISO	Clock	Slow-speed clock for the parallel-input serial-output (PISO) shift register
SWN_SR273_7	Input	Turns on or off the TG switches connecting the DFF outputs inside the input shift register to the BIT lines of the SRAM array. It is important because if turned on during the SRAM reading out to BIT lines, DFF outputs in the shift register can conflict with SRAM outputs.
SWN_DFF10	Input	Controls the switches connecting the WORD line-selecting shift register outputs to the WORD lines of the memory array. This enables us to input any pattern inside the register first, and then connecting to the WORD lines to select a block of memory array.
SWN_SROUT	Input	SWN_SROUT controls the two different switches in an inverted way. SWN_SROUT either connects the PIPO register output to the PISO register input and disconnects the internal DFF connection inside PISO or connects a PISO DFF output to the following PISO DFF input and disconnects PIPO to PISO.
PRE_CH	Input	Controls the pre-charge phase of the DAC.

CH_SHARE	Input	Controls the charge-sharing stage of the DAC.
TOGGLE	Input	Enables the toggling mode of the DAC.
PTUNE	Input	Inputs the global tuning voltage for the threshold inverters.
PTUNEDELAY	Input	Supplies the tuning voltage for delay-insertion inverters.
NTUNEDELAY	Input	Supplies the tuning voltage for delay-insertion inverters.
CSPTUNE	Input	Inputs the global tuning voltage to tune the amount of current supplied by current-source inverters.
CSNTUNE	Input	Inputs the amount of global tuning voltage to tune the amount of current supplied by current-source inverters.
START	Input	Control signal for the switches connecting the current-source inverter output to the VN wire, and thus, effectively starting decoding.
LOAD	Input	Control signal turning on the switches connecting DAC output to the decoder VN memory. Interestingly, this pin has to keep turned on during the DAC operation and before the decoder actually starts. However, once the DAC outputs are loaded into the VN capacitance, the LOAD switches have to be turned off before turning on the START switches.
VDDBY2	Power	There are two VDDBY2 pins. Supplies good amount of charging current to the DAC at the VDD/2 voltage. Also, supplies VDD/2 voltage for the calibration circuitry.
V_REF	Power (Current Supply)	It draws out 20μA current for the current mirror used in OpAmps inside the calibration circuitry and the analog MUX.
TERMCHECK	Output	Indicates parity-check satisfaction, i.e., the decoding is complete.

It is important to note that the pins VDD_SPECIAL and GND_SPECIAL are shorted to the pins VDD_D1 and GND_D2 respectively. The VDD_DAC and GND_DAC, VDD_D2 and GND_D2, VDD_A and GND_A are not shorted to corresponding VDD or ground pins. On the board, they should be kept separate as well to bar digital noise to affect the sensitive analog voltages inside the core decoder. Therefore, there are four different ground areas in the whole chip as marked by non-maskable PSUB2 rectangles in the physical design for LVS matching.

Now that the pins connected to the bondpads of the chip are known, our plan to test this will be delineated. A printed circuit board will be designed to input and output the control and the data signals. The data bits to be input will be generated from a MATLAB code and saved into a memory card. Similarly, the decoded bits will also be saved in the memory card and read by MATLAB code to perform the bit-error-rate test. A microcontroller can feed the control signals when it is suitably coded in C.

The steps to perform the full testing of the chip is as follows.

1. First, set zero for all the control signals and clock signals. Turn on all the power supply and ground pins – both the digital and analog power. Also, supply DAC power and ground and VDDBY2 pin. Connect 20 μA current source at V_REF.

2. Supply PMOS tuning voltage at PTUNE pin and calibrate the threshold inverter on chip. Also, supply appropriate CSPTUNE and CSNTUNE for calibration of the current-source inverter. Provide appropriate voltages at PTUNEDELAY and NTUNEDELAY pins.

3. Turn off the connection between the input shift register and the BIT lines by putting SWN_SR273_7=0. Therefore, any digital data propagating through the input shift register will not reach the memory array.

4. Provide digital pulses of 50% duty cycle to the CLK_SR273_7 pin. Simultaneously, the binary data has to be fed in through the INPUT_DIGITAL pin. Figure 6.1 shows the waveforms. Once the shift register is fully loaded in with the 273*7 bits, stop sending any clock pulses and anymore data bits. Set CLK_SR273_7=0. At this point, all the bits necessary to construct 273 LLRs corresponding to a codeword have been loaded in.

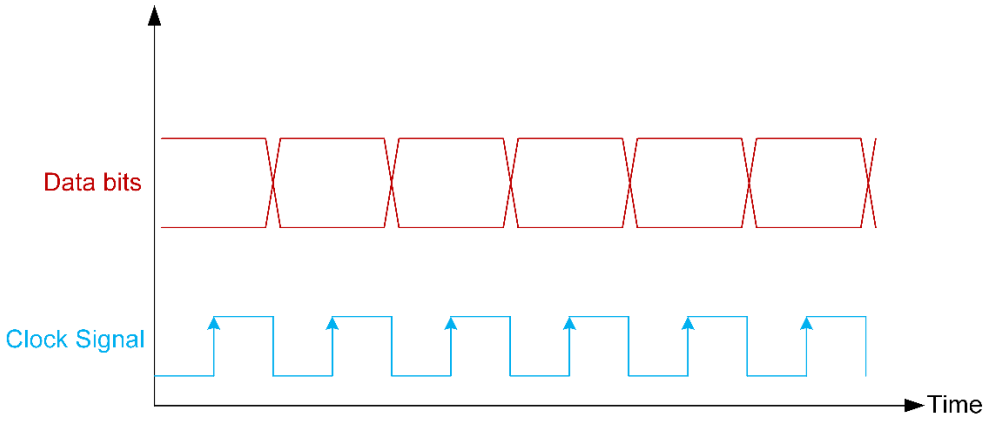


Figure 6.1: Clock signal is being fed to the CLK_SR273_7 pin and the data bits are being fed to the INPUT_DIGITAL pin simultaneously.

5. Connect the input shift register to the memory array by selecting SWN_SR273_7=1. This action activates the BIT lines of the memory array. Now, in synchronization with the clock, feed the WORD line-selecting shift register so that it outputs 10000 00000. Then, stop clocking. Therefore, all the SRAMs connected to WORD<1> will write in the data on the BIT lines.

6. To load the bits for the second codeword, repeat the steps 2 to 4, except for the fact that this time, the WORD line-selecting shift register has to output 01000 0000 so as to store data into the next section of the memory array. Continue repeating the steps until all the data for 10 codewords are loaded into the memory. Figure 6.2 shows the bits coming out of the WORD line selecting shift register that determine which memory block will be selected for writng different codewords in.

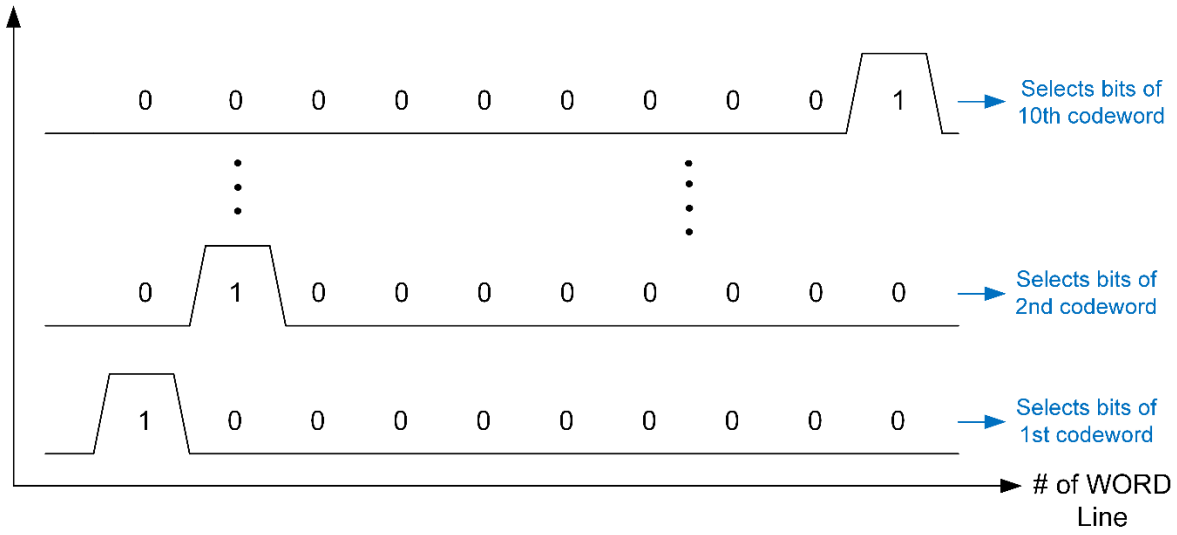


Figure 6.2: 10 bits coming out of the WORD line selecting shift register. The location of 1 in the 10 bit word determines which memory block will be selected for writng different codewords in.

7. Turn off the connection between the input shift register and the memory array. Set the TOGGLE mode for the DAC as either 0 or 1. Yield 10000 00000 at the output of the WORD line-selecting shift register. This activates the BIT lines with the bits of the first codeword.

8. Set LOAD=1 to include the variable node capacitance as a part of the DAC capacitance. Set PRE_CH=1 to precharge the DAC capacitors.

9. Reset PRE_CH=0 and set CH_SHARE=1. Charge sharing between DAC capacitors will take place and appropriate analog voltage will be formed at the variable node.

10. Turn LOAD=0. This disconnects the DAC output from the VN connection. Set START=1. Decoding starts. The end of decoding is indicated by TERMCHECK=1. Figure 6.3 summarizes the changes in all the control signals during one decoding period.

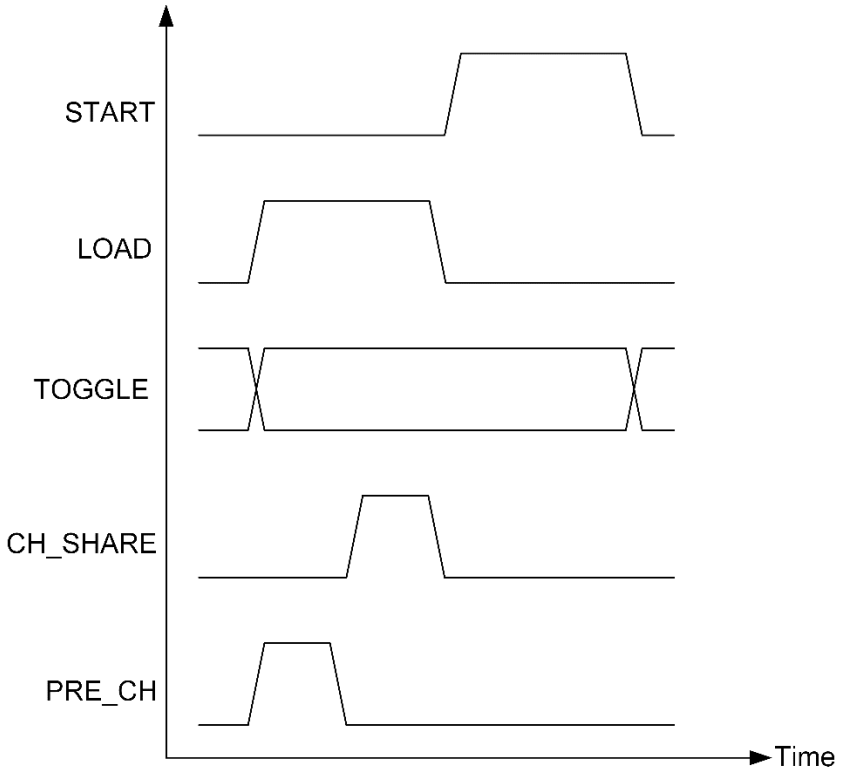


Figure 6.3: Control signals PRE_CH, CH_SHARE, TOGGLE, LOAD, and START during one decoding cycle.

11. Reset START=0, which stops decoding. Turn SWN_SROUT=0 to disconnect PIPO and PISO registers. Supply only one pulse to CLK_PIPO to write and hold the decoded digital data into the PIPO register.

12. Set SWN_SROUT=1. Clock CLK_PISO once to grab the data from PIPO to PISO. During this operation, the serial connection between the individual DFFs inside PISO is off. Now, reset SWN_SROUT=0. This disconnect PIPO to PISO and forms the serial DFF connections inside PISO.

13. Finally, clock CLK_PISO to serially fetch the decoded bits out through OUTPUT_DIGITAL pin.

14. To investigate the analog variable node voltage anytime, first, take the number of the VN wire to check. Suppose the number is ‘N’. Ceiling(N/16) will indicate which analog

MUX to look into. Binary conversion of the number mod(N,16)+1 gives the MUX selector pin-selecting shift register output. For example, to look into the 18th VN level, Ceiling(18/16) = 2nd MUX has to be selected. The shift register output has to be mod(18,16)+1 = 2+1 = 3. In binary, the shift register output will be 0011.

15. Clock in the necessary shift register output into the MUX selector pin-selecting shift register and look into the appropriate MUX output pin from the right side of the chip.

Chapter 6

Conclusion and Future Work

The thesis introduces an energy-efficient high-throughput mixed-signal low-density parity-check (LDPC) decoder in TSMC 65nm CMOS technology with 1.0V supply. The decoder implements the Modified Differential Decoding Binary Message Passing (MDD-BMP) decoding algorithm of LDPC codes in the mixed-signal domain. The analog impairments stemming out of the re-design of some digital parts in analog domain have been investigated for their comparative contribution to decoding performance. To minimize the effects of these impairments, on-chip calibration and global routing of the tuning signals have been proposed and implemented. Furthermore, the implemented decoder also offers modulation of the decoding speed by off-chip control signals. Enough on-chip memory has been placed to store 10 codewords and feed them to the decoder one after another through digital to analog converter. Placement of analog MUXes enables us to debug analog voltages of the variable nodes from off-chip. A termination-check circuit block indicates parity-check satisfaction as the end of decoding. Appropriate number of serial and parallel shift registers have been placed to input data bits from off-chip, to deliver decoded bits to off-chip, and to route control signals for sequential operation of the chip. Finally, the physical design of the chip in TSMC 65nm technology has been almost fully automated by Cadence SKILL code. Over 100 simulations taking into account the parasitic capacitance of long wires in physical design yields an average decoding speed of approximately 1.04 ns in moderate speed mode. Allocating 0.5ns for pre-charging VN capacitance and another 0.5ns for decoder initialization make a total 2.04ns decoding cycle time. This decoding cycle provides a throughput of 134 Gb/s. The calculated average energy per bit at this speed is 1.267 pJ/bit.

The work presented in this thesis has some scopes of future work, as mentioned below.

- Inside the variable node of the core analog decoder, a pair of delay-tuning inverters has been placed. While they can be tuned to insert different amount of delays in the de-embedding path, some sort of delay-calibration technique could be implemented to match the delays through de-embedding loop and VN-CN-VN loop. For example, pulsating

signals can be fed to a stand-alone VN-CN-VN path and to the delay-inserted de-embedding path. Comparing the outputs of these two path at the inputs of an XOR gate and low-pass filtering the XOR output gives zero when the two path delays are perfectly matched.

- The static current consumtion by the core decoder during the pre-charge phase is fairly high. This static power dissipation can be mainly attributed to the threshold inverters that are on during this time. Bringing change in the de-embedding loop to keep these inverters turned off before the decoding starts can be a good task to further reduce the energy per bit reported by this work.

- Although current compression technique for the sparse LDPC description matrix has been able to increase the effective layout area inside the decoder to a large extent, further compression especially in the vertical direction will be a very useful task. Investigation of the current compressed LDPC description matrix can lead to this path.

- The termination-check circuitry used in this design merely indicates the parity-check satisfaction by inquiring the check node outputs. However, the termination does not invoke another subsequent decoding process. Therefore, the decoding period needs to be pre-allocated from the empirical observation over a number of decoding time. The automation in triggering another decoding process at the end of the previous decoding by the termination-check circuitry can potentially increase the throughput of the decoder.

- The analog MUX designed for this work does a good job of indicating whether a VN is above or below threshold and also precisely rendering VN's analog voltage between around 300mV—800mV range. Design of a rail-to-rail analog MUX will be more useful in delivering precise values of all VN voltages, including those lying very close to the supply rails.

References

[1] C. E. Shannon, “A mathematical theory of communication,” Bell Syst. Tech. J., vol. 27, pp. 379−423, 623−656, 1948.

[2] D. J. C. MacKay and R. M. Neal, “Near Shannon Limit performance of low density parity check codes,” Electronics Letters, vol. 32, no. 18, pp. 1645−1646, 1996.

[3] T. Richardson and R. Urbanke, “The capacity of low-density parity-check codes under message-passing decoding,” IEEE Transactions on Information Theory, vol. 47, no. 2, pp. 599-618, Feb. 2001.

[4] Y. Kou, S. Lin, and M. Fossorier, “Low-density parity-check codes based on finite geometries: A rediscovery and new results,” IEEE transactions on Information Theory, vol. 47, no. 7, pp. 2711−2736, Nov. 2001.

[5] R. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA, USA: MIT Press, 1963.

[6] D. J. C. Mackay, “Good error-correcting codes based on very sparse matrices,” IEEE Transactions on Information Theory, vol. 45, no. 2, March 1999.

[7] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, “Factor graphs and the sum-product algorithm,” IEEE Transactions on Information Theory, vol. 47, pp. 498-519, Feb. 2001.

[8] S. Hemati and A. Banihashemi, “Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes,” IEEE Transactions on Communications, vol. 54, no. 1, Jan. 2006.

[9] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*. New York: Academic, 1970.

[10] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*. Philadelphia, PA: SIAM, 1995.

[11] H. Xiao, S. Tolouei, and A. H. Banihashemi, “Successive relaxation for decoding of LDPC codes,” 24th Queen’s Biennial Symposium on Communication, Kingston, Ontario, June 2008.

[12] N. Mobini, A. Banihashemi, and S. Hemati, “ A differential binary message passing LDPC decoder,” IEEE Transactions on Communication, vol. 57, no. 9, pp. 2518-2523, Sep. 2009.

[13] K. Cushon, S. Hemati, C. Leroux, S. Mannor, and W. J. Gross, "High-throughput energy-efficient LDPC decoders using differential binary message passing," IEEE Transactions on Signal Processing, vol. 62, no. 3, pp. 619-631, February 2014.

[14] R. M. Tanner, "A recursive approach to low complexity codes," IEEE Transactions on Information Theory, vol. IT-27, pp. 533-547, Sept. 1981.

[15] L. Bazzi, T. Richardson, and R. Urbanke, "Exact thresholds and optimal codes for the binary-symmetric channel and Gallager's decoding algorithm A," IEEE Transactions on Information Theory, vol. 50, no. 9, pp. 2010−2021, Sept. 2004.

[16] N. Miladinovic and M. Fossorier, "Improved bit-flipping decoding of low-density parity-check codes," IEEE Transactions on Information Theory, vol. 51, no. 4, pp. 1594-1606, Apr. 2005.

[17] P. Zarrinkhat and A. Banihashemi, "Threshold values and convergence properties of majority-based algorithms for decoding regular low-density parity-check codes," IEEE Transactions on Communication, vol. 52, no. 12, pp. 2087-2097, Dec. 2004.

[18] G. Cowan, K. Cushon, and W. Gross, "Mixed-signal implementation of differential decoding using binary message passing algorithms," IEEE 26th International Conference on Application-specific Systems, Architectures and Processors, Toronto, July 27-29, 2015.

[19] S. Hemati, A. H. Banihashemi, and C. Plett, "A 0.18-um CMOS analog min-sum iterative decoder for a (32,8) low-density parity-check (LDPC) code," IEEE Journal of Solid-State Circuits, vol. 41, no. 11, pp. 2531−2540, 2006.

[20] A. Darabiha, A. C. Carusone, and F. R. Kschischang, "Power reduction techniques for LDPC decoders," IEEE Journal of Solid-State Circuits, vol. 43, no. 8, pp. 1835−1845, August 2008.

[21] C. Chen, Y. Lin, H. Chang, and C. Lee, "A 2.37-Gb/s 284.8 mW rate-compatible (491,3,6) LDPC-CC decoder," IEEE Journal of Solid-State Circuit, vol. 47, no. 4, pp. 817−831, 2012.

[22] A. R. Abolfazli, Y. R. Shayan, G. E. R. Cowan, "750Mb/s 17pJ/b 90nm CMOS (120,75) TS-LDPC min-sum based analog decoder," IEEE Asian Solid-State Circuits Conference, Singapore, Nov. 11-13, 2013.

[23] C. Cheng, J. Yang, H. Lee, C. Yang, Y. Ueng, “A fully-parallel LDPC decoder architecture using probabilistic min-sum algorithm for high-throughput applications,” IEEE Transactions on Circuits and Systems I, vol. 61, no. 9, pp. 2738−2746, April 2014.

[24] Y. Toriyama and D. Markovic, “A 2.267 Gbps, 93.7 pJ/b non-binary LDPC decoder for storage applications,” IEEE Symposium on VLSI circuits, Kyoto, June 5-8, 2017.

Appendix A

MATLAB Codes

```
close all; clear all; clc;

path='C:\Users\Sanjoy\Desktop\python_idb_sim - modify - 4\matrices\';
H = importdata(strcat(path,'273_191_FG_H.dat'));
Hp=H';    % rows of H denote CNs, while columns of H denote VNs
H8=zeros(272,34);
H1=H8;

for i=1:273
   for j=1:8:272

         H8(i,ceil(j/8)) = sum(Hp(i,j:j+7));
         if H8(i,ceil(j/8))~=0
            H1(i,ceil(j/8))=1;
         else
            H1(i,ceil(j/8))=0;
         end
   end
end



H88=zeros(34,34);

for i=1:8:272
   for j=1:8:272

      H88(ceil(i/8),ceil(j/8)) = sum(sum(H(i:i+7,j:j+7)));

   end
end


H32=zeros(8,34);

for i=1:32:256
   for j=1:34
      H32(ceil(i/32),j) = sum(H1(i:i+32,j));
   end
end

max(max(H32));


%%%%%%%%%%%%
Htemp=zeros(13,34);
for j=1:34
   for i=1:20:260
      Htemp(ceil(i/20),j)=sum(H1(i:i+19,j));
   end
end
Htemp;


%%%%%%%%% Location Index Generation from Hp   %%%%%%%%%%%%
H_VN=[]; Htemp=[]; Htemp2=[]; Htemp3 = []; Hindx=[]; Hmult=[]; Hmult_indx=[];

for j=1:273
   for i=1:273
      if Hp(i,j)==1
         H_VN = [H_VN i];  %% row indx for column entries
```

```
        end
    end
end

for i=1:8*17:272*17
    Htemp=H_VN(i:i+8*17-1);    %% choosing bundle of eight CNs
    Htemp2=sort(unique(Htemp));
    Htemp3=Htemp;
    for j=1:length(Htemp)
        Hindx = [Hindx find(Htemp2==Htemp(j))];
    end

    for j=1:length(Htemp3)
        L=find(Htemp3==Htemp(j));
        if length(L)>1
            Htemp3(L(2))=1;
            Htemp3(L(1))=0;
        else Htemp3(L)=0;
        end
    end

Hmult_indx = [Hmult_indx Htemp3];   %% Double locations
end


%% Export CN locations to a .txt file
fileID= fopen('C:\Users\Sanjoy\Desktop\From_MATLAB.txt','w');
fileID2=fopen('C:\Users\Sanjoy\Desktop\From_MATLAB_2.txt','w');
myH=[];
for j=1:273
    for i=1:273
        if Hp(i,j)==1
            fprintf(fileID,'%d ',i);
            myH=[myH i];
        end
    end
end
fclose(fileID);

Hmult_indx=[Hmult_indx zeros(1,17)];
for i=1:length(Hmult_indx)
    fprintf(fileID2, '%d \n',Hmult_indx(i));
end
fclose(fileID2);
```

Appendix B

Cadence SKILL Codes

B.1 Cadence SKILL code which generates the core decoder schematic

```
; We are generating the core decoder schematic

cvid = dbOpenCellViewByType("LibSanjoy" "VN_CN" "schematic" "" "w")
cnid = dbOpenCellViewByType("LibSanjoy" "CN" "symbol" "" "r")
vnid = dbOpenCellViewByType("LibSanjoy" "VN" "symbol" "" "r")

pinMaster = dbOpenCellViewByType("basic" "ipin" "symbol")
;pairinv = dbOpenCellViewByType("LibSanjoy" "testPairInv1" "symbol" "" "r")
;deembed = dbOpenCellViewByType("LibSanjoy" "testDeembed1" "symbol" "" "r")
;vdcid = dbOpenCellViewByType("analogLib" "vdc" "symbol" "" "r")
;gndid = dbOpenCellViewByType("analogLib" "gnd" "symbol" "" "r")
;capid = dbOpenCellViewByType("LibSanjoy" "testcap" "schematic" "" "r")
;tgid = dbOpenCellViewByType("LibSanjoy" "TGswitch" "symbol" "" "r")


f= infile("SkillSanjoy/C.txt")
f2= infile("SkillSanjoy/From_MATLAB_2.txt")
f3= infile("SkillSanjoy/V.txt")

declare(Ind[17])   ; An array of length 17 has been declared
declare(Ind2[17])
declare(Ind3[17])

declare(matlab[17*273])
declare(matlab2[17*273])
declare(matlab3[17*273])

con=0
for(i 1 length(matlab)
    fscanf(f "%d" con) ; con is a temporary variable
    matlab[i-1] = con
)
close(f)

con=0
for(i 1 length(matlab2)
    fscanf(f2 "%d" con) ; con is a temporary variable
    matlab2[i-1] = con
)
close(f2)

con=0
for(i 1 length(matlab3)
    fscanf(f3 "%d" con) ; con is a temporary variable
    matlab3[i-1] = con
)
close(f3)
```

```
; First, let's load 17 entries from matlab[] into Ind[].


for(p (1-1)*17+1 273*17       ; length(matlab)
      if(mod(p 17)==1 then    ; if loop will end at the very end of all the codes in this file

               CN = (p-1)/17+1 ;ceiling(p/18)+1
               Eight = (p-1)/136+1
               if(mod(CN 8)==1 || mod(CN 8)==2 || mod(CN 8)==3 || mod(CN 8)==4 then
                        last4=0  ; last 4 CNs of the Eight
                        num = mod(CN 8)
                        E = mod(CN 8) ; used to place 0th, 138th, 139th, 140th at proper places
               else
                        last4=1
                               if(mod(CN 8)==0 then
                                 num = 8
                                 E = 4
                        else
                                 num = mod(CN 8)
                                 E = mod(CN 4)
                         )
               )
               r=0
               for(q p p+16
                     Ind[r]=matlab[q-1]
                     r=r+1
                     )
               r=0
               for(q p p+16
                     Ind2[r]=matlab2[q-1]
                     r=r+1
                     )
               r=0
               for(q p p+16
                     Ind3[r]=matlab3[q-1]
                     r=r+1
                     )


; ##########################################     CN box placement
##########################################

dy=2*(num-1)
dx=12*(Eight-1)

Xcnin=0.25
Xth=0.25
Xvss=1.3125
Xvdd=0.9375
Xcnout=2
Xptune=0.9375
Xntune=1.3125

Ycnin=0.125
Yth=-0.125
Yvss=-0.6250
Yvdd=-0.625
Ycnout=0
Yptune=0.6250
Yntune=0.6250


instid = dbCreateInst(cvid cnid "" dx:dy "R0")
schCreatePin(cvid pinMaster "VDD" "inputOutput" nil dx+Xvdd:dy+Yvdd "R90")
schCreatePin(cvid pinMaster "VSS" "inputOutput" nil dx+Xvss:dy+Yvss "R90")
schCreatePin(cvid pinMaster "PTUNE" "inputOutput" nil dx+Xptune:dy+Yptune "R270")
schCreatePin(cvid pinMaster "NTUNE" "inputOutput" nil dx+Xntune:dy+Yntune "R270")
wireID = schCreateWire(cvid "draw" "full" list(dx+Xcnout:dy+Ycnout dx+Xcnout+0.2:dy+Ycnout)
0.0625 0.0625 0)
wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xcnout+0.2:dy+Ycnout) strcat("CN<" sprintf(a
"%d" CN) ">") "lowerRight" "R0" "stick" 0.0625 nil)
```

```
;schCreatePin(cvid pinMaster strcat("CN<" sprintf(a "%d" CN) ">") "inputOutput" nil
dx+Xcnout:dy+Ycnout "R180")

;schCreatePin(cvid pinMaster strcat("CN" sprintf(a "%d" CN) "_VN<0:16>") "inputOutput" nil
dx+Xcnin:dy+Ycnin "R0")

;schCreatePin(cvid pinMaster "IN<0:16>" "inputOutput" nil dx+Xcnin:dy+Ycnin "R0")
;schCreatePin(cvid pinMaster "TH<0:16>" "inputOutput" nil dx+Xth:dy+Yth "R0")

sprintf(label1 "VN<%d" Ind[0])
for(i 2 17
      sprintf(labelsub ",%d" Ind[i-1])
      label1 = strcat(label1 labelsub)
   )
label1 = strcat(label1 ">")

wireID = schCreateWire(cvid "draw" "full" list(dx+Xcnin:dy+Ycnin dx+Xcnin-0.2:dy+Ycnin) 0.0625
0.0625 0)
wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xcnin-0.2:dy+Ycnin) label1 "lowerRight" "R180"
"stick" 0.025 nil)

label1 = strcat(sprintf(a "TH%d" Ind[0]) "_CN" sprintf(a "%d" CN) ",")
for(i 2 16
      label1 = strcat(label1 sprintf(a "TH%d" Ind[i-1]) "_CN" sprintf(a "%d" CN) ",")
   )
label1 = strcat(label1 sprintf(a "TH%d" Ind[16]) "_CN" sprintf(a "%d" CN))   ; 17th iter separate
coz no comma following it

wireID = schCreateWire(cvid "draw" "full" list(dx+Xth:dy+Yth dx+Xth-0.2:dy+Yth) 0.0625 0.0625 0)
wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xth-0.2:dy+Yth) label1 "lowerRight" "R180"
"stick" 0.025 nil)



; ##########################################     VN box placement
##########################################

dy=20+3*(num-1)
dx=12*(Eight-1)

Xstart=0
Xstartbar=0
Xload=0
Xloadbar=0

Xvss=1.5
Xvdd=0.8125

XptuneDelay=0.875
XntuneDelay=1
Xcsptune=1.3125
Xcsntune=1.4375
Xvn=2.3125
Xvnth=2.3125
Xcn=2.3125
Xvnchan=2.3125

Ystart=0
Ystartbar=-0.25
Yload=-0.5625
Yloadbar=-0.75

Yvss=-1.0625
Yvdd=-1.0625

YptuneDelay=0.8125
YntuneDelay=0.8125
Ycsptune=0.8125
Ycsntune=0.8125
Yvn=0.1250
```

```
Yvnth=-0.1250
Ycn=-0.3750
Yvnchan=0.375

instid = dbCreateInst(cvid vnid "" dx:dy "R0")
schCreatePin(cvid pinMaster "VDD" "inputOutput" nil dx+Xvdd:dy+Yvdd "R90")
schCreatePin(cvid pinMaster "VSS" "inputOutput" nil dx+Xvss:dy+Yvss "R90")
schCreatePin(cvid pinMaster "PTUNEDELAY" "inputOutput" nil dx+XptuneDelay:dy+YptuneDelay "R270")
schCreatePin(cvid pinMaster "NTUNEDELAY" "inputOutput" nil dx+XntuneDelay:dy+YntuneDelay "R270")
schCreatePin(cvid pinMaster "CSPTUNE" "inputOutput" nil dx+Xcsptune:dy+Ycsptune "R270")
schCreatePin(cvid pinMaster "CSNTUNE" "inputOutput" nil dx+Xcsntune:dy+Ycsntune "R270")
schCreatePin(cvid pinMaster "START" "input" nil dx+Xstart:dy+Ystart "R0")
schCreatePin(cvid pinMaster "STARTBAR" "input" nil dx+Xstartbar:dy+Ystartbar "R0")

schCreatePin(cvid pinMaster "LOAD" "input" nil dx+Xload:dy+Yload "R0")
schCreatePin(cvid pinMaster "LOADBAR" "input" nil dx+Xloadbar:dy+Yloadbar "R0")

;schCreatePin(cvid pinMaster strcat("VN<" sprintf(a "%d" CN) ">") "inputOutput" nil dx+Xvn:dy+Yvn
"R0")

wireID = schCreateWire(cvid "draw" "full" list(dx+Xvn:dy+Yvn dx+Xvn+0.2:dy+Yvn) 0.0625 0.0625 0)
wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xvn+0.2:dy+Yvn) strcat("VN<" sprintf(a "%d" CN)
">") "lowerRight" "R0" "stick" 0.0625 nil)

wireID = schCreateWire(cvid "draw" "full" list(dx+Xvnchan:dy+Yvnchan dx+Xvnchan+0.2:dy+Yvnchan)
0.0625 0.0625 0)
wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xvnchan+0.2:dy+Yvnchan) strcat("VNCHAN<"
sprintf(a "%d" CN) ">") "lowerRight" "R0" "stick" 0.0625 nil)

;wireID = schCreateWire(cvid "draw" "full" list(dx+Xchan:dy+Ychan dx+Xchan-0.2:dy+Ychan) 0.0625
0.0625 0)
;wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xchan-0.2:dy+Ychan) strcat("VNCHAN<" sprintf(a
"%d" CN) ">") "lowerRight" "R0" "stick" 0.0625 nil)

sprintf(label1 "CN<%d" Ind3[0])
for(i 2 17
      sprintf(labelsub ",%d" Ind3[i-1])
      label1 = strcat(label1 labelsub)
   )
label1 = strcat(label1 ">")
wireID = schCreateWire(cvid "draw" "full" list(dx+Xcn:dy+Ycn dx+Xcn+0.2:dy+Ycn) 0.0625 0.0625 0)
wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xcn+0.2:dy+Ycn) label1 "lowerRight" "R180"
"stick" 0.025 nil)



label1 = strcat(sprintf(a "TH%d" CN) "_CN" sprintf(a "%d" Ind3[0]) ",")
for(i 2 16
      label1 = strcat(label1 sprintf(a "TH%d" CN) "_CN" sprintf(a "%d" Ind3[i-1]) ",")
   )
label1 = strcat(label1 sprintf(a "TH%d" CN) "_CN" sprintf(a "%d" Ind3[16]))   ; 17th iter
separate coz no comma following it

wireID = schCreateWire(cvid "draw" "full" list(dx+Xvnth:dy+Yvnth dx+Xvnth+0.2:dy+Yvnth) 0.0625
0.0625 0)
wireLab = schCreateWireLabel(cvid car(wireID) (dx+Xvnth+0.2:dy+Yvnth) label1 "lowerRight" "R0"
"stick" 0.0625 nil)


)
)

schCreatePin(cvid pinMaster "VN<1:273>" "inputOutput" nil -1:0 "R0")
schCreatePin(cvid pinMaster "VNCHAN<1:273>" "inputOutput" nil -1:-0.25 "R0")
schCreatePin(cvid pinMaster "CN<1:273>" "inputOutput" nil -1:-0.5 "R0")
```

B.2 Cadence SKILL code which generates the core decoder layout

```
; We are generating the core decoder layout

cvid = dbOpenCellViewByType("LibSanjoy" "VN_CN" "layout" "" "w")
xorid = dbOpenCellViewByType("LibSanjoy" "XOR1" "layout" "" "r")
xor2id = dbOpenCellViewByType("LibSanjoy" "XOR2" "layout" "" "r")
xor_138 = dbOpenCellViewByType("LibSanjoy" "XOR1_L138" "layout" "" "r")
xor2_138 = dbOpenCellViewByType("LibSanjoy" "XOR2_L138" "layout" "" "r")
xor_274 = dbOpenCellViewByType("LibSanjoy" "XOR1_L274" "layout" "" "r")
xor2_274 = dbOpenCellViewByType("LibSanjoy" "XOR2_L274" "layout" "" "r")
xor_139 = dbOpenCellViewByType("LibSanjoy" "XOR1_L139" "layout" "" "r")
xor2_139 = dbOpenCellViewByType("LibSanjoy" "XOR2_L139" "layout" "" "r")
xor_17 = dbOpenCellViewByType("LibSanjoy" "XOR1_17" "layout" "" "r")
xor2_17 = dbOpenCellViewByType("LibSanjoy" "XOR2_17" "layout" "" "r")

m3id = dbOpenCellViewByType("LibSanjoy" "M4" "layout" "" "r")
m3sid = dbOpenCellViewByType("LibSanjoy" "M4_SHORT" "layout" "" "r")
m3vsid = dbOpenCellViewByType("LibSanjoy" "M4_VERYSHORT" "layout" "" "r")
m2id = dbOpenCellViewByType("LibSanjoy" "M2" "layout" "" "r")
ptunebar = dbOpenCellViewByType("LibSanjoy" "PTUNEBAR" "layout" "" "r")

viaid = dbOpenCellViewByType("LibSanjoy" "viaM3M4" "layout" "" "r")
viaid2 = dbOpenCellViewByType("LibSanjoy" "viaM3M4_2" "layout" "" "r")
m2m3 = dbOpenCellViewByType("LibSanjoy" "viaM2M3" "layout" "" "r")
pairinv1 = dbOpenCellViewByType("LibSanjoy" "INV_TH1" "layout" "" "r")
pairinv2 = dbOpenCellViewByType("LibSanjoy" "INV_TH2" "layout" "" "r")
;ThPairDelay = dbOpenCellViewByType("LibSanjoy" "THPAIRDELAY" "layout" "" "r")
;deembed1 = dbOpenCellViewByType("LibSanjoy" "DEEMBED1" "layout" "" "r")
;deembed2 = dbOpenCellViewByType("LibSanjoy" "DEEMBED2" "layout" "" "r")
vn = dbOpenCellViewByType("LibSanjoy" "VN" "layout" "" "r")
vn2 = dbOpenCellViewByType("LibSanjoy" "VN2" "layout" "" "r")
sw = dbOpenCellViewByType("LibSanjoy" "SWITCH_LOAD" "layout" "" "r")
sw2 = dbOpenCellViewByType("LibSanjoy" "SWITCH_LOAD2" "layout" "" "r")
lshape = dbOpenCellViewByType("LibSanjoy" "L_SHAPE" "layout" "" "r")
lshape2 = dbOpenCellViewByType("LibSanjoy" "L_SHAPE2" "layout" "" "r")

vnperlane1 = dbOpenCellViewByType("LibSanjoy" "VN_PER_LANE1" "layout" "" "r")
vnperlane2 = dbOpenCellViewByType("LibSanjoy" "VN_PER_LANE2" "layout" "" "r")

m7 = dbOpenCellViewByType("LibSanjoy" "M7" "layout" "" "r")
shield_m5 = dbOpenCellViewByType("LibSanjoy" "SHIELD_M5" "layout" "" "r")

dpo = dbOpenCellViewByType("LibSanjoy" "VN_CN_DOD_DPO" "layout" "" "r")

L137= 137
L138= 138
L139= 139
L274 = 274+12
L275 = 275+12

f= infile("SkillSanjoy/From_MATLAB.txt")
f2= infile("SkillSanjoy/From_MATLAB_2.txt")
declare(Ind[17])
declare(Ind2[17])
declare(Pos[17])
declare(matlab[17*273])
declare(matlab2[17*273])
con=0
for(i 1 length(matlab)
    fscanf(f "%d" con) ; con is a temporary variable
    matlab[i-1] = con
)
close(f)

con=0
for(i 1 length(matlab2)
    fscanf(f2 "%d" con)
    matlab2[i-1] = con
```

```
)
close(f2)

;###  First, let's load 17 entries from matlab[] into Ind[]. ###

for(p (1-1)*17+1 273*17                              ;length(matlab)
      if(mod(p 17)==1 then    ; if loop will end at the very end of all the codes in this file
               CN = (p-1)/17+1   ;ceiling(p/18)+1
               Eight = (p-1)/136+1
               Four = (p-1)/68+1
               if(mod(CN 8)==1 || mod(CN 8)==2 || mod(CN 8)==3 || mod(CN 8)==4 then
                        last4=0
                        num = mod(CN 8)
                        E = mod(CN 8) ; used to place 0th, 138th, 139th, 140th at proper places
               else
                        last4=1
                        if(mod(CN 8)==0 then
                                 num = 8
                                 E = 4
                        else
                                 num = mod(CN 8)
                                 E = mod(CN 4)
                         )
               )
               r=0
               for(q p p+16
                     Ind[r]=matlab[q-1]
                     r=r+1
                     )
               r=0
               for(q p p+16
                     Ind2[r]=matlab2[q-1]
                     r=r+1
                     )

; ### put real positions in Pos ###
for(i 0 16
      if(Ind[i]>136 then
        Pos[i]=Ind[i]+12
      else
        Pos[i]=Ind[i]
      )
)
Ind=Pos
; First let's place the XORs   ################################################

yoff_in1=0.995
yoff_in2=0.435
yoff_out=1.555

x=0
y=0
dx2=11   ;12
dy2=0.2

dx_xor= 22*(Eight-1)           ;24*(Eight-1)   ;39*(Eight-1)  ; 13*(Eight-1)
dx_conn= 1.44*(num-1)     ; 2*(CN-1) ; X distance between each CN connections
dx= 22*(CN-1)       ;24*(CN-1)    ; 39*(CN-1)
dx_four = 11*(Four-1)     ; 12*(Four-1)   ; 1.44*4 = 5.76, used 7.00 willingly ; if dx_xor leaps
by 26, this will leap by half of that
dx_4conn = 1.44*(E-1)
dy= 6.25        ; 3.25
lenx=0
leny=0

for(i 2 16
      if(i!=9 then
             leny=Ind[i-1]*dy
             if(Ind2[i-1]==1 then
                   dbCreateInst(cvid xor2id "" dx_xor-1:leny "R0")   ; Omit -1 if distance from
1st to 2nd column is 12.
```

```
             else
                   dbCreateInst(cvid xorid "" dx_xor:leny "R0")
                  )
             )
)

leny=Ind[17-1]*dy
     if(Ind2[17-1]==1 then
           dbCreateInst(cvid xor2_17 "" dx_xor-1:leny "R0")
     else
           dbCreateInst(cvid xor_17 "" dx_xor:leny "R0")
        )



; extra 5 XORs at 274+12th(blue), 275+12th(purple), 137th(pink), 139th(pink), 138th(Green) place
for(i L274 L275
      leny=i*dy+(E-1)*2*dy
      if(last4==0 then
                dbCreateInst(cvid xor_274 "" dx_xor:leny "R0")
      else
                dbCreateInst(cvid xor2_274 "" dx_xor:leny "R0")
                )
)
for(i L137 L139
       leny=i*dy+(E-1)*3*dy
       if(i==L139 then
                if(last4==0 then
                      dbCreateInst(cvid xor_139 "" dx_xor:leny "R0")
                else
                      dbCreateInst(cvid xor2_139 "" dx_xor:leny "R0")
                 )
       )
       if(i==L137 then
                if(last4==0 then
                      dbCreateInst(cvid xor_274 "" dx_xor:leny "R0")
                else
                      dbCreateInst(cvid xor2_274 "" dx_xor:leny "R0")
                 )
       )

       if(i==L138 then
                if(last4==0 then
                      dbCreateInst(cvid xor_138 "" dx_xor:leny "R0")
                else
                      dbCreateInst(cvid xor2_138 "" dx_xor:leny "R0")
                 )
       )
)


; Now, let's place the M3 wires for blue xor inputs   ###################################

for(i 1 17
      if(i==2 || i==6 || i==10 || i==14 then
      leny = Ind[i-1]*dy
      dbCreateInst(cvid viaid "" dx_4conn+dx_four:yoff_out+(dy2+0.08)*Ind2[i-1]+leny "R0")
      leny = Ind[i]*dy
      dbCreateInst(cvid viaid "" dx_4conn+dx_four:yoff_in2+(dy2+0.08)*Ind2[i]+leny "R0")

      for(j Ind[i-1] Ind[i]-1
            leny=j*dy
            if(j!=Ind[i]-1 then
               dbCreateInst(cvid m3id "" dx_4conn+dx_four:yoff_out+leny-0.0 "R0")  ; -1 so that
2nd in eight XORs can also catch them
            else
                dbCreateInst(cvid m3sid "" dx_4conn+dx_four:yoff_out+leny-0.0 "R0")  ; NO Y shift
for metal bars in case of 2nds in Eight
             )
       )
```

```
       )
      if(i==17 then
      leny = Ind[i-1]*dy
      dbCreateInst(cvid viaid "" dx_4conn+dx_four:yoff_out+(dy2+0.08)*Ind2[i-1]+leny "R0")
      leny = L274*dy+(E-1)*2*dy
      dbCreateInst(cvid viaid "" dx_4conn+dx_four:yoff_in2+(dy2+0.08)*last4+leny "R0")
      for(j Ind[i-1] L274-1+(E-1)*2
            leny=j*dy
            if(j!=L274-1+(E-1)*2 then
                dbCreateInst(cvid m3id "" dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
            else
                dbCreateInst(cvid m3sid "" dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
            )
      )
      )
)

for(i 1 17
      if(i==4 || i==8 || i==12 || i==16 then
      leny = Ind[i-1]*dy
      dbCreateInst(cvid viaid "" dx_4conn+dx_four:yoff_out+(dy2+0.08)*Ind2[i-1]+leny "R0")
      leny = Ind[i-2]*dy
      dbCreateInst(cvid viaid "" dx_4conn+dx_four:yoff_in1+(dy2+0.08)*Ind2[i-2]+leny "R0")
                  for(j Ind[i-2] Ind[i-1]-1
                  leny=j*dy
                  dbCreateInst(cvid m3id "" x+dx_4conn+lenx+dx_four:yoff_in1+leny-0.0 "R0") ; -
0.5 so that 2nd in eight XORs can also catch them
                  )
             )
)

; Now, draw 1 blue lines ##############################################

;x=0.4
x=0.24

for(i 1 17
      if(i==3 || i==11 then
         leny = Ind[i-1]*dy
         dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*Ind2[i-1]+leny "R0")
         leny = Ind[i+1]*dy
         dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in2+(dy2+0.08)*Ind2[i+1]+leny "R0")

              for(j Ind[i-1] Ind[i+1]-1
                           leny=j*dy
                           if(j!=Ind[i+1]-1 then
                           dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                           else
                           dbCreateInst(cvid m3sid "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                           )
               )
      )

         leny = L274*dy+(E-1)*2*dy
         dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*last4+leny "R0")
         leny = L275*dy+(E-1)*2*dy
         dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in2+(dy2+0.08)*last4+leny "R0")

        leny=L274*dy+(E-1)*2*dy
         dbCreateInst(cvid m3sid "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
)

for(i 1 17
      if(i==7 || i==15 then
      leny = Ind[i-1]*dy
      dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*Ind2[i-1]+leny "R0")

      leny = Ind[i-3]*dy
      dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in1+(dy2+0.08)*Ind2[i-3]+leny "R0")
                  for(j Ind[i-3] Ind[i-1]-1
                  leny=j*dy
```

```
                    dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_in1+leny-0.0 "R0")
                    )
               )
)



; We draw TWO purple lines  ######################################################

;x=0.8
x=0.24*2

for(i 1 17
      if(i==13 then
         leny = Ind[i-1]*dy
         dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*Ind2[i-1]+leny "R0")
         leny = L139*dy+(E-1)*3*dy
         dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in2+(dy2+0.08)*last4+leny "R0")
        if(Ind[i-1]<L139+(E-1)*3 then
             for(j Ind[i-1] L139-1+(E-1)*3
                           leny=j*dy
                           if(j!=L275-1+(E-1)*3 then
                           dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                           else
                           dbCreateInst(cvid m3sid "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                           )
          )
        else
              for(j L139+(E-1)*3 Ind[i-1]    ; -1 is omitted otherwise bar cannot reach 13th XOR
OUT
                           leny=j*dy
                           ;if(j!=Ind[i-1]-1 then
                           dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_in2+dy2*last4+leny
"R0")
                           ;else
                           ;dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_in2+dy2*last4+leny
"R0")
                           ;)
          )
        )
      )

)


for(i 1 17
      if(i==5 then
      leny = Ind[i-1]*dy
      dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*Ind2[i-1]+leny "R0")  ;
last4 in y-direction is only for L137-139s & L274-275s
      leny = L137*dy+(E-1)*3*dy
      dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in2+(dy2+0.08)*last4+leny "R0")
      if(Ind[i-1]<L137+3*(E-1) then
                  for(j Ind[i-1] L137+3*(E-1)-1
                  leny=j*dy
                  if(j!=L137+3*(E-1)-1 then
                      dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                  else
                      dbCreateInst(cvid m3sid "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                  )
                  )
       else
                  for(j L137+3*(E-1) Ind[i-1]
                  leny=j*dy
                  ;if(j!=Ind[i-1]-1 then
                      dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                  ;else
                  ;   dbCreateInst(cvid m3sid "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")
                  ;)
                  )
```

```
       )
       )
)

;x=1.2
x=0.24*3

leny=L139*dy+3*(E-1)*dy
dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in1+(dy2+0.08)*last4+leny "R0")
leny=L275*dy+2*(E-1)*dy
dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*last4+leny "R0")

for(j L139+3*(E-1) L275+2*(E-1)-1
      leny=j*dy
      dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_in1+leny-0.0 "R0")
)


; Draw the 1 pink line   ######################################################

;x=1.6
x=0.24*4

leny = L138*dy+(E-1)*3*dy
dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in2+(dy2+0.08)*last4+leny "R0")
leny = L137*dy+(E-1)*3*dy
dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*last4+leny "R0")
dbCreateInst(cvid m3sid "" x+dx_4conn+dx_four:yoff_out+leny-0.0 "R0")

leny = L139*dy+(E-1)*3*dy
dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*last4+leny "R0")
leny = L138*dy+(E-1)*3*dy
dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_in1+(dy2+0.08)*last4+leny "R0")
dbCreateInst(cvid m3id "" x+dx_4conn+dx_four:yoff_in1+leny-0.0 "R0")  ; 0.2 was there in place of
0.0



; Draw the 1 Green line (Vertical CN output)   ###################################


x=0.24*5

leny=L138*dy+3*(E-1)*dy
dbCreateInst(cvid viaid "" x+dx_4conn+dx_four:yoff_out+(dy2+0.08)*last4+leny "R0")

for(j -1 293   ; for(j Ind[0]-1 Ind[16]
       leny=j*dy
       dbCreateInst(cvid m3id "" x+dx_4conn+lenx+dx_four:yoff_out+leny-1 "R0")
)
leny=L138*dy+3*(E-1)*dy






;#############################################################################
;#################     Here We place the Labels to match LVS     ###########
;#############################################################################

Yin1 = 1.05
Yin2 = 0.485
Yout = 1.61
Yvdd = 1.97
Yvss = 0.16

X = 1.00    ;X doesn't matter while placing the labels
DX2 = 11      ;12     ;for the vdd, vss, sub labels on XOR2
```

```
for(i 2 17
     if(i!=9 then
             leny = Ind[i-1]*dy
             dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*Ind2[i-1]:Yvdd+leny "VDD"
"centerLeft" "R0" "roman" 0.25)
             dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*Ind2[i-1]:Yvss+leny "VSS"
"centerLeft" "R0" "roman" 0.25)
     )
)

for(i 2 17
      if(i==2 || i==4 || i==6 || i==8 || i==10 || i==12 || i==14 || i==16 then
             leny = Ind[i-1]*dy
;            dbCreateLabel(cvid list("M3" "pin") X+DX2*Ind2[i-1]+dx_xor:Yin1+dy2*Ind2[i-1]+leny
strcat("CN" sprintf(a "%d" CN) "_IN<" sprintf(a "%d" (i-1)) ">") "centerLeft" "R0" "roman" 0.25)
;            dbCreateLabel(cvid list("M3" "pin") X+DX2*Ind2[i-1]+dx_xor:Yin2+dy2*Ind2[i-1]+leny
strcat("CN" sprintf(a "%d" CN) "_IN<" sprintf(a "%d" (i-2)) ">") "centerLeft" "R0" "roman" 0.25)
      )

      if(i==17 then
             leny = Ind[i-1]*dy
;              dbCreateLabel(cvid list("M3" "pin") X+DX2*Ind2[i-1]+dx_xor:Yin1+dy2*Ind2[i-1]+leny
"VSS" "centerLeft" "R0" "roman" 0.25)
;            dbCreateLabel(cvid list("M3" "pin") X+DX2*Ind2[i-1]+dx_xor:Yin2+dy2*Ind2[i-1]+leny
strcat("CN" sprintf(a "%d" CN) "_IN<" sprintf(a "%d" (i-1)) ">") "centerLeft" "R0" "roman" 0.25)
      )
)


              ; VDD, VSS, sub! etc labels do not need Y shift, but input or output labels do
need Y shift ... both need X shift anyway
leny = L274*dy+(E-1)*2*dy
;dbCreateLabel(cvid list("M3" "pin") X+DX2*last4+dx_xor:Yin1+dy2*last4+leny "VSS" "centerLeft"
"R0" "roman" 0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvdd+leny "VDD" "centerLeft" "R0" "roman"
0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvss+leny "VSS" "centerLeft" "R0" "roman"
0.25)


leny = L275*dy+(E-1)*2*dy
;dbCreateLabel(cvid list("M3" "pin") X+DX2*last4+dx_xor:Yin1+dy2*last4+leny "VSS" "centerLeft"
"R0" "roman" 0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvdd+leny "VDD" "centerLeft" "R0" "roman"
0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvss+leny "VSS" "centerLeft" "R0" "roman"
0.25)


leny = L137*dy+(E-1)*3*dy
;dbCreateLabel(cvid list("M3" "pin") X+DX2*last4+dx_xor:Yin1+dy2*last4+leny "VSS"  "centerLeft"
"R0" "roman" 0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvdd+leny "VDD" "centerLeft" "R0" "roman"
0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvss+leny "VSS" "centerLeft" "R0" "roman"
0.25)


leny = L138*dy+(E-1)*3*dy
dbCreateLabel(cvid list("M3" "pin") X+DX2*last4+dx_xor:Yout+dy2*last4+leny strcat("CN<" sprintf(a
"%d" CN) ">") "centerLeft" "R0" "roman" 0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvdd+leny "VDD" "centerLeft" "R0" "roman"
0.25)
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvss+leny "VSS" "centerLeft" "R0" "roman"
0.25)


leny = L139*dy+(E-1)*3*dy
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvdd+leny "VDD" "centerLeft" "R0" "roman"
0.25)
```

```
dbCreateLabel(cvid list("M1" "pin") X+dx_xor+DX2*last4:Yvss+leny "VSS" "centerLeft" "R0" "roman"
0.25)




;#############################################################################################
;#############################################################################################
;                                    Threshold InvPair Placement
;#############################################################################################
X=2.50-0.165+0.04
dx_4conn = 0.48*(E-1) ; 2 wiring tracks for each CN ; wiring-to-wiring distance = 0.14. So, total
= 2* (2* 0.14) =0.56
dx_four = 11*(Four-1)        ; 12*(Four-1)
dx2=11   ;12

for(i 1 17
      leny = Ind[i-1]*dy
      lenx = dx2*Ind2[i-1]
      if(Ind2[i-1]==1 then
                dbCreateInst(cvid pairinv2 "" X+dx_xor+dx2*Ind2[i-1]:leny "R0")
      else
                dbCreateInst(cvid pairinv1 "" X+dx_xor+dx2*Ind2[i-1]:leny "R0")
               )
)

; Now, we will place the vias for PairInvs

Xvia = 5.76  ; 1.44*4 = 5.76

Ypair_out = 2.035
Yxor_in1 = 0.995
Yxor_in2 = 0.435
Yxor_out = 1.555


for(i 1 17
      leny = Ind[i-1]*dy

      if(mod(i 2)==1 then
         dbCreateInst(cvid viaid "" Xvia+dx_4conn+dx_four:leny+Ypair_out+(0.28+0.08)*Ind2[i-1]
"R0") ;pairinv vias for odd  ; vertical distance between INV_TH1 and INV_TH2 bars is 0.28, not
dy2 here
      else
         dbCreateInst(cvid viaid ""
Xvia+dx_4conn+0.24+dx_four:leny+Ypair_out+(0.28+0.08)*Ind2[i-1] "R0") ;pairinv vias - even
      )
)

; Now, placing XOR vias and M3 bars to connect them

for(i 1 17
      leny = Ind[i-1]*dy

      if(i==2 || i==4 || i==6 || i==8 || i==10 || i==12 || i==14 || i==16 then
             dbCreateInst(cvid viaid "" Xvia+dx_four+dx_4conn:Yxor_in2+(dy2+0.08)*Ind2[i-1]+leny
"R0") ;odd pairinv outputs goes into x=9.0 XOR IN2 vias
             dbCreateInst(cvid viaid "" Xvia+dx_four+dx_4conn+0.24:Yxor_in1+(dy2+0.08)*Ind2[i-
1]+leny "R0") ;even pairinv outputs goes into x=9.4 XOR IN1 vias
             dbCreateInst(cvid m3vsid "" Xvia+dx_four+dx_4conn+0.24:Yxor_in1+leny "R0") ; very
short metals

             for(j Ind[i-2] Ind[i-1]-1
                  leny = j*dy
                  if(j!=Ind[i-1]-1 then
                      dbCreateInst(cvid m3id "" Xvia+dx_four+dx_4conn:Ypair_out+leny "R0")
                  else
                     dbCreateInst(cvid m3sid "" Xvia+dx_four+dx_4conn:Ypair_out+leny "R0")
                     )
```

```
              )
       )
      if(i==17 then
             dbCreateInst(cvid viaid "" Xvia+dx_four+dx_4conn:Yxor_in2+(dy2+0.08)*Ind2[i-1]+leny
"R0") ;only XOR IN2 via for 17th XOR
             dbCreateInst(cvid m3sid "" Xvia+dx_four+dx_4conn:Yxor_in2+dy2*Ind2[i-1]+leny-0.5
"R0")
             )
)


; Place the VDD, VSS, INpair, OUTpair Labels to match LVS
Xoff = X+0.5
Xin = 3.4

Yvdd=2.3
Yvss = 0.05
Yin = 1.00
Yout = 2.09


for(i 1 17
      leny = Ind[i-1]*dy
      lenx = dx2*Ind2[i-1]
      dbCreateLabel(cvid list("M1" "pin") Xoff+ dx2*Ind2[i-1]+dx_xor:Yvdd+leny "VDD" "centerLeft"
"R0" "roman" 0.25)
      dbCreateLabel(cvid list("M1" "pin") Xoff+ dx2*Ind2[i-1]+dx_xor:Yvss+leny "VSS" "centerLeft"
"R0" "roman" 0.25)

;      dbCreateLabel(cvid list("M3" "pin") Xoff+ dx2*Ind2[i-1]+dx_xor:Yout+leny+dy2*Ind2[i-1]
strcat("TH<" sprintf(a "%d" i-1) ">") "centerLeft" "R0" "roman" 0.25)

;      dbCreateLabel(cvid list("M1" "pin") Xin+ dx2*Ind2[i-1]+dx_xor:Yin+leny strcat("IN<"
sprintf(a "%d" i-1) ">") "centerLeft" "R0" "roman" 0.25)

;      dbCreateLabel(cvid list("M3" "label") Xptune+0.10+ dx2*Ind2[i-1]+dx_xor:Yin+leny "Ptune"
"centerLeft" "R0" "roman" 0.25)
;      dbCreateLabel(cvid list("M3" "label") Xntune+0.10+ dx2*Ind2[i-1]+dx_xor:Yin+leny "Ntune"
"centerLeft" "R0" "roman" 0.25)

;       if(Ind[i-1]<=136 then
;       dbCreateLabel(cvid list("PC" "label") Xoff+ dx2*Ind2[i-1]+dx_xor:Yin+leny strcat("VN<"
sprintf(a "%d" Ind[i-1]) ">") "centerLeft" "R0" "roman" 0.25)
;       else
;       dbCreateLabel(cvid list("PC" "label") Xoff+ dx2*Ind2[i-1]+dx_xor:Yin+leny strcat("VN<"
sprintf(a "%d" Ind[i-1]-12) ">") "centerLeft" "R0" "roman" 0.25)
;;       )

)


;#############################################################################################
;#############################################################################################
;                                        VN_PER_LANE Placement
;#############################################################################################

X=3.725
dx_four = 11*(Four-1)        ; 12*(Four-1)
dx2=11    ; 12
dx_4conn = 1.44*(E-1)

Yin2= 2.885

for(i 1 17
      leny = Ind[i-1]*dy
      lenx = dx2*Ind2[i-1]
      if(Ind2[i-1]==1 then
                dbCreateInst(cvid vnperlane2 "" X+dx_xor+dx2*Ind2[i-1]:leny "R0")
      else
                dbCreateInst(cvid vnperlane1 "" X+dx_xor+dx2*Ind2[i-1]:leny "R0")
             )
```

```
)


Yvss = 0.355
Yvdd = 2.6
for(i 1 17
      leny = Ind[i-1]*dy
      lenx = dx2*Ind2[i-1]
      dbCreateLabel(cvid list("M1" "pin") X+1.5+dx2*Ind2[i-1]+dx_xor:Yvdd+leny "VDD" "centerLeft"
"R0" "roman" 0.25)
      dbCreateLabel(cvid list("M1" "pin") X+1.5+dx2*Ind2[i-1]+dx_xor:Yvss+leny "VSS" "centerLeft"
"R0" "roman" 0.25)

       dbCreateInst(cvid viaid "" 1.2+dx_4conn+dx_four:Yin2+(0.38+0.08)*Ind2[i-1]+leny "R0")
;       label1 = strcat("TH" sprintf(a "%d" Ind[i-1]) "_CN" sprintf(a "%d" CN))
;       dbCreateLabel(cvid list("M3" "pin") 1.2+dx_4conn+dx_four:Yin2+0.38*Ind2[i-1]+leny label1
"centerLeft" "R0" "roman" 0.25)
)







)
)


; ####################################   PTUNE and NTUNE vertical bar and horizontal via+bar
placement ###################
; The two brackets are done above. Because ptune and ntune bars will be placed once, not in every
iteration.

Xptune= 2.955
Xntune= 2.675
XptuneDelay=4.655
XntuneDelay=4.145
Xcsptune=9.29
Xcsntune=9.09
Xstart=10.31
Xstartbar=10.11
Xstart2=0.65
Xstartbar2=-1.295


for(k 1 69            ; there are 273/4 = 70 vertical columns
      for(i 1 293+1
            leny=(i-1)*dy
            instid = dbCreateInst(cvid m2id "" Xptune+(k-1)*dx2:leny "R0")
            instid = dbCreateInst(cvid m2id "" Xntune+(k-1)*dx2:leny "R0")
            instid = dbCreateInst(cvid m2id "" XptuneDelay+(k-1)*dx2:leny "R0")
            instid = dbCreateInst(cvid m2id "" XntuneDelay+(k-1)*dx2:leny "R0")
            instid = dbCreateInst(cvid m2id "" Xcsptune+(k-1)*dx2:leny "R0")
            instid = dbCreateInst(cvid m2id "" Xcsntune+(k-1)*dx2:leny "R0")
            instid = dbCreateInst(cvid m2id "" Xstart+(k-1)*dx2:leny "R0")
            instid = dbCreateInst(cvid m2id "" Xstartbar+(k-1)*dx2:leny "R0")
          )
          dbCreateLabel(cvid list("M2" "pin") Xptune+0.05+(k-1)*dx2:5 "PTUNE" "centerLeft" "R0"
"roman" 0.25)
          dbCreateLabel(cvid list("M2" "pin") Xntune+0.05+(k-1)*dx2:5 "NTUNE" "centerLeft" "R0"
"roman" 0.25)
          dbCreateLabel(cvid list("M2" "pin") XptuneDelay+0.05+(k-1)*dx2:5 "PTUNEDELAY"
"centerLeft" "R0" "roman" 0.25)
          dbCreateLabel(cvid list("M2" "pin") XntuneDelay+0.05+(k-1)*dx2:5 "NTUNEDELAY"
"centerLeft" "R0" "roman" 0.25)
          dbCreateLabel(cvid list("M2" "pin") Xcsptune+0.05+(k-1)*dx2:5 "CSPTUNE" "centerLeft"
"R0" "roman" 0.25)
          dbCreateLabel(cvid list("M2" "pin") Xcsntune+0.05+(k-1)*dx2:5 "CSNTUNE" "centerLeft"
"R0" "roman" 0.25)
```

```
         dbCreateLabel(cvid list("M2" "pin") Xstart+0.05+(k-1)*dx2:5 "START" "centerLeft" "R0"
"roman" 0.25)
         dbCreateLabel(cvid list("M2" "pin") Xstartbar+0.05+(k-1)*dx2:5 "STARTBAR" "centerLeft"
"R0" "roman" 0.25)
)

instid = dbCreateInst(cvid ptunebar "" 0:0 "R0")
instid = dbCreateInst(cvid ptunebar "" 0:0.24 "R0")
instid = dbCreateInst(cvid ptunebar "" 0:0.48 "R0")
instid = dbCreateInst(cvid ptunebar "" 0:0.72 "R0")
instid = dbCreateInst(cvid ptunebar "" 0:0.96 "R0")
instid = dbCreateInst(cvid ptunebar "" 0:1.20 "R0")
instid = dbCreateInst(cvid ptunebar "" -4:1.44 "R0")
instid = dbCreateInst(cvid ptunebar "" -4:1.68 "R0")

for(k 1 69
    dbCreateInst(cvid m2m3 "" Xptune+(k-1)*dx2:0 "R0")
    dbCreateInst(cvid m2m3 "" Xntune+(k-1)*dx2:0.24 "R0")
    dbCreateInst(cvid m2m3 "" XptuneDelay+(k-1)*dx2:0.48 "R0")
    dbCreateInst(cvid m2m3 "" XntuneDelay+(k-1)*dx2:0.72 "R0")
    dbCreateInst(cvid m2m3 "" Xcsptune+(k-1)*dx2+0.04:0.96 "R0")
    dbCreateInst(cvid m2m3 "" Xcsntune+(k-1)*dx2-0.04:1.20 "R0")
    dbCreateInst(cvid m2m3 "" Xstart+(k-1)*dx2+0.04:1.44 "R0")
    dbCreateInst(cvid m2m3 "" Xstartbar+(k-1)*dx2-0.04:1.68 "R0")
    )

for(i 1 293
      leny=(i-1)*dy
      instid = dbCreateInst(cvid m2id "" Xstart2-2:leny "R0")     ; Placing the vertical load,
loadbar M2 wires
      instid = dbCreateInst(cvid m2id "" Xstartbar2-2:leny "R0")
)

dbCreateLabel(cvid list("M2" "pin") -3.25:5 "LOAD" "centerLeft" "R0" "roman" 0.25)
dbCreateLabel(cvid list("M2" "pin") -1.3:5 "LOADBAR" "centerLeft" "R0" "roman" 0.25)


;dbCreateInst(cvid m2m3 "" Xstart2-2:1.44 "R0")    ; disable M2M3 via connecting LOAD bars to
START bars.
;dbCreateInst(cvid m2m3 "" Xstartbar2-2:1.68 "R0")


; ###################  Connect P/Ntune, P/NtuneDelay, CSP/Ntune, START/BAR horizontally
###################

for(i 1 288   ; 288 selected instead of 293+1 because of easiness in mod, otherwise some vias
cross 293 bar
              leny = (i-1)*dy
              instid = dbCreateInst(cvid ptunebar "" 0:4.7+leny "R0")

              if(mod(i 8)==1 then

                      for(k 1 69
                      dbCreateInst(cvid m2m3 "" Xptune+(k-1)*dx2+0.04:4.74+leny "R0")
                      dbCreateInst(cvid m2m3 "" Xntune+(k-1)*dx2+0.04:4.74+leny+6.25*1 "R0")
                      dbCreateInst(cvid m2m3 "" XptuneDelay+(k-1)*dx2+0.04:4.74+leny+6.25*2
"R0")
                      dbCreateInst(cvid m2m3 "" XntuneDelay+(k-1)*dx2+0.04:4.74+leny+6.25*3
"R0")
                      dbCreateInst(cvid m2m3 "" Xcsptune+(k-1)*dx2+0.04:4.74+leny+6.25*4 "R0")
                      dbCreateInst(cvid m2m3 "" Xcsntune+(k-1)*dx2-0.04:4.74+leny+6.25*5 "R0")
                      dbCreateInst(cvid m2m3 "" Xstart+(k-1)*dx2+0.04:4.74+leny+6.25*6 "R0")
                      dbCreateInst(cvid m2m3 "" Xstartbar+(k-1)*dx2-0.04:4.74+leny+6.25*7 "R0")
                      )

               )
)
```

```
; %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%   VN MG bar, SWITCH_START, L_SHAPE  Placement
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; The two brackets are done above. Because VN wires will be placed just once, not in every
iteration.
yoff_VN=-0.5
for(i 1 136
      leny=i*dy
      instid = dbCreateInst(cvid vn "" -0.5:yoff_VN+leny "R0")
      dbCreateLabel(cvid list("M6" "pin") -0.5:yoff_VN+0.05+leny strcat("VN<" sprintf(a "%d" i)
">") "centerLeft" "R0" "roman" 0.25)
      instid = dbCreateInst(cvid sw "" -4:-2.06+leny "R0")
      dbCreateLabel(cvid list("M1" "pin") -3.85:-2.06+1.60+leny strcat("VNCHAN<" sprintf(a "%d"
i)  ">") "centerLeft" "R0" "roman" 0.25)
      dbCreateLabel(cvid list("M1" "pin") -2.5:-2.06-0.25+leny "VSS" "centerLeft" "R0" "roman"
0.25)
      dbCreateLabel(cvid list("M1" "pin") -2.5:3.42+leny "VDD" "centerLeft" "R0" "roman" 0.25)

      xshift=mod(i 6)
      instid = dbCreateInst(cvid lshape "" -4.2-0.24*xshift:yoff_VN+leny-0.10 "MX")
      instid = dbCreateInst(cvid lshape2 "" 765+0.24*xshift:yoff_VN+leny-0.10 "MX")
      )

for(i 149 285
      leny=i*dy
      instid = dbCreateInst(cvid vn "" -0.5:yoff_VN+leny "R0")
      dbCreateLabel(cvid list("M6" "pin") -0.5:yoff_VN+0.05+leny strcat("VN<" sprintf(a "%d" i-
12) ">") "centerLeft" "R0" "roman" 0.25)
      instid = dbCreateInst(cvid sw2 "" -4:-2.06+leny "R0")
      dbCreateLabel(cvid list("M1" "pin") -3.85:-2.06+1.60+leny strcat("VNCHAN<" sprintf(a "%d"
i-12)  ">") "centerLeft" "R0" "roman" 0.25)
      dbCreateLabel(cvid list("M1" "pin") -2.5:-2.06-0.25+leny "VSS" "centerLeft" "R0" "roman"
0.25)
      dbCreateLabel(cvid list("M1" "pin") -2.5:3.42+leny "VDD" "centerLeft" "R0" "roman" 0.25)

      xshift=mod((i-149) 6)
      instid = dbCreateInst(cvid lshape "" -4.2-0.24*xshift:yoff_VN+leny "R0")
      instid = dbCreateInst(cvid lshape2 "" 765+0.24*xshift:yoff_VN+leny "R0")
      )




;instid = dbCreateInst(cvid m7 "" 0:0 "R0")

for(i 1 69
      lenx=(i-1)*11
      dbCreateInst(cvid shield_m5 "" lenx:0 "R0")
      )



; ##### FOR SHIELDING and viaM5M6 and viaM6M7 #####

via_shield = dbOpenCellViewByType("LibSanjoy" "VIA_SHIELD" "layout" "" "r")

Xoff1 = 1.45
Xoff2 = 4.03
Xoff3 = 6.45
Xoff4 = 8.62
Yoff1 = 5.43  ; the bottom bar of VN = VSS
Yoff2 = 5.99  ; the above bar of VN = VDD

for(i 1 69
      for(j 1 273
           if(j<137 then
           instid = dbCreateInst(cvid via_shield "" Xoff1+(i-1)*11:Yoff2+(j-1)*6.25 "R0")
           instid = dbCreateInst(cvid via_shield "" Xoff3+(i-1)*11:Yoff2+(j-1)*6.25 "R0")

           instid = dbCreateInst(cvid via_shield "" Xoff2+(i-1)*11:Yoff1+(j-1)*6.25 "R0")
```

```
            instid = dbCreateInst(cvid via_shield "" Xoff4+(i-1)*11:Yoff1+(j-1)*6.25 "R0")

            else

            instid = dbCreateInst(cvid via_shield "" Xoff1+(i-1)*11:Yoff2+(j-1+12)*6.25 "R0")
            instid = dbCreateInst(cvid via_shield "" Xoff3+(i-1)*11:Yoff2+(j-1+12)*6.25 "R0")

            instid = dbCreateInst(cvid via_shield "" Xoff2+(i-1)*11:Yoff1+(j-1+12)*6.25 "R0")
            instid = dbCreateInst(cvid via_shield "" Xoff4+(i-1)*11:Yoff1+(j-1+12)*6.25 "R0")
            )
            )
)

; #### Place VIA_M7_M8 to connect to the power grid

via_M7M8 = dbOpenCellViewByType("LibSanjoy" "VIA_M7_M8" "layout" "" "r")

Xoff1 = 1.45
Xoff2 = 4.03
Xoff3 = 6.45
Xoff4 = 8.62

Yoff1 = 0.20
Yoff2 = 2.20

for(i 1 69
      for(j 1 294
            instid = dbCreateInst(cvid via_M7M8 "" Xoff1+(i-1)*11:Yoff2+(j-1)*6.25 "R0")
            instid = dbCreateInst(cvid via_M7M8 "" Xoff3+(i-1)*11:Yoff2+(j-1)*6.25 "R0")

            instid = dbCreateInst(cvid via_M7M8 "" Xoff2+(i-1)*11:Yoff1+(j-1)*6.25 "R0")
            instid = dbCreateInst(cvid via_M7M8 "" Xoff4+(i-1)*11:Yoff1+(j-1)*6.25 "R0")
            )
)

; #### Place VIA_M7_M8 to connect switches_START to the power grid

Xswvdd=-1.22
Xswvss=-3.56

Yoff1=0.20
Yoff2=2.20

for(j 1 293
    instid = dbCreateInst(cvid via_shield "" Xswvdd:Yoff2+(j-1)*6.25 "R0")
    instid = dbCreateInst(cvid via_shield "" Xswvss:Yoff1+(j-1)*6.25 "R0")

    instid = dbCreateInst(cvid via_M7M8 "" Xswvdd:Yoff2+(j-1)*6.25 "R0")
    instid = dbCreateInst(cvid via_M7M8 "" Xswvss:Yoff1+(j-1)*6.25 "R0")
)

; #### Place Vertical M5 to supply power to SWITCH_START

m5 = dbOpenCellViewByType("LibSanjoy" "M5" "layout" "" "r")
instid = dbCreateInst(cvid m5 "" -3.575:0 "R0")
instid = dbCreateInst(cvid m5 "" -1.24:0 "R0")


;
##############################################################################################
##########################################################################
; ###################################################################      POWER GRID
##############################################################################

pgrid_analog = dbOpenCellViewByType("LibSanjoy" "PGRID_M8_M9_ANALOG" "layout" "" "r")
pgrid = dbOpenCellViewByType("LibSanjoy" "PGRID_M8_M9" "layout" "" "r")

Xoff=-3.56
Yoff=-0.62

for(j 1 293
```

```
      for(i 1 34
          instid = dbCreateInst(cvid pgrid_analog "" Xoff+(i-1)*22:Yoff+(j-1)*6.25 "R0")
          )

      for(i 35 35
          instid = dbCreateInst(cvid pgrid "" Xoff+(i-1)*22:Yoff+(j-1)*6.25 "R0")
          )
)









; ##### For LVS ######


m6 = dbOpenCellViewByType("LibSanjoy" "M6" "layout" "" "r")
viaM5M6 = dbOpenCellViewByType("LibSanjoy" "VIA_M5_M6" "layout" "" "r")

;instid = dbCreateInst(cvid m6 "" -4:1832 "R0")
;instid = dbCreateInst(cvid m6 "" -4:1833.76 "R0")

;instid = dbCreateInst(cvid viaM5M6 "" -3.38:1834 "R0")
;instid = dbCreateInst(cvid viaM5M6 "" -0.97:1832.1 "R0")

;for(i 1 69
;instid = dbCreateInst(cvid viaM5M6 "" 9.30+(i-1)*11:1834 "R0")
;instid = dbCreateInst(cvid viaM5M6 "" 7.88+(i-1)*11:1832.1 "R0")
;)




; METAL DENSITY FILL

dm6 = dbOpenCellViewByType("LibSanjoy" "VN_CN_DM6" "layout" "" "r")

for(i 1 293
      for(j 1 34
            leny = (i-1)*6.25
            lenx = (j-1)*22
            instid = dbCreateInst(cvid dm6 "" 2+lenx:0.5+leny "R0")
            )
)

; #### Now, we ground-connect all the M6 pieces used for density filling

Xoff1 = 1.45
Xoff3 = 6.45
Yoff = 0.6  ; the bottom bar of VN


for(i 1 69
      for(j 1 273
            if(j<137 then
```

```
            instid = dbCreateInst(cvid via_shield "" Xoff1+(i-1)*11:Yoff+(j-1)*6.25 "R0")
            instid = dbCreateInst(cvid via_shield "" Xoff3+(i-1)*11:Yoff+(j-1)*6.25 "R0")

            else

            instid = dbCreateInst(cvid via_shield "" Xoff1+(i-1)*11:Yoff+(j-1+12)*6.25 "R0")
            instid = dbCreateInst(cvid via_shield "" Xoff3+(i-1)*11:Yoff+(j-1+12)*6.25 "R0")
            )
            )
)



; ## DOD DPO FILL INTO BLANK SPACES NOT COVERED BY ANY CELLVIEW

;f4= infile("SkillSanjoy/DOD_DPO.txt")
;declare(Hdpo[272*17])
;declare(dum[4*17])

;con=0
;for(i 1 length(Hdpo)
;    fscanf(f4 "%d" con)
;    Hdpo[i-1] = con
;)
;close(f4)

;for(i 1 272*17
;      if(mod(i,17*4)==1 then
;           col = (i-1)/(17*4)+1
 ;          r=0
;           for(q i i+17*4-1
;                dum[r]=Hdpo[q-1]
;                r=r+1
;                )

;for(m 2 17*4
;j=dum[m-2]
;k=dum[m-1]
;       if(k>j+1 then
;               for(lp j+1 k-1
;               leny = lp*dy
;               dbCreateInst(cvid dpo "" col*11:leny "R0")
;               )
;       )
;)

;)
;)
```

B.3 Cadence SKILL code which generates the input shift register schematic and layout

```
; SR273_7 : In the first section, scematic. Last section, layout

cvid = dbOpenCellViewByType("LibSanjoy" "SR273_7" "schematic" "" "w")
dff7id = dbOpenCellViewByType("LibSanjoy" "DFF7" "symbol" "" "r")

pinMaster = dbOpenCellViewByType("basic" "ipin" "symbol")



dy=4

Xvdd=0.5625
Xvss=1.0625
Xclk=0.5625
Xswn=0.8125
Xswp=1.0625
Xin=0
Xbit=1.6250

Yvdd=0.5625
Yvss=0.5625
Yclk=-2.25
Yswn=-2.25
Yswp=-2.25
Yin=-0.8125
Yb1=-1.6250
Yb2=-1.50
Yb3=-1.375
Yb4=-1.25
Yb5=-1.125
Yb6=-1.00
Yb7=-0.875
Ybb1=-0.75
Ybb2=-0.625
Ybb3=-0.50
Ybb4=-0.375
Ybb5=-0.25
Ybb6=-0.125
Ybb7=0


for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid dff7id "" 0:leny "R0")
      schCreatePin(cvid pinMaster "VDD" "inputOutput" nil Xvdd:leny+Yvdd "R270")
      schCreatePin(cvid pinMaster "VSS" "inputOutput" nil Xvss:leny+Yvss "R270")
      schCreatePin(cvid pinMaster "CLK" "inputOutput" nil Xclk:leny+Yclk "R90")
      schCreatePin(cvid pinMaster "SWN" "inputOutput" nil Xswn:leny+Yswn "R90")
      schCreatePin(cvid pinMaster "SWP" "inputOutput" nil Xswp:leny+Yswp "R90")
      if(i!=1 then
      wireID = schCreateWire(cvid "draw" "full" list(Xin:leny+Yin Xin-0.2:leny+Yin) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xin-0.2:leny+Yin) strcat("B7_row<" sprintf(a "%d" i-1)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      )

      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb1 Xbit+0.2:leny+Yb1) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb1) strcat("B1_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb2 Xbit+0.2:leny+Yb2) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb2) strcat("B2_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
```

```
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb3 Xbit+0.2:leny+Yb3) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb3) strcat("B3_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb4 Xbit+0.2:leny+Yb4) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb4) strcat("B4_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb5 Xbit+0.2:leny+Yb5) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb5) strcat("B5_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb6 Xbit+0.2:leny+Yb6) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb6) strcat("B6_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb7 Xbit+0.2:leny+Yb7) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb7) strcat("B7_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)


      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb1 Xbit+0.2:leny+Ybb1) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb1) strcat("BB1_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb2 Xbit+0.2:leny+Ybb2) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb2) strcat("BB2_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb3 Xbit+0.2:leny+Ybb3) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb3) strcat("BB3_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb4 Xbit+0.2:leny+Ybb4) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb4) strcat("BB4_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb5 Xbit+0.2:leny+Ybb5) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb5) strcat("BB5_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb6 Xbit+0.2:leny+Ybb6) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb6) strcat("BB6_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb7 Xbit+0.2:leny+Ybb7) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb7) strcat("BB7_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)

)

      schCreatePin(cvid pinMaster "IN" "input" nil Xin:Yin "R0")

      schCreatePin(cvid pinMaster "B1_row<1:273>" "inputOutput" nil Xbit+0.4:Yb1 "R180")
      schCreatePin(cvid pinMaster "B2_row<1:273>" "inputOutput" nil Xbit+0.4:Yb2 "R180")
      schCreatePin(cvid pinMaster "B3_row<1:273>" "inputOutput" nil Xbit+0.4:Yb3 "R180")
      schCreatePin(cvid pinMaster "B4_row<1:273>" "inputOutput" nil Xbit+0.4:Yb4 "R180")
      schCreatePin(cvid pinMaster "B5_row<1:273>" "inputOutput" nil Xbit+0.4:Yb5 "R180")
      schCreatePin(cvid pinMaster "B6_row<1:273>" "inputOutput" nil Xbit+0.4:Yb6 "R180")
      schCreatePin(cvid pinMaster "B7_row<1:273>" "inputOutput" nil Xbit+0.4:Yb7 "R180")

      schCreatePin(cvid pinMaster "BB1_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb1 "R180")
      schCreatePin(cvid pinMaster "BB2_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb2 "R180")
      schCreatePin(cvid pinMaster "BB3_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb3 "R180")
      schCreatePin(cvid pinMaster "BB4_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb4 "R180")
      schCreatePin(cvid pinMaster "BB5_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb5 "R180")
      schCreatePin(cvid pinMaster "BB6_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb6 "R180")
      schCreatePin(cvid pinMaster "BB7_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb7 "R180")
```

```
;
#############################################################################################
#############################################################
; ###########################################################################   LAYOUT
#####################################################################
;
#############################################################################################
#############################################################


cvid = dbOpenCellViewByType("LibSanjoy" "SR273_7" "layout" "" "w")
dff7id = dbOpenCellViewByType("LibSanjoy" "DFF7" "layout" "" "r")

dy = 6.25

Xvdd=3
Xvss=3
Xclk=0.30
Xswn=6.69
Xswp=7.645
Xin=1.36
Xbit=3

Yvdd=2
Yvss=0.145
Yclk=3.55
Yswn=3.55
Yswp=3.55
Yin=1.035
Yb1=0.325
Yb2=0.60
Yb3=0.80
Yb4=1.085
Yb5=1.28
Yb6=1.57
Yb7=1.77
Ybb1=2.05
Ybb2=2.255
Ybb3=2.53
Ybb4=2.72
Ybb5=3
Ybb6=3.21
Ybb7=3.485




for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid dff7id "" 0:leny "R0")
      dbCreateLabel(cvid list("M1" "pin") Xvdd:Yvdd+leny "VDD" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M1" "pin") Xvss:Yvss+leny "VSS" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xclk:Yclk+leny "CLK" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xswn:Yswn+leny "SWN" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xswp:Yswp+leny "SWP" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M1" "pin") Xin:Yin+leny "IN" "centerLeft" "R0" "roman" 0.10)

      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb1+leny strcat("B1_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb2+leny strcat("B2_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb3+leny strcat("B3_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb4+leny strcat("B4_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
```

```
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb5+leny strcat("B5_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb6+leny strcat("B6_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb7+leny strcat("B7_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb1+leny strcat("BB1_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb2+leny strcat("BB2_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb3+leny strcat("BB3_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb4+leny strcat("BB4_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb5+leny strcat("BB5_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb6+leny strcat("BB6_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb7+leny strcat("BB7_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)


)
```

B.4 Cadence SKILL code which generates the output shift register schematic and layout

```
; SR273_7 : In the first section, scematic. Last section, layout

cvid = dbOpenCellViewByType("LibSanjoy" "SR_OUT" "schematic" "" "w")
unitid = dbOpenCellViewByType("LibSanjoy" "UNIT_PIPO_PISO" "symbol" "" "r")

pinMaster = dbOpenCellViewByType("basic" "ipin" "symbol")



dy=4

Xvdd=0.625
Xvss=0.8125
Xptune=1.5
Xntune=1.6875
Xclkpipo=0.5625
Xclkpiso=1.0625
Xswn=1.5
Xswp=1.6875
Xvn=0
Xinpiso=0
Xoutpiso=2.25

Yvdd=1.75
Yvss=1.75
Yptune=1.75
```

```
Yntune=1.75
Yclkpipo=0
Yclkpiso=0
Yswn=0
Yswp=0
Yvn=1.0625
Yinpiso=0.6875
Youtpiso=0.8750


for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid unitid "" 0:leny "R0")
      schCreatePin(cvid pinMaster "VDD" "inputOutput" nil Xvdd:leny+Yvdd "R270")
      schCreatePin(cvid pinMaster "VSS" "inputOutput" nil Xvss:leny+Yvss "R270")
      schCreatePin(cvid pinMaster "PTUNE" "inputOutput" nil Xptune:leny+Yptune "R270")
      schCreatePin(cvid pinMaster "NTUNE" "inputOutput" nil Xntune:leny+Yntune "R270")
      schCreatePin(cvid pinMaster "CLK_PIPO" "inputOutput" nil Xclkpipo:leny+Yclkpipo "R90")
      schCreatePin(cvid pinMaster "CLK_PISO" "inputOutput" nil Xclkpiso:leny+Yclkpiso "R90")
      schCreatePin(cvid pinMaster "SWN" "inputOutput" nil Xswn:leny+Yswn "R90")
      schCreatePin(cvid pinMaster "SWP" "inputOutput" nil Xswp:leny+Yswp "R90")

      wireID = schCreateWire(cvid "draw" "full" list(Xvn:leny+Yvn Xvn-0.2:leny+Yvn) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xvn-0.2:leny+Yvn) strcat("VN<" sprintf(a "%d" i) ">")
"lowerRight" "R0" "stick" 0.0625 nil)

      if(i!=273 then
      wireID = schCreateWire(cvid "draw" "full" list(Xoutpiso:leny+Youtpiso
Xoutpiso+0.2:leny+Youtpiso) 0.0625 0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xoutpiso+0.2:leny+Youtpiso) strcat("OUT_PISO<"
sprintf(a "%d" i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      )

      if(i==273 then
      schCreatePin(cvid pinMaster "OUT_CHIP" "output" nil Xoutpiso:leny+Youtpiso "R180")
      )

      if(i!=1 then
      wireID = schCreateWire(cvid "draw" "full" list(Xinpiso:leny+Yinpiso Xinpiso-
0.2:leny+Yinpiso) 0.0625 0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xinpiso-0.2:leny+Yinpiso) strcat("OUT_PISO<" sprintf(a
"%d" i-1) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      )

)
      schCreatePin(cvid pinMaster "VN<1:273>" "inputOutput" nil Xvn-0.6:Yvn "R0")







;
#############################################################################################
#############################################################
; ###########################################################################   LAYOUT
#####################################################################
;
#############################################################################################
#############################################################


cvid = dbOpenCellViewByType("LibSanjoy" "SR_OUT" "layout" "" "w")
unitid = dbOpenCellViewByType("LibSanjoy" "UNIT_PIPO_PISO" "layout" "" "r")


dy=6.25
```

```
Xvdd=4
Xvss=4
Xptune=0.6
Xntune=0.34
Xclkpipo=1.54
Xclkpiso=8.285
Xswn=8
Xswp=7.72
Xvn=1.04
Xinpiso=9.45
Xoutpiso=14

Yvdd=2
Yvss=0.15
Yptune=2.85
Yntune=2.85
Yclkpipo=2.85
Yclkpiso=2.85
Yswn=2.85
Yswp=2.85
Yvn=1
Yinpiso=1
Youtpiso=1



for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid unitid "" 0:leny "R0")
      dbCreateLabel(cvid list("M1" "pin") Xvdd:Yvdd+leny "VDD" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M1" "pin") Xvss:Yvss+leny "VSS" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xptune:Yptune+leny "PTUNE" "centerLeft" "R0" "roman"
0.10)
      dbCreateLabel(cvid list("M2" "pin") Xntune:Yntune+leny "NTUNE" "centerLeft" "R0" "roman"
0.10)
      dbCreateLabel(cvid list("M3" "pin") Xclkpipo:Yclkpipo+leny "CLK_PIPO" "centerLeft" "R0"
"roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xclkpiso:Yclkpiso+leny "CLK_PISO" "centerLeft" "R0"
"roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xswn:Yswn+leny "SWN" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xswp:Yswp+leny "SWP" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M1" "pin") Xvn:Yvn+leny strcat("VN<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

      if(i==273 then
      dbCreateLabel(cvid list("M1" "pin") Xoutpiso:Youtpiso+leny "OUT_CHIP" "centerLeft" "R0"
"roman" 0.10)
      )



)
```

B.5 Cadence SKILL code which generates the memory array schematic and layout

```
; SRAM273_70 : In the first section, scematic. Last section, layout

cvid = dbOpenCellViewByType("LibSanjoy" "SRAM273_70" "schematic" "" "w")
sram70id = dbOpenCellViewByType("LibSanjoy" "SRAM70" "symbol" "" "r")

pinMaster = dbOpenCellViewByType("basic" "ipin" "symbol")



dy=4

Xvdd=0.625
Xvss=1.25

Xbit=2.125
Yword=3.4375

Yvdd=0.4375
Yvss=0.4375

Xw1=1.5
Xw2=1.375
Xw3=1.25
Xw4=1.125
Xw5=1.00
Xw6=0.875
Xw7=0.75
Xw8=0.625
Xw9=0.50
Xw10=0.375

Yb1=1.125
Yb2=1.25
Yb3=1.375
Yb4=1.5
Yb5=1.625
Yb6=1.75
Yb7=1.875
Ybb1=2.00
Ybb2=2.125
Ybb3=2.25
Ybb4=2.375
Ybb5=2.50
Ybb6=2.625
Ybb7=2.75


for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid sram70id "" 0:leny "R0")
      schCreatePin(cvid pinMaster "VDD" "inputOutput" nil Xvdd:leny+Yvdd "R90")
      schCreatePin(cvid pinMaster "VSS" "inputOutput" nil Xvss:leny+Yvss "R90")

      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb1 Xbit+0.2:leny+Yb1) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb1) strcat("B1_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb2 Xbit+0.2:leny+Yb2) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb2) strcat("B2_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb3 Xbit+0.2:leny+Yb3) 0.0625
0.0625 0)
```

```
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb3) strcat("B3_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb4 Xbit+0.2:leny+Yb4) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb4) strcat("B4_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb5 Xbit+0.2:leny+Yb5) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb5) strcat("B5_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb6 Xbit+0.2:leny+Yb6) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb6) strcat("B6_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Yb7 Xbit+0.2:leny+Yb7) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Yb7) strcat("B7_row<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)


      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb1 Xbit+0.2:leny+Ybb1) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb1) strcat("BB1_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb2 Xbit+0.2:leny+Ybb2) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb2) strcat("BB2_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb3 Xbit+0.2:leny+Ybb3) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb3) strcat("BB3_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb4 Xbit+0.2:leny+Ybb4) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb4) strcat("BB4_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb5 Xbit+0.2:leny+Ybb5) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb5) strcat("BB5_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb6 Xbit+0.2:leny+Ybb6) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb6) strcat("BB6_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xbit:leny+Ybb7 Xbit+0.2:leny+Ybb7) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xbit+0.2:leny+Ybb7) strcat("BB7_row<" sprintf(a "%d"
i) ">") "lowerRight" "R0" "stick" 0.0625 nil)

      schCreatePin(cvid pinMaster "WORD1" "inputOutput" nil Xw1:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD2" "inputOutput" nil Xw2:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD3" "inputOutput" nil Xw3:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD4" "inputOutput" nil Xw4:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD5" "inputOutput" nil Xw5:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD6" "inputOutput" nil Xw6:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD7" "inputOutput" nil Xw7:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD8" "inputOutput" nil Xw8:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD9" "inputOutput" nil Xw9:leny+Yword "R270")
      schCreatePin(cvid pinMaster "WORD10" "inputOutput" nil Xw10:leny+Yword "R270")


)


      schCreatePin(cvid pinMaster "B1_row<1:273>" "inputOutput" nil Xbit+0.4:Yb1 "R180")
      schCreatePin(cvid pinMaster "B2_row<1:273>" "inputOutput" nil Xbit+0.4:Yb2 "R180")
      schCreatePin(cvid pinMaster "B3_row<1:273>" "inputOutput" nil Xbit+0.4:Yb3 "R180")
      schCreatePin(cvid pinMaster "B4_row<1:273>" "inputOutput" nil Xbit+0.4:Yb4 "R180")
      schCreatePin(cvid pinMaster "B5_row<1:273>" "inputOutput" nil Xbit+0.4:Yb5 "R180")
      schCreatePin(cvid pinMaster "B6_row<1:273>" "inputOutput" nil Xbit+0.4:Yb6 "R180")
      schCreatePin(cvid pinMaster "B7_row<1:273>" "inputOutput" nil Xbit+0.4:Yb7 "R180")
```

```
      schCreatePin(cvid pinMaster "BB1_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb1 "R180")
      schCreatePin(cvid pinMaster "BB2_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb2 "R180")
      schCreatePin(cvid pinMaster "BB3_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb3 "R180")
      schCreatePin(cvid pinMaster "BB4_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb4 "R180")
      schCreatePin(cvid pinMaster "BB5_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb5 "R180")
      schCreatePin(cvid pinMaster "BB6_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb6 "R180")
      schCreatePin(cvid pinMaster "BB7_row<1:273>" "inputOutput" nil Xbit+0.4:Ybb7 "R180")




;
################################################################################################
##########################################################
; ###########################################################################   LAYOUT
###################################################################
;
################################################################################################
##########################################################


cvid = dbOpenCellViewByType("LibSanjoy" "SRAM273_70" "layout" "" "w")
sram70id = dbOpenCellViewByType("LibSanjoy" "SRAM70" "layout" "" "r")

dy = 6.25

Xvdd=3
Xvss=3


Xbit=3

Yvdd=2.29
Yvss=0.105

Yb1=0.325
Yb2=0.60
Yb3=0.80
Yb4=1.085
Yb5=1.28
Yb6=1.57
Yb7=1.77
Ybb1=2.05
Ybb2=2.255
Ybb3=2.53
Ybb4=2.72
Ybb5=3
Ybb6=3.21
Ybb7=3.485

Yw=0.255
Xw1=9.65
Xw2=28.83
Xw3=48.02
Xw4=67.18
Xw5=86.36
Xw6=105.55
Xw7=124.735
Xw8=143.925
Xw9=163.085
Xw10=182.275


for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid sram70id "" 0:leny "R0")
      dbCreateLabel(cvid list("M1" "pin") Xvdd:Yvdd+leny "VDD" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M1" "pin") Xvss:Yvss+leny "VSS" "centerLeft" "R0" "roman" 0.10)
```

```
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb1+leny strcat("B1_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb2+leny strcat("B2_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb3+leny strcat("B3_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb4+leny strcat("B4_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb5+leny strcat("B5_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb6+leny strcat("B6_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Yb7+leny strcat("B7_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb1+leny strcat("BB1_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb2+leny strcat("BB2_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb3+leny strcat("BB3_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb4+leny strcat("BB4_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb5+leny strcat("BB5_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb6+leny strcat("BB6_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xbit:Ybb7+leny strcat("BB7_row<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

      dbCreateLabel(cvid list("M2" "pin") Xw1:Yw+leny "WORD1" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw2:Yw+leny "WORD2" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw3:Yw+leny "WORD3" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw4:Yw+leny "WORD4" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw5:Yw+leny "WORD5" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw6:Yw+leny "WORD6" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw7:Yw+leny "WORD7" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw8:Yw+leny "WORD8" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw9:Yw+leny "WORD9" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xw10:Yw+leny "WORD10" "centerLeft" "R0" "roman" 0.10)


)
```

B.6 Cadence SKILL code which generates all 273 DAC schematic and layout

```
; DAC273 : In the first section, scematic. Last section, layout

cvid = dbOpenCellViewByType("LibSanjoy" "DAC273" "schematic" "" "w")
dacid = dbOpenCellViewByType("LibSanjoy" "DAC" "symbol" "" "r")

pinMaster = dbOpenCellViewByType("basic" "ipin" "symbol")



dy=4

Xvdd=0.5625
Xvss=1.0625
Xvddby2=0.8125
Xpre=0.5
Xprebar=0.6875
Xch=0.9375
Xchbar=1.125
Xtog=1.375
Xtogbar=1.5625
Xout=2.0625

Xd=0

Yvdd=-0.4375
Yvss=-0.4375
Yvddby2=-0.4375
Ypre=2.375
Yprebar=2.375
Ych=2.375
Ychbar=2.375
Ytog=2.375
Ytogbar=2.375
Yout=1

Yd1=0
Yd2=0.125
Yd3=0.25
Yd4=0.375
Yd5=0.5
Yd6=0.625
Yd7=0.75
Ydb1=0.875
Ydb2=1
Ydb3=1.125
Ydb4=1.25
Ydb5=1.375
Ydb6=1.5
Ydb7=1.625


for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid dacid "" 0:leny "R0")
      schCreatePin(cvid pinMaster "VDD" "inputOutput" nil Xvdd:leny+Yvdd "R90")
      schCreatePin(cvid pinMaster "VSS" "inputOutput" nil Xvss:leny+Yvss "R90")
      schCreatePin(cvid pinMaster "VDDBY2" "inputOutput" nil Xvddby2:leny+Yvddby2 "R90")
      schCreatePin(cvid pinMaster "PRECH" "inputOutput" nil Xpre:leny+Ypre "R270")
      schCreatePin(cvid pinMaster "PRECHBAR" "inputOutput" nil Xprebar:leny+Yprebar "R270")
      schCreatePin(cvid pinMaster "CHSH" "inputOutput" nil Xch:leny+Ych "R270")
      schCreatePin(cvid pinMaster "CHSHBAR" "inputOutput" nil Xchbar:leny+Ychbar "R270")
      schCreatePin(cvid pinMaster "TOGGLE" "inputOutput" nil Xtog:leny+Ytog "R270")
      schCreatePin(cvid pinMaster "TOGGLEBAR" "inputOutput" nil Xtogbar:leny+Ytogbar "R270")
```

```
      wireID = schCreateWire(cvid "draw" "full" list(Xout:leny+Yout Xout+0.4:leny+Yout) 0.0625
0.0625 0)
      schCreateWireLabel(cvid car(wireID) (Xout+0.2:leny+Yout) strcat("OUT<" sprintf(a "%d" i)
">") "lowerRight" "R180" "stick" 0.0625 nil)

      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Yd1 Xd-0.2:leny+Yd1) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Yd1) strcat("B1_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Yd2 Xd-0.2:leny+Yd2) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Yd2) strcat("B2_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Yd3 Xd-0.2:leny+Yd3) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Yd3) strcat("B3_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Yd4 Xd-0.2:leny+Yd4) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Yd4) strcat("B4_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Yd5 Xd-0.2:leny+Yd5) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Yd5) strcat("B5_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Yd6 Xd-0.2:leny+Yd6) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Yd6) strcat("B6_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Yd7 Xd-0.2:leny+Yd7) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Yd7) strcat("B7_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)


      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Ydb1 Xd-0.2:leny+Ydb1) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Ydb1) strcat("BB1_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Ydb2 Xd-0.2:leny+Ydb2) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Ydb2) strcat("BB2_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Ydb3 Xd-0.2:leny+Ydb3) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Ydb3) strcat("BB3_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Ydb4 Xd-0.2:leny+Ydb4) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Ydb4) strcat("BB4_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Ydb5 Xd-0.2:leny+Ydb5) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Ydb5) strcat("BB5_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Ydb6 Xd-0.2:leny+Ydb6) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Ydb6) strcat("BB6_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)
      wireID = schCreateWire(cvid "draw" "full" list(Xd:leny+Ydb7 Xd-0.2:leny+Ydb7) 0.0625 0.0625
0)
      schCreateWireLabel(cvid car(wireID) (Xd-0.2:leny+Ydb7) strcat("BB7_ROW<" sprintf(a "%d" i)
">") "lowerRight" "R0" "stick" 0.0625 nil)



)


      schCreatePin(cvid pinMaster "B1_ROW<1:273>" "inputOutput" nil Xd-0.8:Yd1 "R0")
      schCreatePin(cvid pinMaster "B2_ROW<1:273>" "inputOutput" nil Xd-0.8:Yd2 "R0")
```

```
      schCreatePin(cvid pinMaster "B3_ROW<1:273>" "inputOutput" nil Xd-0.8:Yd3 "R0")
      schCreatePin(cvid pinMaster "B4_ROW<1:273>" "inputOutput" nil Xd-0.8:Yd4 "R0")
      schCreatePin(cvid pinMaster "B5_ROW<1:273>" "inputOutput" nil Xd-0.8:Yd5 "R0")
      schCreatePin(cvid pinMaster "B6_ROW<1:273>" "inputOutput" nil Xd-0.8:Yd6 "R0")
      schCreatePin(cvid pinMaster "B7_ROW<1:273>" "inputOutput" nil Xd-0.8:Yd7 "R0")

      schCreatePin(cvid pinMaster "BB1_ROW<1:273>" "inputOutput" nil Xd-0.8:Ydb1 "R0")
      schCreatePin(cvid pinMaster "BB2_ROW<1:273>" "inputOutput" nil Xd-0.8:Ydb2 "R0")
      schCreatePin(cvid pinMaster "BB3_ROW<1:273>" "inputOutput" nil Xd-0.8:Ydb3 "R0")
      schCreatePin(cvid pinMaster "BB4_ROW<1:273>" "inputOutput" nil Xd-0.8:Ydb4 "R0")
      schCreatePin(cvid pinMaster "BB5_ROW<1:273>" "inputOutput" nil Xd-0.8:Ydb5 "R0")
      schCreatePin(cvid pinMaster "BB6_ROW<1:273>" "inputOutput" nil Xd-0.8:Ydb6 "R0")
      schCreatePin(cvid pinMaster "BB7_ROW<1:273>" "inputOutput" nil Xd-0.8:Ydb7 "R0")

      schCreatePin(cvid pinMaster "OUT<1:273>" "inputOutput" nil Xout+0.8:Yout "R180")



;
###########################################################################################################
###########################################################
; ###################################################################################   LAYOUT
#####################################################################
;
###########################################################################################################
###########################################################


cvid = dbOpenCellViewByType("LibSanjoy" "DAC273" "layout" "" "w")
dacid = dbOpenCellViewByType("LibSanjoy" "DAC" "layout" "" "r")

dy = 6.25

Xvdd=20
Xvss=20
Xvddby2=11.325
Xpre=20
Xprebar=20
Xch=20
Xchbar=20
Xtog=205.38
Xtogbar=210.245
Xout=214.76

Xd=-3.6



Yvdd=5.765
Yvss=0.1
Yvddby2=4.685
Ypre=3.98
Yprebar=3.78
Ych=4.45
Ychbar=4.27
Ytog=5.08
Ytogbar=5.08
Yout=5.50


Yd1=0.335
Yd2=0.625
Yd3=0.82
Yd4=1.095
Yd5=1.30
Yd6=1.565
Yd7=1.775
Ydb1=2.07
Ydb2=2.255
Ydb3=2.54
```

```
Ydb4=2.74
Ydb5=3.015
Ydb6=3.22
Ydb7=3.51


for(i 1 273
      leny=(i-1)*dy
      dbCreateInst(cvid dacid "" 0:leny "R0")
      dbCreateLabel(cvid list("M1" "pin") Xvdd:Yvdd+leny "VDD" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M1" "pin") Xvss:Yvss+leny "VSS" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xvddby2:Yvddby2+leny "VDDBY2" "centerLeft" "R0" "roman"
0.10)
      dbCreateLabel(cvid list("M3" "pin") Xpre:Ypre+leny "PRECH" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xprebar:Yprebar+leny "PRECHBAR" "centerLeft" "R0"
"roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xch:Ych+leny "CHSH" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xchbar:Ychbar+leny "CHSHBAR" "centerLeft" "R0" "roman"
0.10)
      dbCreateLabel(cvid list("M2" "pin") Xtog:Ytog+leny "TOGGLE" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xtogbar:Ytogbar+leny "TOGGLEBAR" "centerLeft" "R0"
"roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xout:Yout+leny strcat("OUT<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

      dbCreateLabel(cvid list("M3" "pin") Xd:Yd1+leny strcat("B1_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Yd2+leny strcat("B2_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Yd3+leny strcat("B3_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Yd4+leny strcat("B4_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Yd5+leny strcat("B5_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Yd6+leny strcat("B6_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Yd7+leny strcat("B7_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

      dbCreateLabel(cvid list("M3" "pin") Xd:Ydb1+leny strcat("BB1_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Ydb2+leny strcat("BB2_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Ydb3+leny strcat("BB3_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Ydb4+leny strcat("BB4_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Ydb5+leny strcat("BB5_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Ydb6+leny strcat("BB6_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xd:Ydb7+leny strcat("BB7_ROW<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

)
```

B.7 Cadence SKILL code which generates all analog MUX layout

```
; ALL_ANALOG_MUX : layout

cvid = dbOpenCellViewByType("LibSanjoy" "ALL_ANALOG_MUX" "layout" "" "w")
muxid = dbOpenCellViewByType("LibSanjoy" "ANALOG_MUX" "layout" "" "r")
muxextra = dbOpenCellViewByType("LibSanjoy" "ANALOG_MUX_EXTRA" "layout" "" "r")


dy=6.25
dymux=100

Xvdd=6.7
Xvss=10
Xvref=8.4
Xin=-1.60
Xout=20.2
Xsel1=4.085
Xselb1=4.37

Yvdd=30.15
Yvss=26.5
Yvref=42.3
Yin=0.10
Yout=30.2
Ysel1=0
Yselb1=0

for(i 1 18
      lenymux=(i-1)*dymux
      if(i==18 then
               dbCreateInst(cvid muxextra "" 0:lenymux "R0")
               else
               dbCreateInst(cvid muxid "" 0:lenymux "R0")
      )

      dbCreateLabel(cvid list("M1" "pin") Xvdd:Yvdd+lenymux "VDD" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M1" "pin") Xvss:Yvss+lenymux "VSS" "centerLeft" "R0" "roman" 0.10)
      dbCreateLabel(cvid list("M2" "pin") Xvref:Yvref+lenymux "V_REF" "centerLeft" "R0" "roman"
0.10)
      dbCreateLabel(cvid list("M2" "pin") Xout:Yout+lenymux strcat("OUT<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)

      dbCreateLabel(cvid list("M3" "pin") Xsel1:Ysel1+lenymux "SEL<1>" "centerLeft" "R0" "roman"
0.10)
      dbCreateLabel(cvid list("M3" "pin") Xsel1-1.65:Ysel1+lenymux "SEL<2>" "centerLeft" "R0"
"roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xsel1-1.65*2:Ysel1+lenymux "SEL<3>" "centerLeft" "R0"
"roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xsel1-1.65*3:Ysel1+lenymux "SEL<4>" "centerLeft" "R0"
"roman" 0.10)

      dbCreateLabel(cvid list("M3" "pin") Xselb1:Yselb1+lenymux+1 "SELB<1>" "centerLeft" "R0"
"roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xselb1-1.65:Yselb1+lenymux+1 "SELB<2>" "centerLeft"
"R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xselb1-1.65*2:Yselb1+lenymux+1 "SELB<3>" "centerLeft"
"R0" "roman" 0.10)
      dbCreateLabel(cvid list("M3" "pin") Xselb1-1.65*3:Yselb1+lenymux+1 "SELB<4>" "centerLeft"
"R0" "roman" 0.10)
)


for(i 1 273
      leny=(i-1)*dy
      dbCreateLabel(cvid list("M1" "pin") Xin:Yin+leny strcat("IN<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
      )
```

B.8 Cadence SKILL code which generates termination check circuit's partial layout

```
; TERMCHECK : layout

cvid = dbOpenCellViewByType("LibSanjoy" "TERMCHECK" "layout" "" "w")
unitid = dbOpenCellViewByType("LibSanjoy" "UNIT_TERMCHECK" "layout" "" "r")

dx1=1.44
dx2=11

Xlab=0.325
Ylab=0.80


for(i 1 273
      j=mod(i-1,4)
      k=(i-1)/4
      lenx=k*dx2+j*dx1
      dbCreateInst(cvid unitid "" lenx:0 "R0")
      dbCreateLabel(cvid list("M4" "pin") Xlab+lenx:Ylab strcat("CN<" sprintf(a "%d" i) ">")
"centerLeft" "R0" "roman" 0.10)
)
```