POLITECNICO
DI TORINO

Master Degree in Electronic Engineering

VLSI architecture of a Multiple-Error-Correction Polar-Codes decoder

Politecnico di Torino
Turin, Italy

Supervisor:
Prof. Maurizio Martina
Co-supervisor:
Prof. Guido Masera

Student:
Nicola Antonio Travaglini

April 4, 2019

Contents

1 Introduction

The information sent through a digital communication system is affected by the noise introduced by the channel. As a consequence, channel encoding is exploited to protect the message from the external interferences by using redundancy. This, however, increases the complexity, the latency and the bandwidth required.
In 1948, Claude Shannon proved in [1] the existence of the so-called **channel capacity**, a limit rate at which information can be reliably transmitted over an information channel. Since then, several capacity achieving codes have been presented, like the LDPC codes, however, none of them had an explicit construction. This went on until 2009, when Arikan introduced in [2] the Polar Codes, which are the first codes proved to achieve channel capacity when the code length tends to infinity and they have been receiving attention since they have been chosen as an official channel coding in the 5G standard.
The idea behind the Polar Codes is the channel polarization: a noisy channel is transformed into two sets: one with a lower noise level, the other with a higher noise level. By recursively applying this technique over the resulting channels, the difference in reliability between good channels and bad channels gets deeper and deeper. In the end, when the number of synthesized channels is enough large, almost all of them tend to two opposite ends: no-noise or noisy behaviour.
This scheme is exploited in the following way: **information** bits are sent over the noiseless channels, while fixed bits (called **frozen** and usually set to zero) are assigned to the noisy ones. The positions of the frozen bits are known by the decoder that, using the **Successive Cancellation** (SC) algorithm, is able to correct the errors due to the channel noise.
Even if polar codes have, theoretically, many positive properties, their actual implementation presents some drawback: practical error rate values are achieved when the code length is very large (e.g. 2^{20}) and the SC decoder will introduce high latencies due to this. Moreover, SC decoders provide FER values inferior to other codes with similar length.
Several techniques, like the SC-List [3], have been developed to reduce this performance loss at the price of increased complexity. In [4], however, an alternative approach was proposed based on the performing of many decoding attempts by flipping the bits that turned out unreliable during the first decoding process. This method is called **Successive Cancellation Flip** (SCF) and it keeps the low memory requirement of the original SC. A further evolution has been introduced in [5] called **Progressive Bit Flipping**, where a set of unfrozen bits, that are more likely responsible for the first error, is found and then used to correct multiple errors.
In this thesis, first the SC, SCF and PBF algorithm have been software implemented via C code, then a new simplified method, called PBF-constrained, has been derived from the PBF and finally a VLSI architecture, able to correct up to two errors, is proposed for the PBF-C. The architecture presented in this thesis is the first one proposed in literature (to the best of my knowledge) able to correct more than one error with an SCF approach.

2 Literature Review

The generic digital communication system is composed of several elements as shown in the figure below.

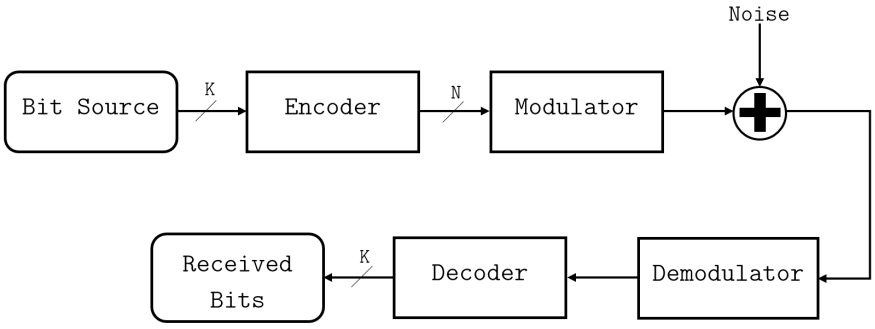


Figure 1: Digital communication system block scheme

At the beginning of the chain, a bit source sends a stream of K bits, then an encoder performs the channel encoding to limit the effect of the channel disturbances. By doing so, the number of bits transmitted is increased from K to N, but the reliability of the system is also increased. Therefore, while K bits contain the information sent, the code length will be on N bits. The ratio between K and N is called **code ratio** and it can be seen as a measurement of the redundancy introduced by the encoder:

$$R = \frac{K}{N}$$

The lower the rate R is, the higher is the redundancy introduced and the error correction capability of the system.
The binary stream is then passed from the encoder to a modulator which adapts the symbols to be transmitted over the communication channel. Many types of disturbances occur over the channel, leading to the introduction of some noise over the transmitted signal. The signal, then, undergoes the demodulator, which transforms the received symbols into **log-likelihood ratios** (LLR). The LLRs represent the logarithm of the ratio between the probability that a transmitted bit x_i is 0 given the received symbol y_i, over the probability that the same transmitted bit is 1:

$$LLR_i = \log\left(\frac{P(y_i|x_i = 0)}{P(y_i|x_i = 1)}\right)$$

The LLRs are then passed to the decoder which is able to provide the K received bits, which should be a faithful representation of what was transmitted.

2.1 Successive Cancellation

As explained in [2], a polar code PC (N,K) of code length $N = 2^n, n \in \mathbb{Z}^+$ and rate $R = K/N$ is a linear block code that divides N bit-channels in K reliable ones and

$N - K$ unreliable ones. Information bits are transmitted on the reliable channels, while fixed values, usually zero and known to both transmitter and receiver, are sent on the unreliable channels.
The recursive process with which a total of N channels are obtained can be represented by a binary matrix multiplication as $\mathbf{x} = \mathbf{u}\mathbf{G}^{\otimes n}$, where $\mathbf{u} = \{u_0, u_1, ..., u_{N-1}\}$ is the input vector, $\mathbf{x} = \{x_0, x_1, ..., x_{N-1}\}$ is the codeword, and the generator matrix $\mathbf{G}^{\otimes n}$ is the n-th Kronecker product of the polarization matrix $\mathbf{G} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.
As an example, if the code length is $N = 8$, the generator matrix will be:

$$\mathbf{G}^{\otimes 3} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The operations performed by the SC decoding algorithm can be seen as a binary tree search, where the tree is explored from the highest stage S = n, with priority given to the left branch. Fig.2 portrays an example of SC decoding tree, for a PC(8,4).

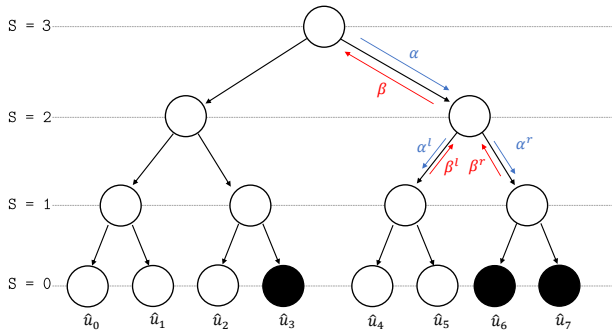


Figure 2: SC decoding tree for a PC (8,4)

Let us consider the first node at the stage S=2. This node sends a vector of LLRs $\alpha = \{\alpha_0, \alpha_1, ..., \alpha_{2^S-1}\}$ to its left child. Then, the value of the stage is decreased to S=1, the previous child node becomes the active one and sends to its left child the message $\alpha^l = \{\alpha_0^l, \alpha_1^l, ..., \alpha_{2^{S-1}-1}^l\}$. At S=0, the node uses the vector received to decode the first bit, $\widehat{u}_0$, then it sends this bit to its parent node at stage 1. Now the active node will transmit $\alpha^r = \{\alpha_0^r, \alpha_1^r, ..., \alpha_{2^{S-1}-1}^r\}$ to its right child which will decode the second bit $\widehat{u}_1$. At this point, the first node at the stage S=2 will receive a vector of partial sums from its left child and will use it to compute the vector α^r to be sent to the right child at stage 1.
To sum up, at the root node (in this example S=3) the LLRs are initialized as the

channel LLRs y_0^{N-1}, each node at stage S computes the left and right LLR vectors to be sent to its child nodes as

$$\alpha_i^l = sgn(\alpha_i)sgn(\alpha_{i+2^{S-1}})min(|\alpha_i|, |\alpha_{i+2^{S-1}}|) \tag{1}$$

and

$$\alpha_i^r = \alpha_{i+2^{S-1}} + (1 - 2 \cdot \beta_i^l)\alpha_i \tag{2}$$

Then, the nodes receive the partial sums from their left child node as:

$$\beta_i = \begin{cases} \beta_i^l \oplus \beta_i^r, \text{ if } i \leq 2^{S-1} \\ \\ \beta_i^r, \text{ otherwise} \end{cases} \tag{3}$$

where $\oplus$ is the bitwise XOR operation, and $0 \leq i < 2^S$. At the leaf stage, the values of the estimated bit vector $\widehat{u}_0^{N-1}$ are computed as:

$$\beta_i = \begin{cases} 0, \text{ when } \alpha_i \geq 0 \text{ or } i \in \mathbf{F}; \\ \\ 1, \text{ otherwise} \end{cases} \tag{4}$$

where $\mathbf{F}$ represents the set of frozen bits.

The computational complexity of the SC decoding is $O(N \log N)$ since there are $N \log(N + 1)$ nodes and each node only needs to be activated once, while the memory complexity can be reduced to O(N).

2.2 Error Propagation in SC decoding

While infinite-length are capacity achieving, when the code length is finite, the channels are nor completely noisy nor completely noise-free. Therefore, during the SC decoding, errors can still happen. In particular, the wrong bit decisions can be caused by channel noise or by error propagation due to a previous erroneous bit decisions. However, it is possible to state that the first erroneous decision must be caused by the channel noise, since no previous error is possible at that point, and it leads to erroneous partial sums which can corrupt the internal LLR values computed in the following stages.
In [4] it was shown that an SC-Oracle, avoiding all wrong decisions caused by the channel, is able to strongly reduce the noise disturbances improving the SC decoding performance. Moreover, it was also illustrated that a failed SC process is mostly due to a just one erroneous decision caused by the channel noise.

2.3 SC-Flip

The goal of SC-Flip decoding [4] is to identify and correct the first error that occurs during SC decoding without the aid of an oracle.
A cyclic redundancy check (CRC) code with a C-bit remainder is used to encode the information bits. At the end of the SC decoding, the estimated codeword is

considered correct if the CRC check is successful, otherwise, if the CRC fails, a number T of LLRs with the smallest magnitude, representing the bit estimations with the lowest reliability, are stored and sorted. Then, the SC decoding is repeated, but the bit associated with the smallest LLR is flipped. If the CRC fails again, a new SC attempt is performed, flipping the index corresponding to the next smallest LLR. The algorithm stops when the CRC gives a positive result or after T unsuccessful attempts.
In the worst-case scenario, the latency of an SC-F decoder is T times the one of the SC decoder.
The results in [4] show that the performance of the SC flip decoder with T = 32 is almost identical to that of the SC list decoder with L = 2, but with half the memory complexity.

2.4 Progressive Bit Flipping

The SC-Flip shows that the error correction capability of an SC-decoder can be improved by correcting the first wrong bit-decision by flipping it. Other works, like [6] improved the original SC-Flip algorithm by dynamically detecting the positions to be flipped in order to correct even more than one error. However, the first flip is taken from the Unfrozen set, which could be extremely large for long polar codes. A new approach was proposed in [5], where the authors suggested that, by investigating the distribution of the first erroneous bit decision during the SC decoding, it is possible to narrow down the search scope of possible flips to an unfrozen subset CS, called *critical set*, which is much smaller than the unfrozen set. As a consequence, the decoder only needs to consider CS for the flipping position, and, since there could be other errors besides the first one, the CS set method can be used to identify the positions of the following errors. By correcting nested errors, the PBF is able to compete with the SCL decoder.

2.5 SC Architecture Implementations

A first SC decoder implementation was described by Arikan in [2] with a butterfly structure. Even if many aspects for the hardware implementation were not discussed, like the control unit or the resource sharing, the author suggested that the decoder could be implemented with $Nlog_2N$ processing elements with N registers to store the partial results and other N registers would be used to store the channel LLRs. A structure like this would require $2N - 2$ clock cycles to complete the decoding of one vector. Of course, this architecture can be optimized by increasing the resource sharing, like the processing elements.
A first observation regards the fact that at stage S, only 2^S processing elements could be employed in order to get all the LLRs needed to be sent to the stage below. With this idea, in [7] a tree architecture with the same throughput and scheduling of the butterfly architecture, but employing a lower number of Pes and registers, is

introduced. Moreover, by noticing that the maximum number of Pes required to be used concurrently is N/2, the authors propose a second architecture called line decoder, similar to the tree one, but with a lower complexity due to the merging of some PEs.
In a following article [8], the same authors exploit a second observation: since the N/2 Pes of the line architecture are used all together only in two clock cycles, by using a lower number of them it is possible to achieve a great reduction in complexity with a small increase in the number of clock cycles required to complete the operation. This architecture is called **semi-parallel**. In these schemes, the PEs implement the hardware friendly functions (eq.1 and eq.2).
A limit of all these implementations is given by the partial sum unit, which requires a large area and affects the maximum frequency. These problems grow with the code length N.
In [9] a new scheme, HPPSN, was proposed that can be integrated with the semi-parallel scheme without requiring extra clock cycles and providing area efficiency since only the LLRs and partial sum memories dimension will depend on the code length N.

3 Software Overview

The first step of the present study involved the derivation of a software implementation of the SC decoder.
This code is used in a template for the simulation of a transmitter and a receiver, where the encoder and the decoder were originally missing. Many functions developed for the SC decoder software were then used to implement the PBF software.
The BER and FER curves obtained from the simulation of the decoders will be shown after the overview of each code.

3.1 SC Decoder Software Implementation

3.1.1 Main

The Main code reads two text files to get the initial inputs: the first one (located in the *pccodes* folder) contains the indexes to be used as the information set, sorted based on their reliability from best to worst for a given noise standard deviation, the second text file (*input* in the dat folder) stores other information like the number of frames to be simulated, the initial SNR value and its step, the seed for the random generators, and many other parameters.
Following the variables declaration, the first text file is opened and the value of N (length of the code) is acquired. It is asked to the user to provide a value for K (dimension of the information set I) and the decoding scheme to be used:

- 1: SC;
- 2: SC-OA;
- 3: SC-Flip.

In the case of SCF, the user has to give also the maximum number of tries T and the length of the CRC *CRC_length*.
At this point, the vector I is dynamically allocated and filled with the first K values from the text file. As for the SCF, the following CRC_length values are stored in the vector R which will provide the indexes to store the remaining of the CRC operation. Now, the text file is closed, the vector I (and eventually R) is sorted in ascending way, the vector Frozen is filled with the indexes from 0 to N not included in I (and R) and the **init_CRC** function creates the polynomial for the CRC based on CRC_length.
At this point, the loop for the frame generation starts: a vector (with dimension K) is generated through an LFSR. This vector represents the uncoded information and it is passed to the **PC_encoder** function that provides the encoded N-bit vector (coded). The noise samples are generated from the function **Gaussian**, then *coded* is BPSK modulated and summed to the noise samples multiplied by the standard deviation. From this signal (*received*) the channel LLR parameters are obtained in fixed point (with 2 bits for the decimal part). The **SCF_decoder** stores in the

vector *dec* the K decoded bits. The following functions count the errors and save the results in a text file.

3.1.2 SC_tools.c

All the functions related to the SC decoding have been collected in the SC_tools library.

3.1.3 PC_encoder

void* **PC_encoder**(int* uncoded, int N, int* I, int K, int* input_vector, int* poly, int*R, int CRC_length, int dec_mode, int* coded)

- *uncoded* is the K-bit vector provided by the LFSR;
- *N* is the code length;
- *I* is the information set;
- *K* is the dimension of I;
- *input_vector* is filled by this function, it has the value of *uncoded* in the information indexes (and the remainder of the CRC in the next *CRC_length* most reliable indexes);
- *poly* is the $(CRC_length + 1)$-bit vector used in the CRC computation;
- *R* is the set containing the indexes related to the CRC remaining;
- *CRC_length* is the dimension of R;
- *dec_mode* specify the decoding scheme;
- *coded* is the encoded vector provided by this function. It is obtained multiplying input_vector and the Kronecker matrix G with dimensions NxN.

First, *input_vector* is obtained from the *uncoded* vector. In case of SCF, the function **CRC_function** stores in it the remainder too. After this, the G matrix is initialized as a NxN matrix filled with zeros and, then, its top left corner is assigned as T2 = $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. The idea is to cyclic fill the G matrix with 2 quadrants equal to the top left corner and one equal to zero. The loop ends when the G matrix is completely filled. An example of G for N = 8 is portrayed below, where the dark-blue quadrant is copied into each of the two lighter-blue quadrants beneath it.

1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	0
1	1	0	0	1	1	0	0
1	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1

It is possible now to compute the *j-th* element of *coded* as the product between *input_vector* and the G matrix by summing the rows of a *j-th* column of G skipping the rows i if $input\_vector[i] = 0$.

3.1.4 Init_CRC

void **init_CRC**(int* poly, int length)

- poly is passed as a zero vector, the function stores in it the bits to be used in the CRC operation;
- length is the CRC length. poly has dimension length+1.

Based on the value of *length*, a case selects the polynomial value as a hexadecimal number. A loop converts this number in binary and saves it in the poly vector leaving its first bit empty. This first bit is later set to 1. As an example, if the length of the CRC is 4, the corresponding polynomial value is 0x3 and the value saved in *poly* by the function is 1 0 0 1 1.

3.1.5 CRC_function

int **CRC_function**(int* Signal, int* poly, int* I, int K, int* R, int length, int encode_or_decode)

- Signal provides the vector on which the CRC is performed;
- poly is the polynomial used in the CRC operation;
- I is the information set;
- K is the dimension of I;
- R contains the indexes of the remainder bits;
- length is the length of the CRC and the dimension of R;
- encode_or_decode = 0 → encoder side: encode the remainder bits in the input_vector (provided as *Signal*);
- encode_or_decode = 1 → decoder_side: check if the CRC fails to retrieve the remainder previously computed from u_hat (which contains the N decoded bits and it is provided as Signal).

When this function is called from the encoder (*encode_or_decode*= 0) it inserts the remainder of the CRC in *input_vector*, which is given as an input with the name *Signal*.
First, the K information bits are taken from the vector and stored into an auxiliary one called *Signal_copy* which is padded with *length* 0s at the end (the dimension of *Signal_copy* is K+length). The CRC computation is implemented with a loop that first skips sequences of zeros in the *Signal_copy*, then, when a 1 is found, it stores the result of the bitwise XOR between *length+1* bits of *Signal_copy* (starting from the first index storing a 1) and the polynomial vector.
At the end of the loop, the remaining of the CRC (given by the last *length* bits of *Signal_copy*) is stored in the input vector in the indexes provided by the vector R.
When CRC_function is called from the decoder, it takes the K information bits from the hard decoded input (provided as *Signal*) and saves them in the first K slots of *Signal_copy*, then it extracts the remainder bits from *Signal* and stores them in the last *length* positions of *Signal_copy*.
In the end, it returns 0 if the CRC between *Signal_copy* and the polynomial fails, 1 otherwise.

3.1.6 SC_decoder

void* **SC_decoder**(int N, int* Y, int* I, int* Frozen, int* reference, int* u_hat, int* alpha_0, int dec_mode, int k, int max_val_int, int min_val_int)

- N is the length of the code;

- Y is the array containing the LLRs of the channel;
- I is the array containing the indexes corresponding to the location of information bits;
- Frozen is the frozen set
- reference is the input vector (of length N) prior to the PC encoding;
- u_hat is the hard decision vector (N elements)
- alpha0 contains the LLRs of the LEAF stage (stage 0). It passed to **LLR_-sorter** in SCF decoding;
- dec_mode specifies the decoding mode: 1 for SC, 2 for SC-OA, 3 for SC-Flip;
- k is the position of the bit to be flipped (in the first call of the function, $k = -1$ to avoid flips);
- max_val_int and min_val_int are the maximum and minimum values of the LLRs.

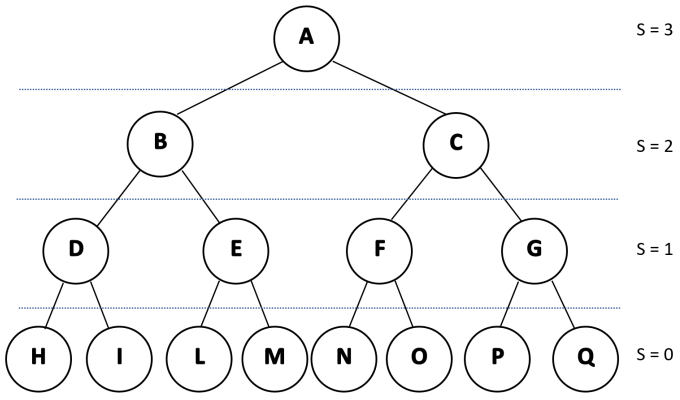
After some variable declarations, the highest stage is computed as $S = \frac{logN}{log2}$ and some vectors and matrices are dynamically allocated:

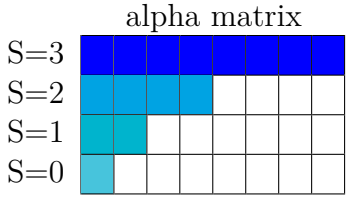
alpha is a matrix with S+1 rows and N columns. It stores the soft LLRs of each stage. Since α_{left} and α_{right} are saved in the same slots, the top stage requires N columns, the one below it requires N/2 columns, and so on until the stage 0 (leaf node) that requires just 1 bit.

beta is a matrix with S+1 rows and N/2 columns. It stores the hard LLR parameters of each stage;

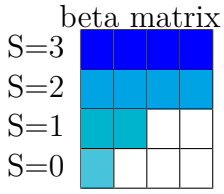
flag is a vector with S+1 elements used to tell if a node at a certain stage has already been visited arriving from the bottom of the tree.

To better understand how these matrices and vectors are used, let us consider the example of the tree associated with a PC with N = 8.





The first row of the alpha matrix (corresponding to the root node A) is filled with the 8 LLRs received from the channel (the input Y), then the node A computes and propagates the 4 α_{left} to the node B in the stage below. These parameters are saved in the 4 slots provided by the second row of the alpha matrix. When the root node is reached again and it computes the 4 α_{rigth} to be sent to the node C, they are stored in the same slots where the 4 α_{left} of the second stage were saved. The soft LLRs stored in the row corresponding to S = 0 (one slot) are saved in the vector *alpha_0*, so that they can be sorted in case of SC-F decoding.



Let us consider that the decoding process is now arrived at the node C. Considering the hard-decision bits with the lower case corresponding to the leaf node that computed them (h is produced by H and so on) the beta matrix is storing at this point the following values:

S=3	h+i+l+m	i+m	l+m	m
S=2	h+i	i	l+m	m
S=1	l	m		
S=0	m			

Table 1: Example of the content of the beta matrix

Since the stage 3 will not be reached a second time from the bottom of the tree, there is no need to allocate memory to store its β_{right} values (which will not be computed). The values stored in the S = 0 row are saved in the *u_hat* vector.

In the alpha and beta matrices, the white cells are allocated but not used.

About the **flag** vector, flag[S] stores 0 if the node has not been visited, while 1 means that the next node to be visited is in the stage above (if the decoder is in D and flag[1] = 0, the next node will be I, otherwise B).

The **SC_decoder** function is derived from the tree diagram and it is based on a state machine with 4 states:

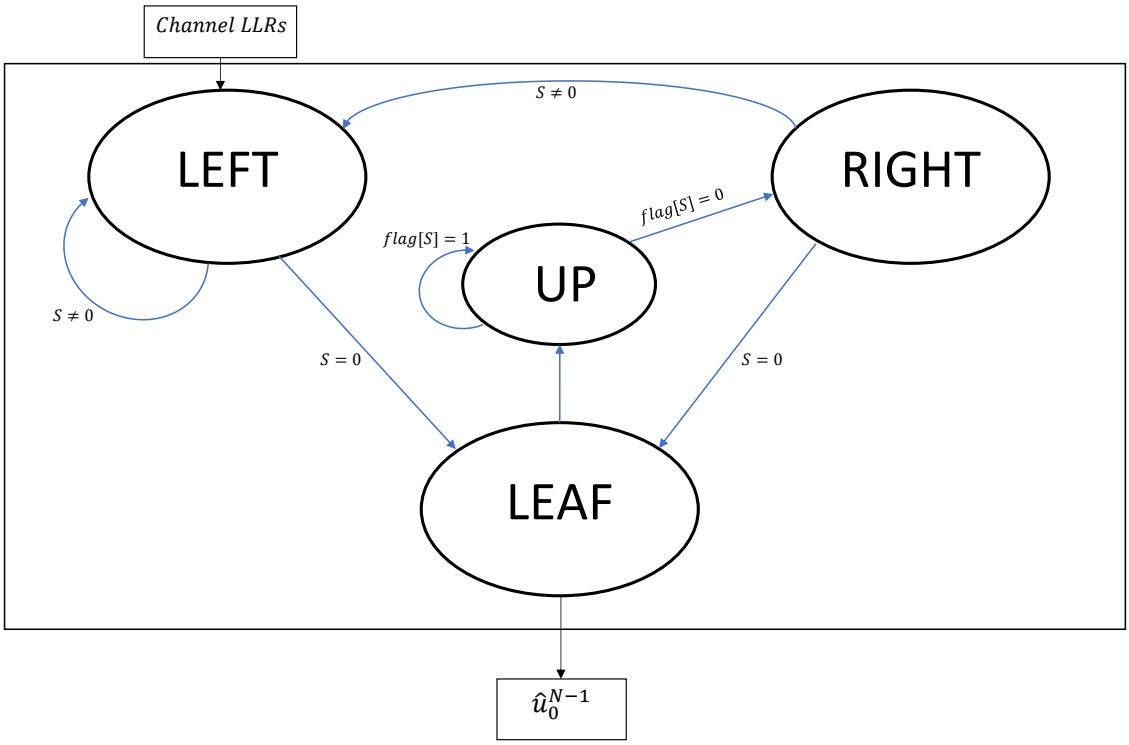


Figure 3: FSM used in SC_decoder

- **LEFT:** is the initial state. It computes and propagates α_{left} to the stage below, then it decreases S by one unit. At this point, the next state will be LEFT if $S \neq 0$, LEAF if $S = 0$;

- **RIGHT:** computes and propagates α_{right} to the stage below, then it decreases S by one unit. The next state will be LEFT if $S \neq 0$, LEAF if $S = 0$;

- **LEAF:** reached when $S = 0$. It computes the hard decision bits based on the received soft LLRs, fills the row of the beta matrix corresponding to the first stage and, in case of SC-OA or SCF, performs the bit-flips. The next state is always UP;

- **UP:** increases S by one unit, then check the content of flag[S]. If it is 0, the β_{left} values are computed and stored in the beta matrix and the next state is set to RIGHT. If flag[S] is 1 the β_{right} parameters are evaluated then the next state is set to UP. The hard LLRs are evaluated only for the stages above 1 and the content of flag[S] is flipped after it has been checked.

When the N-th hard decision bit has been evaluated, the loop in which the FSM evolves is stopped.

3.1.7 LLR_sorter

void* **LLR_sorter**(int* alpha, int* U, int T, int N, int K, int* I, int length, int*R)

- alpha is the vector containing the soft LLRs obtained at the leaf stage during the SC decoding;
- U is the vector that will store the indexes of smallest soft LLRs corresponding to information bits;
- T is the maximum number of tries in the SCF decoding. The dimension of U is T;
- N is the length of the PC;
- K is the dimension of the information set;
- I is the information set;
- length is the length of the CRC;
- R is the set containing the indexes with the remainder of the CRC.

The function **LLR_sorter** is used to sort the soft LLRs obtained by a previous SC decoding.
A structure is defined with two fields:

alpha_c: stores the copy of an LLR associated with an information (or remainder) bit;

index: stores the index of the LLR parameter.

At this point, the absolute value of the LLRs are sorted in ascending order and their index field is moved accordingly. In the end, the vector U is filled with the first T indexes which refer to the smallest LLRs.

3.1.8 SCF_decoder

void* **SCF_decoder**(int N, int K, int* Y, int* I, int* Frozen, int*R, int*poly, int* reference, int dec_mode, int T, int CRC_length, int* decoded, int max_val_int, int min_val_int)

- N is the length of the code;
- K is the length of the information set I;
- Y is the array containing the LLRs of the channel;
- I is the array containing the indexes corresponding to the location of information bits;
- Frozen is the frozen set;
- R is the set containing the indexes with the remainder of the CRC;

- poly is the polynomial used in the CRC computation;
- reference is the input vector (of length N) prior to the PC encoding;
- dec_mode specifies the decoding mode: 1 for SC, 2 for SC-OA, 3 for SC-Flip;
- T is the maximum number of tries in the SCF decoding;
- CRC_length is the length of the CRC;
- decoded will contain the K hard decision bits corresponding to the information bits;
- max_val_int and min_val_int are the maximum and minimum values of the LLRs.

This is the function called in the Main to obtain the decoded vector. First, it calls the **SC_decoder** function providing as flipping index $k = -1$, so to avoid flips in the first call. Then, in case of SCF (dec_mode = 3) the **CRC_function** is called comparing the reference (input_vector) and the hard decision vector u_hat obtained from the SC_decoder. If the CRC fails (and $T > 0$) the function **LLR_sorter** is called and stores the T smallest LLRs in the vector U. Then, the SC decoding is repeated up to T times, flipping the bit obtained in the index k provided by the vector U. After each decoding, the CRC operation is applied and if it succeeds the loops of the T SC decoding is stopped.
In the end, the vector *decoded* is filled with the K hard decision bits corresponding to the information bits.

3.1.9 Tools.c

The tools.c file contains some minor functions used to improve the readability of the code:

- int **min_int**(int A, int B)→ returns the minimum value among the integers A and B;
- int **sign_int**(int A) → returns the sign of the integer A;
- int **boxplus**(int a, int b) → represents the boxplus operator used to compute α_{left};
- void* **quick_sort**(int sx, int dx, int* vector) → sort the vector. It is called with $sx = 0$ and

  $dx = size\ of\ the\ vector - 1$.

3.1.10 Results

The simulations have been carried out by using the fixed point format with 2 bits to represent the radix part, 4 bits for the channel LLRs and 6 bits for the internal LLRs. The following curves represent the results obtained by simulating 10^5 frames with N = 1024 and K = 512.

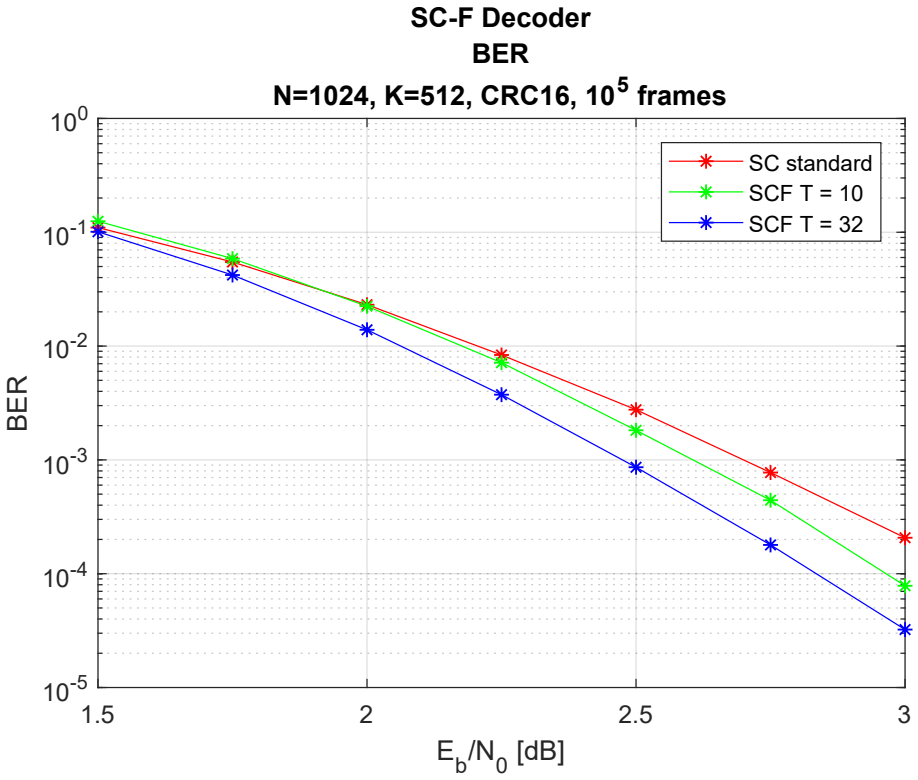


Figure 4: BER curves of the SC and SCF decoders

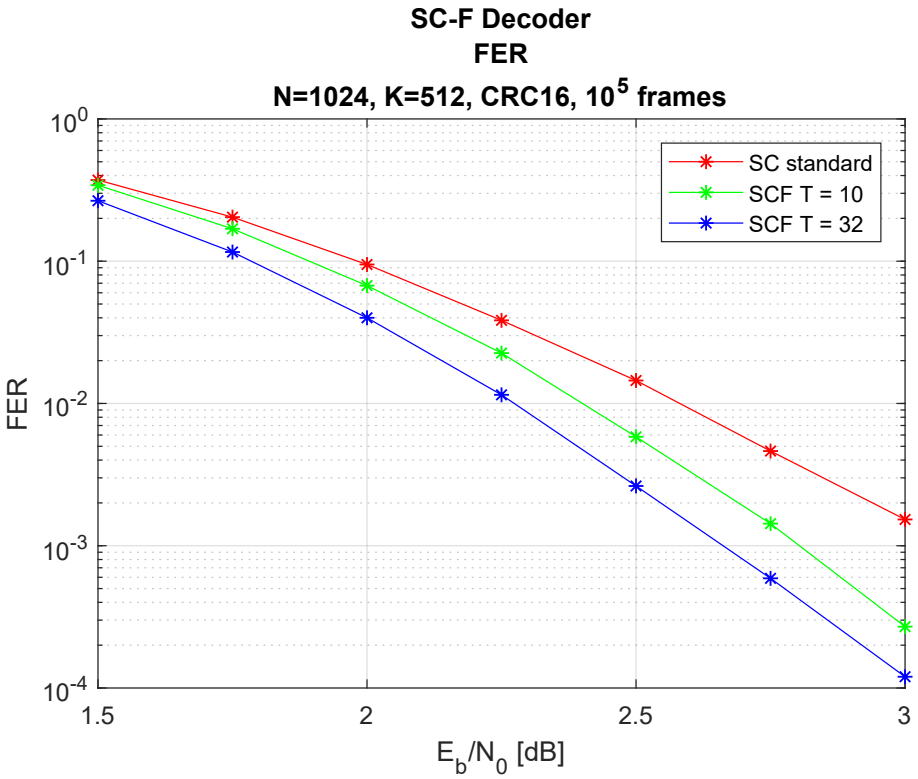


Figure 5: FER curves of the SC and SCF decoders

3.2 PBF Decoder Software Implementation

The PBF algorithm described in [5] shows that by using the concept of critical sets it is possible to locate and correct more than one error. Therefore, a software implementation of this decoding scheme has been derived with a maximum depth equal to four levels, which means that 4 errors are corrected at most.
Since most of the code includes parts developed for the SCF software, only the new implementations will be discussed.

3.2.1 Main

As seen for the SCF software, the Main code requires to read two text files: the first one contains the indexes to be used as the information set, the second text file stores other information like the number of frames to be simulated, the initial SNR value and its step, the seed for the random generators, and other parameters.
This time, the software asks to provide the number K of information bits, the maximum depth of the decoder (from 0 to 4) and the number of bits to be used for the CRC computations. The only difference now is in the decoder function **SCPBF_-decoder**.

3.2.2 Custom types

In order to implement the PBF decoding, several types have been created. This section explains their meaning and use.

- **type_flip**: used to create arrays where the elements are arrays of 4 integers. The idea is that one element of type type_flip contains the flip for each level to reach a certain node. As an example:
  type_flip vect $= \{7, 9, 12, /\}$
  means that the decoder flips the seventh bit for the level 1, the ninth for the second level and the twelfth for the third stage. The last element of the example is undefined: it will contain the fourth and last flip allowed for this implementation.

- **type_node**: represents a node of the PBF decoding tree. It is a structure with two fields: *dimension*, which tells how many children the node has, and *Child*, an array of type_flip. As an example let us consider the binary tree corresponding to a 6/16 PC and the corresponding PBF tree:

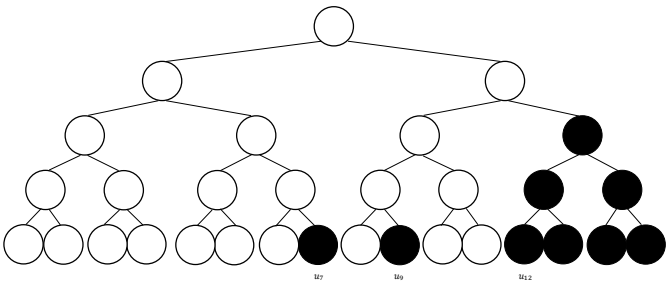


Figure 6: Binary tree of a PC with N = 16

In the binary tree in fig.6, the black leaves represent unfrozen bits. A node in a higher stage is black if its children are black. This is used to better identify the critical set, which in this example is $\{7, 9, 12\}$. Therefore, as shown in fig.7, the level 0 of the PBF tree is linked to three nodes and each of them has its own critical set.

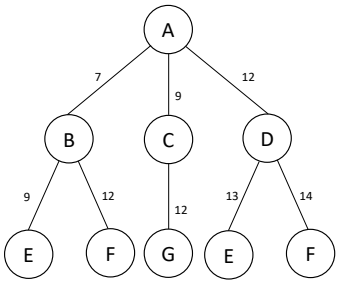


Figure 7: PBF tree of a PC with N = 16

The level 0 is represented by just the node A, which is described as a type_node type in the following way:

DIM A = 3											
7	/	/	/	9	/	/	/	12	/	/	/

Figure 8: Node A of the PBF tree

3.2.3 SCPBF_decoder.c

This function implements the PBF decoding scheme.
After the variables declarations and the vectors allocations, a first standard SC decoding is performed: if the CRC fails the PBF algorithm starts. First, a counter used to keep track of the number of SC performed is increased. This variable is used to compute the mean value of the LLRs required by the pruning functions **Enochild** and **Enoselect**. To do so, the absolute values of the LLRs are accumulated in the acc_LLR vector by the **Accumulate** function, therefore, after the increasing of SC_-count variable, the LLRs are stored in acc_LLR.
Now, the critical set is evaluated: the function **Critical_Set** writes in the vector C_S the indexes belonging to the critical set. After this, the elements of C_S are properly stored in the member Child of the type_node array Node. At level 0, this array contains just one element (one node), like shown in fig.8.
The nodes are visited according to the U vector, which contains the sorted index of the nodes according to a certain metric (ascending order of $\frac{|LLR_i|}{\mu_i}$). At level 0, the U vector contains just one index set to 0.
A variable (exit_loop) to control the external while loop is set to 1: when a future CRC succeeds or all the nodes of the maximum level have been evaluated with no success, it is set to 0 to exit from the while loop.
After the declaration of the while loop, the level is increased, then, if this part is reached by higher levels, the set of nodes and the U vector are updated for the new level.
An external for loop is used to visit all the elements of the Nodes array (when this loop is reached for the first time, there will be just one node), however, since we want to visit the nodes of a certain level according to the metric previously explained, the index of the loop is used to extract the index of the U vector associated with the node with progressively increasing metric.
At this point, for each element of the Child member of the node selected, the SC decoding is applied: however, the difference with the standard SC scheme, is that now all the indexes provided by the Child member are flipped. As an example, if the actual level is 2 the array Node provided by the previous level is:

DIM B = 2								DIM C = 1				DIM D = 2							
7	9	/	/	7	12	/	/	9	12	/	/	12	13	/	/	12	14	/	/

Figure 9: type_node array Node built in level 1

Then, the SC_decoder function takes as an input one element of the Child member of a node, e.g considering the first node of the array, it has two elements in the Child field, so SC_decoder will take the array $\{7, 9, /, /\}$, flips the seventh and ninth bits, then, if the CRC fails, it will work with $\{7, 12, /, /\}$.
After each SC decoding, the SC_count and the acc_LLR vector are updated and the CRC is performed. If it fails, before considering the following child node, the child nodes for the next level must be created. In order to reduce the number of nodes generated, the functions Enochild and Enotselect are used.
If the result of Enochild is false, the node is allowed to create child nodes, so the Critical_Set function is used on the visited node and its critical set is written in the vector C_S. With the Enotselect function, some vales from the C_S vector are discarded and the resulting critical set is saved in the new_CS vector. Now, an auxiliary array of type_node, Node_aux is used to store these new nodes: the previous flips are copied from the Node array, the new flip is taken by the new_CS vector.
When all the nodes of a certain level have been visited, the algorithm copies Node_aux in Node and frees the content of Node_aux, then the level is increased and the operations are done again. This continues until a CRC succeeds or all the elements of the array Node have been visited on the maximum level.

3.2.4 PBF_tools

This library contains a set of functions used by the PBF_decoder.

3.2.5 Critical_Set

- *Node*: pointer to a node;
- *C_S*: vector in which the critical set of the node will be stored;
- *N*: length of the PC;
- *Unfrozen*: indexes corresponding to the unfrozen bits;
- *level*: actual level in the decoding process, it is used as an index to fill the child member of the node;
- *flip*: index flipped in the previous SC, the critical set starts from the index next to it.

This function fills dimension and child members of the node provided as input and writes its critical set in the C_S vector. Let us consider, as an example, the PC code with rate R = 6/16 seen in fig.6. The idea to derive its critical set relies on the possibility to discern its sub-blocks. To do so, the tree is represented as the following table:

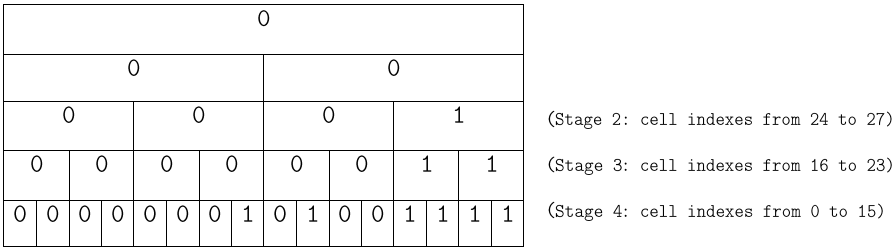


Figure 10: Binary tree sub-block representation

However, the implementation of this table has been done as a 1D array.
In order to fill this array, it is first allocated and set to store all zeros, then, an index in the last row (stage 4 of the example) is filled with 1 if its index belongs to the Unfrozen vector and if it is greater than the flip (in this way the indexes prior to the flip are seen as frozen).

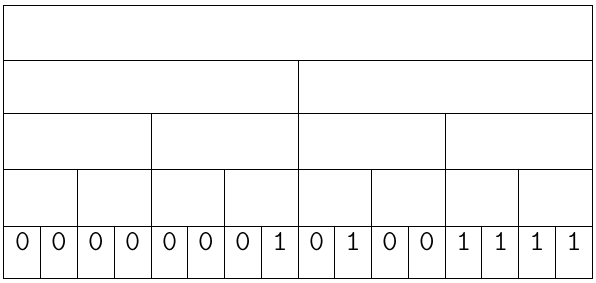


Figure 11: First step to fill the sub-block array

The void cells store a zero.
Now, this stage 4 row is analysed: a cell in the next row will store a 0 if the two cells below it are different from each other, otherwise, it they both are equal to 1, the cell will contain a 1. In the first case, the index associated with the 1 is saved in the vector C_S according to the formula: $CS[z] = (cell_{index} mod(2^{Stage})) \cdot (2^{Stage_{max} - Stage})$ where $Stage_{max}$ is equal to the number of stages of the binary tree.

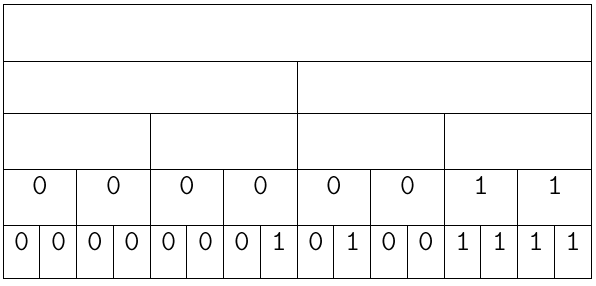


Figure 12: Second step to fill the sub-block array

This process is iterated until a row with all zeros is obtained (it must exist, otherwise it means that all bits are unfrozen). Finally, the number of indexes written in C_S is provided as the return value.

3.2.6 Metric_qs

This function simply sorts with the quick sort algorithm the metric vector U in ascending order according to the member LLR_over_mean.

3.2.7 E_nochild and E_noselect

These two functions are used to prune some nodes (or some edges) in order to keep down the number of possible flips to be performed. E_nochildis computes N1 counting the number of bits, belonging to Unfrozen, between the flip index and N, while N2 is the number of bits whose LLRs fail in achieving a certain threshold.
In both N1 and N2, the bits belonging to the critical set are excluded from the counting. Then, if $\frac{N2}{N1}$ is below a threshold w_{level} provided as an input, E_nochild returns 0, meaning that the node is allowed to generate child nodes.
If E_nochild returns 0, E_noselect is used to prune some edges. It considers the LLR corresponding to the child: if it is above a certain threshold, the child is discarded. The values used to compute the thresholds are provided in [5].

3.2.8 Results and Considerations

The performances, in terms of BER and FER, have been obtained for each level of the PBF decoder and compared with the results of the SC and SCF decoding schemes.

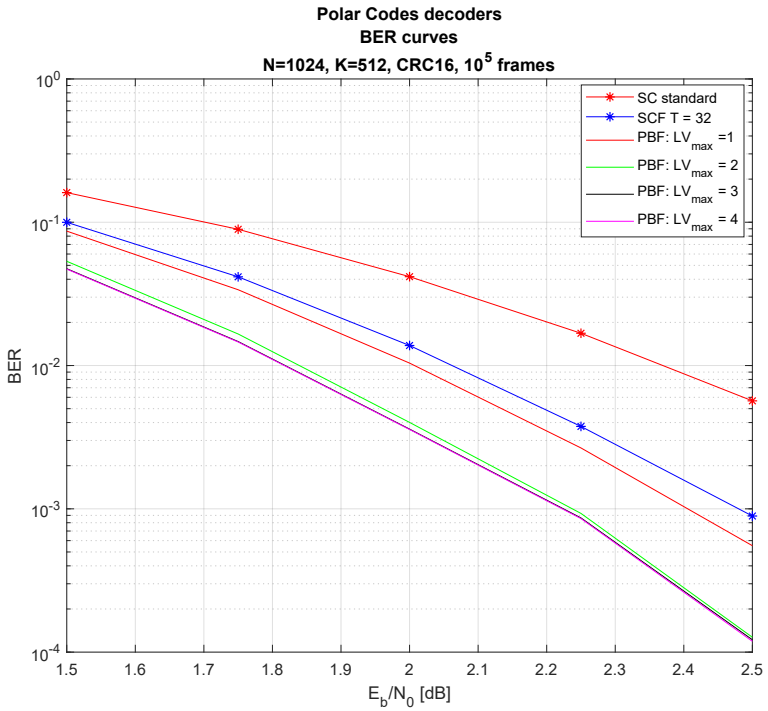


Figure 13: BER curves for the levels 1, 2, 3 and 4 of the PBF algorithm

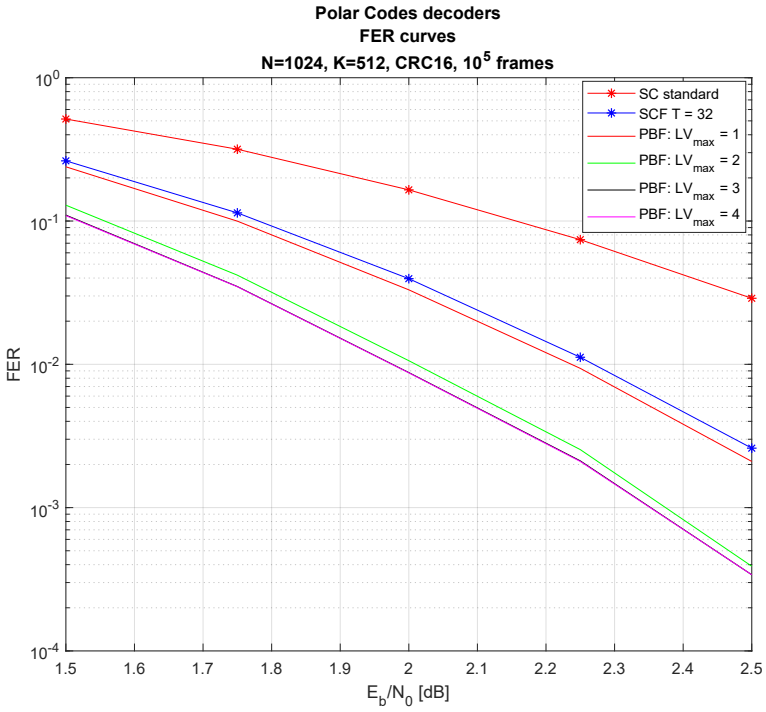


Figure 14: FER curves for the levels 1, 2, 3 and 4 of the PBF algorithm

The PBF gain over the SCF increases with the number of levels considered, since these schemes are able to correct more than one error.
It is interesting to notice that the curve corresponding to the third level is very close to the one obtained with a level 2 PBF.

4 Simplified PBF Attempts

The PBF has shown good performance, but the techniques used to prune the not reliable nodes are quite complex, involving parameters depending on the mean values of the previously computed LLR, on the square root of such values, on the level and even on the current value of SNR. Moreover, the value of the parameters to be used in the thresholds Enochild and Enoselect were provided in[5] as numbers for just some values of $\frac{E_b}{N_0}$, so that it was not possible to simulate the algorithm past an SNR of 2.5 dB. Therefore, some attempts to simplify the metric have been performed.
The results of the 1-level-PBF with the standard SCF seen in fig.14 shows that the gain of the first algorithm over the latter is not too high. Moreover, that PBF involved more than one hundred flips, while the SCF just 32. Therefore, the first attempt to simplify the PBF was based on the assumption that if at level 1 the algorithm is similar in performance to the SCF, then just T flip can be performed at level 1 and each of those nodes can generate at most T children.
This revised PBF workes in the following way: during the level 0 (standard SC decoding) the indexes corresponding to the LLR associated with the positions provided by the critical set are stored and sorted in ascending order. In case of failure of the CRC, the level 1 is reached, where T attempts are made by flipping the indexes corresponding to the smallest LLRs. During each of these attempts, the LLRs associate with the critical set of the flip are sorted and the first T of them are stored. If at the end of the T attempts of the level 1 the CRC still fails, the second level starts: each previous attempt is rerun in the same order, but, this time, for each try a new flip, taken from the sorted list created during the level 1, is added, so that there are T try for each attempt done in the first level. Therefore, at most T^2 attempts are performed during this phase.
The scheme below illustrates this process.

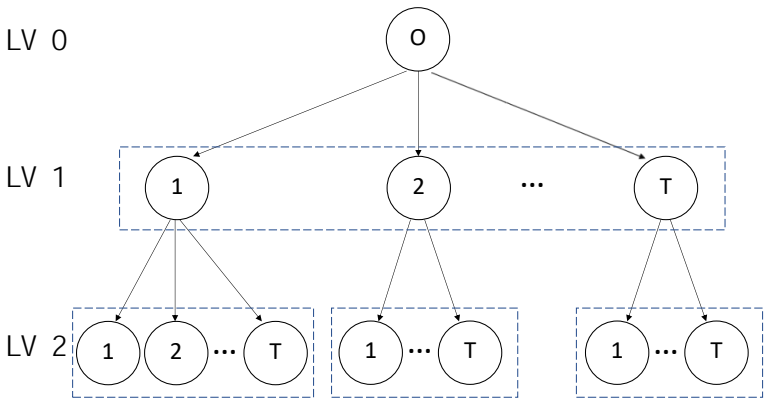


Figure 15: Simplified PBF scheme

However, the performance in terms of BER and FER are far from the ones shown by the standard PBF, so this approach has been rejected.

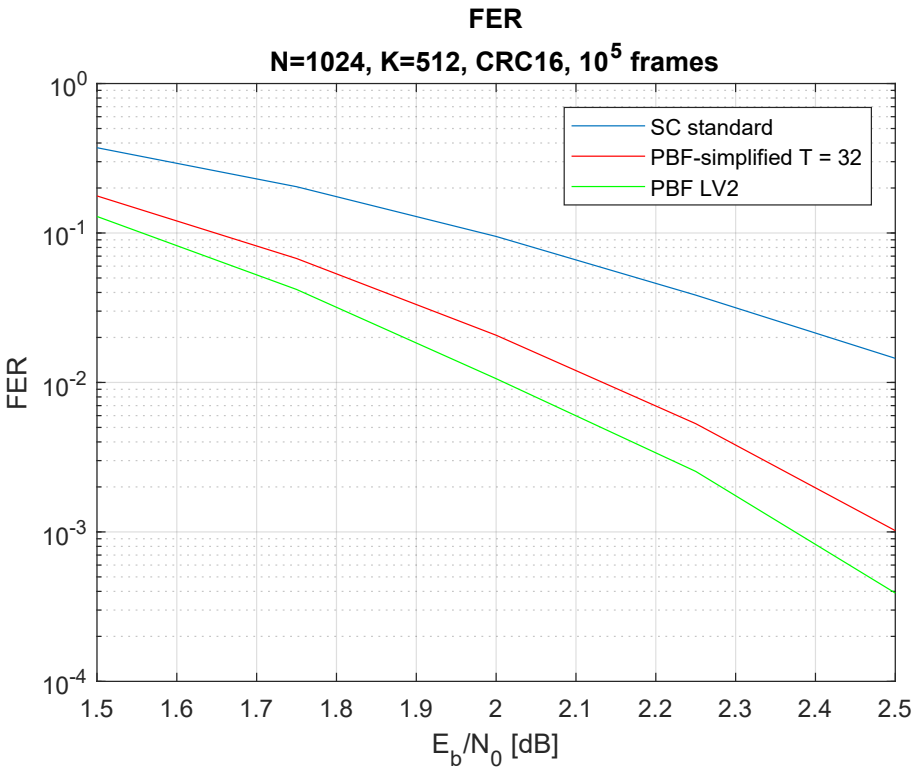


Figure 16: FER curves for the standard PBF LV2 and for the simplified attempt of it

4.1 Progressive Bit Flipping Constrained

The following approach is very similar to the first one but, this time, it is only applied on the second level. By using the critical set, a number dim_CS of attempts are performed in the first level and, during this, the T most unreliable positions, taken from the critical set of the flip, are stored. If the decoding in the level 1 fails, at level 2 at most T attempts are performed for each node of the upper level. With this method, a maximum of $dim\_CS \cdot T$ attempts are performed on the second level, therefore the maximum latency, in the worst-case scenario, is $dim\_CS \cdot (T+1)$ times the one of the standard SC.
Fig.17 portrays a graphical illustration of the proposed algorithm.

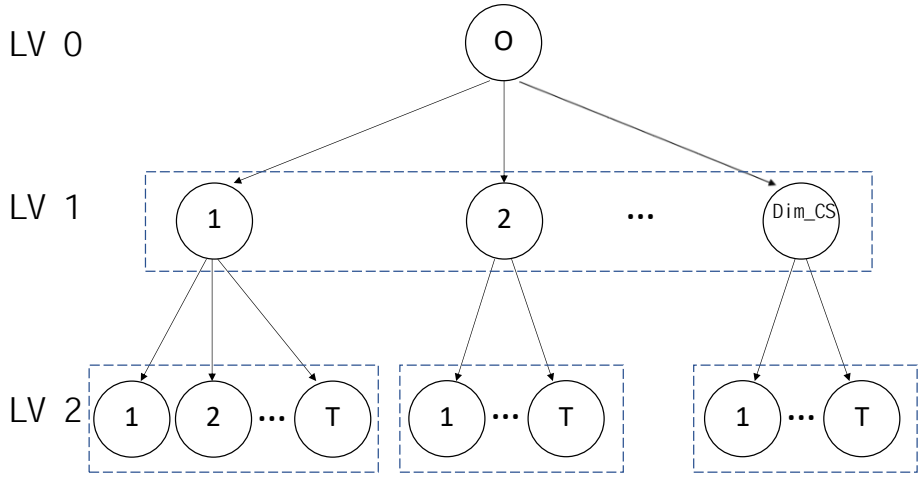


Figure 17: PBF-Constrained scheme

The following pseudo-code describes the software implementation:

Algorithm 1 PBF-C Algorithm

level = 0
$\hat{u}_0^{N-1} \leftarrow SC(y_0^{N-1})$
if $CRC(\hat{u}_0^{N-1})$ **then** $= failure$
    Generate CS_0
    Sort indexes of CS based on the ascending order of $|LLR_i|$
    $level++$
    for $(i = 0; i < dim_C S; i++)$ **do**
        $\hat{u}_0^{N-1} \leftarrow SCF(y_0^{N-1}, CS\_0[i])$
        if $CRC(\hat{u}_0^{N-1}) = failure$ **then**
            Generate CS_1(u_i)
            Sort and store T indexes of CS_1(u_i) in ascending order of $|LLR_i|$
        else break
        end if
    end for
    $i = 0$
    $level++$
    while $(CRC(\hat{u}_0^{N-1}) = failure\ OR\ i < dim\_CS)$ **do**
        for $(i = 0; i < dim\_CS; i++)$ **do**
            for $(j = 0; j < T; j++)$ **do**
                $\hat{u}_0^{N-1} \leftarrow SCF(y_0^{N-1}, CS\_0[i], CS\_1[j])$
                if $(CRC(\hat{u}_0^{N-1}) = success)$ **then**
                    break
                end if
            end for
        end for
    end while
end if

The performance of this second algorithm is shown in the figures below.

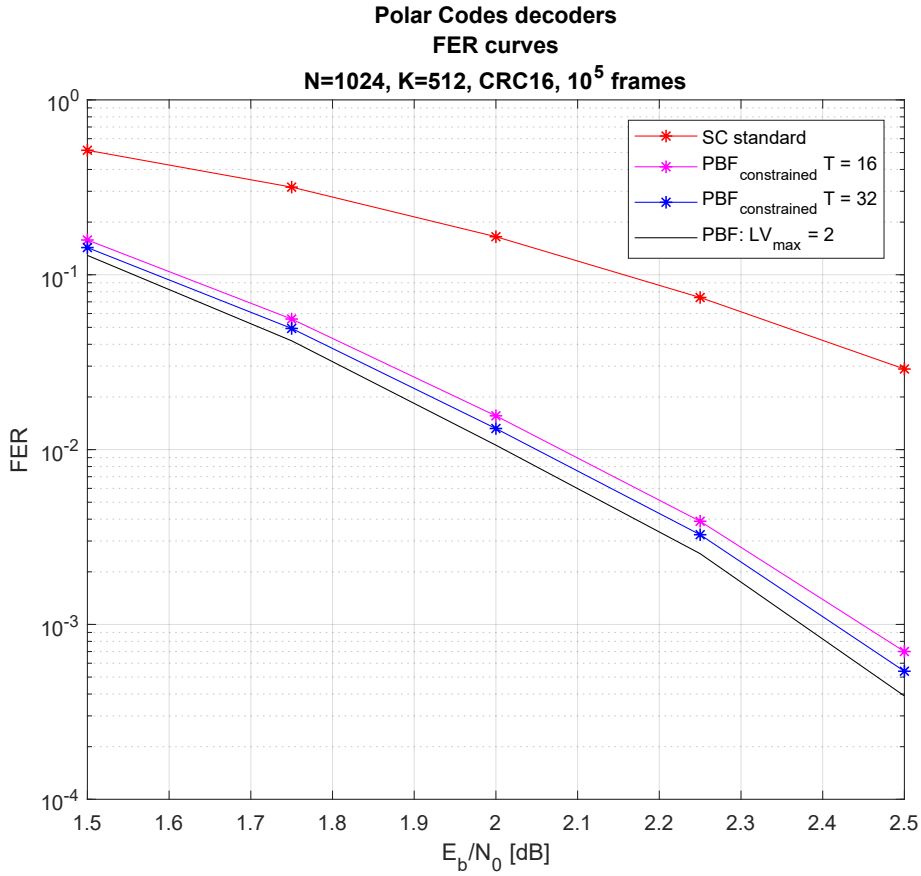


Figure 18: FER curves for the standard PBF LV2 and for the PBF-C algorithm proposed

This time, the curves follow the results seen for the PBF scheme, with an acceptable loss when the number of attempts is T=32. The name for this algorithm has been chosen as **PBF-Constrained** since the nodes of the level 1 are constrained to produce at most T children.
With respect to the standard PBF, the constrained version allows the pruning of unreliable nodes without involving square roots, mean values of parameters, fixed point arithmetic and it allowed to simulate the performance for values of SNR greater than 2.5 dB as shown in fig.19.

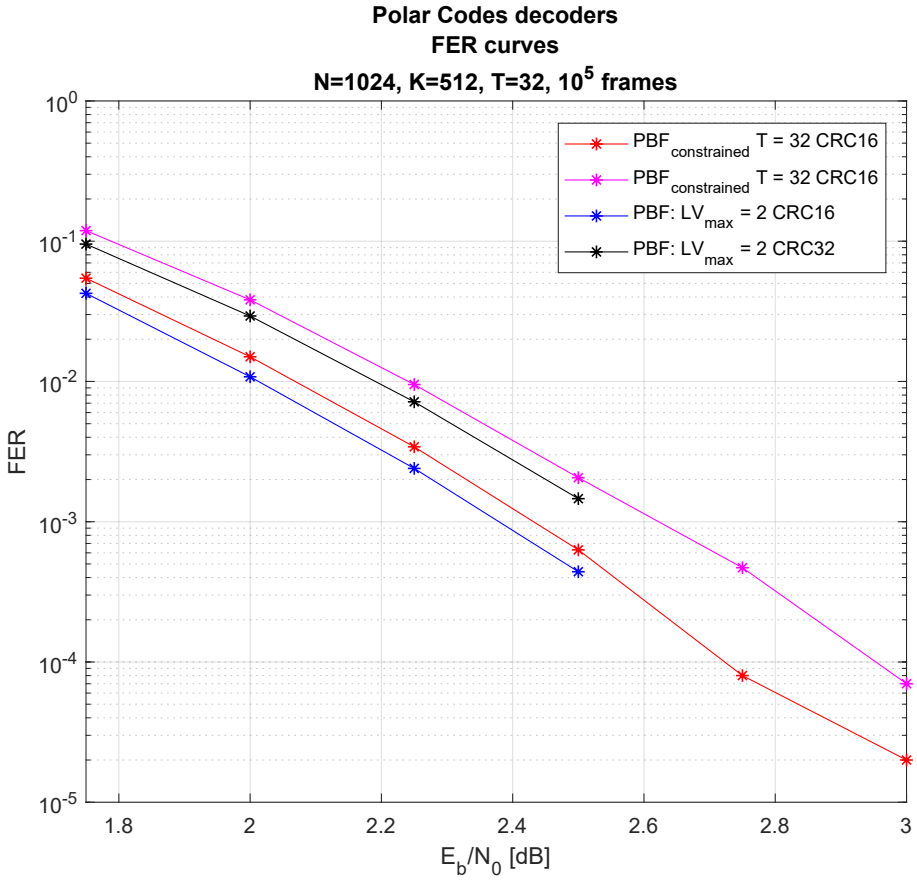


Figure 19: FER curves past 2.5dB

As a result, the PC decoder architecture proposed in this thesis will be based the PBF-Constrained algorithm. It must be noted that this method is very close to a work parallel to this one [10], where an Early Termination technique for the PBF is introduced, but a different and more complex sorting metric is used and no hardware implementation is proposed.

5 PBF-C Architecture

The architecture proposed for the PBF-Constrained decoder and shown in fig.20 is made of a Control Unit that manages the signals to be used based on the level reached by the decoding, an SC module that is used to execute the SC decoding, a CRC-16 block, that verifies the result of the decoder, two sorting networks, so that the indexes to be flipped are organized based on the absolute value of their LLR, a set of ROMs and RAMs to store the critical sets and four LUTs to get the proper address of the memories.
The architecture has been designed for a codelength N=1024 with a number of processing elements equal to M=64.

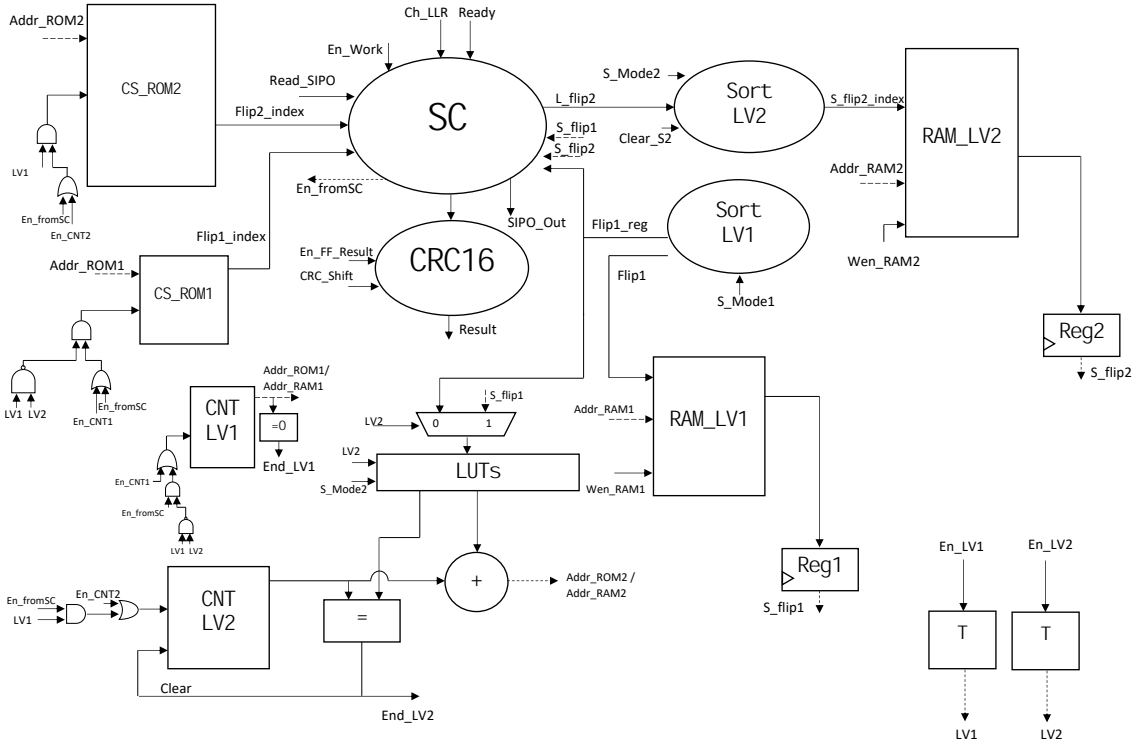


Figure 20: PBF-C decoder architecture

The decoding process is divided into three phases:

- **Level 0**: while a standard SC decoding is being performed, the magnitude of the LLRs belonging to the critical set of the level 0, stored in CS_ROM1, are sorted by the network Sort_LV1. If the SC decoding fails, the next phase starts;

- **Level 1**: considering **dim_CS** the dimension of the critical set of the level 0, a maximum number of dim_CS decoding attempts are tried, during which one bit is flipped and the **T** least reliable LLRs of the critical set of the bit being flipped are sorted and stored by Sort_LV2. At the start of each attempt, the index to be flipped is provided by the first register of Sort_LV1 and it is then

stored in RAM_LV1. In this way, the level 1 phase can start as soon as the level 0 phase ends, without waiting to store all the indexes into RAM_LV1. This, however, introduces the need of a second, smaller, sorting network. At the end of each attempt, the content of the second sorting network is saved into RAM_LV2 while the CRC is computed (if T is equal to the CRC length there is no need to wait for the end of the writing process);

- **Level 2:** after dim_CS failed attempts in the level 1, each of them is repeated a maximum of T times, flipping an additional index.

5.1 SC module

The SC module is composed of a controller and a datapath containing a memory for storing the internal LLRs and M=64 processing elements (m=6) implementing the semi-parallel architecture described in [8].

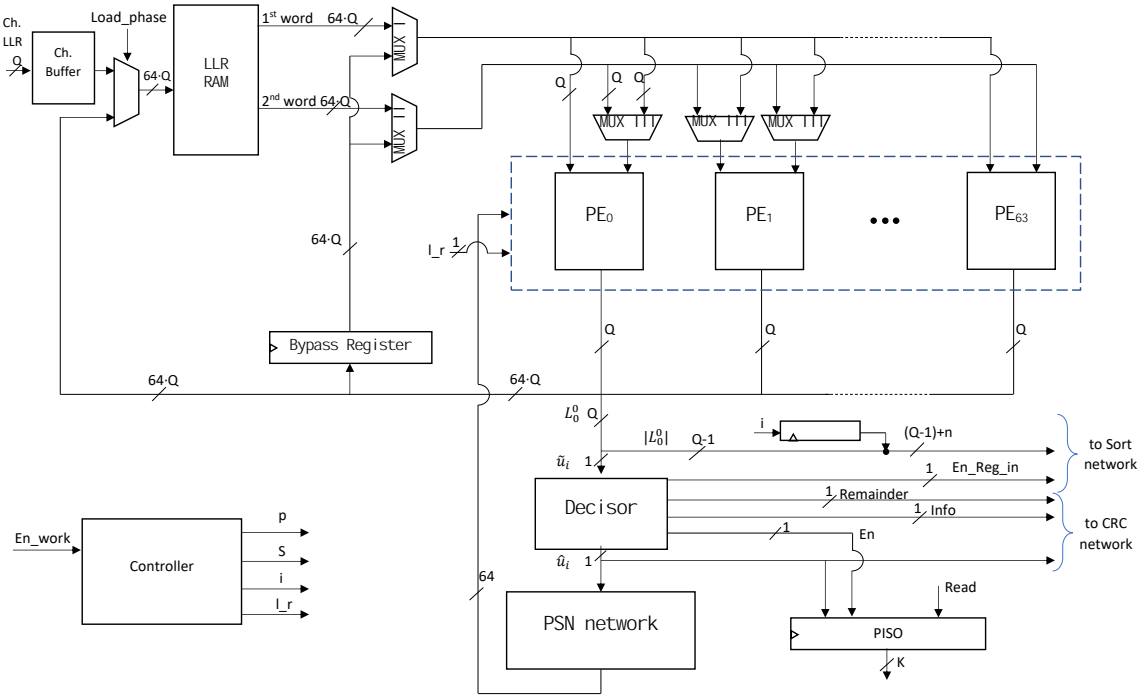


Figure 21: Scheme of the SC architecture

5.1.1 SC Controller

The SC Controller computes the control signals required by the SC module. These signals are the number of decoded bits i ($0 \leq i < N$), the current stage S ($0 \leq S < \log_2(N)$), the current portion of stage p ($0 \leq p < \frac{2^S}{M}$) and a signal to issue the computation of a left or right message l_r (left if $l_r = 0$, right if $l_r = 1$).

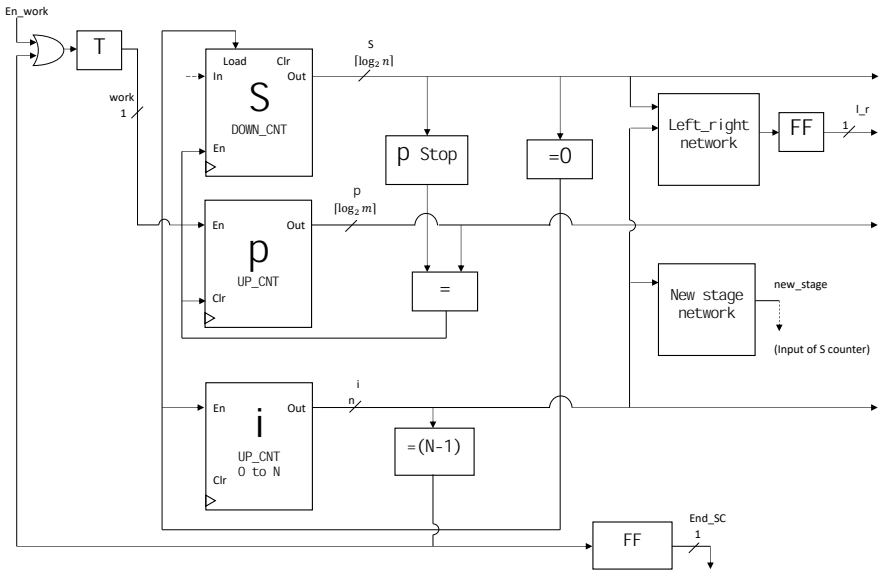


Figure 22: Scheme of the SC Controller

The stage portion p is provided by an up-counter starting from 0. When the output of this counter reaches the stop condition of p ($\frac{2^S}{M}$), the S counter is enabled. The check of this condition is provided by a combinational circuit that first computes $\frac{2^S}{M}$ by left shifting p of S positions and then by right shifting the result by $m = log_2 M$ positions then, it compares this result with the actual value of p. If they are the same, the p counter is cleared and S is updated (and, therefore, even the stop condition of p).
The stage index S is provided by a down counter starting from $log_2(N) - 1$, so, in this case, from $S = 9$, activated whenever p reaches its stop condition. When $S = 0$, the counter that provides the number of decoded i bits is enabled and the S counter is updated with a new value that depends on i. In particular, the new stage is the position of the first zero in the binary representation of i. As an example, let us consider the following cases for N = 8 (n=3):

$$i = 2 \rightarrow i_{bin} = 010 \rightarrow newstage = 0;$$

$$i = 3 \rightarrow i_{bin} = 011 \rightarrow newstage = 2;$$

$$i = 5 \rightarrow i_{bin} = 101 \rightarrow newstage = 1;$$

This index is obtained by the combinational circuit New_stage_network. This obtains the position of the first zero with an AND operation between the negated value of i ($\bar{i}$) and the value $i + 1$. This produces a binary number with a 1 in the position of the first 0 of the original number. Finally, with a log_2 operation, the index is retrieved.

$$i_{bin} = 011 \rightarrow \bar{i} = 100 \rightarrow i + 1 = 100 \rightarrow \bar{i} \ \& \ (i + 1) = 100 \rightarrow newstage = 2$$

The scheme of this circuit is shown below.

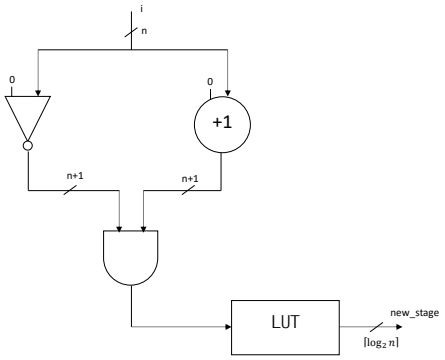


Figure 23: Scheme of the New_stage_network

As it can be seen in fig.23, the i signal is zero padded on the MSB side, so to avoid overflow when i=N-1. Moreover, the $\log_2$ operation is implemented by the means of a LUT, which also provides $newstage = log_2 N - 1$ when $i = N - 1$. The LUT used in the main architecture with N=9 is shown in fig.24.

LUT	
IN	OUT
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512, 1024	9

Figure 24: LUT in the New_stage_network

The last signal the controller provides is the l_r signal, used by the PEs to execute the computation of the left or the right message. This value is obtained by computing the mod2 of $\frac{i}{2^S}$.
All these operations are described in the following timing diagram.

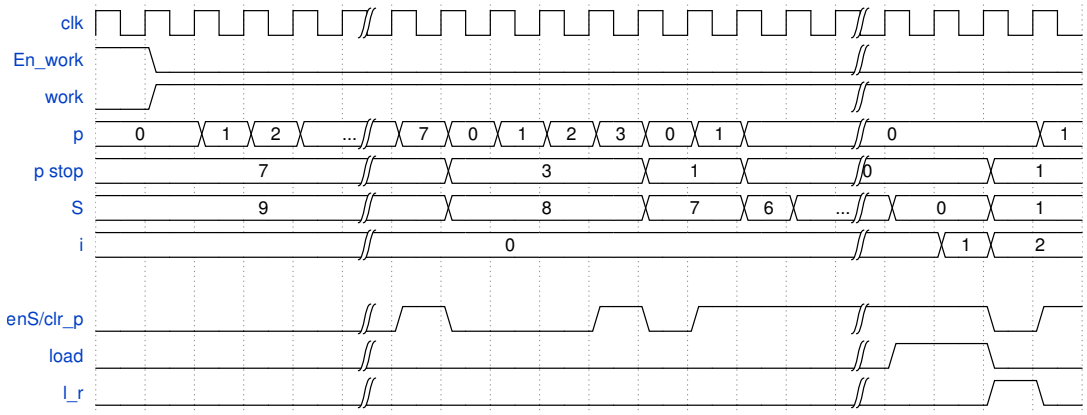


Figure 25: Timing diagram of the controller

When the Controller receives the pulse En_work from the external CU, the FF T commutes this pulse into a constant signal (work) that serves as an enabler for the p Counter. When the value of p reaches its stop condition, a signal, acting as a clear for the p counter and as an enable for the S counter, is issued. When S reaches the value 0, i is updated and a load signal is brought high to let start the S counter from the new_stage value.

5.1.2 PSN Network

In [9] an efficient PSN to be integrated with the semi-parallel scheme is described. This network can be divided into two parts, one to be used when $S \leq m$ (parallel updated scheme), the other when $S > m$ (serial updated scheme), and it is shown in the figure below.

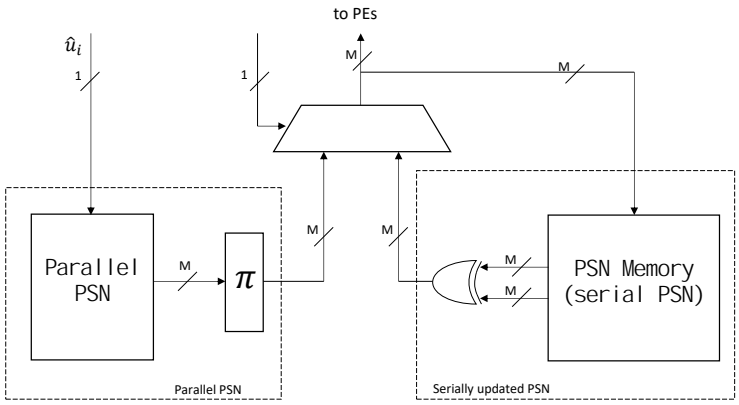


Figure 26: PSN scheme proposed in [9]

When $S \leq m$, all the PSN required by all the operations in the stage are computed together by the Parallel PSN by multiplying the newly received decoded bit u_i with the output of a generator matrix. The result is saved in M 1-bit registers (FFs) and provided to the PEs.

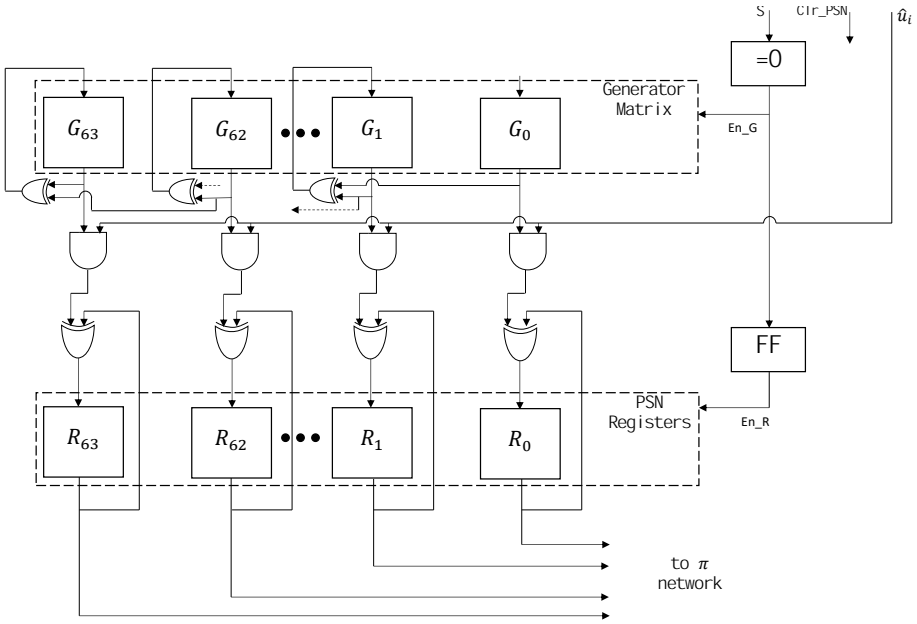


Figure 27: Parallel PSN scheme

Since the bits are decoded when $S = 0$, this condition is used to enable the generator matrix and the output registers to compute the new partial sums. These value, however, are not directly sent to the PEs, but they undergo the π network that, based on the value of i and S, provide the right link between the registers and the PEs. When i is equal to an integer multiple of M and p is equal to 1, the registers and the generator matrix (which is also a set of M FFs) are cleared. For the sake of simplicity, let us consider as an example a scenario with only 4 PEs and the partial sums being represented by capital letters of the alphabet (e.g. $A = u_0, B = u_1, ...$), the behaviour of the parallel PSN is shown in the following timing diagram.

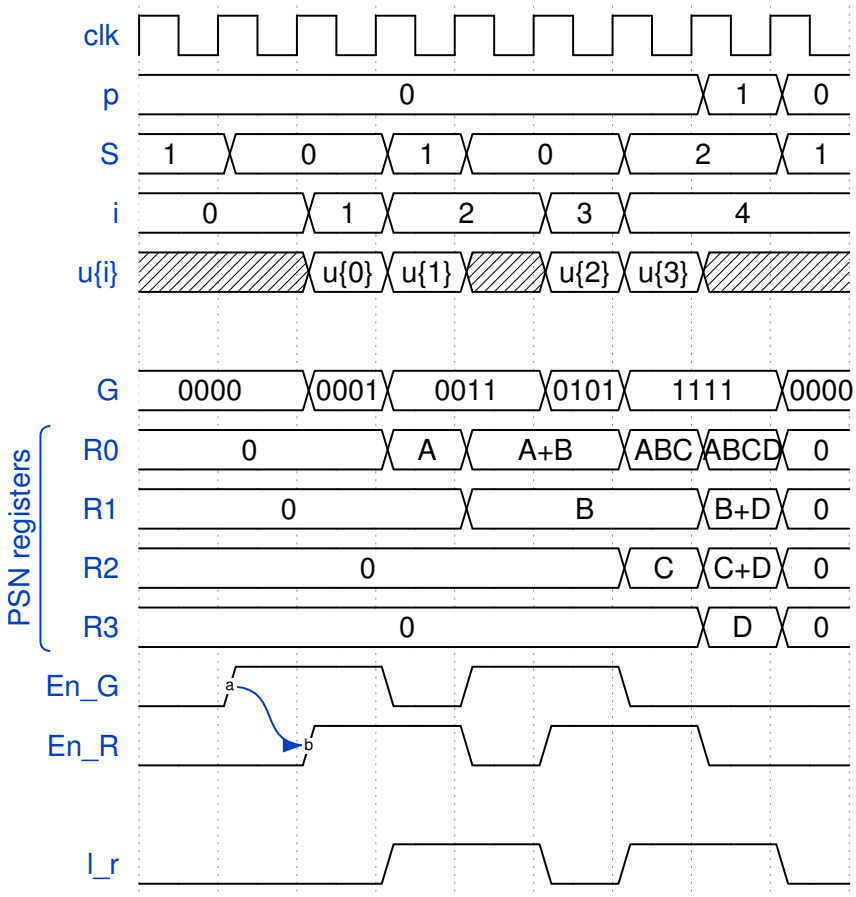


Figure 28: Parallel PSN timing diagram

When u_0 is being decoded, the generator matrix is enabled providing ”0001” (the LSB is connected to Reg0). In the following clock cycle, u_0 is sampled and multiplied with the generator matrix, therefore only the first register (R0) will be updated and its content is used to compute u_1 (notice that during the computation of u_1, the signal l_r is high). Then, the multiplication between the generator matrix, now containing ”0011”, and the just decoded u_1 will update R0 with ”A+B and R1 with B. After the computation of u_2, the first register, R0, contains the data A+B+C which is never used in the SC decoding, however, it is sent to a PE which is not used for the computation of u_3, in fact, the PE designed to compute u_3 will receive only R2 containing the required value C. After the decoding of u_3, all the registers are updated, since the generator matrix contains ”1111” and their content is sent to all the PEs to compute the internal LLRs. Moreover, these partial sums are also sent to the serially-updated PSN. After this, the content of the matrix and of the registers is cleared.

In the decoder architecture, when $S \leq m$, the active PEs are the first 2^S. Therefore, the π connection logic block is used to provide the proper partial sum value to the right PE. For N=1024 and M=64, the π network consists of 2 multiplexers with 32 inputs (for PE_0 and PE_1), 2 multiplexers with 16 inputs (for PE_2 and PE_3), 4 multiplexers with 8 inputs (for the PEs from 4 to 7), 8 multiplexers with 4

inputs (for the PEs from 8 to 15) and 16 2-to-1 multiplexers (for the PEs from 16 to 31). The inputs of the multiplexers are the content of the 64 registers containing the partial sums in the parallel PSN, and are selected by a counter activated when a particular bit of i changes. At the start and at the end of the 64 partial sums computations, the counters output is equal to their max values. The remaining PEs (from 32 to 63) are directly connected to the proper register.

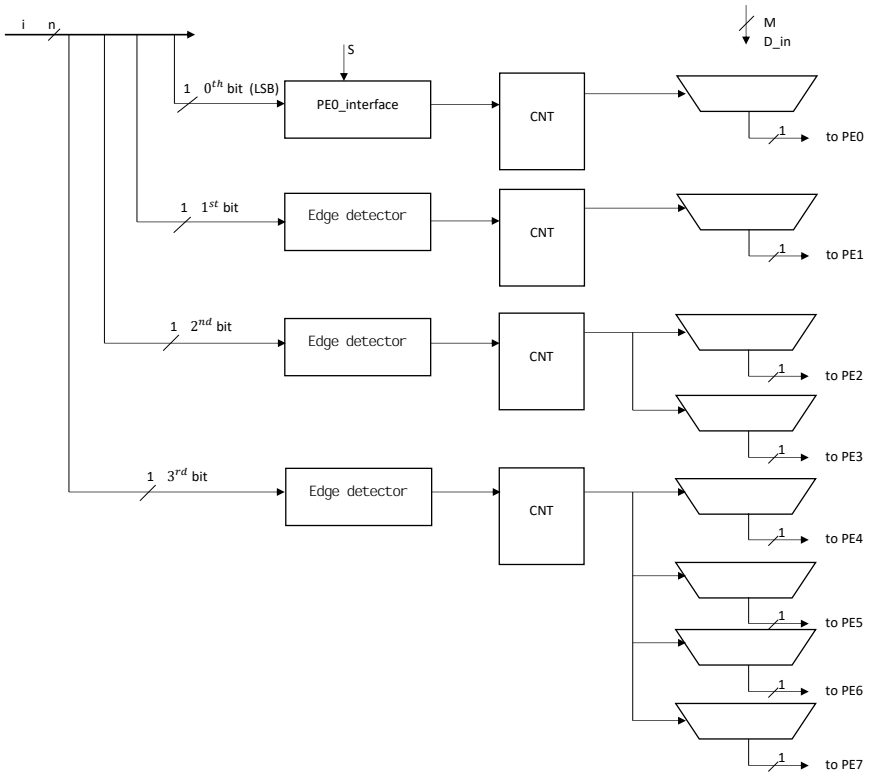


Figure 29: Scheme of the π circuit

In fig.29 the scheme of the π network is shown. The part regarding the connections for the PEs from 8 to 31 is not shown due to space constraints. The scheme of the selectors of the multiplexers is the same for all the sets of multiplexers, with the only exception being the selector for PE_0, since, unlike the others, it has to provide the same register more than once. As an example, the register R_0 is provided to PE_0 every time i is a power of 2 (therefore there is an additional 2-to-1 multiplexer in this scheme to simplify the selection logic). This is shown in fig.30, where the value of S is used to select registers corresponding to a selector with a lower value than the one provided by the counter (which is activated whenever the LSB of i is 1).

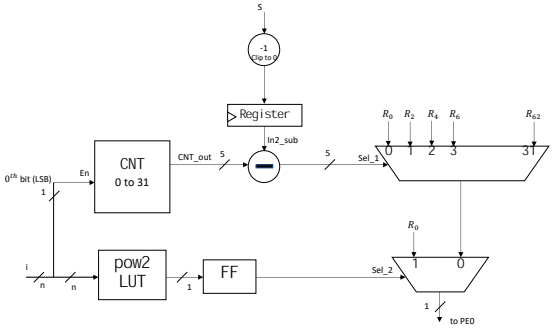


Figure 30: Interface between the parallel PSN and PE0

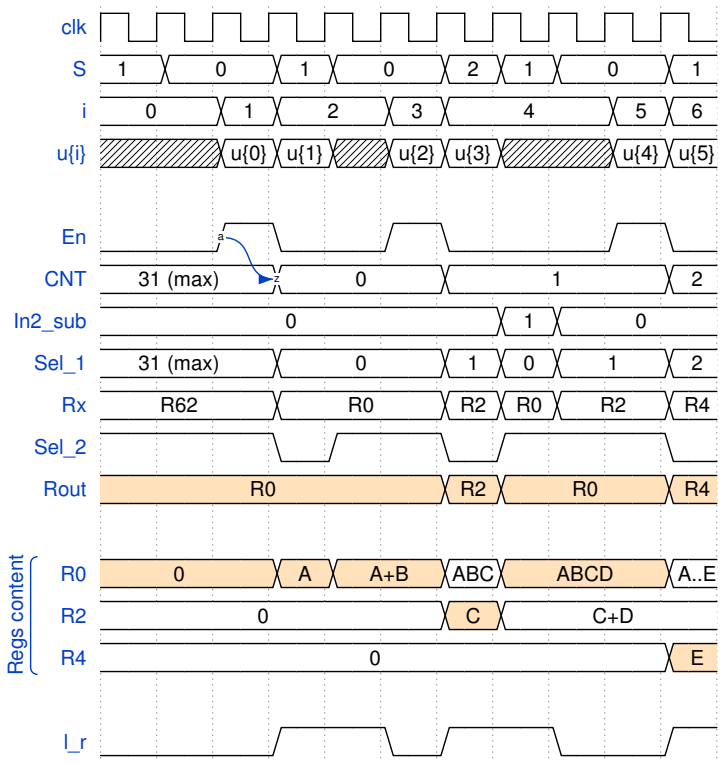


Figure 31: Timing of interface between the parallel PSN and PE0

In the timing of fig.31 the values provided to the PE are enlightened and it is possible to notice that the proper value is passed with the right timing (as an example, the partial sum C is used to compute u_3).
As for the other selectors, they share the same scheme, with the difference being the number of multiplexers present, the dynamic of the counter and the bit whose edge variation is used to activate it. The scheme in fig.32 shows the example of the circuit for the interface between the parallel PSN and the PEs from 8 to 15.

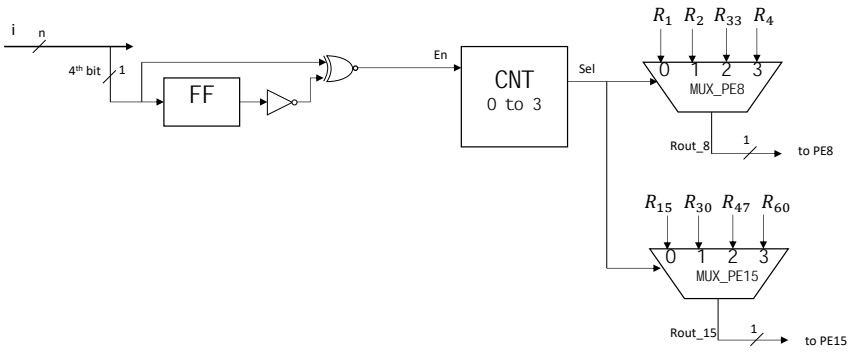


Figure 32: interface between the parallel PSN and PEs from 8 to 15

The timing of the interface for the PE_8 in fig.33 shows how the variation of the 4^{th} bit of i (the LSB is the 0^{th}) is used as an enable for the counter. The value provided in the beginning (R4) is not actually used by the PE.

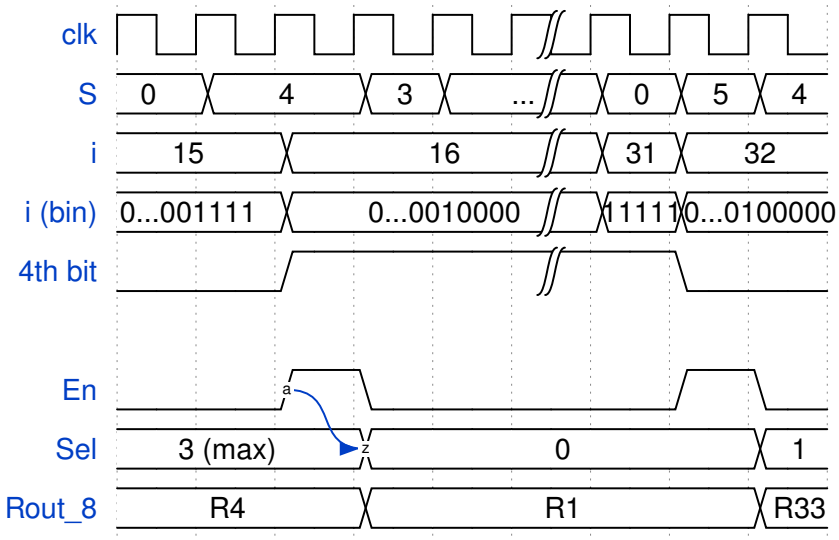


Figure 33: Timing of interface between the parallel PSN and PE8

The PEi with $16 \leq i \leq 31$ are respectively connected to the register Ri with $\frac{i}{2}$ when S=5 and with R_i when S=6. The inputs of the other multiplexers for the PEs interface with the PSN network are shown in the appendix.

As for the serially-updated part, it is required to store a total of N/2 partial sums in a memory element, since when i>N/2, the previously computed partial sums have already been used. When i becomes a multiple of M, the content of the registers of the parallel PSN is passed to the PEs and it is also saved as a word in the memory of the serial scheme. Then, for the partial stages with $p \geq 1$, the partial sums for the PEs are obtained by reading and XOR-ing two words from the memory. In this case, the interface towards the PEs is straightforward the first bit from the XOR

goes to PE_0, the second to PE_1, and so on. As for the memory structure, even if [9] describes the use of a tri-port memory, in this thesis it has been designed as a register bank to satisfy timing constraints. Therefore, each word of the memory is replaced by a register containing M partial sums, therefore $\frac{N}{2M}$ registers of M bits are required. A one-hot enable is used to select one of these registers for the writing operation. The two read ports of the memory are replaced by a tree of multiplexers.

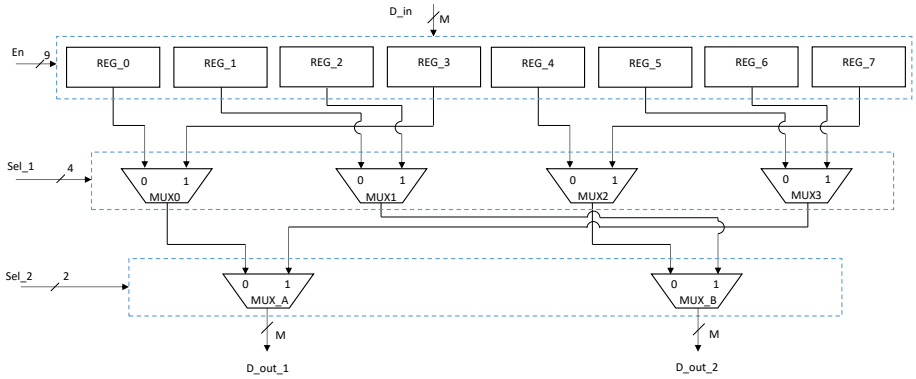


Figure 34: Register bank of the serially-updated PSN scheme

The selectors of the multiplexers and the enable of the registers are provided by a small FSM.

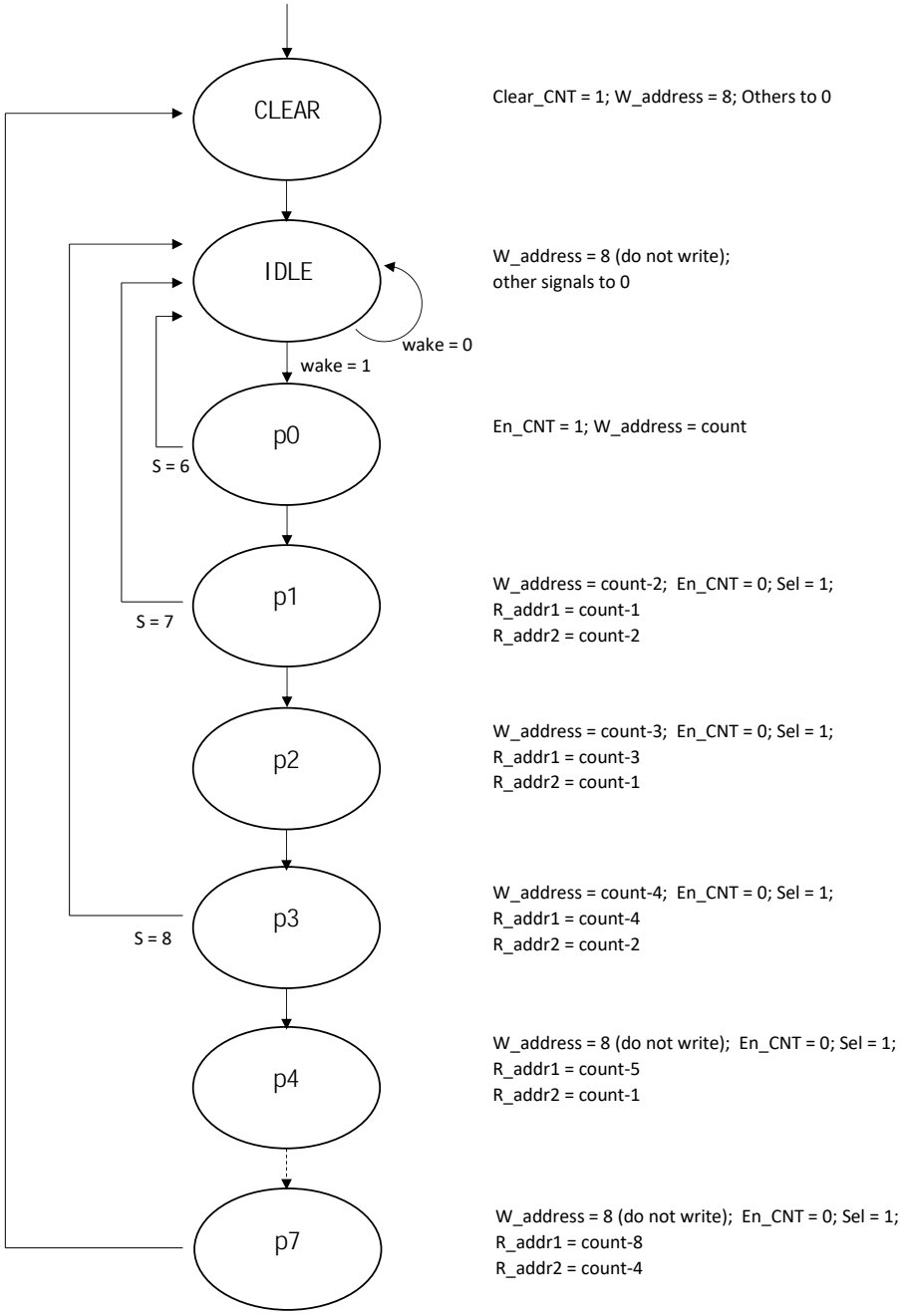


Figure 35: FSM of the PSN circuit

After the CLEAR state, the FSM is in the IDLE state until the pulse signal *wake* is '1' that happens when i becomes a multiple of M. To obtain this signal, it could be used an edge detector on the 7^{th}LSB of i, therefore it could be used the same signal found in the πnetwork. After the IDLE, in the p0 state, a counter is abilitated and the data from the parallel network are saved on the register with the address given by the old value of the counter. The following states are used to select the proper register to write and the ones to read from. From the state p4 it is not mandatory to keep performing write operations and, after the state p7, the machine returns to

the CLEAR state where the counter is cleared. This counter is also cleared at the end of the decoding attempt.

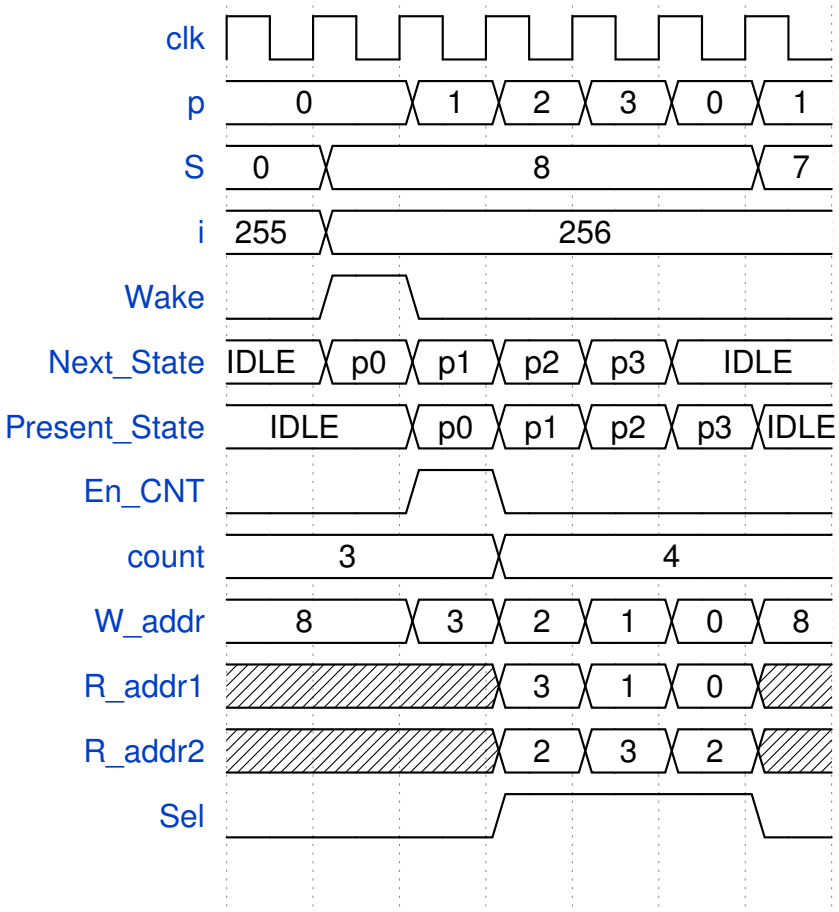


Figure 36: Timing of the FSM in the PSN circuit

The timing of the FSM is shown in fig.36, where the signal *Sel* is used to switch the output of the PSN between the parallel and the serially-updated part. The outputs *W_addr, R_addr1, R_addr2* of the FSM are sent to some LUTs to obtain the signals used to control the bank of registers. The content of the LUTs is shown below.

IN	Sel1	Sel2	En_reg
0	0000	00	00000001
1	0000	00	00000010
2	0100	00	00000100
3	1000	00	00001000
4	0000	01	00010000
5	0000	10	00100000
6	0001	10	01000000
7	0010	01	10000000
8	/	/	00000000

Figure 37: Outputs of the LUTs used in the PSN

The whole PSN architecture is portrayed in the following scheme.

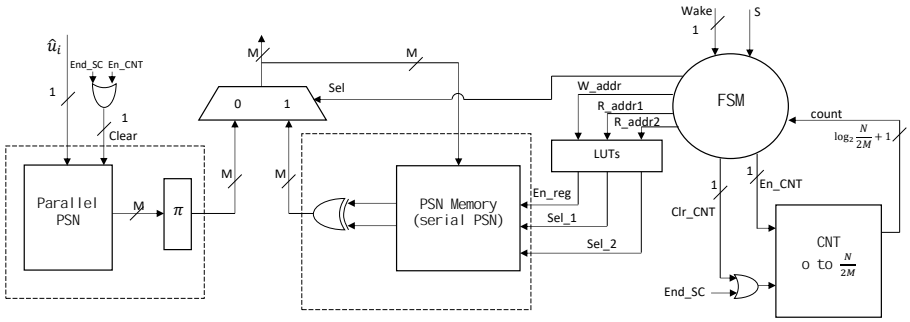


Figure 38: PSN architecture

This PSN architecture allows to reduce the area compared to other PSN schemes, moreover, only the memory for the partial sums (in the serially-updated part) is related to the value N. A final example for a code with N=8 and M=2 is shown below.

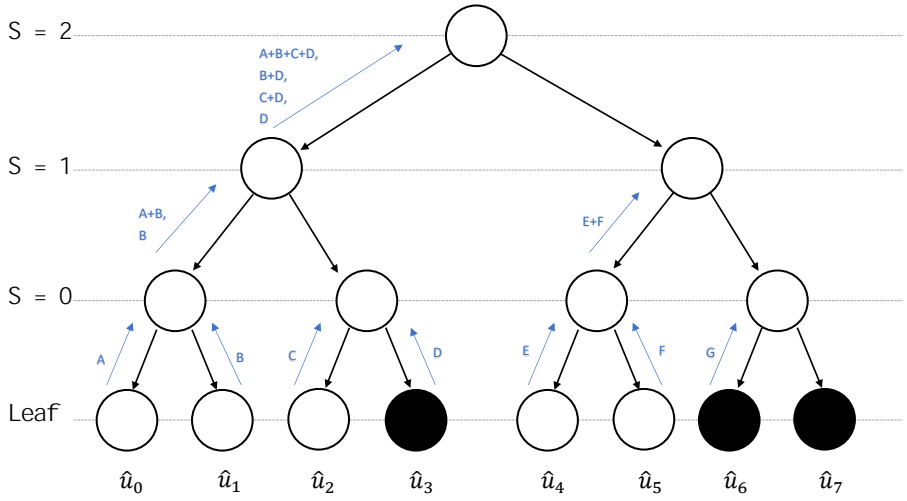


Figure 39: Partial sums passed over the tree of a N=8 PC

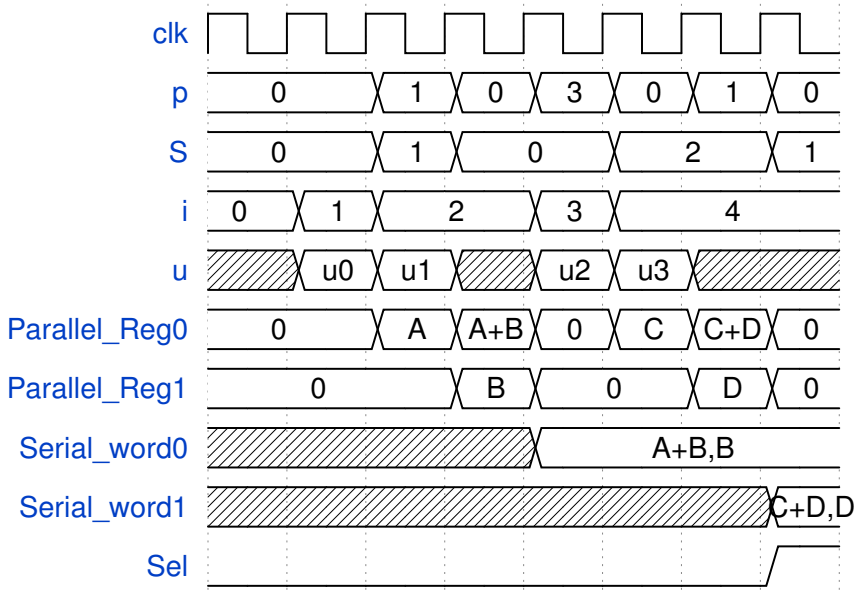


Figure 40: PSN timing for a N=8 PC with 2 PEs

A consequence of the implementation of the HPPSN in the semi-parallel architecture is that the scheduling of the operations will be different compared to what was proposed in the original semi-parallel scheme [8]. In the previous example, when i=4 and S=2, the value provided to the 2 PEs are first C+D and D, then, in the following clock cycle, the partial sums are obtained by reading A+B, B and C+D, D from the memory words and performing a XOR operation between them so to get the values A+B+C+D, B+D. This leads to a different way to fill the LLRs memory.

5.1.3 LLR Memory

In the original semi-parallel architecture, the LLRs memory words contains M LLRs organized in bit reverse way. Moreover, 2 consecutive words are read (starting from the top of the memory block referring to the actual stage) and one word is written (starting from the top of the region referring to the stage below). By implementing the HPPSN, however, the scheduling is different and the cell memory must be placed in a different way inside the word. In [9] a double permutation for the LLRs is proposed: first the address of the cell is divided into two part, $b_S b_{S-1} ... b_m$ is the word address, while $b_{m-1} ... b_0$ is the cell index. Then, considering j as the binary index of an LLR, the cell where LLR_j must be put is given by right rotating the word address by one position and by bit-reversing the cell index. Therefore, for a polar code with N=32 and M=4, the filled LLR RAM will be like the one shown in fig.41.

LLR memory for a PC N=32 M=8

								Word Address
S = 4	L_0^5	L_4^5	L_2^5	L_6^5	L_1^5	L_5^5	L_3^5	L_7^5	0000
	L_{16}^5	L_{20}^5	L_{18}^5	L_{22}^5	L_{17}^5	L_{21}^5	L_{19}^5	L_{23}^5	0001
	L_8^5	L_{12}^5	L_{10}^5	L_{14}^5	L_9^5	L_{13}^5	L_{11}^5	L_{15}^5	0010
	L_{24}^5	L_{28}^5	L_{26}^5	L_{30}^5	L_{25}^5	L_{29}^5	L_{27}^5	L_{31}^5	0011
S = 3	L_0^4	L_4^4	L_2^4	L_6^4	L_1^4	L_5^4	L_3^4	L_7^4	0100
	L_8^4	L_{12}^4	L_{10}^4	L_{14}^4	L_9^4	L_{13}^4	L_{11}^4	L_{15}^4	0101
S = 2	L_0^3	L_4^3	L_2^3	L_6^3	L_1^3	L_5^3	L_3^3	L_7^3	0110
	/	/	/	/	/	/	/	/	0111
S = 1	L_0^2	L_2^2	L_1^2	L_3^2	/	/	/	/	1000
	/	/	/	/	/	/	/	/	1001
S = 0	L_0^1	L_1^1	/	/	/	/	/	/	1010
	/	/	/	/	/	/	/	/	1011

Figure 41: LLR memory for an N=32 M=8 PC

To better understand, let us consider the second cell of the second row of the memory. This cell is the 9^{th} cell of the stage 4, therefore its binary representation is 01001, with the double permutation it is transformed into 10100, therefore the 9^{th} cell of the stage S=4 will contains the 20^th LLR sent to the decoder.

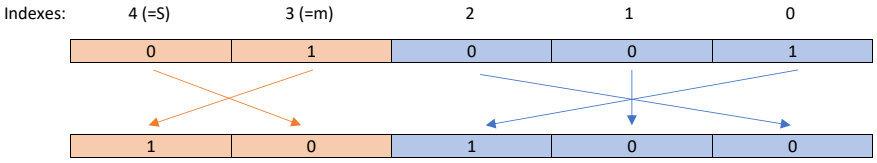


Figure 42: Example of the double permutation

As for the scheduling, The first words to be read are the ones corresponding to the address 0010 and 0011. Their content is used to compute 8 internal LLRs that are saved in 0101. Then the words 0000 and 0001 are read and the result is put into 0100. When $S \leq m$ the scheduling is the same of the standard semi-parallel scheme. It should be noticed that when $S \leq m$ the internal LLRs just produced must be used immediately, therefore while the result is being written in the RAM memory,

the PEs will get the proper LLRs from a by-pass buffer. In particular, when $S = m$ the upper word is taken from the buffer and the bottom one from the memory. As for the N=1024 M=64 PC code, its LLR RAM has 42 words of 64 LLR each. The first 16 words belongs to the stage 9, the following 8 to the stage 8, then 4 to the stage 7 and the all the other stages have 2 words. To keep the addressing and the connection to the PEs simple, in both [8] and [9] two words are allocated even when $S < m$. This, however, introduces a redundancy as shown in fig.41.
The scheme used to control the Ram memory is the following:

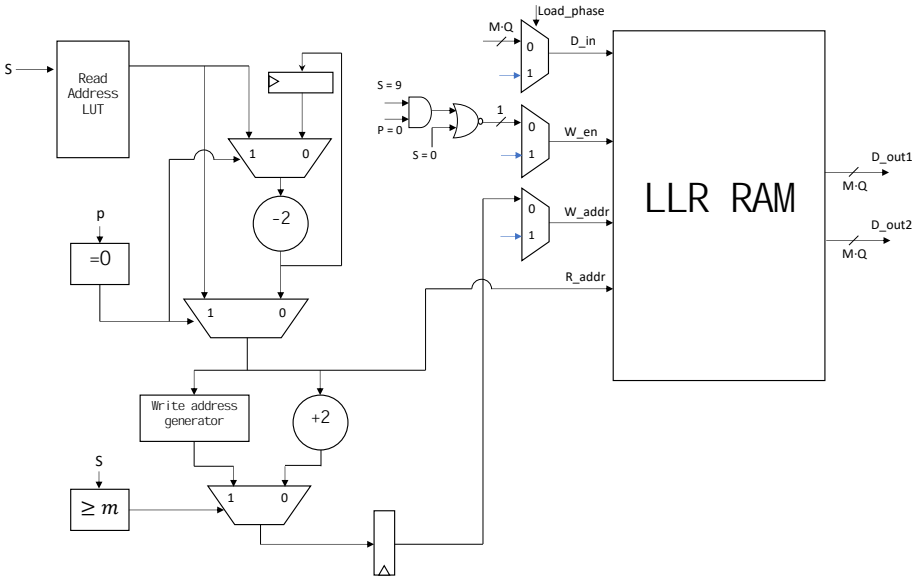


Figure 43: Scheme of the LLR Memory controller

The stage signal S is used to get from a LUT the first address for that stage. The following addresses when $S > m$ and $p > 0$ will be the previous one minus 2, while when $S \leq m$ the address to be read is simply taken from the LUT.
The write address is derived from the read address by summing 2^{S-m+1} to it and by swapping the LSB with the bit in position $S - m$ or, alternatively, by using a second LUT.
The write enable W_en is low when S=0 since the LLRs of the leaf stage are not saved in the memory, and even when S=9 and p=0.
Since the LLR memory is filled by a buffer at the beginning of the decoding, the Data_in, W_en and W_addr ports have a multiplexer to switch between the signals provided by the buffer (blue arrows) and the ones provided by the SC circuit.

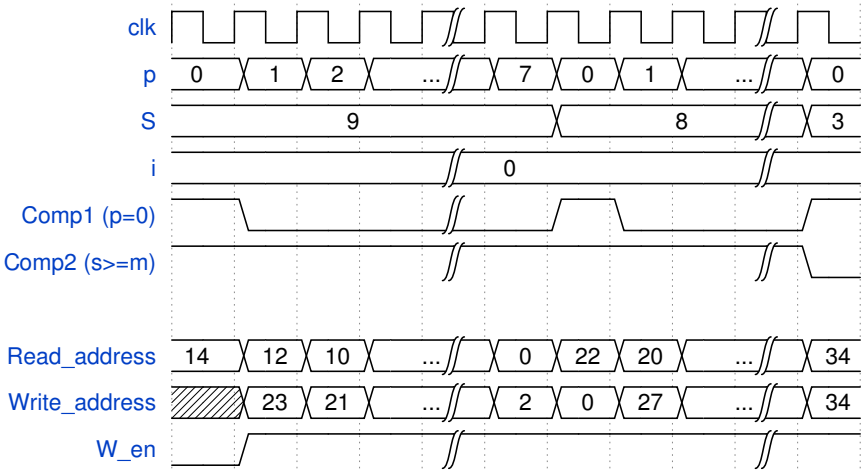


Figure 44: Timing of the LLR Memory for a PC with N=1024 and M=64

The timing shows how the read addresses are provided to the memory (notice that for a single address two consecutive words are read). Moreover, when $S \leq m$ there is a conflict, however in those stages the values to the PEs are provided by the by-pass register, so the write operation is executed correctly and the read one is performed but the data obtained from the memory are not passed to the PEs.

5.1.4 LLR Memory-PEs Interface

The different scheduling introduced with the HPPSN requires different connections between the LLR memory and the M PEs. When $S > m$ the inputs to the PEs are taken from the two words read from the memory. In particular, the PE_i will have as inputs the i^{th} cells from the two words. When $S = m$ the first word is taken from the by-pass register while the second word is read from the memory, while, when $S < m$ all the LLRs provided to the PEs are taken from the by-pass register word, in particular, the PE_i gets the content of the $2 \cdot i^{th}$ and the $2 \cdot (i+1)^{th}$ cells of the bypass register word.

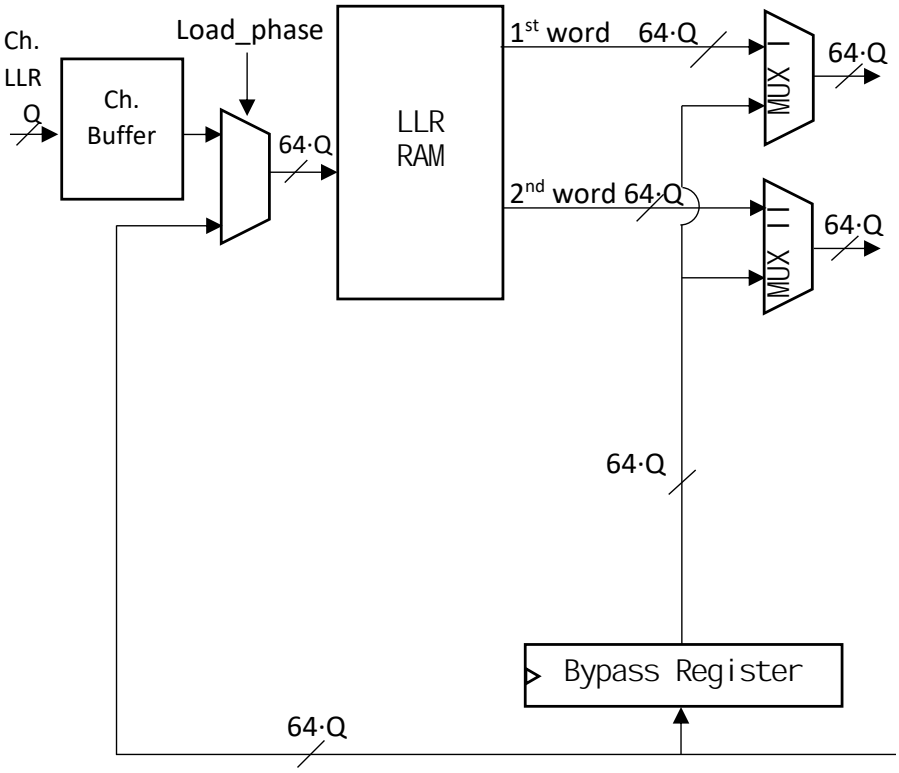


Figure 45: Timing of the LLR Memory for a PC with N=1024 and M=64

In the previous figure, the read ports of the memory are connected to two 2M·Q-to-MQ multiplexer, where the first input is a word form the memory and the second is the content of the bypass register. When $S > m$ the MUX-I and MUX-II in fig.45 output the memory words, when $S = m$ the MUX-I gives the content of the bypass buffer, MUX-II the second word of the memory. Finally, when $S < m$ the output is the data provided by the bypass register.

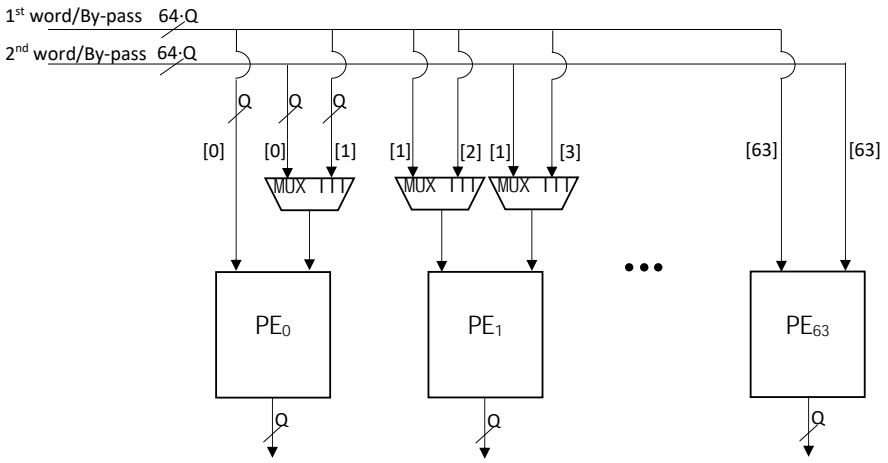


Figure 46: Timing of the LLR Memory for a PC with N=1024 and M=64

In fig.46 the multiplexers 2Q-to-Q MUX-III implement the different connections with the words previously described. Since only $\frac{M}{2}$ PEs are used when $S < m$, the PEs from 1 to 31 will require 2 MUX-III, while only the second input of PE_0 has one MUX-III since the first input is always connected to the same cell of the word in both connection configurations. Therefore, there will be a total of $M-1$ MUX-III. When $S \geq m$ the MUX-III provides the first connection type, when $S < m$ the second type. Therefore, the selector of the MUX-III can be taken from an OR gate with inputs the selectors of MUX-I and MUX-II. In the same figure, the square brackets contain the cell index of the word to which the inputs of the PEs are connected.

5.1.5 Processing Element

All the M processing elements have the same scheme, described in [8] and portrayed in fig.47.
A PE takes as inputs two LLRs (e.g. $L_A$and L_B), the 1-bit signal l_r provided by the SC controller and used to execute the computation of the left or the right message and the partial sum β used in the right message evaluation. The output of the PE is a single LLR (e.g. L_C). The LLRs are provided in sign and magnitude form since, as shown in equations eq.1 and eq.2, the operations to be performed on the LLRs, like the absolute value or the sign retrieval, are very easy to implement with the sign and magnitude format.
The magnitudes undergo a comparator which provides 1 if $|L_A| < |L_B|$, 0 otherwise. This result is used to select which is the maximum and the minimum of the two LLRs. If l_r is 0, so α_{left} is being evaluated, the last two multiplexers provide one the sign of L_C as the XOR between the signs of L_A and L_B, the other selects the minimum magnitude between $|L_A|$ and $|L_B|$. If l_r is 1, then α_{right} is being computed. In this case, the magnitude of the output will be provided by an unsigned adder where the eventual subtraction is implemented by selecting the 2s'C version of the minim magnitude input as one input of the adder, while the sign is provided by a small logic as described in [8].

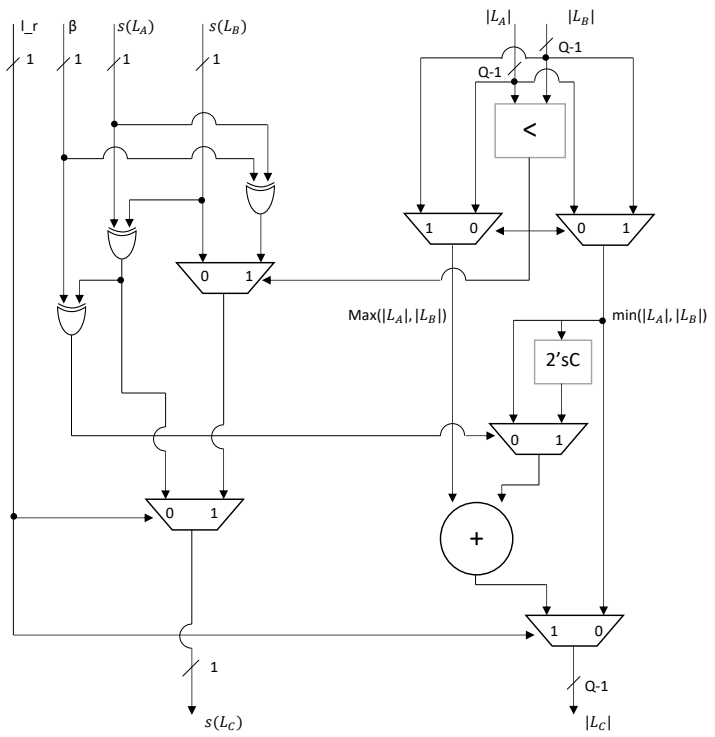


Figure 47: PE circuit

5.1.6 Channel Buffer

At the beginning of the decoding, the LLR memory is filled with the LLRs received from the channel. In particular, the LLRs are serially transmitted in groups of M, then when the buffer is full, the content is written in the proper memory word. Moreover, the channel LLRs are sent with the same order with which they are generated. However, their order in the RAM word is different, therefore it is the buffer duty to organize the LLRs to match the memory cell scheme.

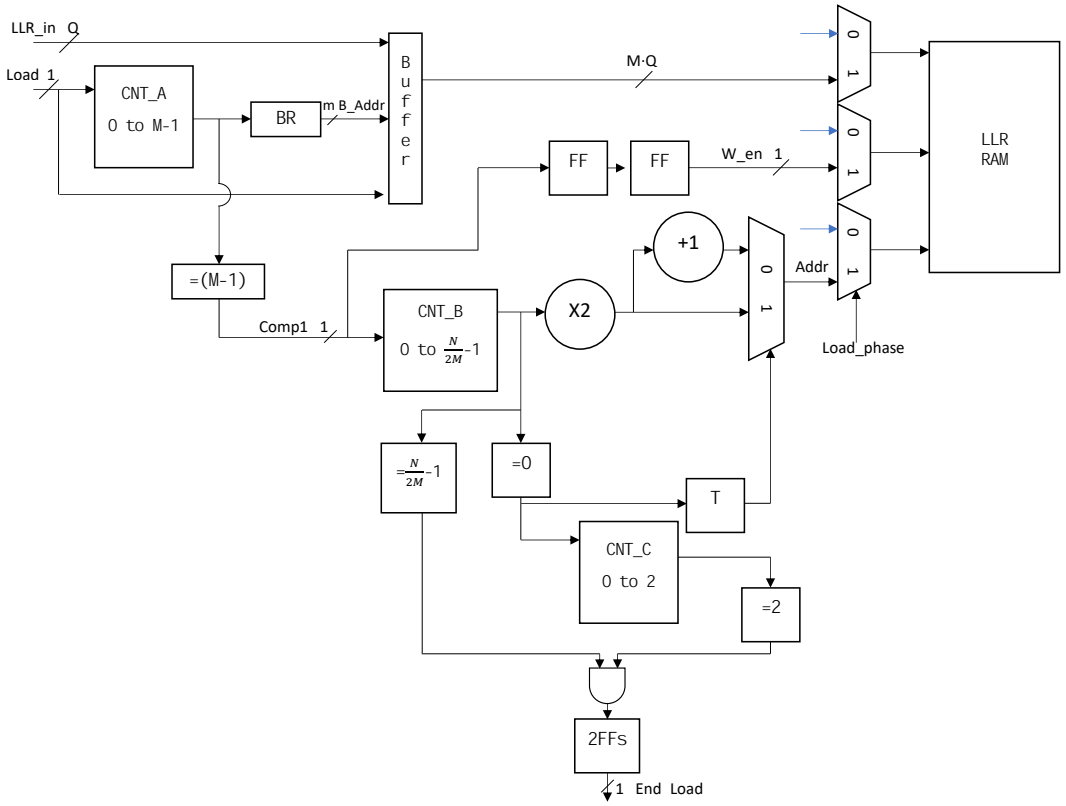


Figure 48: Channel buffer

When the decoder receives the signal *Ready* high, it means that M LLRs are serially being sent to it. The signal *Ready* activates a first counter (CNT_A in the picture) that generates the cell address where the LLR must be placed. This address is simply the bit-reverse permutation of the output of the counter. In the scheme this permutation is represented by the BR block, however, its implementation is just done by changing the order of the wires. The signal *Ready* is also used as the write enable for the buffer, which has room for M LLRs. To better understand, let us consider the memory seen in fig.41. The first LLR sent to the decoder is L_0^5, the counter gives 000, therefore the LLR is saved in the first cell of the buffer. Then for L_1^5 the output of the counter is 001, whose bit reverse permutation is 100, so this LLR is saved in the fifth cell, and so on.
When M LLRs are in the buffer, so the output of CNT_A is equal to M-1, a second counter (CNT_B in the scheme) is enabled to provide the address of the LLR RAM where the content of the buffer must be saved.
There will be a total of $\frac{N}{M}$ words to be saved, the first half will be stored in the even addresses of the RAM (LLR_RAM_Address $= 2 \cdot CNT_B$). After the first $\frac{N}{2M}$ words have been saved (so the output of CNT_B is $\frac{N}{2M} - 1$), a third counter (CNT_C) is activated and the remaining words will be saved in the odd addresses of the RAM (LLR_RAM_Address $= (2 \cdot CNT_B) + 1$). In the circuit scheme, the x2 multiplier can be implemented by simply shifting the wires, while the +1 adder by inserting 1 as LSB after the wire shifting.

When all the LLRs have been stored into the RAM, the output of the third counter is 2 and this condition is provided as an input to an AND gate together with the $CNT_B = \frac{N}{2M} - 1$ condition. The output of this gate is used to clear the third counter (not shown in the scheme) and it is also used to signal the end of the loading phase. It should be noticed that the selector of the multiplexers at the input ports of the RAM provides the values from the buffer circuit since the signal *Load_phase* is kept high during the entirety of this process.

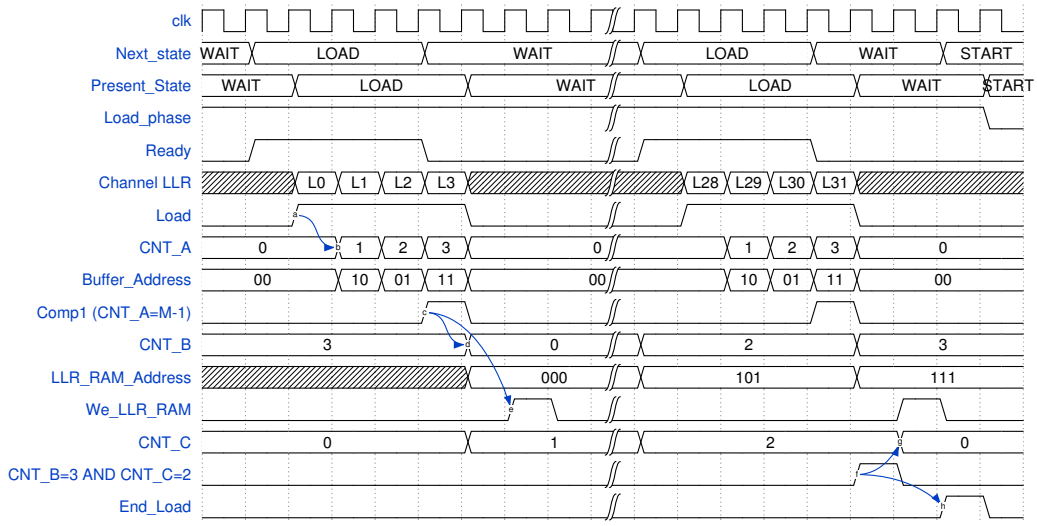


Figure 49: Timing of the channel buffer

The timing in fig.49 refers to a code with N=32 and M=4 and it is shown the storing of the first and last groups of LLRs.

5.1.7 Decisor

When the stage S=0 is reached during the SC decoding, the sign of L_0^0 is evaluated to obtain the hard decision bit. In particular, in an SC decoding, the decoded bit will be 0 if $s(L_0^0) \geq 0$ or the index i is a frozen bit, 1 otherwise.
The PBF-C decoding decisor, however, needs also to send signals to other components to get the indexes to be flipped from the memory elements, to let the sorting networks sample correctly the bit just decoded and also to flip the bit during the level 1 and level 2 phases.
The index i is also compared with the output of a LUT containing the positions of the remainder. This result is used with one clock delay as an enable for a shift register in the CRC circuit used to store the remainder bits and to provide them to the CRC computation part when the SC decoding ends.

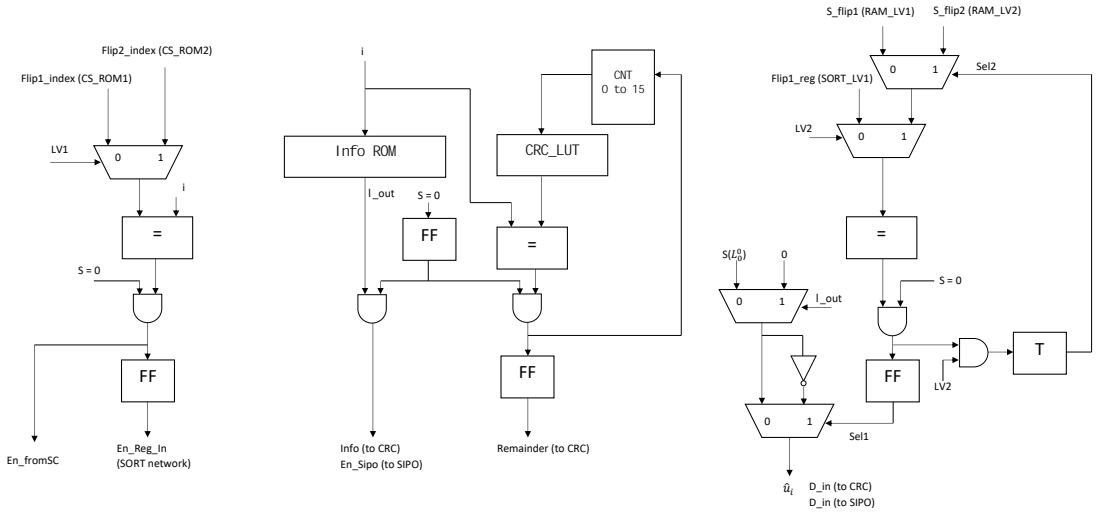


Figure 50: Decisor circuit for the PBF-C

As it can be seen in fig.50, the indexes to be provided to the sorters are taken from the ROMs, then, based on the level phase, one of them is selected to be compared to the index i and when S=0 the result, *En_fromSC* is sent to the outer counters to get new addresses for the ROMs and to the ROMs itself as read enable. A small combinational circuit on the read enable port of the two ROMs, let the memories be activated only during the proper level. This signal is also sent with one clock delay to the sorters (which again will be activated only during their proper levels using some gates). An example for this, considering a level 0 phase, is portrayed in the timing diagram in fig.51 where, as an example, the index 2 represents an info bit, while the index 3 represents a remainder bit. Both of them belongs to the critical set of the example, so the sorter will be activated.

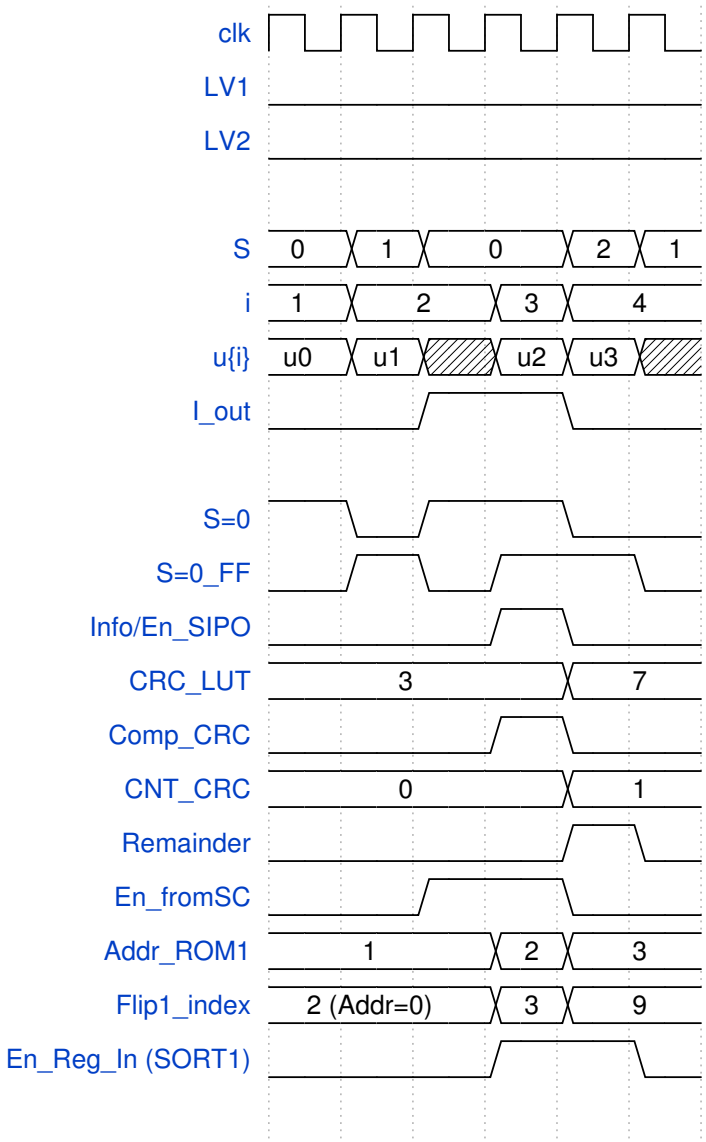


Figure 51: Decisor during level 0 phase

The index i is also sent as an address to an N-bit memory containing the positions of the information bits (in this memory the indexes of the CRC remainder are seen as frozen). If the bit selected in this way is equal to 1 (so i is an information bit), both the CRC and the SIPO will sample the bit decoded. The sign of L_0^0 is sent to a first multiplexer whose output is the sign itself or the value zero based on the output of the Info_ROM (notice that the output of the multiplexer when an info bit is being evaluated is 0 when the magnitude of L_0^0 is greater than zero, 1 otherwise). This bit goes then into a second multiplexer that will select its negated value if the flip conditions are true. As an example, during the level 2 phase, two flips will be performed: after the first one a FF toggle is activated an this will let select the second index to be flipped. After the second flip, the toggle is enabled again and its output returns low. An example of this is shown in fig.52.

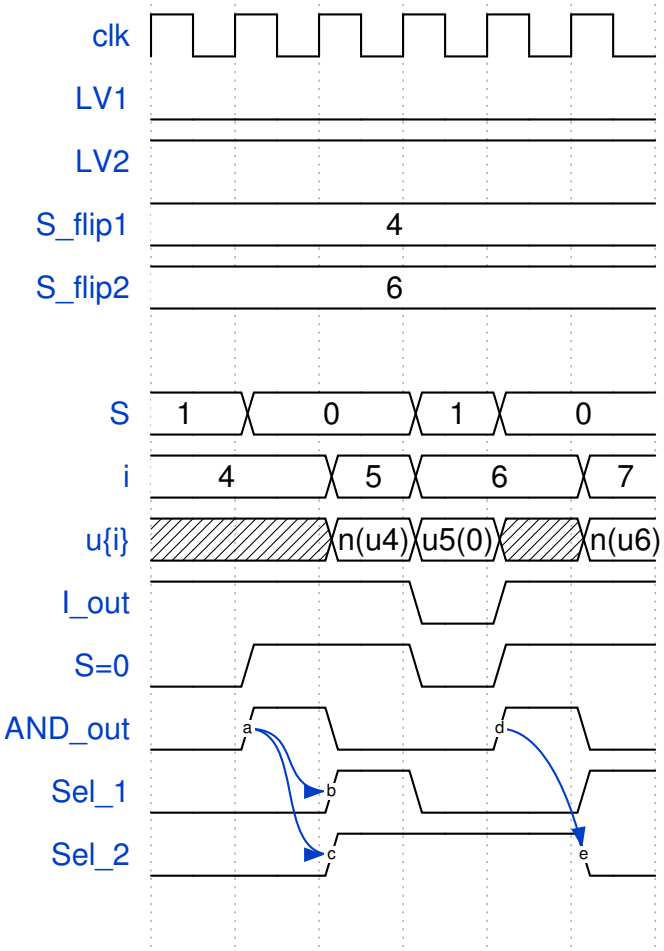


Figure 52: Decisor during level 2 phase

The timing shows that when the signal Sel1 is high the decoded bit is flipped ($\hat{u}_4 = not(u_4)$), while if the output of the Info_ROM is 0, so it is the decoded bit ($\hat{u}_5 = 0$).

5.2 CRC-16 Circuit

Since the PBF-Constrained is based on the SC-Flip decoding, a CRC remainder is encoded in the message sent to the decoder. Therefore, during the decoding phase, the CRC apply on the information bits the same polynomial used by the encoder on the original message. It should be noted that, during the encoding phase, the K-bit information vector is padded with R zeros then, this K+R-bit vector is divided by the CRC polynomial so that the last R positions contains the remainder of the CRC. Then the PC encoder transmits the K info bits on the most reliable channels and the R remainder bits on the next R most reliable channels. Doing so, the final encoded message sent to the channel will have information and remainder positions distributed over its length. In order to perform the CRC check at the decoder side, it is necessary to obtain again the structure of a vector with the first K positions

filled with information bits and the last R slots with the CRC remainder.

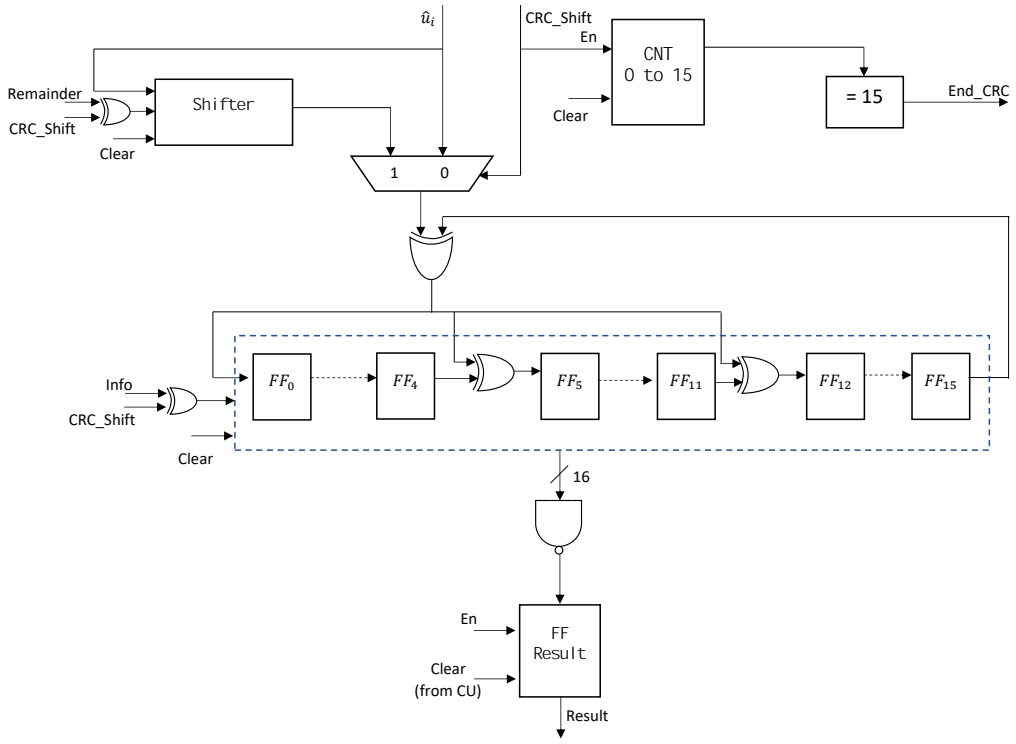


Figure 53: CRC-16 circuit

The CRC check operation has been implemented with the same structure of a LFSR, so there are 16 FFs with some XOR gates between some of them to implement the polynomial

$$x^{16} + x^{12} + x^5 + 1.$$

This implementation fits well with the way the inputs are provided to the CRC circuit: since the LFSR accepts one bit per clock cycle, whenever an information bit is decoded, it is sent to the LFSR structure together with the signal *Info* used as an enable. On the other side, when a bit associated with a remainder position is decoded, it is sent to a 16-bit FIFO instead: at the end of the SC decoding, there is a period of 16 clock cycles in which the signal *CRC_Shift* is high and used as an enabler for both the FIFO and the LFSR so that the content of the FIFO is sent to the LFSR scheme. After this period, a counter will signal the end of the CRC evaluation, the content of all the FFs of the LFSR is checked with a NAND gate: if all the registers contain a zero, the check is considered successful and the decoded bits are considered correct. Otherwise, the CRC check is considered failed and a new flip attempt is performed. In both cases, the FIFO and the LFSR are cleared with the signal of En_FF provided by the CU to sample the result which is in turn cleared by the control unit when the total decoding ends.

5.3 Insertion Sorter

In the PBF-Constrained scheme, there 2 networks to sort the LLRs: one to sort the dim_CS LLRs of the critical set of the level 0, the second to sort the T LLRs of the critical sets of the level 1. It has been chosen to use two networks instead of one so that it is possible to start immediately the level 1 decoding by taking the index to be flipped from the sorter without waiting to transfer the content of the sorter to a RAM.
As seen in the implementation of the SC-Flip decoder provided in [11], the sorting networks implement the insertion sorting algorithm since their inputs (the index i and the absolute value of the LLR) are provided at most one per clock cycle.
The control signal Mode is 0 during the SC decoding to get new data from the decoding phase, while it is 1 when the index content of the first register Reg_0 has to be provided to the SC module or it has to be saved in a RAM.

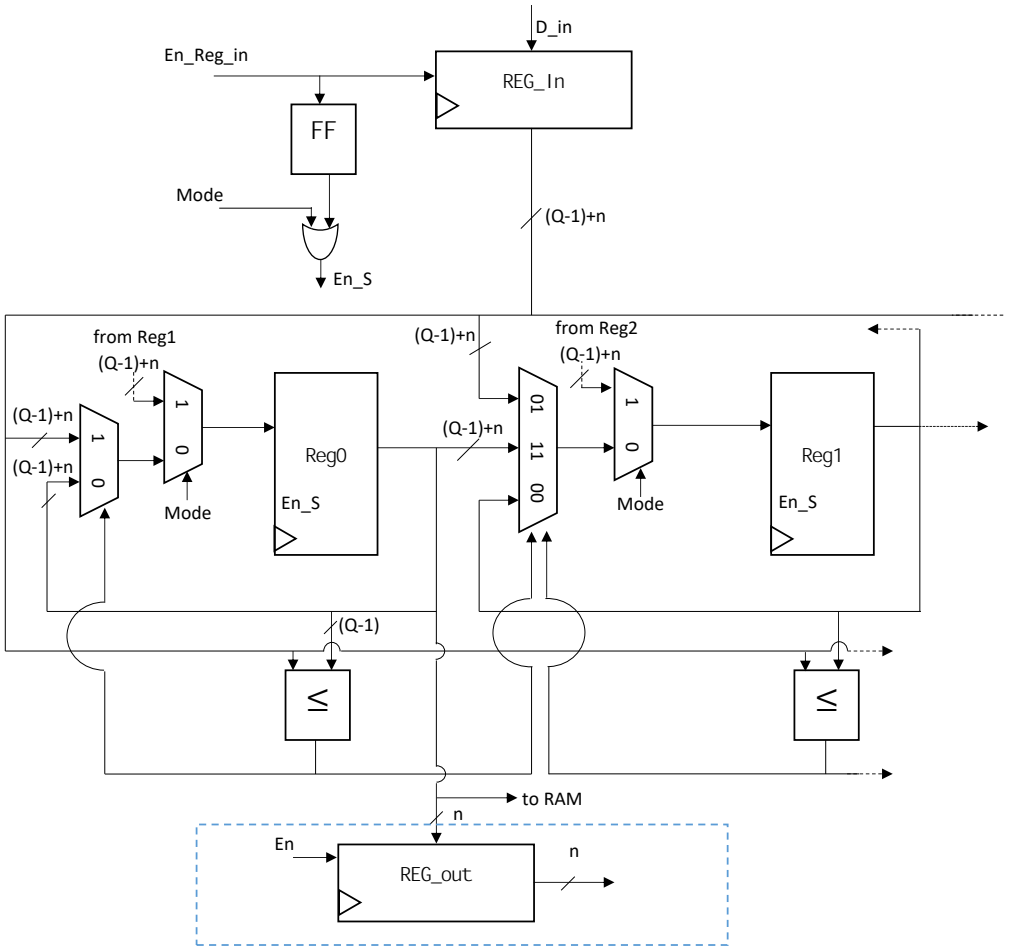


Figure 54: Insertion sorter scheme

The insertion sorting scheme is made of dim_CS registers for the network used to sort the critical set of the level 0, while of T registers for the second circuit. All the LLRs are sorted in the same clock cycle every time a new data has to be

inserted. These networks are used to sort the absolute values of the LLRs (Q-1 bits) of a critical set in ascending order. Moreover, since what to be flipped is an index, the registers have a parallelism of (Q-1)+n bits to accommodate also the index i corresponding to the LLR.
When a bit belonging to a critical set is decoded, the index i and the magnitude of the LLR associated with that bit are sent to the sorting network. Here, they are saved in a first register called Reg_In, then the LLR content is compared in parallel with the LLR content of all the registers of the network through comparators: if the result of the comparison is 1, then the new LLR is smaller than the LLR stored in that register, the opposite if the result is 0. Therefore, let us consider the results of the comparators of three consecutive registers when a new data is introduced to the network: 100 $\rightarrow$ the new data is greater than the one in the first register and smaller than the ones in the other two registers: the first register keeps its content, the new data is written into the second register and the content of the second register is passed to the third register.
When S_Mode = 1, the index i stored in the first register (Reg_0) is provided to a RAM and each register sends its content to its left register. It should be noted that S_Mode = 1 is also used to enable the registers (with the exception of the Reg_In) and it is sent by the CU. The sorting network for the level 0 (Sort1) receives S_-Mode1 = 1 as a pulse signal each time a new flip has to be performed. In this way, the content of the Reg_0 is stored into a RAM (RAM1) and it is also kept into the Reg_out (this register is not present in the second sorter), then Reg_0 is updated with the content of Reg_1. As for the second sorter (Sort2), S_Mode2 = 1 is sent as a continuous signal at the end of each flip attempt of the level 1.

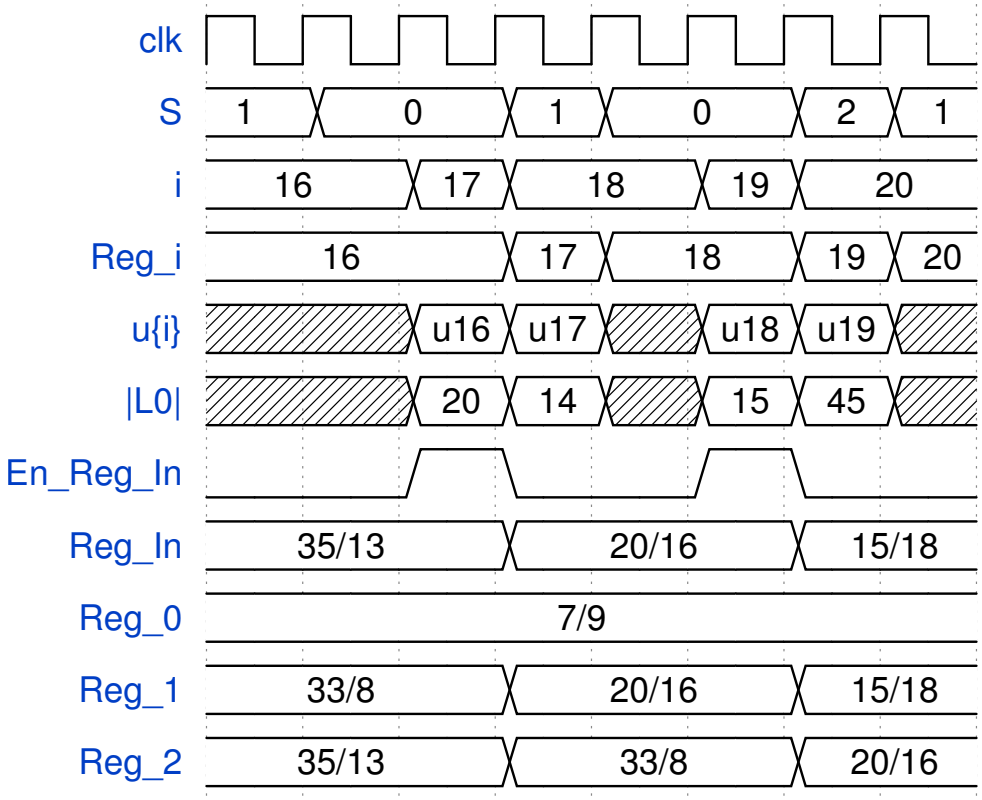


Figure 55: Timing of the insertion sorter when S_Mode = 0, with positions 16 and 18 belonging to a critical set

In the timing diagram, the content of the registers is provided as the value of $|L_0|$ / the value of i, where the symbol "/" is used just to show the separation of the two fields.

5.4 PBF-C Level 0

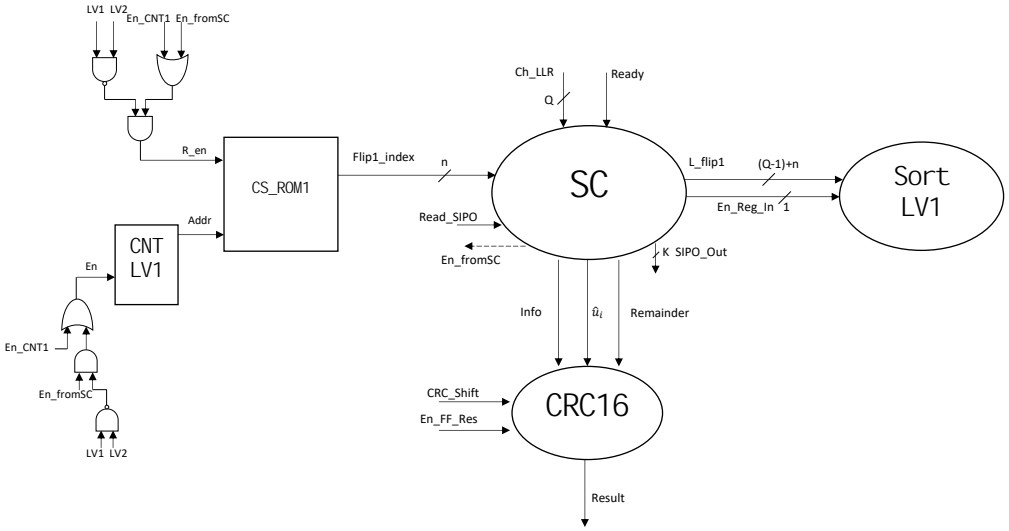


Figure 56: PBF-C level 0 architecture

At the beginning of the decoding, a standard SC is performed on the LLRs provided by the channel. During this operation, the content of the ROM storing the critical set of the level 0 (CS_ROM1) is read and when the signal i has the same value of the data read from the ROM, the absolute value of the LLR corresponding to that index for S=0 is sent to the sorting network SORT_LV1 along with an enable and the address of the ROM is increased.
When the SC decoding ends, all the dim_CS LLRs of the critical set are inside the sorting circuit. Then, when the CRC check ends, the FF containing the result is read. If the result is 1 the SC decoding is considered successful, the SIPO outputs its content and then a clear state is reached where the sorting networks and other registers are brought back to their initial value.
Finally, the FSM remains in the WAIT state until a new decoding operation starts. Otherwise, if the result of the CRC is 0, the level 1 is reached.

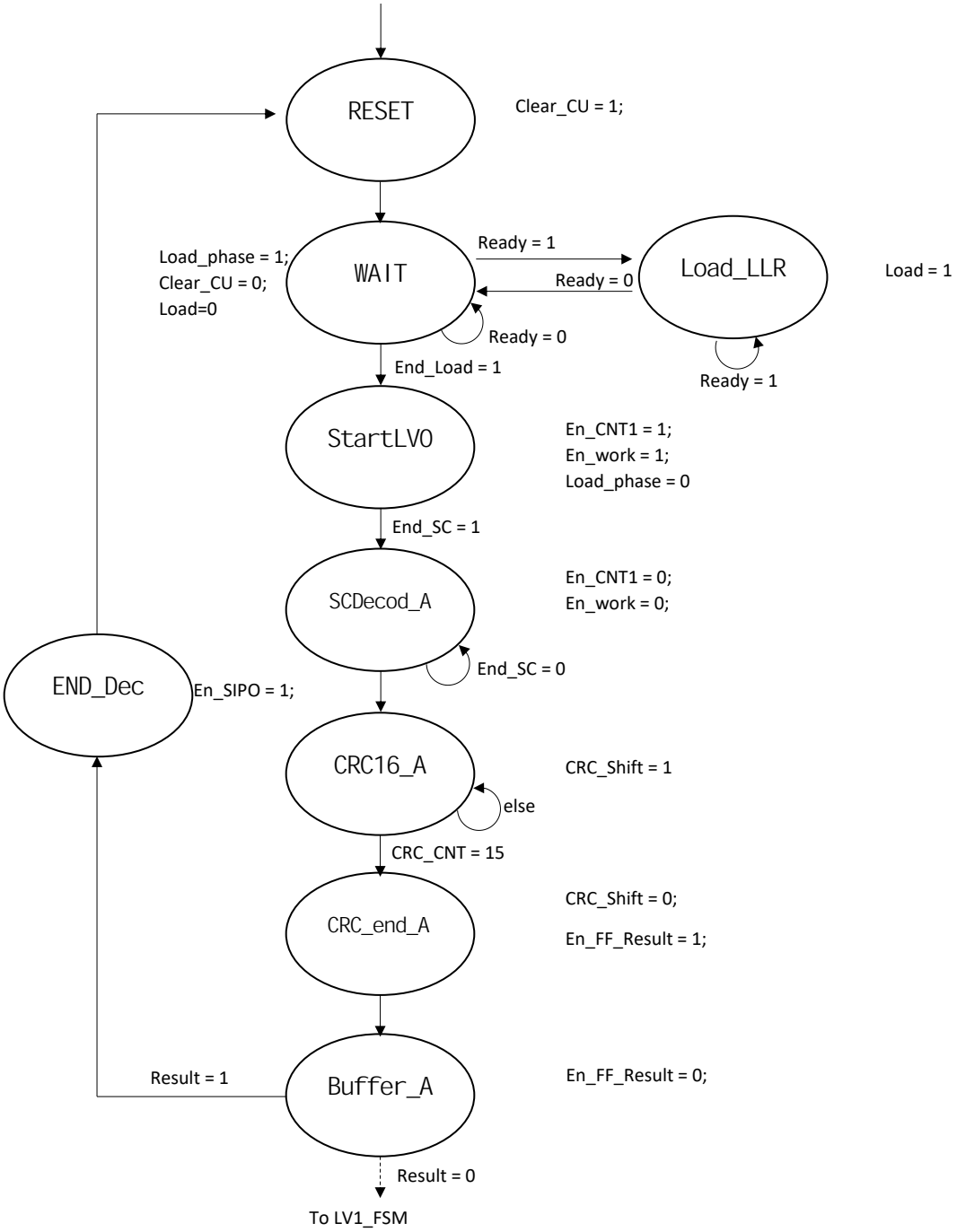


Figure 57: FSM for the LV0 part

While the first part of this FSM has already been described in 5.1.6, the remaining part is shown in the following timing diagram.

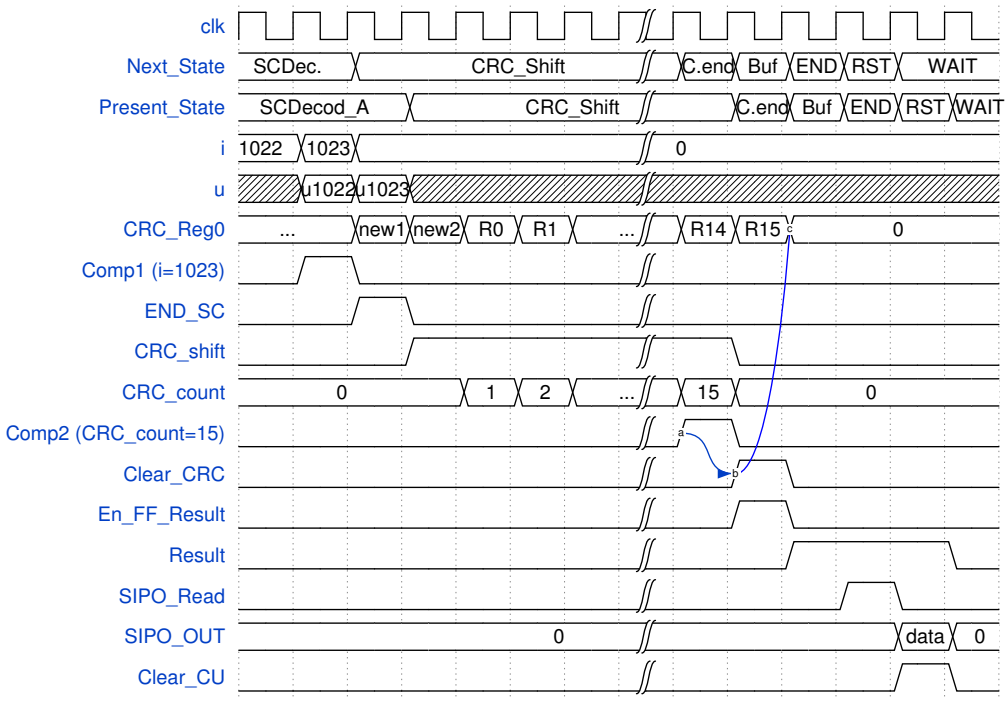


Figure 58: Timing of the LV0 with successful decoding

The timing diagram, in fig.58 portrays the case where the decoding ends correctly, therefore the content of the FF_Result is 1 and the End_Dec state is reached enabling the read of the SIPO. In this diagram the SIPO is cleared just after one clock cycle by the Clear command from the CU, however, it is simple to implement an architecture able to keep the SIPO content for a longer time.

5.5 PBF-C Level 1

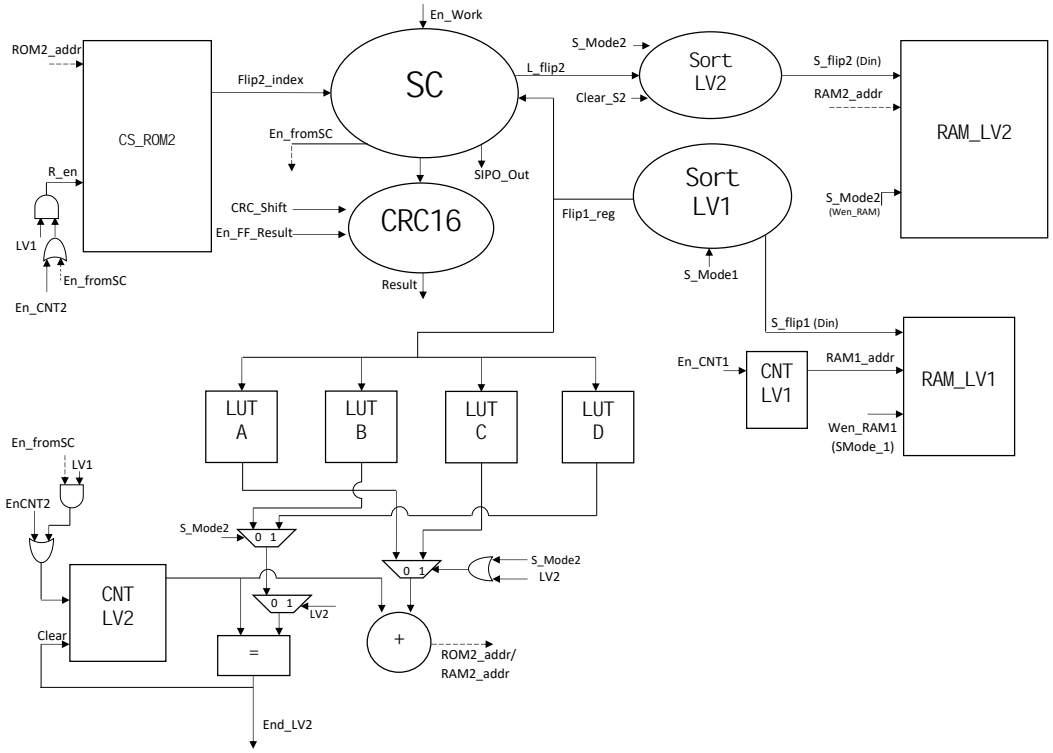


Figure 59: Scheme of the level 1 phase

When the standard SC decoding fails, the FSM starts the level 1 phase. Here, for each decoding attempt, the least reliable index, stored in Reg0 of the network SORT_LV1, is used to flip the position corresponding to it and it is also saved into RAM_LV1, in this way, if all the dim_CS attempts of the level 1 fail, this RAM will store the first indexes to be flipped during the level 2 phase. The index to be flipped is kept in the output register of SORT_LV1 and it is used to access the ROM containing the critical set of said index. This is done by the mean of some LUTs: LUT_A stores the first address of ROM_LV2 containing the first element of the critical set of the index, while LUT_B provides the dimension of said set so that only the correct amount of LLRs are sent to the second sorting network. The LUT_C is used to get the first address of the RAM_LV2 where the index must be saved, while LUT_D provides the number between 0 and T-1 of indexes that will be saved in the RAM_LV2 for that critical set. The addresses for the ROM_LV2 and the RAM_LV2 is obtained by the sum of the output of the counter CNT_LV2 and the content of the LUT_A (for the ROM) or of the LUT_C (for the RAM). The input of the adder is chosen with a multiplexer whose selector is the output of an OR gate with inputs LV2 (since during the level 2 phase only LUT_C and LUT_D are used) and S_Mode2 (it is 1 during the writing phase in the RAM_LV2, in this way the LUT_C is selected as input for the adder). If the parallelism of the RAM_LV2 address is lower than the one of the address of the ROM_LV2, the proper MSBs of

the output of the adder are not connected to the address port of the RAM. The counter is cleared during the SC decoding when its output is equal to the content of LUT_B or to the output of LUT_D during the writing in the RAM_LV2 phase, therefore the selector of the multiplexer for the comparator is the same used for the multiplexer for the input of the adder.
To better understand the structure of the memories and of the LUTs, let us consider, as an example, the case of PBF-C applied to an N=16 PC with a number of attempts at the second level equal to T=2.

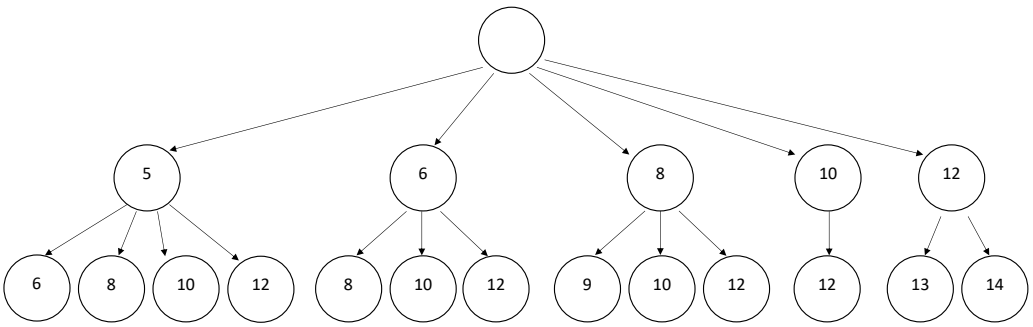


Figure 60: PBF for a PC with N=16

In fig.60 is portrayed the PBF of an N=16 PC with critical set with dimension 5 for the level 0, then each node will be constrained to a maximum of 2 child nodes with the PBF-C approach.
During the level 0 phase, the content of the CS_ROM1 is sorted in the SORT_LV1 network, then, with each decoding attempt, the RAM_LV1 is filled with the sorted order of indexes. The content of the CS_ROM1 and the RAM_LV1 (at the end of the level 1 phase) is shown below.

CS_ROM1
5
6
8
10
12

CS_RAM1
8
6
5
10
12

Figure 61: Contents of CS_ROM1 and RAM_LV1

The critical sets of each index contained in the CS_ROM1 is stored in the CS_-ROM2. The first address of each critical set block is the content of LUT_A. Since a maximum of 2 attempts are tried, the dimension of the RAM_LV2 is smaller compared to the ROM. Here, the first addresses of the critical sets are provided by LUT_C.

	CS_ROM2	Address
CS(5)	6	0
	8	1
	10	2
	12	3
CS(6)	8	4
	10	5
	12	6
CS(8)	9	7
	10	8
	12	9
CS(10)	12	10
CS(11)	13	11
	14	12

	CS_RAM2	Address
CS(5)	6	0
	8	1
CS(6)	8	2
	10	3
CS(8)	9	4
	10	5
CS(10)	12	6
CS(11)	13	7
	14	8

Figure 62: Contents of CS_ROM2 and RAM_LV2

The content of the LUTs is shown below.

LUT A	
IN	OUT
5	0
6	4
8	7
10	10
12	11

LUT B	
IN	OUT
5	5
6, 8	4
10	3
12	2

LUT C	
IN	OUT
5	0
6	2
8	4
10	6
12	7

LUT D	
IN	OUT
5, 6, 8, 12	1
10	0

The FSM flow is shown in fig.63

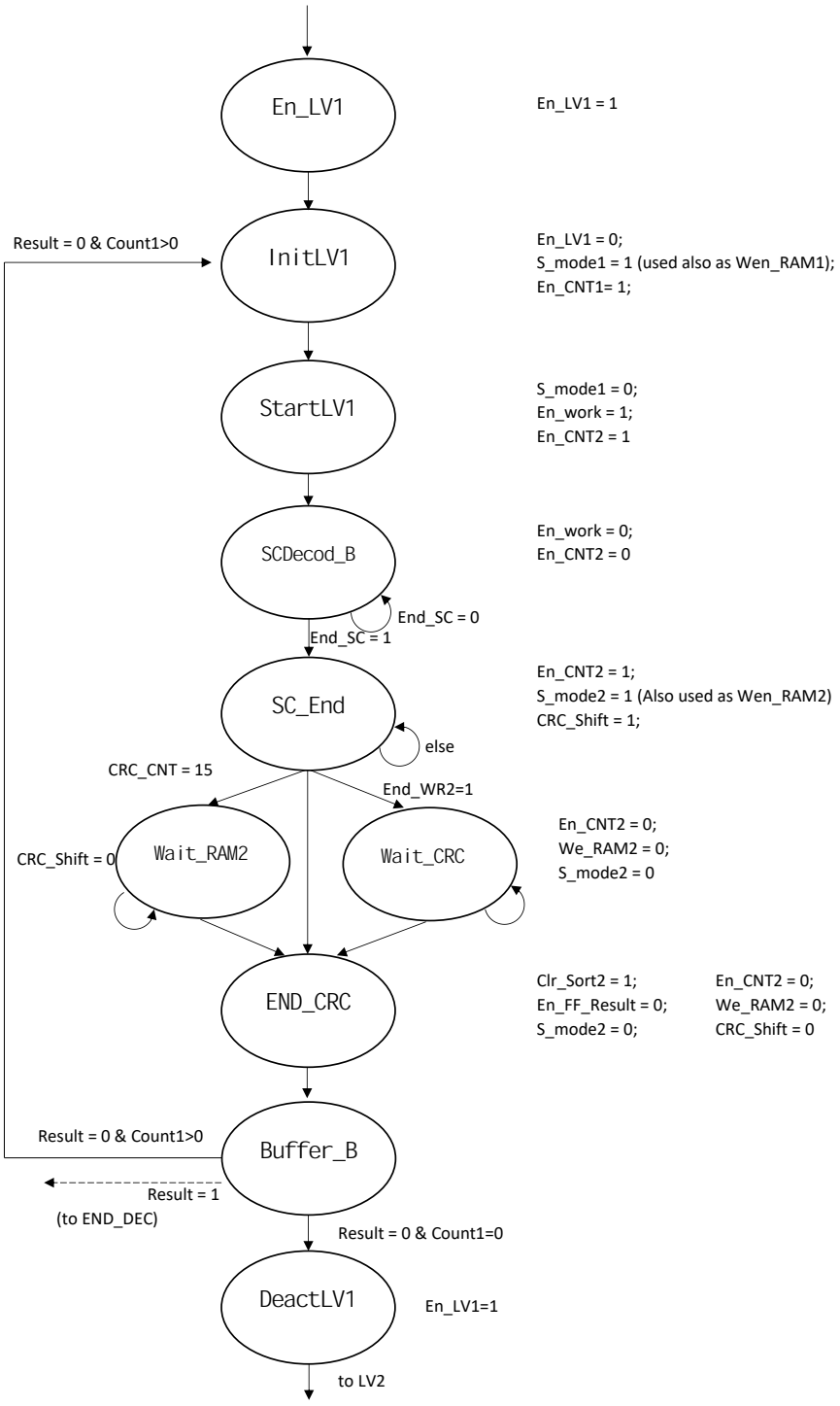


Figure 63: FSM of the level 1 phase

Compared to the previous FSM for the level 0, here, before the SC decoding, there are some states used to set the proper signals and to provide the index to be flipped. The signal LV1 is used as input of many combinational logics at the input

of enable ports and, it will be brought back to 0 only when all the dim_CS attempts of the level 1 fail.
After the LV1 signal has been set, one index is taken from the SORT_LV1 network and provided as data (S_flip1) to be written in the RAM_LV1, to the SC module, so that, when i=Flip1_reg, the hard decoded bit is flipped, and to the LUTs that will provide the signals to access the CS_ROM2 and to clear the counter. The address of the RAM_LV1 is given by CNT_LV1 enabled together with En_CNT1 as a pulse. In the following state, the first index of the critical set for the second level is taken from the output of CS_ROM2 and it is kept stable since the read enable of this ROM is then lowered. In the same state (StartLV1) the signal En_work is brought high enabling the start of the SC decoding (more precisely, it will be an SC-Flip decoding since the index provided by Flip1_reg will be flipped). During the decoding, the sorting network SORT_LV2 orders the combination of LLR+index in ascending order based on the magnitude of the LLR.
At the end of the decoding, the CRC and the RAM_LV2 writing phase starts simultaneously, so CRC_Shift and S_Mode2 are brought high. If one of them finishes before the other (as an example, the writing process will require more latency when the dimension of the critical set of the index that is being flipped is greater than the length of the CRC) the FSM waits for the other operation to be completed. The end condition of the writing phase is given when the signal End_LV2 is high (so when the output of CNT_LV2 is equal to the content of the LUT_D).
After this, the registers in the SORT_LV2 network are cleared (this operation was not required for the SORT_LV1 circuit since it is used only once) and the result of the CRC is checked. If the CRC failed, a new attempt is performed, the CNT_LV1 is updated and the CS_ROM1 provides the next least reliable index to be flipped. During the last attempt, the CNT_LV1 provides 0. If the CRC fails again, the FSM will deactivate the LV1 signal and the level 2 phase starts.
The following diagrams describe these phases for the PC with N=16 seen in the previous example.

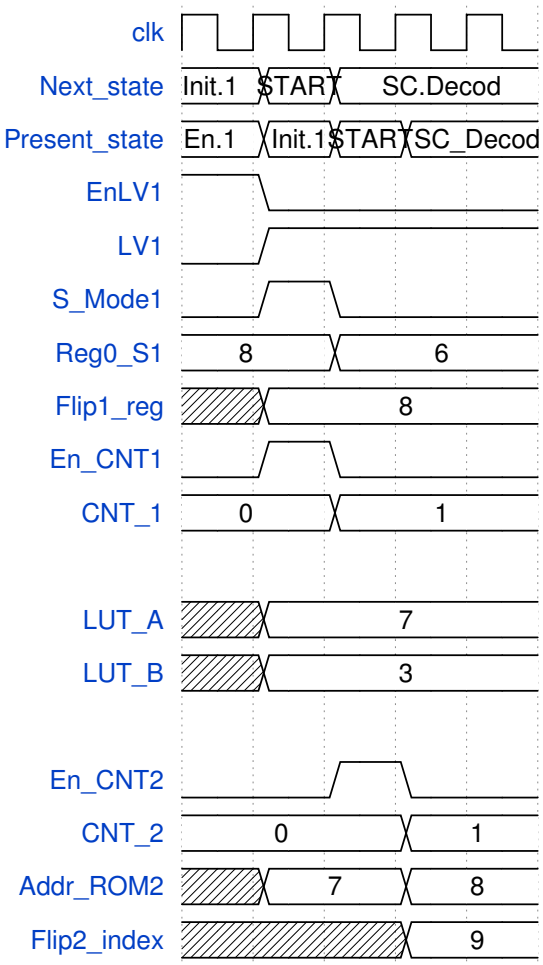


Figure 64: Timing of the first phase of the level 1 for the N=16 PC

In this timing diagram it can be noticed that the content of the REG0 (8) of the first sorting network is written into the RAM_LV1 (the write enable is the same S_-Mode1 signal used to take the data from the sorting network) in the address provided by the output of the CNT_LV1. In a similar manner, the index to be sorted during the SC phase is taken using as the address of the CS_ROM2 the sum of LUT_A and the output of CNT_LV2. During the SC phase, the enable of the CNT_LV2 is provided by a signal from the SC module when i is equal to Flip2_index and S=0.

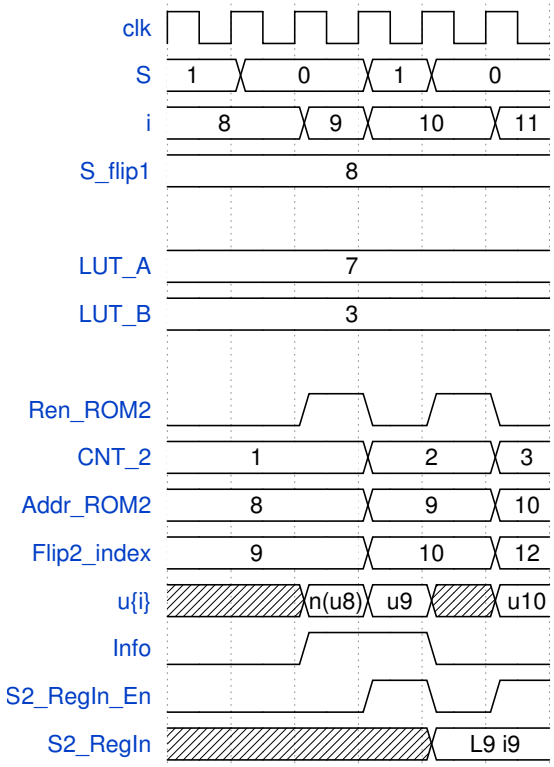


Figure 65: SC decoding during the level 1 phase

During the SC phase, the bit associated with the index equal to S_flip1 is flipped (n(u8)) and a new Flip2_index is taken when i=Flip2_index and S=0.

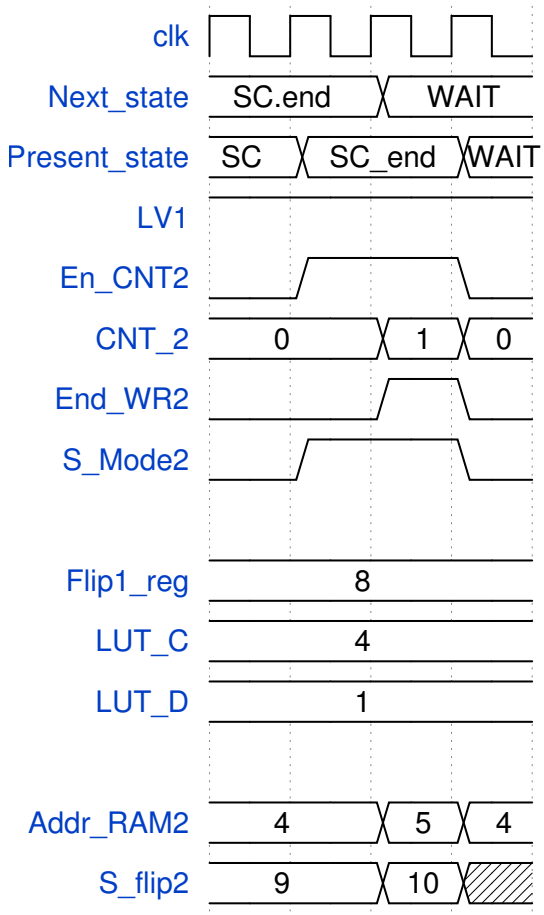


Figure 66: Writing in the RAM_LV2 phase

When the SC ends the CNT_LV2 has already been cleared by its comparison with LUT_B, therefore its output is 0. Then, the counter is used with the LUT_C to obtain the addresses where the sorted indexes have to be stored. When CNT_LV2 gives the same value found in LUT_D, the writing phase ends.

5.6 PBF-C Level 2

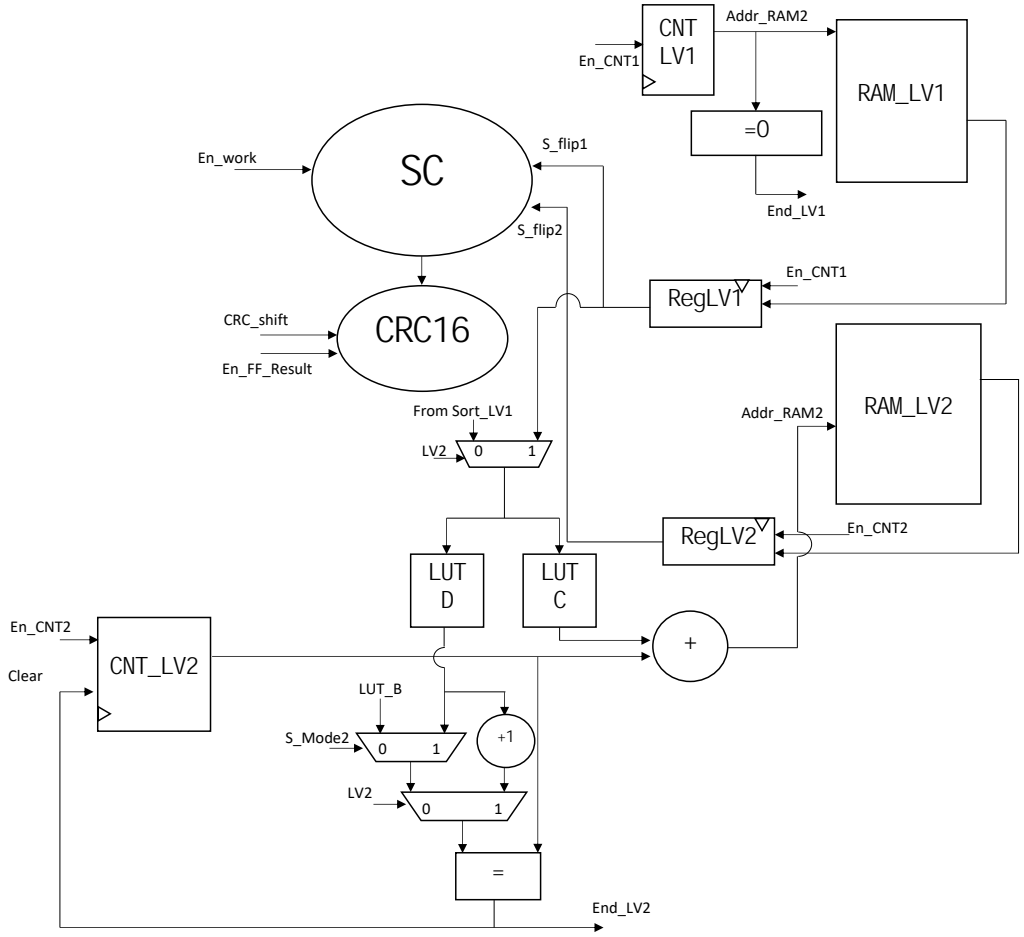


Figure 67: Scheme of the level 2 phase

After all the dim_CS flip attempts of the level 1 have failed, for each of them a maximum of T double flip tries are performed during the level 2 phase.
Now, all the indexes to be flipped are provided by the two RAMs. The first flip index is read from RAM_LV1 and it is sent to the LUTs LUT_C, to get the address of the RAM_LV2, where its constrained critical set is stored, and to LUT_D to understand how many attempts have to be performed for that index.

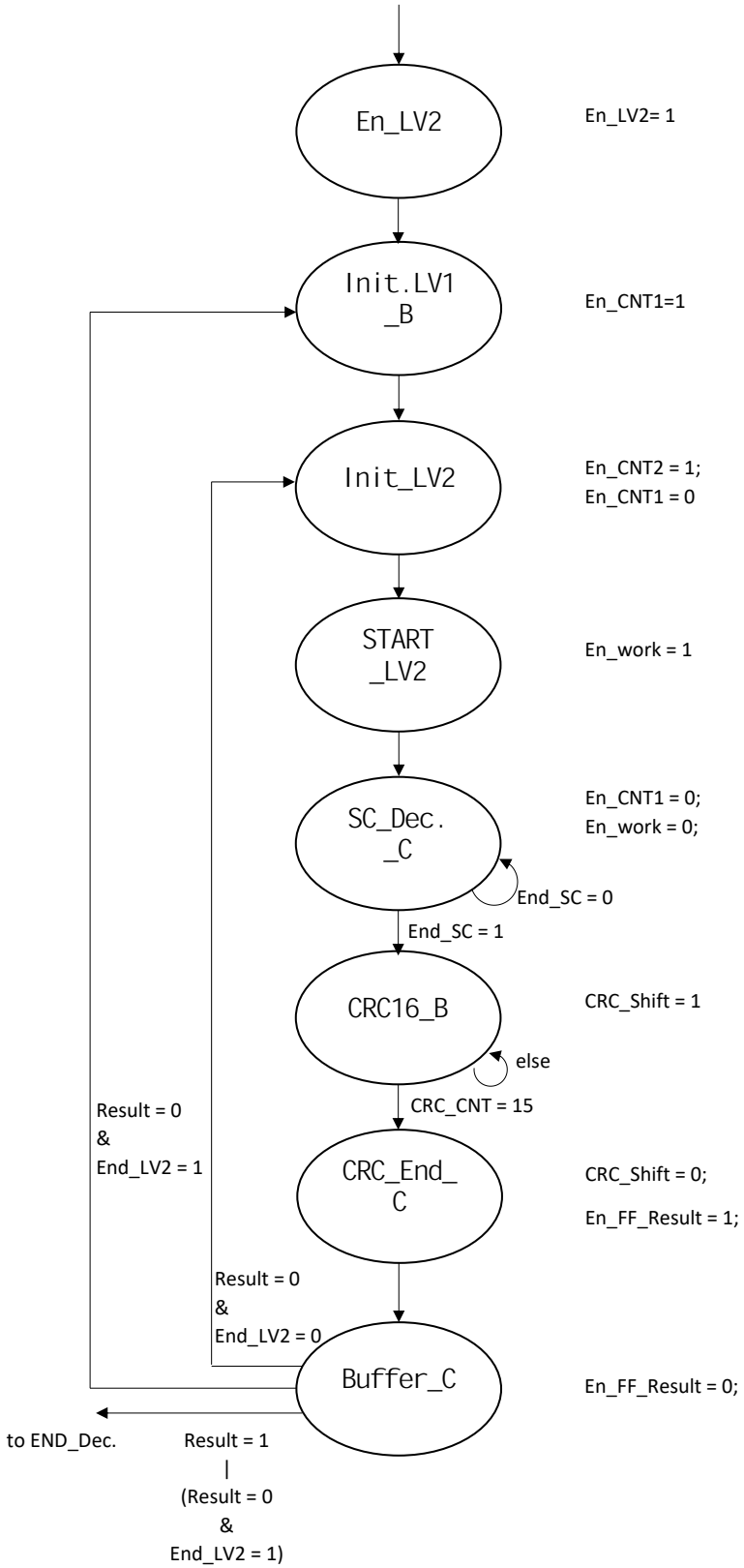


Figure 68: FSM of the level 2 phase

As shown in the FSM in fig.68, when a decoding attempt fails, the first flip index is kept, while the second one is taken from the new read from ROM_LV2 by increasing the CNT_2. When the critical set of the index of the level 1 has been completely explored (End_LV2 = 1) and the CRC still fails, a new index is taken from the ROM_LV1 and then it is used to get the second flip to be performed from the RAM_LV2.
When both CNT_1 and CNT_2 reach 0 (End_LV1=1, End_LV2=1) and the result from the CRC check is zero, the content of the SIPO is read and the result of the CRC, provided as an output of the whole decoder, is zero, signalling the failure of the decoding process and the architecture waits for a new set of LLRs from the channel to start the decoding of a new message.

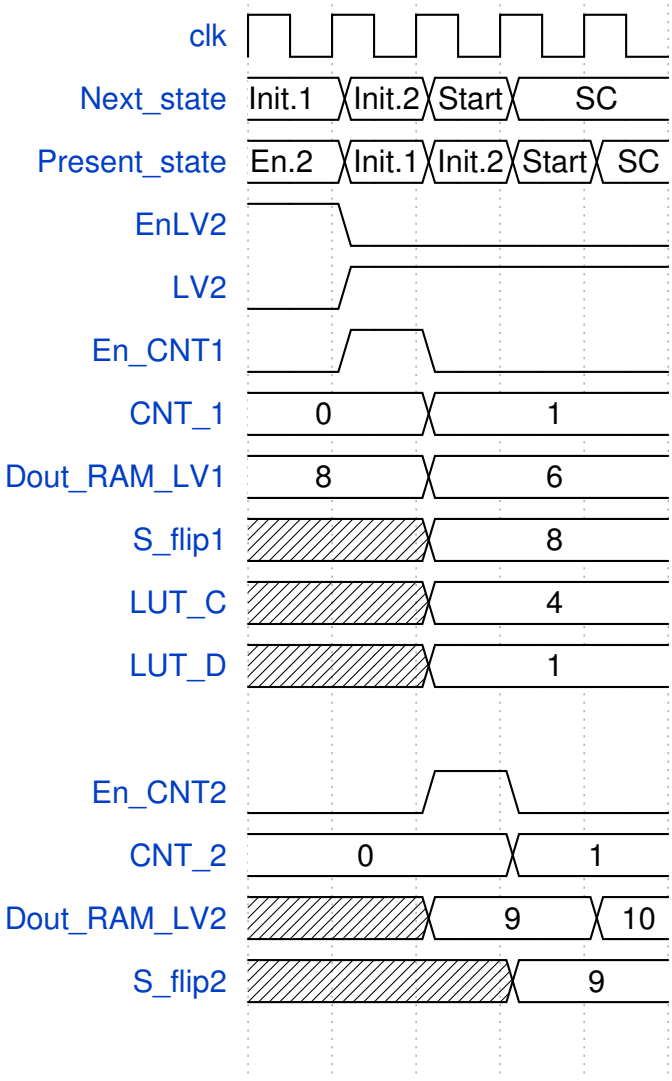


Figure 69: Timing of the second level phase

The timing in fig.69 shows the acquisition of the indexes to be flipped at the start of the level 2 phase. Notice that these indexes are taken from two registers,

while the RAMs output the following flips.

5.7 Memory Resources Considerations

A preliminary evaluation of the memory resources needed to implement the proposed architecture has been carried out considering the number of bits required by the main blocks.
The dimensions of the ROMs and the RAMs depend on the number of elements found in the critical sets, therefore on the positions of the unfrozen bits. This makes the estimation of their rows quite difficult, while the number of columns has the same parallelism of the signal i since these memories contain indexes.

Memory required [Kb] by the main blocks			
Hardware Resources	Code Properties		
	K=512 dim_CS = 125 T=32	K=512 dim_CS = 125 T=16	K=768 dim_CS = 96 T=32
CS_ROM1	12.50	12.50	0.96
RAM_LV1	12.50	12.50	0.96
CS_ROM2	78.93	78.90	47.11
RAM_LV2	35.40	19.06	2.62
SORT_LV1	2.14	2.14	1.50
SORT_LV2	0.54	0.27	0.51
LUT_A	2.00	2.00	1.25
LUT_B	2.00	2.00	1.25
LUT_C	1.50	1.30	1.15
LUT_D	0.38	0.18	0.38

Table 2: Preliminary memory resources estimation

Table 2 shows a first estimation for three different PCs with N=1024. The ROM used for storing the critical sets of the first level (CS_ROM2) is the one storing the highest amount of bits. However, it should be taken into account that the area used by a ROM is approximately 1/6 of a RAM with the same dimensions. Therefore a proper implementation of the architecture should be carried out in the future to better understand if this decoder can compete with the SCL decoder.

6 Conclusion

Polar codes have been receiving attentions due to their properties and their use in the upcoming 5G standard, however, the SCL decoder, known to provide the best performances, is characterized by a high complexity in terms of the required resources, so an alternative to it is being researched.
In this thesis, a software implementation of the SC, SCF, and PBF decoding algorithms has been provided and their curves have been discussed, then the PBF-C, a new method that simplifies the PBF scheme, has been presented showing that with T=32 it gives small loss in performance compared to the standard PBF but simplifying the pruning technique and allowing to simulate past 2.5 dB.
An architecture for the PBF-C has been proposed where the SC module has been designed with the HPPSN proposed in [9]. Moreover, this architecture is the first one in literature based on the SCF scheme that is able to correct more than one error.
Finally, a preliminary study on the required resources has been carried out to understand the possible margin compared with an SCL implementation, however, more accurate studies must be carried out in the future in order to confirm if this decoding scheme could represent an alternative to the SCL decoder.

Appendix: Parallel PSN - PEs Interface

PE1	i=2 S=1	i=4 S=2	i=6 S=1	i=8 S=3	i=10 S=1	i=12 S=2	i=14 S=1	i=16 S=4
	R1	R2	R5	R4	R9	R10	R13	R8
	i=18 S=1	i=20 S=2	i=22 S=1	i=24 S=3	i=26 S=1	i=28 S=2	i=30 S=1	i=32 S=5
	R17	R18	R21	R20	R25	R26	R29	R16
	i=34 S=1	i=36 S=2	i=38 S=1	i=40 S=3	i=42 S=1	i=44 S=2	i=46 S=1	i=48 S=4
	R33	R34	R37	R36	R41	R42	R45	R40
	i=50 S=1	i=52 S=2	i=54 S=1	i=56 S=3	i=58 S=1	i=60 S=2	i=62 S=1	i=64 S=6
	R49	R50	R53	R52	R57	R58	R61	R32

PE2	i=4 S=2	i=8 S=3	i=12 S=2	i=16 S=4	i=20 S=2	i=24 S=3	i=28 S=2	i=32 S=5
	R1	R2	R9	R4	R17	R18	R25	R8
	i=36 S=2	i=40 S=3	i=44 S=2	i=48 S=4	i=52 S=2	i=56 S=3	i=60 S=2	i=64 S=6
	R33	R34	R41	R36	R49	R50	R57	R16
PE3	i=4 S=2	i=8 S=3	i=12 S=2	i=16 S=4	i=20 S=2	i=24 S=3	i=28 S=2	i=32 S=5
	R3	R6	R11	R12	R19	R22	R27	R14
	i=36 S=2	i=40 S=3	i=44 S=2	i=48 S=4	i=52 S=2	i=56 S=3	i=60 S=2	i=64 S=6
	R35	R38	R43	R44	R51	R54	R59	R48

PE4	i=8 S=3	i=16 S=4	i=24 S=3	i=32 S=5	i=40 S=3	i=48 S=4	i=56 S=3	i=64 S=6
	R1	R2	R17	R4	R33	R34	R49	R8
PE5	i=8 S=3	i=16 S=4	i=24 S=3	i=32 S=5	i=40 S=3	i=48 S=4	i=56 S=3	i=64 S=6
	R5	R10	R21	R20	R37	R42	R53	R40
PE6	i=8 S=3	i=16 S=4	i=24 S=3	i=32 S=5	i=40 S=3	i=48 S=4	i=56 S=3	i=64 S=6
	R3	R6	R19	R12	R35	R38	R51	R24
PE7	i=8 S=3	i=16 S=4	i=24 S=3	i=32 S=5	i=40 S=3	i=48 S=4	i=56 S=3	i=64 S=6
	R7	R14	R23	R28	R39	R46	R55	R56

PE8	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R1	R2	R17	R4
PE9	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R5	R10	R21	R20
PE10	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R3	R6	R19	R12
PE11	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R7	R14	R23	R28
PE12	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R1	R2	R17	R4
PE13	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R5	R10	R21	R20
PE14	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R3	R6	R19	R12
PE15	i=16 S=4	i=32 S=5	i=48 S=4	i=64 S=6
	R7	R14	R23	R28

References

[1] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 623–656, Oct 1948.

[2] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.

[3] I. Tal and A. Vardy, "List decoding of polar codes," *IEEE Transactions on Information Theory*, vol. 61, pp. 2213–2226, May 2015.

[4] O. Afisiadis, A. Balatsoukas-Stimming, and A. Burg, "A low-complexity improved successive cancellation decoder for polar codes," 2014.

[5] Z. Zhang, K. Qin, L. Zhang, H. Zhang, and G. T. Chen, "Progressive bit-flipping decoding of polar codes over layered critical sets," *arXiv preprint arXiv:1712.03332*, 2017.

[6] L. Chandesris, V. Savin, and D. Declercq, "An improved scflip decoder for polar codes," 2017.

[7] C. Leroux, A. J. Raymond, G. Sarkis, I. Tal, A. Vardy, and W. J. Gross, "Hardware implementation of successive-cancellation decoders for polar codes," *Journal of Signal Processing Systems*, vol. 69, pp. 305–315, Dec 2012.

[8] C. Leroux, A. J. Raymond, G. Sarkis, and W. J. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *IEEE Transactions on Signal Processing*, vol. 61, no. 2, pp. 289–299, 2013.

[9] Y. Fan and C.-Y. Tsui, "An efficient partial-sum network architecture for semi-parallel polar codes decoder implementation," *IEEE Transactions on Signal Processing*, vol. 62, no. 12, pp. 3165–3179, 2014.

[10] Z. Zhang, K. Qin, L. Zhang, and G. T. Chen, "Progressive bit-flipping decoding of polar codes: A critical-set based tree search approach," *IEEE Access*, vol. 6, pp. 57738–57750, 2018.

[11] P. Giard, A. Balatsoukas-Stimming, T. C. Muller, A. Bonetti, C. Thibeault, W. J. Gross, P. Flatresse, and A. Burg, "Polarbear: A 28-nm fd-soi asic for decoding of polar codes," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 7, no. 4, pp. 616–629, 2017.