Implementation of LDPC Decoder - Using AHIR Tool Chain

Anurag Gupta
Microelectronics (2015-17)
Instructor: Prof Madhav P. Desai

IIT-Bombay
Department of Electrical Engineering

June 30, 2017

Outline

Motivation

Why LDPC ?

- Channel capacity approaching codes[1]
  - S.Y.Chung shannon limit approaching code: For a white Gaussian noise channel threshold within 0.0045 dB of the Shannon limit with block length of 10^7.
- Decoding time varies linearly proportional to block length
  - Parity check matrix is sparse
- Decoder can be parallelized
  - Decoding algorithms are iterative

[1] S.-Y. Chung, G. D. Forney, Jr., T. J. Richardson, and R. L. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," IEEE Commun. Letters, vol. 5, no. 2, pp. 58–60, February 2001.

LDPC Decoding Algorithms

- Hard decoding algorithms
  - Bit flipping algorithm
- Soft decoding algorithms
  - Sum product decoding
  - Min sum decoding

Min Sum Decoding Algorithm

[2] R. M. Tanner, "A recursive approach to low complexity codes," IEEE Trans. Inform. Theory, vol. IT-27, no. 5,

Min Sum Decoding Algorithm

Tanner Graph:

- LDPC code's parity check equations can be represented by bipartite graph, called the Tanner graph[2].

$$
\begin{array}{c|c}
 & \begin{array}{cccccccc} c1 & c2 & c3 & c4 & c5 & c6 & c7 & c8 \end{array} \\
\hline
\begin{array}{c} r1 \\ r2 \\ r3 \\ r4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}
\end{array}
$$

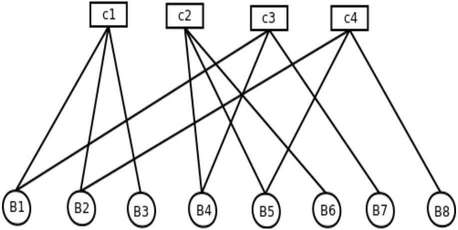


Figure: Tanner Graph

[2] R. M. Tanner, "A recursive approach to low complexity codes," IEEE Trans. Inform. Theory, vol. IT-27, no. 5,

A priori initialization

- A priories are calculated by soft information of the code bits.
- $aPriori[I] = -4 * C[I] * R * \frac{Eb}{No}$
- where C[I] = i^{th} code block
- R = code rate
- $\frac{Eb}{No}$ = signal to noise power ratio

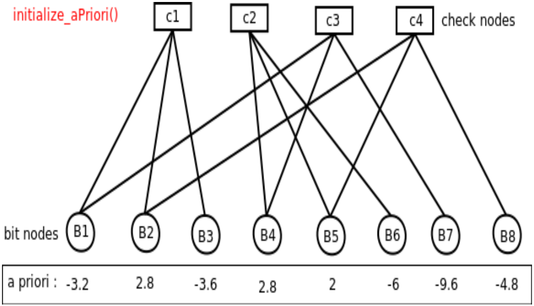


Figure: A priori initialization

Message initialization

- Messages are the information propagating from bit nodes to check nodes.
- These are initialized to a priori of their respective bit node.
- $message[I][J] = aPriori[I]$

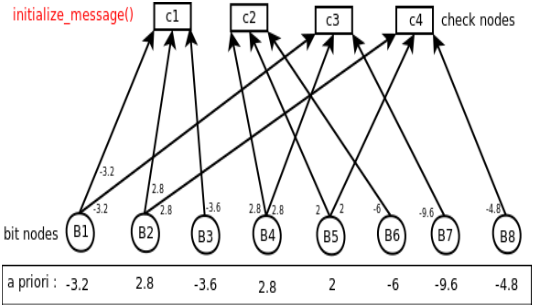
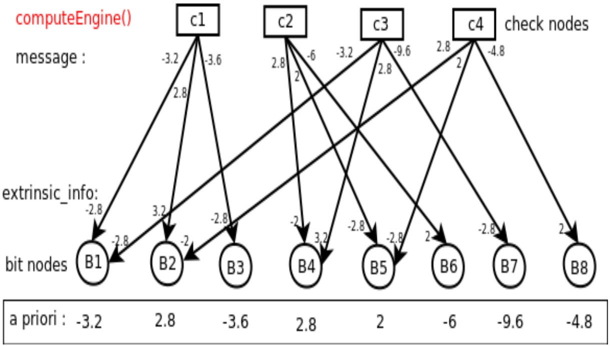


Figure: Message initialization

Extrinsic information calculation

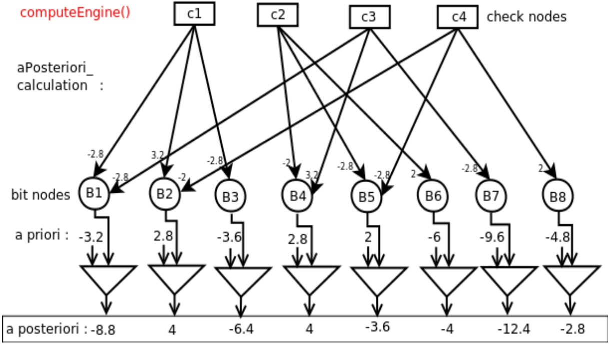
- Extrinsic information of a bit node is calculated as min sum of all the messages connected to that particular check node.



- $|E_{(j,i)}| = Min_{i' \in B_j \ i' \neq i} |M_{j,i'}|$
- $sign(E_{(j,i)}) = \prod_{i' \in B_j \ i' \neq i} sign(M_{j,i'})$

A posteriori calculation

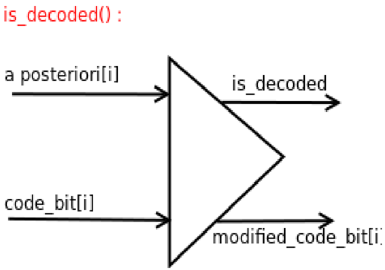
- A posteriori probabilities are the output bit probabilities.
- These are used to modify the code block after every iteration.



- $aPosteriori[I] = \sum_{j \in A_i} E_{j,i} + aPriori[I]$

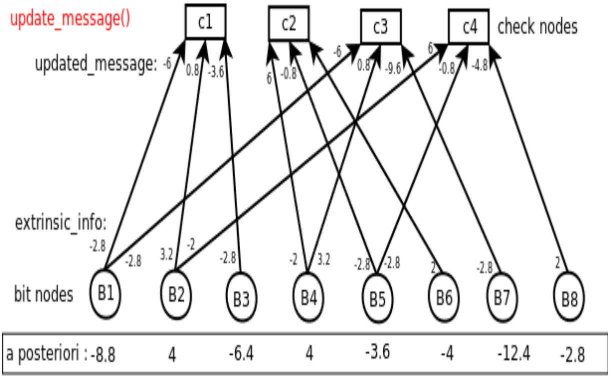
isDecoded block

- This block flips a bit if it is different form hard decision of the a posteriori probability of the bit. Thus, modifies the code block.
- If, no bit got flipped then decoding stops.
- *is_decoded* = 1 ;
  if ∀ i code_bit[I] = hard_decision(aPosteriori[I])



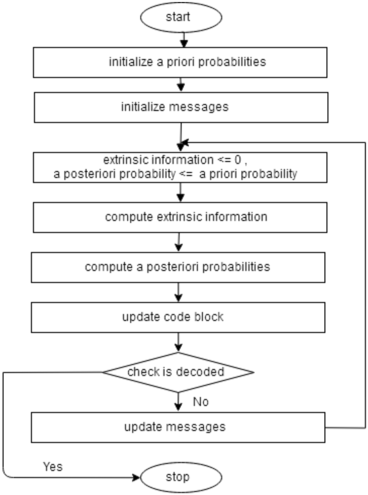
Updating messages

- Messages are updated and transmitted back to start the next iteration of decoding.



- $message_{(j,i)} = aPosteriori[i] - E_{(j,i)}$

LDPC Decoding : Min Sum Decode



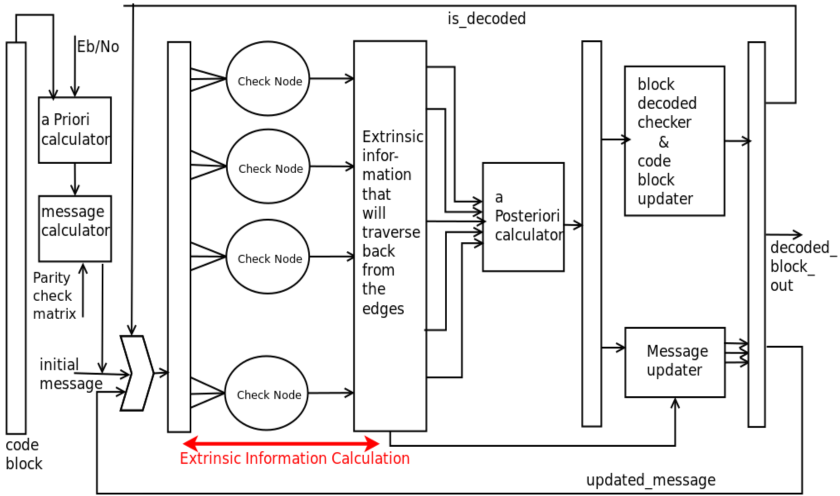
Decoder Implementation

Three different implementation[3] strategies are possible.

1. Serial decoder
   - simple
   - cheap
   - slow
2. Fully parallel decoder
   - complex
   - costly
   - Super fast
3. Partial parallel decoder.
   "Can we effectively partition a bipartite graph corresponding to a LDPC parity check matrix ?"

[3] Muhammad Awais and Carlo Condo, "Flexible LDPC Decoder Architectures," VLSI Design, vol. 2012, Article ID 730835, 16 pages, 2012.

Implementation of Serial Decoder


is_decoded
Eb/No
a Priori calculator
message calculator
Parity check matrix
initial message
code block
Check Node
Check Node
Check Node
Check Node
Extrinsic infor-mation that will traverse back from the edges
a Posteriori calculator
block decoded checker & code block updater
Message updater
decoded_ block_ out
Extrinsic Information Calculation
updated_message

C Level Implementation & Testing

Serial Min Sum Decoder - Quasi-Cyclic Matrix

- Min sum algorithm is implemented for Gaussian channel.
- Quasi-cyclic matrix of block size(n=) 4K, 8K and 12K are formed using Sridhara-Fuja-Tanner algorithm.
- Five different code rates(R=) 0.75, 0.80, 0.85, 0.90 and 0.95 are taken.
- Raw input bit error rate(BER(IN)) is between 10^{-2} to 10^{-3}, converted in form of Eb/No(db) to express input SNR in db.
- BER(OUT) : Output block error rate.
- We have tabulated when the first code block get wrongly decoded till 1 million transmitted blocks.

First error till 1 million blocks

n≃	BER(In)≃	R=0.75	R=0.80
4K	$1.0\text{x}10^{-2}$	-	-
	$0.5\text{x}10^{-3}$	-	-
	$1.0\text{x}10^{-3}$	-	-
8K	$1.0\text{x}10^{-2}$	-	-
	$0.5\text{x}10^{-3}$	-	-
	$1.0\text{x}10^{-3}$	-	-
12K	$1.0\text{x}10^{-2}$	-	-
	$0.5\text{x}10^{-3}$	-	-
	$1.0\text{x}10^{-3}$	-	-

- - : No error found till 1 million blocks.

First error till 1 million blocks

$n \simeq$	BER(In)$\simeq$	R=0.85	R=0.9	R=0.95
4K	1.0×10^{-2}	2.677×10^{3}	1.7819×10^{4}	1
	0.5×10^{-3}	2.4944×10^{4}	1.65511×10^{5}	1.79×10^{2}
	1.0×10^{-3}	5.47550×10^{5}	4.89654×10^{5}	3.328×10^{3}
8K	1.0×10^{-2}	2.3817×10^{4}	1.16847×10^{5}	1
	0.5×10^{-3}	6.9491×10^{4}	1.72263×10^{5}	1.001×10^{3}
	1.0×10^{-3}	9.16505×10^{5}	6.28939×10^{5}	9.338×10^{3}
12K	1.0×10^{-2}	9.705×10^{3}	5.37754×10^{5}	1
	0.5×10^{-3}	5.6400×10^{4}	-	1.318×10^{3}
	1.0×10^{-3}	-	-	1.6920×10^{4}

- - : No error found till 1 million blocks.

Serial Min Sum Decoder - Random Matrix

- Min sum algorithm is implemented for Gaussian channel.
- Random matrix of block size(n=) 4K, 8K and 12K are formed using Mackey's algorithm.
- Five different code rates(R=) 0.75, 0.80, 0.85, 0.90 and 0.95 are taken.
- Raw input bit error rate(BER(IN)) is between 10^{-2} to 10^{-3}, converted in form of Eb/No(db) to express input SNR in db.
- BER(OUT) : Output bit error rate.
- We have tabulated when the first block get wrongly decoded till 1 million transmitted blocks.

First error in 1 million blocks

n≃	BER(ln)≃	R=0.75	R=0.8
4K	1.0×10^{-2}	1.2799×10^{4}	2.0754×10^{4}
	0.5×10^{-3}	5.53727×10^{5}	1.72781×10^{5}
	1.0×10^{-3}	-	6.24436×10^{5}
8K	1.0×10^{-2}	1.92476×10^{5}	8.3898×10^{4}
	0.5×10^{-3}	3.21027×10^{5}	4.6092×10^{4}
	1.0×10^{-3}	-	-
12K	1.0×10^{-2}	2.20022×10^{5}	1.57371×10^{5}
	0.5×10^{-3}	2.17452×10^{5}	9.0158×10^{4}
	1.0×10^{-3}	-	-

- - : No error found till 1 million blocks.

First error in 1 million blocks

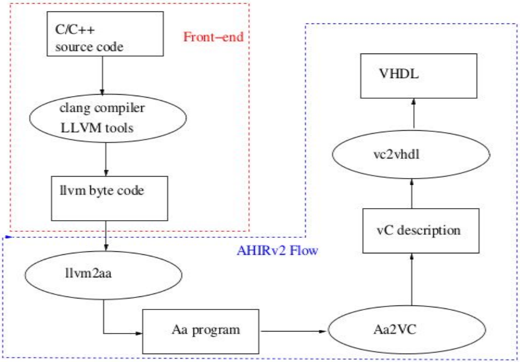
n≃	BER(In)≃	R=0.85	R=0.9	R=0.95
4K	1.0×10^{-2}	3.39×10^{2}	1.259×10^{3}	NA
	0.5×10^{-3}	6.6700×10^{4}	1.65511×10^{5}	NA
	1.0×10^{-3}	3.45503×10^{5}	1.19008×10^{5}	NA
8K	1.0×10^{-2}	5.193×10^{3}	5.947×10^{3}	NA
	0.5×10^{-3}	3.7952×10^{4}	1.1389×10^{4}	NA
	1.0×10^{-3}	-	-	NA
12K	1.0×10^{-2}	1.2894×10^{4}	1.2626×10^{4}	1.1
	0.5×10^{-3}	1.56487×10^{5}	5.4866×10^{4}	1.034×10^{4}
	1.0×10^{-3}	-	-	1.4759×10^{4}

- NA : Not Applicable. (Matrix was not formed)
- - : No error found till 1 million blocks.

Aa to VHDL -AHIR Tool Chain[4]

[4] https://github.com/madhavPdesai/ahir/release/docs/pdf/Overview.pdf.

Aa to VHDL -AHIR Tool Chain[4]



[4] https://github.com/madhavPdesai/ahir/release/docs/pdf/Overview.pdf.

Results


serial min sum decoder
serial min sum decoder - single FP operator
time to decode (ns)
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
64
128
256
512
order of matrix

Results


serial min sum decoder
serial min sum decoder - single FP operator
time to decode (ns)
2200000
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
0
100
200
300
400
500
600
order of matrix

Results : Hardware generated after synthesising the design

	serial min sum decoder	serial min sum decoder (single FP unit)
FF	18,076	19,034
LUT	19,502	20,621
Memory LUT	6	3
I/O	128	128
BRAM	56	56
BUFG	1	1

Partitioning : Objective

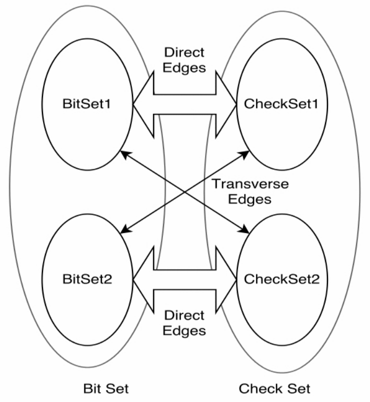


Figure: Partitioning the bipartite graph

Suggested Approach

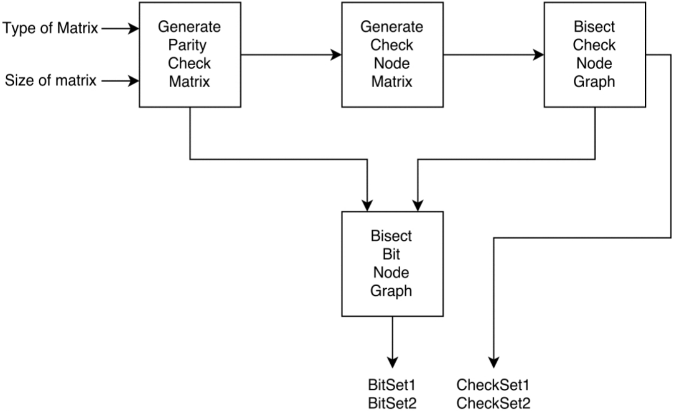


Figure: Block diagram

Partitioning Bit Node Set

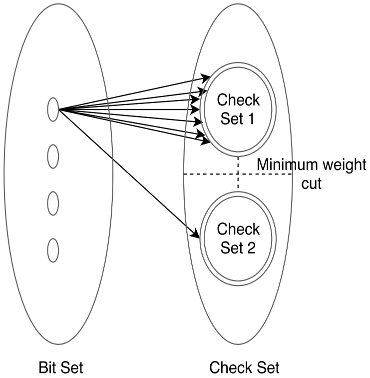


Figure: After partitioning check node set

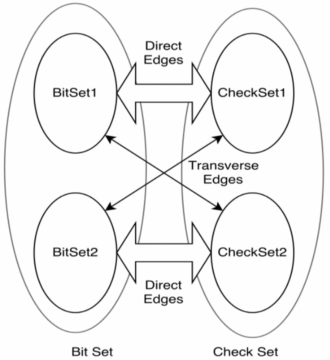
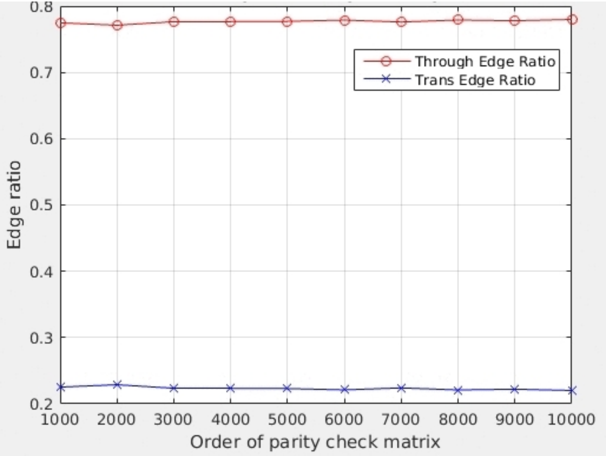
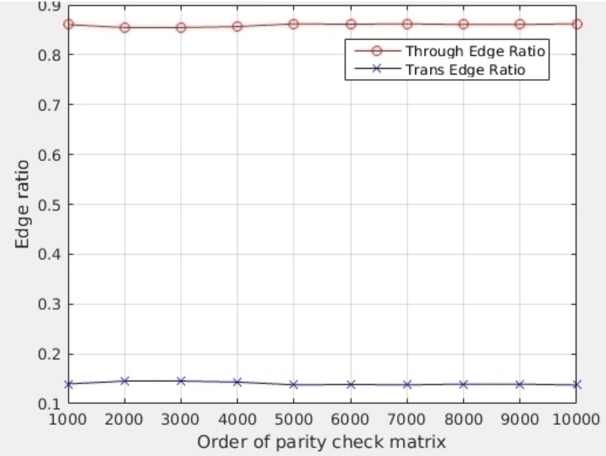


Figure: After partitioning bit node set

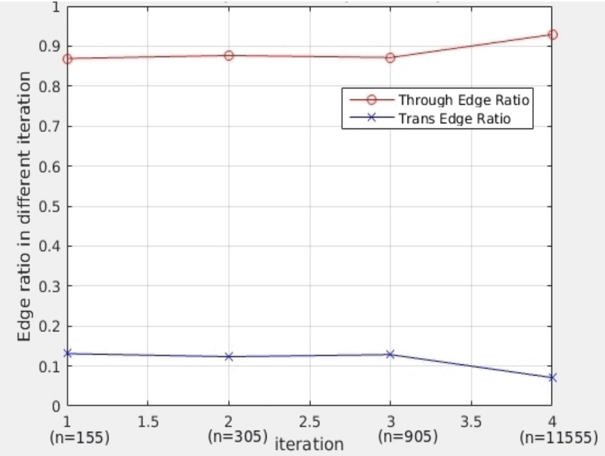
Gallager Parity Check Matrix



Mackay Parity Check Matrix



Quasi Cyclic Parity Check Matrix



Modifying min sum algorithm using partitioning

Modifying min sum algorithm using partitioning

- After partitioning the matrix we get four matrices as follows :

$$H = \left[\begin{array}{c|c} H11 & H12 \\ \hline H21 & H22 \end{array}\right]$$

- example :

$$\left[\begin{array}{c|cccccccc} & c1 & c2 & c3 & c4 & c5 & c6 & c7 & c8 \\ \hline r1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ r2 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ r3 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ r4 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \Rightarrow \left[\begin{array}{c|cccc|cccc} & c4 & c6 & c7 & c1 & c2 & c3 & c8 & c5 \\ \hline r2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ r3 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \hline r1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ r4 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right]$$

- H12 and H21 are highly sparse.

A priori initialization

- A priories are calculated by soft information of the code bits.
- $aPriori[I] = -4 * C[I] * R * \frac{Eb}{No}$
- where C[I] = i^{th} code block
- R = code rate
- $\frac{Eb}{No}$ = signal to noise power ratio

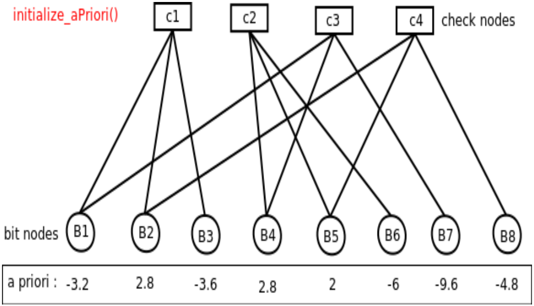
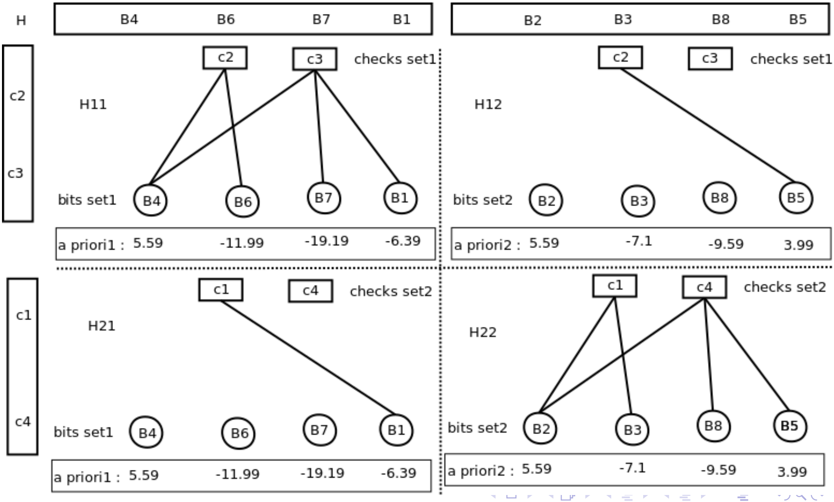


Figure: A priori initialization

A priori initialization



Message initialization

- Messages are the information propagating from bit nodes to check nodes.
- These are initialized to a priori of their respective bit node.
- $message[I][J] = aPriori[I]$

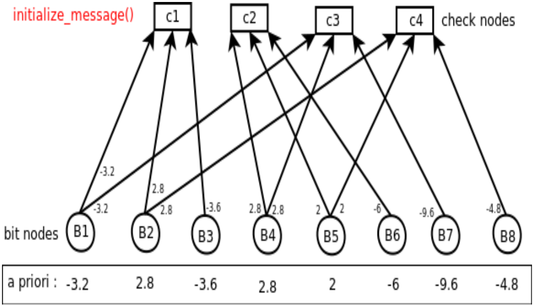
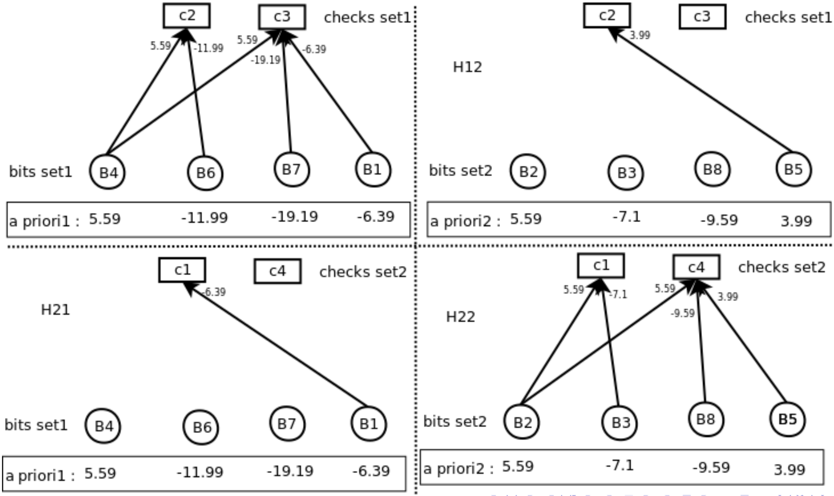


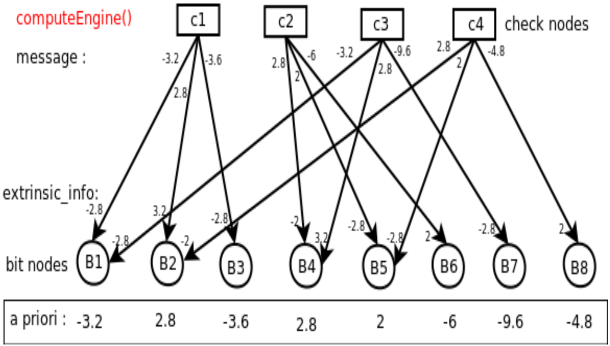
Figure: Message initialization

Message initialization



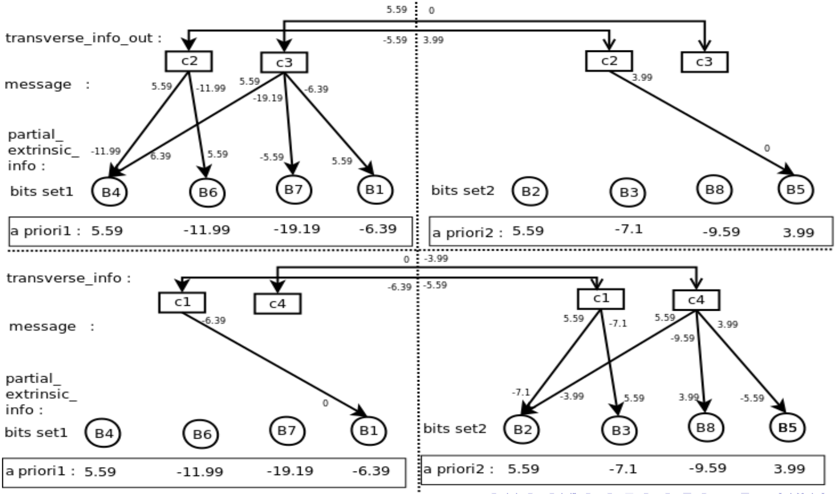
Extrinsic information calculation

- Extrinsic information of a bit node is calculated as min sum of all the messages connected to that particular check node.

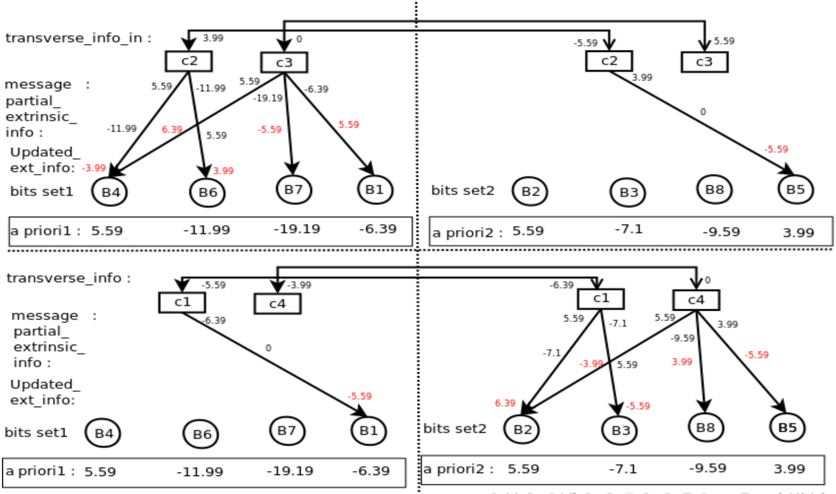


- $|E_{(j,i)}| = Min_{i' \in B_j \;\; i' \neq i} |M_{j,i'}|$
- $sign(E_{(j,i)}) = \prod_{i' \in B_j \;\; i' \neq i} sign(M_{j,i'})$

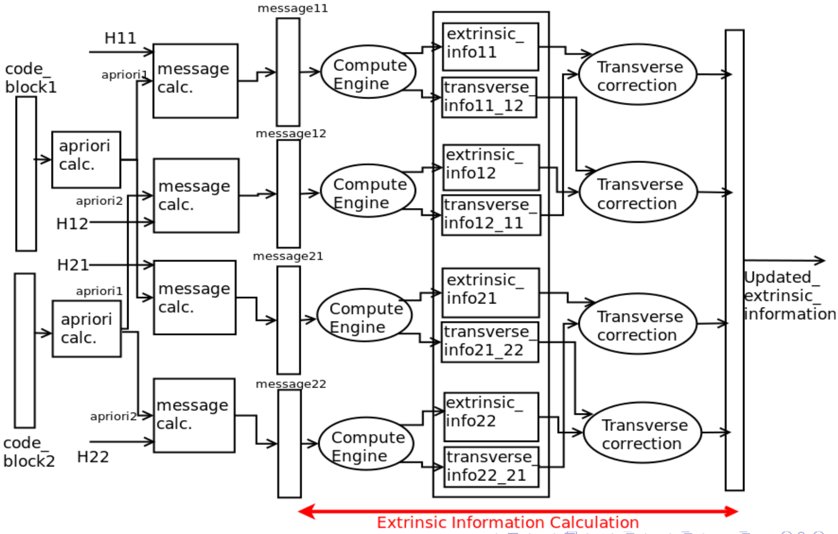
Partial Extrinsic information calculation



Update extrinsic information



C Level Implementation


message11
H11
code_
block1
apriori1
message
calc.
Compute
Engine
extrinsic_
info11
transverse_
info11_12
Transverse
correction
message12
apriori
calc.
message
calc.
Compute
Engine
extrinsic_
info12
transverse_
info12_11
Transverse
correction
apriori2
H12
H21
message21
apriori1
message
calc.
apriori
calc.
Compute
Engine
extrinsic_
info21
transverse_
info21_22
Transverse
correction
Updated_
extrinsic_
information
message22
apriori2
message
calc.
code_
block2
H22
Compute
Engine
extrinsic_
info22
transverse_
info22_21
Transverse
correction
Extrinsic Information Calculation

modifiedMinSumDecode() :

$initialize\_aPriori(aPriori1)$
$initialize\_aPriori(aPriori2)$
$initializeMessage(message11)$
$initializeMessage(message12)$
$initializeMessage(message21)$
$initializeMessage(message22)$
while $nitr \geq Max\_nitr$ **do**
  $initialize\_aPosteriori(aPosteriori1) \Leftarrow aPriori1$
  $initialize\_aPosteriori(aPosteriori2) \Leftarrow aPriori2$
  $initializeExtrinsicInfo(ext\_info11) \Leftarrow 0$
  $initializeExtrinsicInfo(ext\_info12) \Leftarrow 0$
  $initializeExtrinsicInfo(ext\_info21) \Leftarrow 0$
  $initializeExtrinsicInfo(ext\_info22) \Leftarrow 0$

  ...

modifiedMinSumDecode() :

while ... **do**

...

$computeEngine(H11, message11, ext\_info11, trans\_info11\_12)$
$computeEngine(H22, message22, ext\_info22, trans\_info22\_12)$
$computeEngine(H12, message12, ext\_info12, trans\_info12\_11)$
$computeEngine(H21, message21, ext\_info21, trans\_info21\_22)$
$transverseCorrection(H11, transverse\_info12\_11, ext\_info11)$
$transverseCorrection(H22, transverse\_info21\_22, ext\_info22)$
$transverseCorrection(H21, transverse\_info22\_21, ext\_info21)$
$transverseCorrection(H12, transverse\_info11\_12, ext\_info12)$
$update\_aPosteriori(H11, ext\_info11, aPosteriori1)$
$update\_aPosteriori(H22, ext\_info22, aPosteriori2)$
$update\_aPosteriori(H12, ext\_info12, aPosteriori1)$
$update\_aPosteriori(H21, ext\_info21, aPosteriori2)$

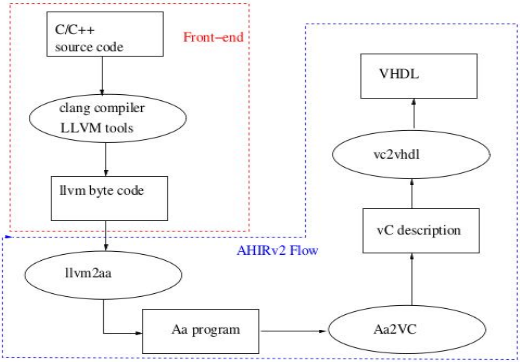
modifiedMinSumDecode() :

while ... **do**
  ...
  $is\_decoded1 = checkIsdecoded(code\_block1, aPosteriori1)$
  $is\_decoded2 = checkIsdecoded(code\_block2, aPosteriori2)$
  if $(is\_decoded1 \&\& is\_decoded2) == 1$ **then**
    break
  else
    $updateMessage(ext\_info11, aPosteriori1, message11)$
    $updateMessage(ext\_info22, aPosteriori2, message22)$
    $updateMessage(ext\_info12, aPosteriori1, message12)$
    $updateMessage(ext\_info21, aPosteriori2, message21)$
  end if
  $nitr++$
end while

Aa to VHDL -AHIR Tool Chain[5]

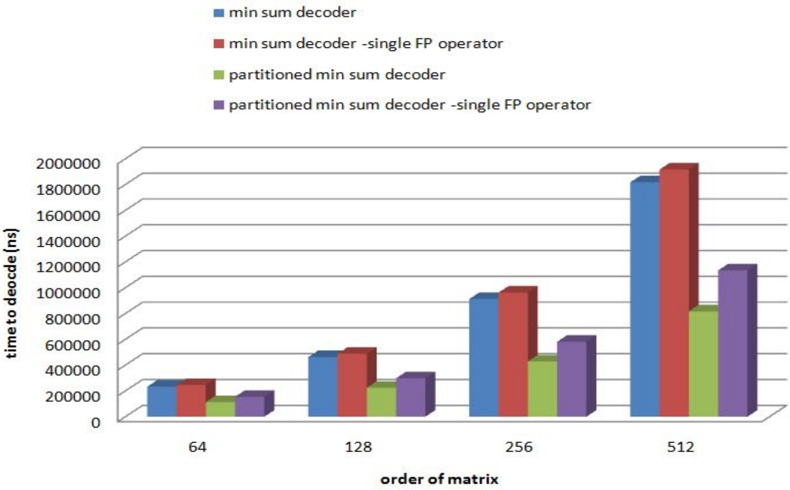
[5] https://github.com/madhavPdesai/ahir/release/docs/pdf/Overview.pdf.

Aa to VHDL -AHIR Tool Chain[5]

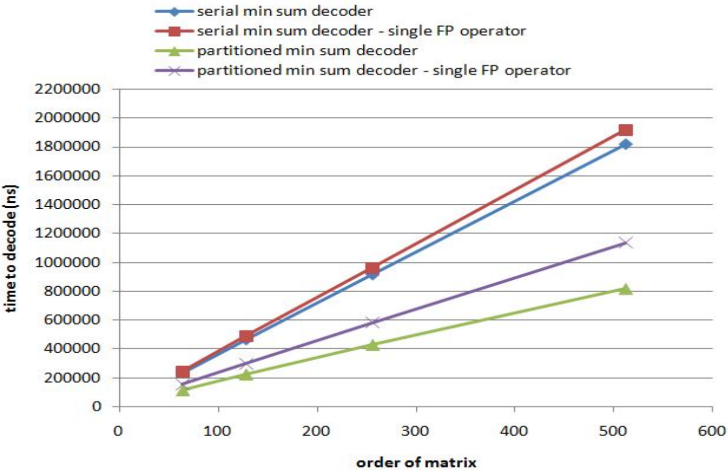


[5] https://github.com/madhavPdesai/ahir/release/docs/pdf/Overview.pdf.

Results


min sum decoder
min sum decoder -single FP operator
partitioned min sum decoder
partitioned min sum decoder -single FP operator
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
time to deocde (ns)
64
128
256
512
order of matrix

Results


serial min sum decoder
serial min sum decoder - single FP operator
partitioned min sum decoder
partitioned min sum decoder - single FP operator
2200000
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
time to decode (ns)
0
100
200
300
400
500
600
order of matrix

Results

	serial min sum decoder	serial min sum decoder (single FP unit)	partitioned min sum decoder	partitioned min sum decoder(single FP unit)
FF	18,076	19,034	49,854	55,988
LUT	19,502	20,621	51,929	60,296
Memory LUT	6	3	23	2
I/O	128	128	128	128
BRAM	56	56	80	80
BUFG	1	1	1	1

Conclusion & Future Work

Type of Matrix	Gallager	Mackay Neal	Quasi-Cyclic
Performance Index	78%	86%	88%

- The results show that a LDPC decoder can be parallelized with good efficiency.
- The implemented 4-way partitioned decoder reduces the time required for decoding to half but uses $2.5\times$ the harder required for single decoder.
- In future we can figure out a way to fold two engines on the top of other two engines to reduce hardware, instead of using four computational engines.
- The extension of the work can have different quantization levels of the floating point values and check for the trade off between accuracy of operation and error correcting threshold.

Thank You

Questions?

Basic definitions

- **random codes vs systematic codes:** parity check matrix is generated randomly in random codes whereas parity check matrix has a specific method of filling $1's$ in matrix in a systematic code.
- **regular codes vs irregular codes:** In regular codes parity check matrix has constant number of $1's$ in row and columns. if w_c is number of $1's$ in a column and $w_r$is number of $1's$ in a row then in a mxn parity check matrix.
  $m.w_r = n.w_c$
  irregular codes we designate the fraction of columns of weight i by v_i and the fraction of rows of weight i by h_i. Collectively the set v and h is called the degree distribution of the code.
  $m.\sum_i h_i.i = n\sum_i v_i.i$

Gallager parity check matrix

- regular codes,random codes
- (n,w_c,w_r) codes. w_c= no of 1's in a column w_r= number of 1's in a row n=block length
  Method of construction:
- divide rows in w_c sets with m/w_c rows in each set.
- All rows of first set of rows contain w_r consecutive once ordered from left to right.
- Every other set of row is random **column permutation** of first set of rows.

Gallager parity check matrix:example

(n,w_c,w_r)=(12,3,4) take m=9

- thus we have 3 set of rows having 9/3=3 rows in a set.

$$
\begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
- & - & - & - & - & - & - & - & - & - & - & - \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
- & - & - & - & - & - & - & - & - & - & - & - \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1
\end{bmatrix}
$$

MacKay Neal parity-check matrix

- regular (n,w_c,w_r) codes,random codes
  Method of construction:
- start from the first column.Place w_c $1^{'}$s in the column randomly.
- Keep a track a $1^{'}$s in a row.
- Keep Repeating the process for other columns. Break only if at any point number of $1^{'}$s in the row becomes greater than w_r.
- If break occurs then go back to some columns and repeat again till all columns get filled.

MacKay Neal parity-check matrix

n=12 m=9

- $$\begin{bmatrix} 1&0&0&0&0&1&0&1&0&1&0&0\\ 1&0&0&1&1&0&0&0&0&0&1&0\\ 0&1&0&0&1&0&1&0&1&0&0&0\\ 0&0&1&0&0&1&0&0&0&0&1&1\\ 0&0&1&0&0&0&1&1&0&0&0&1\\ 0&1&0&0&1&0&0&0&1&0&1&0\\ 1&0&0&1&0&0&1&0&0&1&0&0\\ 0&1&0&0&0&1&0&1&0&1&0&0\\ 0&0&1&1&0&0&0&0&1&0&0&1 \end{bmatrix}$$

Repeat Accumulate Code

- systematic codes,irregular codes
- each parity-bit can be computed one at a time using only the message bits and the one previously calculated parity-bit.
  Method of construction:
- The first (n-m) columns of H correspond to the message bits.
- then rest columns have 1's of weight two, that are placed in a step pattern for the last m columns of H.

Repeat Accumulate Code:example

n=12 m=9

- $$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$
- $c_4 = c_1 \ ; \ c_5 = c_1 \oplus c_4 \ ; \ c_6 = c_5 \oplus c_2 \ ; \ \ldots$

Quasi-Cyclic (QC) parity Check Matrix:

QC matrix can be construction by Sridhara Fuja Tanner (SFT) Method is discussed. Method of Construction of (j,k)-regular QC-LDPC code:

- Construct two sequences $\{s_1, s_2, ..., s_{j-1}\}$ and $\{t_1, t_2, ..., t_{k-1}\}$, whose elements are randomly selected from GF(p), where p is prime and p>2 , $s_i \neq s_x$ & $t_i \neq t_x$ if i $\neq$x.
- Now, form a preliminary matrix E with the elements of GF(p) as follows:

$$E = \begin{bmatrix} e_{0,0} & e_{0,1} & \cdots & e_{0,k-1} \\ e_{1,0} & e_{0,1} & \cdots & e_{0,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ e_{j-1,0} & e_{j-1,1} & \cdots & e_{j-1,k-1} \end{bmatrix} \quad (1)$$

- where (i,j)th element of E is calculated by following quadratic congruential equation for a fix parameter $\kappa \epsilon \{1, 2, ..., p-1\}$ and $\nu_i, \nu_j \epsilon \{1, 2, ..., p-1\}$:

$$e_{i,j} = [\kappa(s_i + t_j)^2 + \nu_i + \nu_j] \tag{2}$$

- So the parity check matrix H is represented by jxk array of circulant permutation of identity matrix.

$$H = \begin{bmatrix} I(e_{0,0}) & I(e_{0,1}) & \cdots & I(e_{0,k-1}) \\ I(e_{1,0}) & I(e_{0,1}) & \cdots & I(e_{0,k-1}) \\ \vdots & \vdots & \ddots & \vdots \\ I(e_{j-1,0}) & I(e_{j-1,1}) & \cdots & I(e_{j-1,k-1}) \end{bmatrix} \tag{3}$$

Where I(x) is pxp identity matrix with row cyclically shifted right by x position.

Sum product decoding

- it is a **soft decision** algorithm.
- bit-flipping decoding accepts an initial hard decision on the received bits as input, whereas the sum-product algorithm accepts the probability of each received bit as input.
- The input bit probabilities are called the a **priori** probabilities
- The bit probabilities returned by the decoder are called the a **posteriori** probabilities
- sum-product decoding these probabilities are expressed as **log-likelihood ratios.**

LLR

- $L(x) = log\dfrac{p(x=0)}{p(x=1)} = log\dfrac{1-p(x=1)}{p(x=1)}$
- If p(x = 0) > p(x = 1) then L(x) is positive.
  and the greater the difference between p(x = 0) and p(x = 1), i.e. the more sure we are that p(x) = 0, the larger the positive value for L(x), and vic. versa.
- Thus,Log likelihood ratios are used to represent the **metrics** for a binary variable x by a single value rather than individual probability of being zero and one.
- Thus the sign of L(x) provides the **hard decision** on x and the magnitude $|L(x)|$ is the **reliability** of this decision

Probabilities in term of LLR

- probability that transmitted bit was one is :
$$P(x=1) = \frac{p(x=1)/p(x=0)}{1+p(x=1)/p(x=0)} = \frac{e^{-L(x)}}{1+e^{-L(x)}}$$
- probability that transmitted bit was zero is :
$$P(x=1) = \frac{p(x=0)/p(x=1)}{1+p(x=0)/p(x=1)} = \frac{e^{L(x)}}{1+e^{L(x)}}$$
- benefit of the logarithmic representation of probabilities is that when probabilities need to be multiplied log-likelihood ratios**(LLR)** need only be **added**, reducing implementation complexity

Sum product decoding:contd...

- aim: compute the maximum a posteriori probability **(MAP)** for each codeword bit on the event N that all parity-check constraints are satisfied
- The extra information about bit i received from the parity-check j is called **extrinsic information** for bit i denoted by E_j, i.
- $E_{(j,i)} = log \frac{\frac{1}{2} + \frac{1}{2} \prod_{i' \in B_j \ \ i' \neq i} tanh(M_{j,i'}/2)}{\frac{1}{2} - \frac{1}{2} \prod_{i' \in B_j \ \ i' \neq i} tanh(M_{j,i'}/2)}$
- Mj, i is message passed from ith bit node to jth check node.
- for first iteration Mj, i is priori of bit i .

Sum product decoding:contd...

After first iteration:

- Each bit has access to the input a priori LLR, ri, and the LLRs from every connected check node. The total LLR of the i-th bit is the sum of these LLRs:
  $LLR = \sum_{j \in A_i} E_{j,i} + r_i$
- Now, take hard decision on the LLR post priori and check whether code satisfies $Hc^T = 0$.
- if yes, then decoding stop. else find $M_{j,i}$ and repeat the process of calculating$\rightarrow$ $M_{j,i}$ $\rightarrow$extrinsic LLR $\rightarrow$ LLR $\rightarrow$ hard decision $\rightarrow$ satisfy Hc^T ...
- But, note that $M_{j,i}$ is not exactly the LLR, it exclude the message generated by the same check node.
  $M_{j,i} = \sum_{j' \in A_i j' \neq j} E_{j,i} + r_i$

Gallager[6] parity check matrix-

- Regular matrix
- Randomly generated
- Short cycles
- Simple construction

---

[6] R. G. Gallager, Low-Density Parity-Check Codes. Cambridge, MA: MIT Press, 1963.

[7] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," IEEE Trans. Inform. Theory, vol. 45, no. 2, pp. 399–431, March 1999.

[8] C. M. Huang, J. F. Huang and C. C. Yang, "Construction of quasi-cyclic LDPC codes from quadratic congruences," in IEEE Communications Letters, vol. 12, no. 4, pp. 313-315, April 2008.

Gallager[6] parity check matrix-

- Regular matrix
- Randomly generated
- Short cycles
- Simple construction

Mackay Neal[7] parity check matrix-

- Irregular matrix
- Randomly generated
- Avoid cycle of four
- Performance is better for large block length

[6] R. G. Gallager, Low-Density Parity-Check Codes. Cambridge, MA: MIT Press, 1963.

[7] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," IEEE Trans. Inform. Theory, vol. 45, no. 2, pp. 399–431, March 1999.

[8] C. M. Huang, J. F. Huang and C. C. Yang, "Construction of quasi-cyclic LDPC codes from quadratic congruences," in IEEE Communications Letters, vol. 12, no. 4, pp. 313-315, April 2008.

Gallager[6] parity check matrix-

- Regular matrix
- Randomly generated
- Short cycles
- Simple construction

Mackay Neal[7] parity check matrix-

- Irregular matrix
- Randomly generated
- Avoid cycle of four
- Performance is better for large block length

Quasi-cyclic[8] parity check matrix-

- Regular matrix
- Systematic matrix
- Performance is better for moderate block length

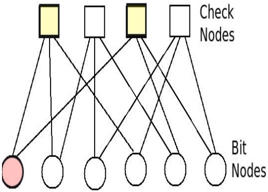
[6] R. G. Gallager, Low-Density Parity-Check Codes. Cambridge, MA: MIT Press, 1963.

[7] D. J. C. MacKay, “Good error-correcting codes based on very sparse matrices,” IEEE Trans. Inform. Theory, vol. 45, no. 2, pp. 399–431, March 1999.

[8] C. M. Huang, J. F. Huang and C. C. Yang, "Construction of quasi-cyclic LDPC codes from quadratic congruences," in IEEE Communications Letters, vol. 12, no. 4, pp. 313-315, April 2008.

Constructing weighted check node incidence matrix:

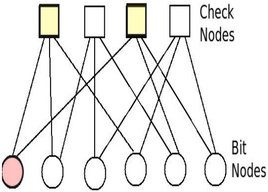
$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

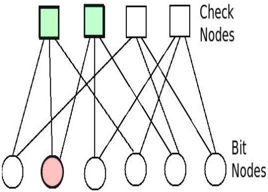


M(1,3)=M(3,1)=1

Constructing weighted check node incidence matrix:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

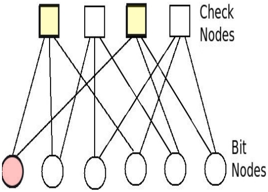


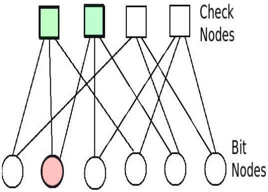


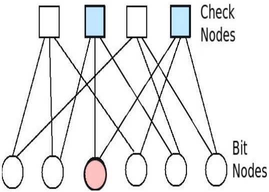
M(1,3)=M(3,1)=1 M(1,2)=M(2,1)=1

Constructing weighted check node incidence matrix:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

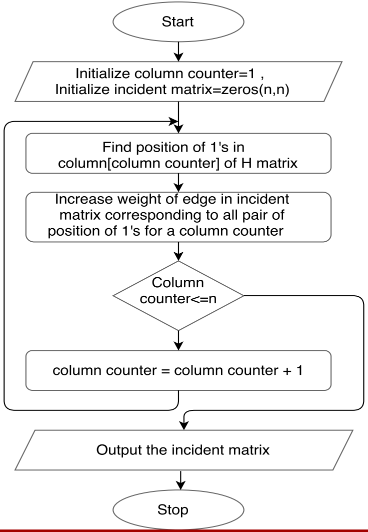




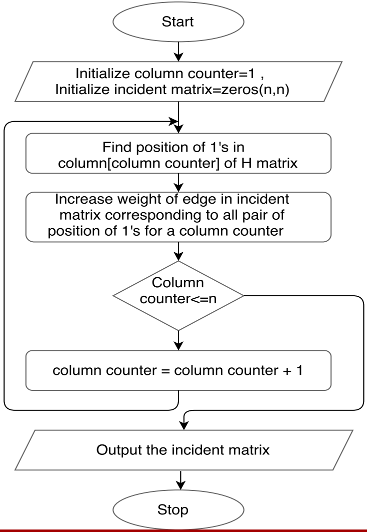


M(1,3)=M(3,1)=1 M(1,2)=M(2,1)=1 M(2,4)=M(4,2)=1

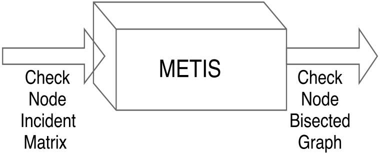
Partitioning check node matrix



Partitioning check node matrix



Partitioning Incident Matrix using METIS[a]



- Minimum weight cut
- Equal partitions

[a] glaros.dtc.umn.edu/gkhome/metis/metis/