ECC for NAND Flash

Osso Vahabzadeh

TexasLDPC Inc.

Overview

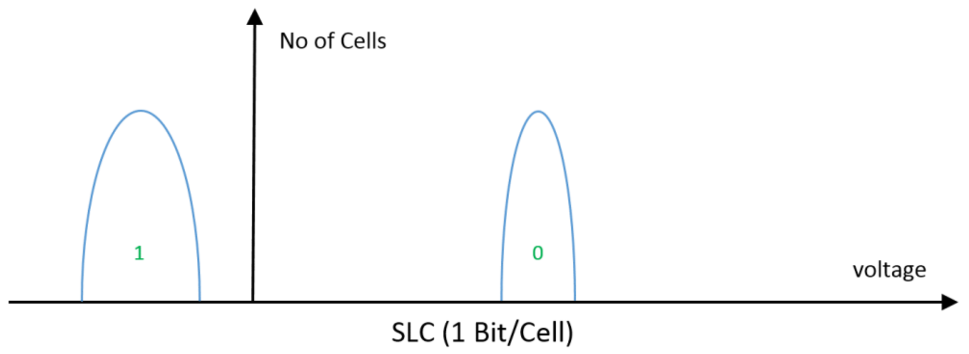
- Why Is Error Correction Needed in Flash Memories?
- Error Correction Codes Fundamentals
- Low-Density Parity-Check (LDPC) Codes
- LDPC Encoding and Decoding Methods
- Decoder Architectures for LDPC Codes

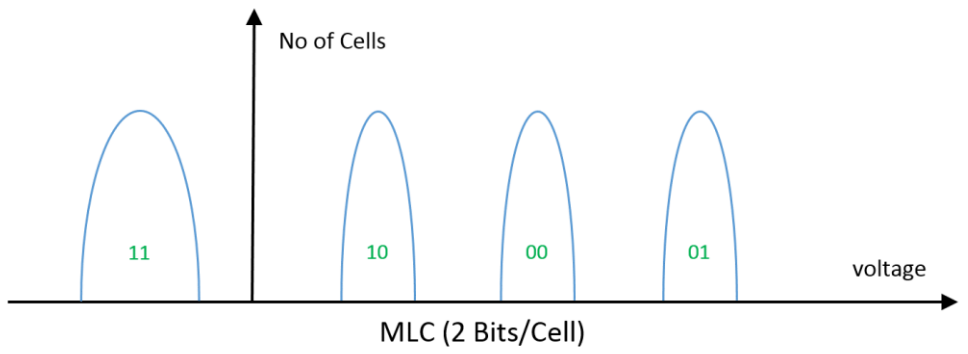
Overview

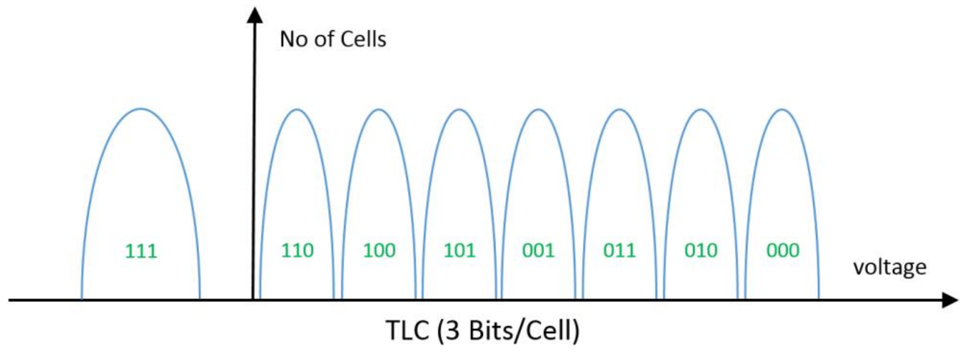
- Why Is Error Correction Needed in Flash Memories?
- Error Correction Codes Fundamentals
- Low-Density Parity-Check (LDPC) Codes
- LDPC Encoding and Decoding Methods
- Decoder Architectures for LDPC Codes

NAND Flash Basics

- Information is stored in a NAND flash cell by inducing a certain voltage to its floating gate.
- To read the stored bit(s), the cell voltage is compared with a set of threshold voltages and a hard-decision bit is sent out.





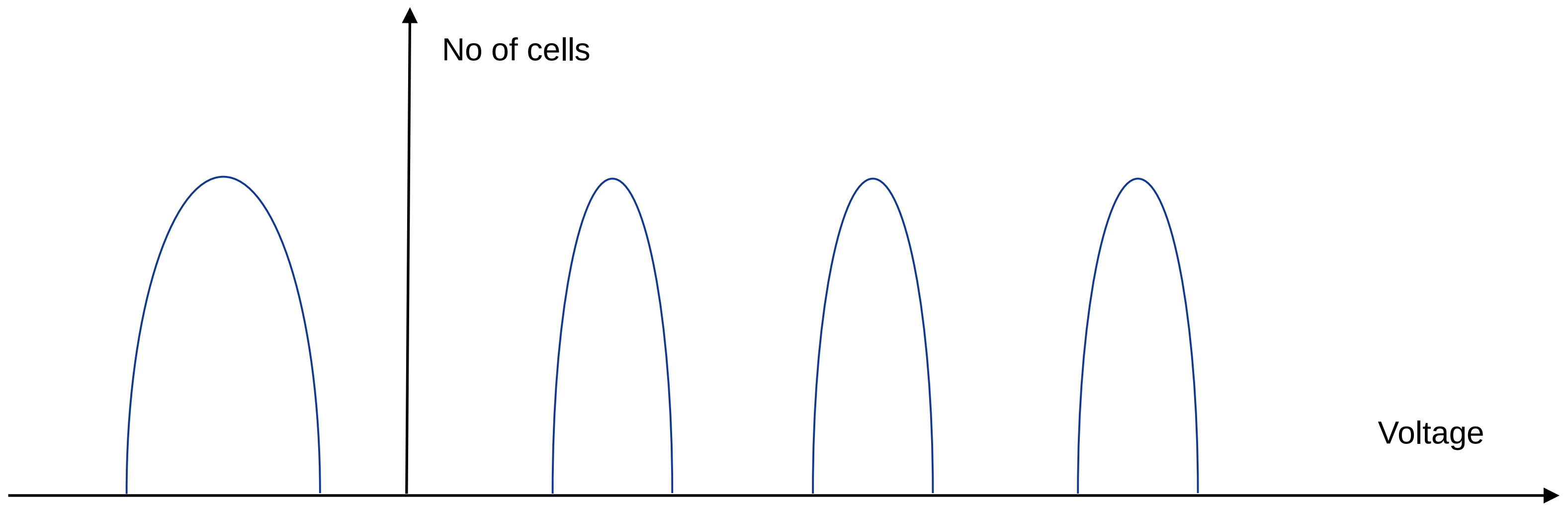


Threshold Voltage Distribution with Increasing Number of P/E Cycles

- Cell voltages change randomly over time and with memory wear out. So read voltage is a random variable.
- As the number of P/E cycles increases, the threshold voltage distributions widen and shift.

Threshold Voltage Distribution with Increasing Number of P/E Cycles

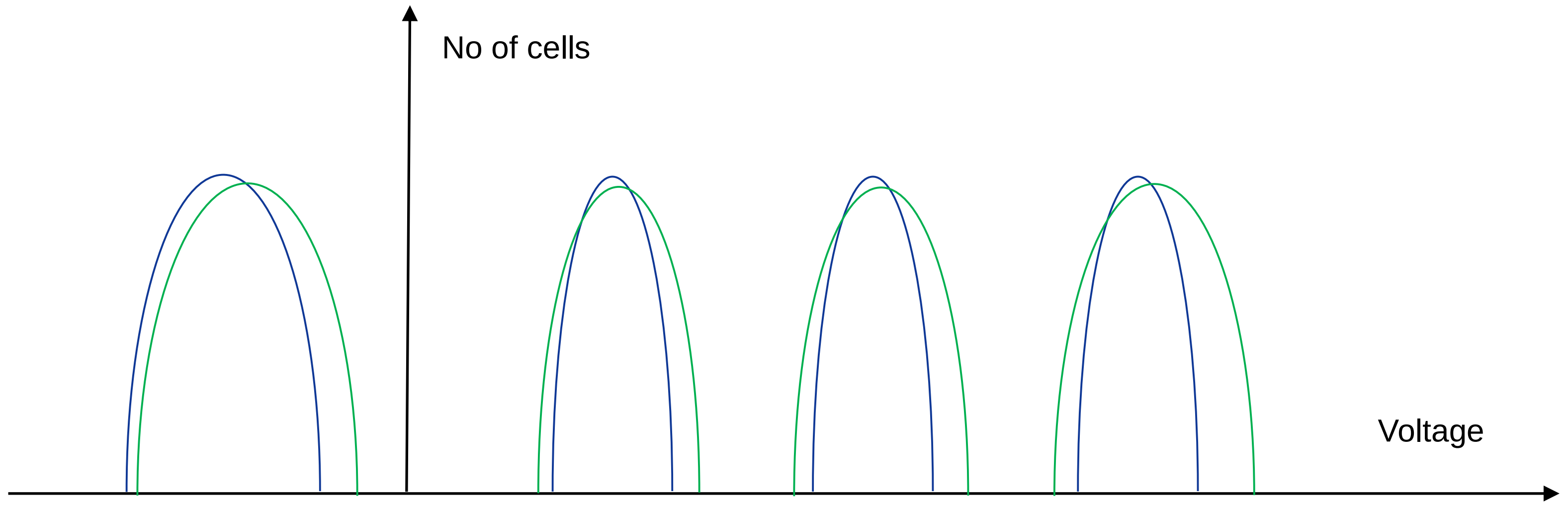
- Cell voltages change randomly over time and with memory wear out. So read voltage is a random variable.
- As the number of P/E cycles increases, the threshold voltage distributions widen and shift.



Threshold Voltage Distribution

Threshold Voltage Distribution with Increasing Number of P/E Cycles

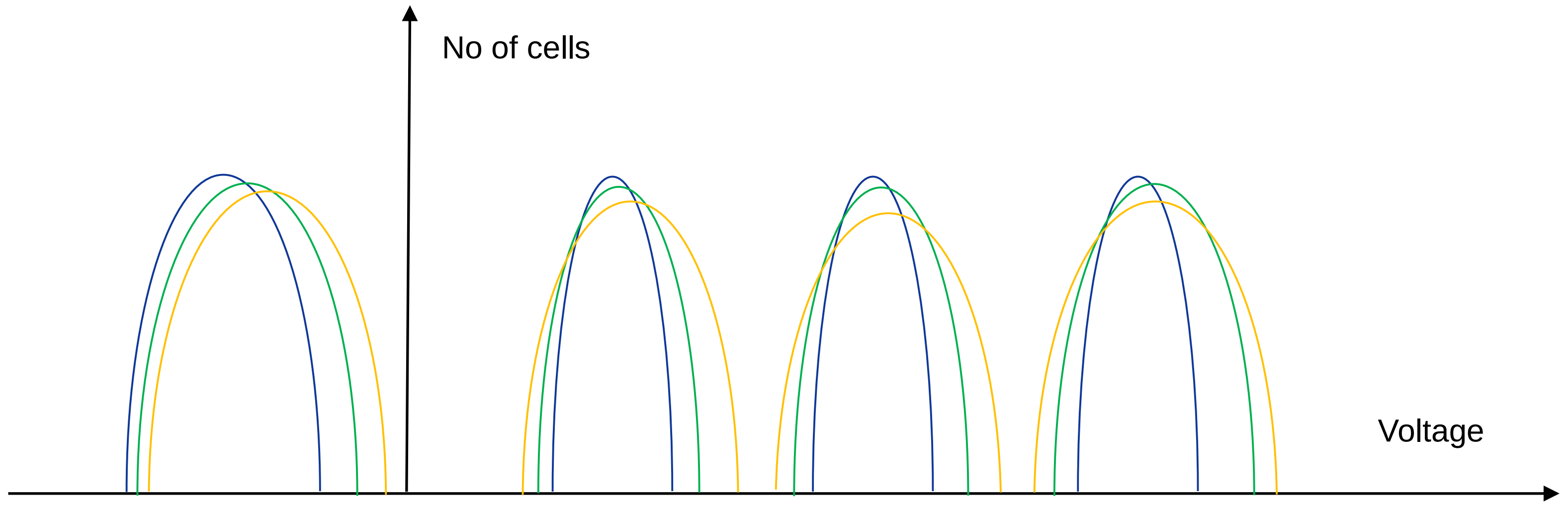
- Cell voltages change randomly over time and with memory wear out. So read voltage is a random variable.
- As the number of P/E cycles increases, the threshold voltage distributions widen and shift.



Threshold Voltage Distribution

Threshold Voltage Distribution with Increasing Number of P/E Cycles

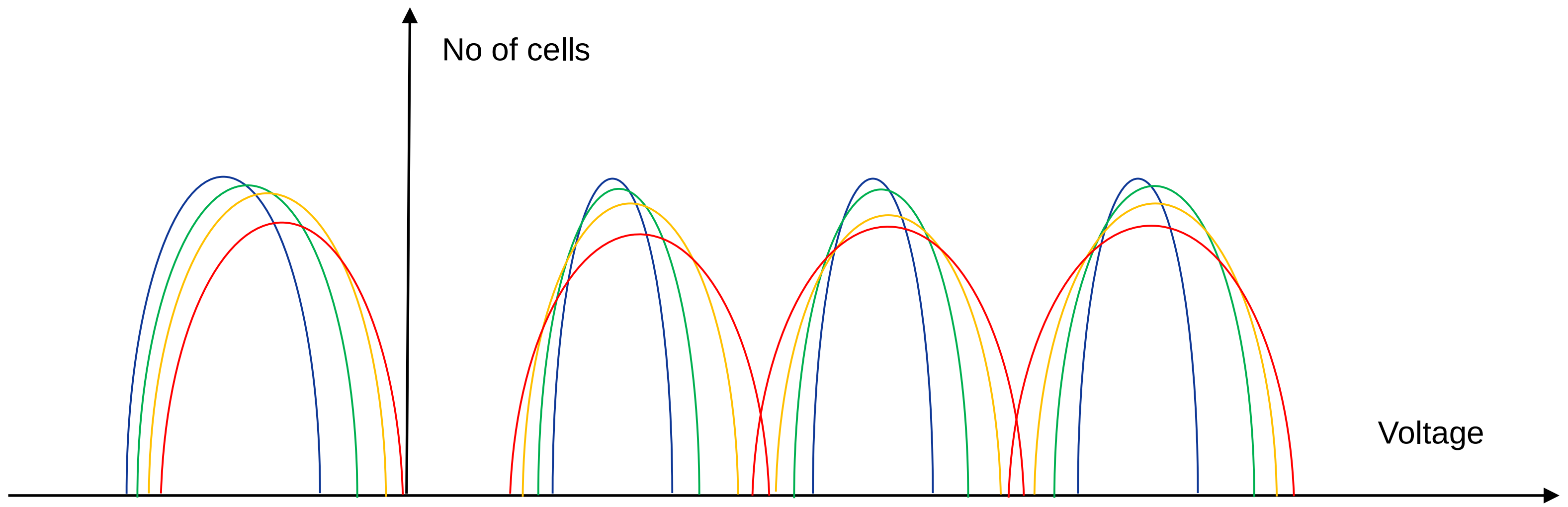
- Cell voltages change randomly over time and with memory wear out. So read voltage is a random variable.
- As the number of P/E cycles increases, the threshold voltage distributions widen and shift.



Threshold Voltage Distribution

Threshold Voltage Distribution with Increasing Number of P/E Cycles

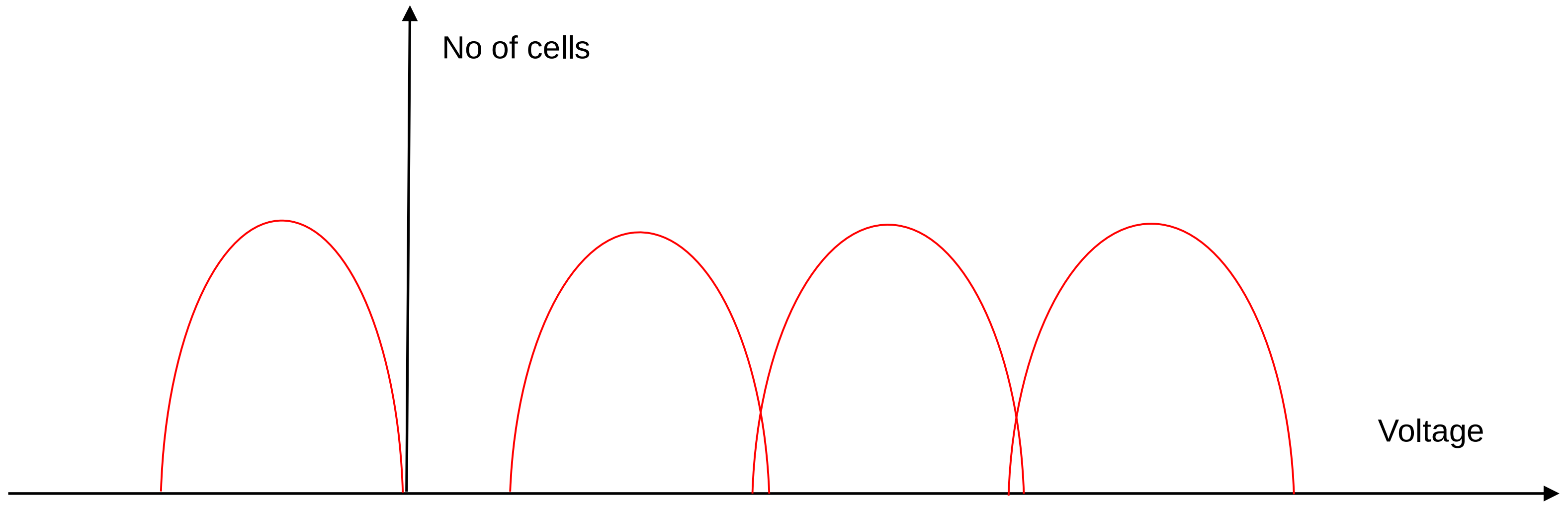
- Cell voltages change randomly over time and with memory wear out. So read voltage is a random variable.
- As the number of P/E cycles increases, the threshold voltage distributions widen and shift.



Threshold Voltage Distribution

Threshold Voltage Distribution with Increasing Number of P/E Cycles

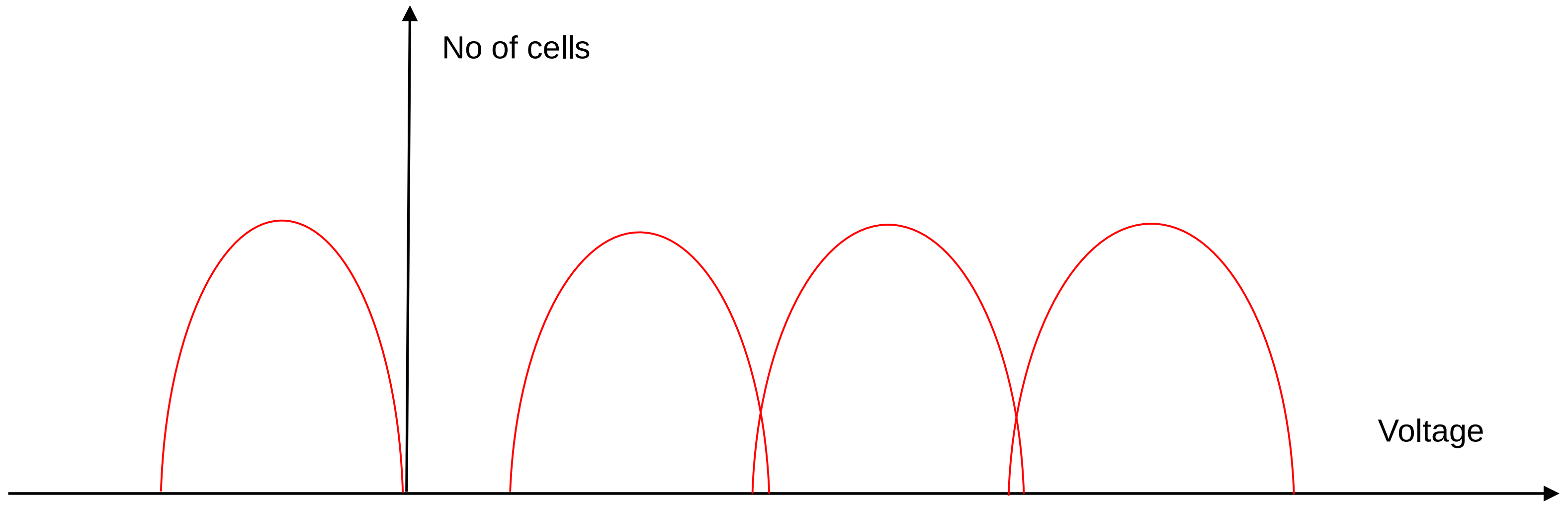
- Cell voltages change randomly over time and with memory wear out. So read voltage is a random variable.
- As the number of P/E cycles increases, the threshold voltage distributions widen and shift.



Threshold Voltage Distribution

Threshold Voltage Distribution with Increasing Number of P/E Cycles

- Cell voltages change randomly over time and with memory wear out. So read voltage is a random variable.
- As the number of P/E cycles increases, the threshold voltage distributions widen and shift.



Threshold Voltage Distribution

- Read errors occur when distributions overlap.

Overview

- Why Is Error Correction Needed in Flash Memories?
- Error Correction Codes Fundamentals
- Low-Density Parity-Check (LDPC) Codes
- LDPC Encoding and Decoding Methods
- Decoder Architectures for LDPC Codes

Error Correction Codes

Error Correction Codes

- Linear Block codes: $(u_1, \ldots, u_k) \rightarrow (c_1, \ldots, c_n), n > k$

Error Correction Codes

- Linear Block codes: $(u_1, \ldots, u_k) \rightarrow (c_1, \ldots, c_n), n > k$

  k: Data block size n: Codeword size $m = n - k$: Number of parity bits

Error Correction Codes

- Linear Block codes: $(u_1, \dots, u_k) \rightarrow (c_1, \dots, c_n), n > k$

  k: Data block size n: Codeword size $m = n - k$: Number of parity bits

- Example: $k = 4, n = 7, m = 3 \rightarrow$ A (7, 4) block code

Error Correction Codes

- Linear Block codes: $(u_1, \ldots, u_k) \rightarrow (c_1, \ldots, c_n), n > k$

  k: Data block size n: Codeword size $m = n - k$: Number of parity bits

- Example: $k = 4, n = 7, m = 3 \rightarrow$ A (7, 4) block code

$$c_1 = u_1$$
$$c_2 = u_2$$
$$c_3 = u_3$$
$$c_4 = u_4$$
$$c_5 = u_1 + u_2 + u_3$$
$$c_6 = u_2 + u_3 + u_4$$
$$c_7 = u_1 + u_2 + u_4$$

Error Correction Codes

- Linear Block codes: $(u_1, \dots, u_k) \rightarrow (c_1, \dots, c_n), n > k$

  k: Data block size n: Codeword size $m = n - k$: Number of parity bits

- Example: $k = 4, n = 7, m = 3 \rightarrow$ A (7, 4) block code

$$\left.\begin{aligned} c_1 &= u_1 \\ c_2 &= u_2 \\ c_3 &= u_3 \\ c_4 &= u_4 \end{aligned}\right\} \text{Information bits} \Longrightarrow \text{Systematic code}$$

$$\begin{aligned} c_5 &= u_1 + u_2 + u_3 \\ c_6 &= u_2 + u_3 + u_4 \\ c_7 &= u_1 + u_2 + u_4 \end{aligned}$$

Error Correction Codes

- Linear Block codes: $(u_1, \ldots, u_k) \rightarrow (c_1, \ldots, c_n), n > k$

  k: Data block size n: Codeword size $m = n - k$: Number of parity bits

- Example: $k = 4, n = 7, m = 3 \rightarrow$ A (7, 4) block code

$$\left.\begin{aligned} c_1 &= u_1 \\ c_2 &= u_2 \\ c_3 &= u_3 \\ c_4 &= u_4 \end{aligned}\right\} \text{Information bits} \Longrightarrow \text{Systematic code}$$

$$\left.\begin{aligned} c_5 &= u_1 + u_2 + u_3 \\ c_6 &= u_2 + u_3 + u_4 \\ c_7 &= u_1 + u_2 + u_4 \end{aligned}\right\} \text{Parity bits} \Longrightarrow \begin{aligned} c_1 + c_2 + c_3 + c_5 &= 0 \\ c_2 + c_3 + c_4 + c_6 &= 0 \\ c_1 + c_2 + c_4 + c_7 &= 0 \end{aligned}$$

Parity check equations

Error Correction Codes

$$\left.\begin{aligned} c_1 &= u_1 \\ c_2 &= u_2 \\ c_3 &= u_3 \\ c_4 &= u_4 \\ c_5 &= u_1 + u_2 + u_3 \\ c_6 &= u_2 + u_3 + u_4 \\ c_7 &= u_1 + u_2 + u_4 \end{aligned}\right\} \Longrightarrow \boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Generator Matrix

Error Correction Codes

$$\left.\begin{aligned} c_1 &= u_1 \\ c_2 &= u_2 \\ c_3 &= u_3 \\ c_4 &= u_4 \\ c_5 &= u_1 + u_2 + u_3 \\ c_6 &= u_2 + u_3 + u_4 \\ c_7 &= u_1 + u_2 + u_4 \end{aligned}\right\} \Longrightarrow \boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Generator Matrix

$$\boldsymbol{G} = \left[\boldsymbol{I}_{k\times k}, \boldsymbol{P}_{k\times(n-k)}\right]$$

Flash Memory Summit

Error Correction Codes

$$\left.\begin{aligned}c_1 &= u_1\\ c_2 &= u_2\\ c_3 &= u_3\\ c_4 &= u_4\\ c_5 &= u_1+u_2+u_3\\ c_6 &= u_2+u_3+u_4\\ c_7 &= u_1+u_2+u_4\end{aligned}\right\} \Longrightarrow \boldsymbol{G} = \begin{bmatrix}1&0&0&0&1&0&1\\0&1&0&0&1&1&1\\0&0&1&0&1&1&0\\0&0&0&1&0&1&1\end{bmatrix}$$

Generator Matrix

$$\boldsymbol{G} = \left[\boldsymbol{I}_{k\times k}, \boldsymbol{P}_{k\times(n-k)}\right]$$

$$\left.\begin{aligned}c_1+c_2+c_3+c_5 &= 0\\ c_2+c_3+c_4+c_6 &= 0\\ c_1+c_2+c_4+c_7 &= 0\end{aligned}\right\} \Longrightarrow \boldsymbol{H} = \begin{bmatrix}1&1&1&0&1&0&0\\0&1&1&1&0&1&0\\1&1&0&1&0&0&1\end{bmatrix}$$

Parity-Check Matrix

Error Correction Codes

$$\left.\begin{aligned} c_1 &= u_1 \\ c_2 &= u_2 \\ c_3 &= u_3 \\ c_4 &= u_4 \\ c_5 &= u_1 + u_2 + u_3 \\ c_6 &= u_2 + u_3 + u_4 \\ c_7 &= u_1 + u_2 + u_4 \end{aligned}\right\} \Longrightarrow \boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Generator Matrix

$$\boldsymbol{G} = \left[\boldsymbol{I}_{k\times k}, \boldsymbol{P}_{k\times(n-k)}\right]$$

$$\left.\begin{aligned} c_1 + c_2 + c_3 + c_5 &= 0 \\ c_2 + c_3 + c_4 + c_6 &= 0 \\ c_1 + c_2 + c_4 + c_7 &= 0 \end{aligned}\right\} \Longrightarrow \boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Parity-Check Matrix

$$\boldsymbol{H} = \left[\boldsymbol{P}^t, \boldsymbol{I}_{(n-k)\times(n-k)}\right]$$

Error Correction Codes

$$\left.\begin{aligned}
c_1 &= u_1\\
c_2 &= u_2\\
c_3 &= u_3\\
c_4 &= u_4\\
c_5 &= u_1 + u_2 + u_3\\
c_6 &= u_2 + u_3 + u_4\\
c_7 &= u_1 + u_2 + u_4
\end{aligned}\right\} \Longrightarrow \boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1\\ 0 & 1 & 0 & 0 & 1 & 1 & 1\\ 0 & 0 & 1 & 0 & 1 & 1 & 0\\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Generator Matrix

$$\boldsymbol{G} = \left[\boldsymbol{I}_{k\times k}, \boldsymbol{P}_{k\times(n-k)}\right]$$

$$\left.\begin{aligned}
c_1 + c_2 + c_3 + c_5 &= 0\\
c_2 + c_3 + c_4 + c_6 &= 0\\
c_1 + c_2 + c_4 + c_7 &= 0
\end{aligned}\right\} \Longrightarrow \boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0\\ 0 & 1 & 1 & 1 & 0 & 1 & 0\\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Parity-Check Matrix

$$\boldsymbol{H} = \left[\boldsymbol{P}^t, \boldsymbol{I}_{(n-k)\times(n-k)}\right]$$

$$\boldsymbol{c} = \boldsymbol{uG} = [\boldsymbol{u}, \boldsymbol{p}] \qquad \boldsymbol{cH}^t = 0 \qquad \boldsymbol{GH}^t = 0$$

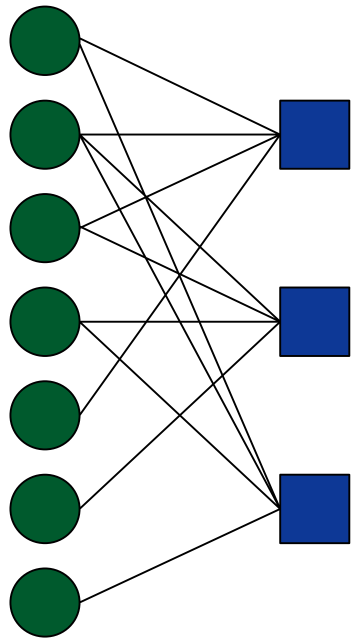
Overview

- Why Is Error Correction Needed in Flash Memories?
- Error Correction Codes Fundamentals
- Low-Density Parity-Check (LDPC) Codes
- LDPC Encoding and Decoding Methods
- Decoder Architectures for LDPC Codes

Tanner Graph Representation of Block Codes

Parity-check matrix:

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

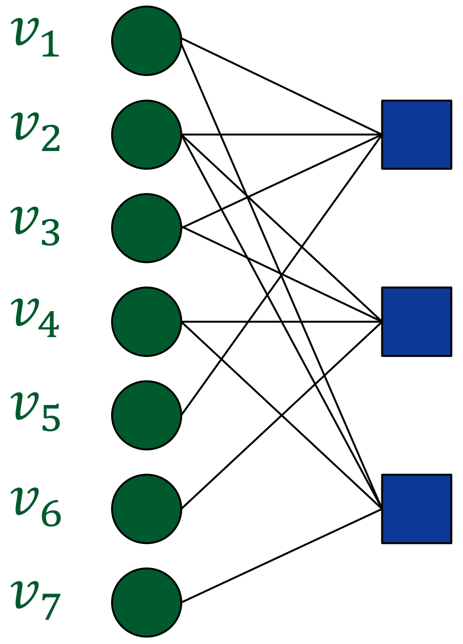


Tanner Graph Representation of Block Codes

Parity-check matrix:

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

Variable nodes: Left nodes



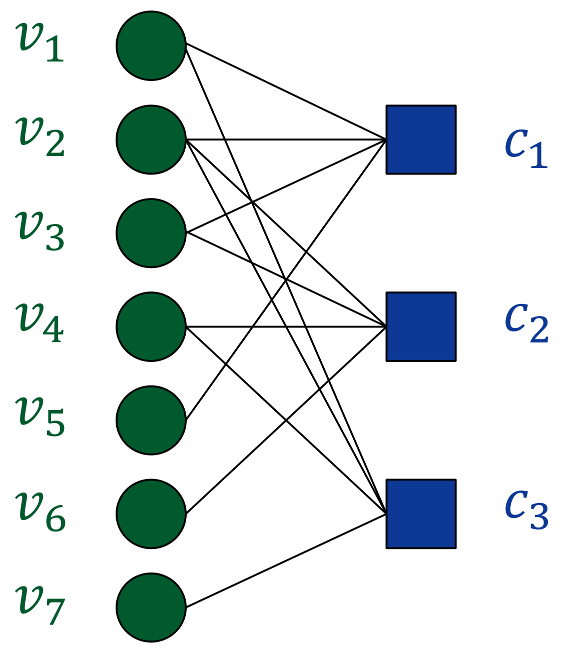
Tanner Graph Representation of Block Codes

Parity-check matrix:

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

Variable nodes: Left nodes

Check nodes: Right nodes



Tanner Graph Representation of Block Codes

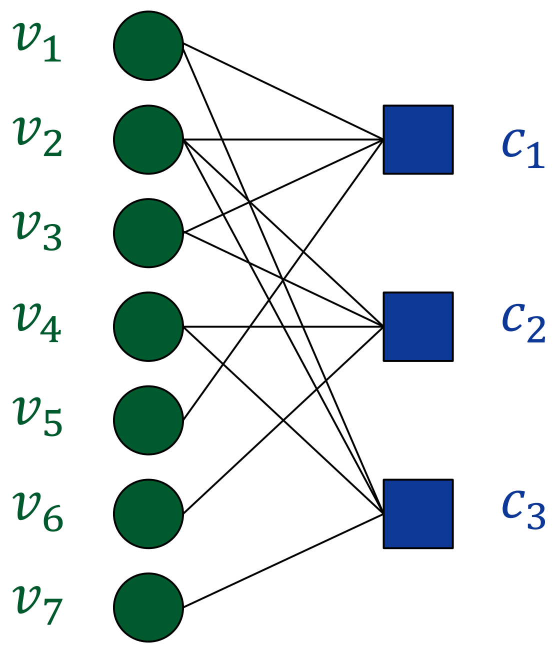
Parity-check matrix:

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

Variable nodes: Left nodes

Check nodes: Right nodes

Edges connect variable nodes and check nodes



Tanner Graph Representation of Block Codes

Parity-check matrix:

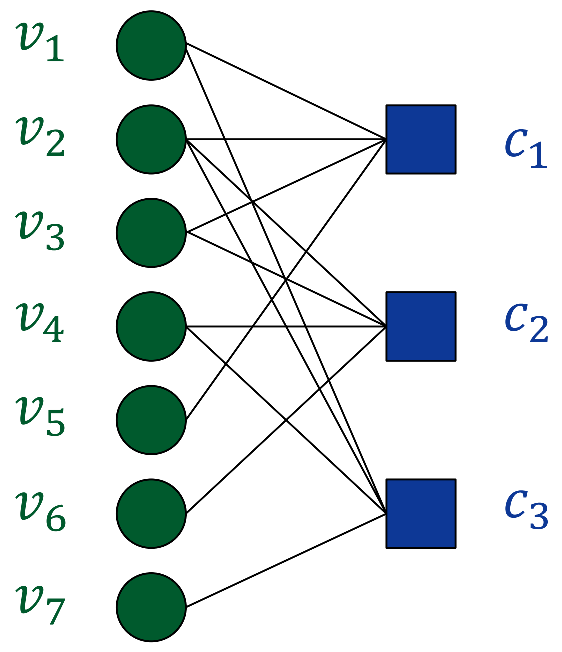
$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Variable nodes: Left nodes

Check nodes: Right nodes

Edges connect variable nodes and check nodes

Each **edge** represents a **'1'** in the $\boldsymbol{H}$ matrix



Tanner Graph Representation of Block Codes

Parity-check matrix:

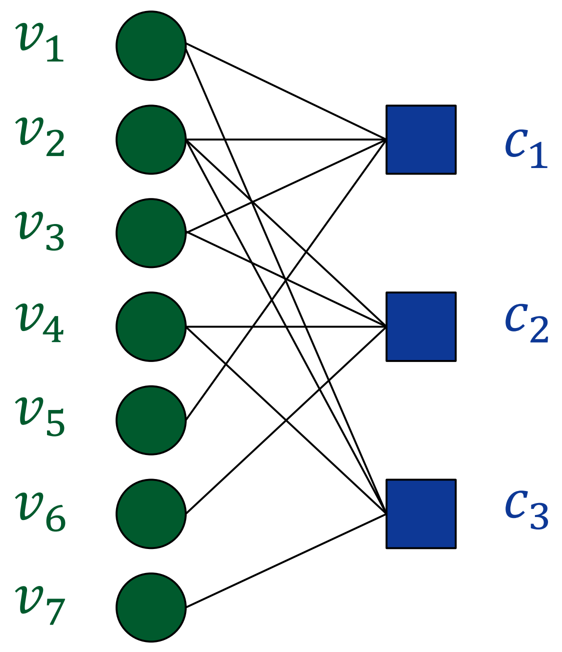
$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Variable nodes: Left nodes

Check nodes: Right nodes

Edges connect variable nodes and check nodes

Each **edge** represents a **‘1’** in the $\boldsymbol{H}$ matrix

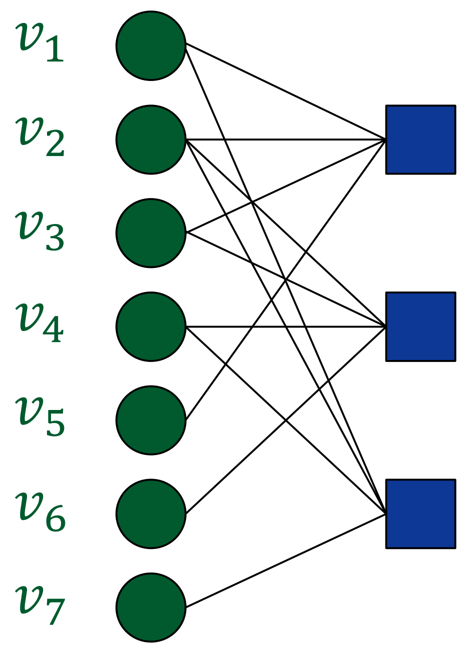


Degree of a node is the number of edges connected to it

Tanner Graph Representation of Block Codes

Parity-check matrix:

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



$c_1: \ v_1 + v_2 + v_3 + v_5 = 0$

$c_2: \ v_2 + v_3 + v_4 + v_6 = 0$

$c_3: v_1 + v_2 + v_4 + v_7 = 0$

Variable nodes: Left nodes

Check nodes: Right nodes

Edges connect variable nodes and check nodes

Each **edge** represents a **‘1’** in the ***H*** matrix

Degree of a node is the number of edges connected to it

Tanner Graph Representation of Block Codes

Parity-check matrix:

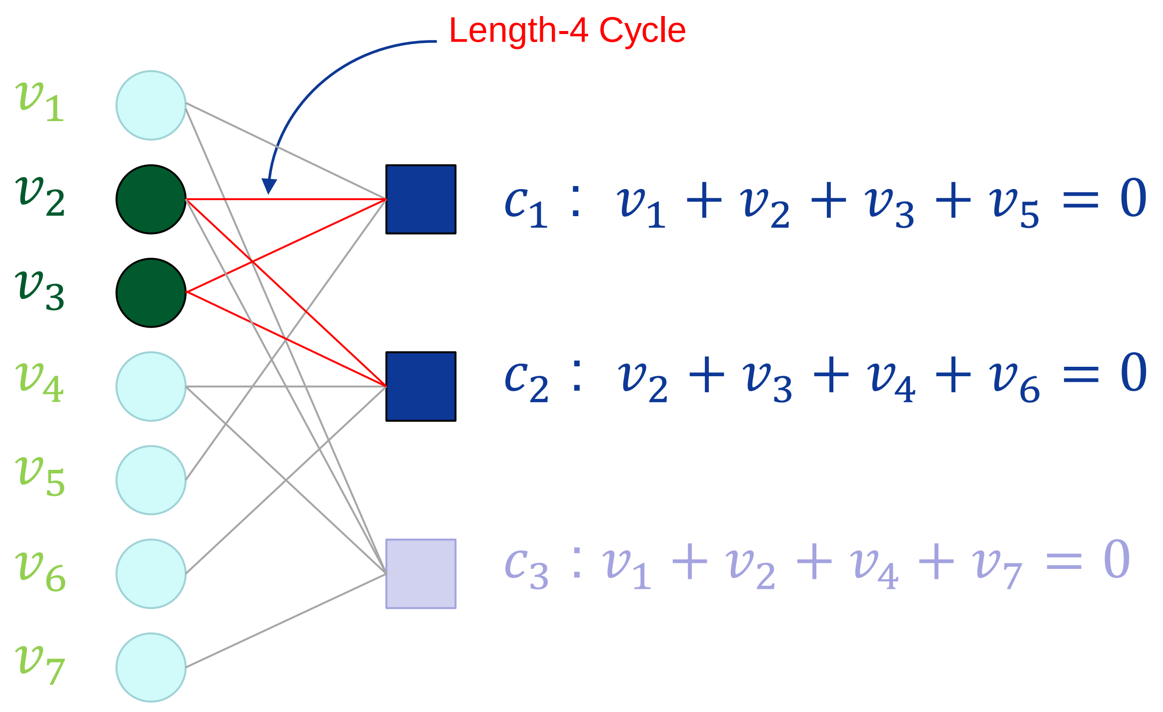
$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Variable nodes: Left nodes

Check nodes: Right nodes

Edges connect variable nodes and check nodes

Each **edge** represents a **‘1’** in the ***H*** matrix



Length-4 Cycle
v_1
v_2
v_3
v_4
v_5
v_6
v_7
$c_1 : v_1 + v_2 + v_3 + v_5 = 0$
$c_2 : v_2 + v_3 + v_4 + v_6 = 0$
$c_3 : v_1 + v_2 + v_4 + v_7 = 0$

Degree of a node is the number of edges connected to it

Tanner Graph Representation of Block Codes

Parity-check matrix:

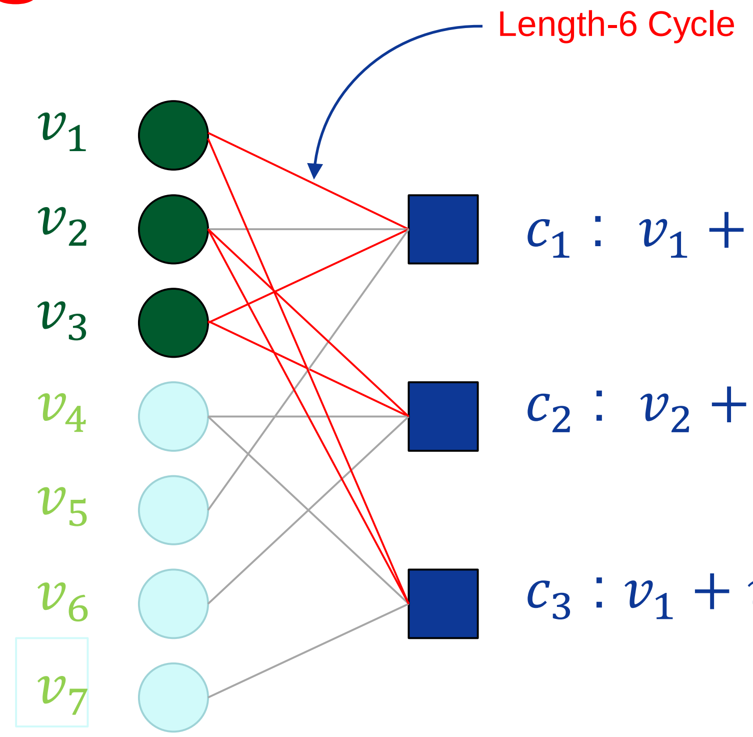
$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Variable nodes: Left nodes

Check nodes: Right nodes

Edges connect variable nodes and check nodes

Each **edge** represents a **'1'** in the $\boldsymbol{H}$ matrix



$c_1 : v_1 + v_2 + v_3 + v_5 = 0$

$c_2 : v_2 + v_3 + v_4 + v_6 = 0$

$c_3 : v_1 + v_2 + v_4 + v_7 = 0$

Degree of a node is the number of edges connected to it

Tanner Graph Representation of Block Codes

Parity-check matrix:

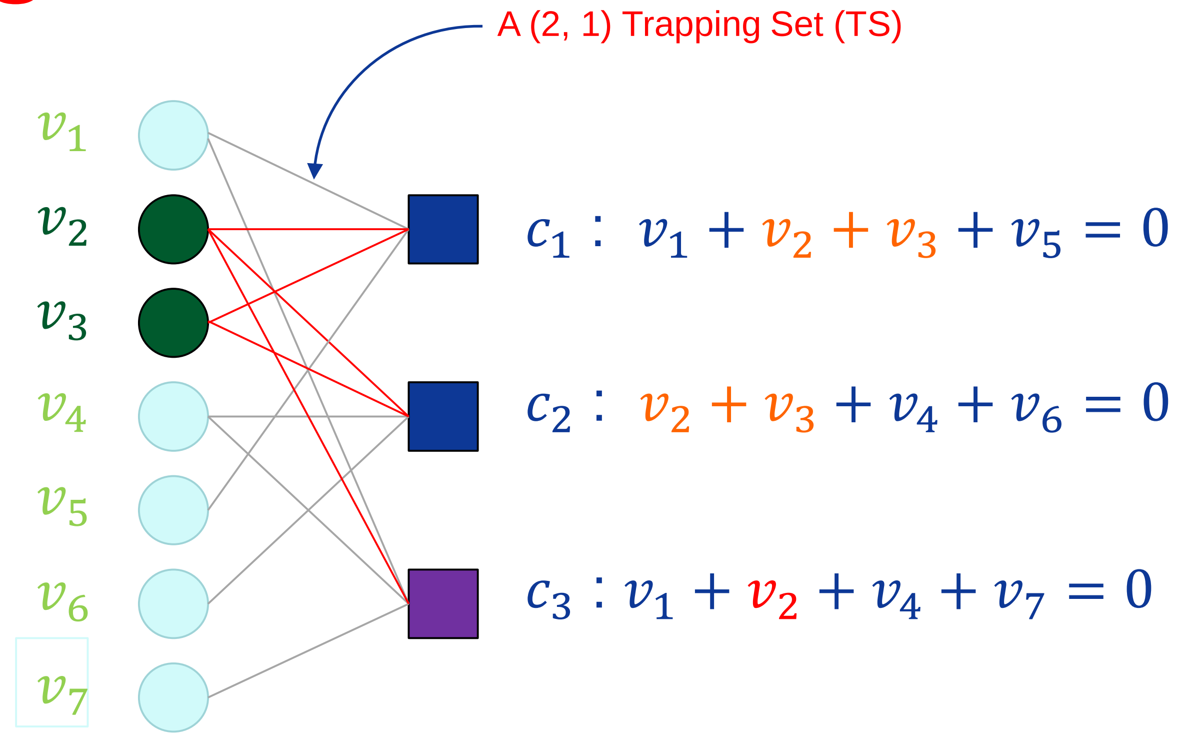
$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Variable nodes: Left nodes

Check nodes: Right nodes

Edges connect variable nodes and check nodes

Each **edge** represents a **‘1’** in the $\boldsymbol{H}$ matrix



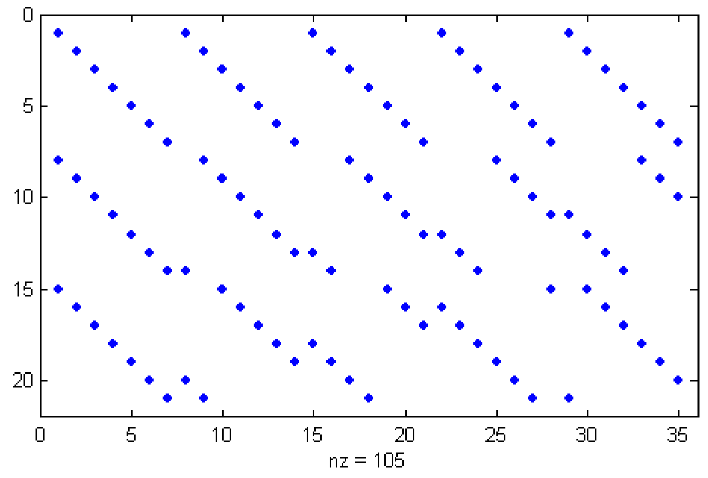
A (2, 1) Trapping Set (TS)
v_1
v_2
v_3
v_4
v_5
v_6
v_7
$c_1 : v_1 + v_2 + v_3 + v_5 = 0$
$c_2 : v_2 + v_3 + v_4 + v_6 = 0$
$c_3 : v_1 + v_2 + v_4 + v_7 = 0$

Degree of a node is the number of edges connected to it

LDPC Codes

- Linear block codes with low-density parity-check matrices
- Number of nonzeros increases linearly with the block length (sparseness)
- Iterative message passing decoders
- Decoding complexity depends linearly on the number of nonzeros and on block length
- The generator matrix is constructed from a sparse parity-check matrix

QC-LDPC Matrix: Example



$$\sigma = \begin{bmatrix} 0 & 0 & \ldots & 0 & 1 \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & .0 & 0 \\ \vdots & & & & \\ 0 & 0 & \ldots & .1 & 0 \end{bmatrix}$$

$$H = \begin{bmatrix} I & I & I & \ldots & I \\ I & \sigma & \sigma^2 & \ldots & \sigma^{r-1} \\ I & \sigma^2 & \sigma^4 & \ldots & \sigma^{2(r-1)} \\ \vdots & & & & \\ I & \sigma^{c-1} & \sigma^{(c-1)2} & \ldots & \sigma^{(c-1)(r-1)} \end{bmatrix}$$

Example H Matrix: Array LDPC code

r: row/ check node degree	= 5
c: column/variable node degree	= 3
Sc: circulant size	= 7
N: code length = Sc × r	= 35

Overview

- Why Is Error Correction Needed in Flash Memories?
- Error Correction Codes Fundamentals
- Low-Density Parity-Check (LDPC) Codes
- LDPC Encoding and Decoding Methods
- Decoder Architectures for LDPC Codes

LDPC Encoding

- For Systematic codes: $\boldsymbol{c} = [\boldsymbol{u}, \boldsymbol{p}]$.
- Suppose $\boldsymbol{H} = [\boldsymbol{H_u}, \boldsymbol{H_p}]$, where $\boldsymbol{H_p}$ is $(n-m) \times (n-m)$ and invertible. Then

$$\boldsymbol{cH}^t = 0$$

$$= [\boldsymbol{u}, \boldsymbol{p}] \times \begin{bmatrix} \boldsymbol{H}_u^{\boldsymbol{t}} \\ \boldsymbol{H}_p^{\boldsymbol{t}} \end{bmatrix}$$

$$= \boldsymbol{uH}_u^{\boldsymbol{t}} + \boldsymbol{pH}_p^{\boldsymbol{t}}$$

$$\boldsymbol{pH}_p^{\boldsymbol{t}} = \boldsymbol{uH}_u^{\boldsymbol{t}}$$

$$\boldsymbol{p} = \boldsymbol{uH}_u^{\boldsymbol{t}} \left(\boldsymbol{H}_p^{-\boldsymbol{1}}\right)^{\boldsymbol{t}}$$

LDPC Encoding

- For Systematic codes: $\boldsymbol{c} = [\boldsymbol{u}, \boldsymbol{p}]$.
- Suppose $\boldsymbol{H} = [\boldsymbol{H_u}, \boldsymbol{H_p}]$, where $\boldsymbol{H_p}$ is $(n-m) \times (n-m)$ and invertible. Then

$$\begin{aligned} \boldsymbol{cH}^t &= 0 \\ &= [\boldsymbol{u}, \boldsymbol{p}] \times \begin{bmatrix} \boldsymbol{H}_u^{\boldsymbol{t}} \\ \boldsymbol{H}_p^{\boldsymbol{t}} \end{bmatrix} \\ &= \boldsymbol{uH}_u^{\boldsymbol{t}} + \boldsymbol{pH}_p^{\boldsymbol{t}} \end{aligned}$$

$$\boldsymbol{pH}_p^{\boldsymbol{t}} = \boldsymbol{uH}_u^{\boldsymbol{t}}$$

$$\boldsymbol{p} \quad = \boldsymbol{uH}_u^{\boldsymbol{t}} \left(\boldsymbol{H}_p^{-\boldsymbol{1}}\right)^{\boldsymbol{t}}$$

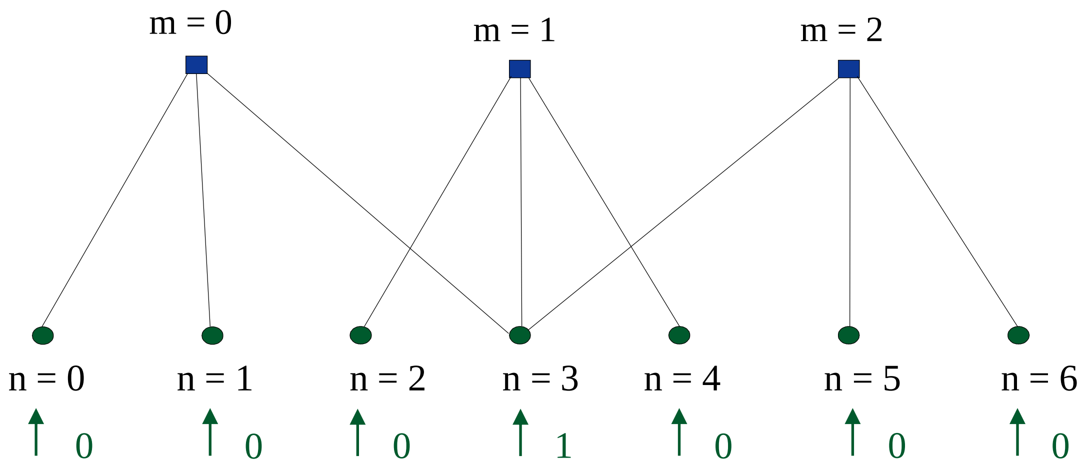
LDPC Decoding (The Bit-Flipping Algorithm), 1/2

LDPC Decoding (The Bit-Flipping Algorithm), 1/2

- Bit flipping is a hard decision (HD) decoding method

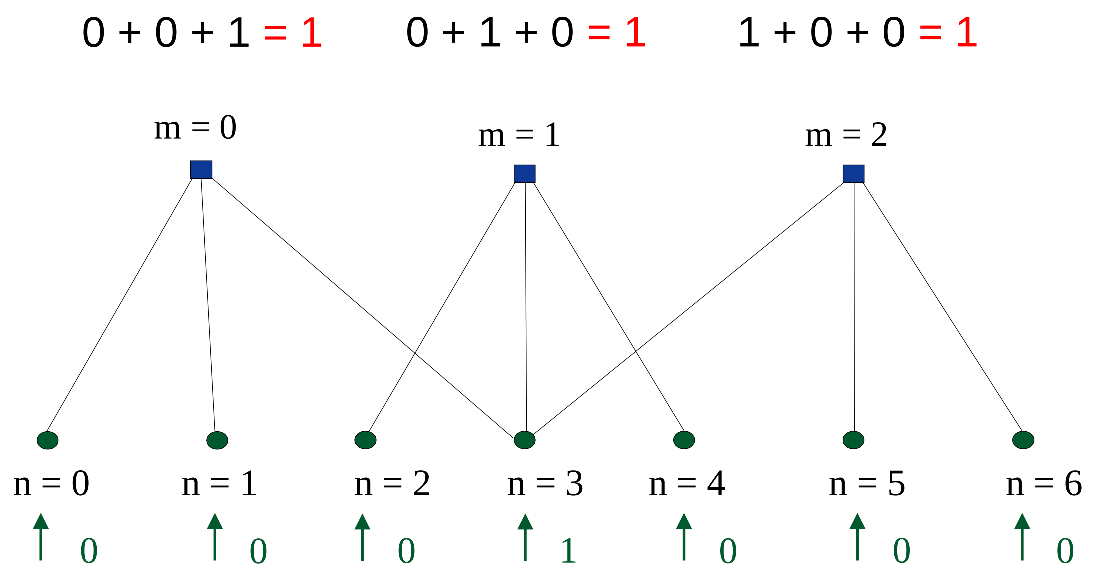
LDPC Decoding (The Bit-Flipping Algorithm), 1/2

- Bit flipping is a hard decision (HD) decoding method



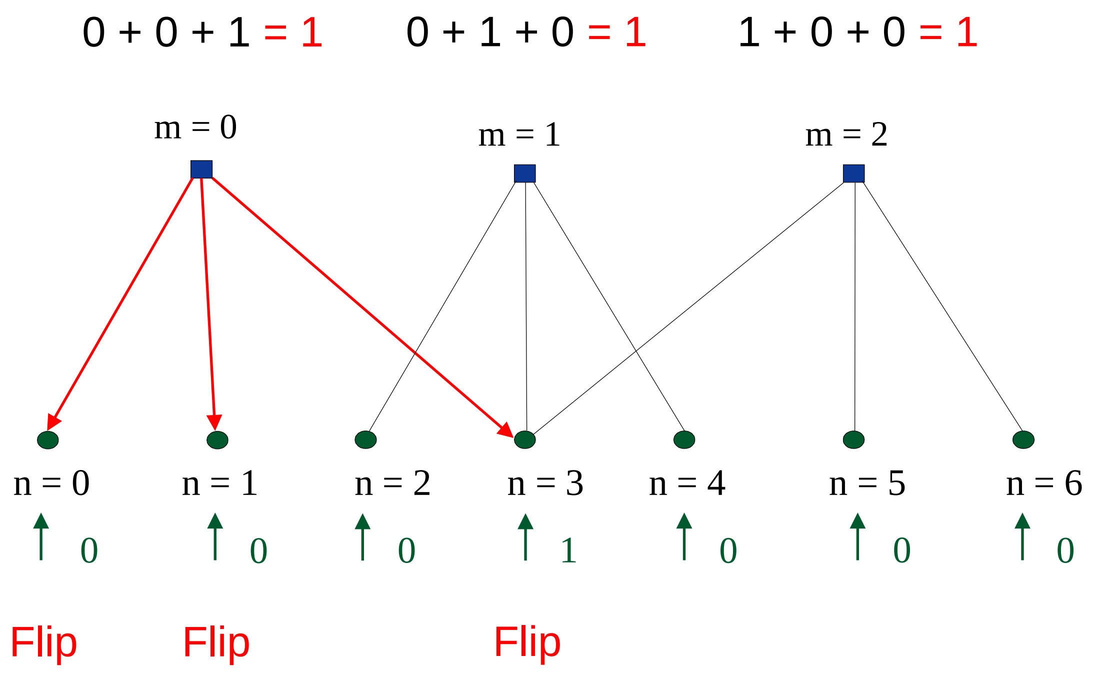
LDPC Decoding (The Bit-Flipping Algorithm), 1/2

- Bit flipping is a hard decision (HD) decoding method



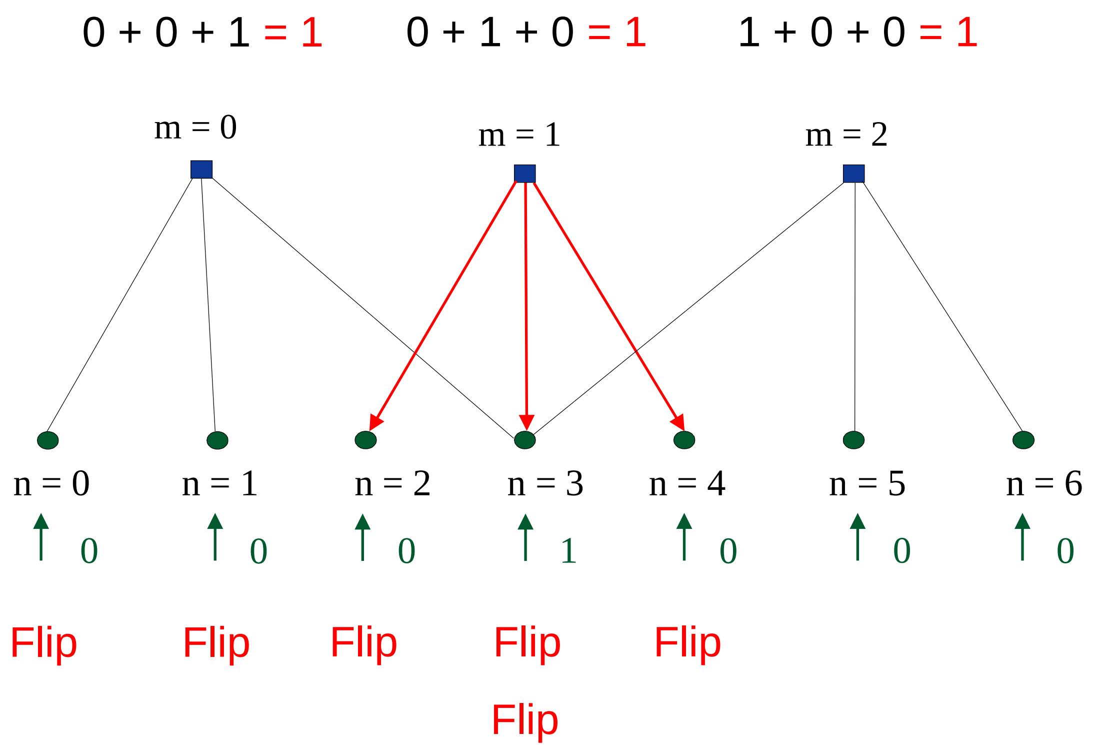
LDPC Decoding (The Bit-Flipping Algorithm), 1/2

- Bit flipping is a hard decision (HD) decoding method



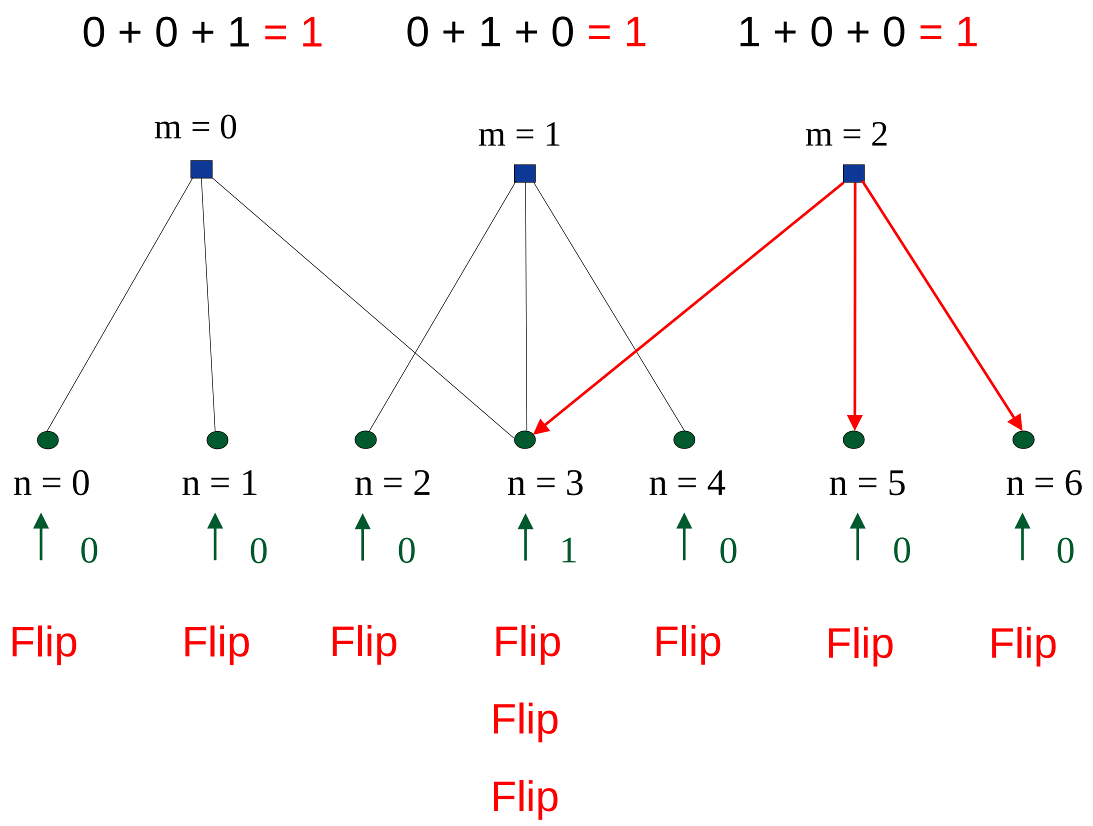
LDPC Decoding (The Bit-Flipping Algorithm), 1/2

- Bit flipping is a hard decision (HD) decoding method



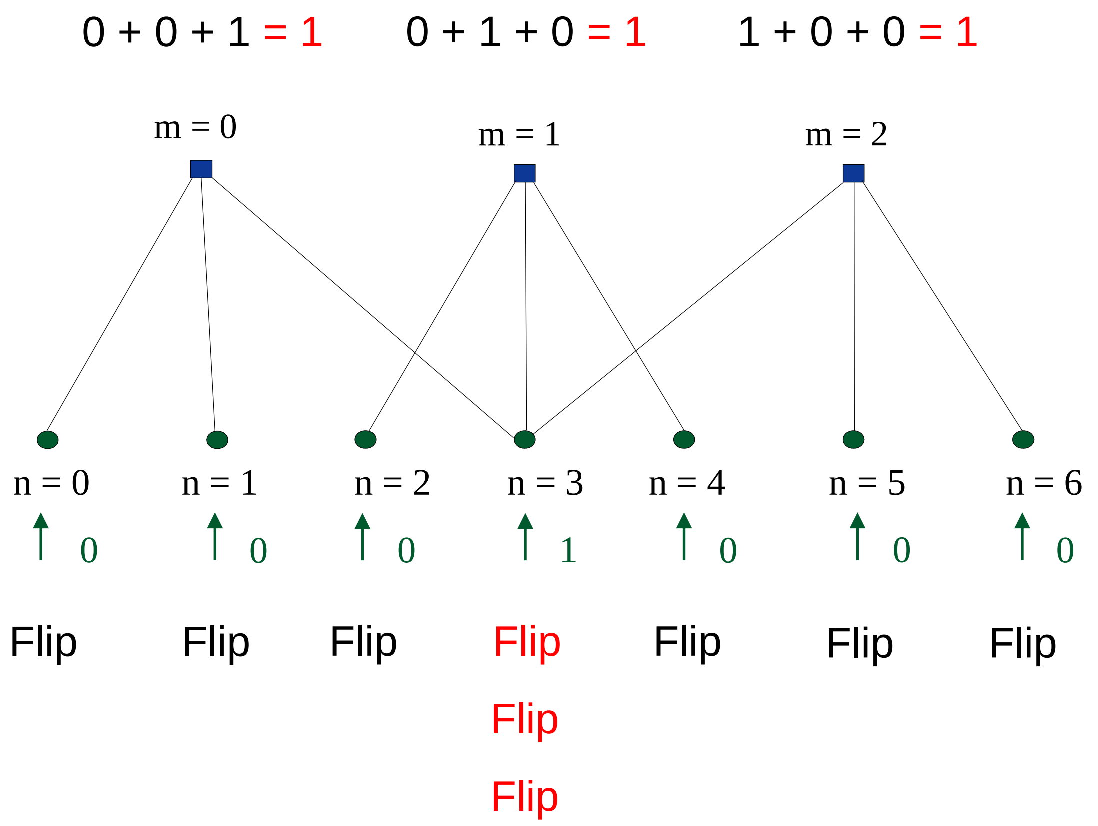
LDPC Decoding (The Bit-Flipping Algorithm), 1/2

- Bit flipping is a hard decision (HD) decoding method



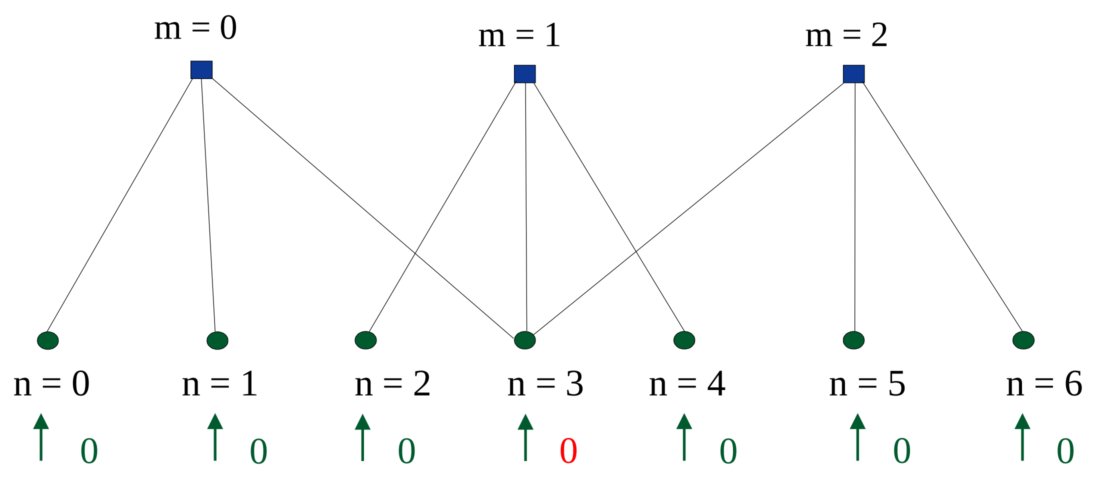
LDPC Decoding (The Bit-Flipping Algorithm), 1/2

- Bit flipping is a hard decision (HD) decoding method



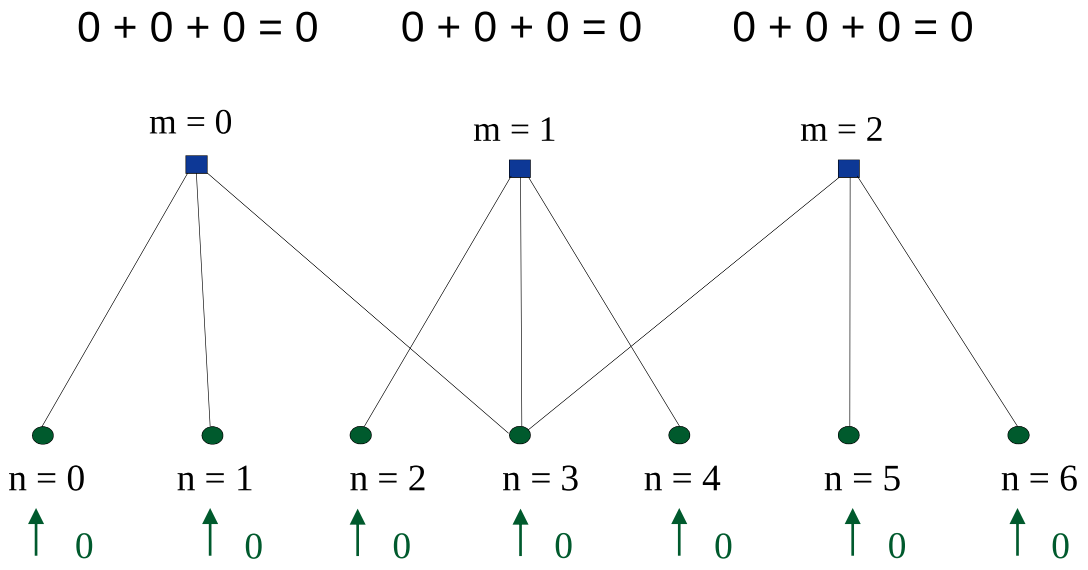
LDPC Decoding (The Bit-Flipping Algorithm), 2/2

- Bit flipping is a hard decision (HD) decoding method



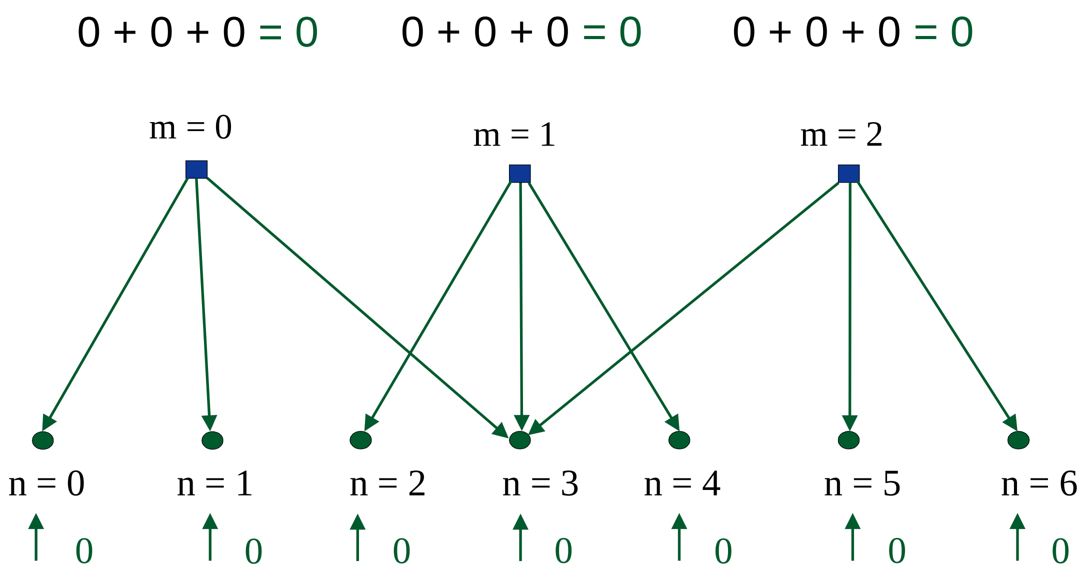
LDPC Decoding (The Bit-Flipping Algorithm), 2/2

- Bit flipping is a hard decision (HD) decoding method



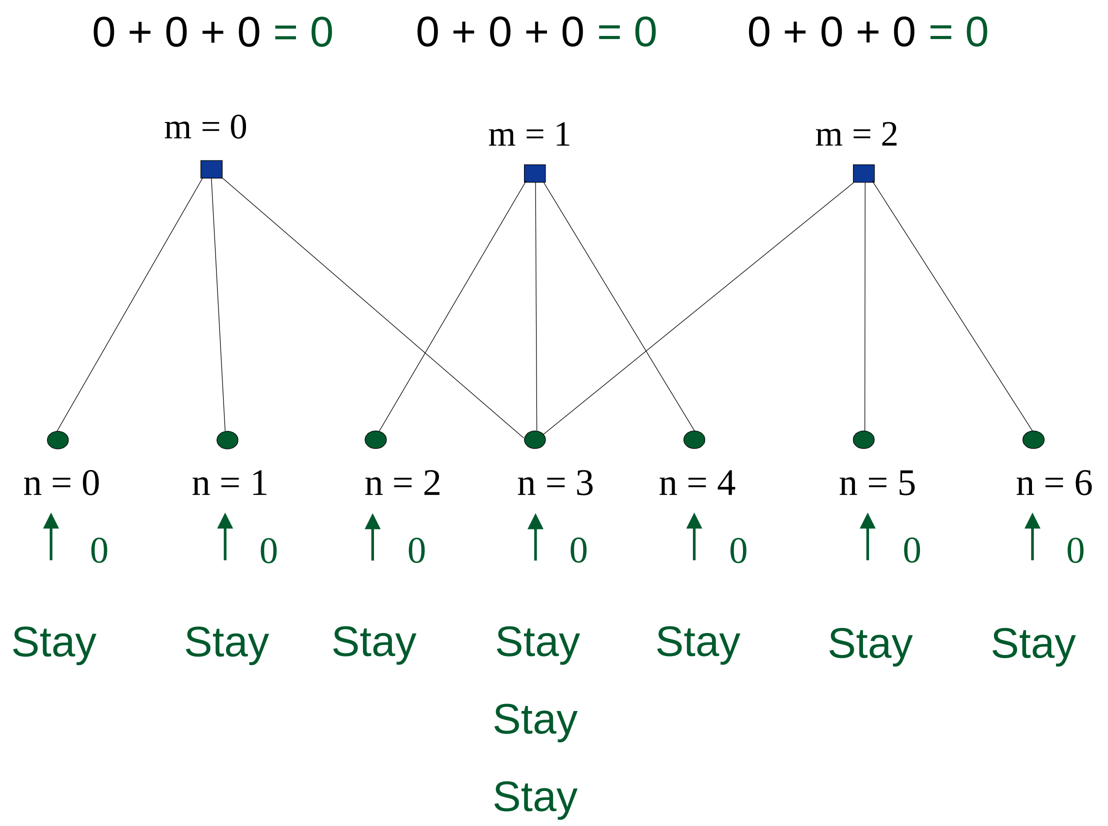
LDPC Decoding (The Bit-Flipping Algorithm), 2/2

- Bit flipping is a hard decision (HD) decoding method



LDPC Decoding (The Bit-Flipping Algorithm), 2/2

- Bit flipping is a hard decision (HD) decoding method

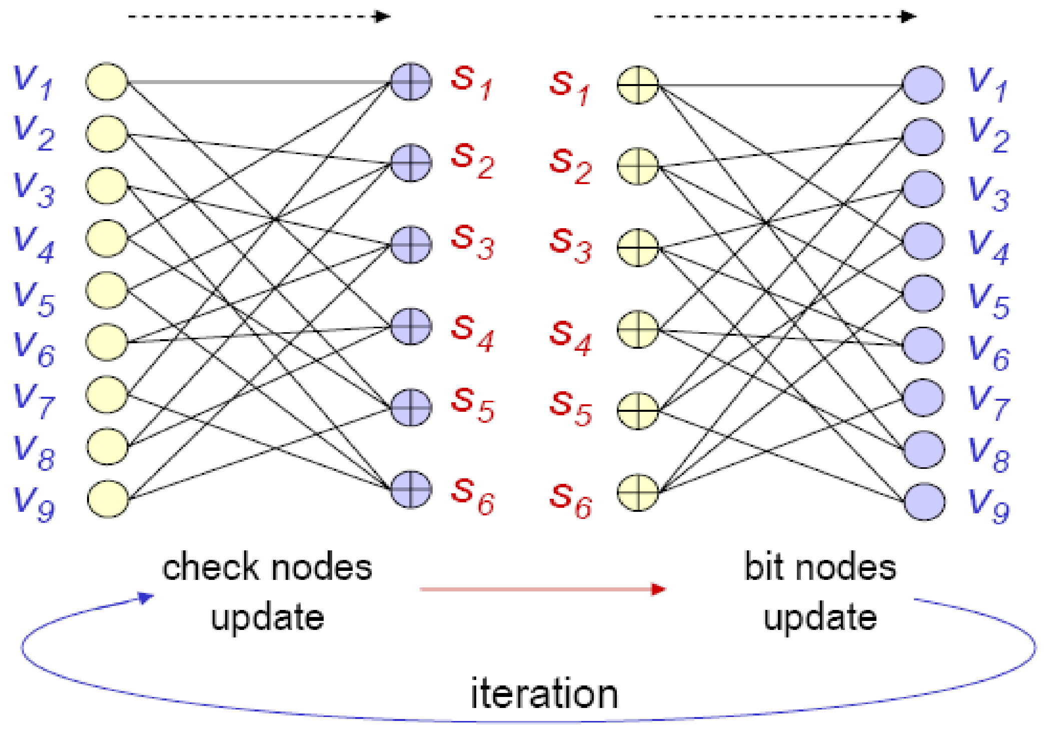


Message Passing Decoding of LDPC Codes, 1/2

check nodes

$$H=\begin{array}{c|c}
\left(\begin{array}{ccccccccc}
1&0&0&1&0&0&1&0&0\\
0&1&0&0&1&0&0&1&0\\
0&0&1&0&0&1&0&0&1\\
1&0&0&0&0&1&0&1&0\\
0&1&0&1&0&0&0&0&1\\
0&0&1&0&1&0&1&0&0
\end{array}\right) & \begin{array}{c} s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6 \end{array}\\
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 &
\end{array}$$

variable nodes

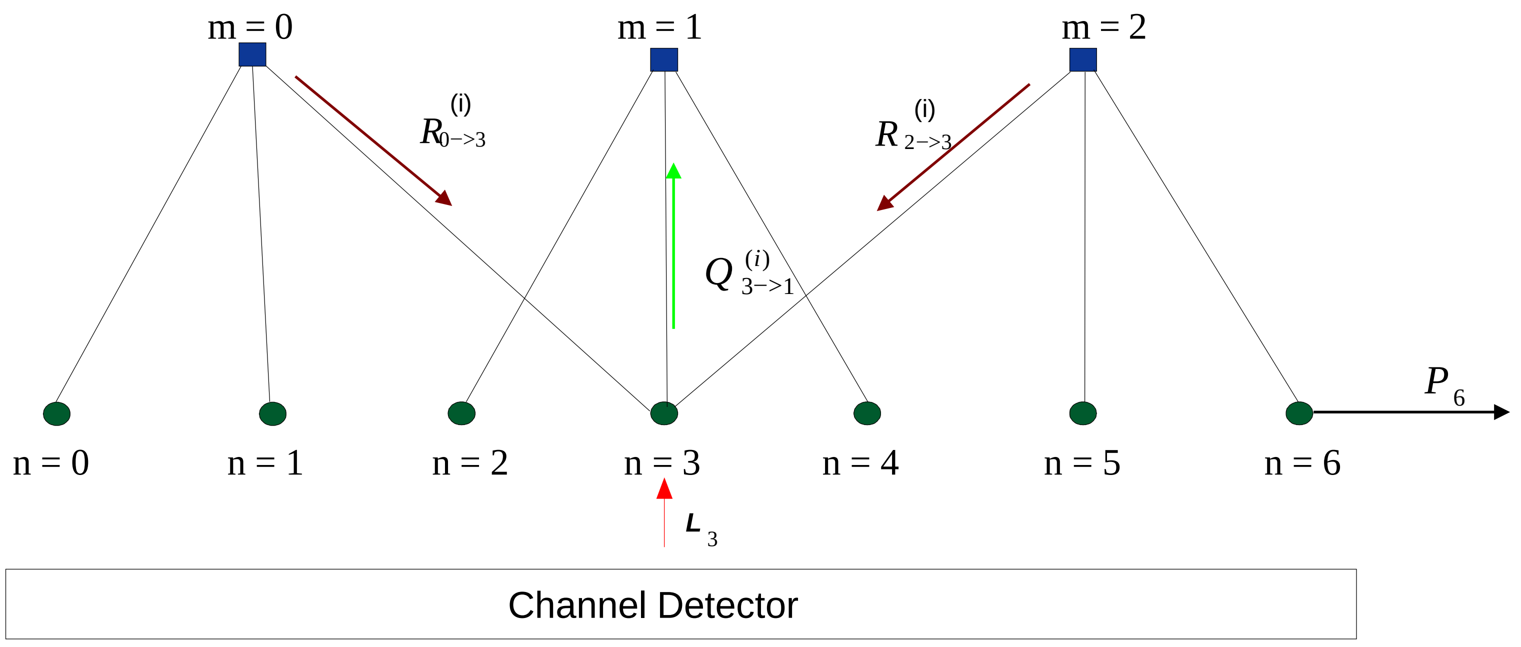


v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9
s_1 s_2 s_3 s_4 s_5 s_6
s_1 s_2 s_3 s_4 s_5 s_6
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9
check nodes update
bit nodes update
iteration

- The decoding is successful when all the parity checks are satisfied (i.e. zero).

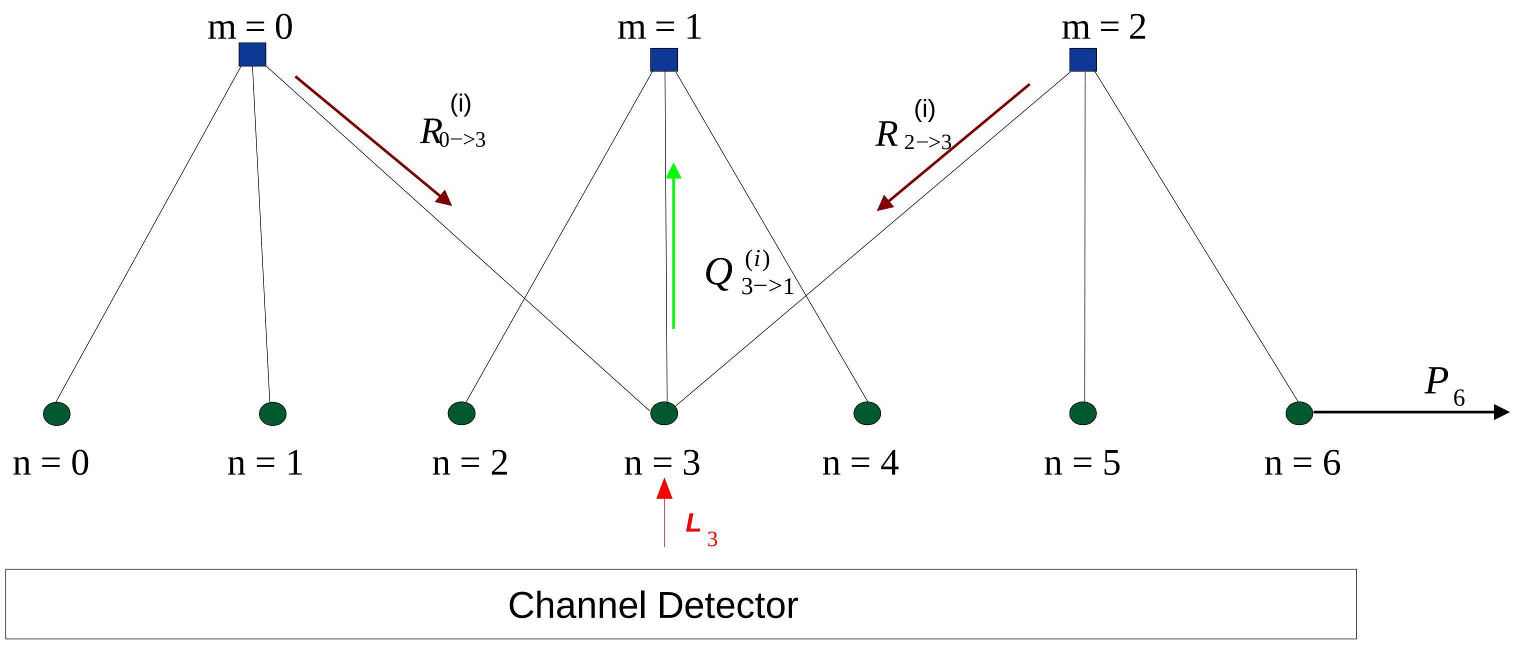
Message Passing Decoding of LDPC Codes, 2/2

- There are four types of LLR messages



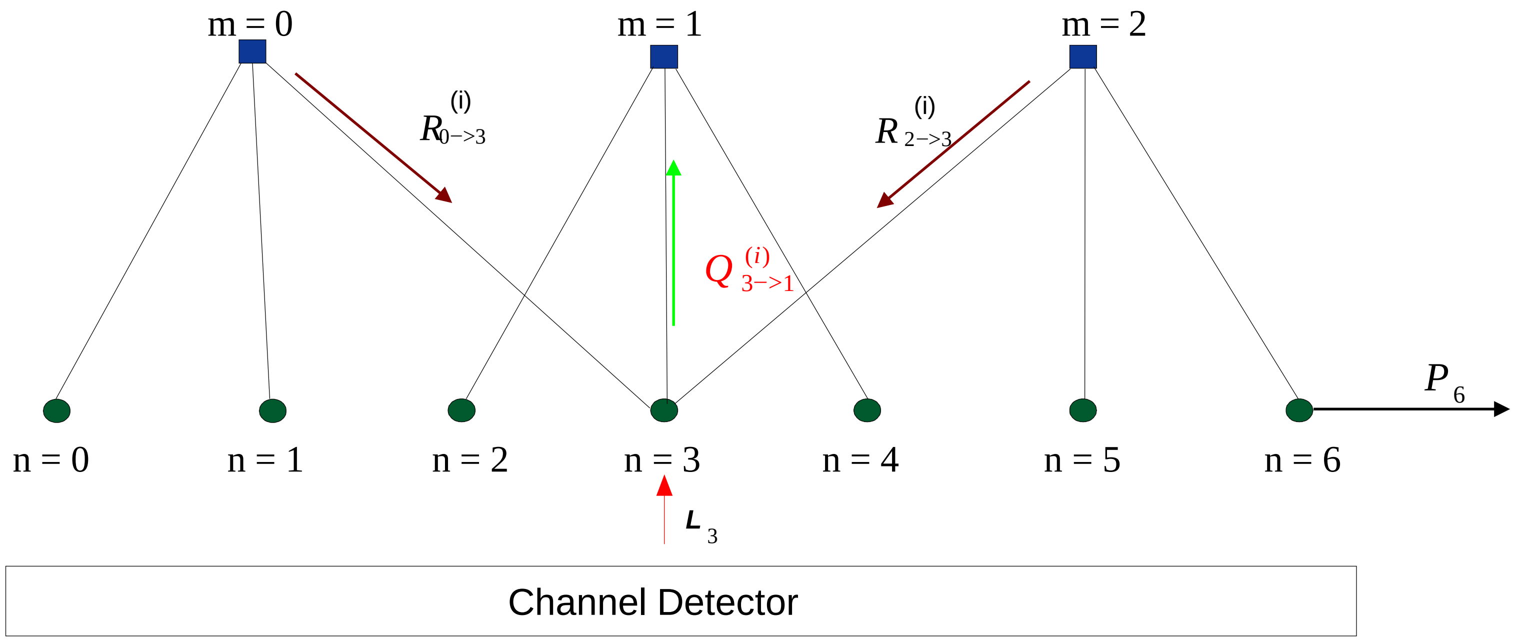
Message Passing Decoding of LDPC Codes, 2/2

- There are four types of LLR messages
  - Message from the channel to the n-th bit node, L_n



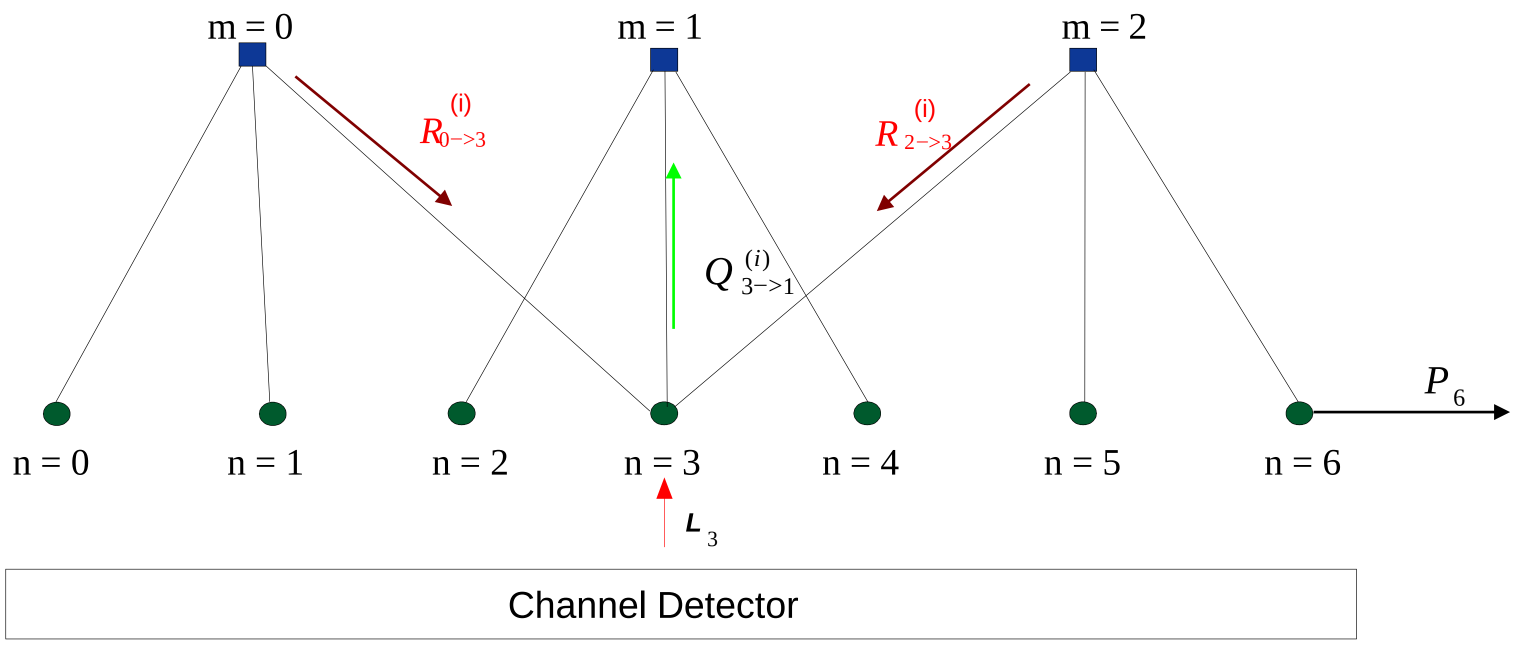
Message Passing Decoding of LDPC Codes, 2/2

- There are four types of LLR messages
  - Message from the channel to the n-th bit node, L_n
  - Message from n-th bit node to the m-th check node $Q^{(i)}_{n->m}$ or simply $Q^{(i)}_{nm}$



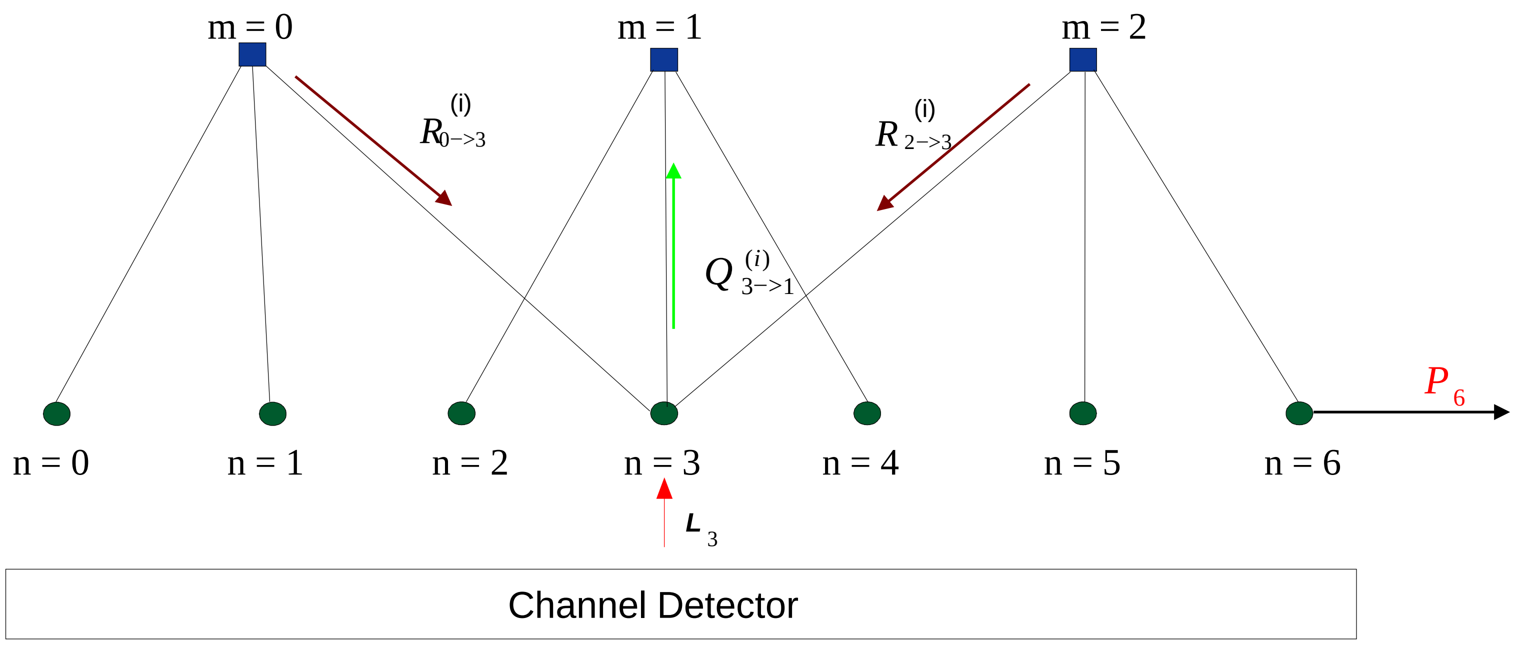
Message Passing Decoding of LDPC Codes, 2/2

- There are four types of LLR messages
  - Message from the channel to the n-th bit node, L_n
  - Message from n-th bit node to the m-th check node $Q^{(i)}_{n->m}$ or simply $Q^{(i)}_{nm}$
  - Message from the m-th check node to the n-th bit node $R^{(i)}_{m->n}$ or simply $R^{(i)}_{mn}$



Message Passing Decoding of LDPC Codes, 2/2

- There are four types of LLR messages
  - Message from the channel to the n-th bit node, L_n
  - Message from n-th bit node to the m-th check node $Q^{(i)}_{n->m}$ or simply $Q^{(i)}_{nm}$
  - Message from the m-th check node to the n-th bit node $R^{(i)}_{m->n}$ or simply $R^{(i)}_{mn}$
  - Overall reliability information for n-th bit-node P_n



The Min-Sum Algorithm, 1/3

Notation used in the equations

x_n is the transmitted bit n,

L_n is the initial LLR message for a bit node (also called as variable node) n, received from channel/detector

P_n is the overall LLR message for a bit node n,

$\hat{x}_n$ is the decoded bit n (hard decision based on P_n),

[Frequency of P and hard decision update depends on decoding schedule]

$\mathrm{M}(n)$ is the set of the neighboring check nodes for variable node n,

$\mathrm{N}(m)$ is the set of the neighboring bit nodes for check node m.

For the i^{th} iteration,

$Q_{nm}^{(i)}$ is the LLR message from bit node n to check node m,

$R_{mn}^{(i)}$ is the LLR message from check node m to bit node n.

The Min-Sum Algorithm, 2/3

(A) check node processing: for each m and $n \in \mathrm{N}(m)$,

$$R_{mn}^{(i)} = \delta_{mn}^{(i)} \kappa_{mn}^{(i)} \qquad (1)$$

$$\kappa_{mn}^{(i)} = \left| R_{mn}^{(i)} \right| = \min_{n' \in \mathrm{N}(m) \setminus n} \left| Q_{n'm}^{(i-1)} \right| \qquad (2)$$

The Min-Sum Algorithm, 2/3

(A) check node processing: for each m and $n \in \mathrm{N}(m)$,

$$R_{mn}^{(i)} = \delta_{mn}^{(i)} \kappa_{mn}^{(i)} \qquad (1)$$

$$\kappa_{mn}^{(i)} = \left| R_{mn}^{(i)} \right| = \min_{n' \in \mathrm{N}(m) \setminus n} \left| Q_{n'm}^{(i-1)} \right| \qquad (2)$$

The sign of check node message $R_{mn}^{(i)}$ is defined as

$$\delta_{mn}^{(i)} = \left(\prod_{n' \in \mathrm{N}(m) \setminus n} \mathrm{sgn}\left(Q_{n'm}^{(i-1)} \right) \right) \qquad (3)$$

where $\delta_{mn}^{(i)}$ takes value of $+1$ or -1

The Min-Sum Algorithm, 3/3

(B) *Variable-node processing*: for each n and $m \in M(n)$:

$$Q_{nm}^{(i)} = L_n + \sum_{m' \in \mathrm{M}(n) \setminus m} R_{m'n}^{(i)} \qquad (4)$$

The Min-Sum Algorithm, 3/3

(B) *Variable-node processing*: for each n and $m \in M(n)$:

$$Q_{nm}^{(i)} = L_n + \sum_{m' \in \mathrm{M}(n) \backslash m} R_{m'n}^{(i)} \qquad (4)$$

(C) P Update and Hard Decision

$$P_n = L_n + \sum_{m \in M(n)} R_{mn}^{(i)} \qquad (5)$$

The Min-Sum Algorithm, 3/3

(B) *Variable-node processing*: for each n and $m \in M(n)$:

$$Q_{nm}^{(i)} = L_n + \sum_{m' \in \mathrm{M}(n) \setminus m} R_{m'n}^{(i)} \qquad (4)$$

(C) P Update and Hard Decision

$$P_n = L_n + \sum_{m \in M(n)} R_{mn}^{(i)} \qquad (5)$$

A hard decision is taken where $\hat{x}_n = 0$ if $P_n \geq 0$, and $\hat{x}_n = 1$ if $P_n < 0$.

The Min-Sum Algorithm, 3/3

(B) *Variable-node processing*: for each n and $m \in M(n)$:

$$Q_{nm}^{(i)} = L_n + \sum_{m' \in M(n) \backslash m} R_{m'n}^{(i)} \qquad (4)$$

(C) P Update and Hard Decision

$$P_n = L_n + \sum_{m \in M(n)} R_{mn}^{(i)} \qquad (5)$$

A hard decision is taken where $\hat{x}_n = 0$ if $P_n \geq 0$, and $\hat{x}_n = 1$ if $P_n < 0$.

If $\hat{x}_n H^T = 0$, the decoding process is finished with $\hat{x}_n$ as the decoder output; otherwise, repeat steps (A) to (C).

The Min-Sum Algorithm, 3/3

(B) *Variable-node processing*: for each n and $m \in M(n)$:

$$Q_{nm}^{(i)} = L_n + \sum_{m' \in \mathrm{M}(n) \backslash m} R_{m'n}^{(i)} \qquad (4)$$

(C) P Update and Hard Decision

$$P_n = L_n + \sum_{m \in M(n)} R_{mn}^{(i)} \qquad (5)$$

A hard decision is taken where $\hat{x}_n = 0$ if $P_n \geq 0$, and $\hat{x}_n = 1$ if $P_n < 0$.

If $\hat{x}_n H^T = 0$, the decoding process is finished with $\hat{x}_n$ as the decoder output; otherwise, repeat steps (A) to (C).

If the decoding process doesn't end within some maximum iteration, stop and output error message.

The Min-Sum Algorithm, 3/3

(B) *Variable-node processing*: for each n and $m \in M(n)$:

$$Q_{nm}^{(i)} = L_n + \sum_{m' \in M(n) \backslash m} R_{m'n}^{(i)} \qquad (4)$$

(C) P Update and Hard Decision

$$P_n = L_n + \sum_{m \in M(n)} R_{mn}^{(i)} \qquad (5)$$

A hard decision is taken where $\hat{x}_n = 0$ if $P_n \geq 0$, and $\hat{x}_n = 1$ if $P_n < 0$.

If $\hat{x}_n H^T = 0$, the decoding process is finished with $\hat{x}_n$ as the decoder output; otherwise, repeat steps (A) to (C).

If the decoding process doesn't end within some maximum iteration, stop and output error message.

Scaling or offset can be applied on R messages and/or Q messages for better performance.

The Min-Sum Algorithm, 3/3

(B) *Variable-node processing*: for each n and $m \in M(n)$:

$$Q_{nm}^{(i)} = L_n + \sum_{m' \in M(n) \setminus m} R_{m'n}^{(i)} \qquad (4)$$

(C) P Update and Hard Decision

$$P_n = L_n + \sum_{m \in M(n)} R_{mn}^{(i)} \qquad (5)$$

A hard decision is taken where $\hat{x}_n = 0$ if $P_n \geq 0$, and $\hat{x}_n = 1$ if $P_n < 0$.

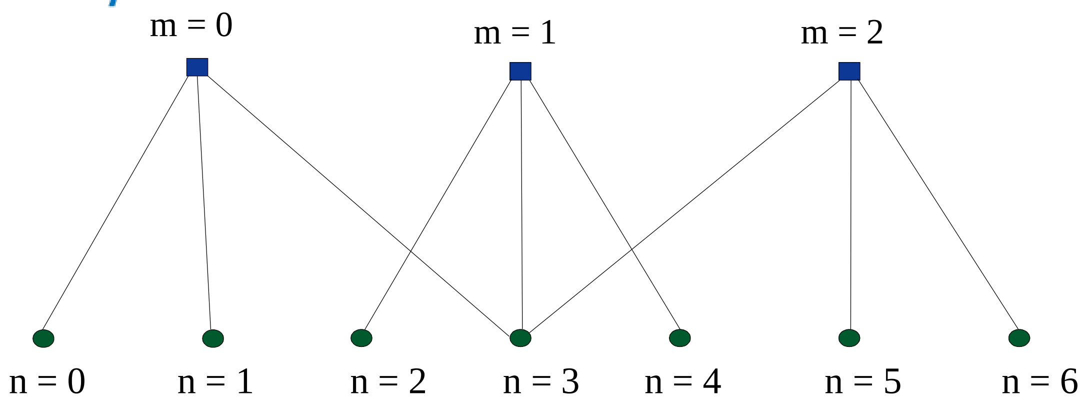
If $\hat{x}_n H^T = 0$, the decoding process is finished with $\hat{x}_n$ as the decoder output; otherwise, repeat steps (A) to (C).

If the decoding process doesn't end within some maximum iteration, stop and output error message.

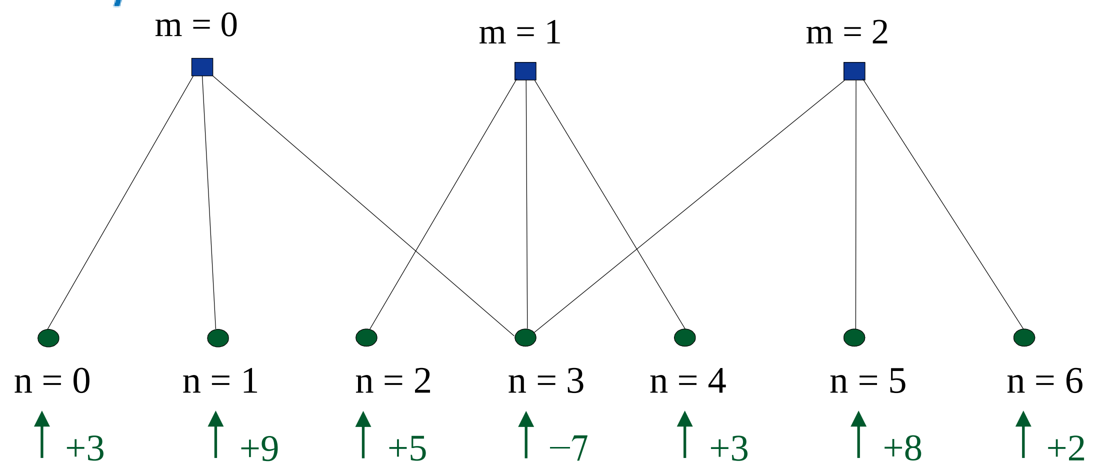
Scaling or offset can be applied on R messages and/or Q messages for better performance.

The Min-Sum algorithm can be used in both hard-decision (HD) and soft-decision (SD) modes. In HD mode, LLRs have same magnitude with lower bit resolution

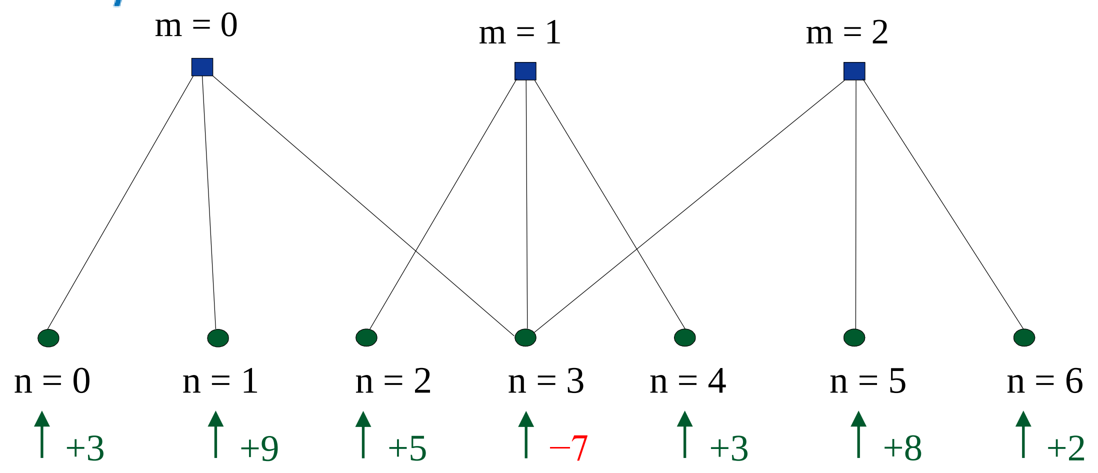
LDPC Decoding Example, 1/3



LDPC Decoding Example, 1/3

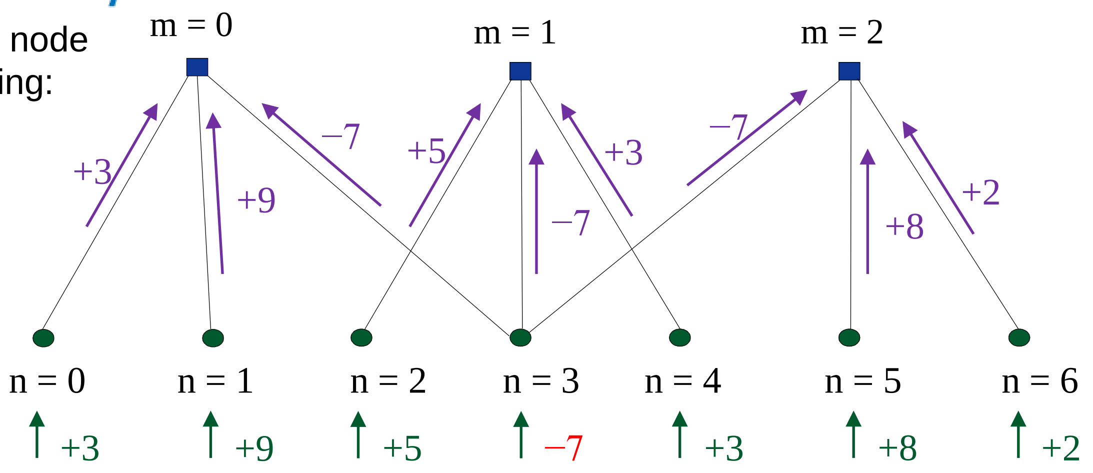


LDPC Decoding Example, 1/3



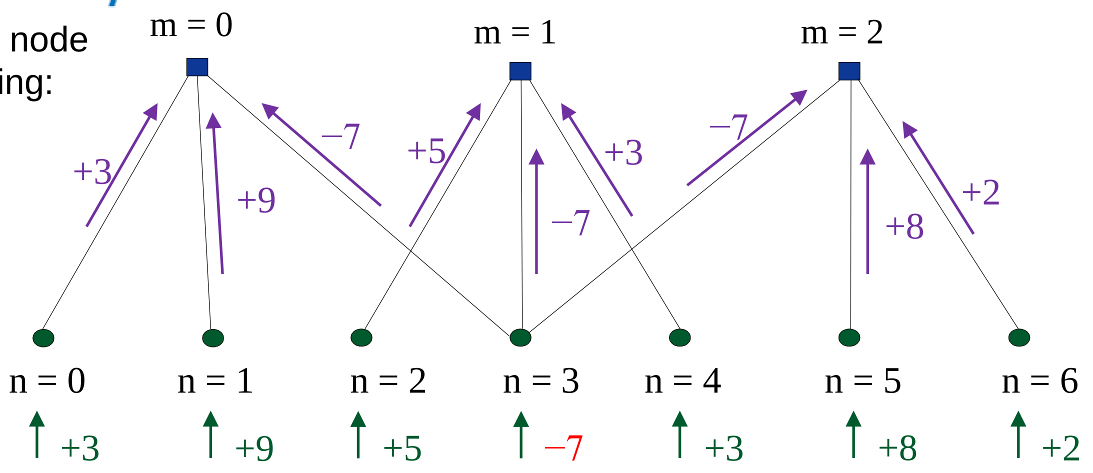
LDPC Decoding Example, 1/3

Variable node processing:

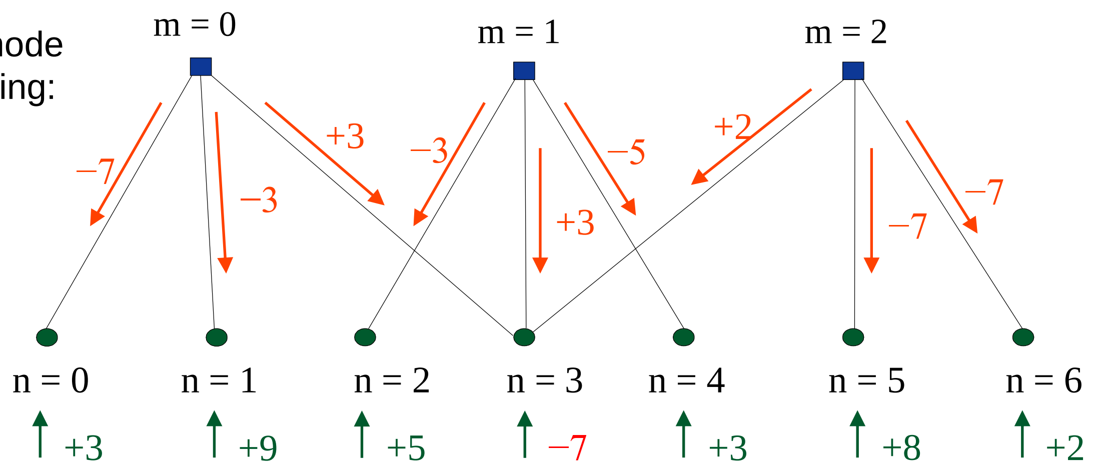


LDPC Decoding Example, 1/3

Variable node processing:

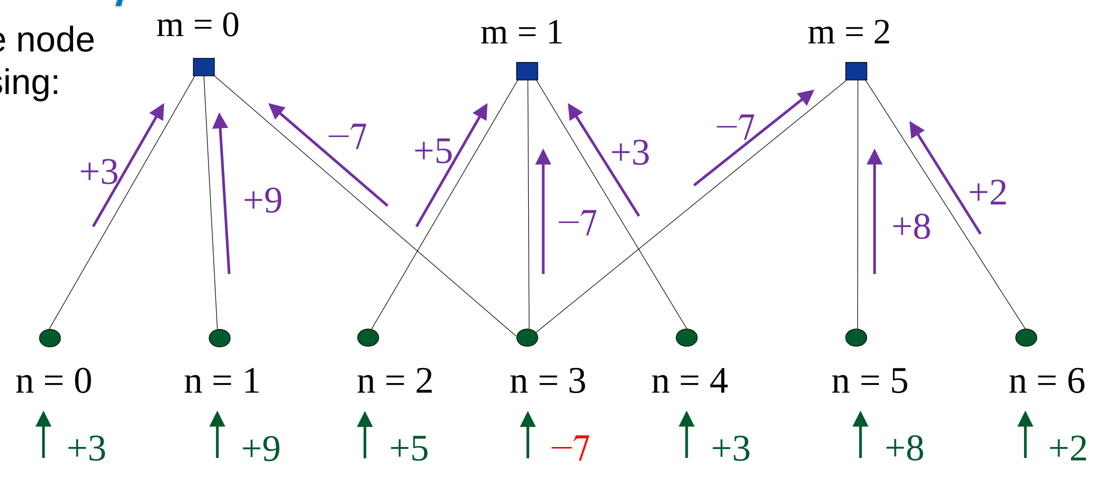


Check node processing:

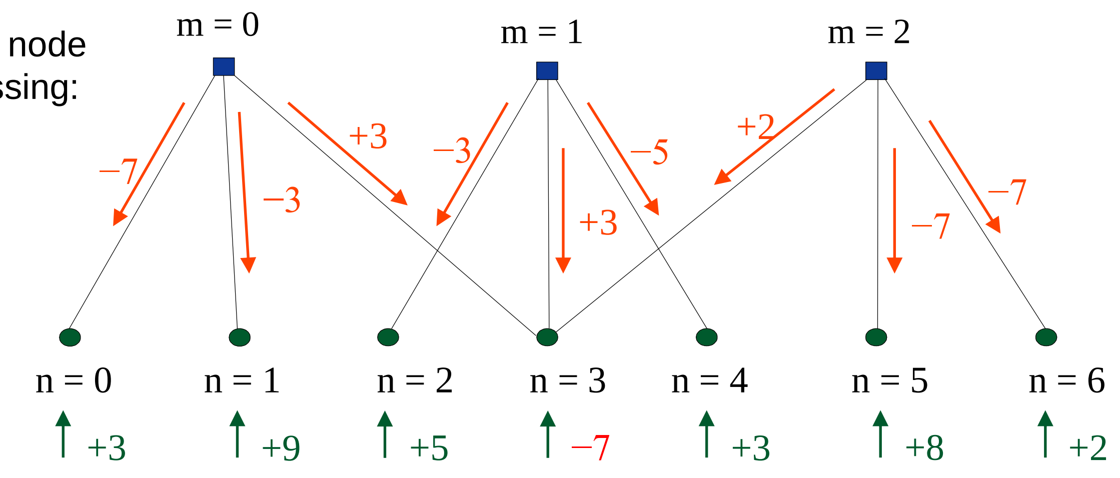


LDPC Decoding Example, 1/3

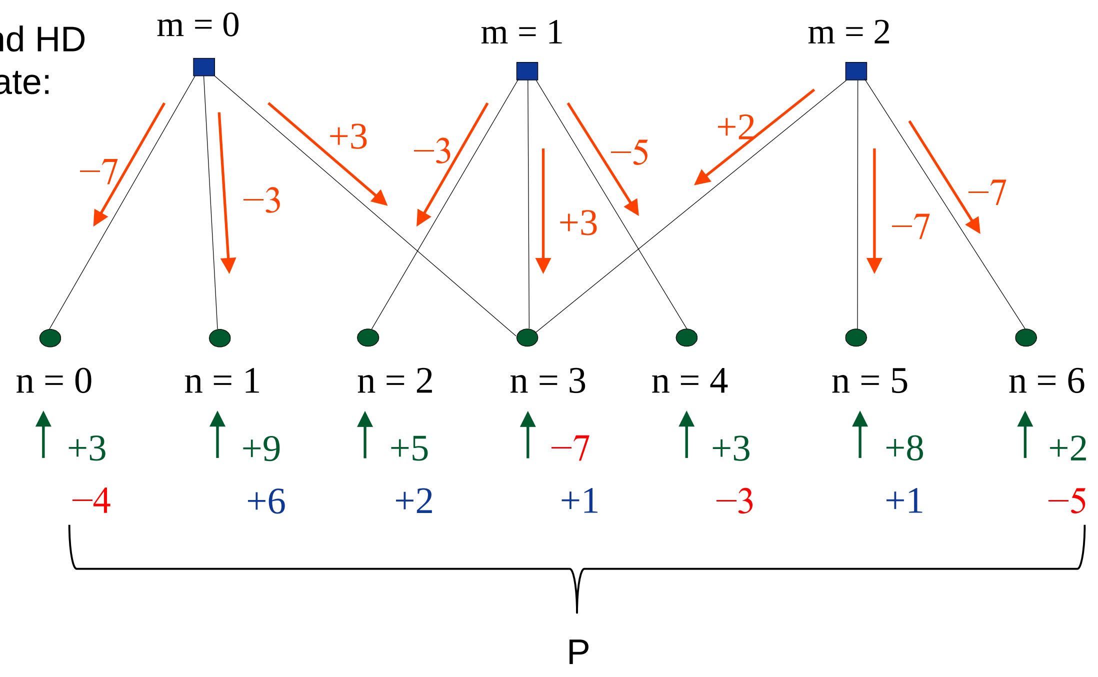
Variable node processing:



Check node processing:

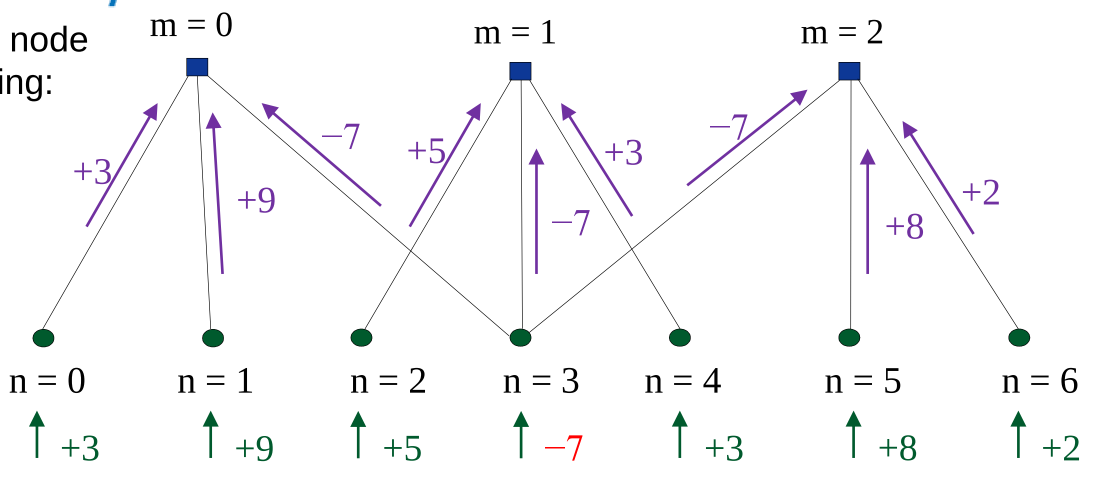


P and HD update:

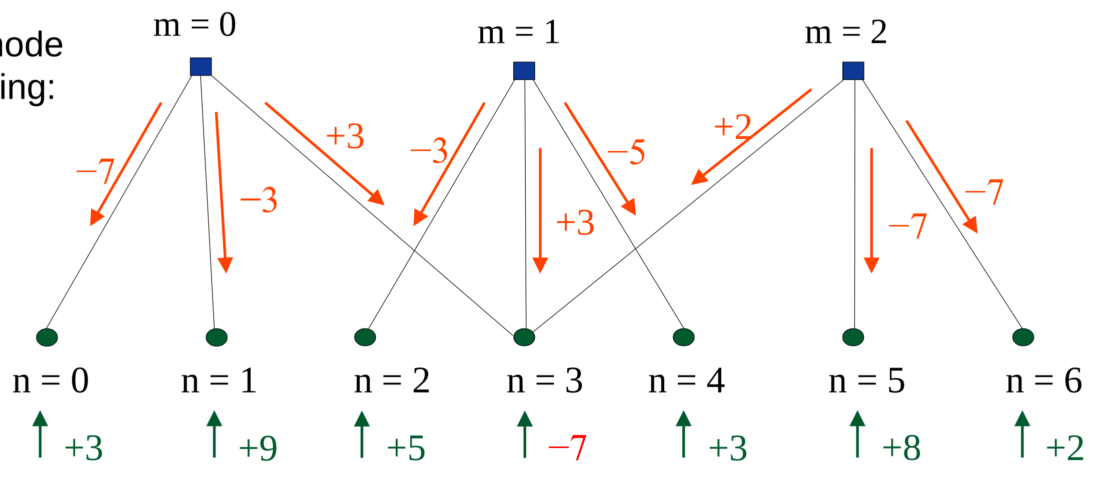


LDPC Decoding Example, 1/3

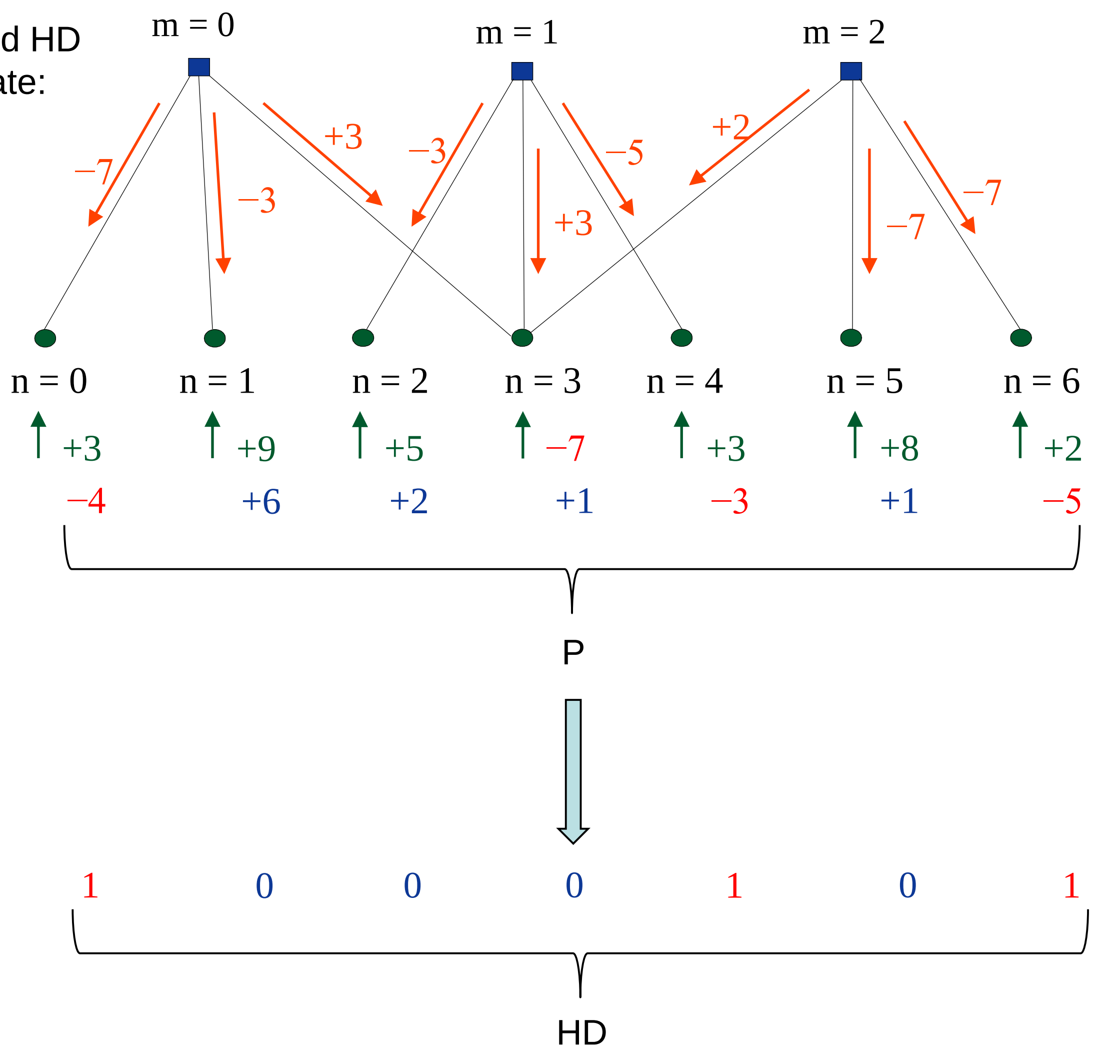
Variable node processing:



Check node processing:

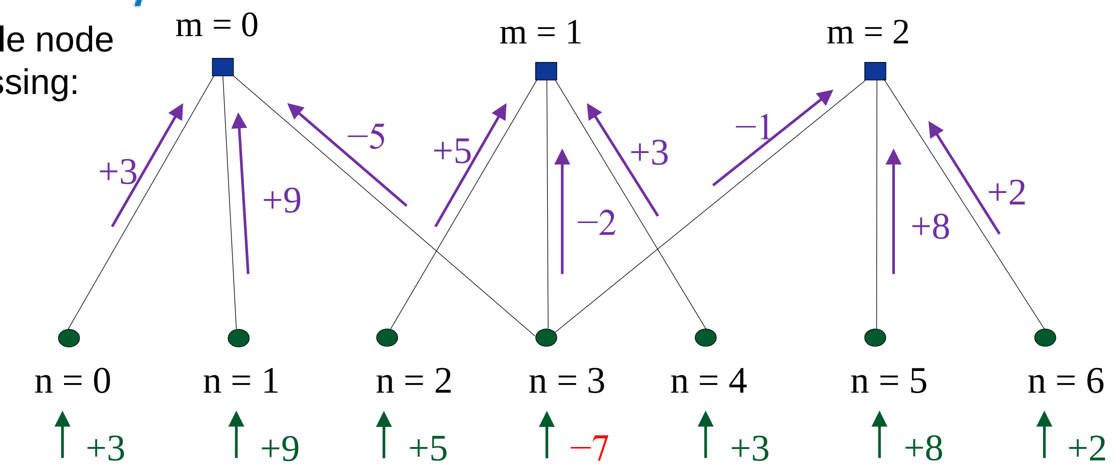


P and HD update:



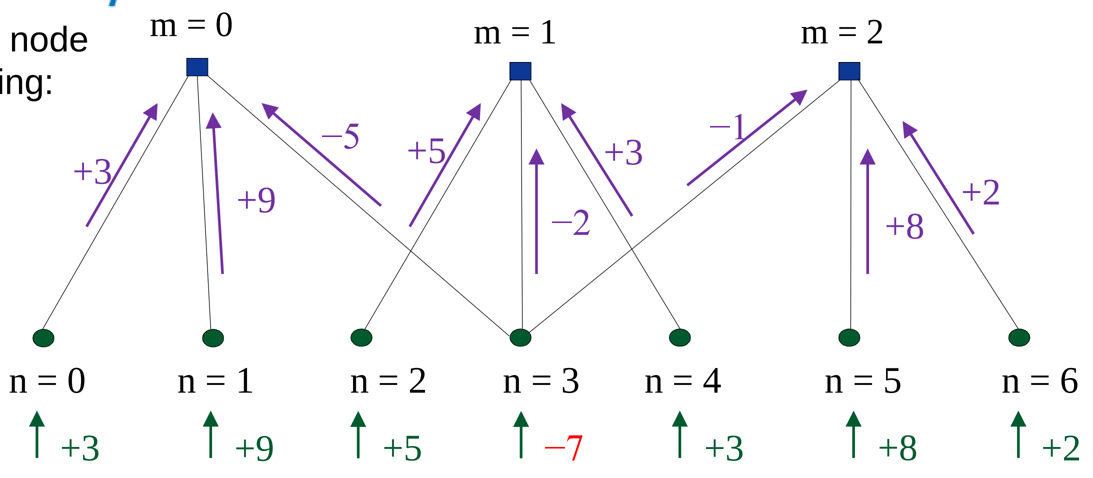
LDPC Decoding Example, 2/3

Variable node processing:

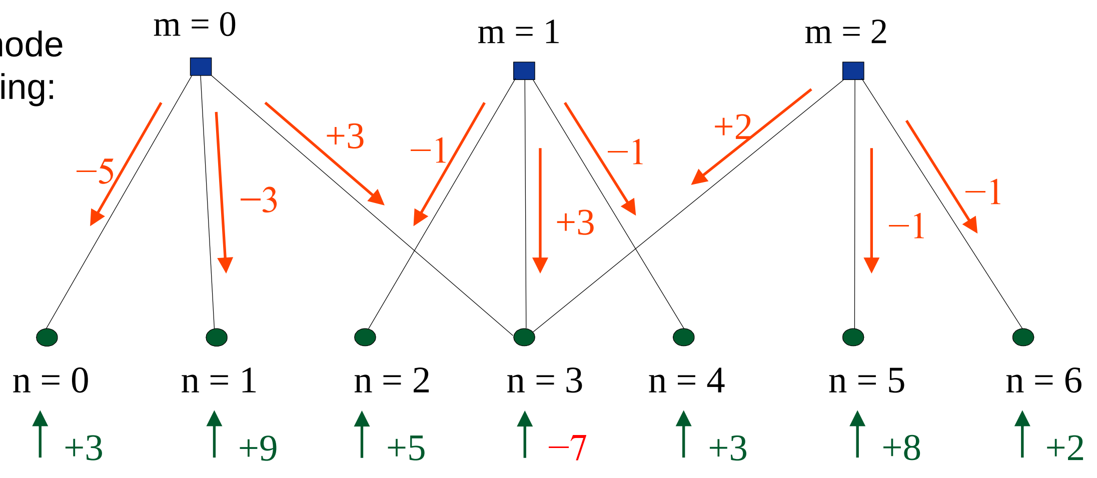


LDPC Decoding Example, 2/3

Variable node processing:

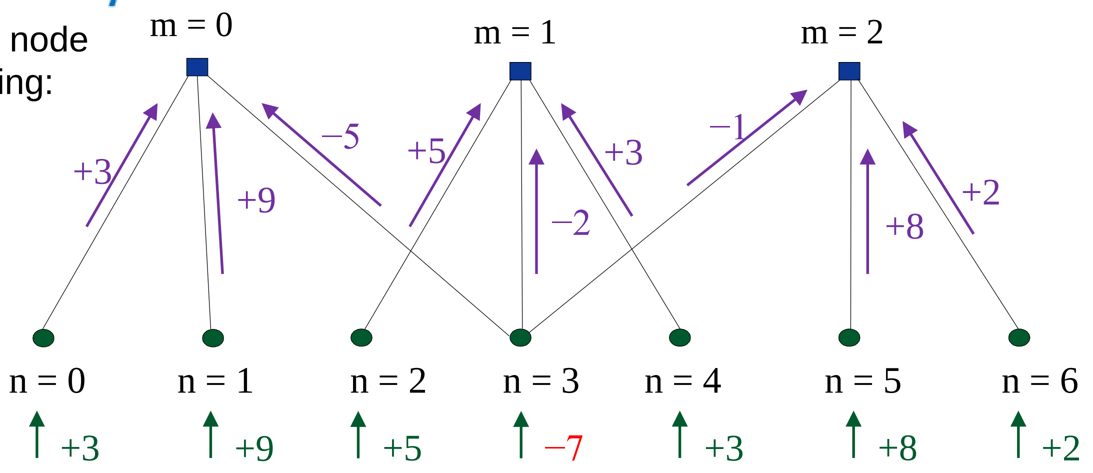


Check node processing:

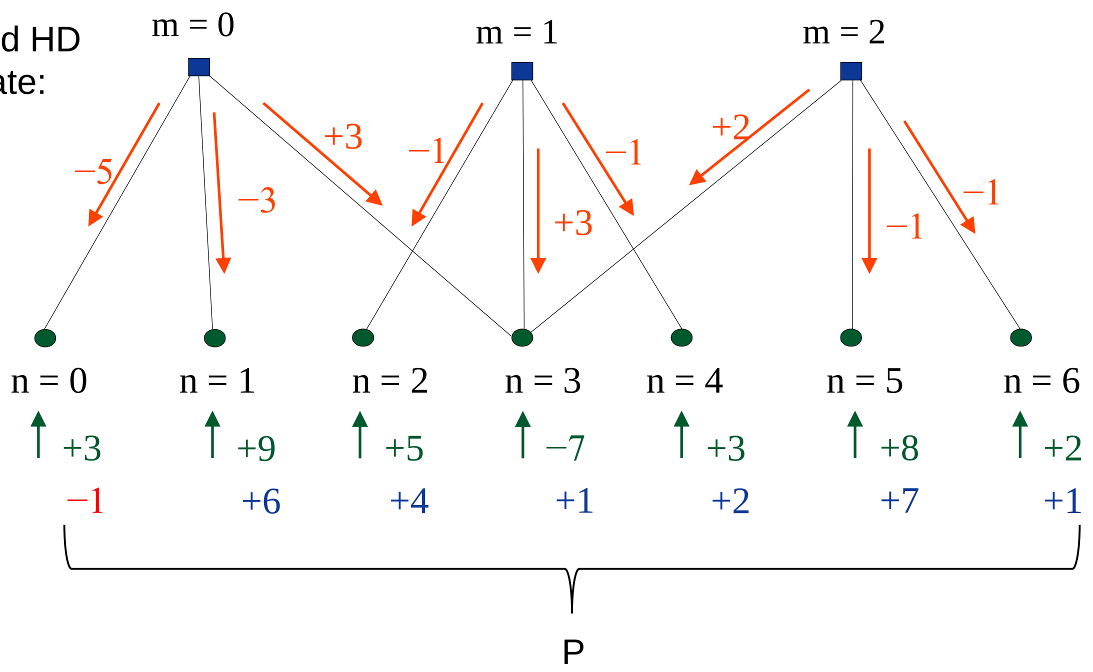


LDPC Decoding Example, 2/3

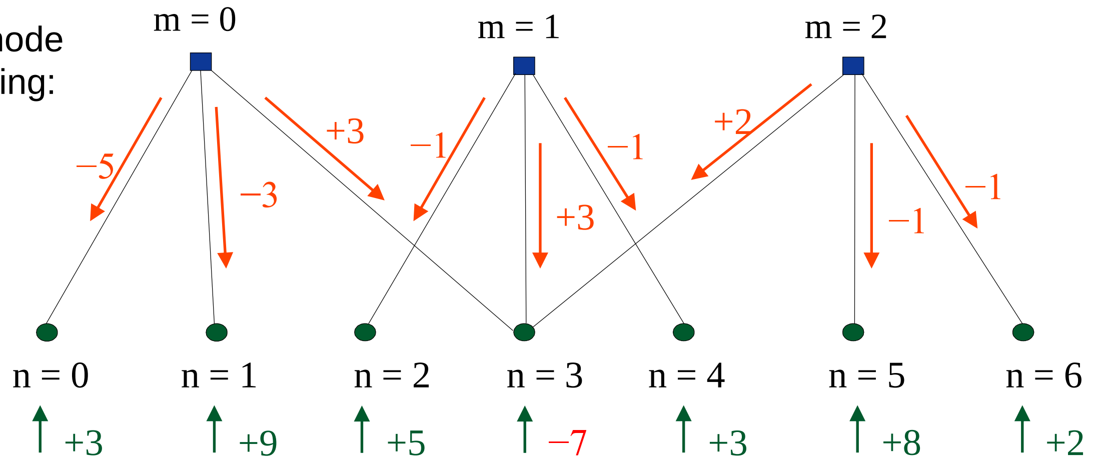
Variable node processing:



P and HD update:

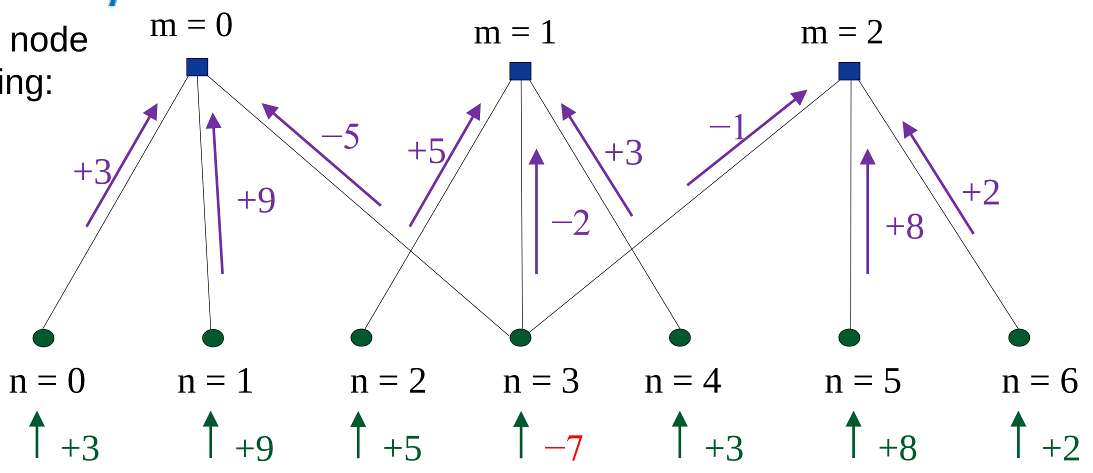


Check node processing:

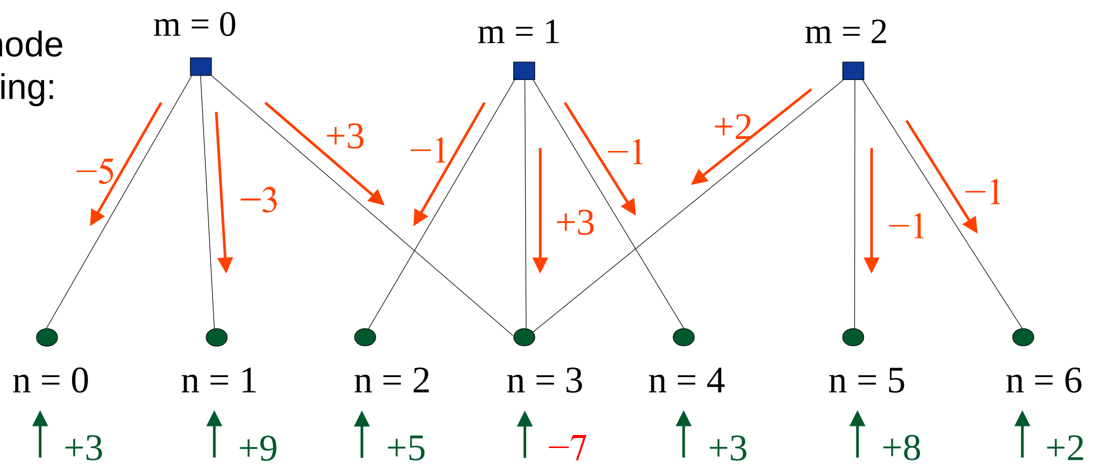


LDPC Decoding Example, 2/3

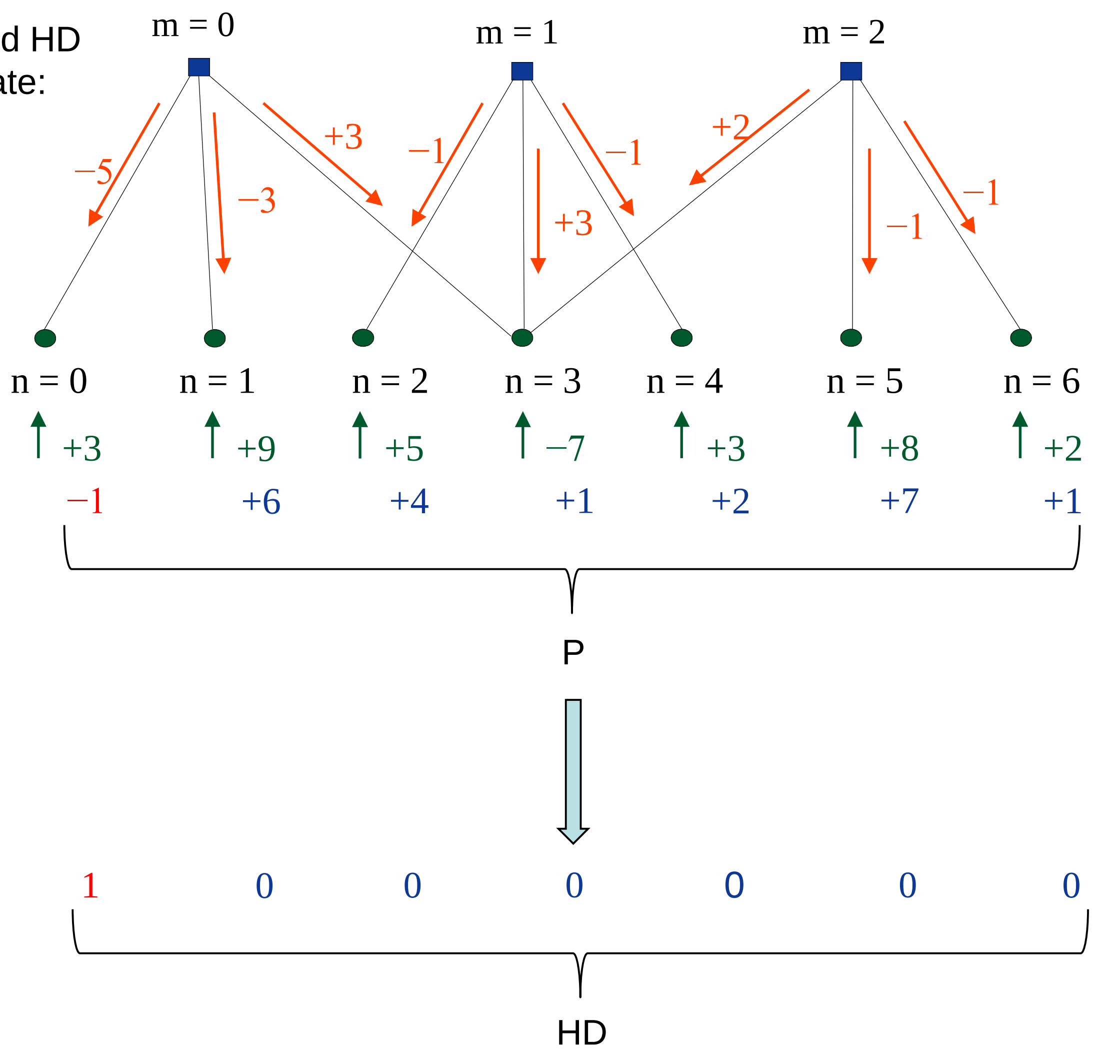
Variable node processing:



Check node processing:

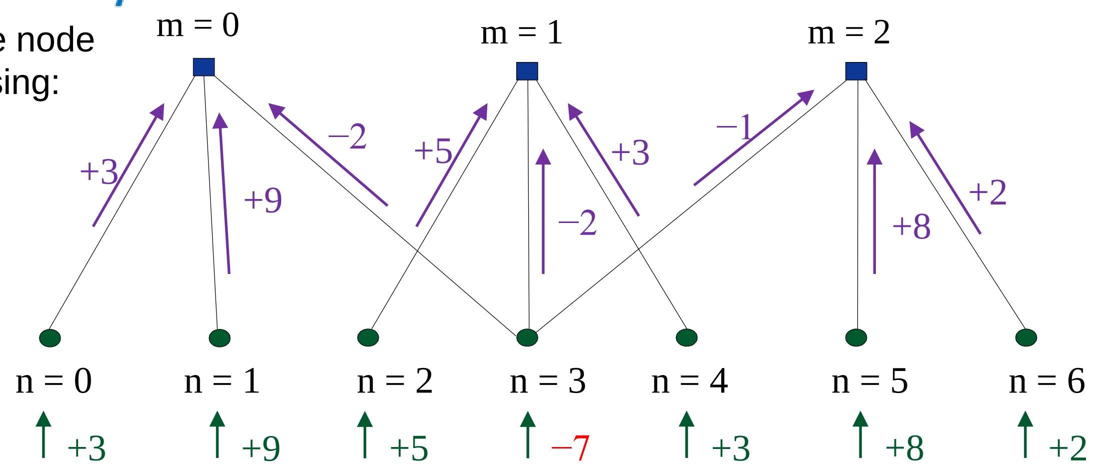


P and HD update:



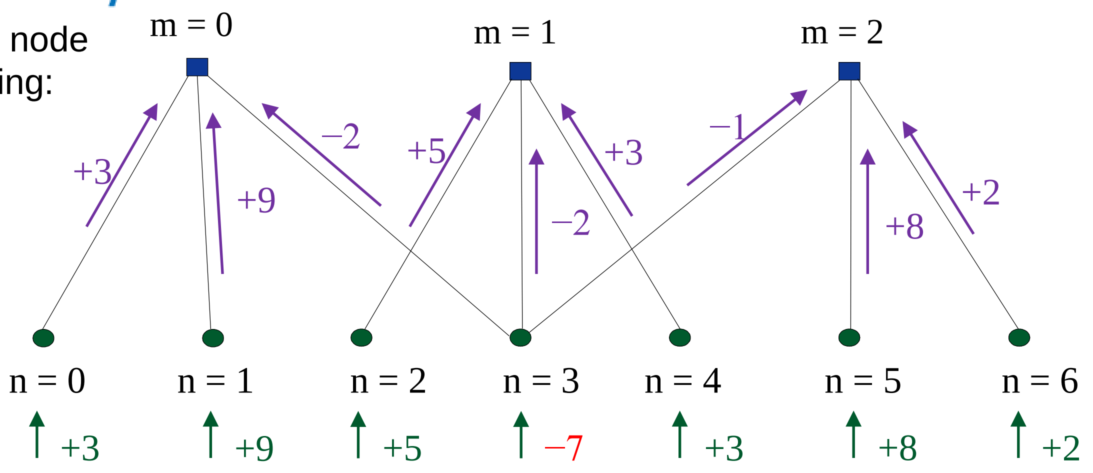
LDPC Decoding Example, 3/3

Variable node processing:

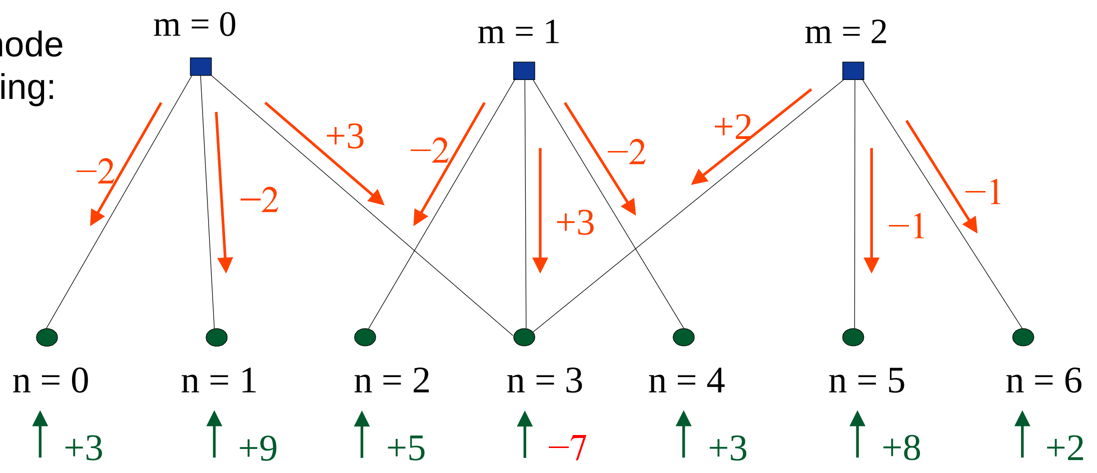


LDPC Decoding Example, 3/3

Variable node processing:

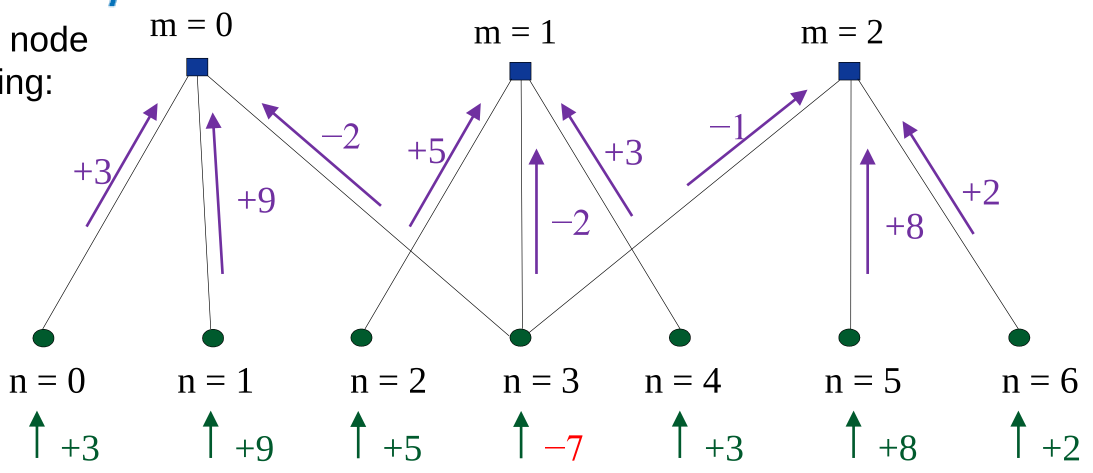


Check node processing:

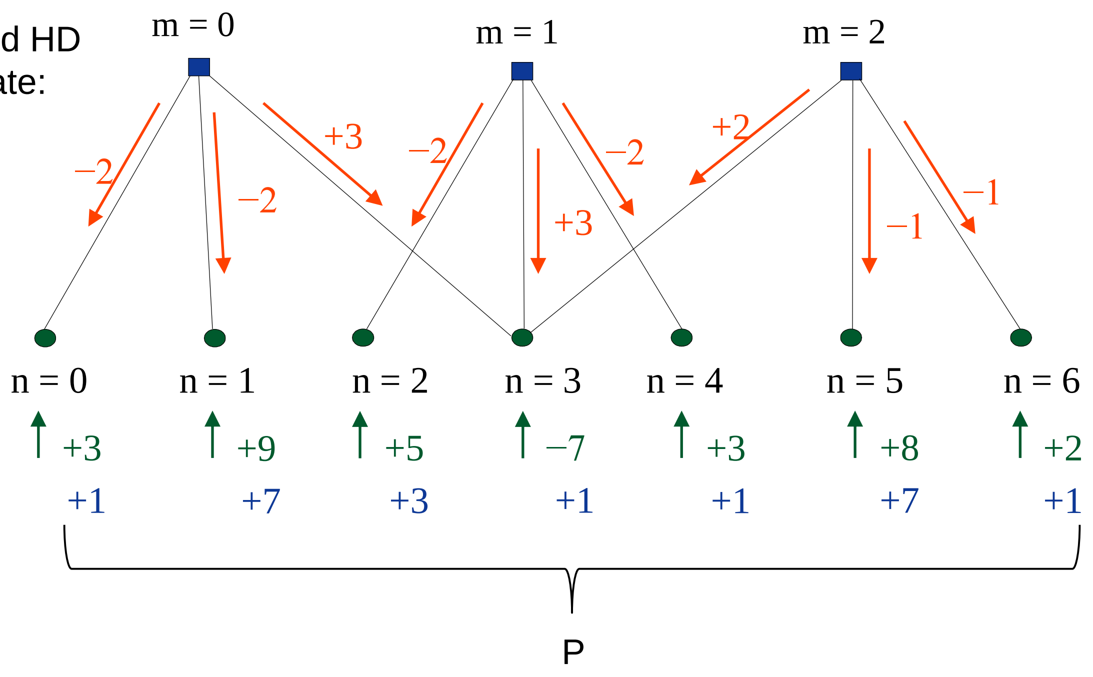


LDPC Decoding Example, 3/3

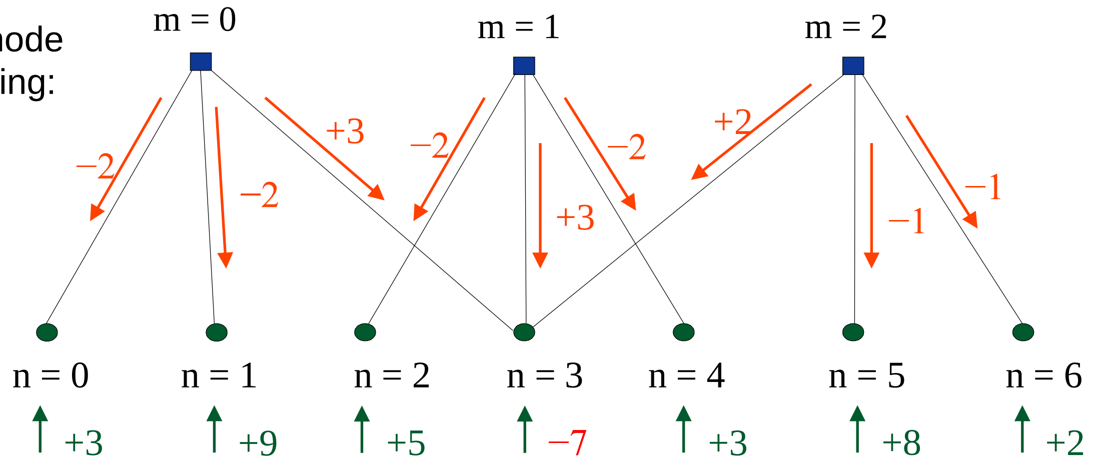
Variable node processing:



P and HD update:

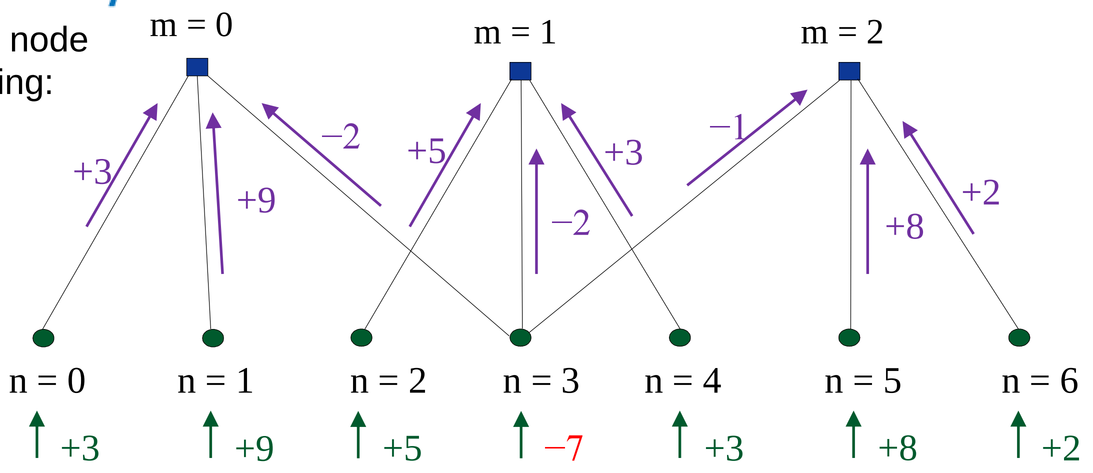


Check node processing:

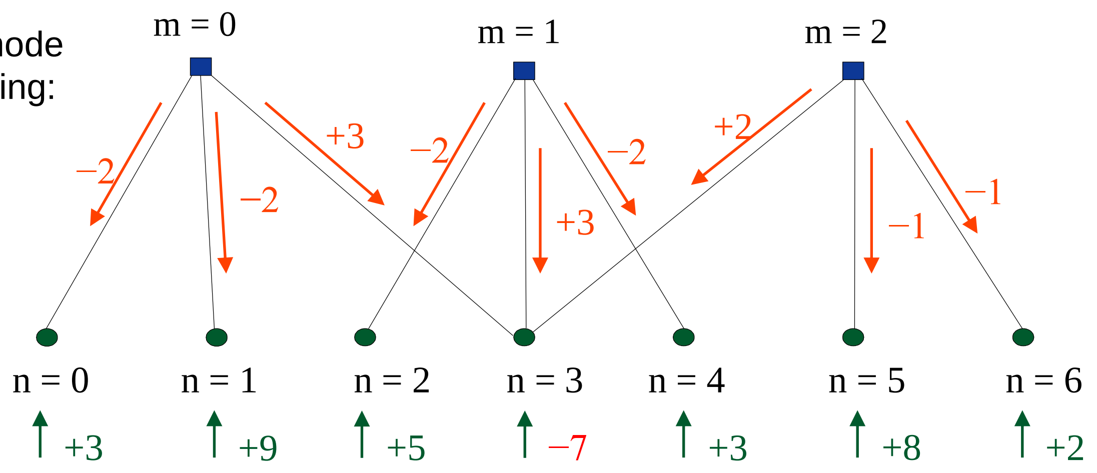


LDPC Decoding Example, 3/3

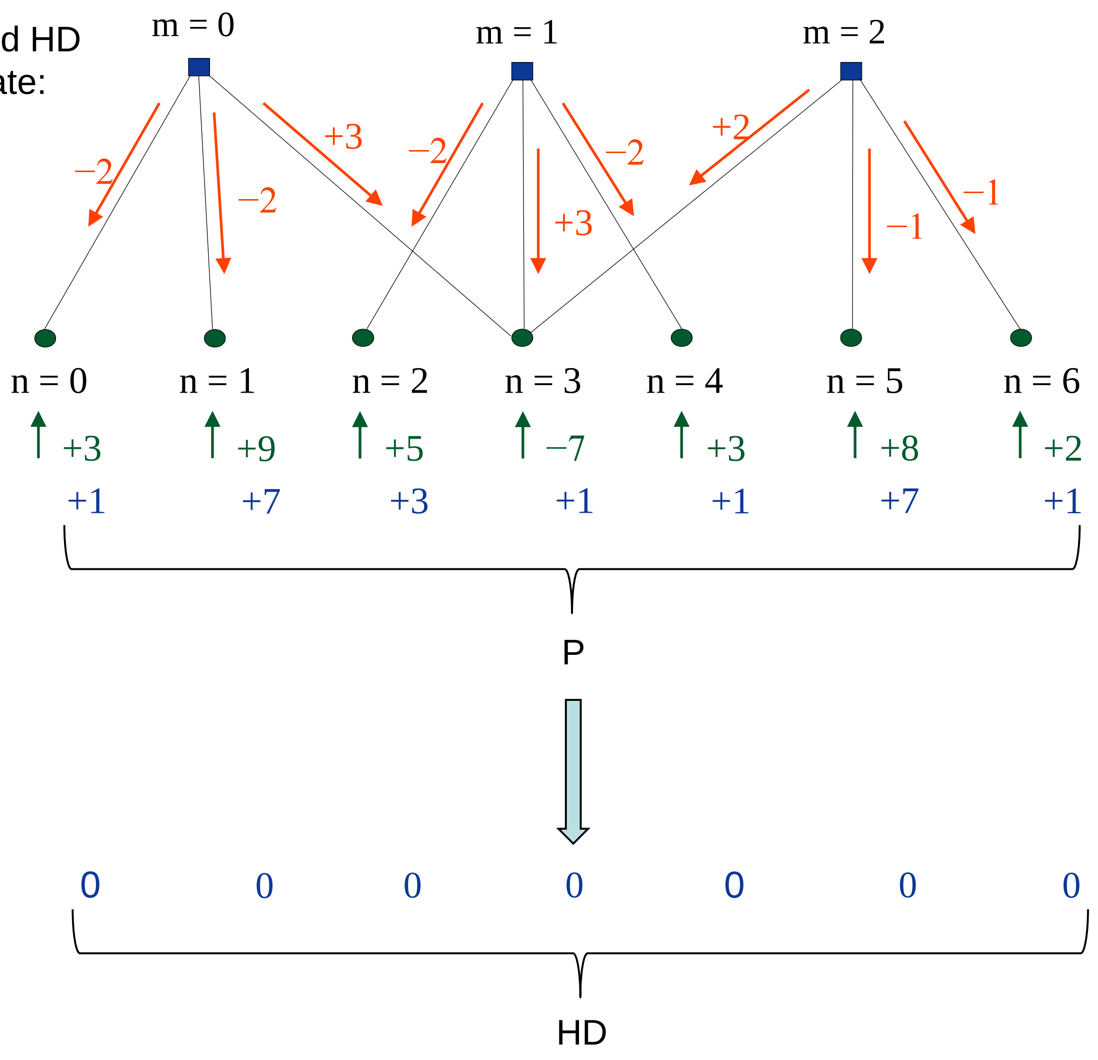
Variable node processing:



Check node processing:



P and HD update:



Overview

- Why Is Error Correction Needed in Flash Memories?
- Error Correction Codes Fundamentals
- Low-Density Parity-Check (LDPC) Codes
- LDPC Encoding and Decoding Methods
- Decoder Architectures for LDPC Codes

Decoder Architectures

- Parallelization is good-but comes at a steep cost for LDPC decoders
- Fully Parallel Architecture:

  - All the check updates in one clock cycle and all the bit updates in one more clock cycle

  - Huge Hardware resources and routing congestion
- Serial Architecture:

  - Check updates and bit updates in a serial fashion

  - Huge memory requirement. Memory in critical path

  - Very low throughput

Semi-parallel Architectures

- Check updates and bit updates using several units.
- Partitioned memory by imposing structure on H matrix.
- Practical solution for most of the applications.
- There are several semi-parallel architectures proposed.
- Complexity differs based on architecture and scheduling.

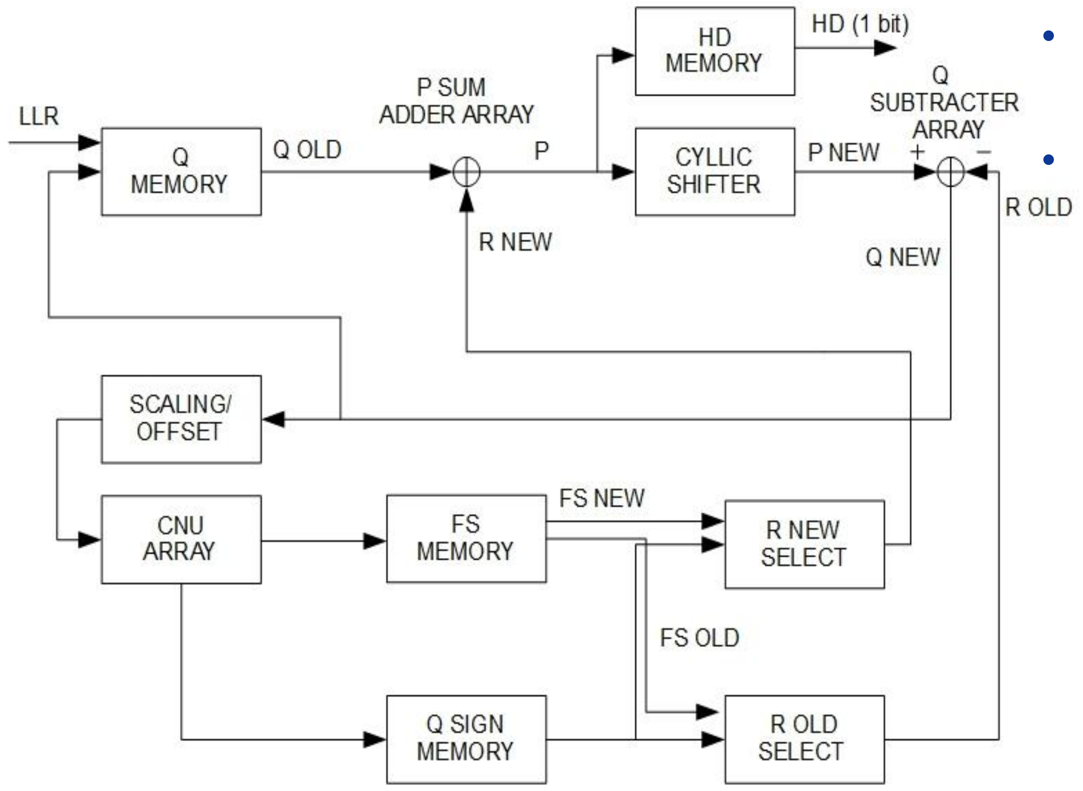
Layered Decoder Architecture

- Optimized Layered Decoding with algorithm transformations for reduced memory and computations

```
$R_{l,n}{}^{(0)} = 0, P_n = L_n{}^{(0)}$                [ Initialization for each new received data frame ]
$\forall i = 1,2,\ldots,it_{max}$                        [ Iteration loop ]
    $\forall l = 1,2,\ldots,j$                           [ Layer loop ]
        $\forall n = 1,2,\ldots,k$                       [ Block column loop ]
            $R_{l,n}{}^{(i)} = f\left(\left[Q_{l,n'}{}^{(i)}\right]^{S(l,n')}\right), \forall n' = 1,2,\ldots,d_{c_l} - 1$
                $(R_{\text{new}} = f(Q_{\text{new}}) = \text{R\_Select}(\text{FS}, \text{Qsign}))$
            $[P_n]^{S(l,n)} = \left[Q_{l,n}{}^{(i)}\right]^{S(l,n)} + R_{l,n}{}^{(i)}$ ,   $(P = Q_{\text{old}} + R_{\text{new}})$
                $P_{\text{new}}$ is then computed by applying delta shift on $P$
            $\left[Q_{l,n}{}^{(i)}\right]^{S(l,n)} = [P_n]^{S(l,n)} - R_{l,n}{}^{(i-1)}$, $(Q_{\text{new}} = P_{\text{new}} - R_{\text{old}})$
```

- Q and R messages are computed for each $p \times p$ block of H where p is the parallelization
- $f(\cdot)$ is the check node processing unit
- $S(l, n')$ is the upward (right) shift for block row (layer) l and block column n'
- d_{c_l} is the degree of layer l

Block Serial Layered Decoder Architecture with On-the-Fly Computation

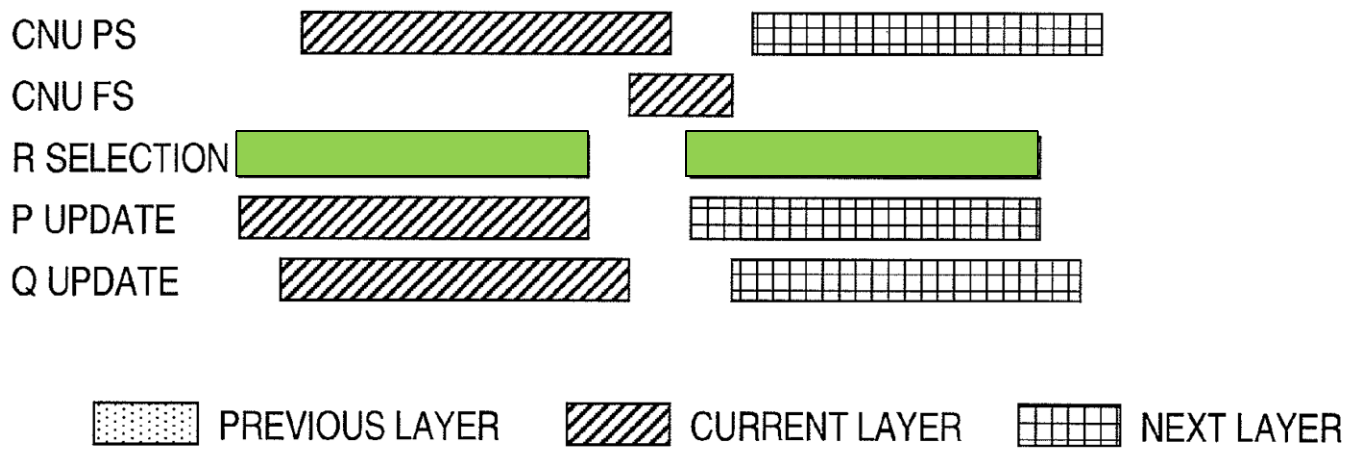


See [8, P1-P6] and references therein for more details on features and implementation.

- Proposed for irregular H matrices
- Goal: minimize memory and re-computations by employing just in-time scheduling
- Advantages compared to other architectures:
  1) Q memory (or L/P/Q memory) can be used to store L/Q/P instead of 3 separate memories- memory is managed at circulant level as at any time for a given circulant we need only L or Q or P.
  2) Only one shifter.
  3) Value-reuse is effectively used for both Rnew and Rold
  4) Low complexity data path design-with no redundant data path operations.
  5) Low complexity CNU design.
  6) Out-of-order processing at both layer and circulant level for all the processing steps such as Rnew and PS processing to eliminate the pipeline and memory access stall cycles.

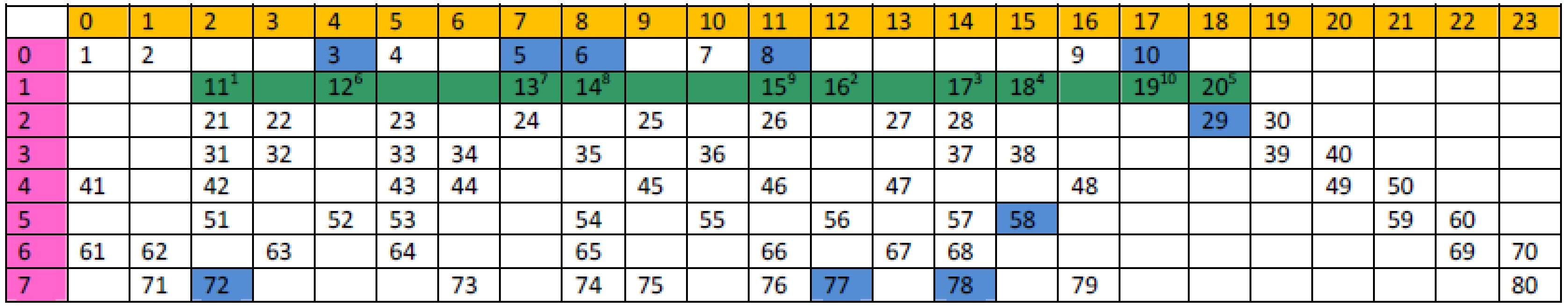
Data Flow Diagram

FIG. 6B



R Selection for R NEW operates out of order to feed the data for PS processing of the next layer.

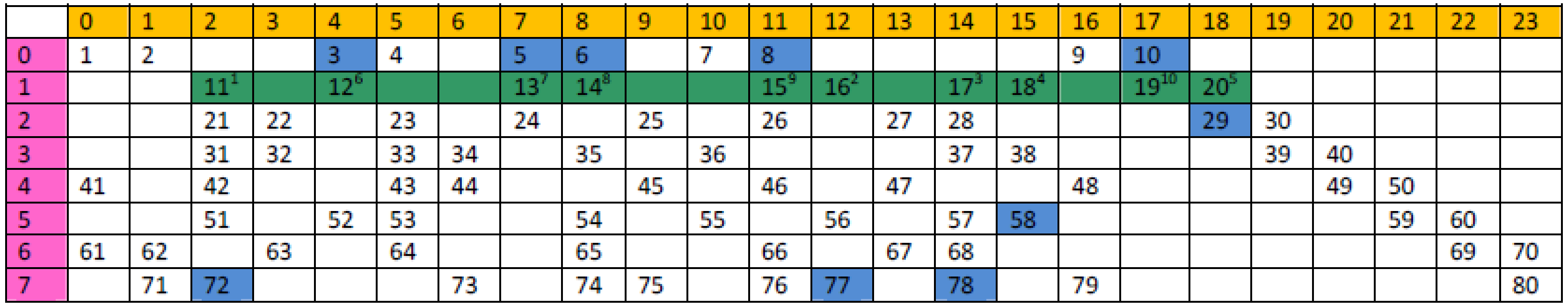
Illustration for out-of-order processing



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	2			3	4		5	6		7	8					9	10						
1			11^{1}		12^{6}			13^{7}	14^{8}			15^{9}	16^{2}		17^{3}	18^{4}		19^{10}	20^{5}					
2			21	22		23		24		25		26		27	28				29	30				
3			31	32		33	34		35		36				37	38				39	40			
4	41		42			43	44			45		46		47			48				49	50		
5			51		52	53			54		55		56		57	58						59	60	
6	61	62		63		64			65			66		67	68								69	70
7		71	72				73		74	75		76	77		78		79							80

- Rate 2/3 code. 8 Layers, 24 block columns. dv, column weight varies from 2 to 6. dc, row weight is 10 for all the layers.
- Non-zero circulants are numbered from 1 to 80. No layer re-ordering in processing. Out-of-order processing for Rnew. Out-of-order processing for Partial state processing.
- **Illustration for 2nd iteration with focus on PS processing of 2nd layer.**
- Rold processing is based on the circulant order 11 16 17 18 20 12 13 14 15 19 and is indicated in green.
- Rnew is based on the circulant order 72 77 78 58 29 3 5 6 8 10 and is indicated in blue.
- Q memory, HD memory access addresses are based on the block column index to which the green circulants are connected to.
- Q sign memory access address is based on green circulant number.
- Superscript indicates the clock cycle number counted from 1 at the beginning of layer 2 processing.

Out-of-order layer processing for R Selection



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	2			3	4		5	6		7	8					9	10						
1			11^{1}		12^{6}			13^{7}	14^{8}			15^{9}	16^{2}		17^{3}	18^{4}		19^{10}	20^{5}					
2			21	22		23		24		25		26		27	28				29	30				
3			31	32		33	34		35		36				37	38				39	40			
4	41		42			43	44			45		46		47			48				49	50		
5			51		52	53			54		55		56		57	58						59	60	
6	61	62		63		64			65			66		67	68								69	70
7		71	72				73		74	75		76	77		78		79							80

- Normal practice is to compute Rnew messages for each layer after CNU PS processing.
- Here the execution of R new messages of each layer is decoupled from the execution of corresponding layer's CNU PS processing. Rather than simply generating Rnew messages per layer, they are computed on basis of circulant dependencies.
- R selection is out-of-order so that it can feed the data required for the PS processing of the second layer. For instance Rnew messages for circulant 29 which belong to layer 3 are not generated immediately after layer 3 CNU PS processing .
- Rather, Rnew for circulant 29 is computed when PS processing of circulant 20 is done as circulant 29 is a dependent circulant of circulant of 20.
- Similarly, Rnew for circulant 72 is computed when PS processing of circulant 11 is done as circulant 72 is a dependent circulant of circulant of 11.
- Here the instruction/computation is computed at precise moment when the result is needed!!!

Out-of-order block processing for Partial State

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	2			3	4		5	6		7	8					9	10						
1			11^{1}		12^{6}			13^{7}	14^{8}			15^{9}	16^{2}		17^{3}	18^{4}		19^{10}	20^{5}					
2			21	22		23		24		25		26		27	28				29	30				
3			31	32		33	34		35		36				37	38				39	40			
4	41		42			43	44			45		46		47			48				49	50		
5			51		52	53			54		55		56		57	58						59	60	
6	61	62		63		64			65			66		67	68								69	70
7		71	72				73		74	75		76	77		78		79							80

- Re-ordering of block processing . While processing layer 2, the blocks which depend on layer 1 will be processed last to allow for the pipeline latency.
- In the above example, the pipeline latency can be 5.
- The vector pipeline depth is 5. So no stall cycles are needed while processing the layer 2 due to the pipelining. In other implementations, the stall cycles are introduced – which will effectively reduce the throughput by a huge margin.
- The operations in one layer are sequenced such that the block that has dependent data available for the longest time is processed first.

Memory organization

- Q memory width is equal to circulant size * 8 bits and depth is number of block columns.
- HD memory width is equal to circulant size * 1 bits and depth is number of block columns.
- Qsign memory width is equal to circulant size * 1 bits and depth is number of non-zero circulants in H-matrix.
- FS memory width is equal to circulant size * (15 bits (= 4 bits for Min1 + 4 bits for Min2 index + 1 bit + 6 bits for Min1 index).
- FS memory access is expensive and number of accesses can be reduced with scheduling.
- For the case of decoder for regular mother matrices (no 0 blocks and no OOP): FS access is needed one time for Rold for each layer; is needed one time for R new for each layer.
- For the case of decoder for irregular mother matrices: FS access is needed one time for Rold for each layer; is needed one time for R new for each non-zero circulant in each layer.

From Throughput Requirements to Design Specification

- Requirements
  - Bit Error Rate (BER)
  - Latency
  - Throughput (bits per sec)
- BER would dictate number of iterations and degree profile (check node degrees and variable node degrees).
- Latency determined by decoder processing times
- Throughput = Number of bits processed per clock * clock frequency

  Suppose: Number of block columns = Nb

  Circulant Size = Sc

  Average Variable Node Degree = AVND
- Throughput = (Nb * Sc) / (Nb * AVND * Iterations) * clock frequency

  Sc is usually set to less than 128 for smaller router.

Flash Memory Summit

References

1. Y. Cai, E. F. Haratsch, *et al.*, “Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling,” *Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium*, 2013.
2. R. Tanner, “A recursive approach to low-complexity codes,” *IEEE Trans. on info. Theory,* 27.5, pp. 533-547, 1981.
3. R. G. Gallager, “Low density parity check codes,” *IRE Trans. Info. Theory, IT-8:21-28, Jan 1962.*
4. R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.
5. W. Ryan and S. Lin, *Channel codes: classical and modern*, Cambridge University Press, 2009.
6. Levine, *et. al.,* “Implementation of near Shannon limit error-correcting codes using reconfigurable hardware,” *IEEE Field-Programmable Custom Computing Machine,* 2000.
7. E. Yeo, “VLSI architectures for iterative decoders in magnetic recording channels,” *IEEE Trans. Magnetics*, vol. 37, no.2, pp. 748-55, March 2001.
8. K. K. Gunnam, “LDPC Decoding: VLSI Architectures and Implementations,” *Flash Memory Summit,* 2013*.*

Some of these slides are used from other references with permission.

Flash Memory Summit

References

Several features presented in this tutorial are covered by the following patents by Texas A&M University System (TAMUS):

[P1] U.S. Patent 8359522, Low density parity check decoder for regular LDPC codes.
[P2] U.S. Patent 8418023, Low density parity check decoder for irregular LDPC codes.
[P3] U.S. Patent 8555140, Low density parity check decoder for irregular LDPC codes.
[P4] U.S. Patent 8656250,Low density parity check decoder for regular LDPC codes.
[P5] U.S. Patent 9112530, Low density parity check decoder.
[P6] U.S. Patent 20150311917, Low density parity check decoder.

Some of these slides are used from other references with permission.