

Error correcting codes

II

Still not in the textbook :(

Hamming codes are alright

- We can do much better than Hamming codes
- Things are not quite so clear-cut when dealing with more awesome codes, though
 - Decoding suddenly becomes guesswork

Low-Density Parity Codes

- Very simple idea:
 - Encode (e.g.) 3 bits of data in 8 bits
 - Make sure that all 8 bits are necessary for determining the original 3 bits

3-bit code space

000
001
010
011
100
101
110
111

8-bit code space

00000000
00000001
...
00110011
...
01001111
...
01111100
...
10010100
...
10100111
...
11011011
...
11101000
...

3-bit code space

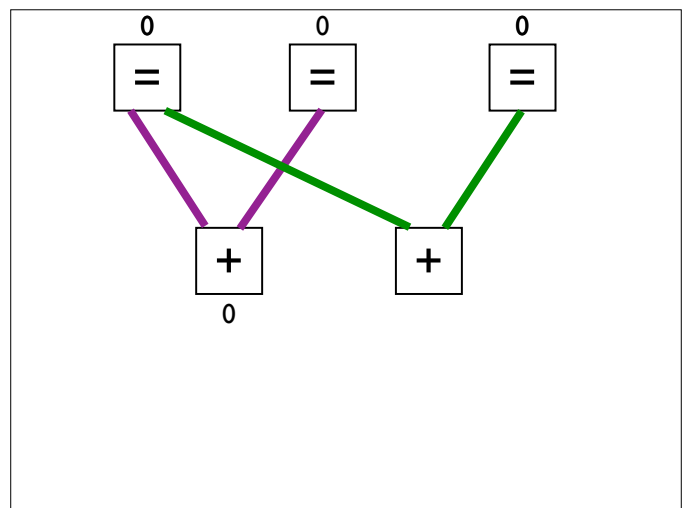
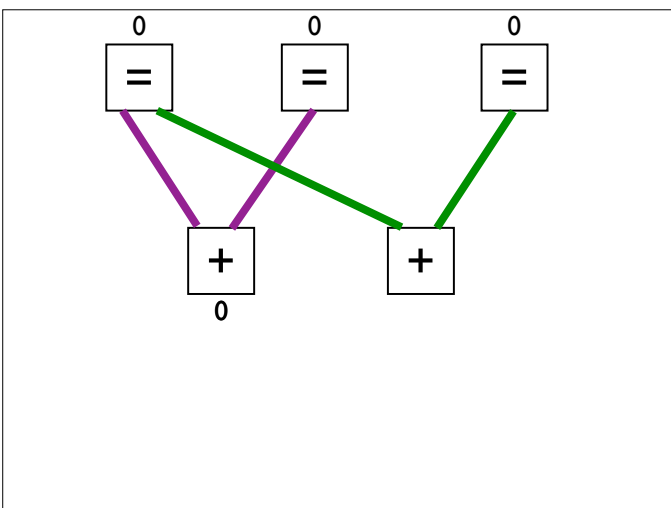
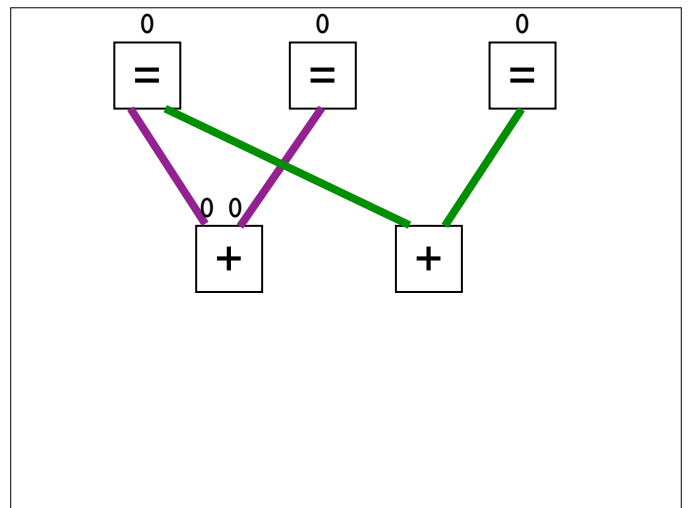
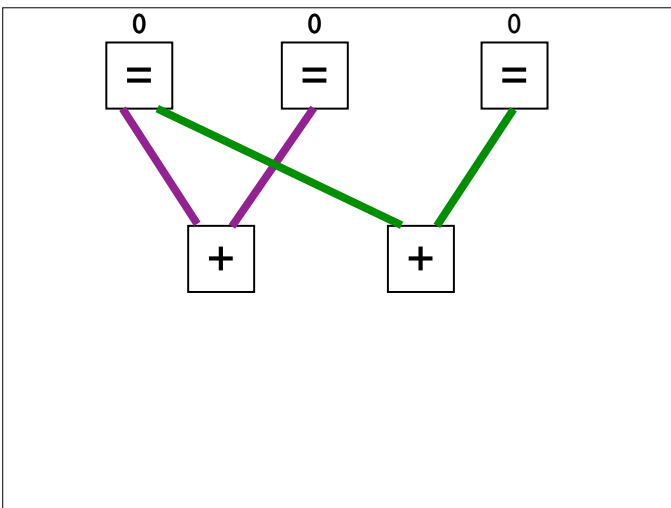
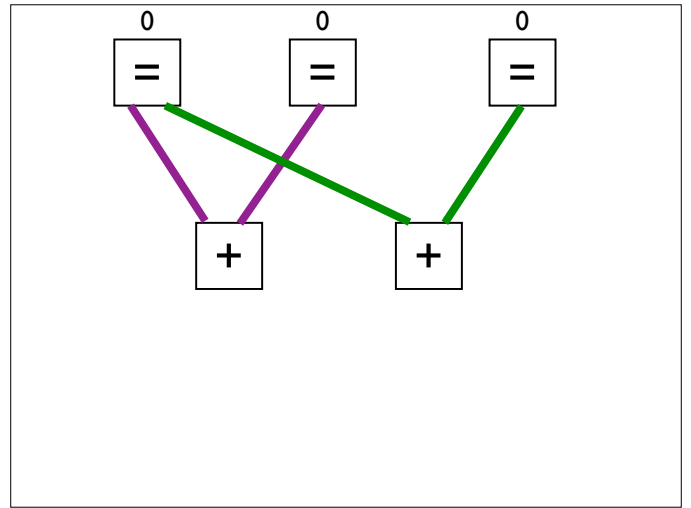
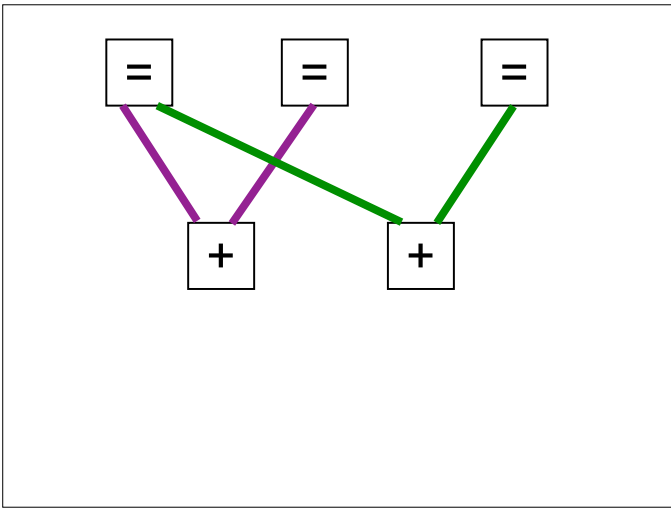
000
001
010
011
100
101
110
111

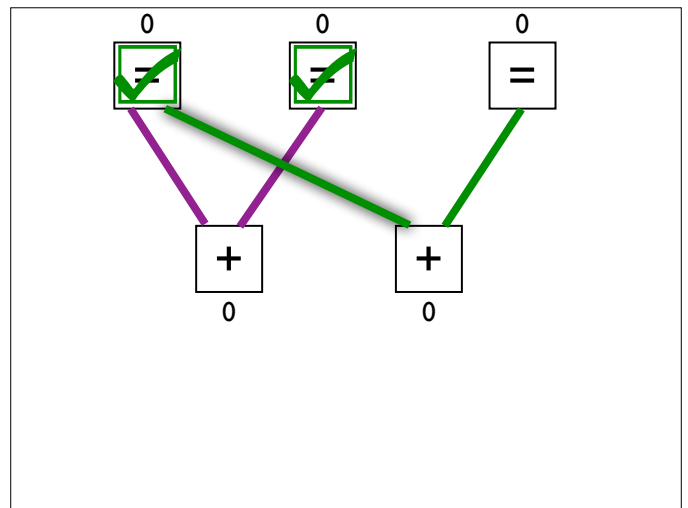
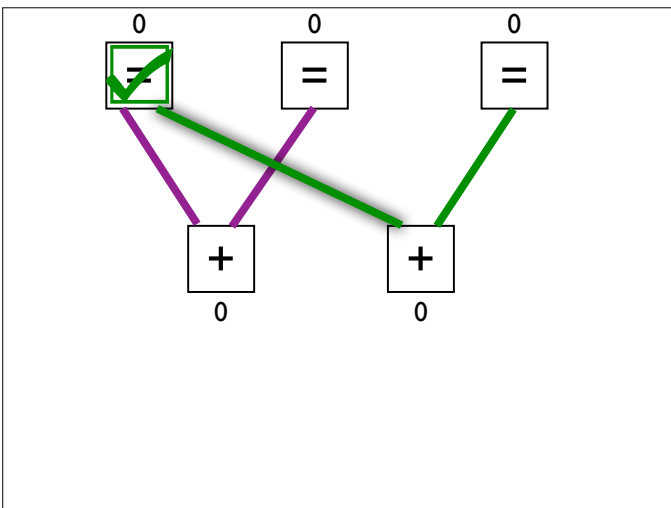
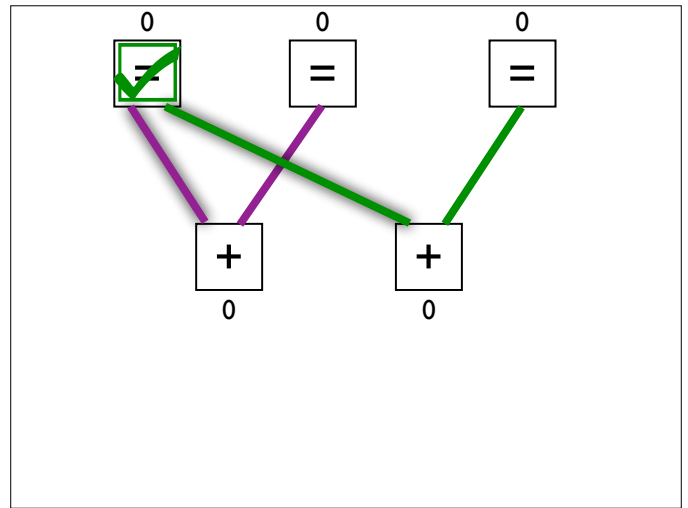
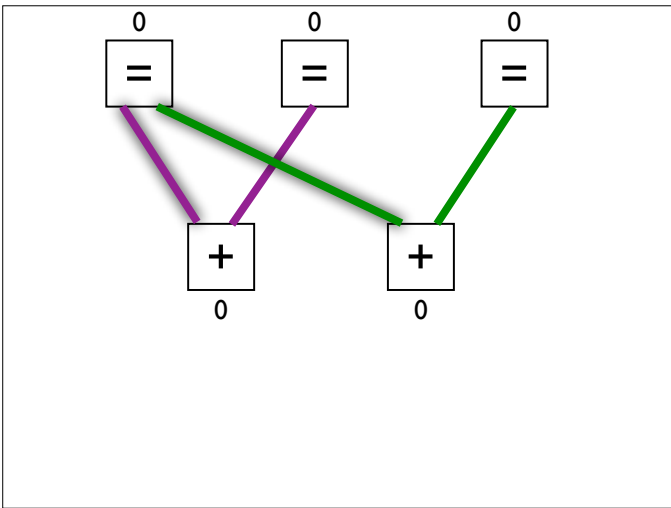
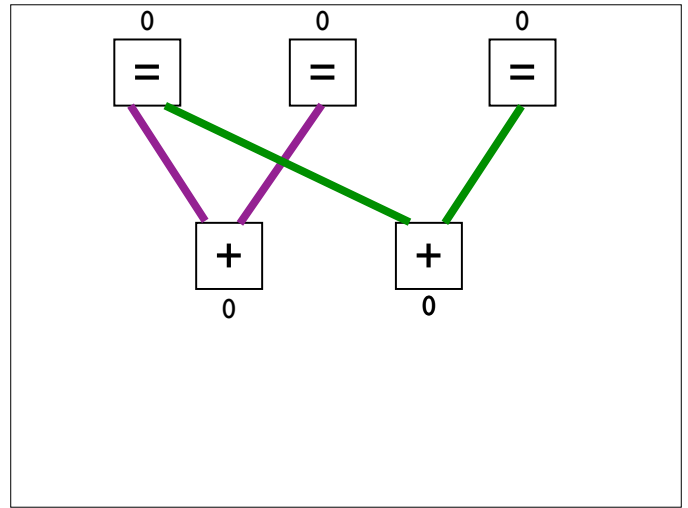
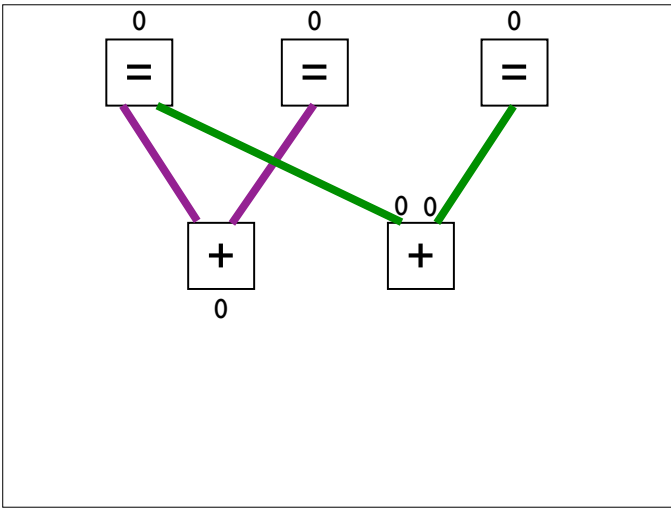
8-bit code space

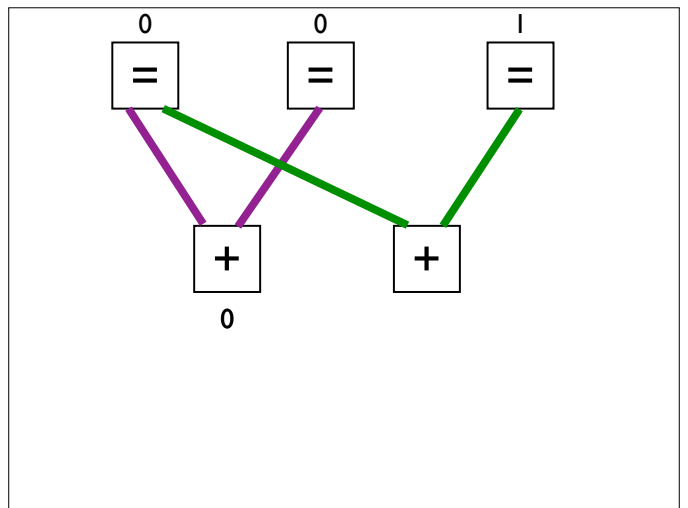
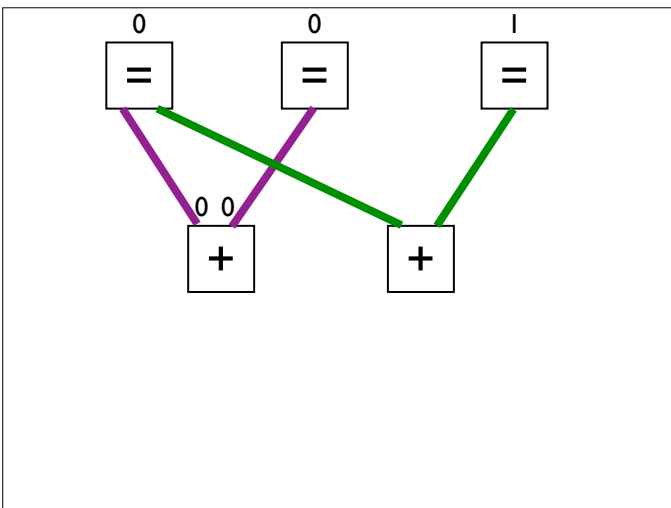
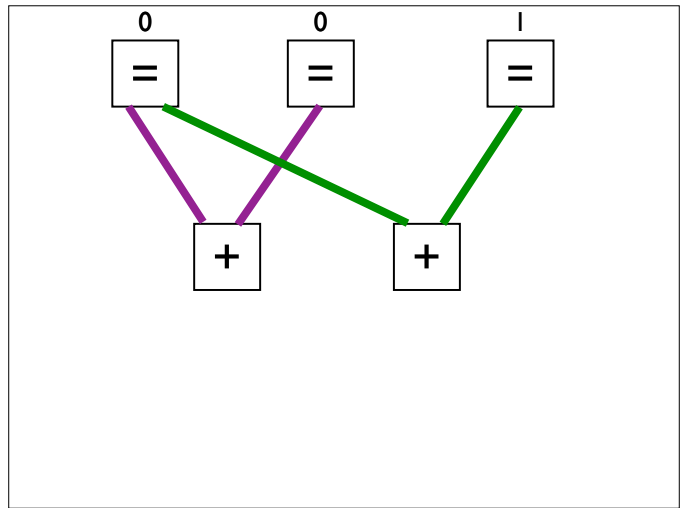
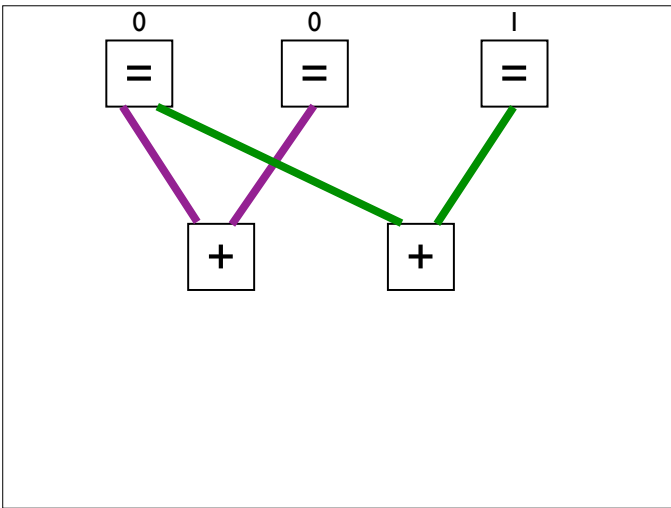
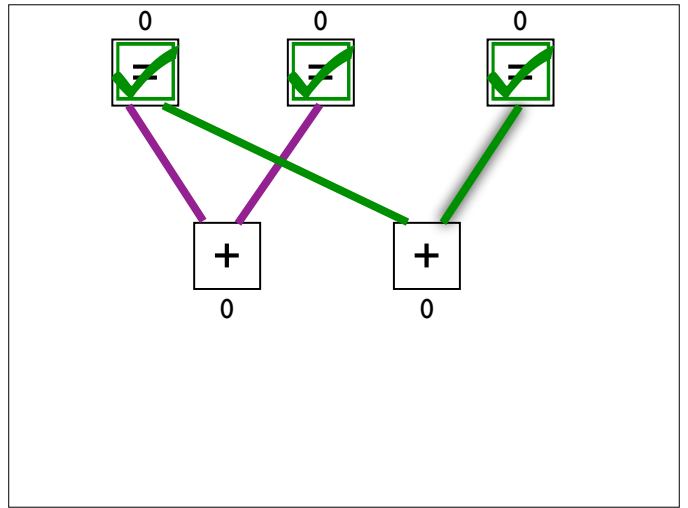
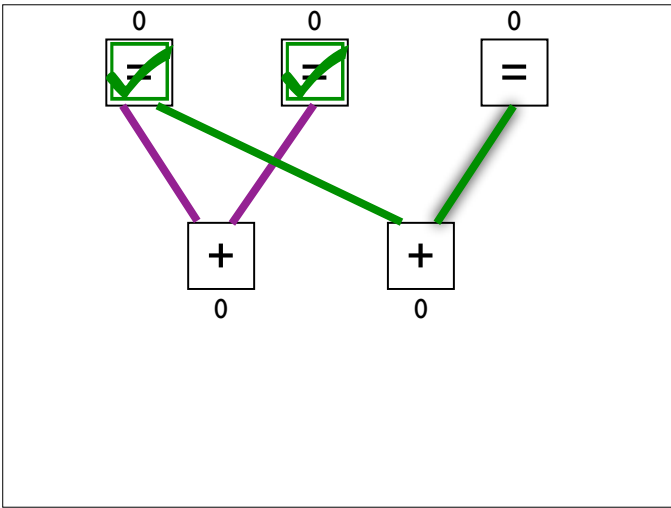
00000000
00000001
...
00110011
...
01001111
...
01111100
...
10010100
...
10100111
...
11011011
...
11101000
...

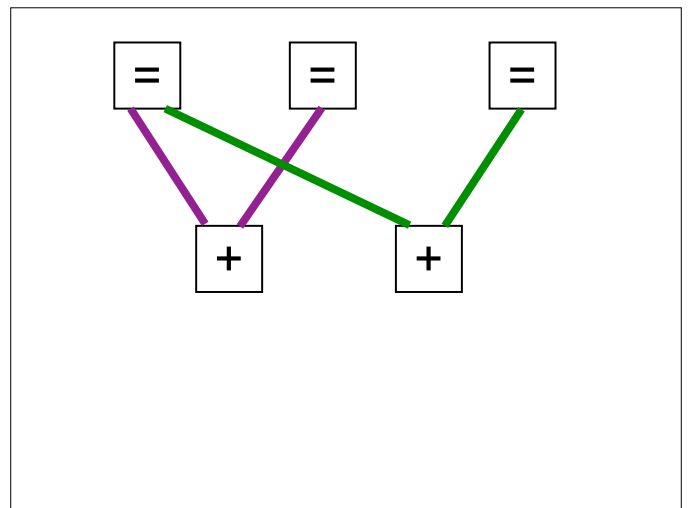
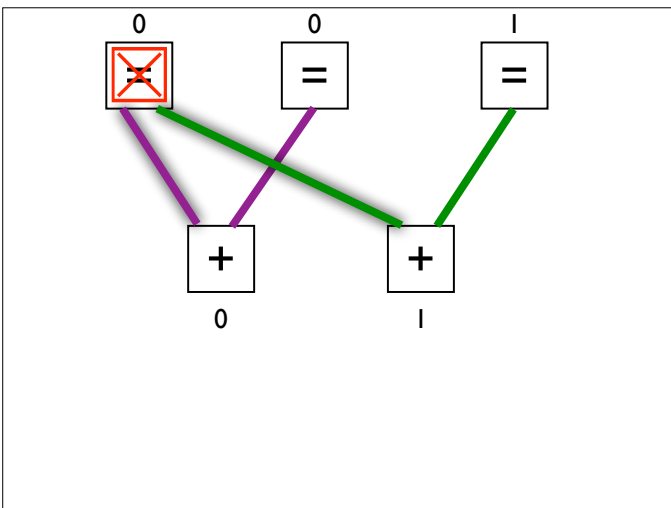
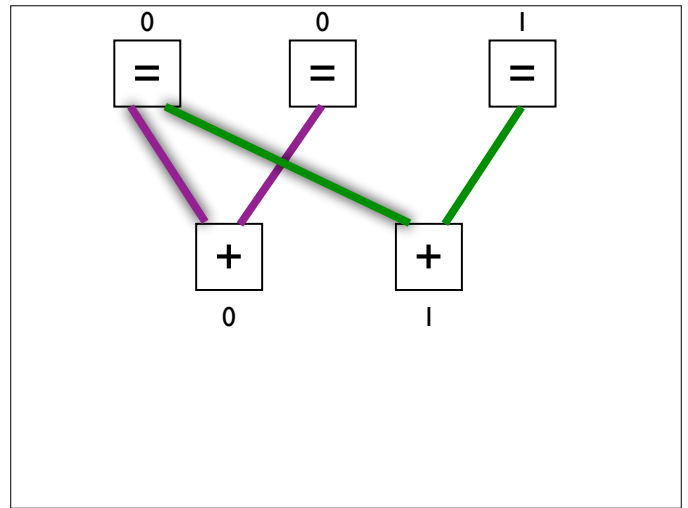
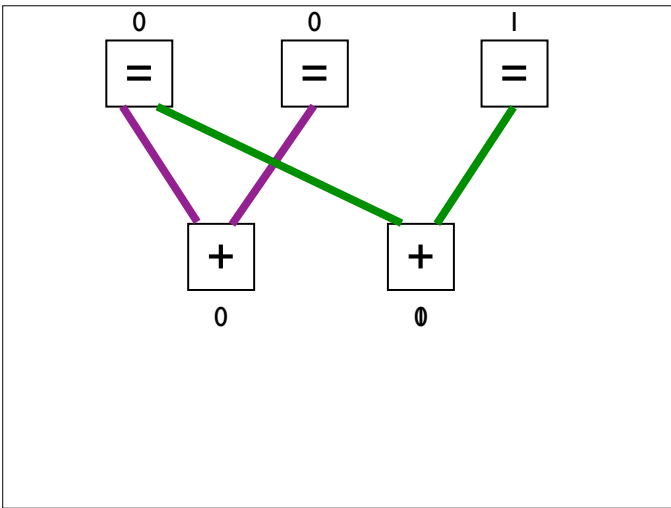
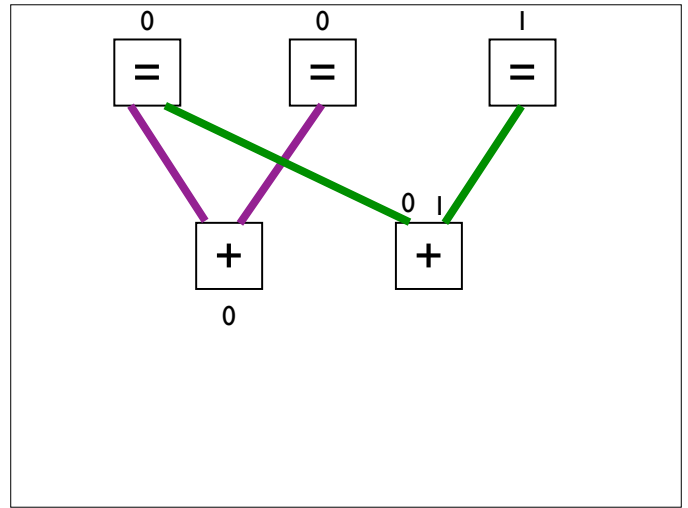
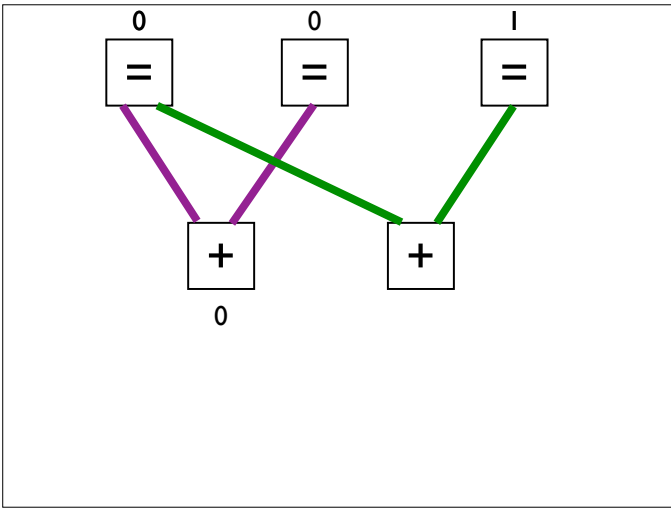
Low-Density Parity Codes

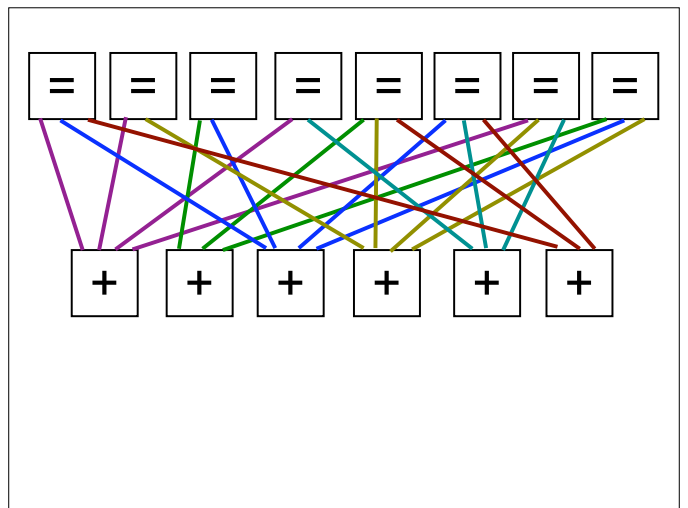
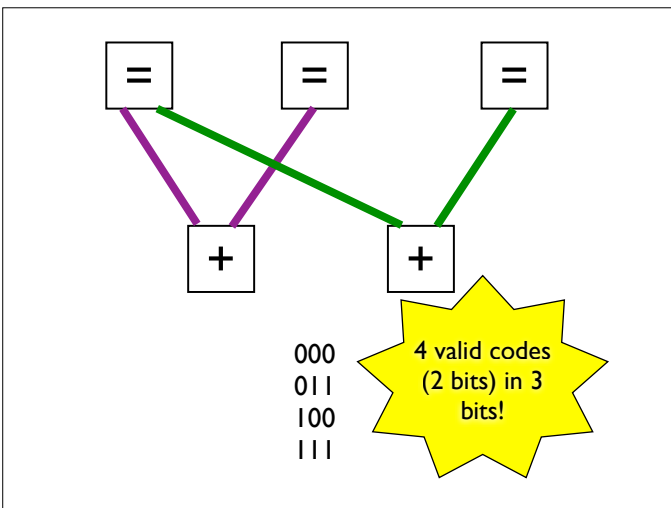
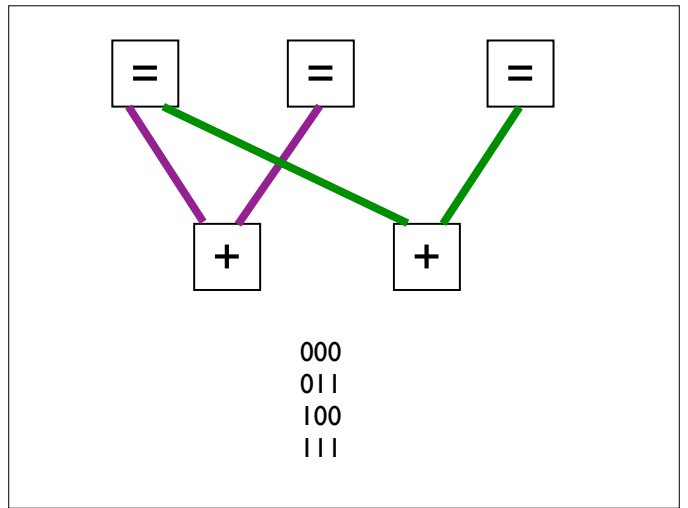
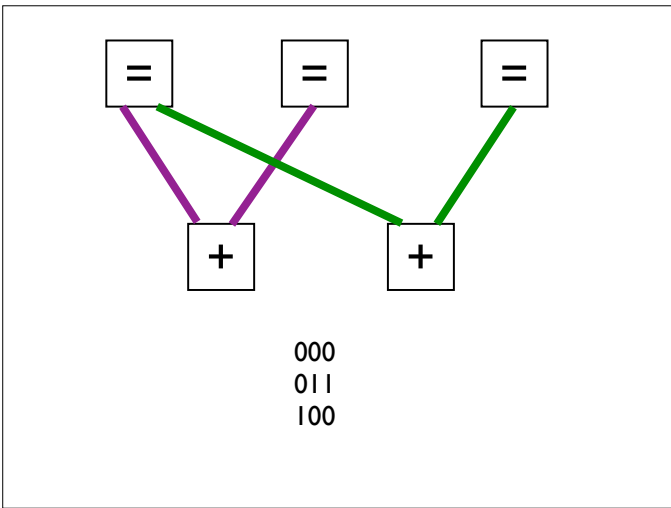
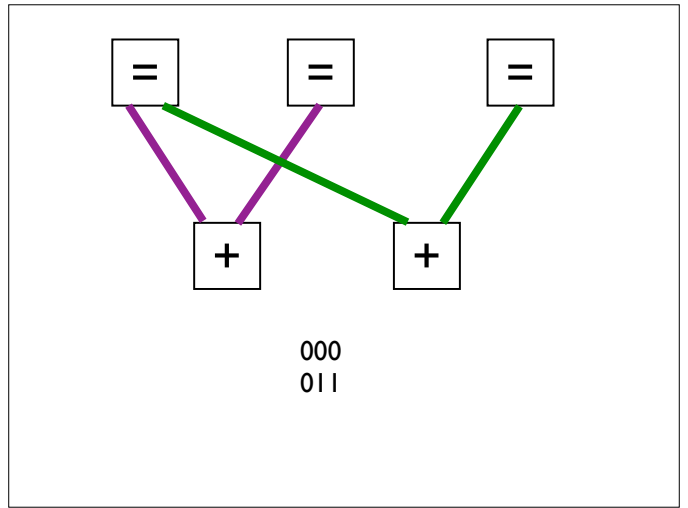
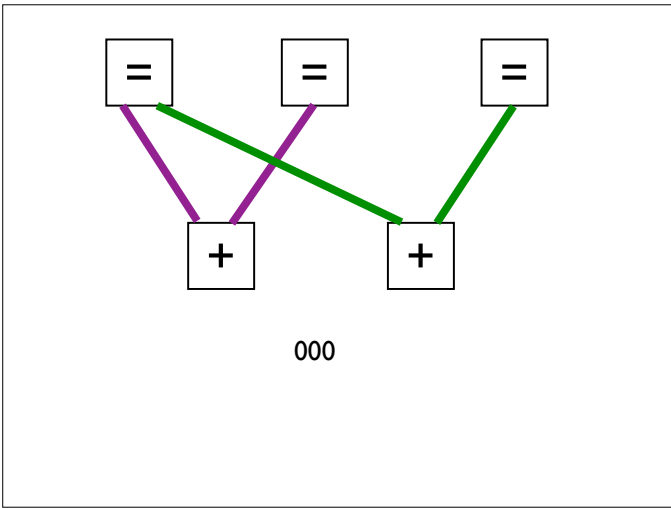
- There is no such thing as a “data bit” or a “parity bit”
- No bit is any more or less important than any other
- This scheme extends very well to *any* length of data and *any* amount of redundancy
 - None of this (7, 4) or (15, 11) nonsense

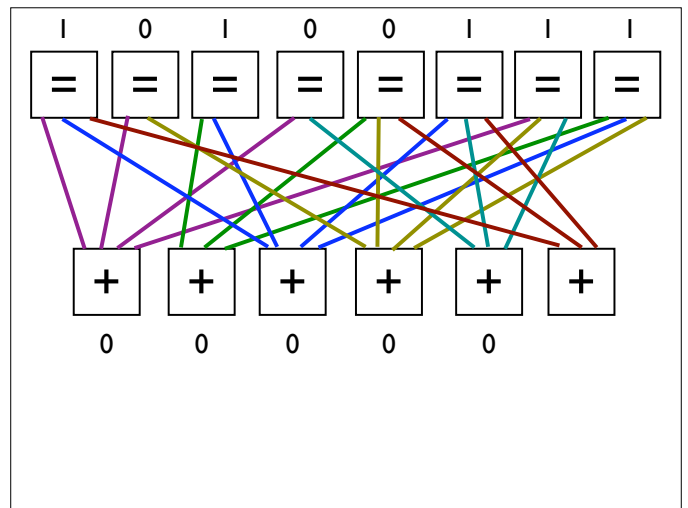
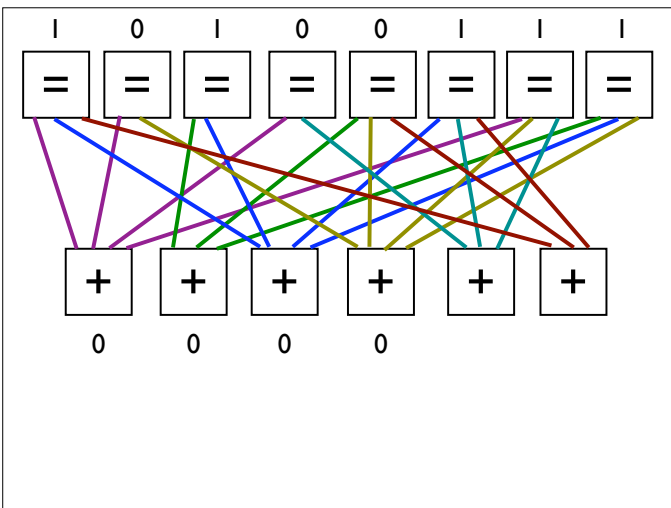
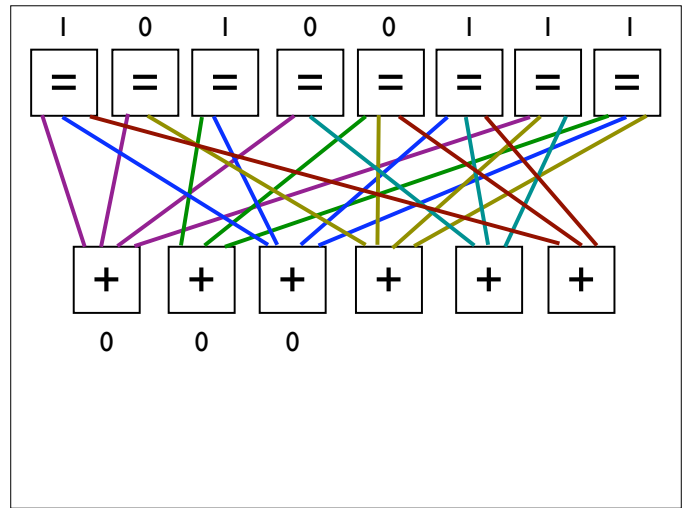
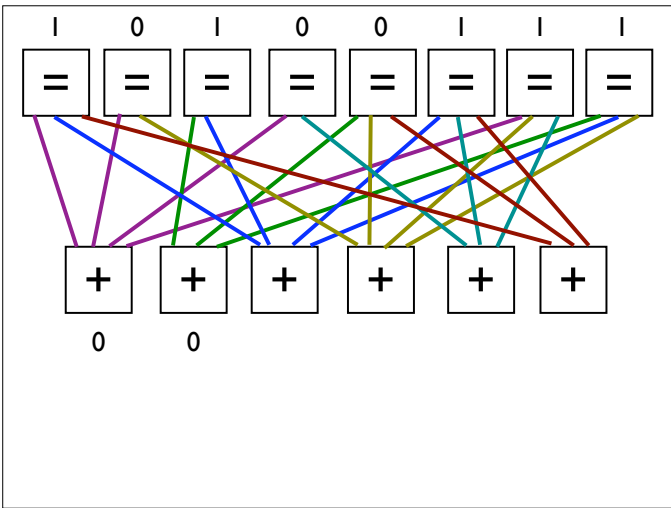
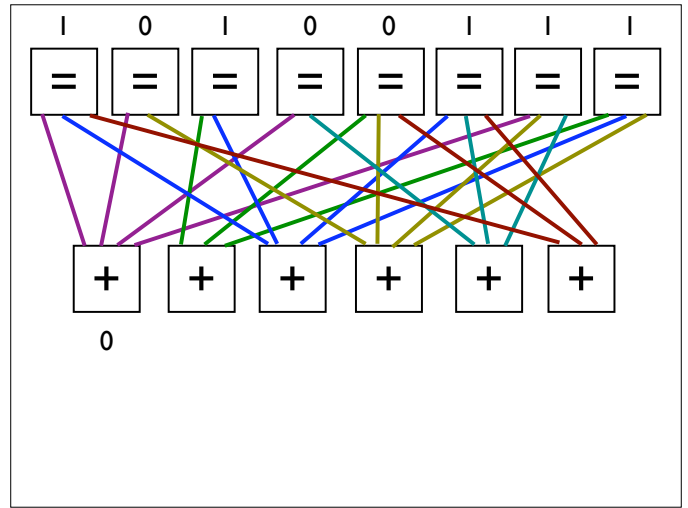
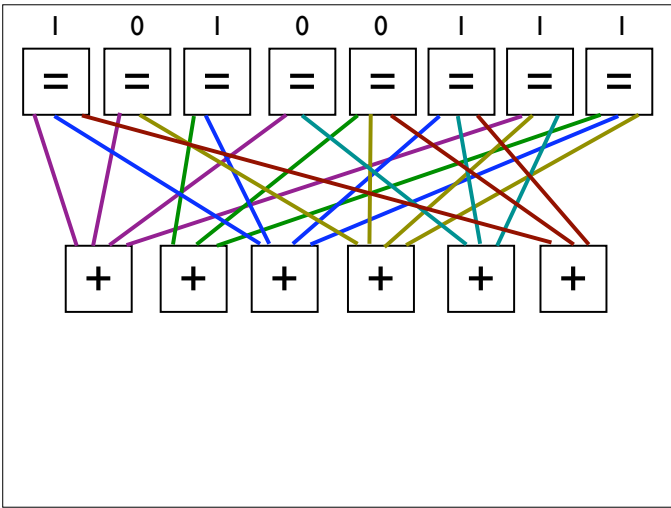


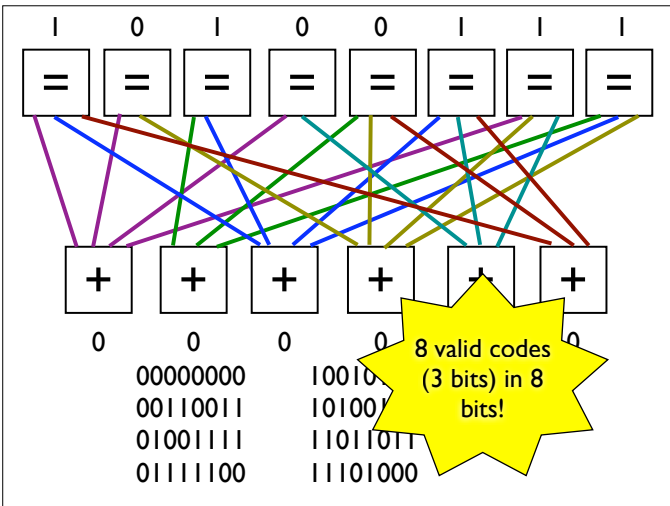
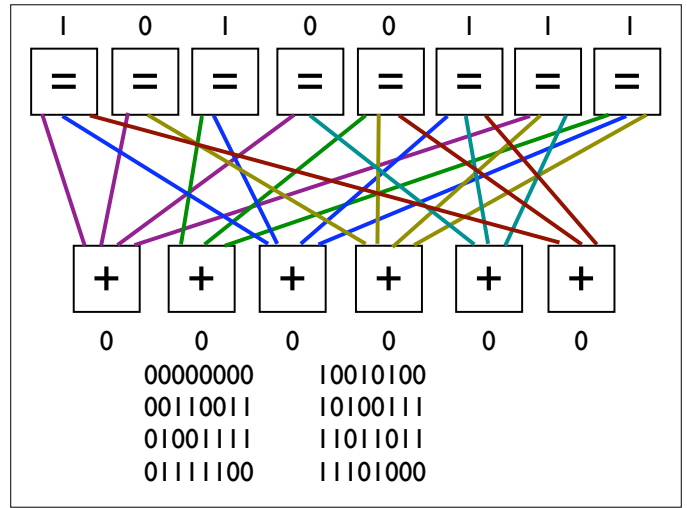
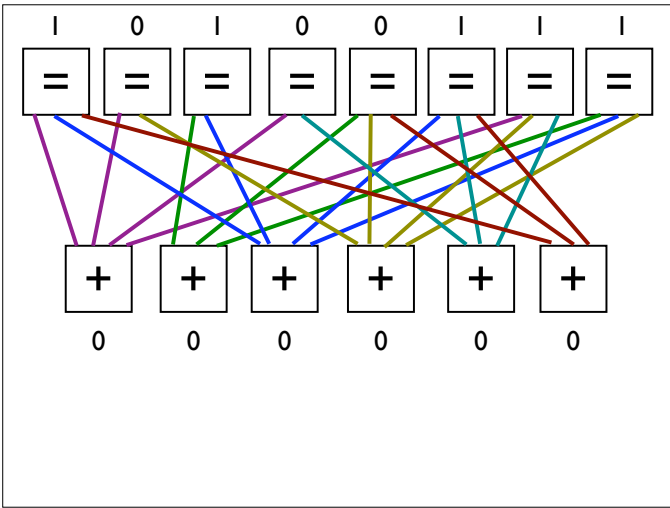












Low-Density Parity Codes

- 8 bits are encoding 3 bits
- Huge redundancy
- Yet no one single bit is useless
- Well...actually the last two bits are mutually redundant
- And the first 3 bits encode the data

Low-Density Parity Codes

- 8 bits are encoding 3 bits
- Huge redundancy
- Yet no one single bit is useless
- Well...actually the last two bits are mutually redundant
- And the first 3 bits encode the data

00000000	10010100
00110011	10100111
01001111	11011011
01111100	11101000

Encoding LDPC: the easy part

- Remember Hamming codes?
- We had two matrices: the check matrix and the generator matrix

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \Big| I_3$$

$$G = \left(I_4 \left| \begin{array}{ccc} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array} \right. \right)$$

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \Big| I_3$$

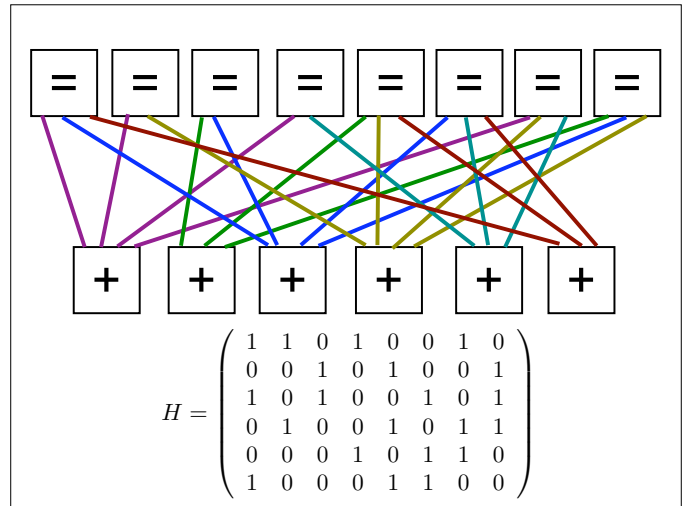
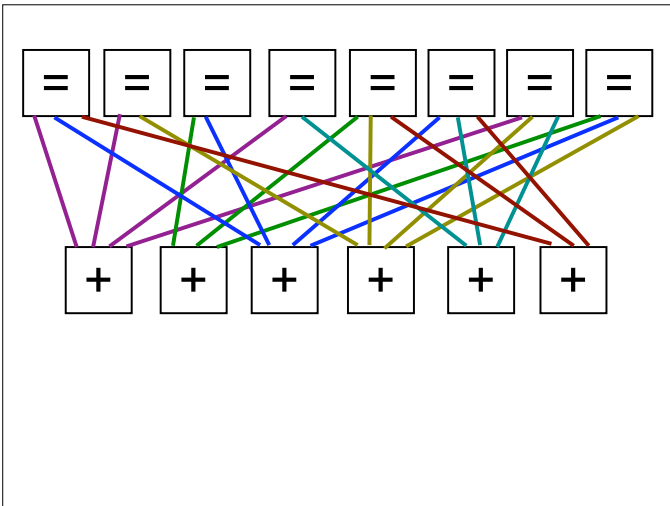
$$G = \left(I_4 \left| \begin{array}{ccc} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array} \right. \right)$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot (1 \ 0 \ 1 \ 0) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

- ## Hamming revisited
- On Monday, we got a code of 1011010 with the Hamming code
 - Today we got 1010101
 - Remember that the Hamming code stores its bits as P,P,D,P,D,D,D
 - We've "perverted" the Hamming code to reorder it into D,D,D,D,P,P,P

- ## Why did we do this?
- Hamming codes and Low-Density Parity Codes share something in common
 - Almost every practical error correction code shares the same thing in common!
 - They're linear codes
 - Essentially this means we can describe the codes as matrix multiplications

- ## Linear codes
- Every linear code has a check matrix and a generator matrix
 - Given one it's easy to find the other
 - Transform the check matrix into the form $[M \mid I_k]$
 - The generator matrix will be $[I_{n-k} \mid M^T]$



Encoding is not too bad

- Determining H (the check matrix) is pretty easy
- Constructing G (the generator matrix) is just a bunch of row operations which I am far too lazy to do for this example
- Encoding is multiplying G by our data vector

Decoding is hard :(

- If there are no errors, decoding is actually quite easy!
- Determining where the errors were is a different matter
- We don't have the Hamming property of having the codes set up to tell us the index of which bit got flipped

Decoding is hard :(

- If there are no errors, decoding is actually quite easy!
- Determining where the errors were is a different matter
- We don't have the Hamming property of having the codes set up to tell us the index of which bit got flipped

00000000	10010100
00110011	10100111
01001111	11011011
01111100	11101000

Decoding is hard :(

- If we get 00000010, it's obvious the intended code was 00000000
- It's the only one that was even close!
- What about 10110001?
- This is a big problem if we construct codes with little redundancy!

Decoding is hard :(

- If we get 00000010, it's obvious the intended code was 00000000
- It's the only one that was even close!
- What about 10110001?
- This is a big problem if we construct codes with little redundancy!

00000000	10010100
00110011	10100111
01001111	11011011
01111100	11101000

Decoding is hard :(

- The codes I've constructed here have an interesting property:
 - No 2 codes differ by fewer than 2 bit flips
 - Some differ by 3 bit flips

Decoding is hard :(

- The codes I've constructed here have an interesting property:
 - No 2 codes differ by fewer than 2 bit flips
 - Some differ by 3 bit flips

00000000	10010100
00110011	10100111
01001111	11011011
01111100	11101000

Low-Density Parity Codes

- Good news: LDPC are the best we can do!
- We can construct codes that get arbitrarily close to the Shannon limit (though sadly not exactly to the Shannon limit)
- Bad news: this is only true if we have a perfect decoder
- Worse news: a perfect decoder is NP-complete :(

Decoding LDPC

- A perfect decoder essentially compares the received message against every possible valid message
 - We then pick the one that's "closest" (fewest bits flipped)
- If we're encoding 100 bits of data, that's 2^{100} comparisons D: D: D:

LDPC and you!

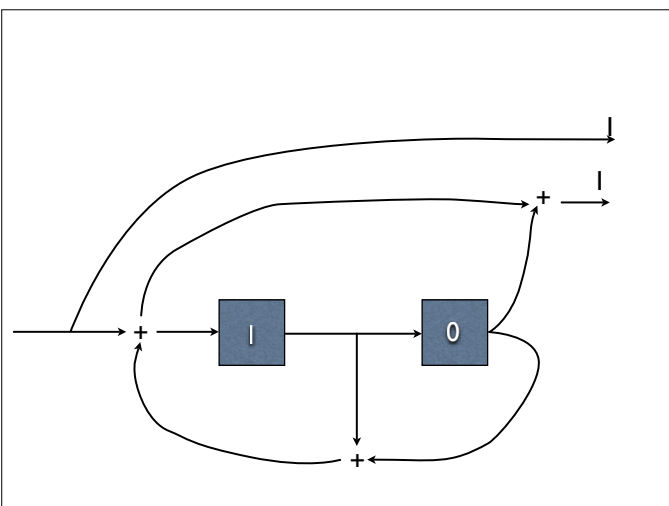
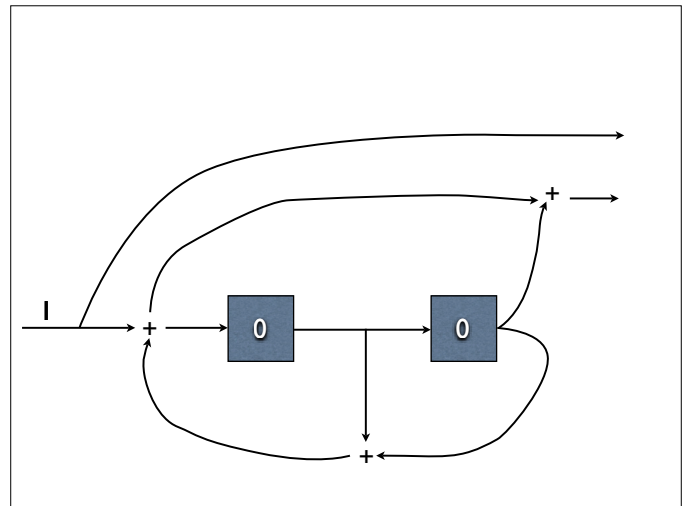
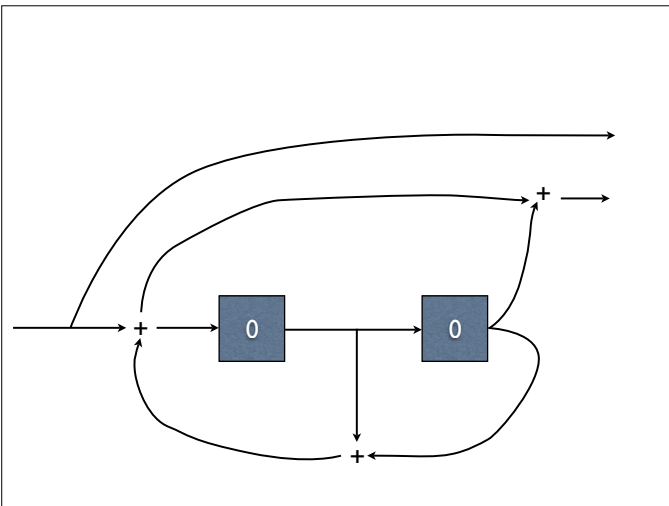
- Sub-optimal decoders can be very practical
- Turbo codes pwned the 1990s
- LDPC have been pwning the 2000s and currently stand as the best error correction codes in the world
- Used in 10 gigabit Ethernet
 - Woah! Error correction at the link level!

A note on constructing codes

- Let n be the number of “equal sign boxes”
- Let k be the number of “plus sign boxes”
- No matter how you hook them up, you’ll get $n - k + 1$ data bits and $k - 1$ redundant bits
- Coming up with good codes is still hard work, though!

Convolution codes

- Sort of anti-climactic after low-density parity codes
- Convolution codes are not the super best things ever :(
- They’re used everywhere (usually concatenated with other codes)
- They don’t have the computational cost of LDPCs or turbo codes



Convolution codes

- Trivial to encode, even on super underpowered hardware
- All you need is one register and a few XOR operations
- The last slide had a 1/2 convolution code (1 input bit, 2 output bits)

Puncture codes

- To the right is a "puncture matrix"
 - Read it column-wise
- For the first input bit, pass through both output bits
- For the second input bit, pass through only the second output bit
- For the third input bit, pass through only the first output bit
- Turned a 1/2 code into a 3/4 code

1	0	1
1	1	0

Decoding

- Decoding convolution codes is done through a Viterbi decoder
- Walk through the encoding process one step at a time, determining the most likely input

Viterbi decoding

- E.g., in our example, both registers had values 0 and our input was 1
- The output was (1, 1)
- If we assume the probability of a bit getting flipped is 0.1, our probability table actually looks like:

(0, 0)	0.01	(1, 0)	0.10
(0, 1)	0.10	(1, 1)	0.79

On input 0

(0, 0)	0.79	(1, 0)	0.10
(0, 1)	0.10	(1, 1)	0.01

On input 1

(0, 0)	0.01	(1, 0)	0.10
(0, 1)	0.10	(1, 1)	0.79

Viterbi decoder

- We look at the output we got and pick the input that had the highest probability of giving that output
- We update the state (i.e., registers) and keep going
- We never backtrack

Error correction code round-up

- Hamming codes
 - The first codes
 - Deterministic decoding
 - Lots of redundancy for little benefit
- Low-Density Parity Codes
 - Very flexible
- We can get arbitrarily close to the Shannon limit
- Optimal decoding is NP-complete
- Convolution codes
 - Super easy to encode
 - Fairly easy to decode, too

Reed-Solomon codes

- We transmit more values than are needed to uniquely describe the lower-degree polynomial
- The receiving end can figure out the original coefficients
- In linear algebra you (probably) learned how to make a best-fit lower-degree polynomial
- This works wonderfully! But computationally it is quite inefficient

Information theory wrap-up

- This concludes our whirlwind tour of information theory for now
- Later on in the course we'll revisit it
 - Not in so much detail!
 - These things don't "belong" at any one level

Application	Application	Error detection
	Presentation	
	Session	
Transport	Transport	Compression
Network	Network	Error correction
Link	Link	
Physical	Physical	

Physical layer

- You can't do anything at the physical layer!

Link layer

- Error detection is almost universally common, but not strictly necessary
- Error correction is handy in extreme cases
 - Extreme latency (Voyager space probes)
 - Extreme S/N (10 gigabit Ethernet)
- Compression not unheard of
 - 56k modems

Network layer

- The network is just a collection of link layers
- Any error detection/error correction should be taken care of by the link layer

Transport layer

- Error detection/correction is a good idea if we want a good abstraction (being able to read() and write() and pretend nothing goes wrong)
- We don't know what link layers are like for intermediate hops!

Application layer

- Compression makes a lot of sense here
 - There's no one magic bullet for compression; it depends on your data
- Error correction is interesting
 - We can think of more abstract kinds of noise