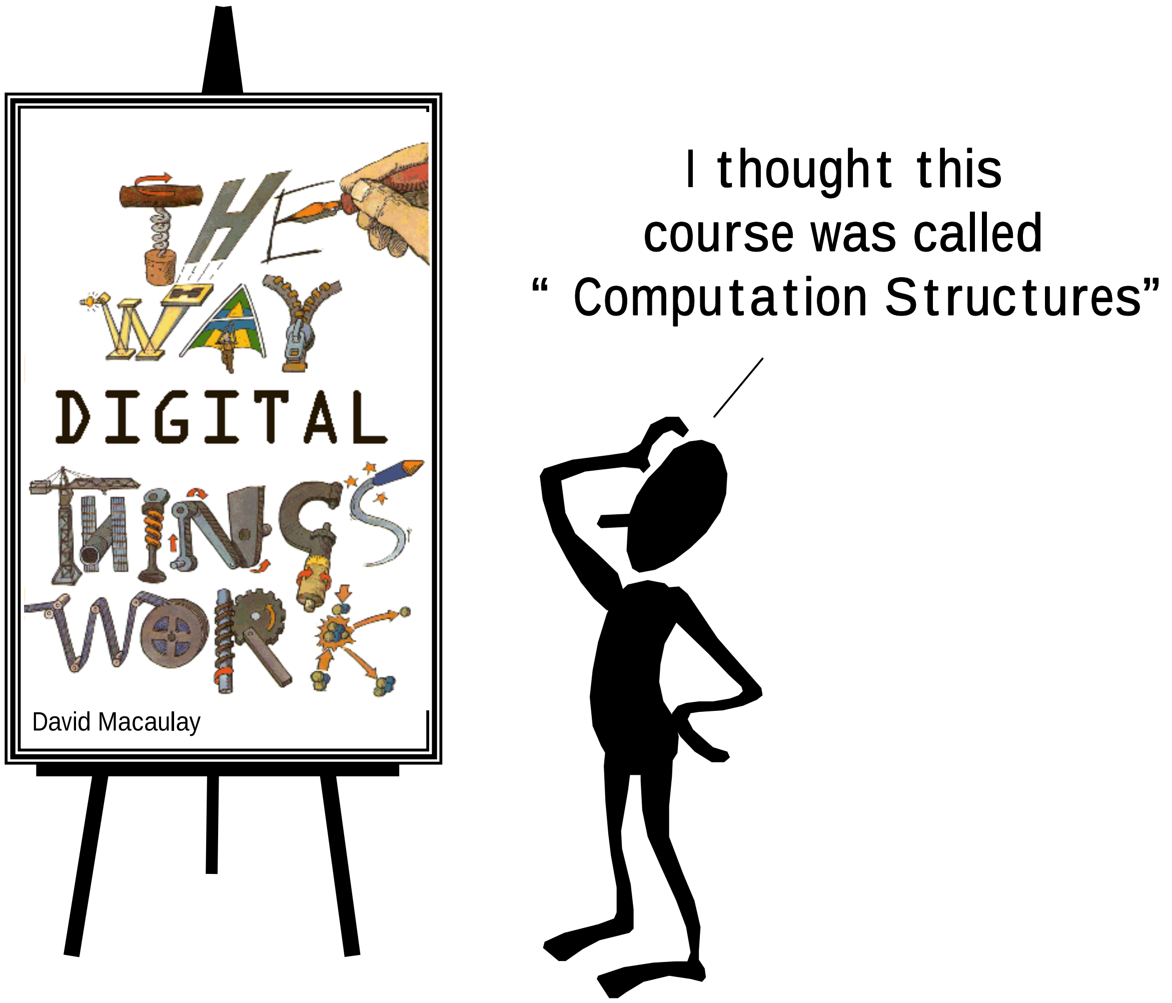
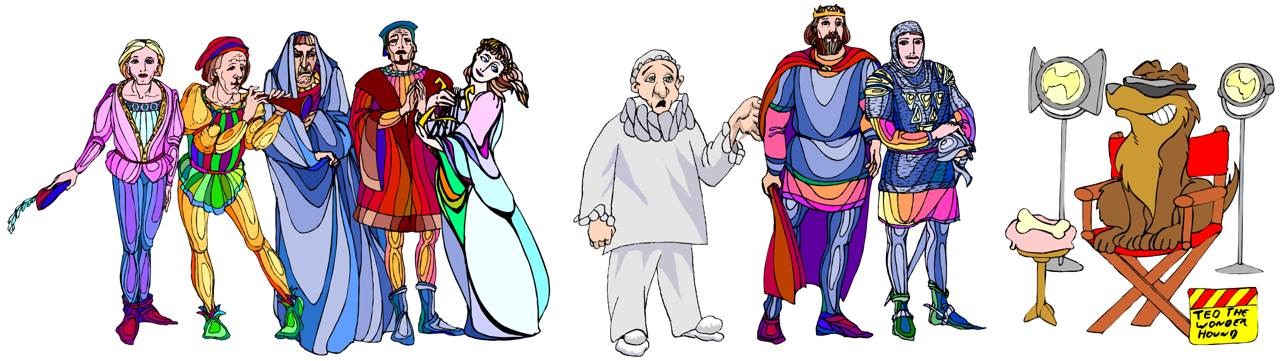
Welcome to 6.004!


THE WAY DIGITAL THINGS WORK
David Macaulay
I thought this course was called "Computation Structures"

Handouts: Lecture Slides, Info Sheet, Calendar

Meet the cast...



Support:	Bryt Bradley (NE43-253)
TAs:	Anthony Hui, Amy Vandiver, Suzette Vandivier, Jason Woolever, Serhii Zhak

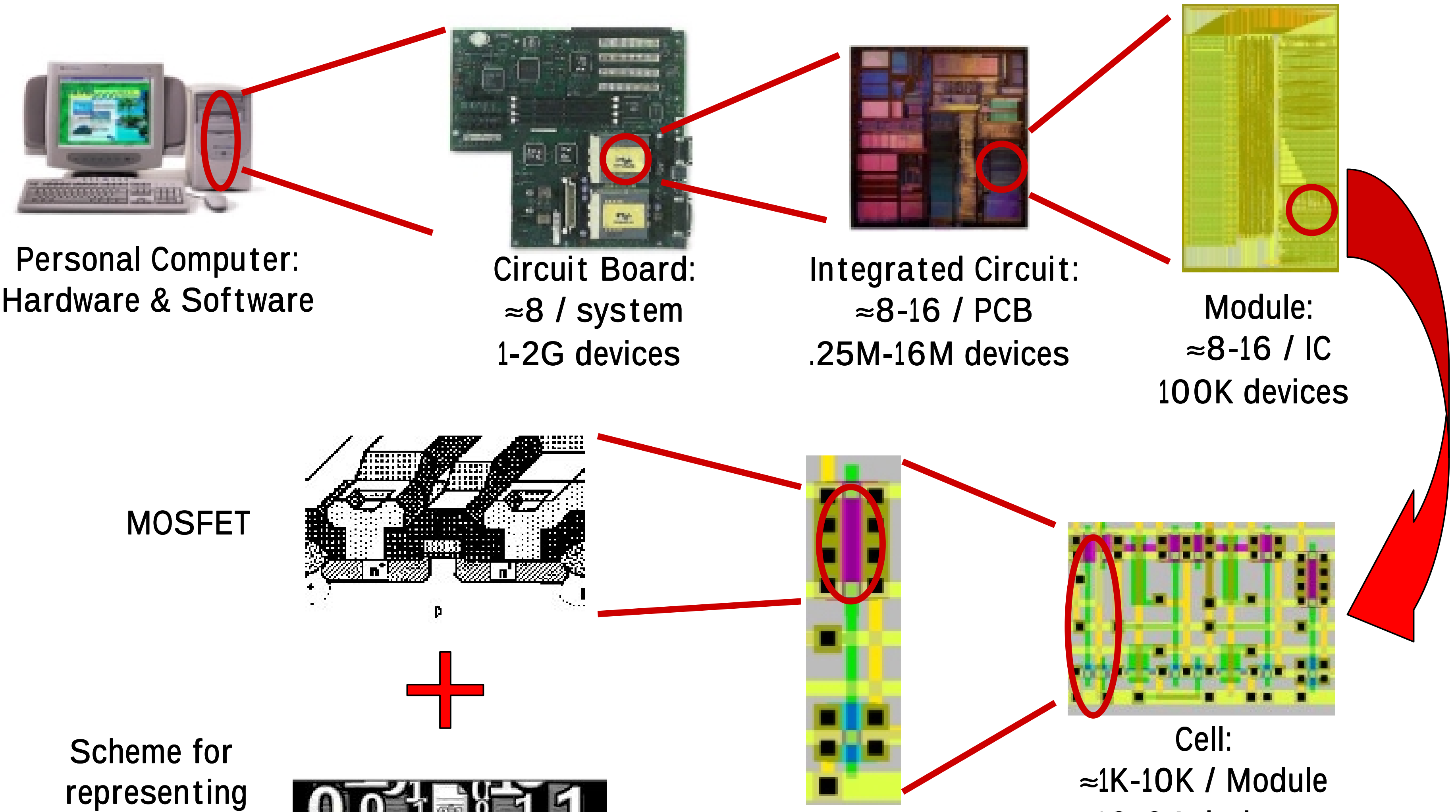
Labs & Courseware:	Chris Terman
Psets & Quizzes:	Steve Ward
Lectures:	Leonard McMillan

Mechanics of 6.004

1) Lectures followed by recitations with examples.
2) Weekly problem sets (except in those weeks with quizzes)
3) Weekly labs (except those weeks with quizzes)

Tuesday	Wednesday	Thursday	Friday
Lecture	Recitation: Tutorial probs.	Lecture PSet Assigned	Recitation: Tutorial probs.
Lecture	Recitation: Tutorial probs.	Lecture PSet Assigned Lab Assigned	Recitation: Tutorial probs. PSet due
Lecture PSet solutions	Recitation: Tutorial probs.	Lecture PSet Assigned Lab Assigned Lab Due	Recitation: Tutorial probs. PSet due

How do you build systems with >1G components?



Personal Computer:
Hardware & Software

Circuit Board:
≈8 / system
1-2G devices

Integrated Circuit:
≈8-16 / PCB
.25M-16M devices

Module:
≈8-16 / IC
100K devices

Cell:
≈1K-10K / Module
16-64 devices

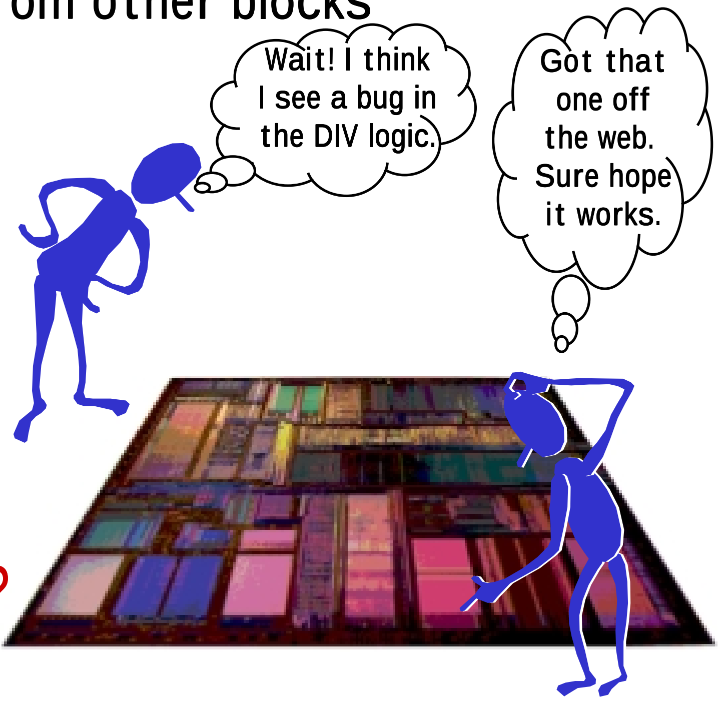
Gate:
≈2-16 / Cell
8 devices

MOSFET

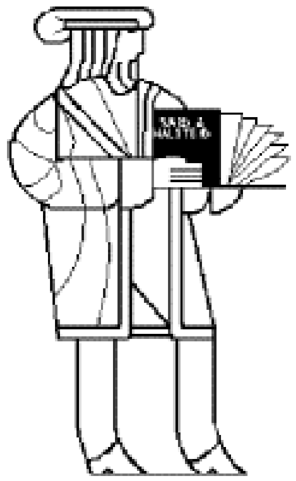
Scheme for representing information

What do we see?

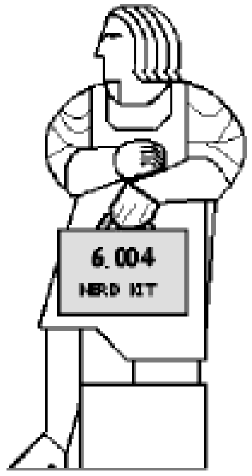
- Structure
  - hierarchical design
    building blocks provide natural boundaries (interfaces) that insulate one block's implementation from other blocks
  - limited complexity at each level
  - reusable building blocks
- A *lot* of design work
  - dividing up the design process
  - many people involved, in small groups
  - getting boundaries "right" is crucial
- What makes a good system design?
  - must be good at many different tasks
    ("good" = flexible & cost-effective)
  - reliable in a wide range of environments
  - accommodates future technical improvements


Wait! I think I see a bug in the DIV logic.
Got that one off the web. Sure hope it works.

Our plan of attack...



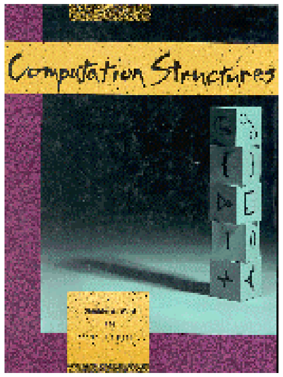
- Understand how things work
- Encapsulate our understanding using appropriate abstractions
- Discover organizational principles



- Roll up our sleeves and design at each level of hierarchy
- Learn engineering tricks
  - history
  - systematic approaches
  - algorithms
  - how to diagnose, fix, and avoid bugs

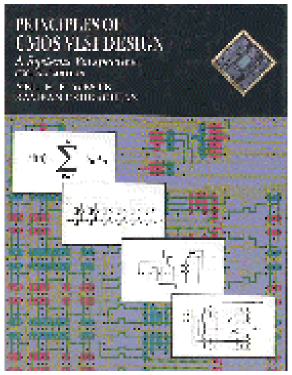
Resources… worth reading

Design Principles



Computation Structures
Ward & Halstead
ISBN 0-262-23139-5

IC technology

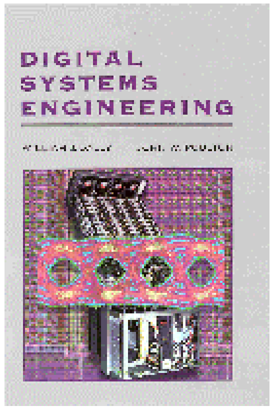


Principles of CMOS VLSI Design
Weste & Eshraghian
ISBN 0-201-53376-6

Architecture

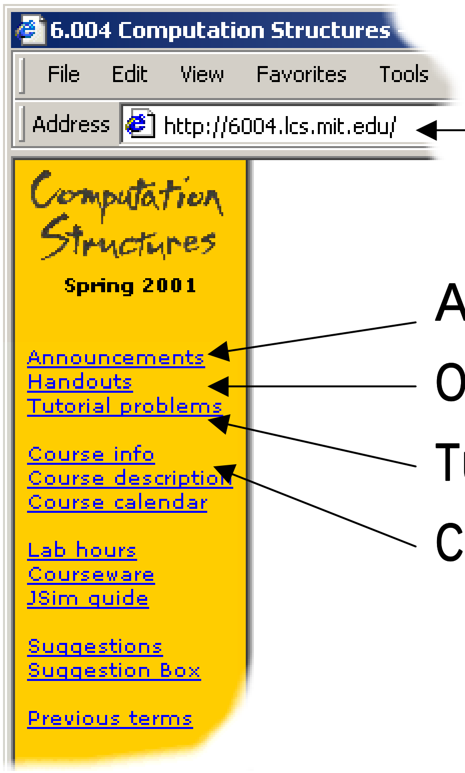
Computer Architecture: A Quantitative Approach
Hennessy & Patterson
ISBN 1-55860-329-8

Systems



Digital Systems Engineering
Dally & Poulton
ISBN 0-521-59292-5

Resources… worth clicking



http://6004.lcs.mit.edu

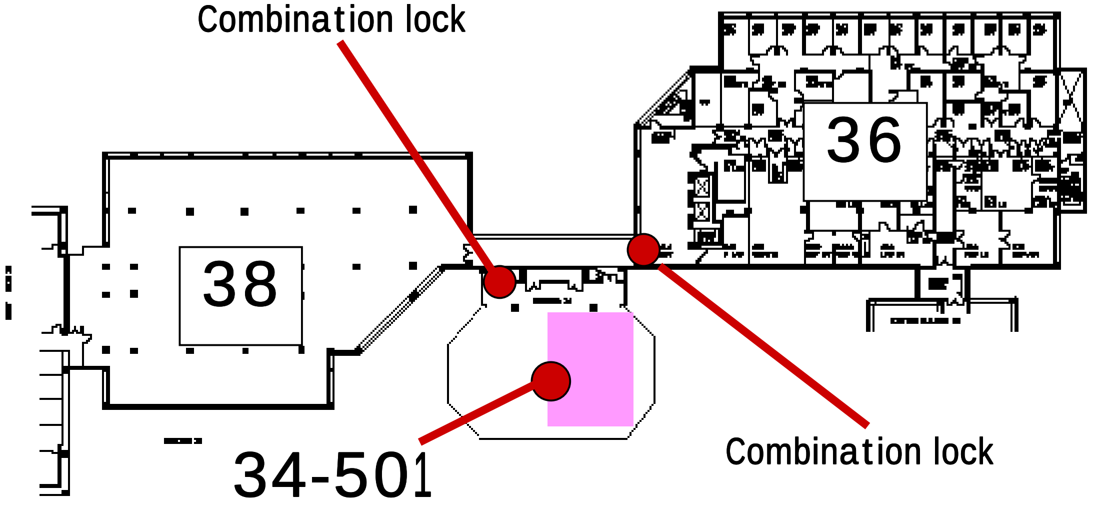
Announcements, pset corrections, etc.

On-line copies of all handouts

Tutorial problems for each lecture

Course information, policies, etc.

Where to find machines and help



What is “computation”?

Computation is about “ processing information”

- Transforming information from one form into another
- Deriving new information from old
- Effective procedure for generating an output given an input sequence
- Computation describes the motion of information through time
- Communication describes the motion of information through space

HAL 9000

What is "information"?

<u>Lexicographers</u>

information, *n*. Knowledge communicated or received *concerning a particular fact or circumstance*.

<u>Philosophers</u>

data < *information* < knowledge < wisdom

<u>Engineers</u>

Information resolves uncertainty. According to our use, it has nothing to do with knowledge or meaning. Information is simply that which cannot be predicted.

Still uncertain what information is?

Suppose you are faced with N equally probable choices, and I give you data that narrows it down to M choices, then I have given you:

$\log_2(N/M)$ bits of information

Information is measured in bits (binary digits, the number of 0/1's required to encode choice(s)

It is often useful to talk about the average amount of information in a stream data. This measure, called Entropy, is given by the following formula:

$$Entropy = \sum_{i=1}^{N} \frac{M_i}{N} \log_2\left(\frac{N}{M_i}\right)$$

Information in message i

Probability of message i

Example:

- When I tell you a coin flip is heads:
  $\log_2(2/1) = 1$ bit

Encoding information

- Encoding describes the process of *assigning representations to information*
- Choosing an appropriate and efficient encoding is a real engineering challenge
- Impacts design at many levels
  - Mechanism (devices, # of components used)
  - Efficiency (bits used)
  - Reliability (noise)
  - Security (encryption)

Fixed-length encodings

If all choices are equally likely (or we have no reason to expect otherwise), then a fixed-length code is often used. Such a code will use at least enough bits to represent the information content.

ex. Decimal digits 10 = {0,1,2,3,4,5,6,7,8,9}

4-bit BCD (binary code decimal)

$$\sum_{i=1}^{10} \frac{1}{10} \log_2(10) = 3.322 < 4\,bits$$

ex. ~84 English characters = {A-Z (26), a-z (26), 0-9 (11), punctuation (8), math (9), financial (4)}

7-bit ASCII (American Standard Code for Information Interchange)

$$\sum_{i=1}^{84} \frac{1}{84} \log_2(84) = 6.392 < 7\,bits$$

Encoding numbers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

03720

Octal - base 8

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

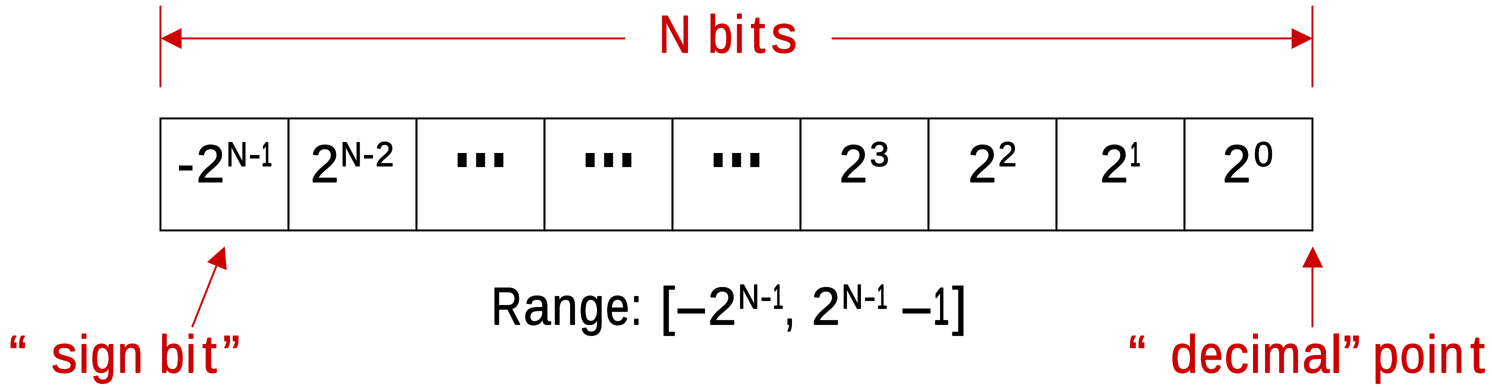
Oftentimes we will find it convenient to cluster groups of bits together for a more compact representation. Two popular groupings are clusters of 3 bits and 4 bits.

0x7d0

Hexadecimal - base 16

0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f

Signed integers: 2's complement



Range: $[-2^{N-1}, 2^{N-1}-1]$

8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

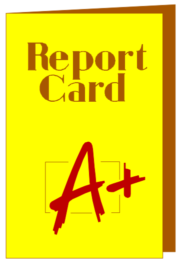
By moving the implicit location of "decimal" point, we can represent fractions too:

$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

When choices aren't equally likely

When the choices have different probabilities (p_i), you get more information when learning of a unlikely choice than when learning of a likely choice

$$\text{Information from choice}_i = \log_2(1/p_i) \text{ bits}$$
$$\text{Entropy of Information} = \Sigma p_i \log_2(1/p_i)$$



Example

choice_i	*p_i*	*$\log_2(1/p_i)$*
"A"	1/2	1 bit
"B"	1/6	2.58 bits
"C"	1/6	2.58 bits
"D"	1/6	2.58 bits

Average information
= (.5)(1) + (3)(.167)(2.58)
= 1.54 bits

Can we find an encoding where transmitting 1000 choices takes 1540 bits on the average? Using two bits for each choice = 2000 bits

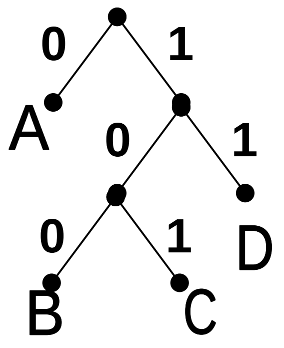
Variable-length encodings

Use shorter bit sequences for high probability choices, longer sequences for less probable choices

choice$_i$	*p_i*	*encoding*
"A"	1/2	**0**
"B"	1/6	**100**
"C"	1/6	**101**
"D"	1/6	**11**

A B A C D A

01000101110



Huffman Decoding Tree

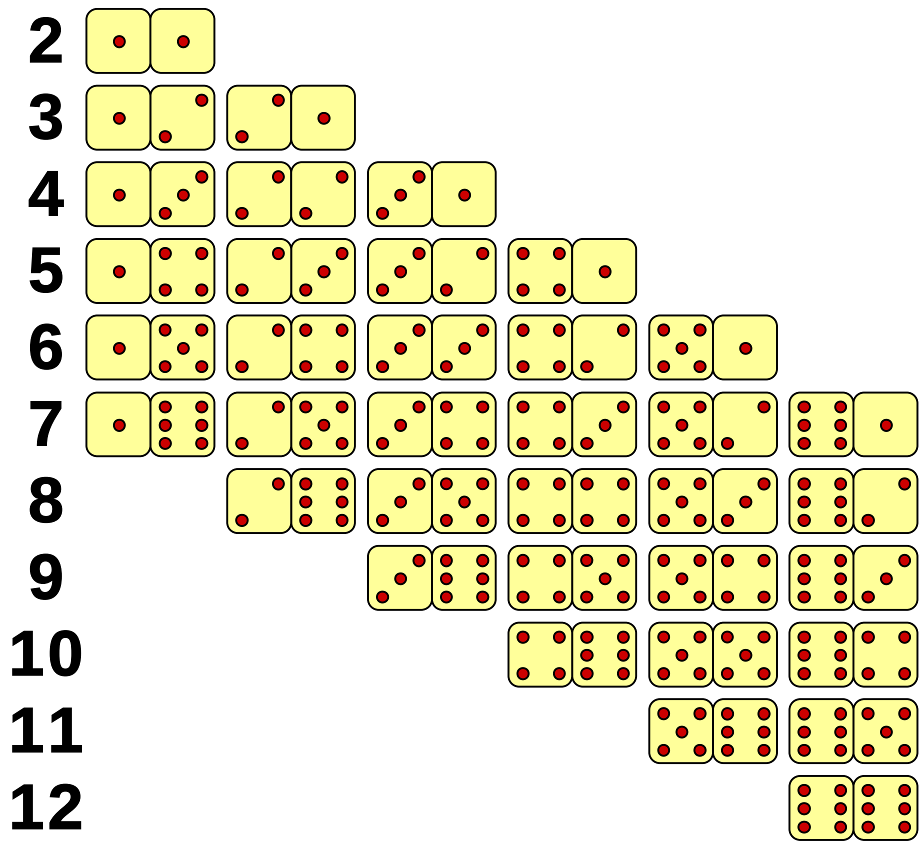
Average information
$= (.5)(1) + (2)(.167)(3) + (.167)(2)$
$= 1.836$ bits

Transmitting 1000 choices takes an average of 1836 bits..better but not optimal

To get a more efficient encoding (closer to information content) we need to encode sequences of choices, not just each choice individually. This is the approach taken by most file compression algorithms...

Another example: sum of 2 dice

How much information is conveyed in the sum of the roll of two dice? Since all outcomes are not equally likely, it depends on the roll. The possibilities are enumerated in the diagram below:



$i_2 = \log_2(36/1) = 5.170$ bits
$i_3 = \log_2(36/2) = 4.170$ bits
$i_4 = \log_2(36/3) = 3.585$ bits
$i_5 = \log_2(36/4) = 3.170$ bits
$i_6 = \log_2(36/5) = 2.848$ bits
$i_7 = \log_2(36/6) = 2.585$ bits
$i_8 = \log_2(36/5) = 2.848$ bits
$i_9 = \log_2(36/4) = 3.170$ bits
$i_{10} = \log_2(36/3) = 3.585$ bits
$i_{11} = \log_2(36/2) = 4.170$ bits
$i_{12} = \log_2(36/1) = 5.170$ bits

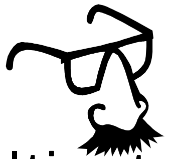
Average information in a roll

Thus, the average information provided by the sum of 2 dice:

$$i_{\text{ave}} = \sum_{i=2}^{12} \frac{M_i}{N} \log_2\left(\frac{N}{M_i}\right) = \sum_i p_i \log_2\left(\frac{1}{p_i}\right) = 3.274 \; \textit{bits}$$

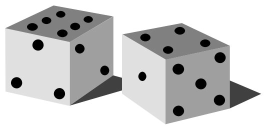
At this point we can consider encoding the sum of a dice roll. A reasonable starting point is to choose 4 bits and encode each roll as a binary-coded integer. That's not so bad. We have a few wasted codes (0,1,13,14,15), but 4 bits is the next largest integer after 3.274. Can we do any better?

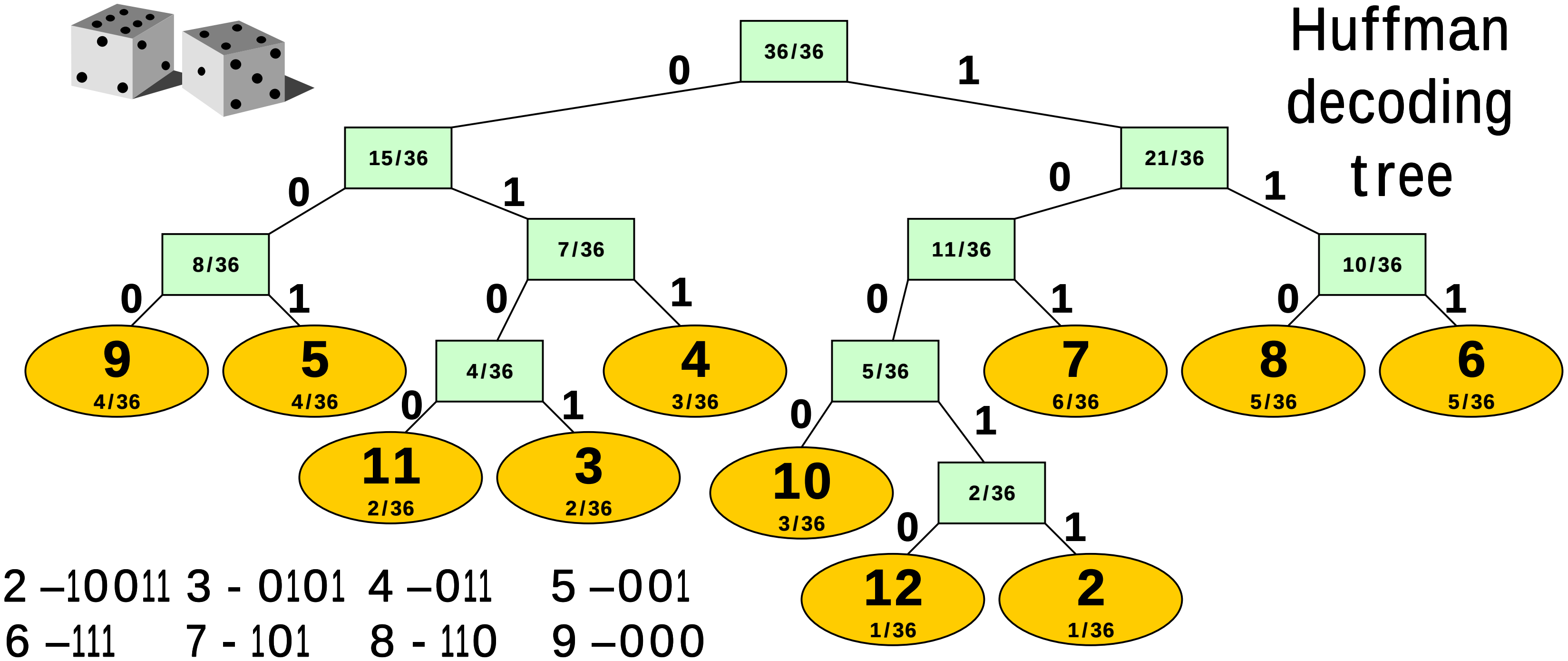
One method for constructing a variable length encoding involves constructing a binary tree. We recursively select the two symbols (the piece of information that we want to encode) with the lowest probability, combine them into a node, and insert this node as a new symbol whose probability is the sum of its children. The process is repeated until only one symbol is left.



Encoding dice rolls

The resulting tree might look something like the one shown below. We can then assign encodings to each symbol by labeling each right-hand child with " 1" and each left-hand child with " 0". The encoding for any particular symbol can be determined by tracing a path from the tree' s root to the symbol of interest. This form of variable-length coding is call Huffman coding.





Huffman decoding tree

2 –10011 3 - 0101 4 –011 5 –001
6 –111 7 - 101 8 - 110 9 –000
10 –1000 11 –0100 12 - 10010

Encoding efficiency

The efficiency of an encoding strategy can be measured by how close the average code size is to the information content (entropy) of the stream. How does this encoding compare to the information content of the roll?

$$b_{ave} = \frac{1}{36}5 + \frac{2}{36}4 + \frac{3}{36}3 + \frac{4}{36}3 + \frac{5}{36}3 + \frac{6}{36}3$$

$$+ \frac{5}{36}3 + \frac{4}{36}3 + \frac{3}{36}4 + \frac{2}{36}4 + \frac{1}{36}5$$

$$b_{ave} = 3.306$$

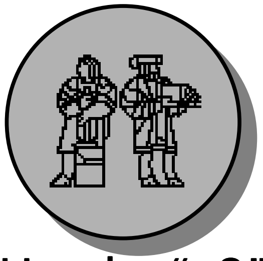
Pretty close. There are tricks for getting closer. Can you think of any?

Encoding considerations

- Encoding schemes that attempt to match the information content of a data stream are essentially removing redundancy. They are *data compression* techniques.
- However, sometimes our goal in encoding information is *increase redundancy*, rather than remove it. Why?
  - Make the information easy to manipulate (fixed-sized encodings)
  - Make the data stream resilient to noise (error detecting and correcting codes)

Error detection and correction

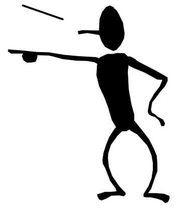
Suppose we wanted to reliably transmit the result of a single coin flip:



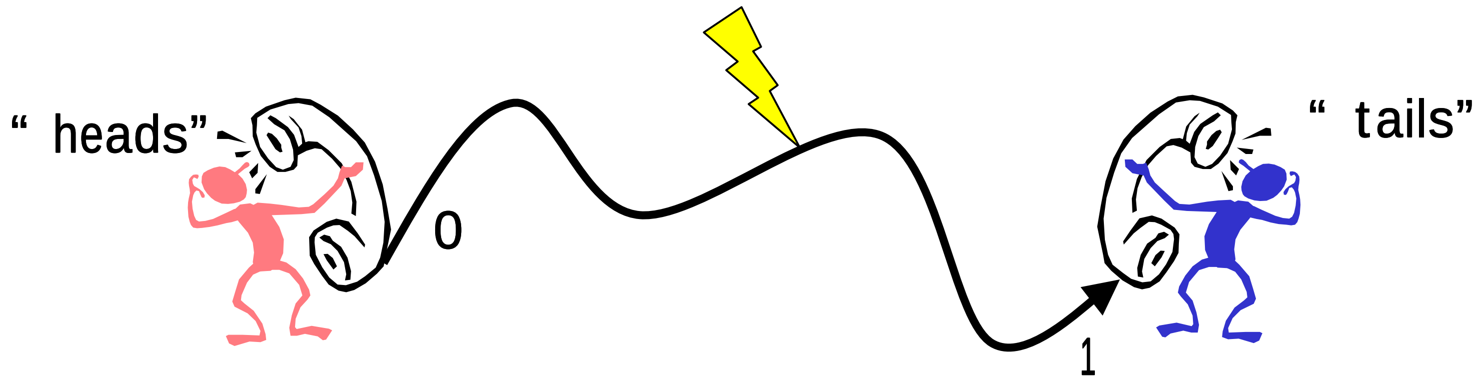
Heads: “ 0”

Tails: “ 1”

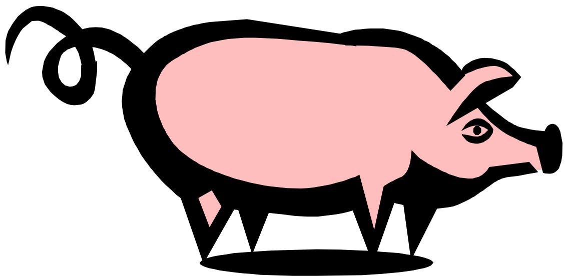
This is a prototype of the “ bit” coin for the new information economy. Value = 12.5¢



Further suppose that during transmission a single-bit error occurs, i.e., a single “ 0” is turned into a “ 1” or a “ 1” is turned into a “ 0”.



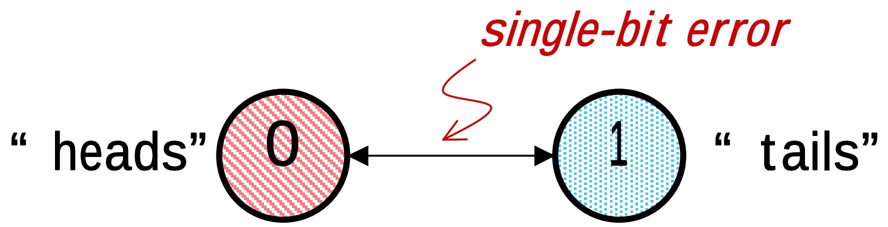
Hamming distance



Hamming distance: The number of digit positions in which the corresponding digits of two encodings of the same length are different

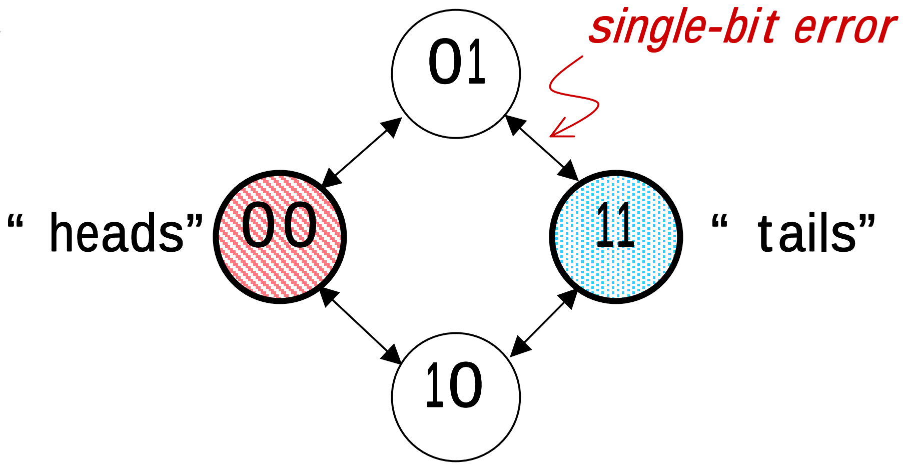
The Hamming distance between a valid binary code word and the same code word with single-bit error is 1.

The problem with our simple encoding is that the two valid code words (“ 0” and “ 1”) also have a Hamming distance of 1. So a single-bit error changes a valid code word into another valid code word…



Error detection

What we need is an encoding where a single-bit error does not produce another valid code word.

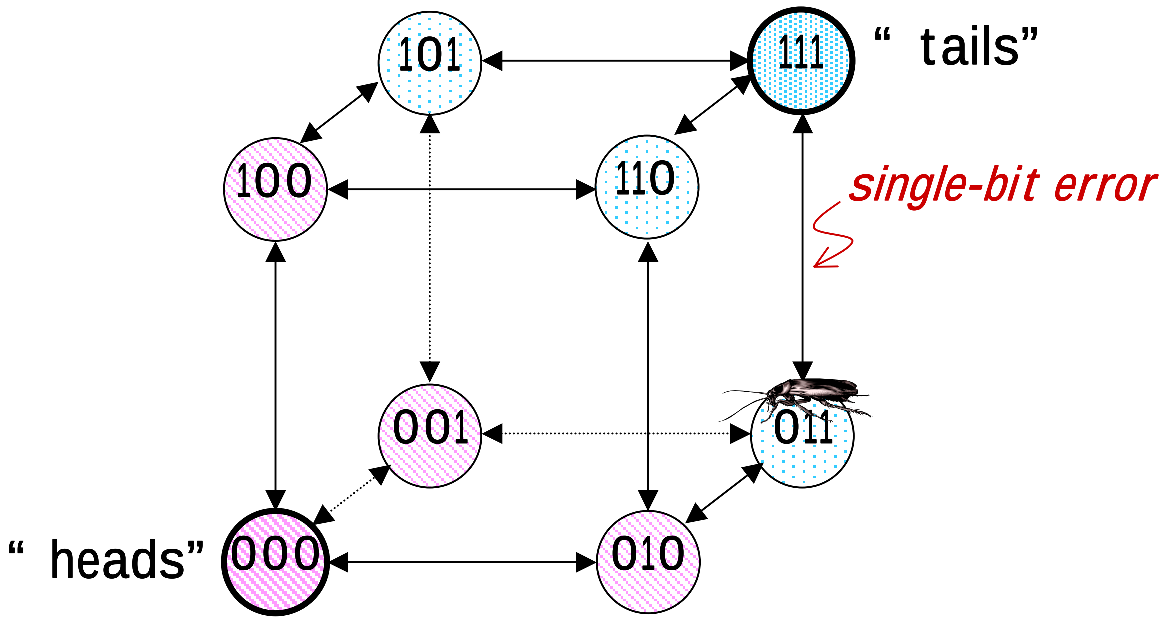


If D is the minimum Hamming distance between code words, we can detect up to (D-1)-bit errors

We can add single-bit error detection to any length code word by adding a *parity bit* chosen to guarantee the Hamming distance between any two valid code words is at least 2. In the diagram above, we’ re using “ even parity” where the added bit is chosen to make the total number of 1’ s in the code word even.

Can we correct detected errors? Not yet…

Error correction



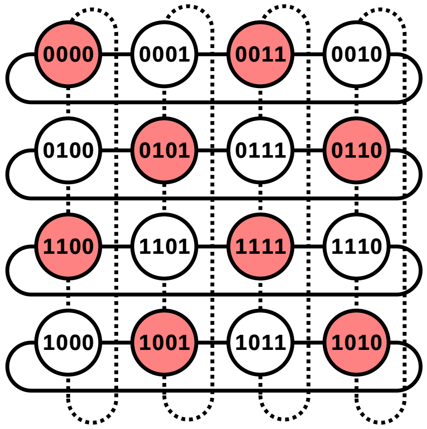
If D is the minimum Hamming distance between code words, we can correct up to $\left\lfloor \frac{D-1}{2} \right\rfloor$-bit errors

By increasing the Hamming distance between valid code words to 3, we guarantee that the sets of words produced by single-bit errors don’t overlap. So if we detect an error, we can perform *error correction* since we can tell what the valid code was before the error happened.

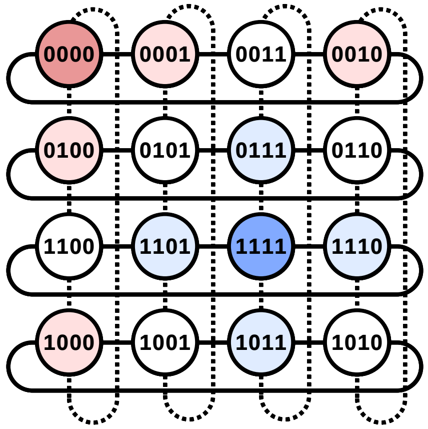
- Can we safely detect double-bit errors while correcting 1-bit errors?
- Do we always need to triple the number of bits?

Quick look at 4 bits

Using 4 bits we can separate 8 encodings with a Hamming distance of 1 to allow for 1-bit error detection, thus, encoding 3 bits of information with 4 bits. Bit patterns indicating errors differ from valid encodings by their parity (i.e. whether the number of 1's used in the pattern is even or odd).



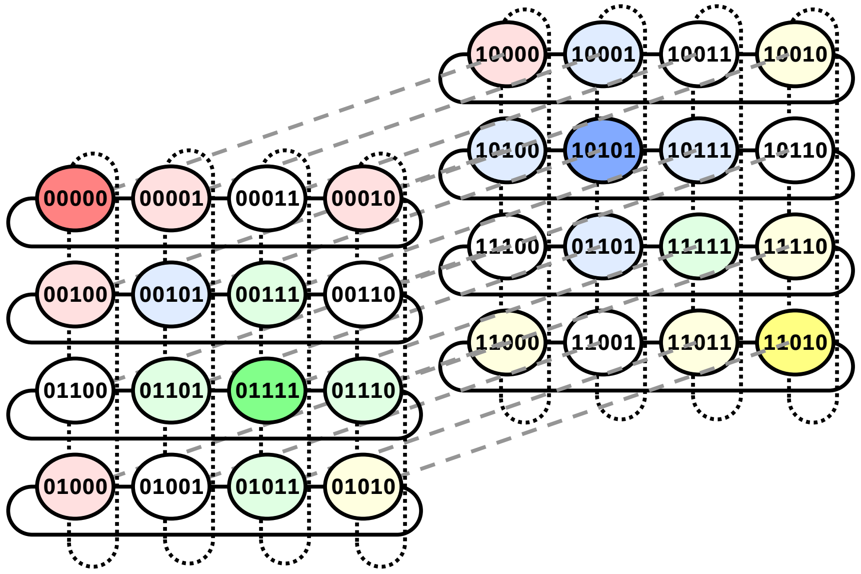
This is a flattened hypercube. Notice how the top 2 rows resemble a 3-bit 3D cube (we'll see this again soon).



Unfortunately, 4 bits are no better than 3 bits in terms of the number of encodings that can be separated with a Hamming distance of 3. The most encodings that we can fit is 2. We can, however, find 2 encodings with a Hamming distance of 4. These encodings allow 1-bit errors to be corrected, and 2-bit errors to be detected.

And 5 bits... (62.5 ¢)

It takes 5 bits before we can find more that 2 encoding separated by a Hamming distance of 3.



Shown on the right are four 5-bit encodings {00000, 01111, 10101, 11010} separated by a Hamming distance of at least 3. With it we can correct any 1-bit error,and we can detect some (but not all) 2-bit errors.

We' ll stop here, because we really need a computer to go much beyond 5 bits, and we' ll need to build one first!

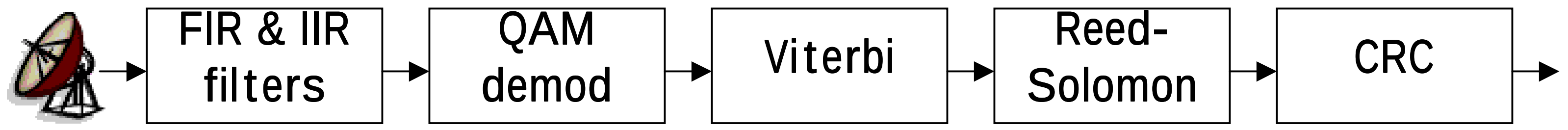
Coding theory is your friend…

We can use encodings to solve many problems:

- detect multi-bit errors: cyclical redundancy check (CRC)
- correct burst errors: Reed-Solomon
- improve S/N ratio: Viterbi, PRML

In fact, multiple codes are often used in combination…

Example: digital satellite downlink (eg, DBS TV)



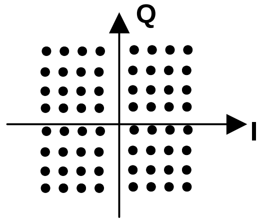
Equalize channel

Decode 6- or 7-bit symbols

Increase effective S/N ratio

Correct burst errors

Detect uncorrected errors

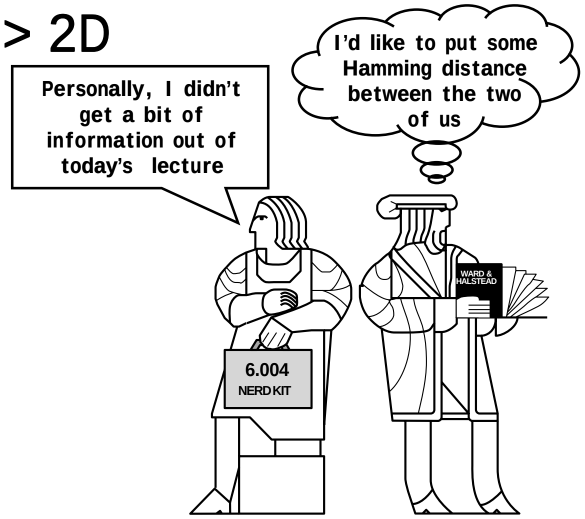


Summary

- Information resolves uncertainty
- Choices equally probable:
  - N choices down to M $\rightarrow \log_2(N/M)$ bits of information
  - use fixed-length encodings
  - encoding numbers: 2's complement signed integers
- Choices not equally probable:
  - choice_i with probability $p_i \rightarrow \log_2(1/p_i)$ bits of information
  - average number of bits = $\Sigma p_i \log_2(1/p_i)$
  - use variable-length encodings
- To detect D-bit errors: Hamming distance > D
- To correct D-bit errors: Hamming distance > 2D

Next time:

- encoding information electrically
- the digital abstraction
- noise margins
- rules for combinational devices


Personally, I didn't get a bit of information out of today's lecture
I'd like to put some Hamming distance between the two of us
6.004 NERD KIT
WARD & HALSTEAD