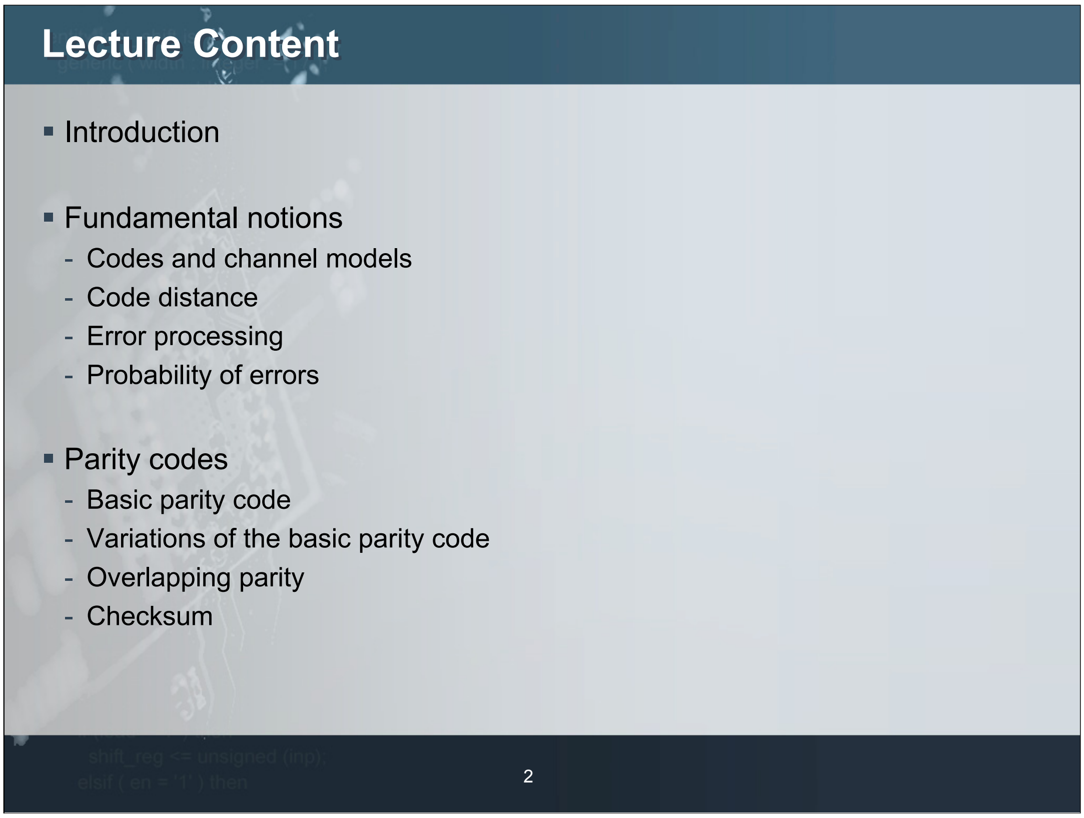
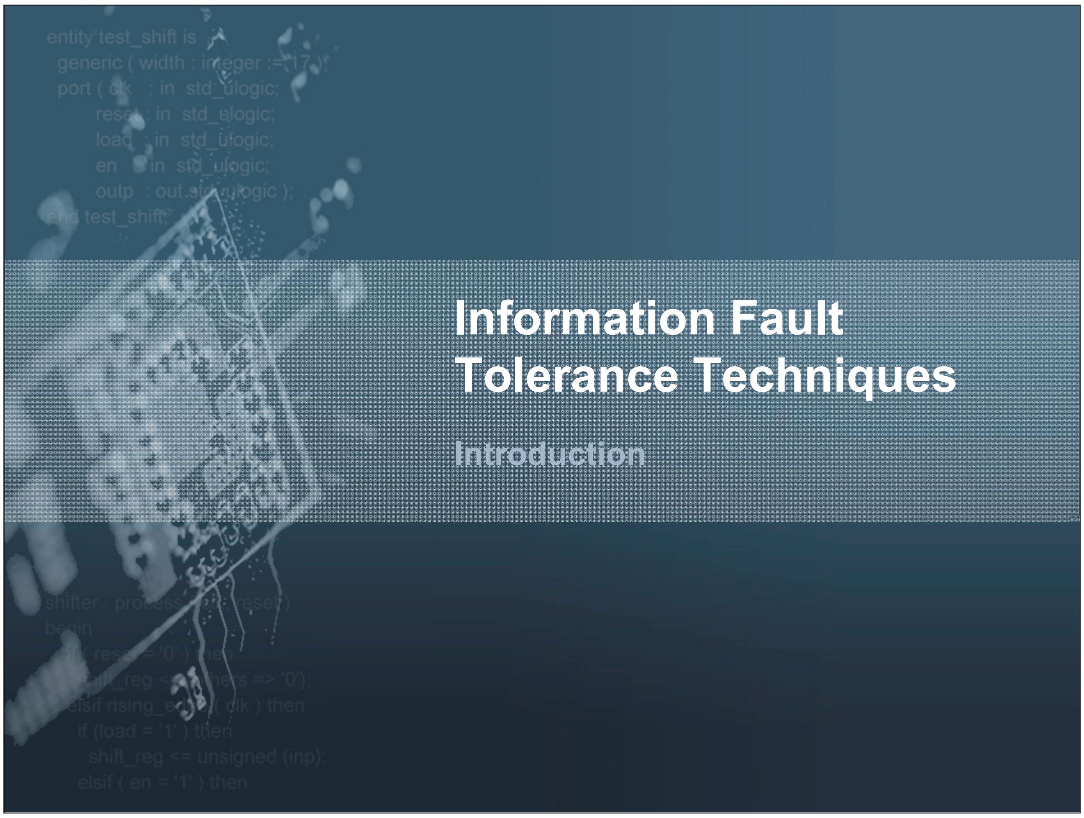
Digitalni sistemi otporni na otkaz

Predavanje VI



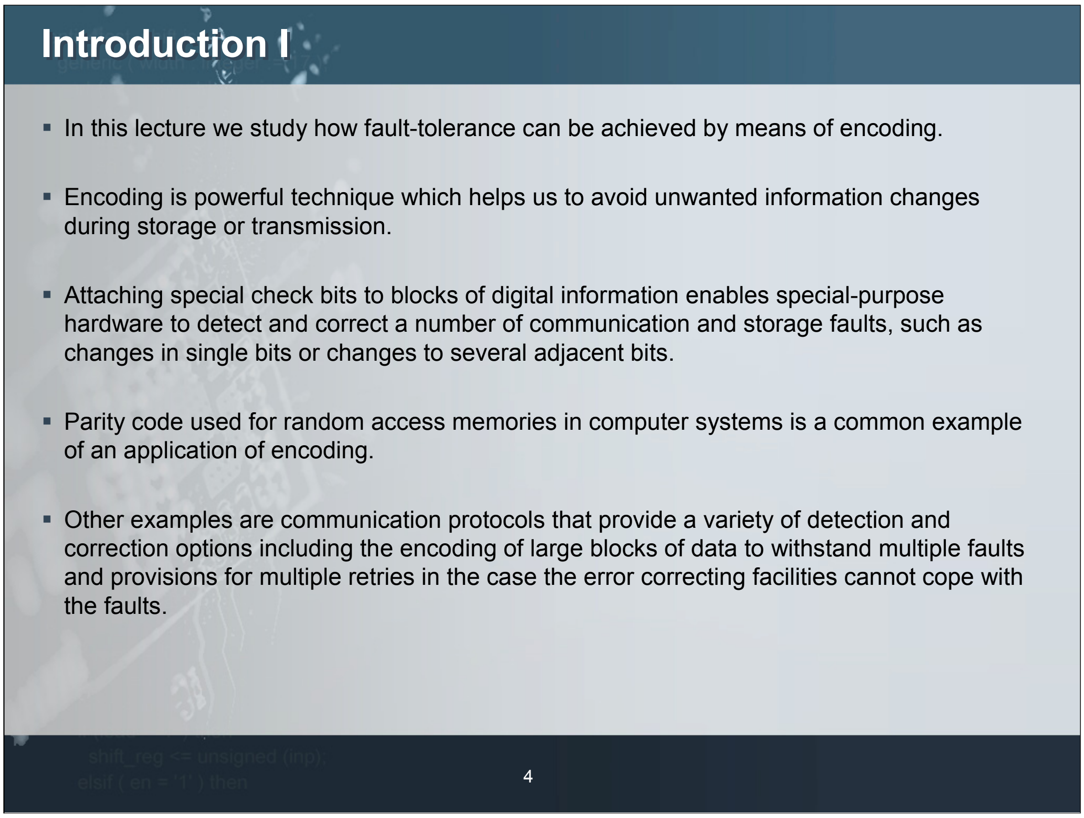
Lecture Content

- Introduction
- Fundamental notions
  - Codes and channel models
  - Code distance
  - Error processing
  - Probability of errors
- Parity codes
  - Basic parity code
  - Variations of the basic parity code
  - Overlapping parity
  - Checksum



Information Fault Tolerance Techniques

Introduction



Introduction I

- In this lecture we study how fault-tolerance can be achieved by means of encoding.
- Encoding is powerful technique which helps us to avoid unwanted information changes during storage or transmission.
- Attaching special check bits to blocks of digital information enables special-purpose hardware to detect and correct a number of communication and storage faults, such as changes in single bits or changes to several adjacent bits.
- Parity code used for random access memories in computer systems is a common example of an application of encoding.
- Other examples are communication protocols that provide a variety of detection and correction options including the encoding of large blocks of data to withstand multiple faults and provisions for multiple retries in the case the error correcting facilities cannot cope with the faults.

Introduction II

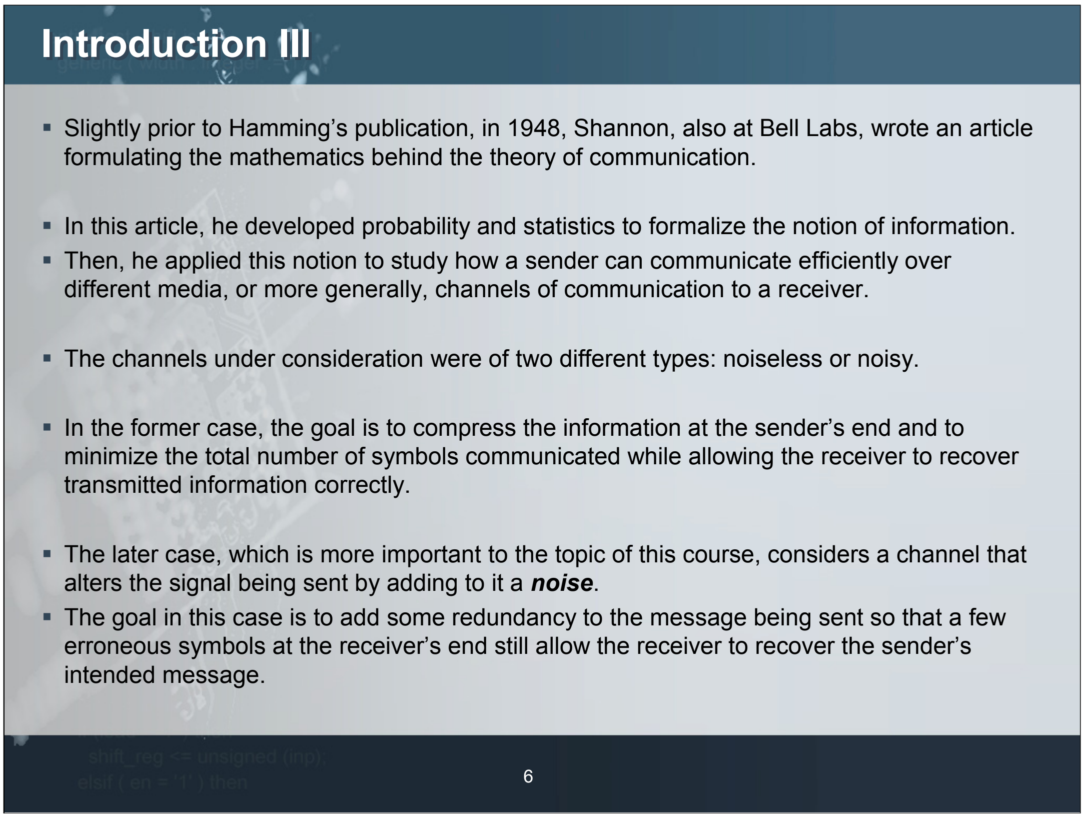
- Coding theory was originated in the late 1940s, by two seminal works by Hamming and Shannon.

- Hamming, working at Bell Laboratories in the USA, was studying possibilities for protecting storage devices from the corruption of a small number of bits by a code which would be more efficient than simple repetition.

- He realized the need to consider sets of words, or *codewords*, where every pair differs in a large number of bit positions.

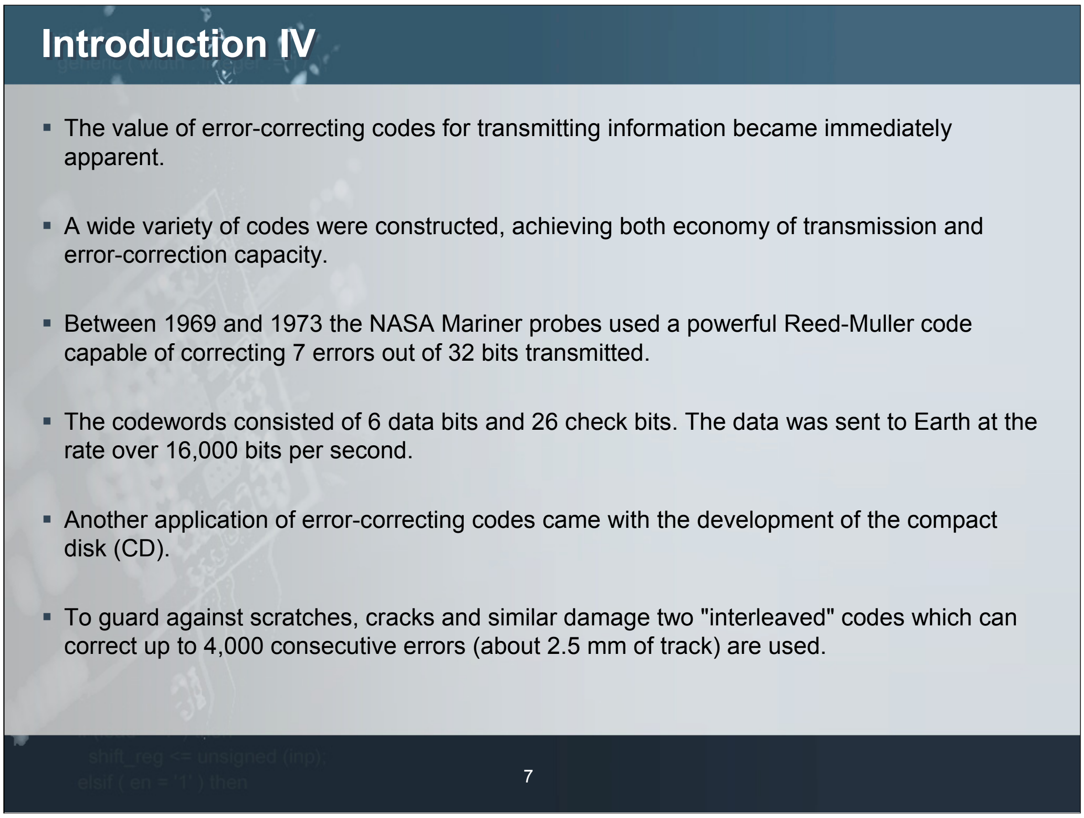
- Hamming defined the notion of distance between two words and observed this was a metric, thus leading to interesting properties. This distance is now called ***Hamming distance***.

- His first attempt produced a code in which four data bits were followed by three check bits which allowed not only the detection but the correction of a single error.



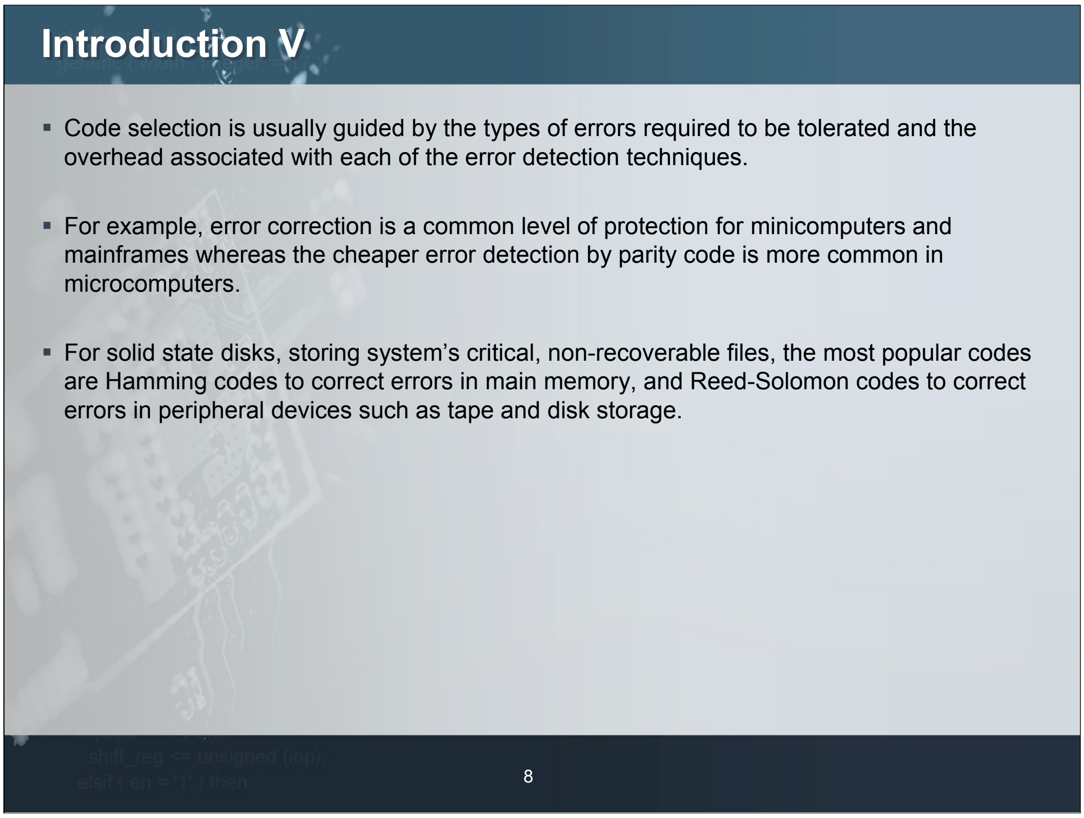
Introduction III

- Slightly prior to Hamming’s publication, in 1948, Shannon, also at Bell Labs, wrote an article formulating the mathematics behind the theory of communication.
- In this article, he developed probability and statistics to formalize the notion of information.
- Then, he applied this notion to study how a sender can communicate efficiently over different media, or more generally, channels of communication to a receiver.
- The channels under consideration were of two different types: noiseless or noisy.
- In the former case, the goal is to compress the information at the sender’s end and to minimize the total number of symbols communicated while allowing the receiver to recover transmitted information correctly.
- The later case, which is more important to the topic of this course, considers a channel that alters the signal being sent by adding to it a ***noise***.
- The goal in this case is to add some redundancy to the message being sent so that a few erroneous symbols at the receiver’s end still allow the receiver to recover the sender’s intended message.



Introduction IV

- The value of error-correcting codes for transmitting information became immediately apparent.
- A wide variety of codes were constructed, achieving both economy of transmission and error-correction capacity.
- Between 1969 and 1973 the NASA Mariner probes used a powerful Reed-Muller code capable of correcting 7 errors out of 32 bits transmitted.
- The codewords consisted of 6 data bits and 26 check bits. The data was sent to Earth at the rate over 16,000 bits per second.
- Another application of error-correcting codes came with the development of the compact disk (CD).
- To guard against scratches, cracks and similar damage two "interleaved" codes which can correct up to 4,000 consecutive errors (about 2.5 mm of track) are used.

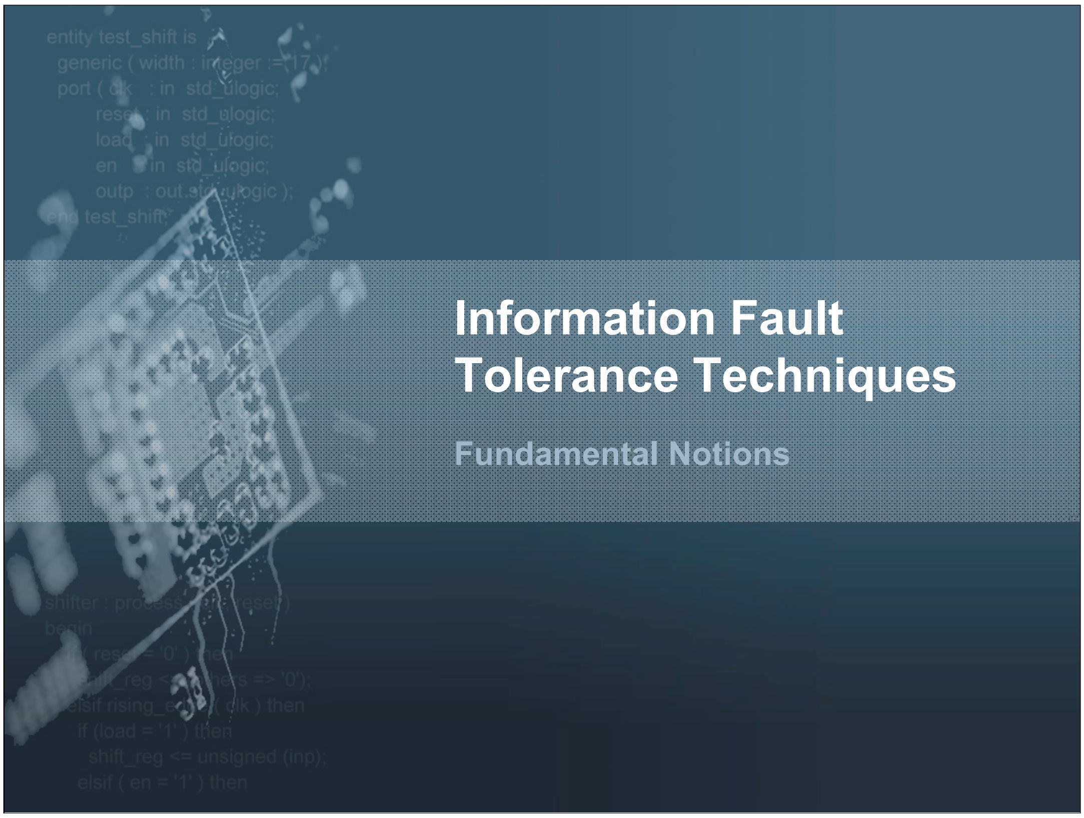


Introduction V

- Code selection is usually guided by the types of errors required to be tolerated and the overhead associated with each of the error detection techniques.

- For example, error correction is a common level of protection for minicomputers and mainframes whereas the cheaper error detection by parity code is more common in microcomputers.

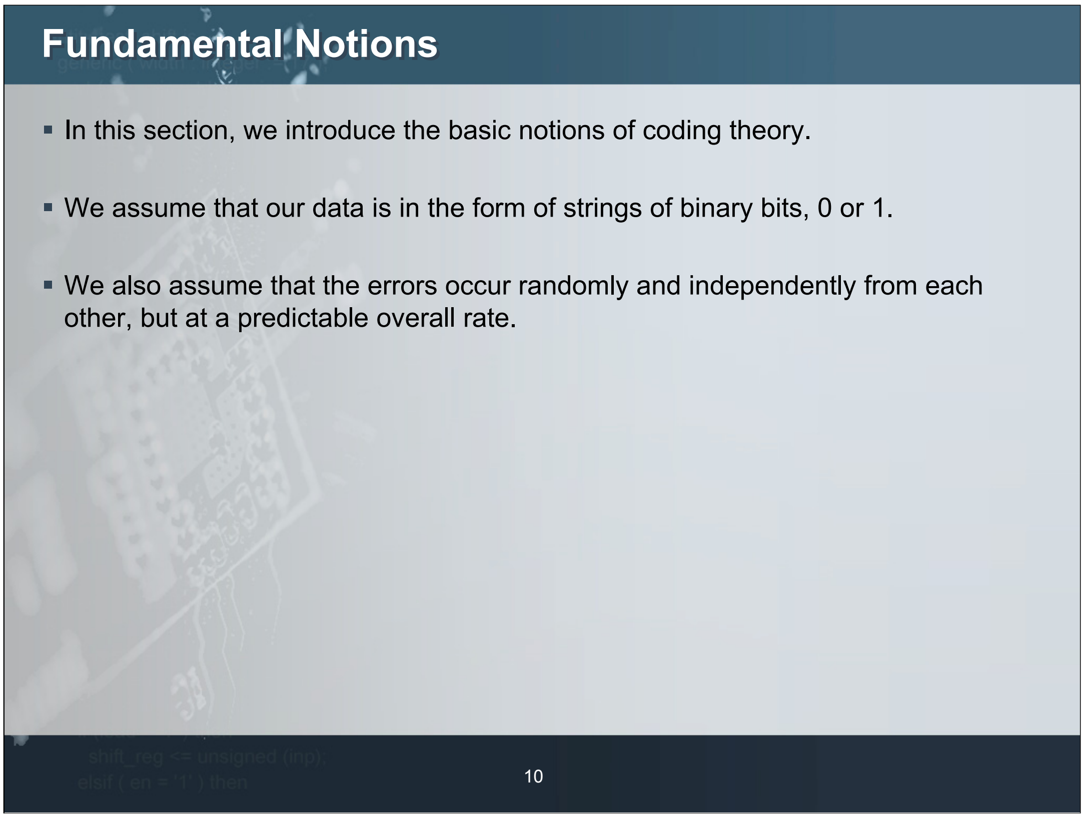
- For solid state disks, storing system's critical, non-recoverable files, the most popular codes are Hamming codes to correct errors in main memory, and Reed-Solomon codes to correct errors in peripheral devices such as tape and disk storage.



entity test_shift is
generic (width : integer := 17);
port (clk : in std_ulogic;
reset : in std_ulogic;
load : in std_ulogic;
en : in std_ulogic;
outp : out std_ulogic);
end test_shift;
shifter : process (clk, reset)
begin
if (reset = '0') then
shift_reg <= (others => '0');
elsif rising_edge (clk) then
if (load = '1') then
shift_reg <= unsigned (inp);
elsif (en = '1') then

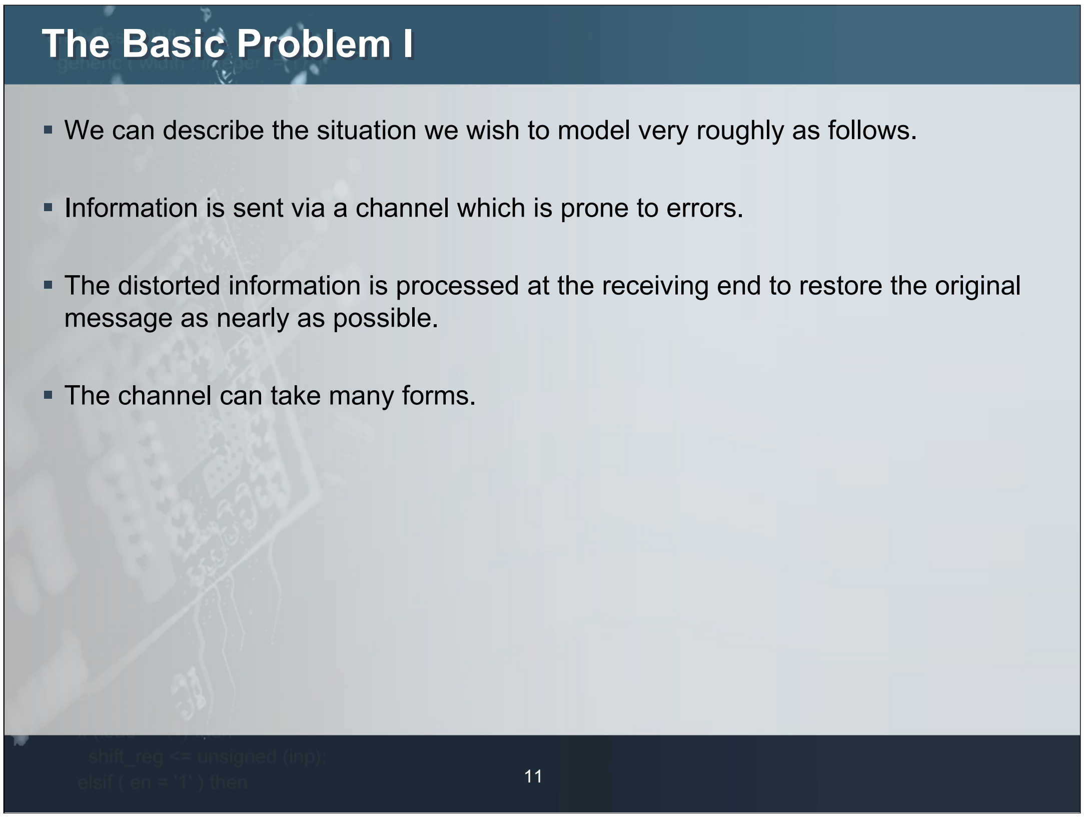
Information Fault Tolerance Techniques

Fundamental Notions



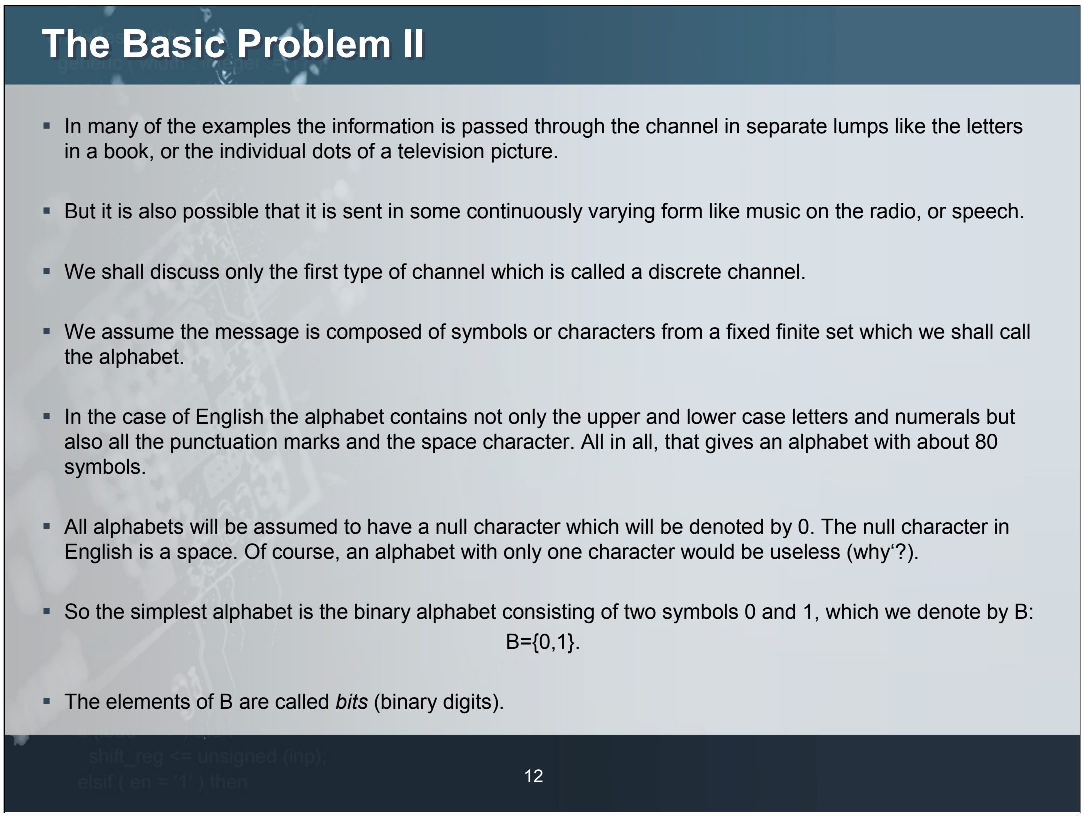
Fundamental Notions

- In this section, we introduce the basic notions of coding theory.
- We assume that our data is in the form of strings of binary bits, 0 or 1.
- We also assume that the errors occur randomly and independently from each other, but at a predictable overall rate.



The Basic Problem I

- We can describe the situation we wish to model very roughly as follows.
- Information is sent via a channel which is prone to errors.
- The distorted information is processed at the receiving end to restore the original message as nearly as possible.
- The channel can take many forms.

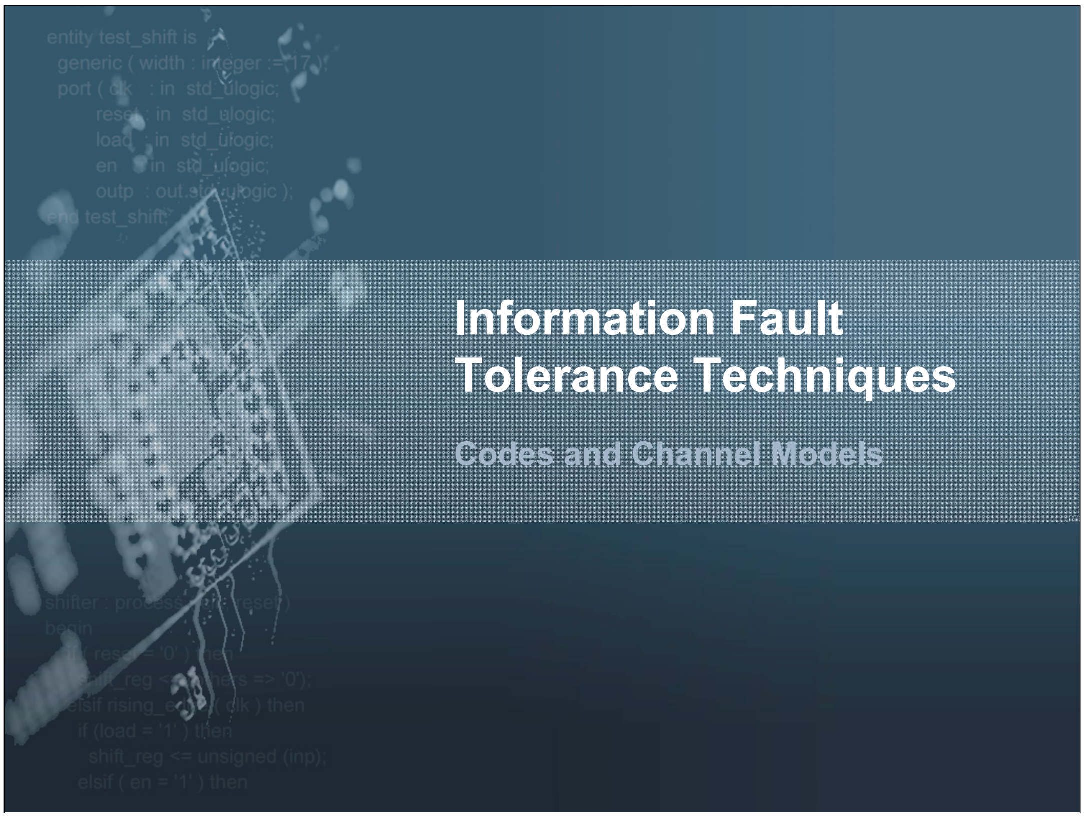


The Basic Problem II

- In many of the examples the information is passed through the channel in separate lumps like the letters in a book, or the individual dots of a television picture.
- But it is also possible that it is sent in some continuously varying form like music on the radio, or speech.
- We shall discuss only the first type of channel which is called a discrete channel.
- We assume the message is composed of symbols or characters from a fixed finite set which we shall call the alphabet.
- In the case of English the alphabet contains not only the upper and lower case letters and numerals but also all the punctuation marks and the space character. All in all, that gives an alphabet with about 80 symbols.
- All alphabets will be assumed to have a null character which will be denoted by 0. The null character in English is a space. Of course, an alphabet with only one character would be useless (why'?).
- So the simplest alphabet is the binary alphabet consisting of two symbols 0 and 1, which we denote by B:

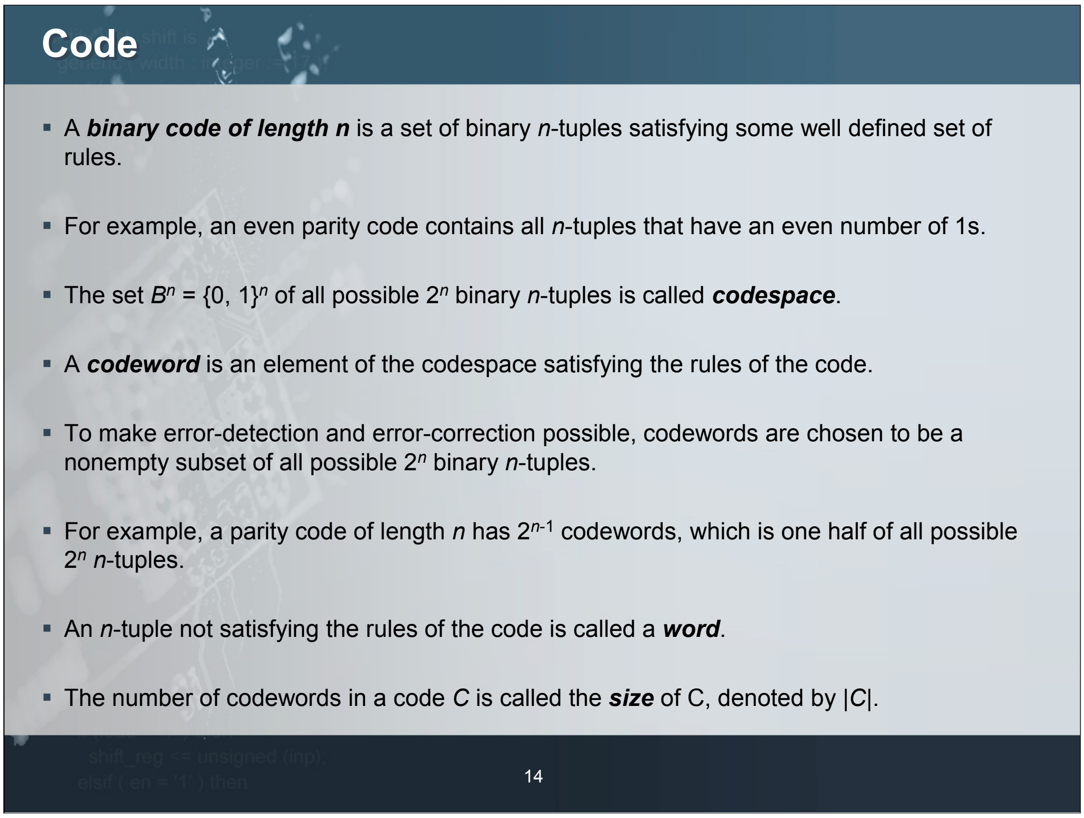
  $$B=\{0,1\}.$$

- The elements of B are called *bits* (binary digits).



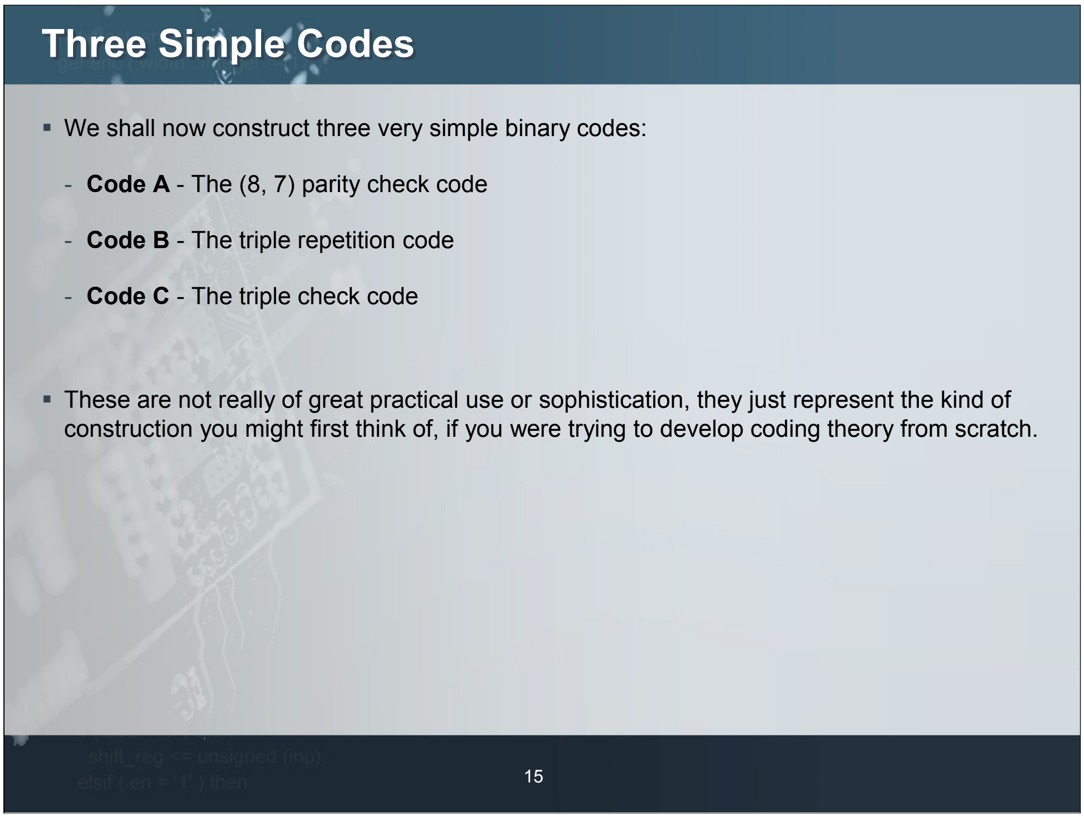
Information Fault Tolerance Techniques

Codes and Channel Models



Code

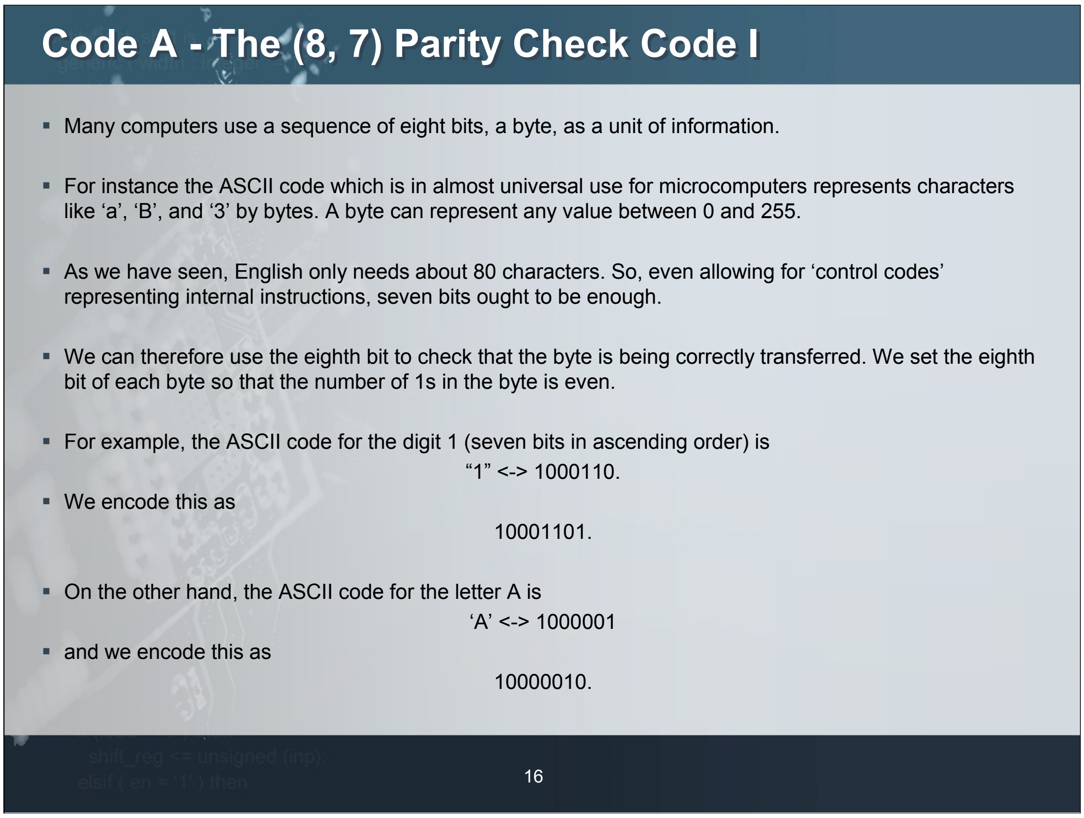
- A ***binary code of length n*** is a set of binary *n*-tuples satisfying some well defined set of rules.
- For example, an even parity code contains all *n*-tuples that have an even number of 1s.
- The set $B^n = \{0, 1\}^n$ of all possible 2^n binary *n*-tuples is called ***codespace***.
- A ***codeword*** is an element of the codespace satisfying the rules of the code.
- To make error-detection and error-correction possible, codewords are chosen to be a nonempty subset of all possible 2^n binary *n*-tuples.
- For example, a parity code of length *n* has 2^{n-1} codewords, which is one half of all possible 2^n *n*-tuples.
- An *n*-tuple not satisfying the rules of the code is called a ***word***.
- The number of codewords in a code *C* is called the ***size*** of C, denoted by $|C|$.



Three Simple Codes

- We shall now construct three very simple binary codes:
  - **Code A** - The (8, 7) parity check code
  - **Code B** - The triple repetition code
  - **Code C** - The triple check code

- These are not really of great practical use or sophistication, they just represent the kind of construction you might first think of, if you were trying to develop coding theory from scratch.



Code A - The (8, 7) Parity Check Code I

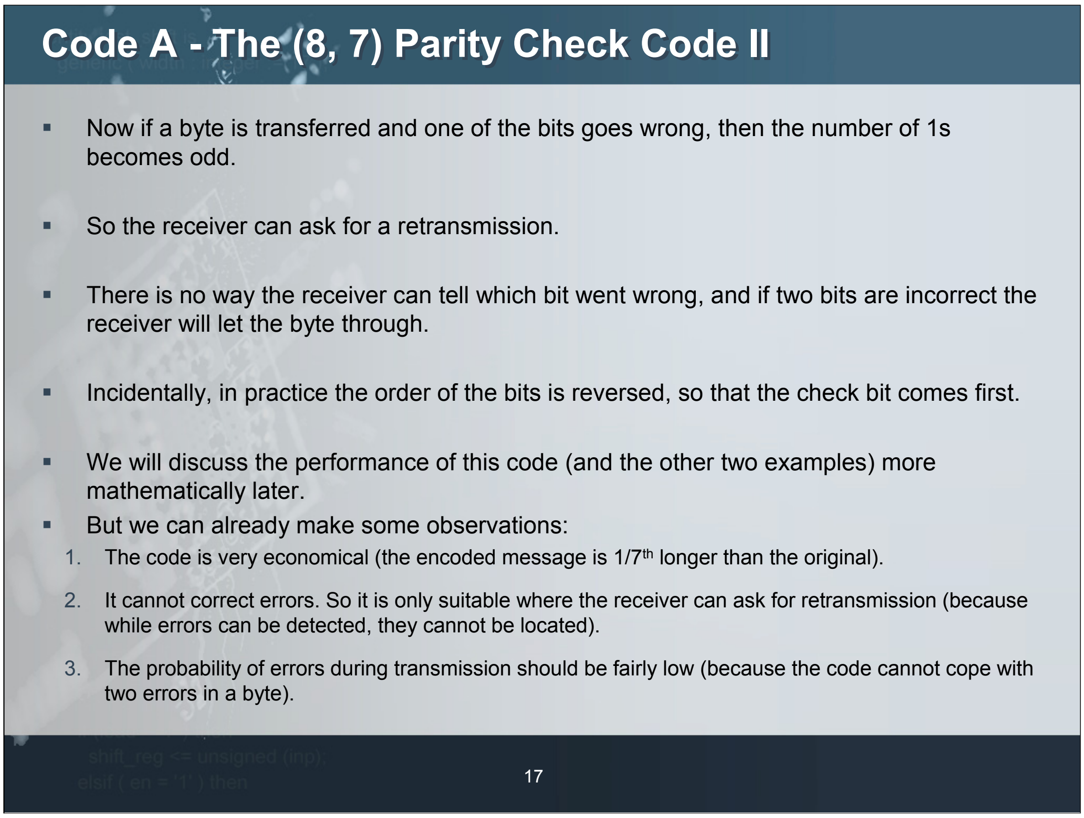
- Many computers use a sequence of eight bits, a byte, as a unit of information.
- For instance the ASCII code which is in almost universal use for microcomputers represents characters like ‘a’, ‘B’, and ‘3’ by bytes. A byte can represent any value between 0 and 255.
- As we have seen, English only needs about 80 characters. So, even allowing for ‘control codes’ representing internal instructions, seven bits ought to be enough.
- We can therefore use the eighth bit to check that the byte is being correctly transferred. We set the eighth bit of each byte so that the number of 1s in the byte is even.
- For example, the ASCII code for the digit 1 (seven bits in ascending order) is

  “1” <-> 1000110.
- We encode this as

  10001101.
- On the other hand, the ASCII code for the letter A is

  ‘A’ <-> 1000001
- and we encode this as

  10000010.



Code A - The (8, 7) Parity Check Code II

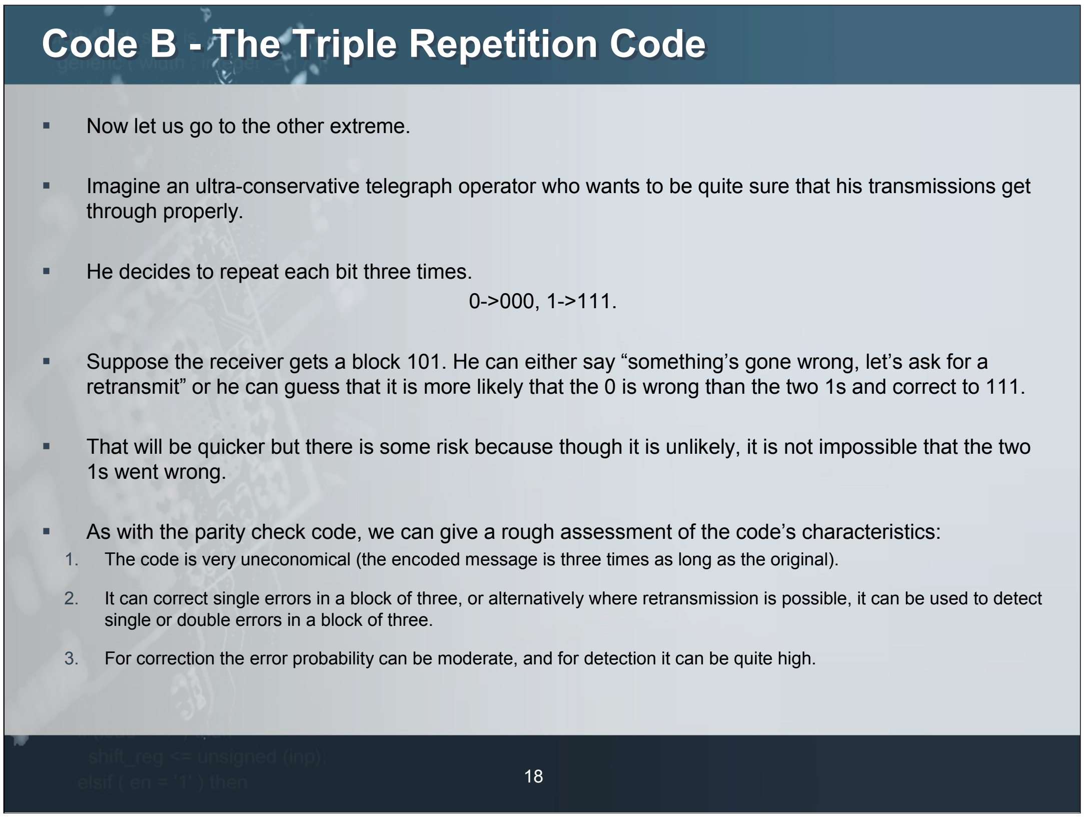
- Now if a byte is transferred and one of the bits goes wrong, then the number of 1s becomes odd.

- So the receiver can ask for a retransmission.

- There is no way the receiver can tell which bit went wrong, and if two bits are incorrect the receiver will let the byte through.

- Incidentally, in practice the order of the bits is reversed, so that the check bit comes first.

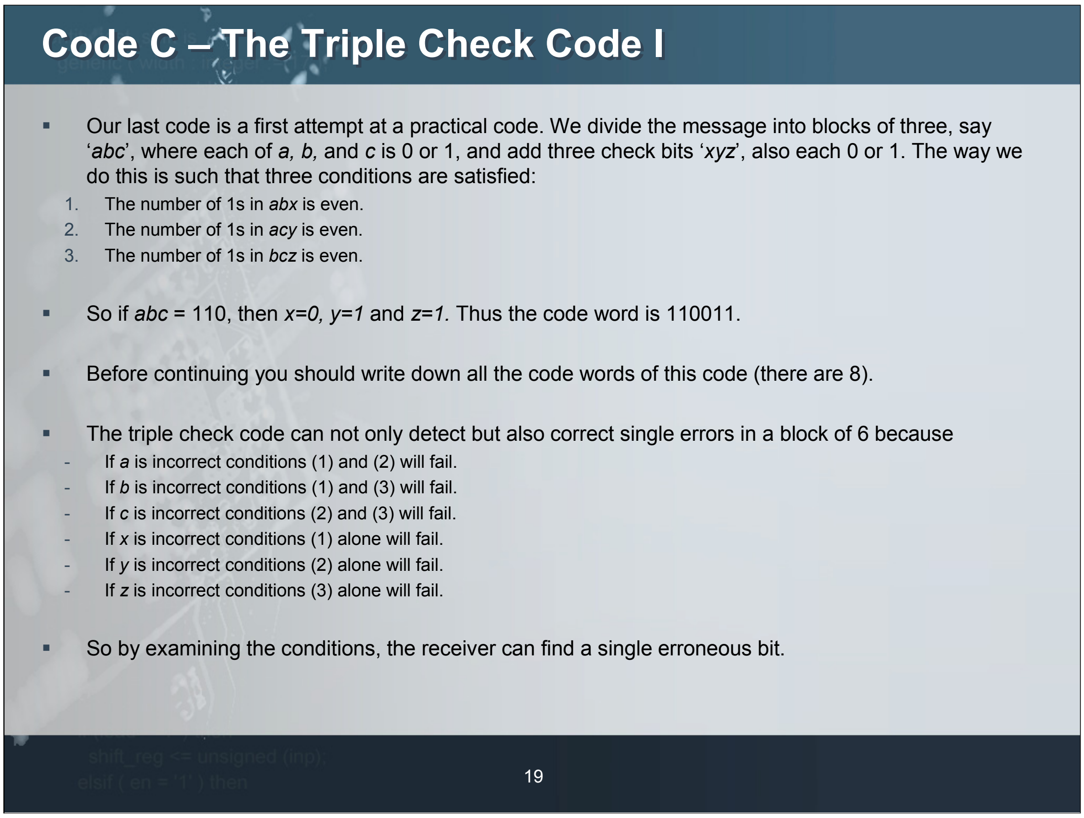
- We will discuss the performance of this code (and the other two examples) more mathematically later.
- But we can already make some observations:
  1. The code is very economical (the encoded message is 1/7th longer than the original).
  2. It cannot correct errors. So it is only suitable where the receiver can ask for retransmission (because while errors can be detected, they cannot be located).
  3. The probability of errors during transmission should be fairly low (because the code cannot cope with two errors in a byte).



Code B - The Triple Repetition Code

- Now let us go to the other extreme.
- Imagine an ultra-conservative telegraph operator who wants to be quite sure that his transmissions get through properly.
- He decides to repeat each bit three times.

  0->000, 1->111.
- Suppose the receiver gets a block 101. He can either say “something’s gone wrong, let’s ask for a retransmit” or he can guess that it is more likely that the 0 is wrong than the two 1s and correct to 111.
- That will be quicker but there is some risk because though it is unlikely, it is not impossible that the two 1s went wrong.
- As with the parity check code, we can give a rough assessment of the code’s characteristics:
  1. The code is very uneconomical (the encoded message is three times as long as the original).
  2. It can correct single errors in a block of three, or alternatively where retransmission is possible, it can be used to detect single or double errors in a block of three.
  3. For correction the error probability can be moderate, and for detection it can be quite high.



Code C – The Triple Check Code I

- Our last code is a first attempt at a practical code. We divide the message into blocks of three, say '*abc*', where each of *a, b,* and *c* is 0 or 1, and add three check bits '*xyz*', also each 0 or 1. The way we do this is such that three conditions are satisfied:

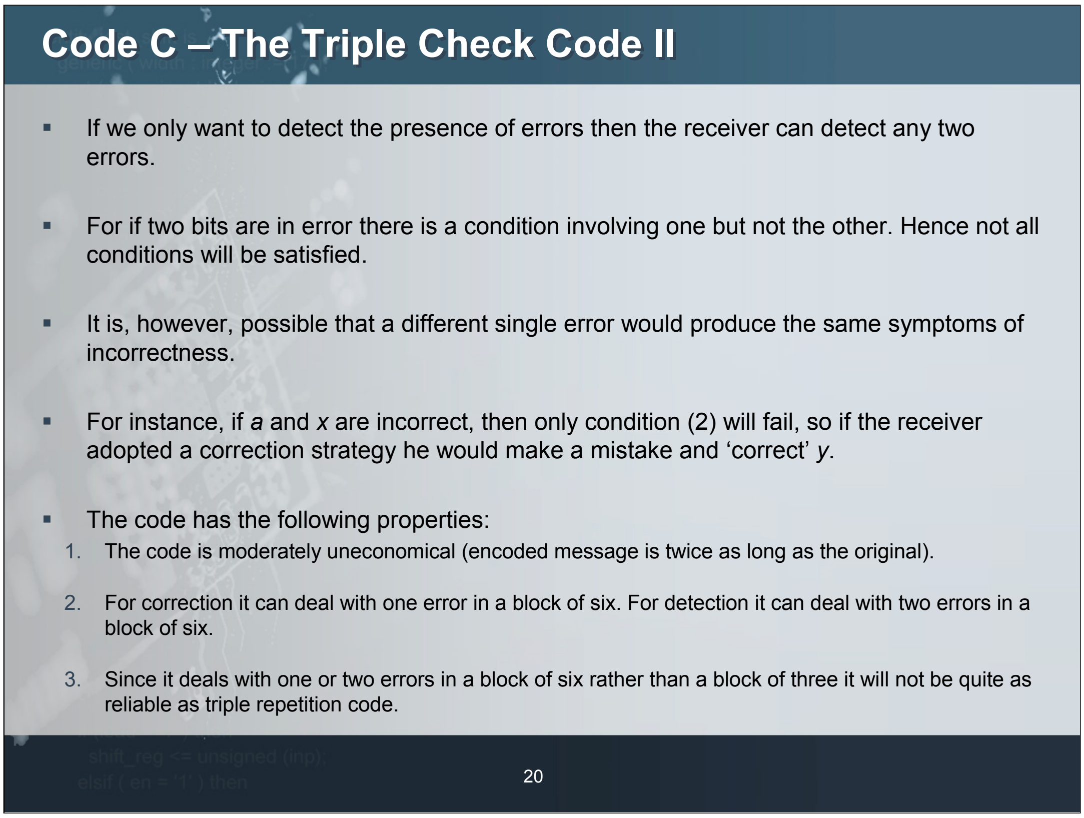
1. The number of 1s in *abx* is even.
2. The number of 1s in *acy* is even.
3. The number of 1s in *bcz* is even.

- So if *abc* = 110, then *x=0, y=1* and *z=1.* Thus the code word is 110011.

- Before continuing you should write down all the code words of this code (there are 8).

- The triple check code can not only detect but also correct single errors in a block of 6 because
  - If *a* is incorrect conditions (1) and (2) will fail.
  - If *b* is incorrect conditions (1) and (3) will fail.
  - If *c* is incorrect conditions (2) and (3) will fail.
  - If *x* is incorrect conditions (1) alone will fail.
  - If *y* is incorrect conditions (2) alone will fail.
  - If *z* is incorrect conditions (3) alone will fail.

- So by examining the conditions, the receiver can find a single erroneous bit.



Code C – The Triple Check Code II

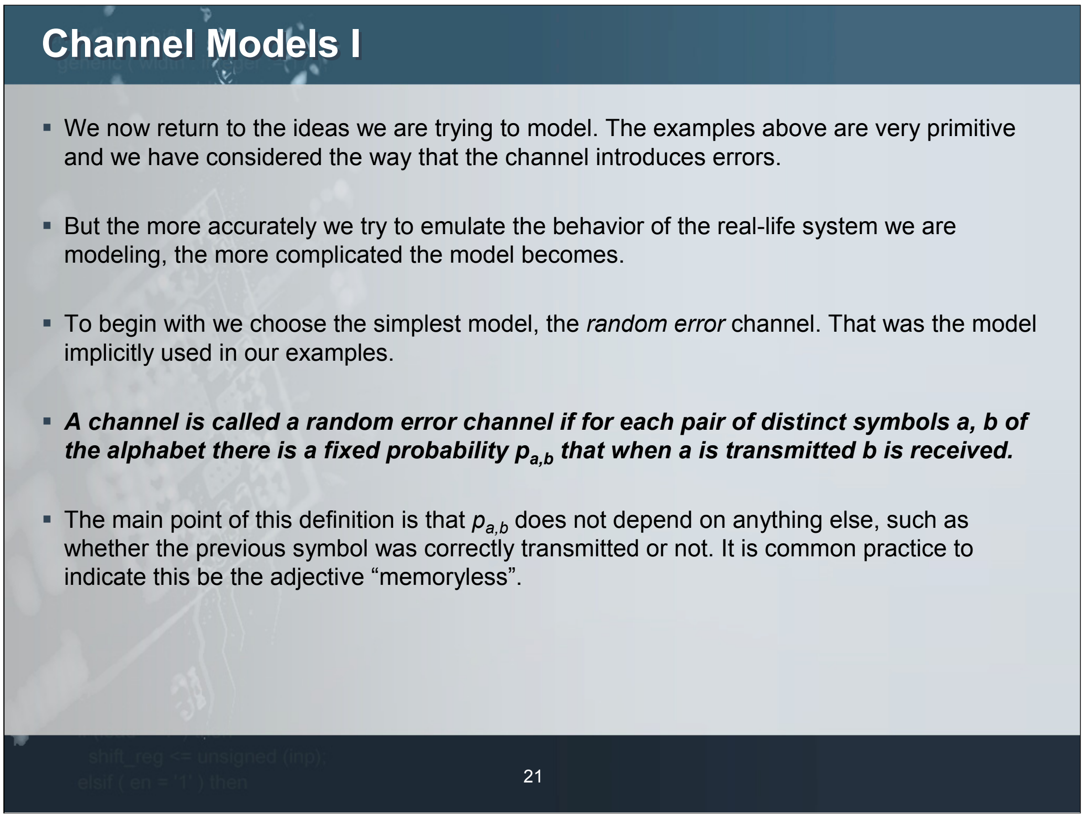
- If we only want to detect the presence of errors then the receiver can detect any two errors.

- For if two bits are in error there is a condition involving one but not the other. Hence not all conditions will be satisfied.

- It is, however, possible that a different single error would produce the same symptoms of incorrectness.

- For instance, if *a* and *x* are incorrect, then only condition (2) will fail, so if the receiver adopted a correction strategy he would make a mistake and 'correct' *y*.

- The code has the following properties:
  1. The code is moderately uneconomical (encoded message is twice as long as the original).
  2. For correction it can deal with one error in a block of six. For detection it can deal with two errors in a block of six.
  3. Since it deals with one or two errors in a block of six rather than a block of three it will not be quite as reliable as triple repetition code.



Channel Models I

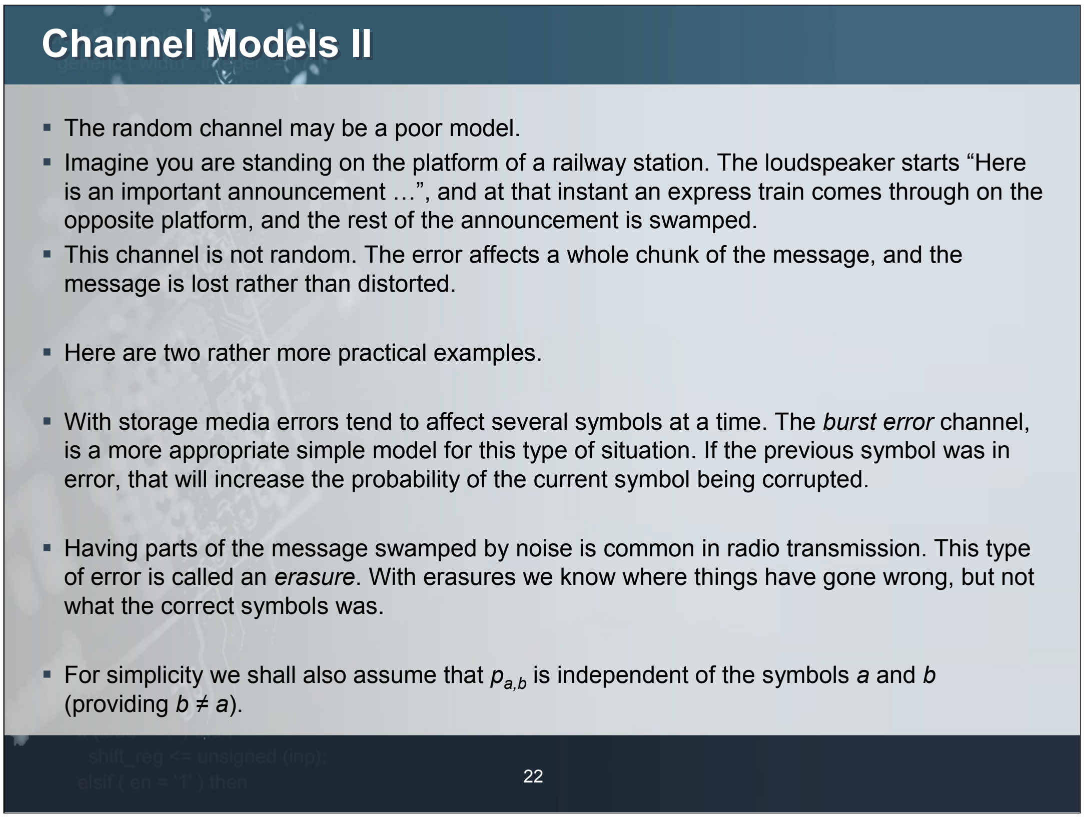
- We now return to the ideas we are trying to model. The examples above are very primitive and we have considered the way that the channel introduces errors.

- But the more accurately we try to emulate the behavior of the real-life system we are modeling, the more complicated the model becomes.

- To begin with we choose the simplest model, the *random error* channel. That was the model implicitly used in our examples.

- ***A channel is called a random error channel if for each pair of distinct symbols a, b of the alphabet there is a fixed probability $p_{a,b}$ that when a is transmitted b is received.***

- The main point of this definition is that $p_{a,b}$ does not depend on anything else, such as whether the previous symbol was correctly transmitted or not. It is common practice to indicate this be the adjective "memoryless".



Channel Models II

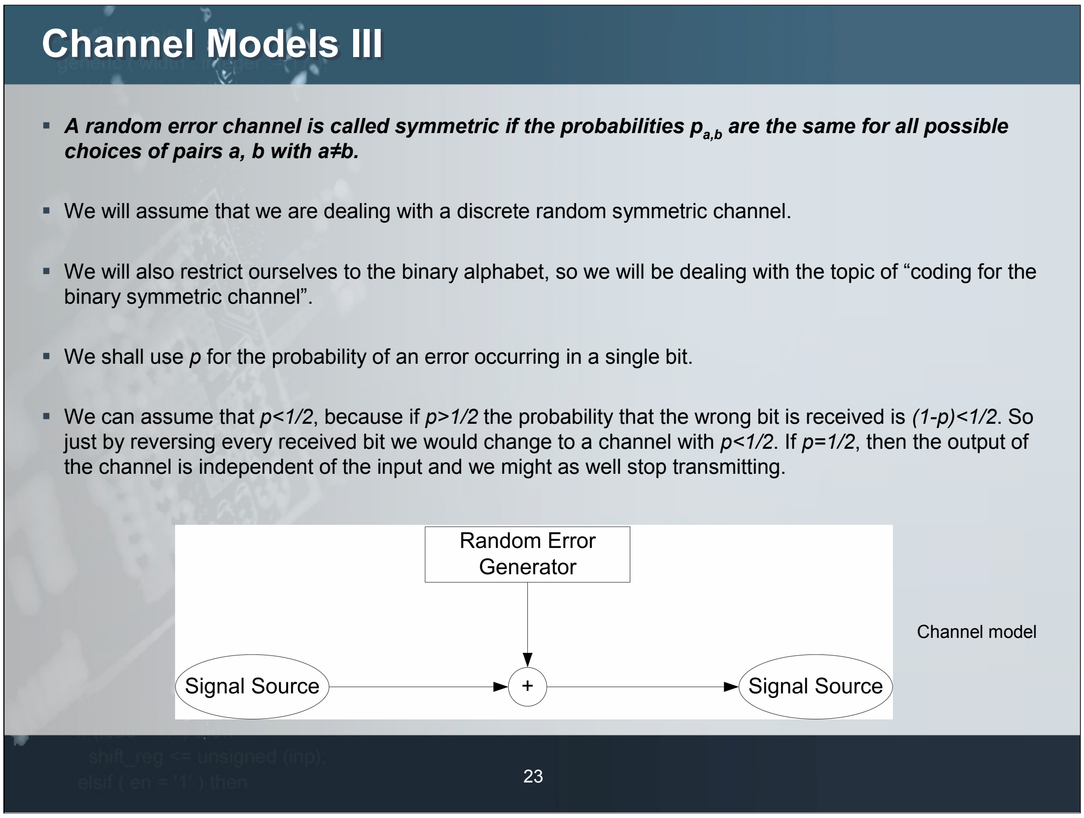
- The random channel may be a poor model.
- Imagine you are standing on the platform of a railway station. The loudspeaker starts “Here is an important announcement …”, and at that instant an express train comes through on the opposite platform, and the rest of the announcement is swamped.
- This channel is not random. The error affects a whole chunk of the message, and the message is lost rather than distorted.

- Here are two rather more practical examples.

- With storage media errors tend to affect several symbols at a time. The *burst error* channel, is a more appropriate simple model for this type of situation. If the previous symbol was in error, that will increase the probability of the current symbol being corrupted.

- Having parts of the message swamped by noise is common in radio transmission. This type of error is called an *erasure*. With erasures we know where things have gone wrong, but not what the correct symbols was.

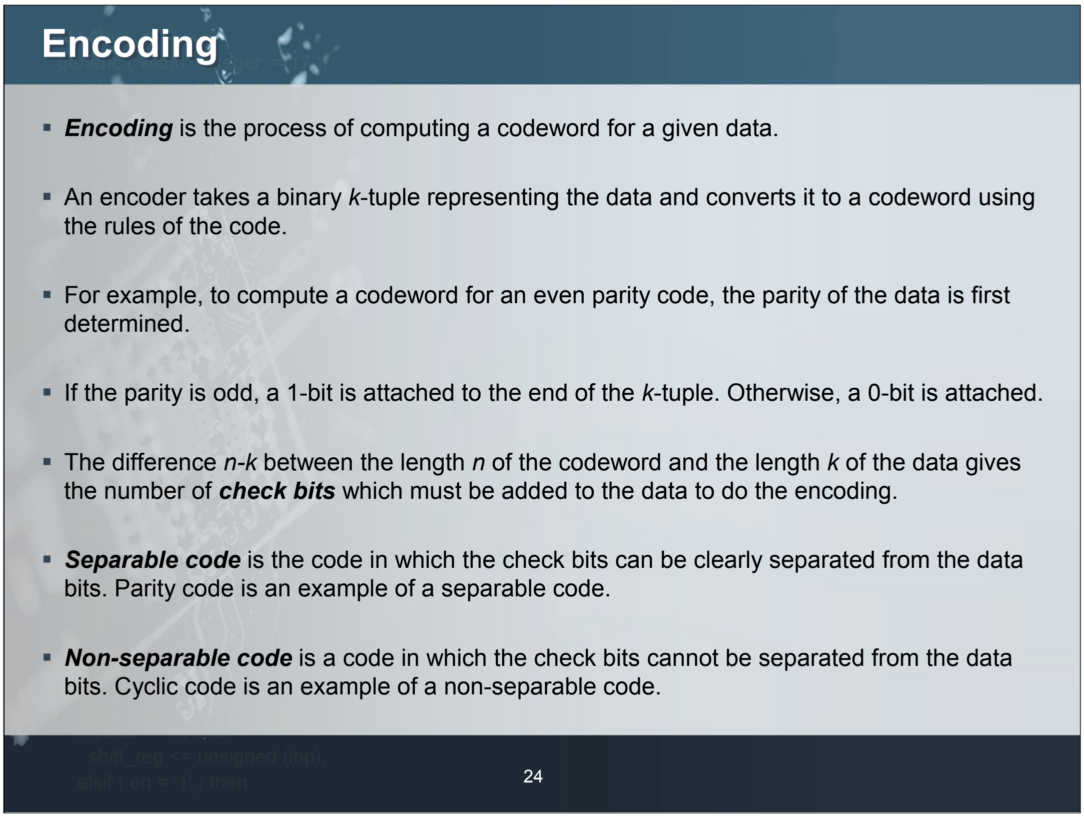
- For simplicity we shall also assume that $p_{a,b}$ is independent of the symbols a and b (providing $b \neq a$).



Channel Models III

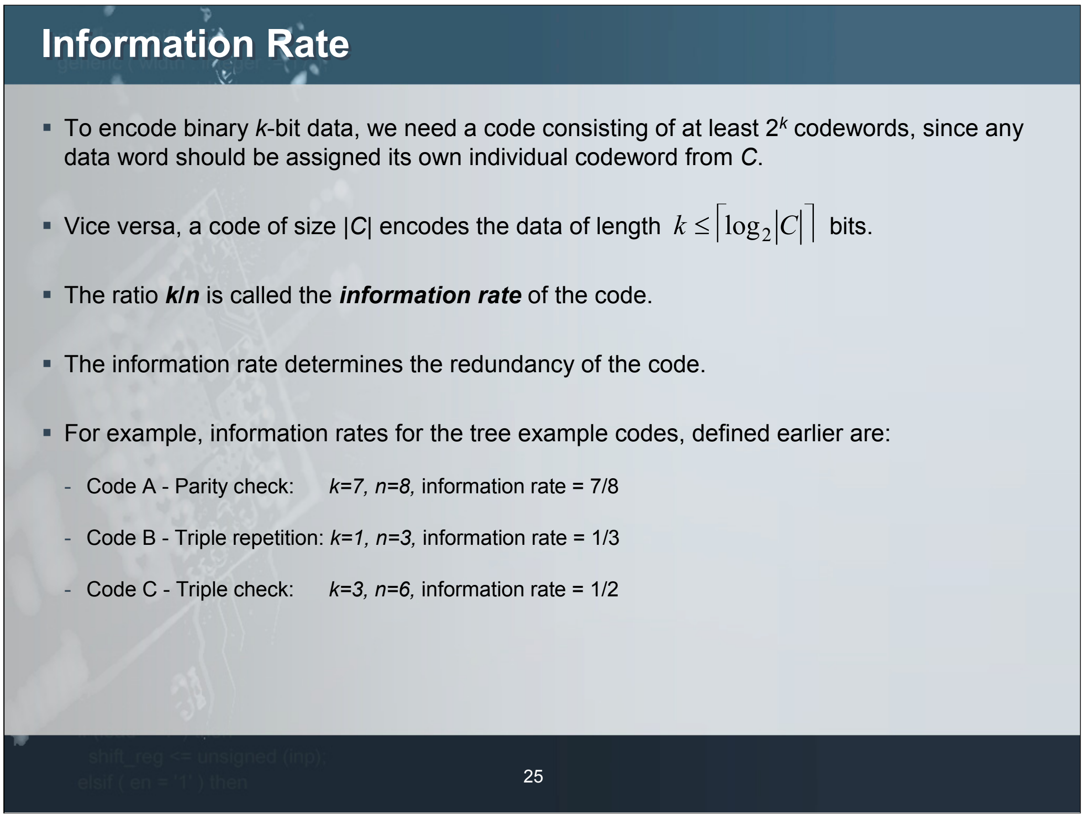
- ***A random error channel is called symmetric if the probabilities $p_{a,b}$ are the same for all possible choices of pairs a, b with a≠b.***
- We will assume that we are dealing with a discrete random symmetric channel.
- We will also restrict ourselves to the binary alphabet, so we will be dealing with the topic of “coding for the binary symmetric channel”.
- We shall use *p* for the probability of an error occurring in a single bit.
- We can assume that *p<1/2*, because if *p>1/2* the probability that the wrong bit is received is *(1-p)<1/2*. So just by reversing every received bit we would change to a channel with *p<1/2*. If *p=1/2*, then the output of the channel is independent of the input and we might as well stop transmitting.

Channel model



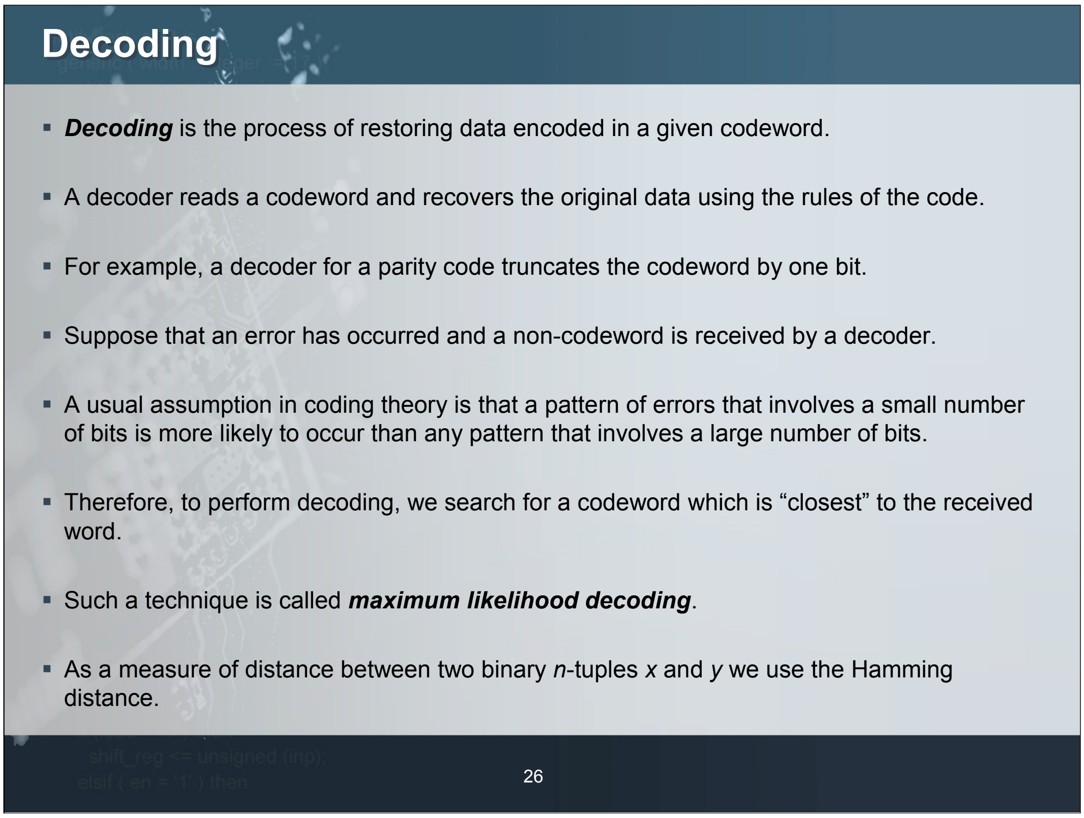
Encoding

- ***Encoding*** is the process of computing a codeword for a given data.
- An encoder takes a binary *k*-tuple representing the data and converts it to a codeword using the rules of the code.
- For example, to compute a codeword for an even parity code, the parity of the data is first determined.
- If the parity is odd, a 1-bit is attached to the end of the *k*-tuple. Otherwise, a 0-bit is attached.
- The difference *n*-*k* between the length *n* of the codeword and the length *k* of the data gives the number of ***check bits*** which must be added to the data to do the encoding.
- ***Separable code*** is the code in which the check bits can be clearly separated from the data bits. Parity code is an example of a separable code.
- ***Non-separable code*** is a code in which the check bits cannot be separated from the data bits. Cyclic code is an example of a non-separable code.



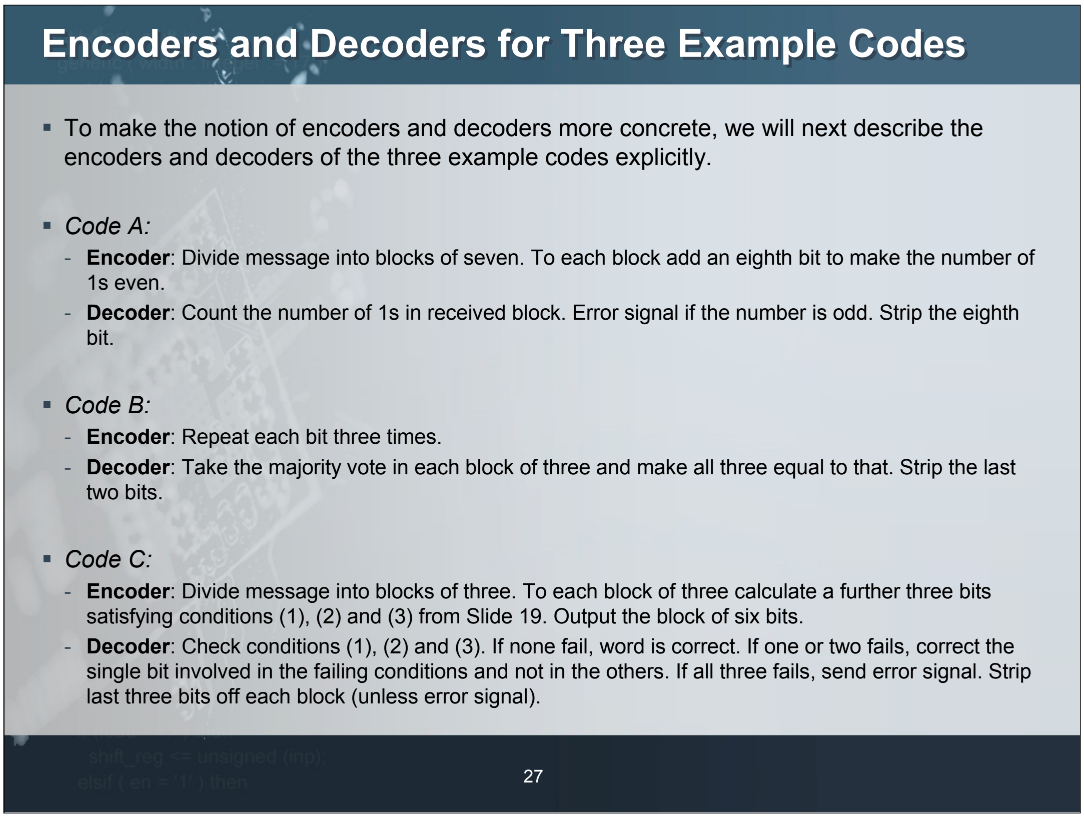
Information Rate

- To encode binary *k*-bit data, we need a code consisting of at least 2^k codewords, since any data word should be assigned its own individual codeword from *C*.
- Vice versa, a code of size |*C*| encodes the data of length $k \le \lceil \log_2 |C| \rceil$ bits.
- The ratio ***k*/*n*** is called the ***information rate*** of the code.
- The information rate determines the redundancy of the code.
- For example, information rates for the tree example codes, defined earlier are:
  - Code A - Parity check: *k=7, n=8,* information rate = 7/8
  - Code B - Triple repetition: *k=1, n=3,* information rate = 1/3
  - Code C - Triple check: *k=3, n=6,* information rate = 1/2



Decoding

- ***Decoding*** is the process of restoring data encoded in a given codeword.
- A decoder reads a codeword and recovers the original data using the rules of the code.
- For example, a decoder for a parity code truncates the codeword by one bit.
- Suppose that an error has occurred and a non-codeword is received by a decoder.
- A usual assumption in coding theory is that a pattern of errors that involves a small number of bits is more likely to occur than any pattern that involves a large number of bits.
- Therefore, to perform decoding, we search for a codeword which is “closest” to the received word.
- Such a technique is called ***maximum likelihood decoding***.
- As a measure of distance between two binary n-tuples x and y we use the Hamming distance.



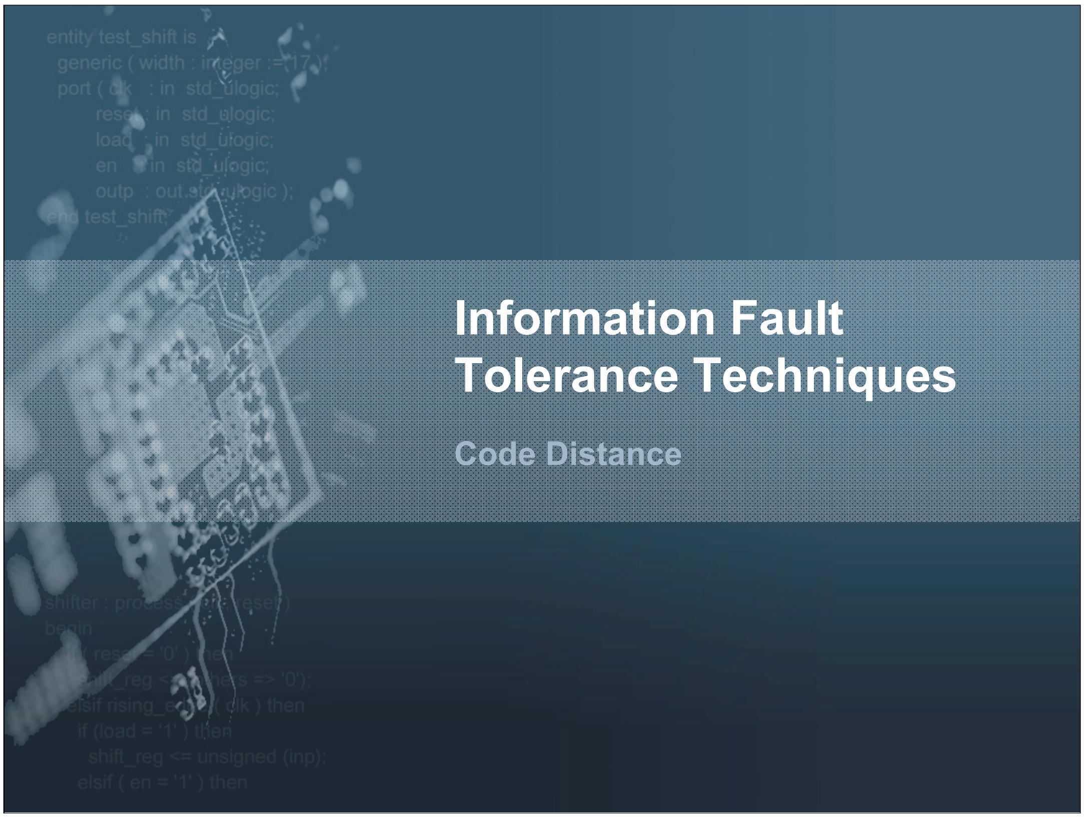
Encoders and Decoders for Three Example Codes

- To make the notion of encoders and decoders more concrete, we will next describe the encoders and decoders of the three example codes explicitly.

- *Code A:*
  - **Encoder**: Divide message into blocks of seven. To each block add an eighth bit to make the number of 1s even.
  - **Decoder**: Count the number of 1s in received block. Error signal if the number is odd. Strip the eighth bit.

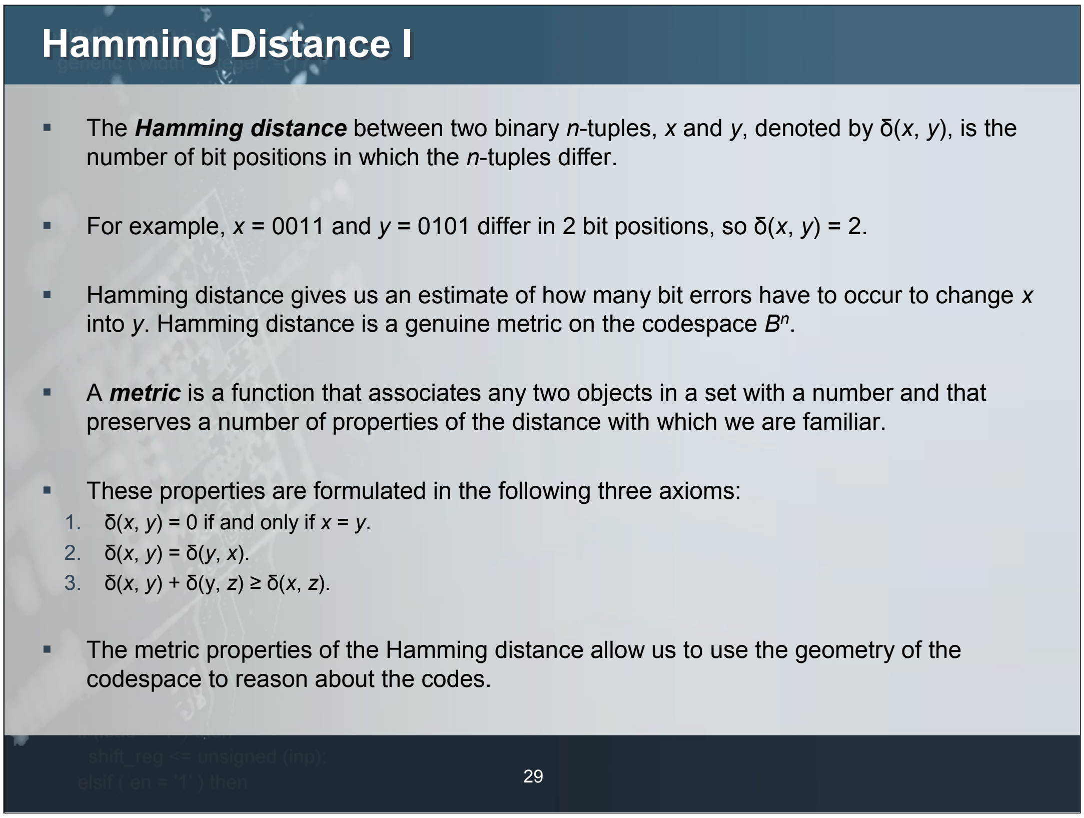
- *Code B:*
  - **Encoder**: Repeat each bit three times.
  - **Decoder**: Take the majority vote in each block of three and make all three equal to that. Strip the last two bits.

- *Code C:*
  - **Encoder**: Divide message into blocks of three. To each block of three calculate a further three bits satisfying conditions (1), (2) and (3) from Slide 19. Output the block of six bits.
  - **Decoder**: Check conditions (1), (2) and (3). If none fail, word is correct. If one or two fails, correct the single bit involved in the failing conditions and not in the others. If all three fails, send error signal. Strip last three bits off each block (unless error signal).



Information Fault Tolerance Techniques

Code Distance



Hamming Distance I

- The ***Hamming distance*** between two binary *n*-tuples, *x* and *y*, denoted by δ(*x*, *y*), is the number of bit positions in which the *n*-tuples differ.

- For example, *x* = 0011 and *y* = 0101 differ in 2 bit positions, so δ(*x*, *y*) = 2.

- Hamming distance gives us an estimate of how many bit errors have to occur to change *x* into *y*. Hamming distance is a genuine metric on the codespace B^n.

- A ***metric*** is a function that associates any two objects in a set with a number and that preserves a number of properties of the distance with which we are familiar.

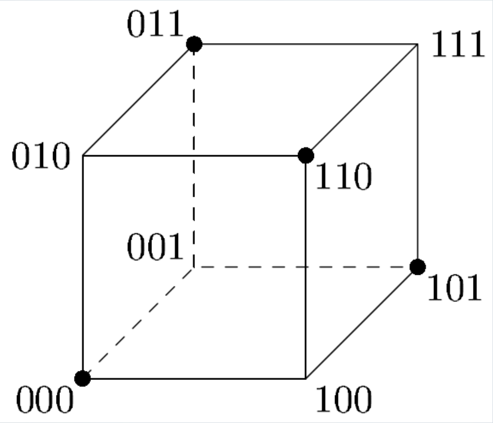
- These properties are formulated in the following three axioms:

1. δ(*x*, *y*) = 0 if and only if *x* = *y*.
2. δ(*x*, *y*) = δ(*y*, *x*).
3. δ(*x*, *y*) + δ(y, *z*) ≥ δ(*x*, *z*).

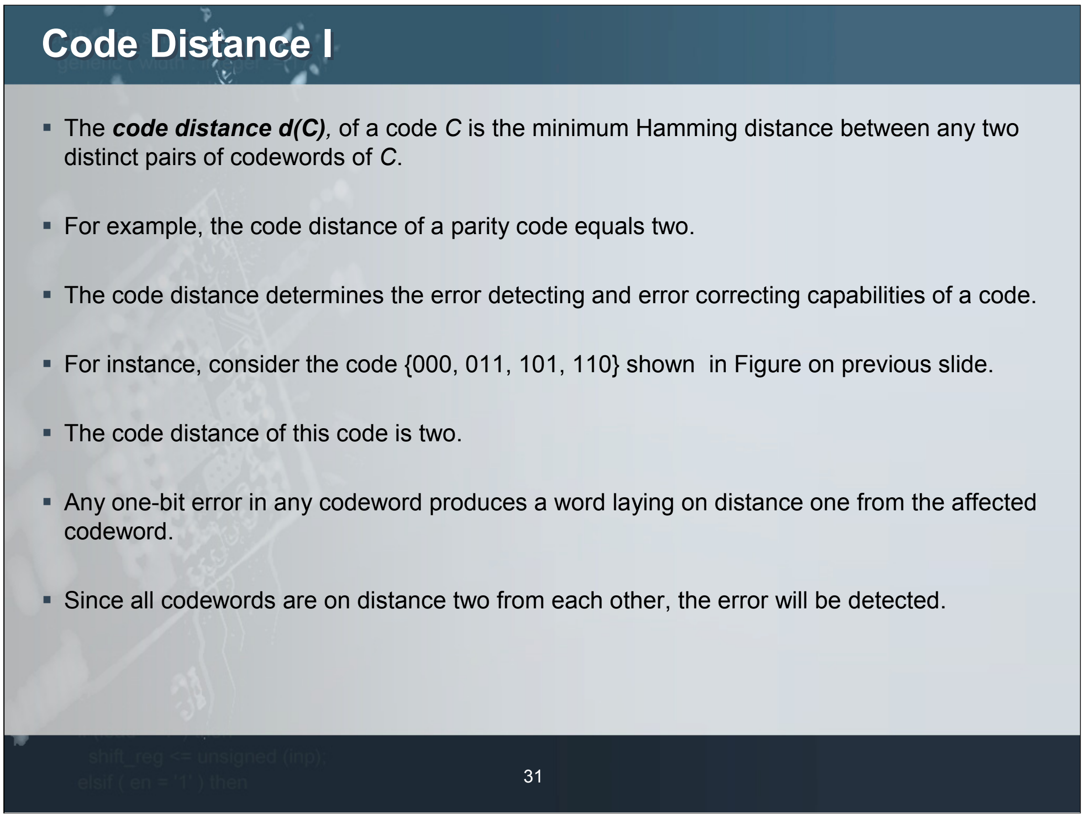
- The metric properties of the Hamming distance allow us to use the geometry of the codespace to reason about the codes.

Hamming Distance II

- As an example, consider the codespace B^3 presented by a three-dimensional cube shown in Figure on the right.
- Codewords {000 ,011, 101, 110} are marked with large solid dots.
- Adjacent vertices differ by a single bit.
- It is easy to see that the Hamming distance satisfies the metric properties listed above, e.g. δ(000, 011) + δ(011, 111) = 2+1 = 3 = δ(000, 111).

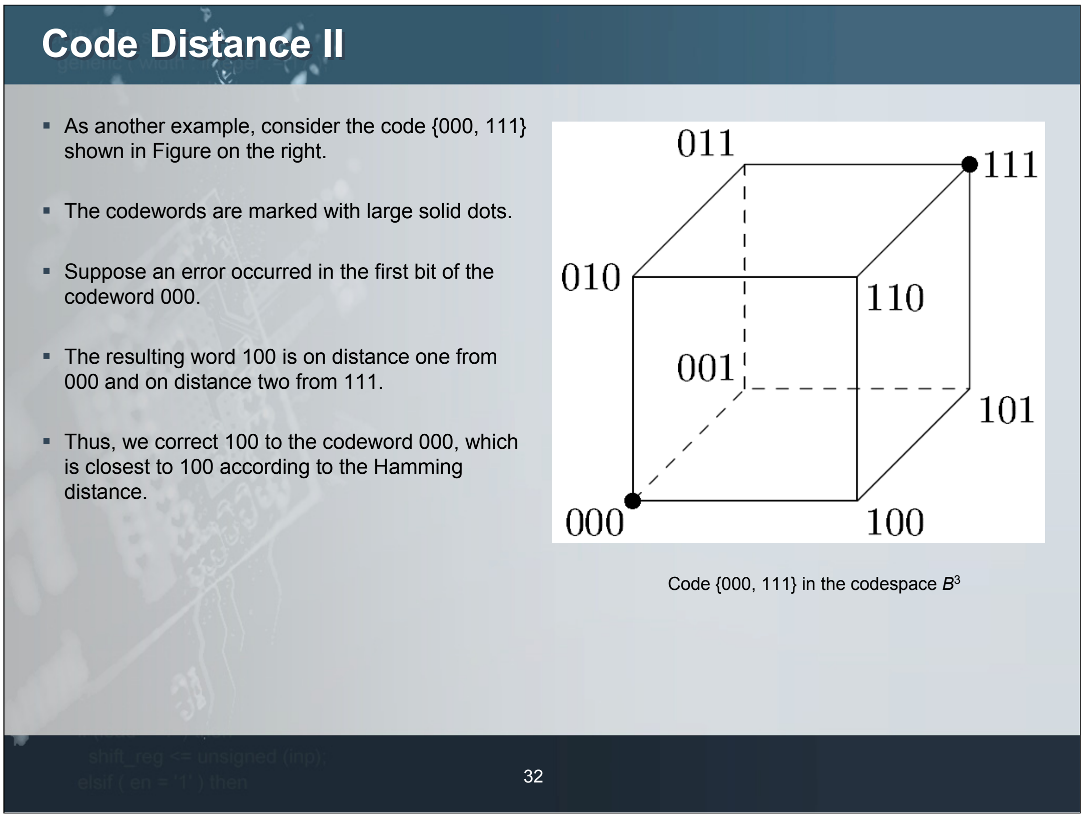


Code {000, 011, 101, 110} in the codespace B^3



Code Distance I

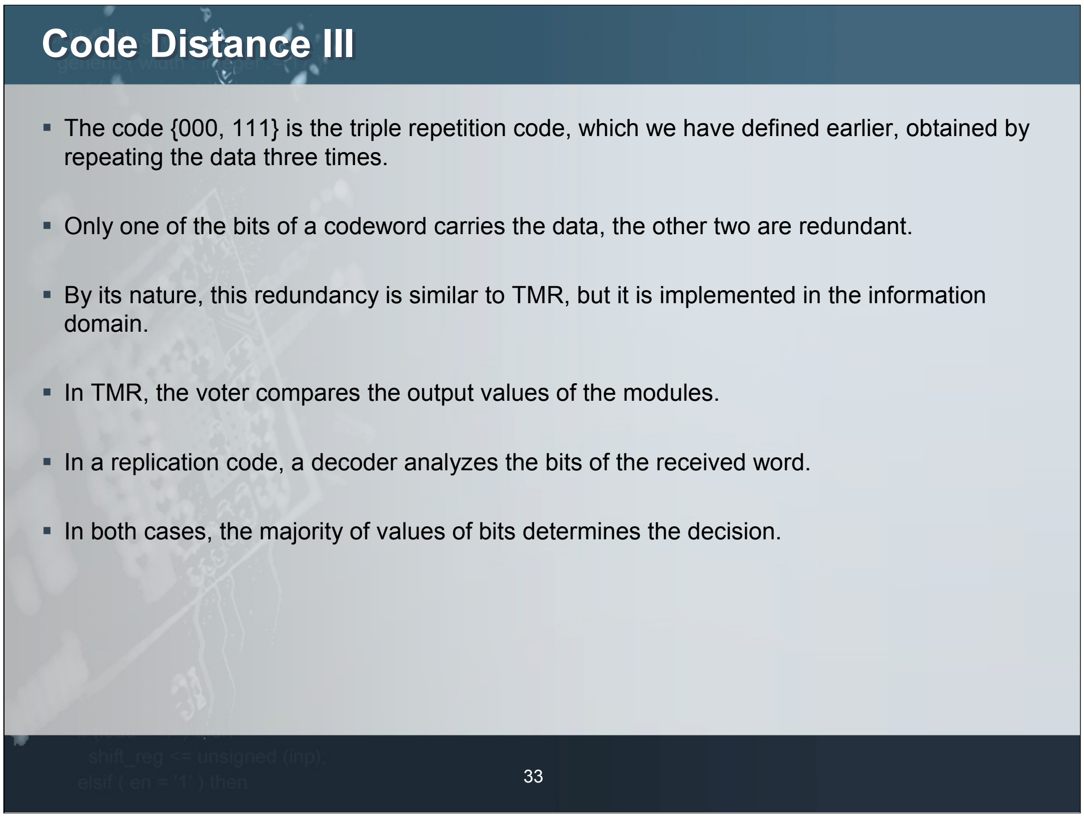
- The ***code distance d(C)****,* of a code *C* is the minimum Hamming distance between any two distinct pairs of codewords of *C*.
- For example, the code distance of a parity code equals two.
- The code distance determines the error detecting and error correcting capabilities of a code.
- For instance, consider the code {000, 011, 101, 110} shown in Figure on previous slide.
- The code distance of this code is two.
- Any one-bit error in any codeword produces a word laying on distance one from the affected codeword.
- Since all codewords are on distance two from each other, the error will be detected.



Code Distance II

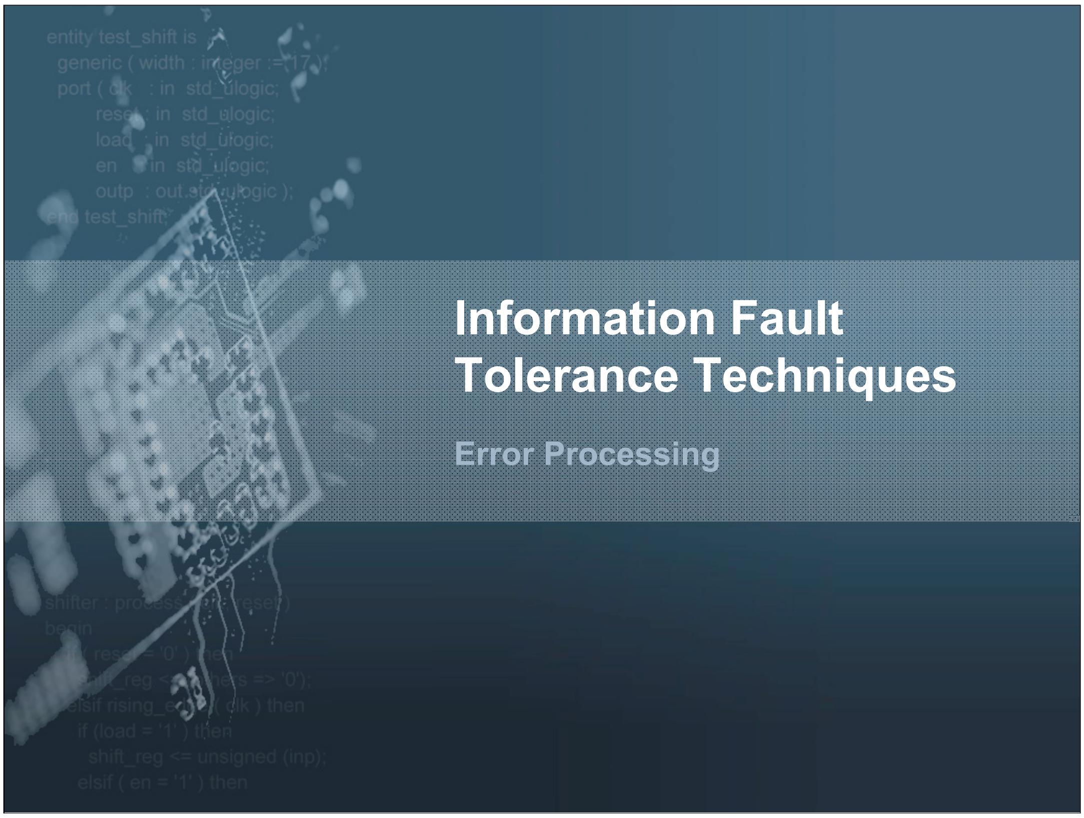
- As another example, consider the code {000, 111} shown in Figure on the right.
- The codewords are marked with large solid dots.
- Suppose an error occurred in the first bit of the codeword 000.
- The resulting word 100 is on distance one from 000 and on distance two from 111.
- Thus, we correct 100 to the codeword 000, which is closest to 100 according to the Hamming distance.

Code {000, 111} in the codespace B^3



Code Distance III

- The code {000, 111} is the triple repetition code, which we have defined earlier, obtained by repeating the data three times.
- Only one of the bits of a codeword carries the data, the other two are redundant.
- By its nature, this redundancy is similar to TMR, but it is implemented in the information domain.
- In TMR, the voter compares the output values of the modules.
- In a replication code, a decoder analyzes the bits of the received word.
- In both cases, the majority of values of bits determines the decision.



Information Fault Tolerance Techniques

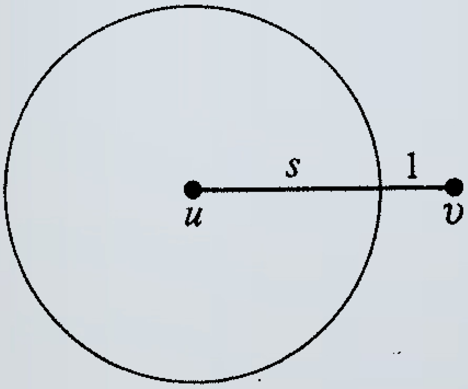
Error Processing

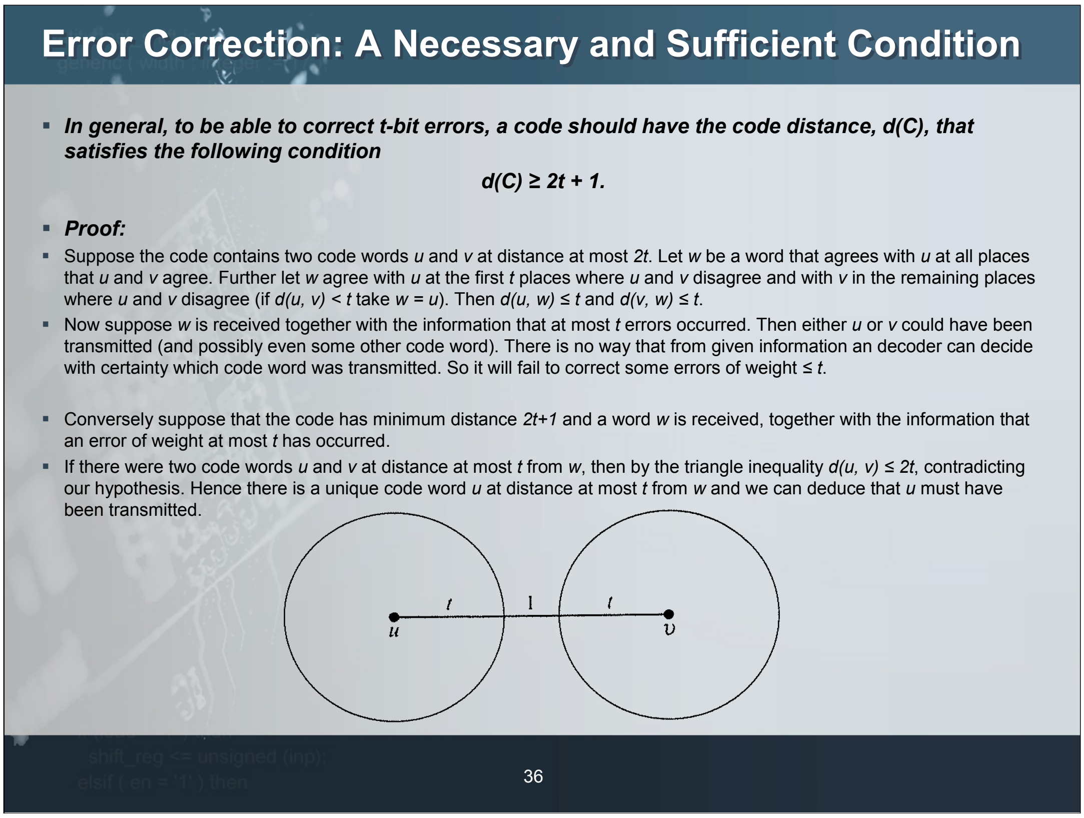
Error Detection: A Necessary and Sufficient Condition

- ***To be able to detect all errors of weight ≤s, the code distance, d(C), should satisfy the following condition***

  $$d(C) \geq s + 1.$$

- ***Proof:***
- If two code words *u* and *v* are at distance at most *s*, then one can be distorted into the other by an error of weight at most *s*.
- In that case there is no way to detected all errors of weight at most *s*.
- Conversely if any two code words are at distance at least *s+1*, then any error of weight *s* will distort a code word into a non-code word.
- An error detector can check whether a received word is a code word or not (for instance by looking it up in a table).
- Hence all errors of weight at most *s* are detectable.



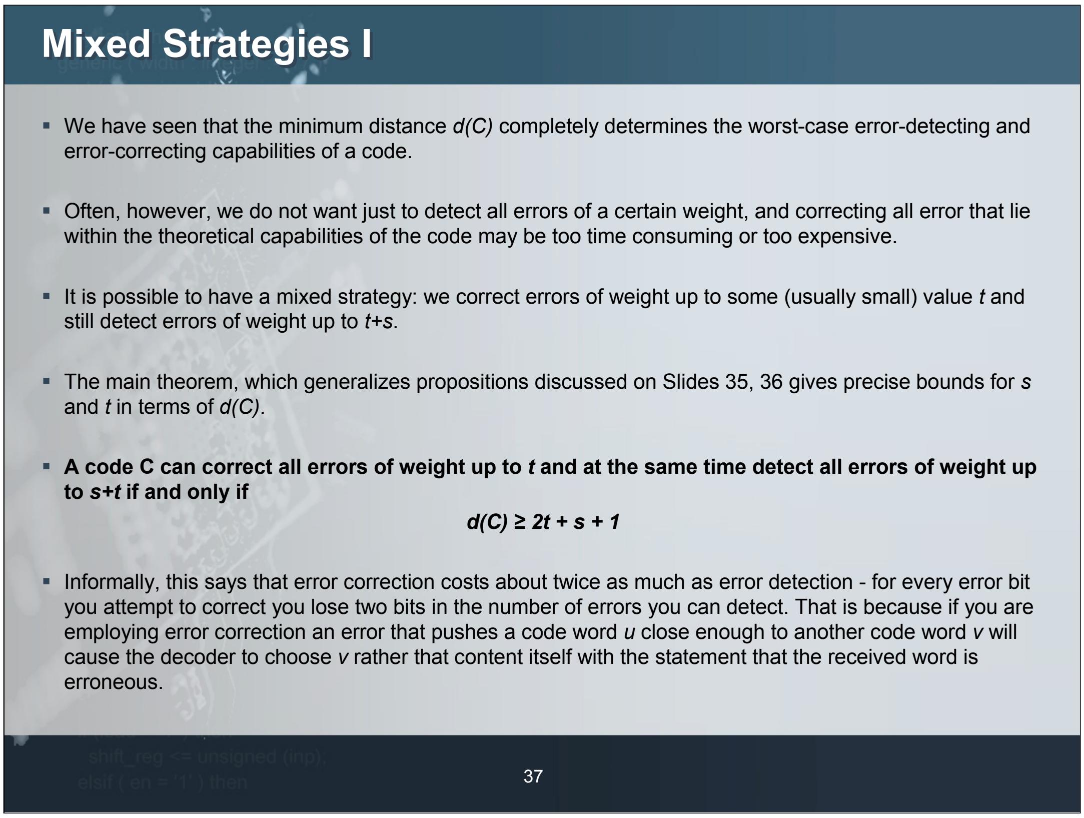


Error Correction: A Necessary and Sufficient Condition

- ***In general, to be able to correct t-bit errors, a code should have the code distance, d(C), that satisfies the following condition***

$$d(C) \geq 2t + 1.$$

- ***Proof:***
- Suppose the code contains two code words *u* and *v* at distance at most *2t*. Let *w* be a word that agrees with *u* at all places that *u* and *v* agree. Further let *w* agree with *u* at the first *t* places where *u* and *v* disagree and with *v* in the remaining places where *u* and *v* disagree (if $d(u, v) < t$ take $w = u$). Then $d(u, w) \leq t$ and $d(v, w) \leq t$.
- Now suppose *w* is received together with the information that at most *t* errors occurred. Then either *u* or *v* could have been transmitted (and possibly even some other code word). There is no way that from given information an decoder can decide with certainty which code word was transmitted. So it will fail to correct some errors of weight ≤ *t*.
- Conversely suppose that the code has minimum distance *2t+1* and a word *w* is received, together with the information that an error of weight at most *t* has occurred.
- If there were two code words *u* and *v* at distance at most *t* from *w*, then by the triangle inequality $d(u, v) \leq 2t$, contradicting our hypothesis. Hence there is a unique code word *u* at distance at most *t* from *w* and we can deduce that *u* must have been transmitted.

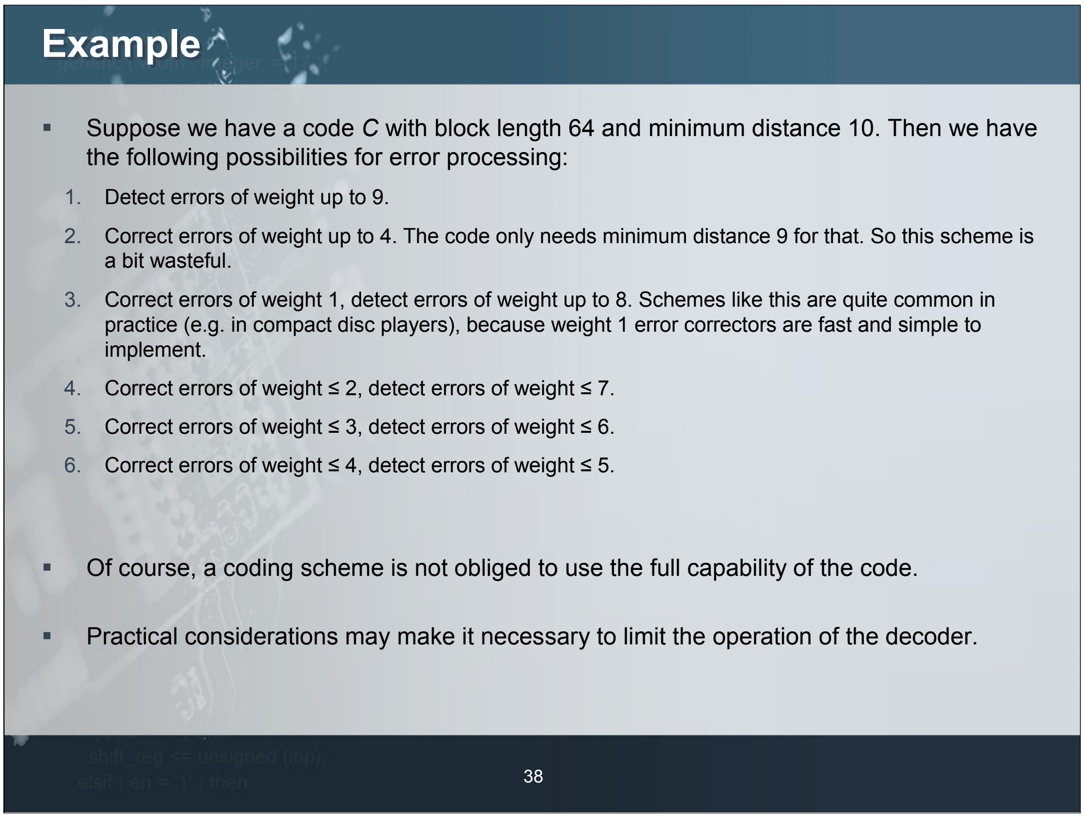


Mixed Strategies I

- We have seen that the minimum distance *d(C)* completely determines the worst-case error-detecting and error-correcting capabilities of a code.
- Often, however, we do not want just to detect all errors of a certain weight, and correcting all error that lie within the theoretical capabilities of the code may be too time consuming or too expensive.
- It is possible to have a mixed strategy: we correct errors of weight up to some (usually small) value *t* and still detect errors of weight up to *t+s*.
- The main theorem, which generalizes propositions discussed on Slides 35, 36 gives precise bounds for *s* and *t* in terms of *d(C)*.
- **A code C can correct all errors of weight up to *t* and at the same time detect all errors of weight up to *s+t* if and only if**

$$d(C) \geq 2t + s + 1$$

- Informally, this says that error correction costs about twice as much as error detection - for every error bit you attempt to correct you lose two bits in the number of errors you can detect. That is because if you are employing error correction an error that pushes a code word *u* close enough to another code word *v* will cause the decoder to choose *v* rather that content itself with the statement that the received word is erroneous.

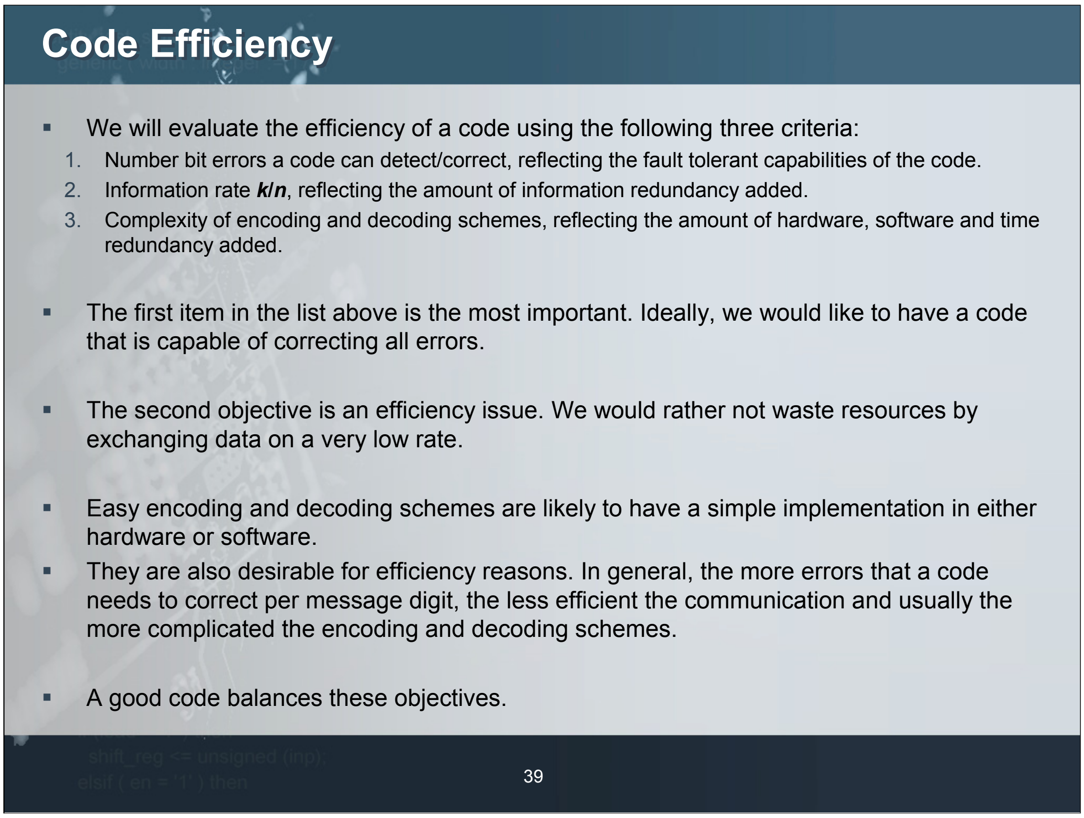


Example

- Suppose we have a code *C* with block length 64 and minimum distance 10. Then we have the following possibilities for error processing:

1. Detect errors of weight up to 9.
2. Correct errors of weight up to 4. The code only needs minimum distance 9 for that. So this scheme is a bit wasteful.
3. Correct errors of weight 1, detect errors of weight up to 8. Schemes like this are quite common in practice (e.g. in compact disc players), because weight 1 error correctors are fast and simple to implement.
4. Correct errors of weight ≤ 2, detect errors of weight ≤ 7.
5. Correct errors of weight ≤ 3, detect errors of weight ≤ 6.
6. Correct errors of weight ≤ 4, detect errors of weight ≤ 5.

- Of course, a coding scheme is not obliged to use the full capability of the code.
- Practical considerations may make it necessary to limit the operation of the decoder.



Code Efficiency

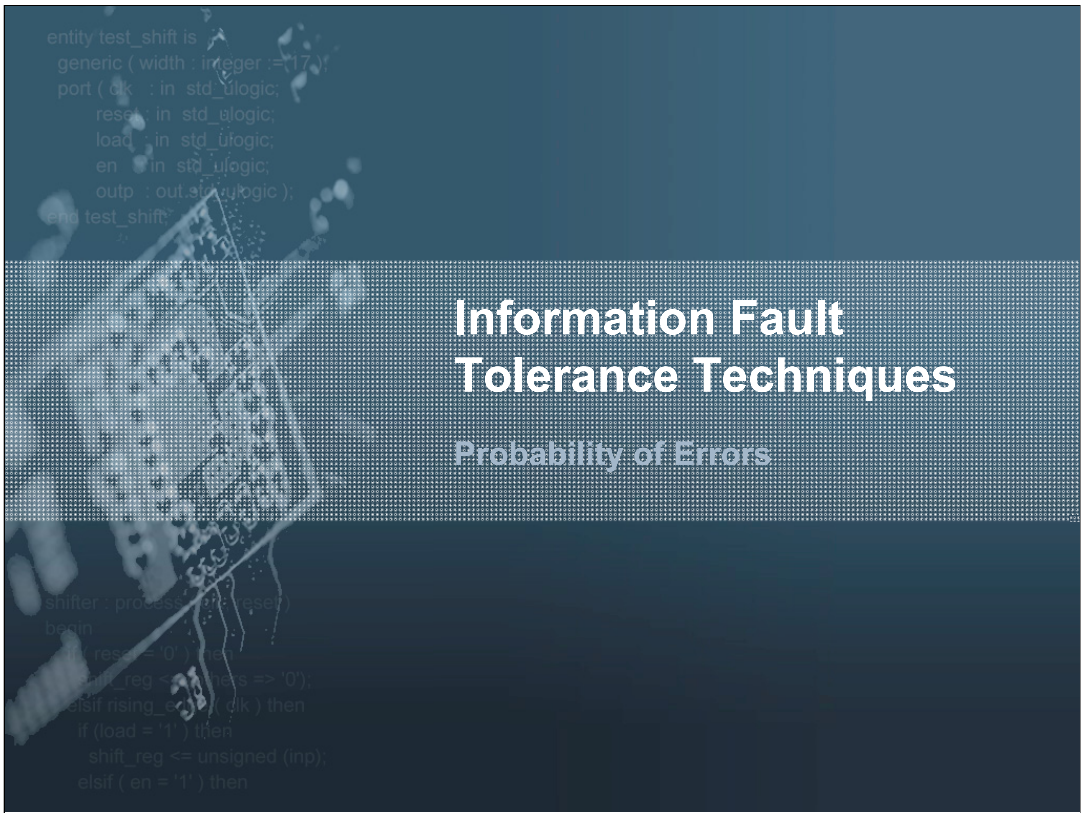
- We will evaluate the efficiency of a code using the following three criteria:
  1. Number bit errors a code can detect/correct, reflecting the fault tolerant capabilities of the code.
  2. Information rate ***k*/*n***, reflecting the amount of information redundancy added.
  3. Complexity of encoding and decoding schemes, reflecting the amount of hardware, software and time redundancy added.

- The first item in the list above is the most important. Ideally, we would like to have a code that is capable of correcting all errors.

- The second objective is an efficiency issue. We would rather not waste resources by exchanging data on a very low rate.

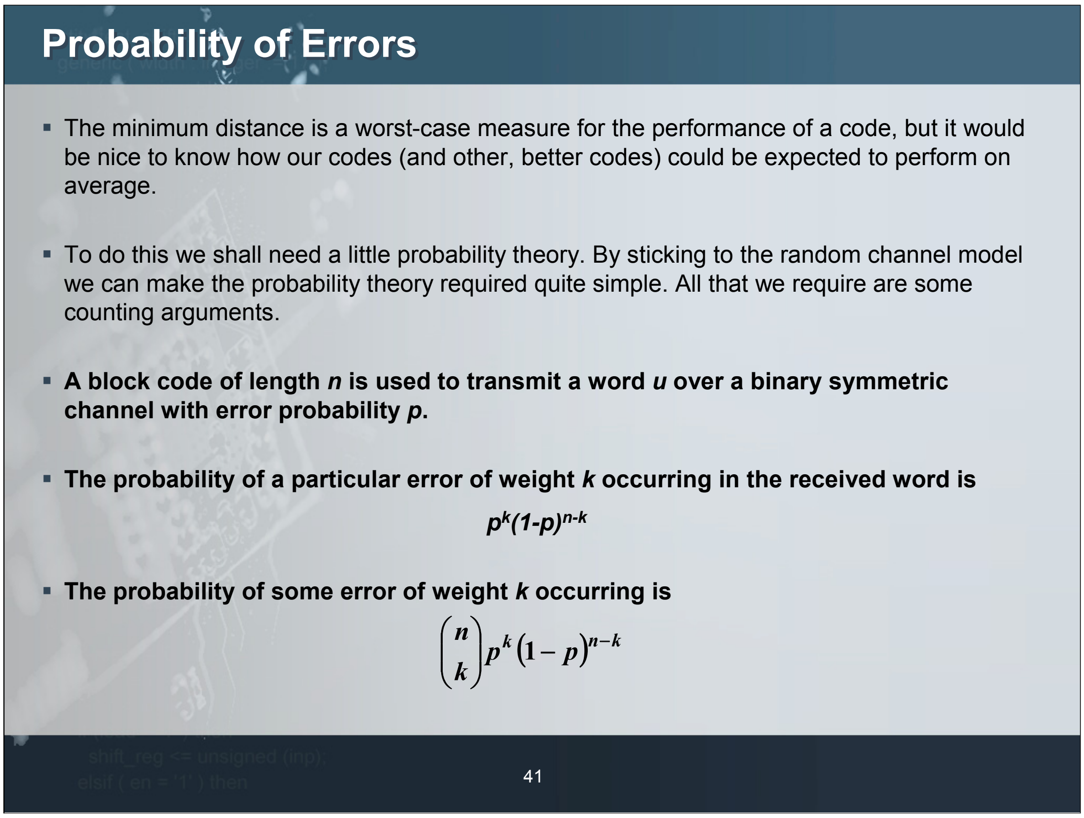
- Easy encoding and decoding schemes are likely to have a simple implementation in either hardware or software.
- They are also desirable for efficiency reasons. In general, the more errors that a code needs to correct per message digit, the less efficient the communication and usually the more complicated the encoding and decoding schemes.

- A good code balances these objectives.



Information Fault Tolerance Techniques

Probability of Errors



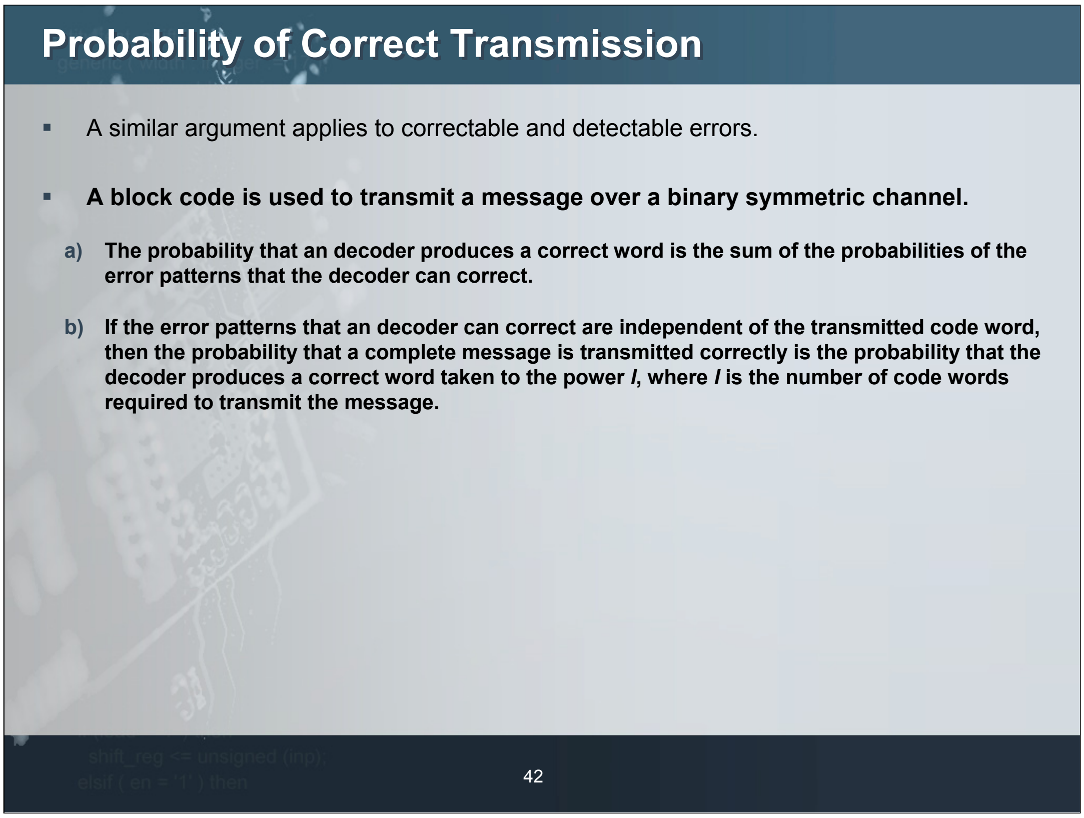
Probability of Errors

- The minimum distance is a worst-case measure for the performance of a code, but it would be nice to know how our codes (and other, better codes) could be expected to perform on average.
- To do this we shall need a little probability theory. By sticking to the random channel model we can make the probability theory required quite simple. All that we require are some counting arguments.
- **A block code of length *n* is used to transmit a word *u* over a binary symmetric channel with error probability *p*.**
- **The probability of a particular error of weight *k* occurring in the received word is**

$$p^k(1-p)^{n-k}$$

- **The probability of some error of weight *k* occurring is**

$$\binom{n}{k} p^k (1-p)^{n-k}$$



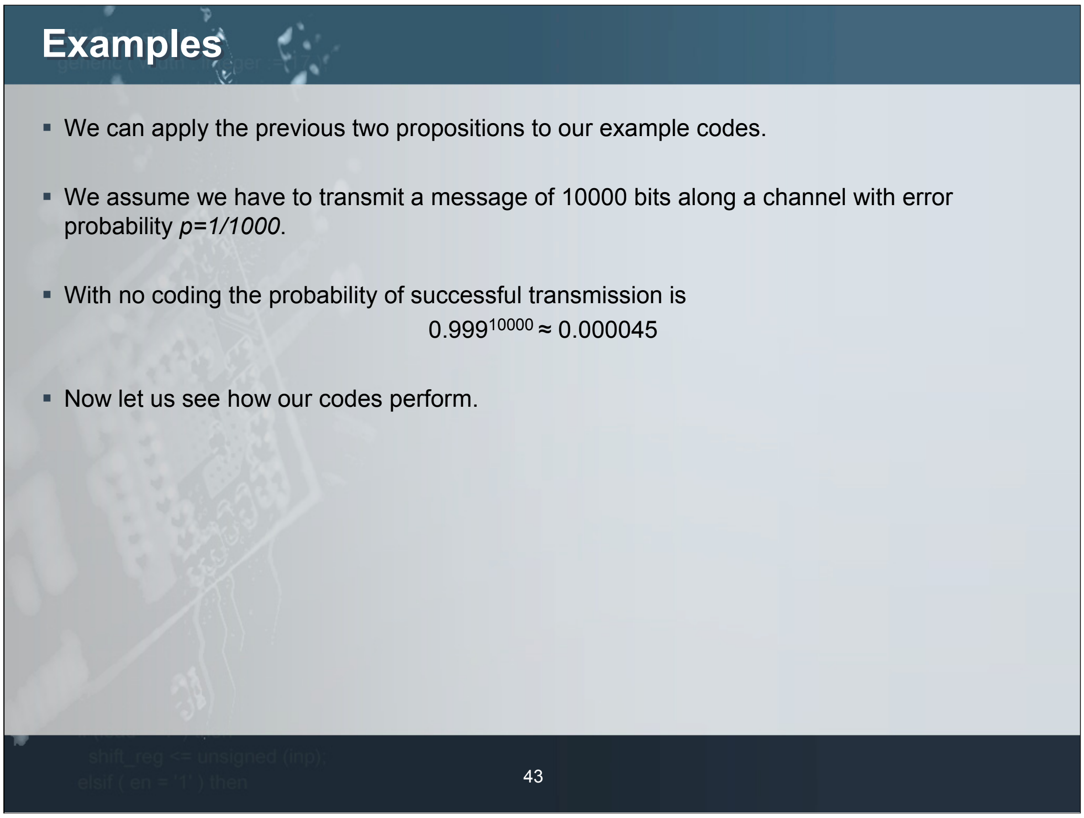
Probability of Correct Transmission

- A similar argument applies to correctable and detectable errors.

- **A block code is used to transmit a message over a binary symmetric channel.**

  a) **The probability that an decoder produces a correct word is the sum of the probabilities of the error patterns that the decoder can correct.**

  b) **If the error patterns that an decoder can correct are independent of the transmitted code word, then the probability that a complete message is transmitted correctly is the probability that the decoder produces a correct word taken to the power *l*, where *l* is the number of code words required to transmit the message.**

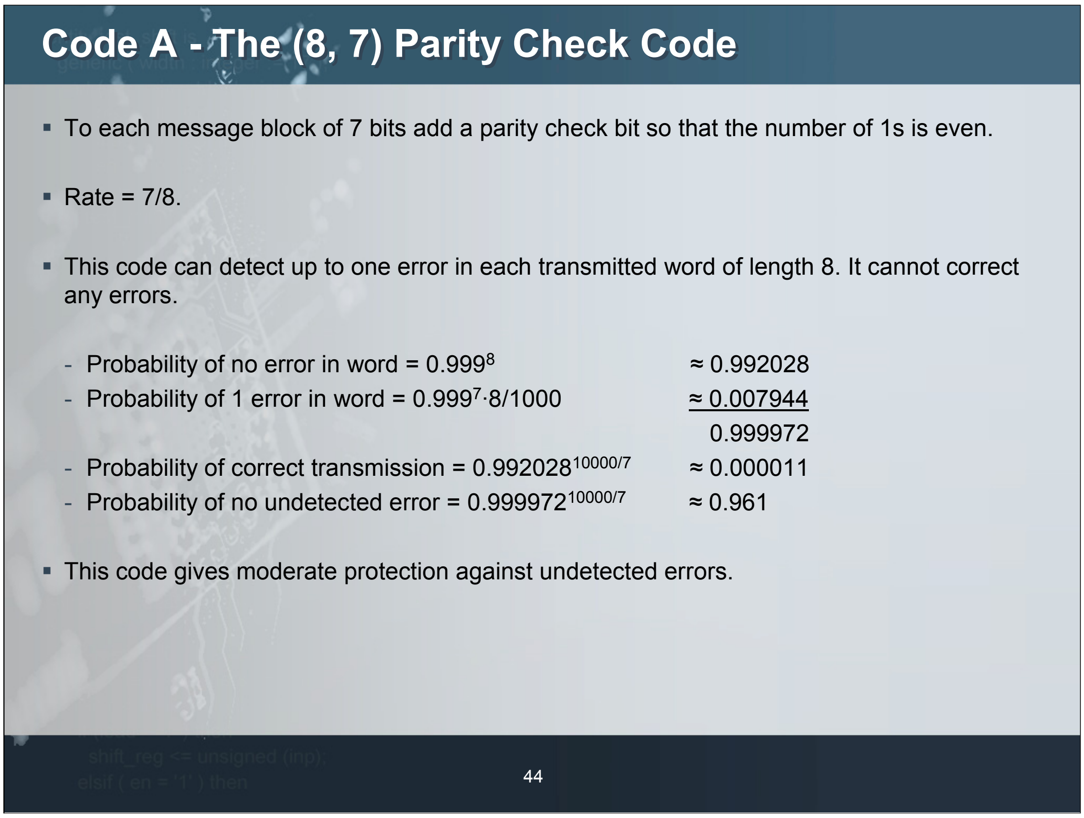


Examples

- We can apply the previous two propositions to our example codes.
- We assume we have to transmit a message of 10000 bits along a channel with error probability *p=1/1000*.
- With no coding the probability of successful transmission is

$$0.999^{10000} \approx 0.000045$$

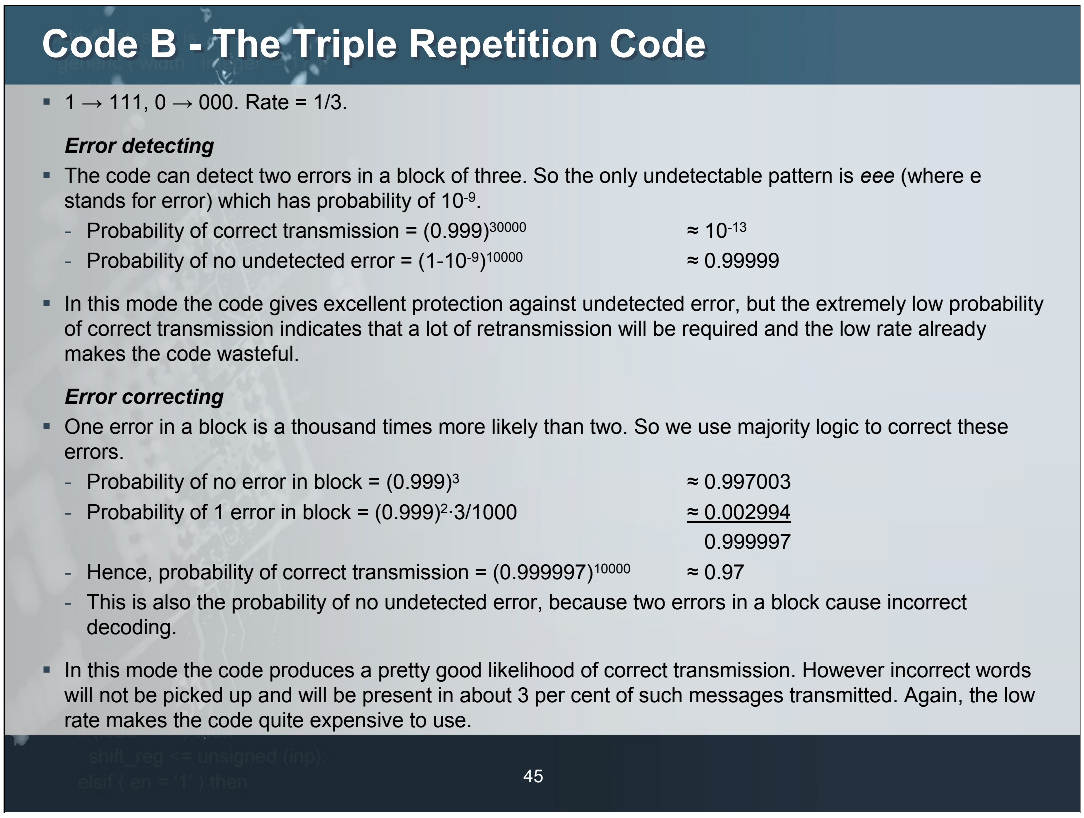
- Now let us see how our codes perform.



Code A - The (8, 7) Parity Check Code

- To each message block of 7 bits add a parity check bit so that the number of 1s is even.
- Rate = 7/8.
- This code can detect up to one error in each transmitted word of length 8. It cannot correct any errors.
  - Probability of no error in word = 0.999^8 ≈ 0.992028
  - Probability of 1 error in word = $0.999^7 \cdot 8/1000$ $\underline{\approx 0.007944}$

    0.999972
  - Probability of correct transmission = $0.992028^{10000/7}$ ≈ 0.000011
  - Probability of no undetected error = $0.999972^{10000/7}$ ≈ 0.961
- This code gives moderate protection against undetected errors.



Code B - The Triple Repetition Code

- $1 \rightarrow 111$, $0 \rightarrow 000$. Rate = 1/3.

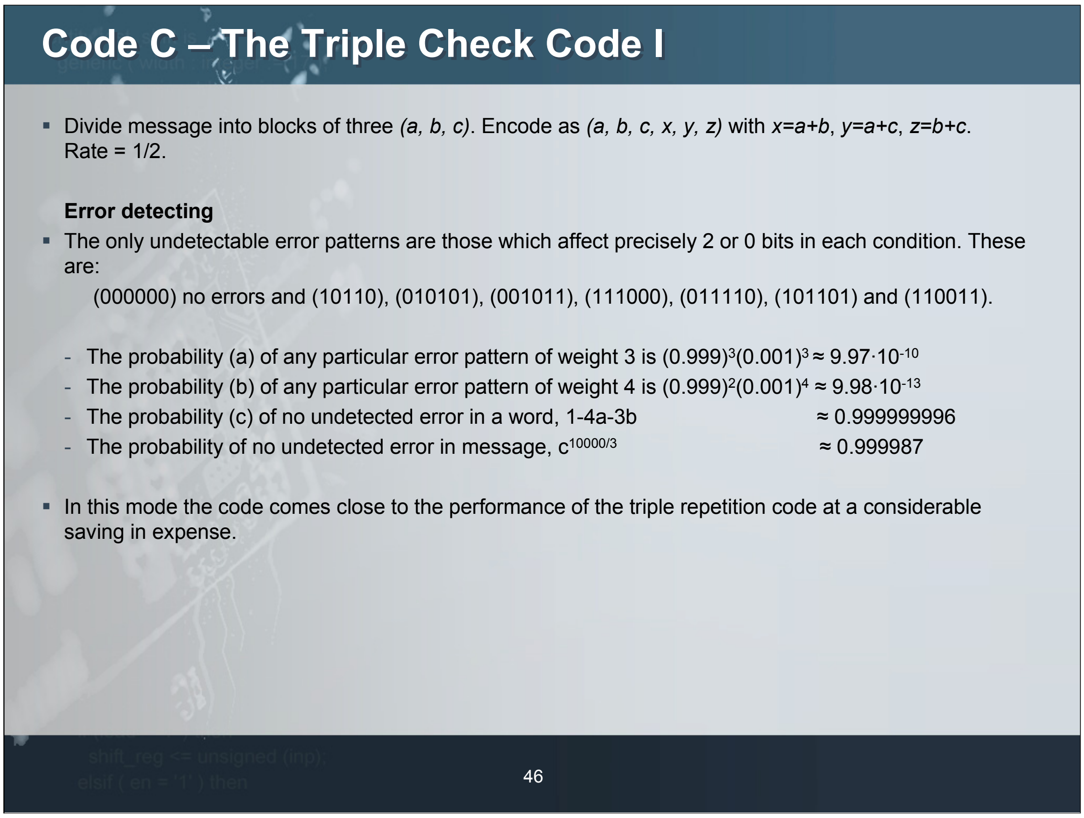
Error detecting

- The code can detect two errors in a block of three. So the only undetectable pattern is *eee* (where e stands for error) which has probability of 10^{-9}.
  - Probability of correct transmission = $(0.999)^{30000}$ $\approx 10^{-13}$
  - Probability of no undetected error = $(1-10^{-9})^{10000}$ ≈ 0.99999
- In this mode the code gives excellent protection against undetected error, but the extremely low probability of correct transmission indicates that a lot of retransmission will be required and the low rate already makes the code wasteful.

Error correcting

- One error in a block is a thousand times more likely than two. So we use majority logic to correct these errors.
  - Probability of no error in block = $(0.999)^3$ ≈ 0.997003
  - Probability of 1 error in block = $(0.999)^2 \cdot 3/1000$ $\approx \underline{0.002994}$

    0.999997
  - Hence, probability of correct transmission = $(0.999997)^{10000}$ ≈ 0.97
  - This is also the probability of no undetected error, because two errors in a block cause incorrect decoding.
- In this mode the code produces a pretty good likelihood of correct transmission. However incorrect words will not be picked up and will be present in about 3 per cent of such messages transmitted. Again, the low rate makes the code quite expensive to use.



Code C – The Triple Check Code I

- Divide message into blocks of three *(a, b, c)*. Encode as *(a, b, c, x, y, z)* with *x=a+b*, *y=a+c*, *z=b+c*. Rate = 1/2.

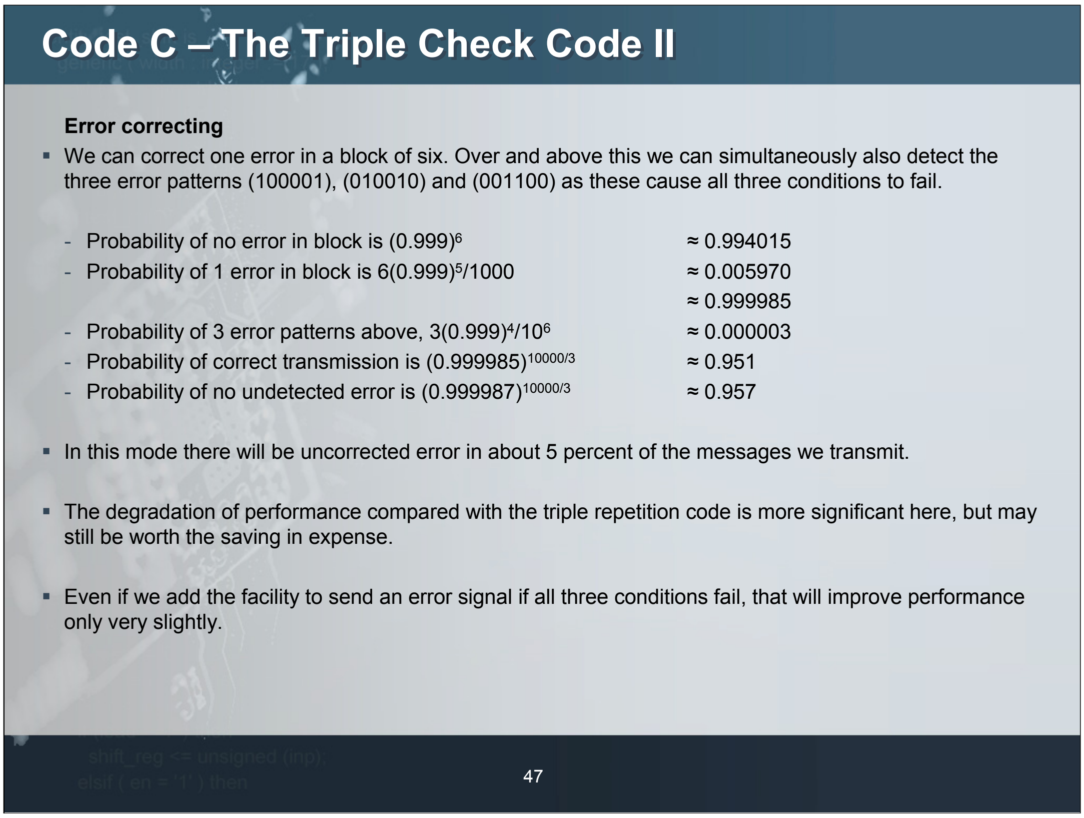
Error detecting

- The only undetectable error patterns are those which affect precisely 2 or 0 bits in each condition. These are:

  (000000) no errors and (10110), (010101), (001011), (111000), (011110), (101101) and (110011).

  - The probability (a) of any particular error pattern of weight 3 is $(0.999)^3(0.001)^3 \approx 9.97 \cdot 10^{-10}$
  - The probability (b) of any particular error pattern of weight 4 is $(0.999)^2(0.001)^4 \approx 9.98 \cdot 10^{-13}$
  - The probability (c) of no undetected error in a word, 1-4a-3b ≈ 0.999999996
  - The probability of no undetected error in message, $c^{10000/3}$ ≈ 0.999987

- In this mode the code comes close to the performance of the triple repetition code at a considerable saving in expense.



Code C – The Triple Check Code II

Error correcting

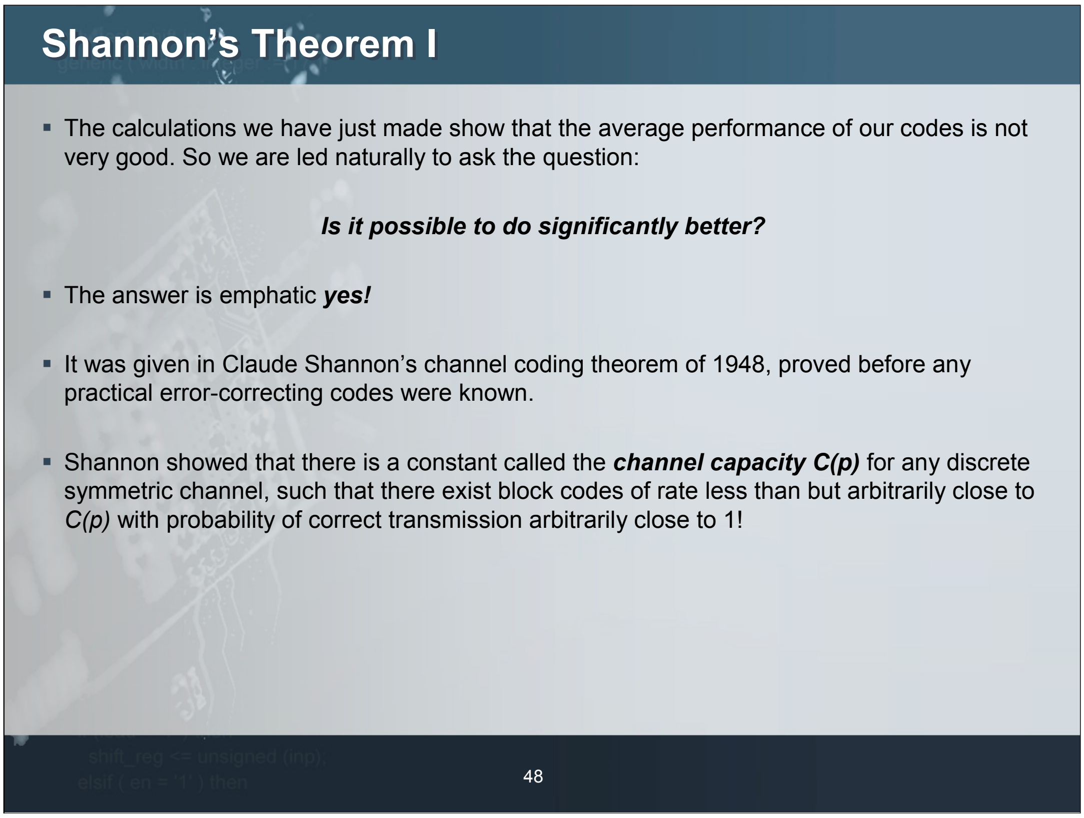
- We can correct one error in a block of six. Over and above this we can simultaneously also detect the three error patterns (100001), (010010) and (001100) as these cause all three conditions to fail.

- Probability of no error in block is $(0.999)^6$	≈ 0.994015
- Probability of 1 error in block is $6(0.999)^5/1000$	≈ 0.005970
	≈ 0.999985
- Probability of 3 error patterns above, $3(0.999)^4/10^6$	≈ 0.000003
- Probability of correct transmission is $(0.999985)^{10000/3}$	≈ 0.951
- Probability of no undetected error is $(0.999987)^{10000/3}$	≈ 0.957

- In this mode there will be uncorrected error in about 5 percent of the messages we transmit.

- The degradation of performance compared with the triple repetition code is more significant here, but may still be worth the saving in expense.

- Even if we add the facility to send an error signal if all three conditions fail, that will improve performance only very slightly.

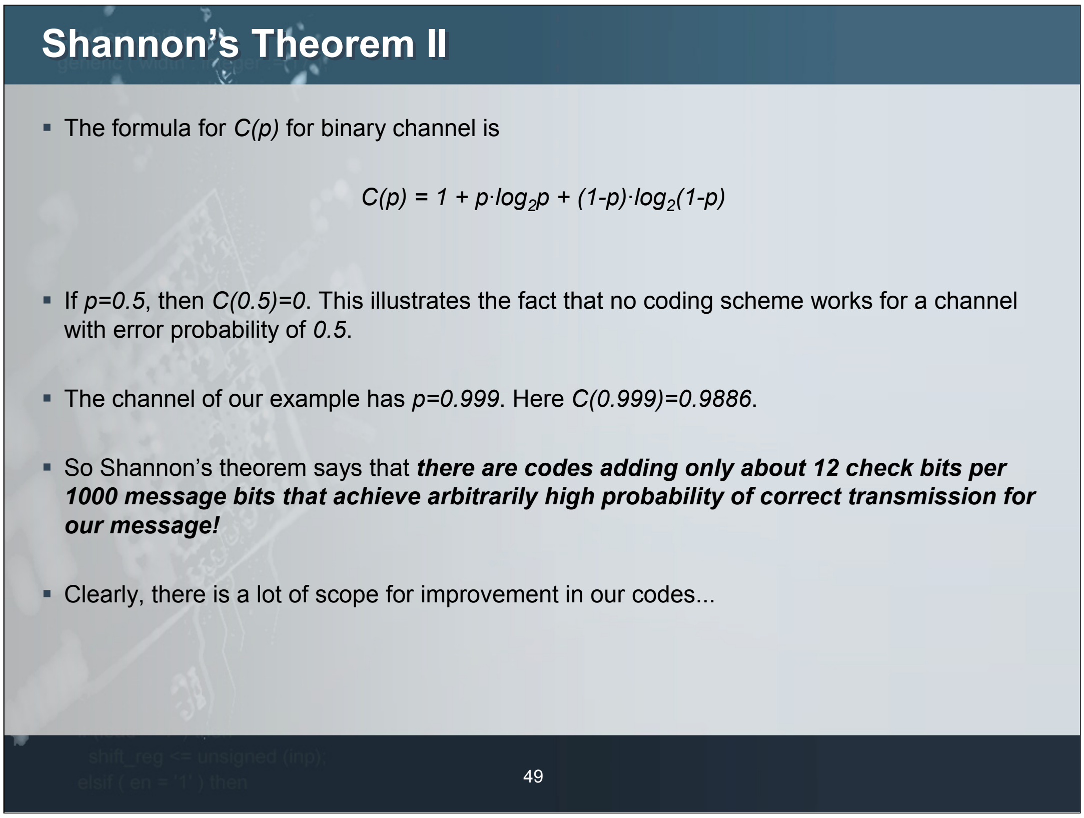


Shannon’s Theorem I

- The calculations we have just made show that the average performance of our codes is not very good. So we are led naturally to ask the question:

Is it possible to do significantly better?

- The answer is emphatic ***yes!***
- It was given in Claude Shannon’s channel coding theorem of 1948, proved before any practical error-correcting codes were known.
- Shannon showed that there is a constant called the ***channel capacity C(p)*** for any discrete symmetric channel, such that there exist block codes of rate less than but arbitrarily close to *C(p)* with probability of correct transmission arbitrarily close to 1!

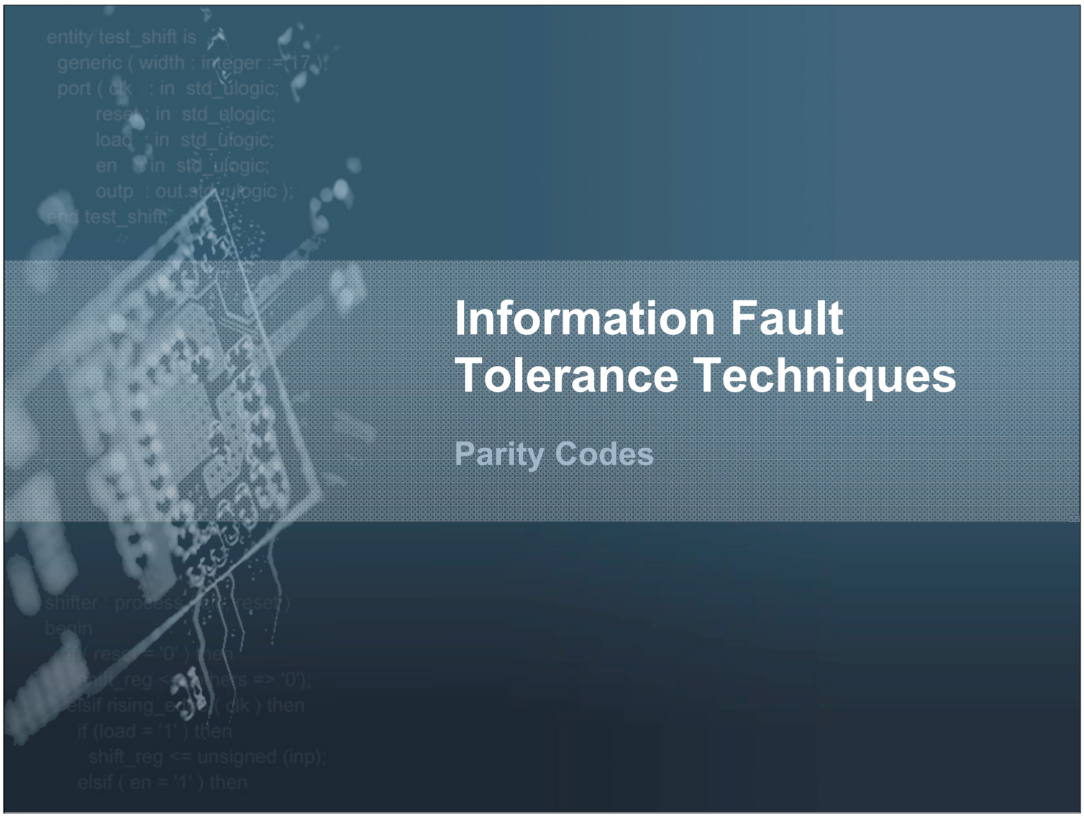


Shannon's Theorem II

- The formula for *C(p)* for binary channel is

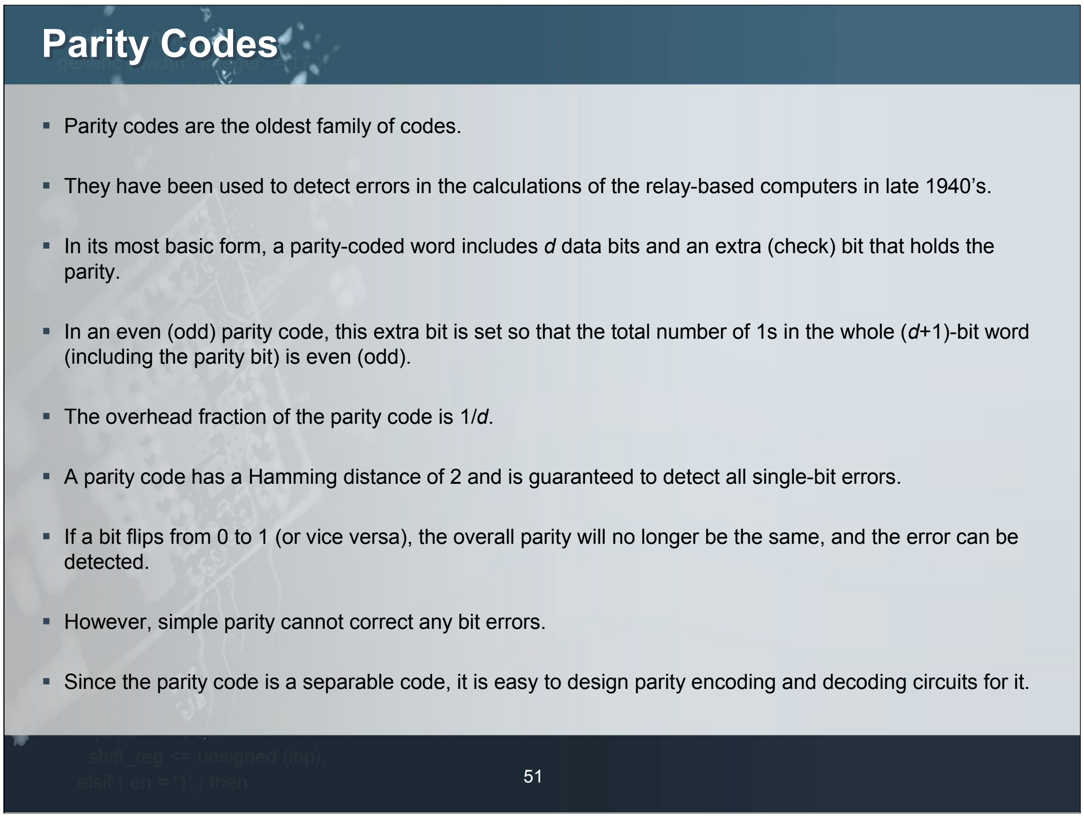
$$C(p) = 1 + p \cdot \log_2 p + (1-p) \cdot \log_2 (1-p)$$

- If *p=0.5*, then *C(0.5)=0*. This illustrates the fact that no coding scheme works for a channel with error probability of *0.5*.
- The channel of our example has *p=0.999*. Here *C(0.999)=0.9886*.
- So Shannon's theorem says that ***there are codes adding only about 12 check bits per 1000 message bits that achieve arbitrarily high probability of correct transmission for our message!***
- Clearly, there is a lot of scope for improvement in our codes...



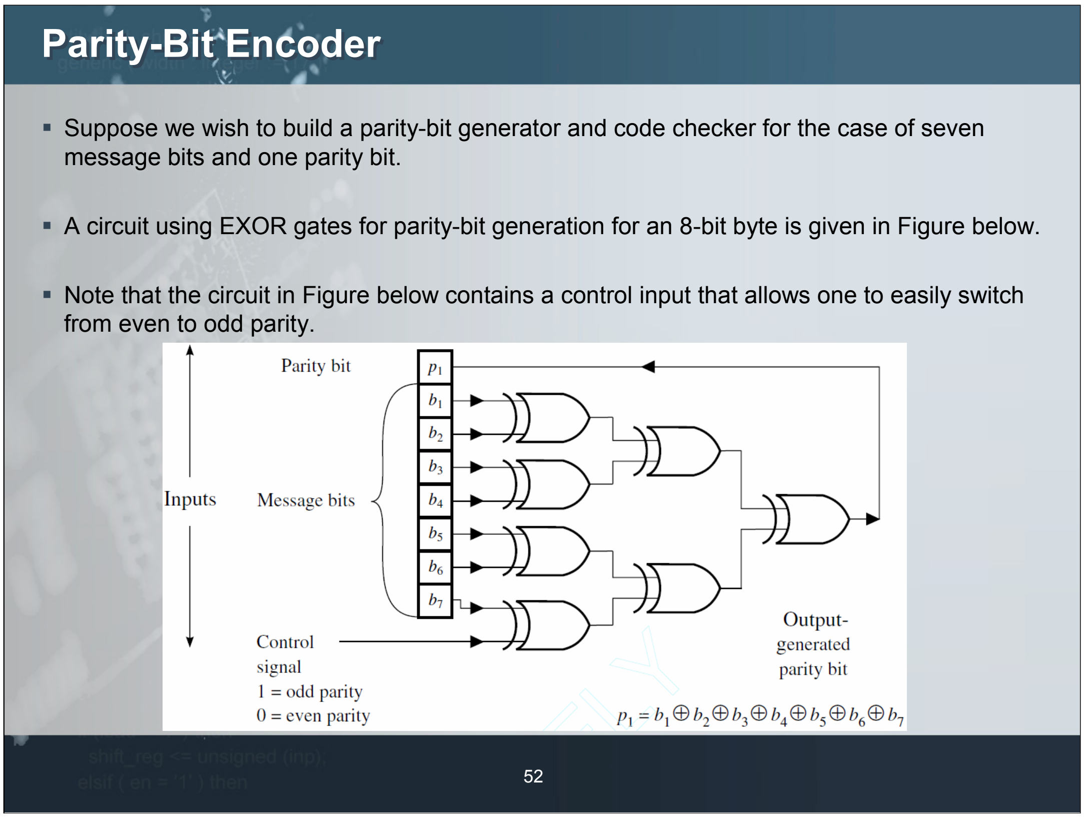
Information Fault Tolerance Techniques

Parity Codes



Parity Codes

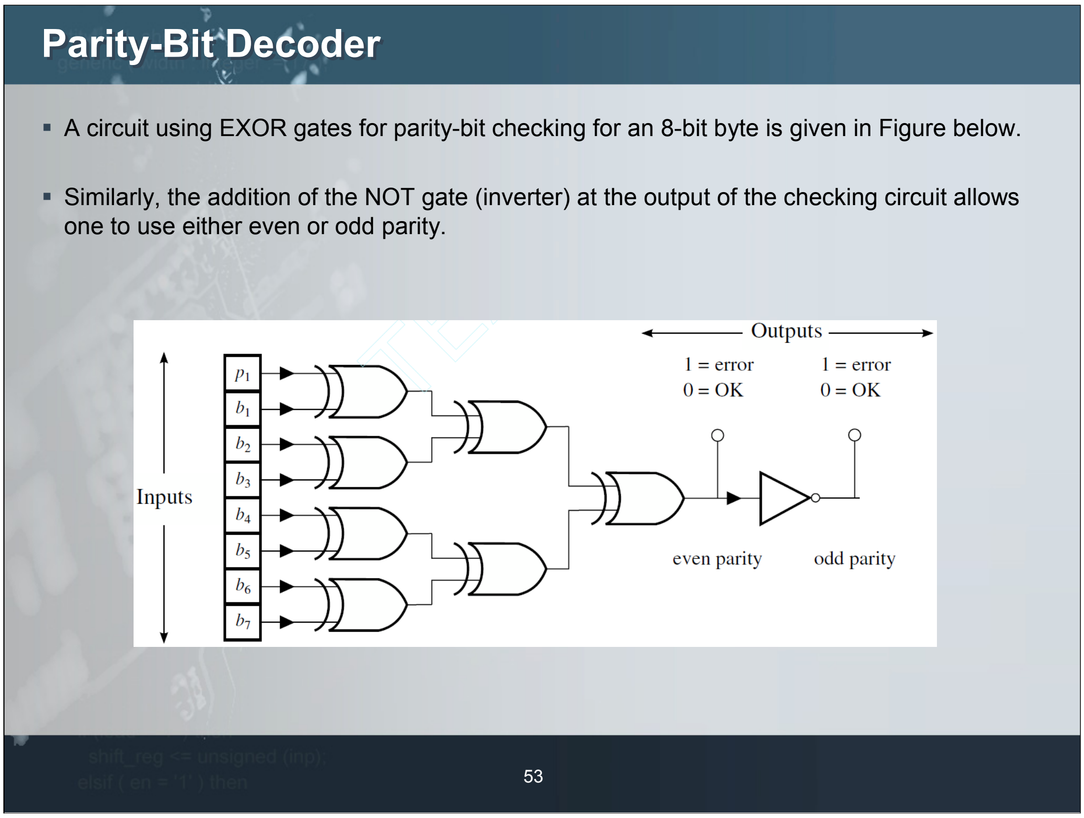
- Parity codes are the oldest family of codes.
- They have been used to detect errors in the calculations of the relay-based computers in late 1940’s.
- In its most basic form, a parity-coded word includes *d* data bits and an extra (check) bit that holds the parity.
- In an even (odd) parity code, this extra bit is set so that the total number of 1s in the whole (*d*+1)-bit word (including the parity bit) is even (odd).
- The overhead fraction of the parity code is 1/*d*.
- A parity code has a Hamming distance of 2 and is guaranteed to detect all single-bit errors.
- If a bit flips from 0 to 1 (or vice versa), the overall parity will no longer be the same, and the error can be detected.
- However, simple parity cannot correct any bit errors.
- Since the parity code is a separable code, it is easy to design parity encoding and decoding circuits for it.



Parity-Bit Encoder

- Suppose we wish to build a parity-bit generator and code checker for the case of seven message bits and one parity bit.
- A circuit using EXOR gates for parity-bit generation for an 8-bit byte is given in Figure below.
- Note that the circuit in Figure below contains a control input that allows one to easily switch from even to odd parity.

$p_1 = b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7$

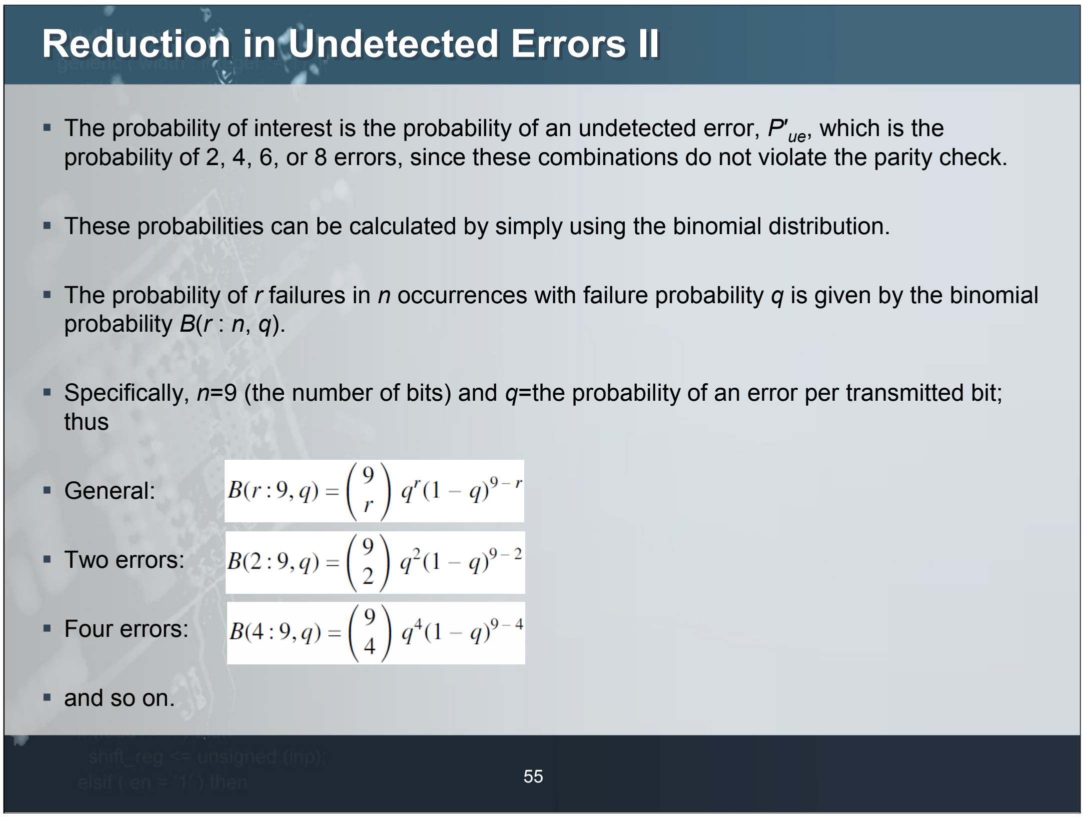


Parity-Bit Decoder

- A circuit using EXOR gates for parity-bit checking for an 8-bit byte is given in Figure below.
- Similarly, the addition of the NOT gate (inverter) at the output of the checking circuit allows one to use either even or odd parity.

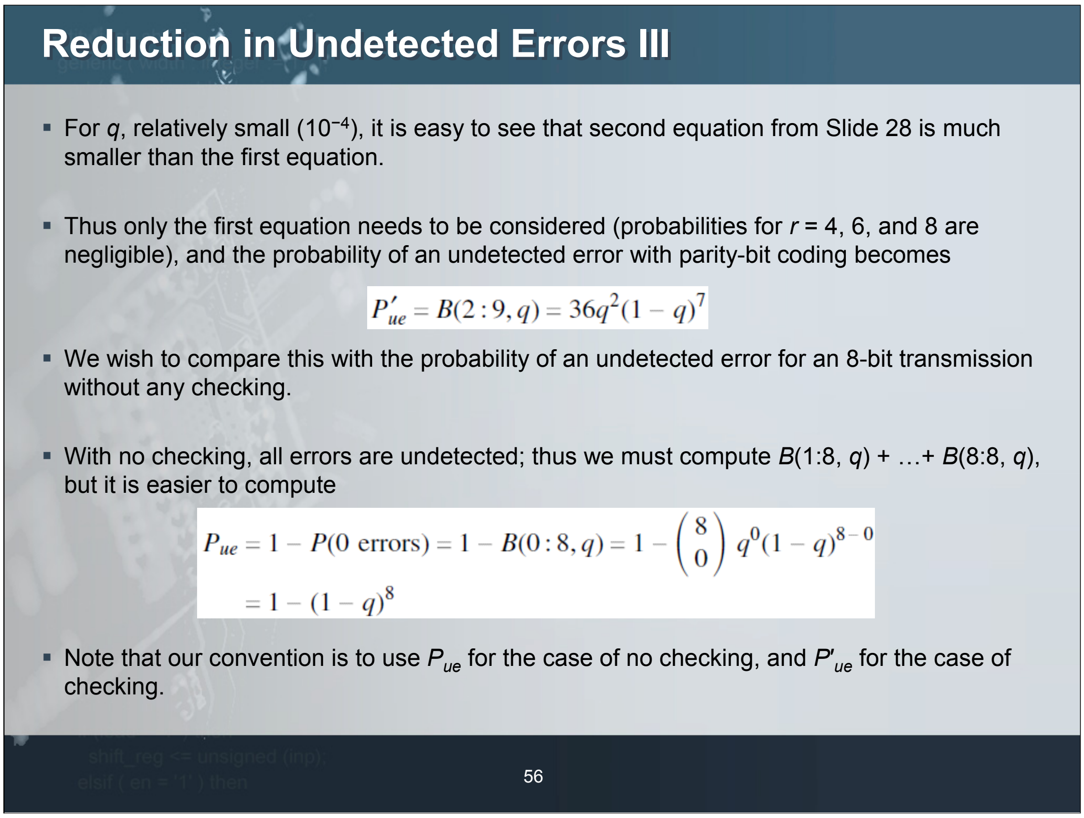
Reduction in Undetected Errors I

- The purpose of parity-bit checking is to detect errors.
- The extent to which such errors are detected is a measure of the success of the code, whereas the probability of not detecting an error, P_{ue}, is a measure of failure.
- In this section we analyze how parity-bit coding decreases P_{ue}.
- Let us consider the addition of a ninth parity bit to an 8-bit message byte.
- The parity bit adjusts the number of ones in the word to an even (odd) number and is computed by a parity-bit generator circuit that calculates the EXOR function of the 8 message bits. Similarly, an EXOR-detecting circuit is used to check for transmission errors.
- If 1, 3, 5, 7, or 9 errors are found in the received word, the parity is violated, and the checking circuit will detect an error.
- This can lead to several consequences, including “flagging” the error byte and retransmission of the byte until no errors are detected.



Reduction in Undetected Errors II

- The probability of interest is the probability of an undetected error, P'_{ue}, which is the probability of 2, 4, 6, or 8 errors, since these combinations do not violate the parity check.
- These probabilities can be calculated by simply using the binomial distribution.
- The probability of *r* failures in *n* occurrences with failure probability *q* is given by the binomial probability *B*(*r* : *n*, *q*).
- Specifically, *n*=9 (the number of bits) and *q*=the probability of an error per transmitted bit; thus
- General: $B(r:9,q) = \binom{9}{r} q^r (1-q)^{9-r}$
- Two errors: $B(2:9,q) = \binom{9}{2} q^2 (1-q)^{9-2}$
- Four errors: $B(4:9,q) = \binom{9}{4} q^4 (1-q)^{9-4}$
- and so on.



Reduction in Undetected Errors III

- For q, relatively small (10^{-4}), it is easy to see that second equation from Slide 28 is much smaller than the first equation.

- Thus only the first equation needs to be considered (probabilities for r = 4, 6, and 8 are negligible), and the probability of an undetected error with parity-bit coding becomes

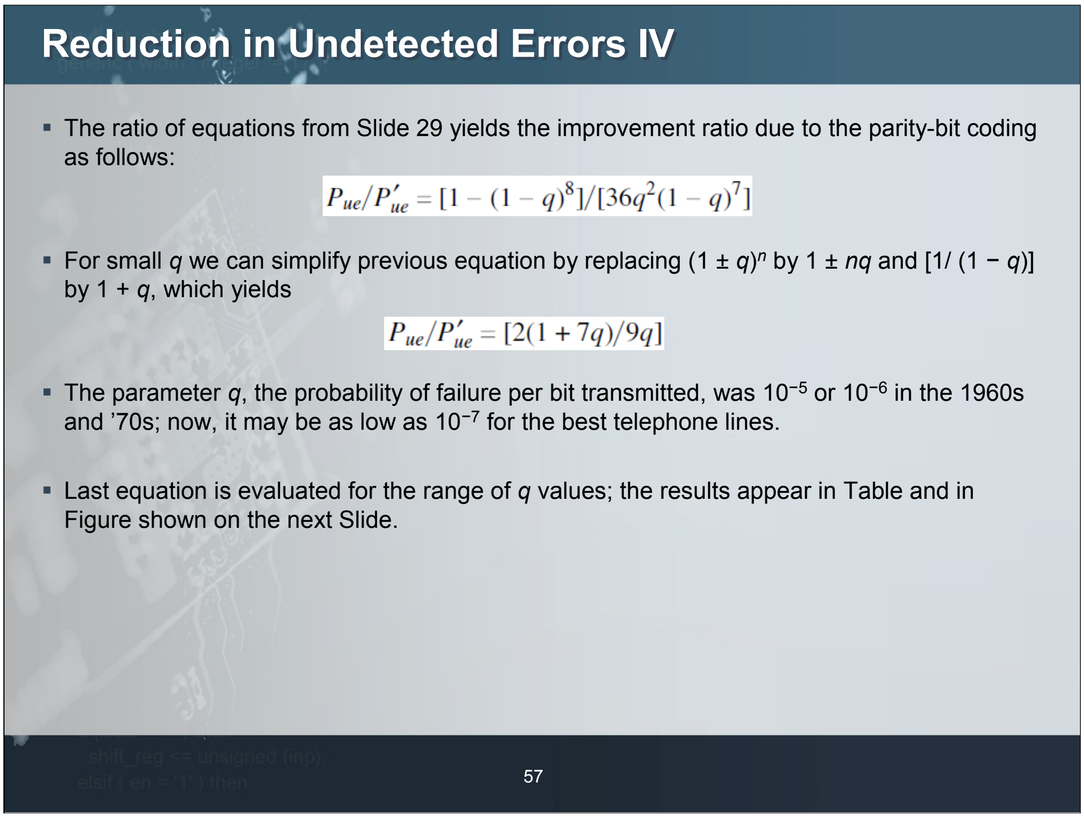
$$P'_{ue} = B(2:9, q) = 36q^2(1-q)^7$$

- We wish to compare this with the probability of an undetected error for an 8-bit transmission without any checking.

- With no checking, all errors are undetected; thus we must compute $B(1{:}8, q)$ + …+ $B(8{:}8, q)$, but it is easier to compute

$$\begin{aligned} P_{ue} &= 1 - P(0 \text{ errors}) = 1 - B(0:8, q) = 1 - \binom{8}{0} q^0(1-q)^{8-0} \\ &= 1 - (1-q)^8 \end{aligned}$$

- Note that our convention is to use P_{ue} for the case of no checking, and P'_{ue} for the case of checking.



Reduction in Undetected Errors IV

- The ratio of equations from Slide 29 yields the improvement ratio due to the parity-bit coding as follows:

$$P_{ue}/P'_{ue} = [1 - (1 - q)^8]/[36q^2(1 - q)^7]$$

- For small q we can simplify previous equation by replacing $(1 \pm q)^n$ by $1 \pm nq$ and $[1/ (1 - q)]$ by $1 + q$, which yields

$$P_{ue}/P'_{ue} = [2(1 + 7q)/9q]$$

- The parameter q, the probability of failure per bit transmitted, was 10^{-5} or 10^{-6} in the 1960s and '70s; now, it may be as low as 10^{-7} for the best telephone lines.

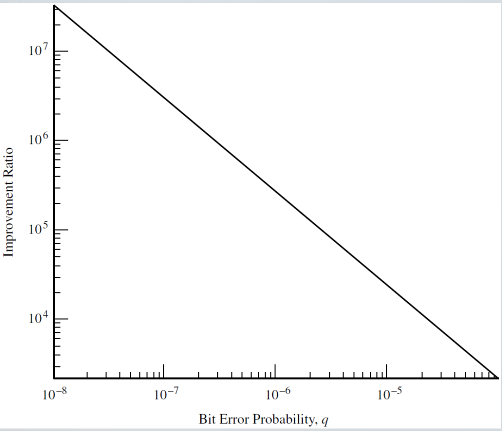
- Last equation is evaluated for the range of q values; the results appear in Table and in Figure shown on the next Slide.

Reduction in Undetected Errors V

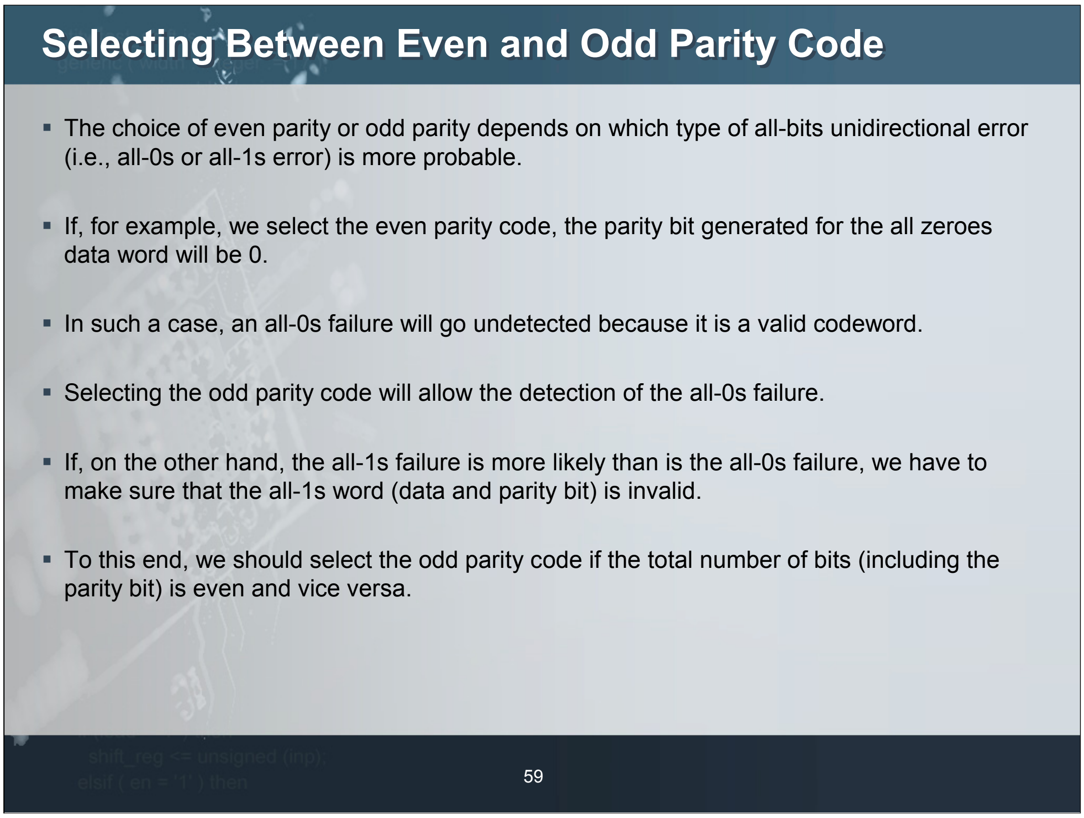
- The improvement ratio is quite significant, and the overhead - adding 1 parity bit out of 8 message bits - is only 12.5%, which is quite modest.
- This probably explains why a parity-bit code is so frequently used.

Bit Error Probability, q	Improvement Ratio: P_{ue}/P'_{ue}
10^{-4}	2.223×10^3
10^{-5}	2.222×10^4
10^{-6}	2.222×10^5
10^{-7}	2.222×10^6
10^{-8}	2.222×10^7

Evaluation of the Reduction in Undetected Errors from Parity-Bit Coding

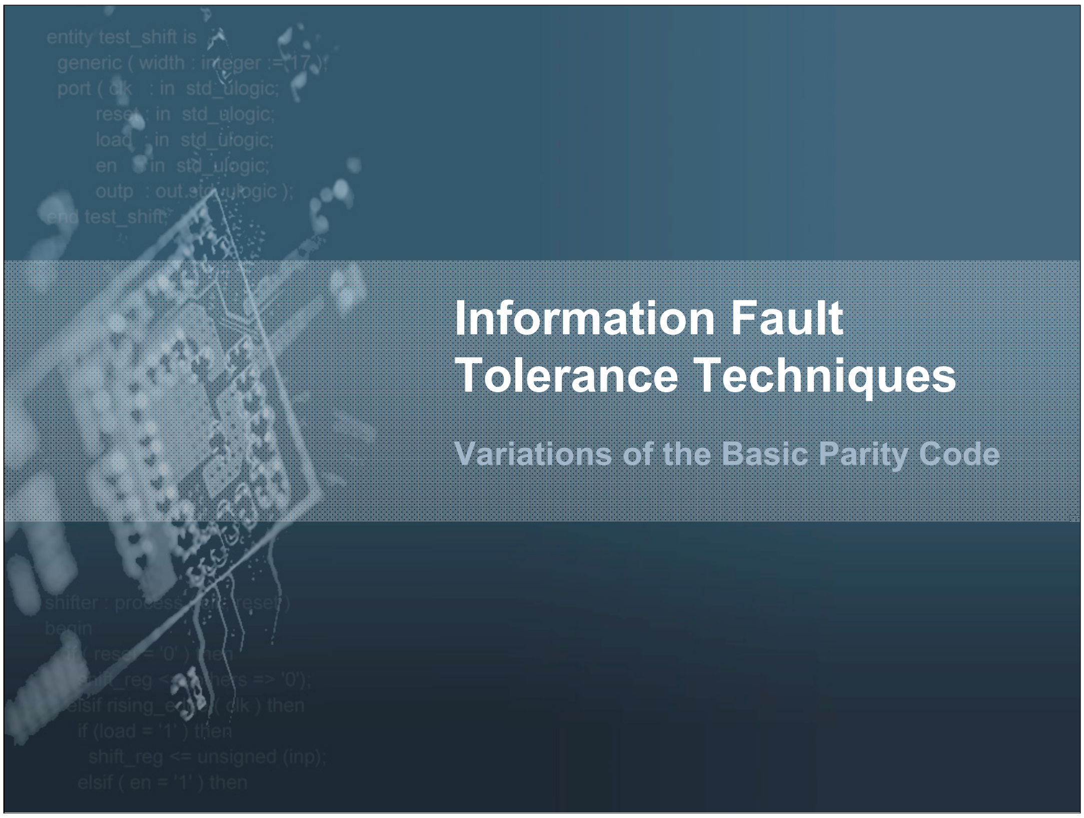


Improvement ratio of undetected error probability from parity-bit coding



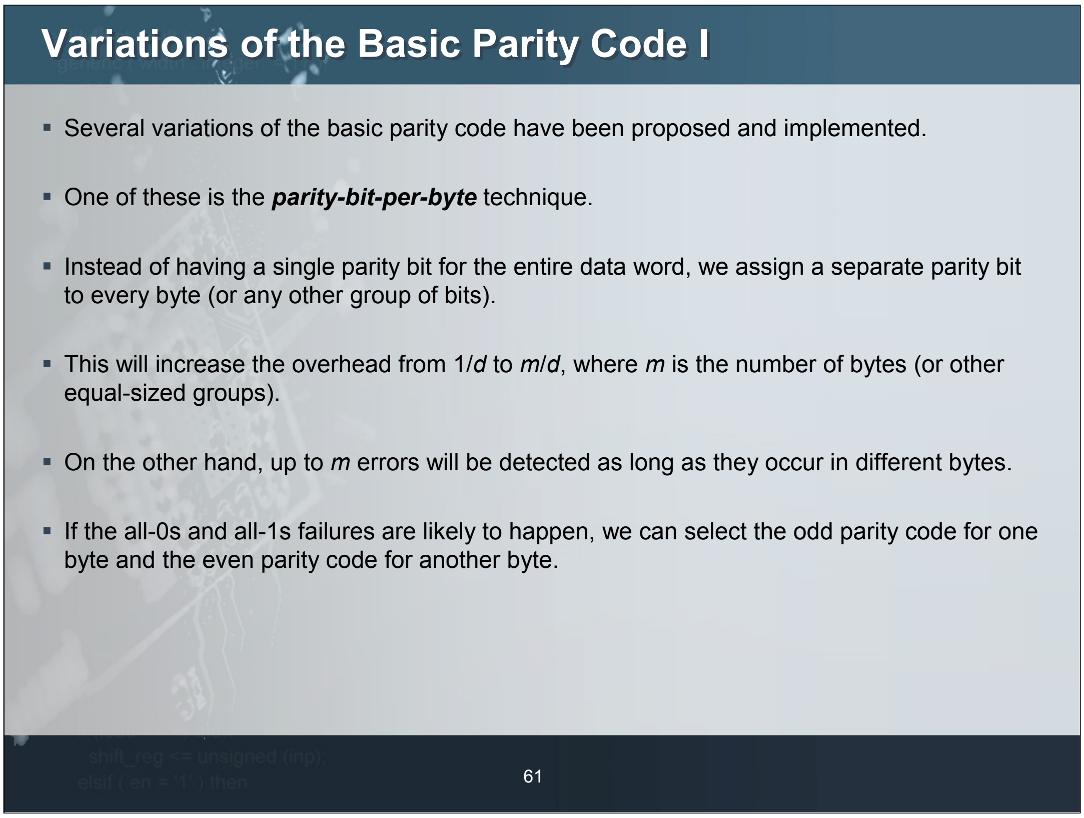
Selecting Between Even and Odd Parity Code

- The choice of even parity or odd parity depends on which type of all-bits unidirectional error (i.e., all-0s or all-1s error) is more probable.
- If, for example, we select the even parity code, the parity bit generated for the all zeroes data word will be 0.
- In such a case, an all-0s failure will go undetected because it is a valid codeword.
- Selecting the odd parity code will allow the detection of the all-0s failure.
- If, on the other hand, the all-1s failure is more likely than is the all-0s failure, we have to make sure that the all-1s word (data and parity bit) is invalid.
- To this end, we should select the odd parity code if the total number of bits (including the parity bit) is even and vice versa.



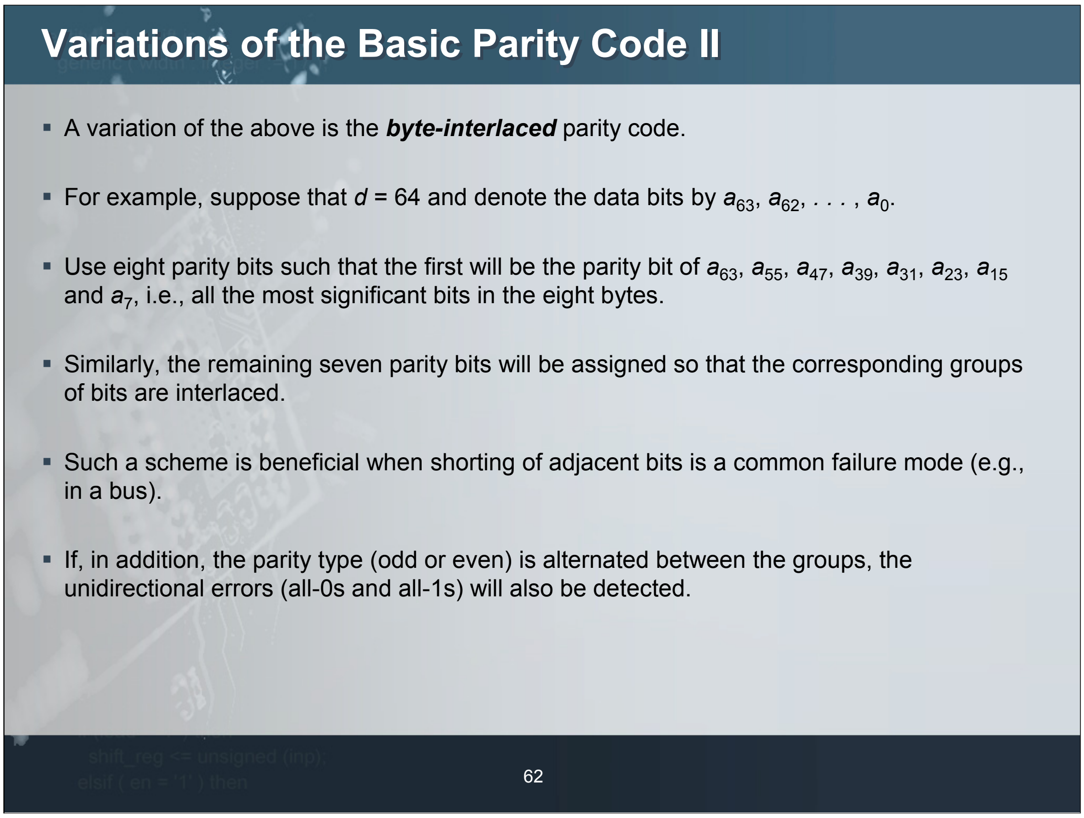
Information Fault Tolerance Techniques

Variations of the Basic Parity Code



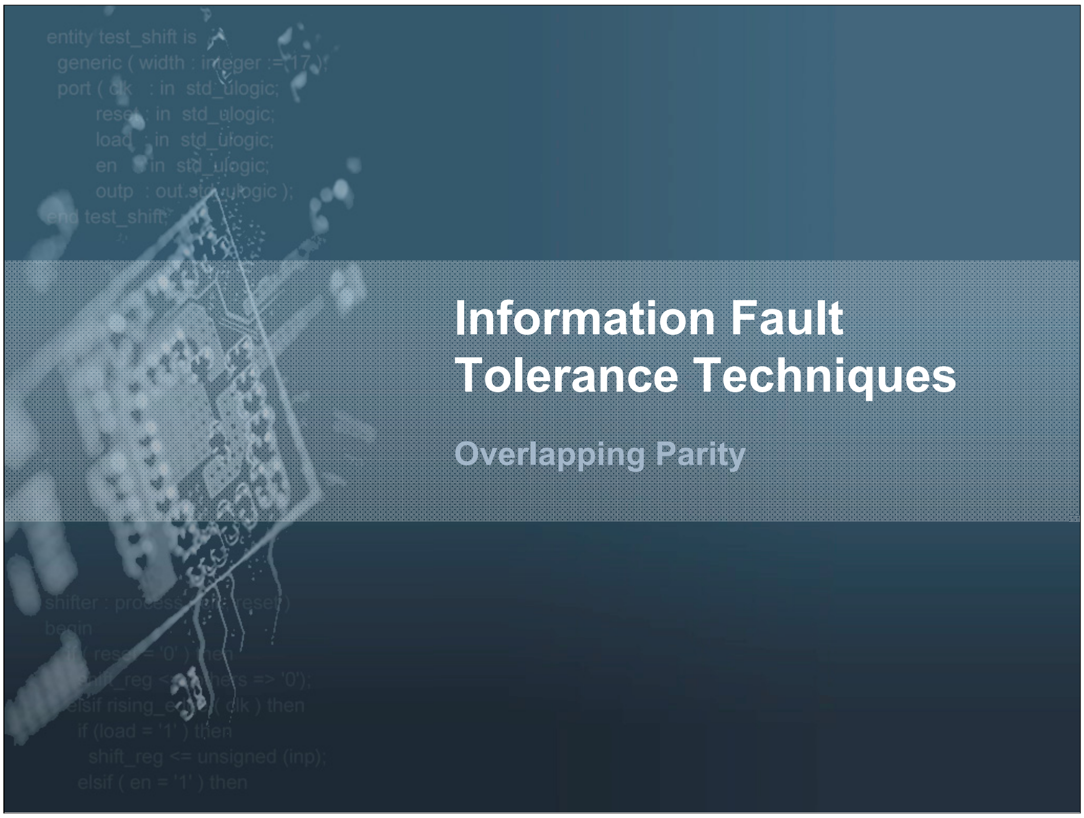
Variations of the Basic Parity Code I

- Several variations of the basic parity code have been proposed and implemented.
- One of these is the ***parity-bit-per-byte*** technique.
- Instead of having a single parity bit for the entire data word, we assign a separate parity bit to every byte (or any other group of bits).
- This will increase the overhead from 1/*d* to *m*/*d*, where *m* is the number of bytes (or other equal-sized groups).
- On the other hand, up to *m* errors will be detected as long as they occur in different bytes.
- If the all-0s and all-1s failures are likely to happen, we can select the odd parity code for one byte and the even parity code for another byte.



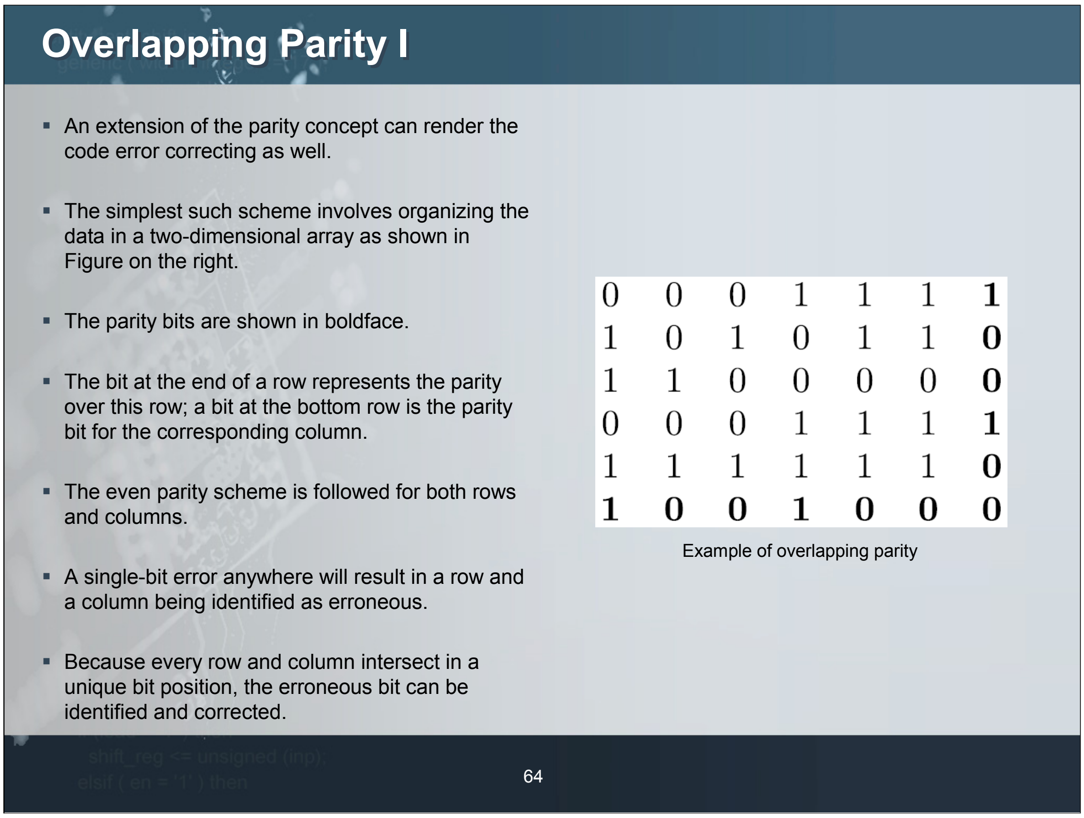
Variations of the Basic Parity Code II

- A variation of the above is the ***byte-interlaced*** parity code.
- For example, suppose that d = 64 and denote the data bits by a_{63}, a_{62}, . . . , a_0.
- Use eight parity bits such that the first will be the parity bit of a_{63}, a_{55}, a_{47}, a_{39}, a_{31}, a_{23}, a_{15} and a_7, i.e., all the most significant bits in the eight bytes.
- Similarly, the remaining seven parity bits will be assigned so that the corresponding groups of bits are interlaced.
- Such a scheme is beneficial when shorting of adjacent bits is a common failure mode (e.g., in a bus).
- If, in addition, the parity type (odd or even) is alternated between the groups, the unidirectional errors (all-0s and all-1s) will also be detected.



Information Fault Tolerance Techniques

Overlapping Parity

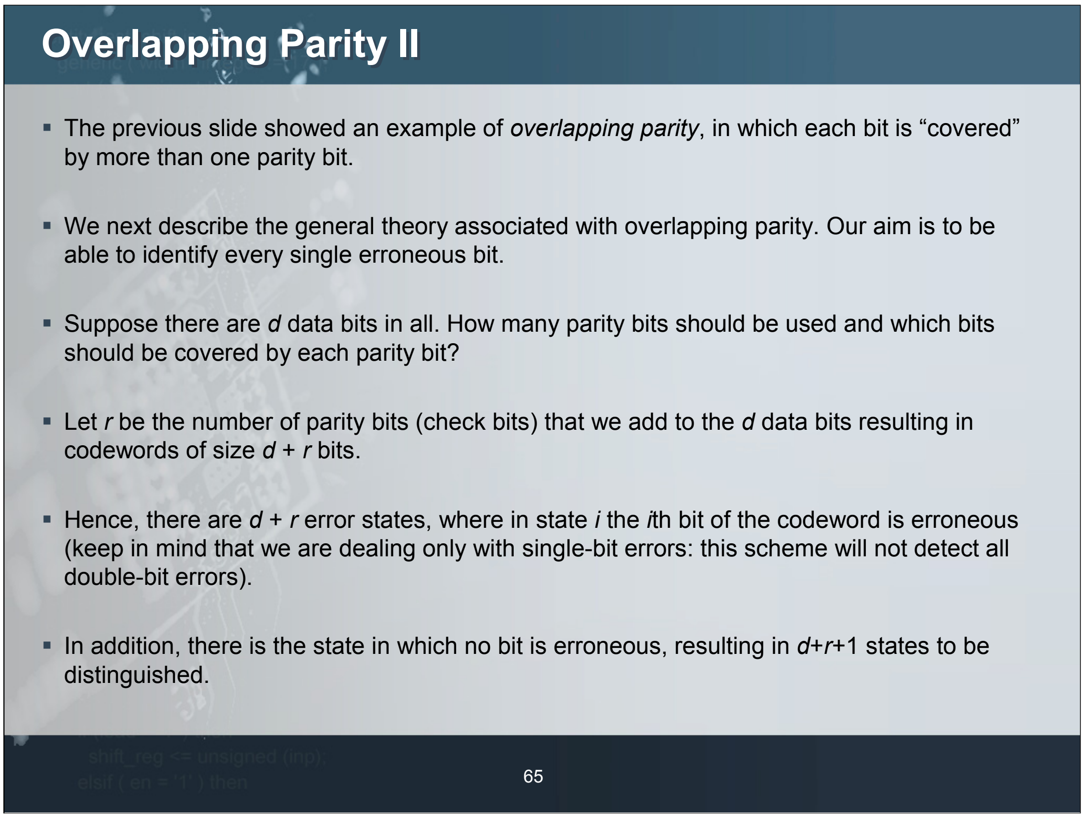


Overlapping Parity I

- An extension of the parity concept can render the code error correcting as well.
- The simplest such scheme involves organizing the data in a two-dimensional array as shown in Figure on the right.
- The parity bits are shown in boldface.
- The bit at the end of a row represents the parity over this row; a bit at the bottom row is the parity bit for the corresponding column.
- The even parity scheme is followed for both rows and columns.
- A single-bit error anywhere will result in a row and a column being identified as erroneous.
- Because every row and column intersect in a unique bit position, the erroneous bit can be identified and corrected.

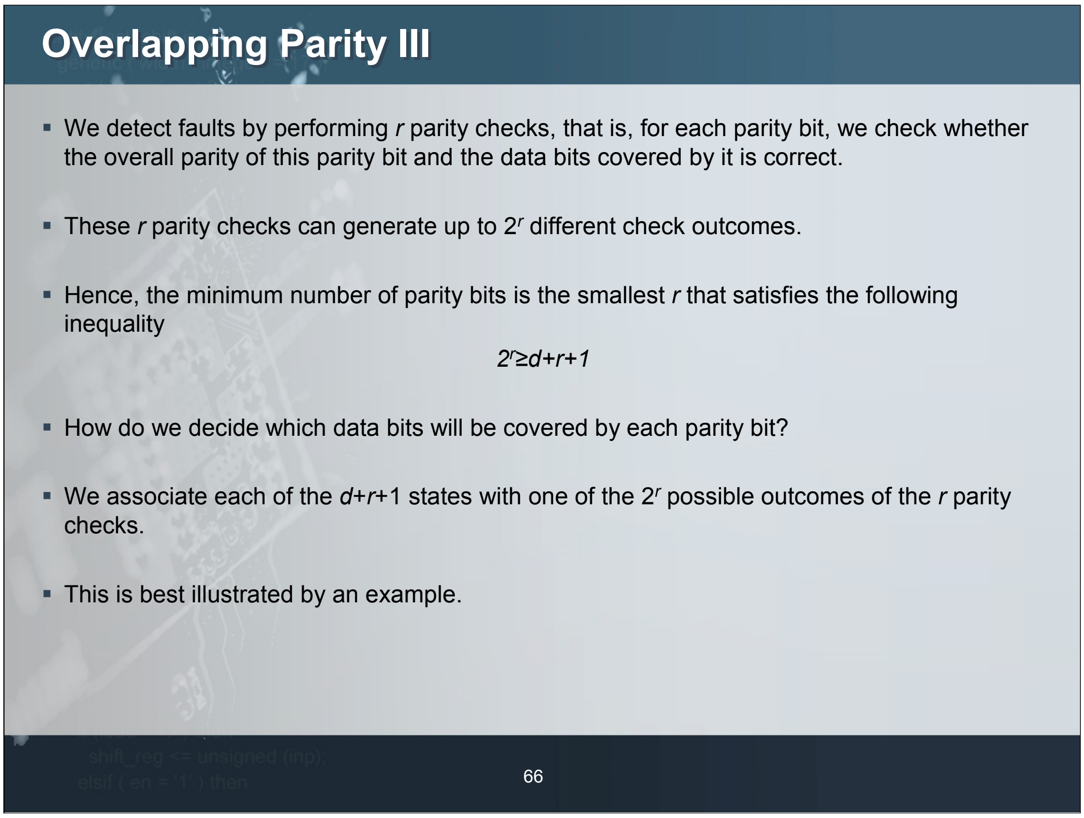
0	0	0	1	1	1	**1**
1	0	1	0	1	1	**0**
1	1	0	0	0	0	**0**
0	0	0	1	1	1	**1**
1	1	1	1	1	1	**0**
1	**0**	**0**	**1**	**0**	**0**	**0**

Example of overlapping parity



Overlapping Parity II

- The previous slide showed an example of *overlapping parity*, in which each bit is “covered” by more than one parity bit.
- We next describe the general theory associated with overlapping parity. Our aim is to be able to identify every single erroneous bit.
- Suppose there are *d* data bits in all. How many parity bits should be used and which bits should be covered by each parity bit?
- Let *r* be the number of parity bits (check bits) that we add to the *d* data bits resulting in codewords of size *d* + *r* bits.
- Hence, there are *d* + *r* error states, where in state *i* the *i*th bit of the codeword is erroneous (keep in mind that we are dealing only with single-bit errors: this scheme will not detect all double-bit errors).
- In addition, there is the state in which no bit is erroneous, resulting in *d*+*r*+1 states to be distinguished.



Overlapping Parity III

- We detect faults by performing *r* parity checks, that is, for each parity bit, we check whether the overall parity of this parity bit and the data bits covered by it is correct.

- These *r* parity checks can generate up to 2^r different check outcomes.

- Hence, the minimum number of parity bits is the smallest *r* that satisfies the following inequality

$$2^r \geq d+r+1$$

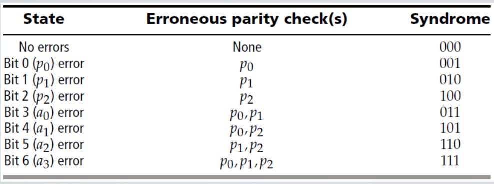
- How do we decide which data bits will be covered by each parity bit?

- We associate each of the $d+r+1$ states with one of the 2^r possible outcomes of the *r* parity checks.

- This is best illustrated by an example.

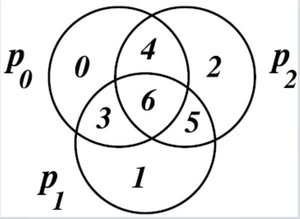
Overlapping Parity Example I

- Suppose we have d = 4 data bits, $a_3a_2a_1a_0$.
- From Equation shown on the previous slide we know that r = 3 is the minimum number of parity bits, which we denote by $p_2p_1p_0$.
- There are 4 + 3 + 1 = 8 states that the codeword can be in.
- The complete 7-bit codeword is $a_3a_2a_1a_0p_2p_1p_0$, i.e., the least significant bit positions 0, 1, and 2 are parity bits and the others are data bits.
- Table on the right shows one possible assignment of parity check outcomes to the states, which is also illustrated in Figure shown on the right.
- The assignment of no errors in the parity checks to the "no errors" state is obvious, as is the assignment for the next three states for which only one parity check is erroneous.
- The assignment of the bottom four states (corresponding to an error in a data bit) to the remaining four outcomes of the parity checks can be done in 4! ways.
- One of these is shown in Table and Figure on the right.
- For example, if the two checks of p_0 and p_2 (and only these) are in error, that indicates a problem with bit position 4, which is a_1.

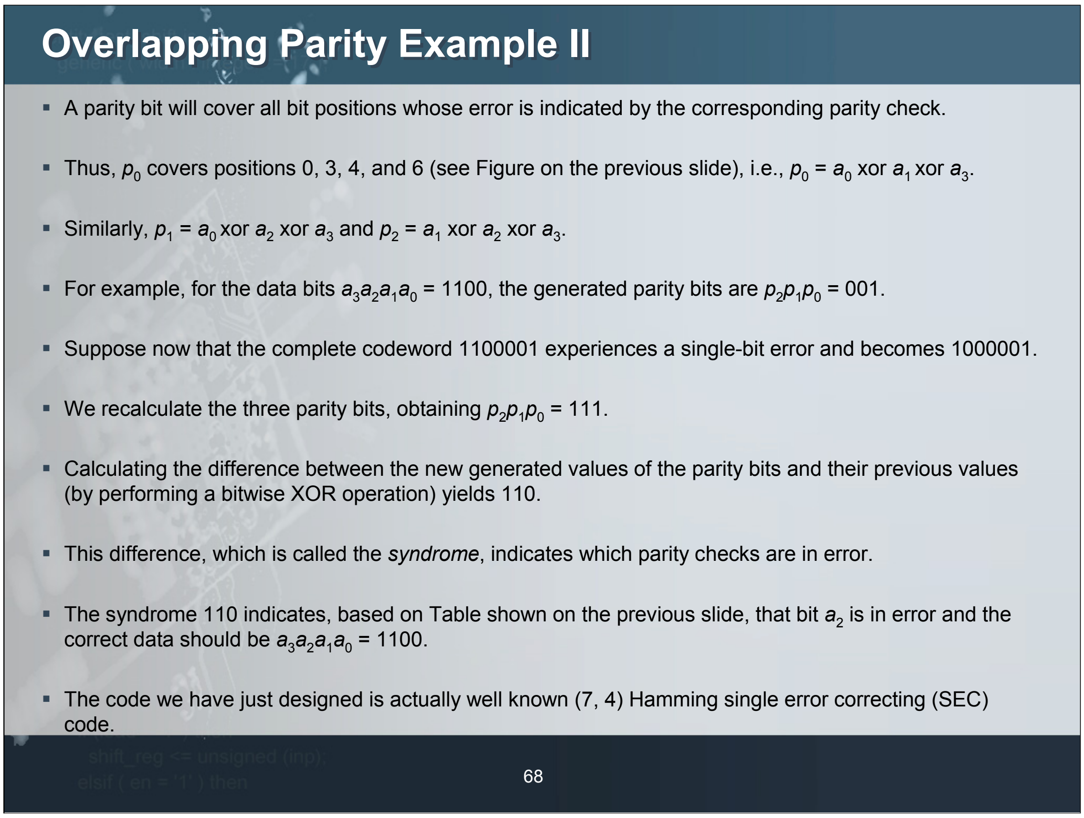


State	Erroneous parity check(s)	Syndrome
No errors	None	000
Bit 0 (p_0) error	p_0	001
Bit 1 (p_1) error	p_1	010
Bit 2 (p_2) error	p_2	100
Bit 3 (a_0) error	p_0, p_1	011
Bit 4 (a_1) error	p_0, p_2	101
Bit 5 (a_2) error	p_1, p_2	110
Bit 6 (a_3) error	p_0, p_1, p_2	111

Example of assignment of parity values to states

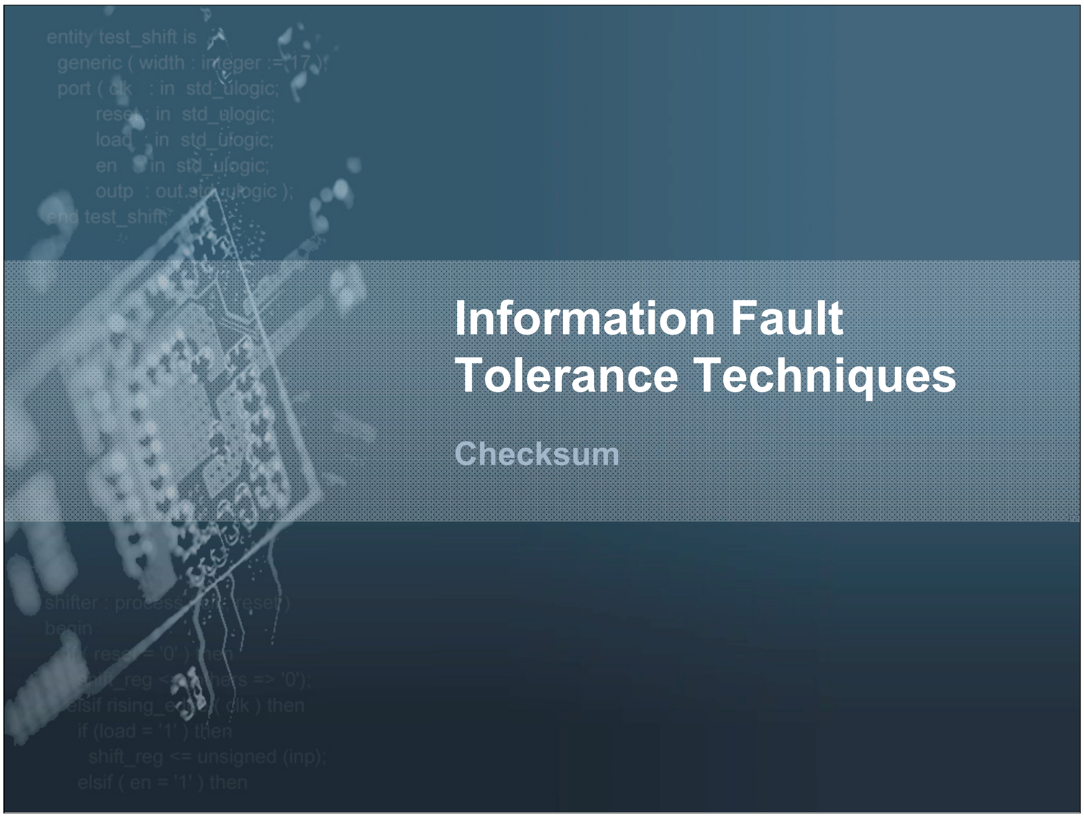


The assignment of parity bits in Table



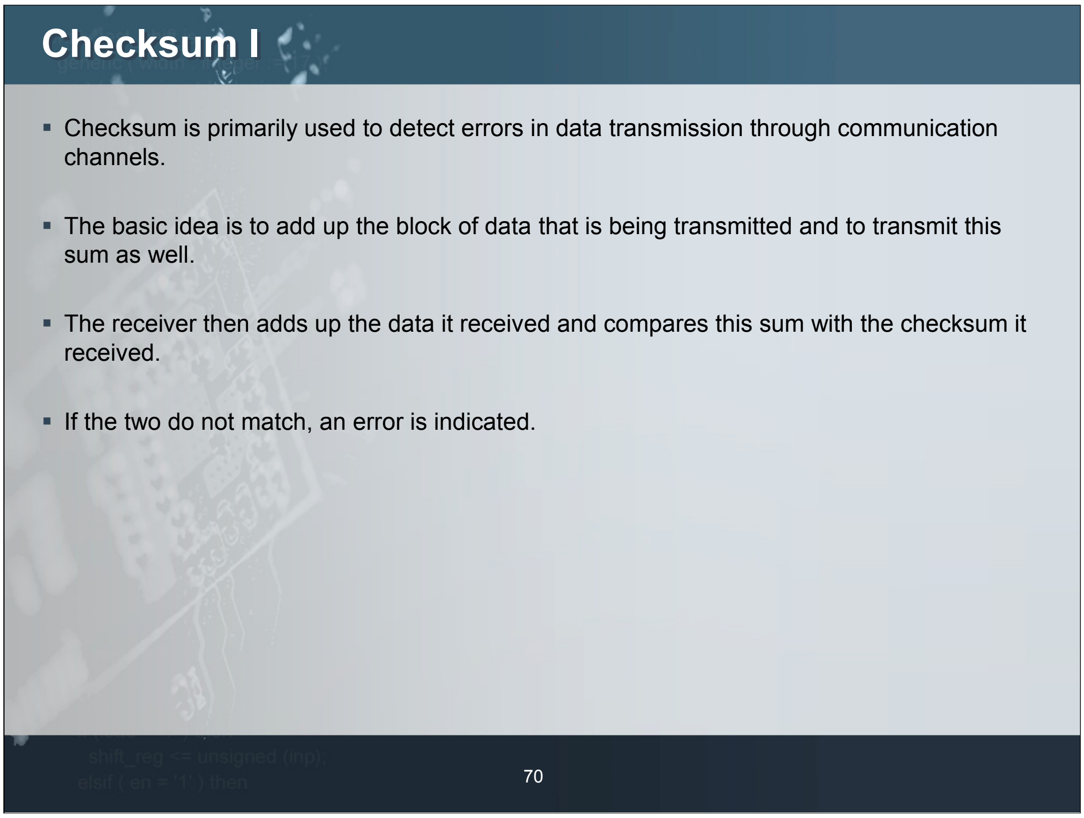
Overlapping Parity Example II

- A parity bit will cover all bit positions whose error is indicated by the corresponding parity check.
- Thus, p_0 covers positions 0, 3, 4, and 6 (see Figure on the previous slide), i.e., $p_0 = a_0$ xor a_1 xor a_3.
- Similarly, $p_1 = a_0$ xor a_2 xor a_3 and $p_2 = a_1$ xor a_2 xor a_3.
- For example, for the data bits $a_3a_2a_1a_0$ = 1100, the generated parity bits are $p_2p_1p_0$ = 001.
- Suppose now that the complete codeword 1100001 experiences a single-bit error and becomes 1000001.
- We recalculate the three parity bits, obtaining $p_2p_1p_0$ = 111.
- Calculating the difference between the new generated values of the parity bits and their previous values (by performing a bitwise XOR operation) yields 110.
- This difference, which is called the *syndrome*, indicates which parity checks are in error.
- The syndrome 110 indicates, based on Table shown on the previous slide, that bit a_2 is in error and the correct data should be $a_3a_2a_1a_0$ = 1100.
- The code we have just designed is actually well known (7, 4) Hamming single error correcting (SEC) code.



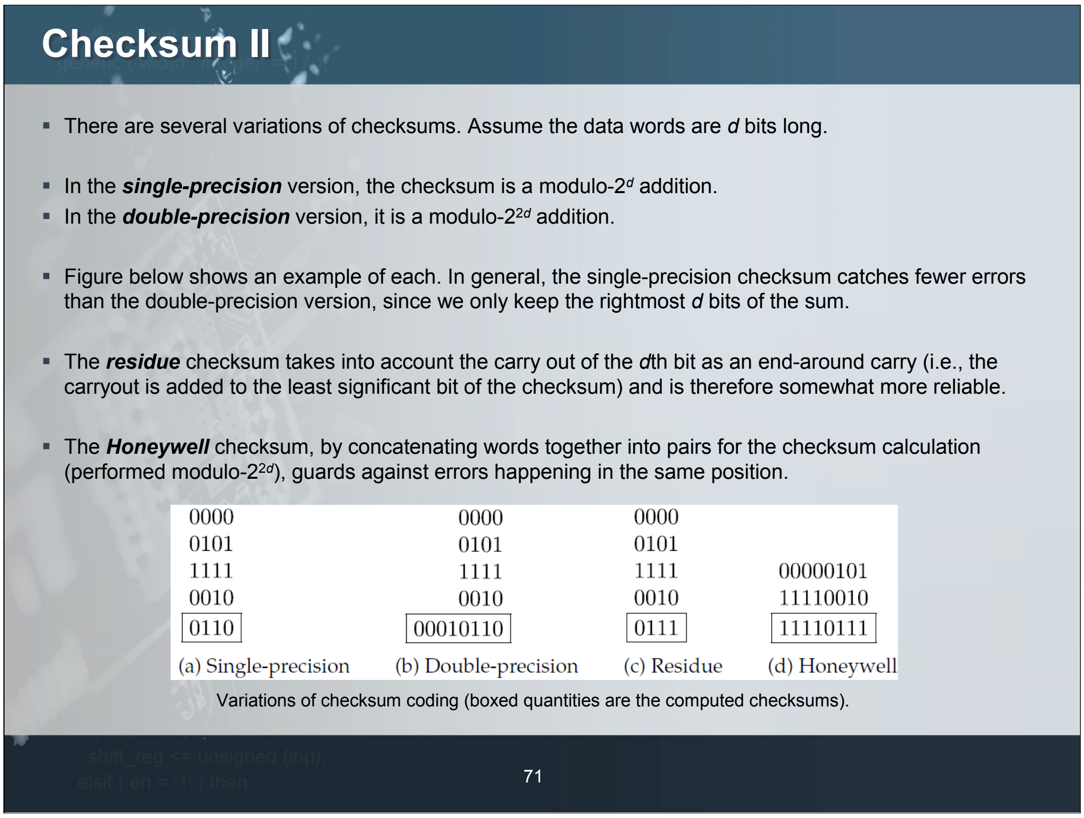
Information Fault Tolerance Techniques

Checksum



Checksum I

- Checksum is primarily used to detect errors in data transmission through communication channels.
- The basic idea is to add up the block of data that is being transmitted and to transmit this sum as well.
- The receiver then adds up the data it received and compares this sum with the checksum it received.
- If the two do not match, an error is indicated.



Checksum II

- There are several variations of checksums. Assume the data words are *d* bits long.

- In the ***single-precision*** version, the checksum is a modulo-2^d addition.
- In the ***double-precision*** version, it is a modulo-2^{2d} addition.

- Figure below shows an example of each. In general, the single-precision checksum catches fewer errors than the double-precision version, since we only keep the rightmost *d* bits of the sum.

- The ***residue*** checksum takes into account the carry out of the *d*th bit as an end-around carry (i.e., the carryout is added to the least significant bit of the checksum) and is therefore somewhat more reliable.

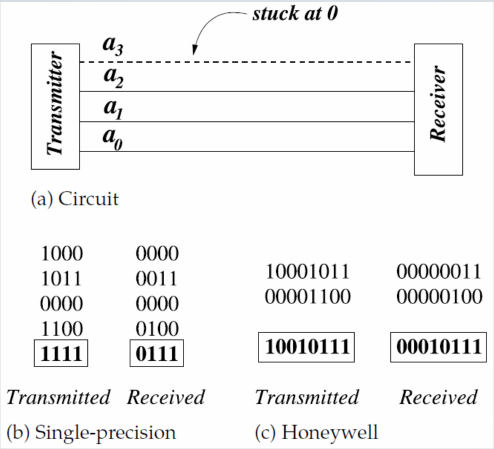
- The ***Honeywell*** checksum, by concatenating words together into pairs for the checksum calculation (performed modulo-2^{2d}), guards against errors happening in the same position.

0000	0000	0000	
0101	0101	0101	
1111	1111	1111	00000101
0010	0010	0010	11110010
$\boxed{0110}$	$\boxed{00010110}$	$\boxed{0111}$	$\boxed{11110111}$
(a) Single-precision	(b) Double-precision	(c) Residue	(d) Honeywell

Variations of checksum coding (boxed quantities are the computed checksums).

Checksum III

- For example, consider the situation in shown Figure below.
- Because the line carrying a_3 is stuck at 0, the receiver will find that the transmitted checksum and its own computed checksum match in the single-precision checksum.
- However, the Honeywell checksum, when computed on the received data, will differ from the received checksum and the error will be detected.
- All the checksum schemes allow error detection but not error location, and the entire block of data must be retransmitted if an error is detected.



Honeywell versus single-precision checksum
(boxed quantities indicate transmitted/received checksum).