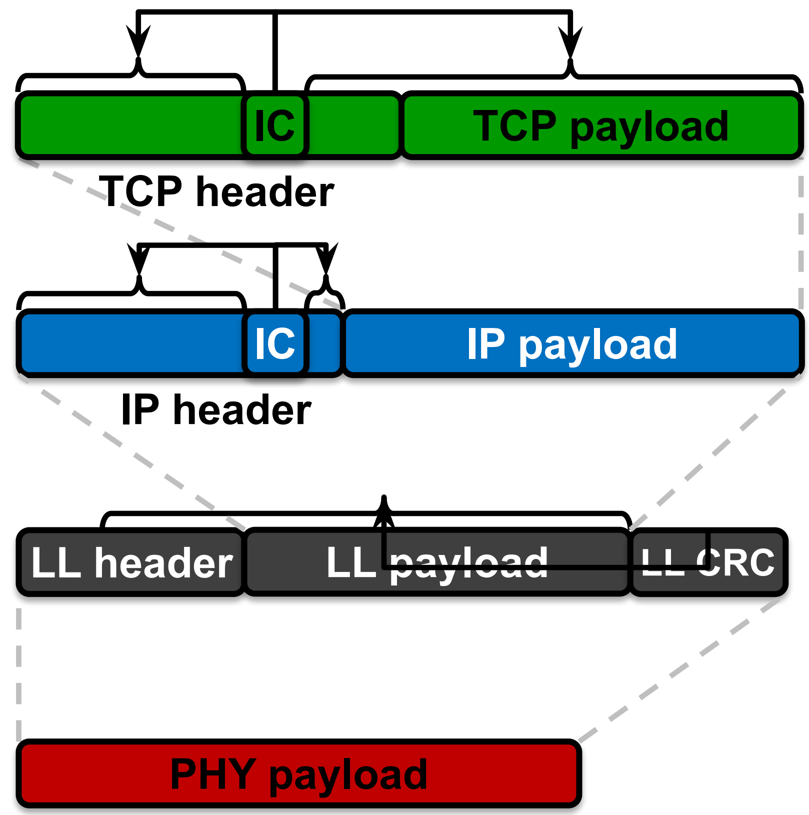
Today

Practical Error control codes

- Internet checksum
- Hamming block code
- Parity check

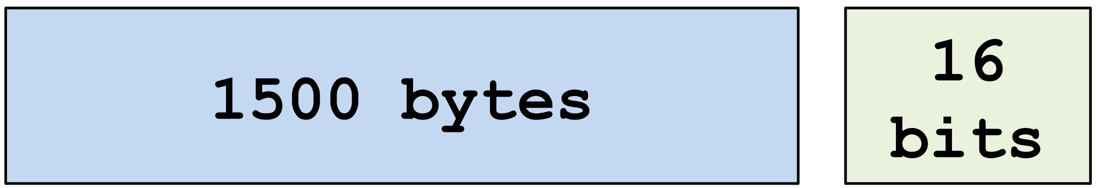
Error control in the Internet stack

- **Transport layer**
  - **Internet Checksum (IC)** over TCP/UDP header, data
- **Network layer (L3)**
  - **IC** over IP header only
- **Link layer (L2)**
  - **Cyclic Redundancy Check (CRC)**
- **Physical layer (PHY)**
  - **Error Control Coding (ECC),** or
  - **Forward Error Correction (FEC)**



Checksums

- Idea: sum up data in N-bit words
  - Widely used in, e.g., TCP/IP/UDP



- Stronger protection than parity

Internet Checksum

- Sum is defined in 1s complement arithmetic (must add back carries)
  - And it's the negative sum
- "*The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words …*" – RFC 791

Internet Checksum

Sending:

1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum

```
0001
f203
f4f5
f6f7
```

Internet Checksum

Sending:

1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum

```
    0001
    f203
    f4f5
    f6f7
 +(0000)
  ------
   2ddf0

    ddf0
  +    2
  ------
    ddf2

    220d
```

Internet Checksum

Receiving:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

```
  0001
  f203
  f4f5
  f6f7
+ 220d
------
```

Internet Checksum

Receiving:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

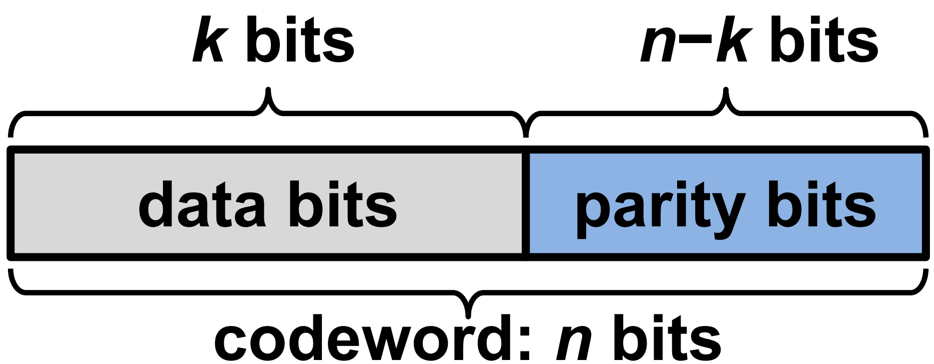
```
   0001
   f203
   f4f5
   f6f7
 + 220d
 ------
  2fffd
   ↓
   fffd
 +    2
 ------
   ffff
   ↓
   0000
```

Internet Checksum

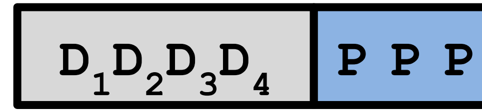
- How well does the checksum work?
  – What is the distance of the code?
  – How many errors will it detect/correct?
- What about larger errors?

Block codes

- Let's **fully generalize the parity bit** for even more error detecting/correcting power
- Split message into ***k*-bit blocks,** and **add *n*−*k* parity bits** to the end of each block:
  - This is called an **(*n*, *k*) block code**

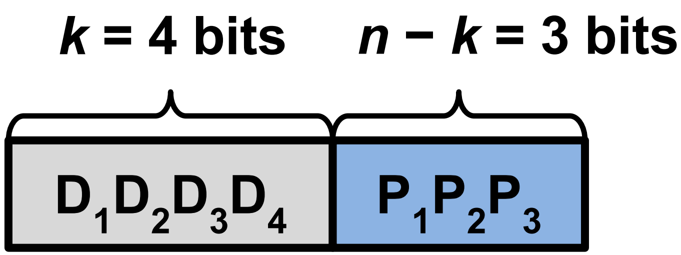


How to design a block code?

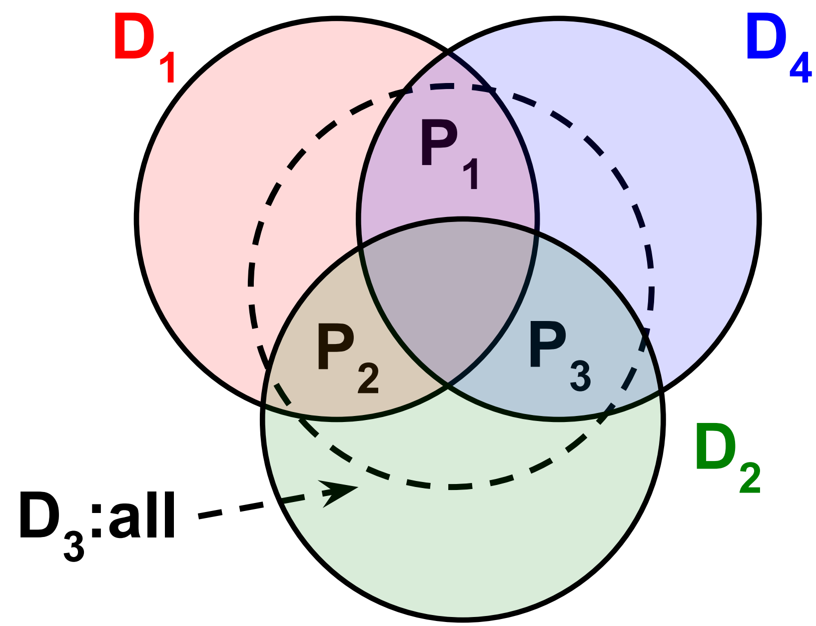


- What if we **repeat the parity bit 3×?**
  - $P = D_1 \oplus D_2 \oplus D_3 \oplus D_4$; $R = 4/7$
  - Flip one data bit, all parity bits flip. So $\boldsymbol{d_{min}} = \mathbf{4}$**?**
    - **No!** Flip another data bit, all parity bits flip back to original values! So $\boldsymbol{d_{min}} = \mathbf{2}$
  - **What happened?** Parity checks either **all failed or all succeeded,** giving **no additional information**

Hamming (7, 4) code

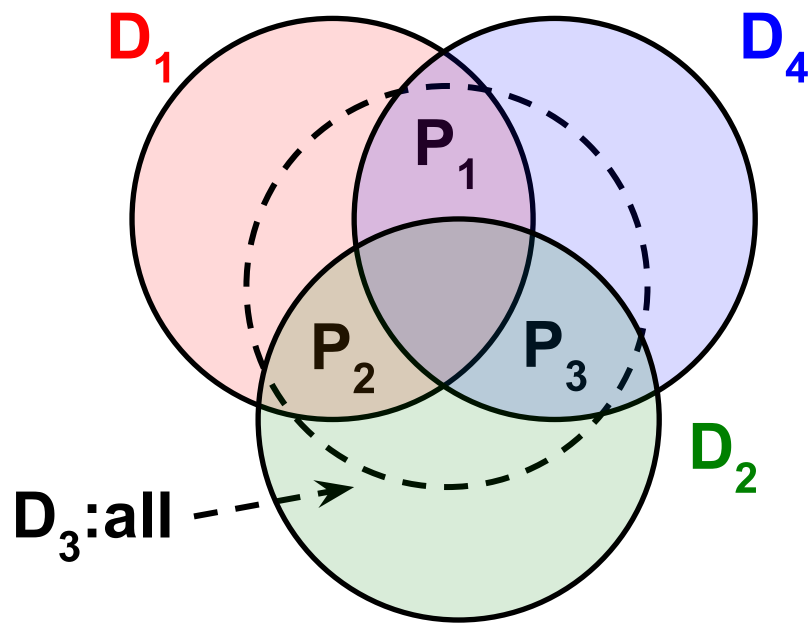


$$P_1 = D_1 \oplus D_3 \oplus D_4$$
$$P_2 = D_1 \oplus D_2 \oplus D_3$$
$$P_3 = D_2 \oplus D_3 \oplus D_4$$



Hamming (7, 4) code: d_{min}

- **Change one data bit,** either:
  - ⇨ Two P_i change, or
  - – Three P_i change
- Change two data bits, either:
  - – Two P_i change, or
  - ⇨ One P_i changes



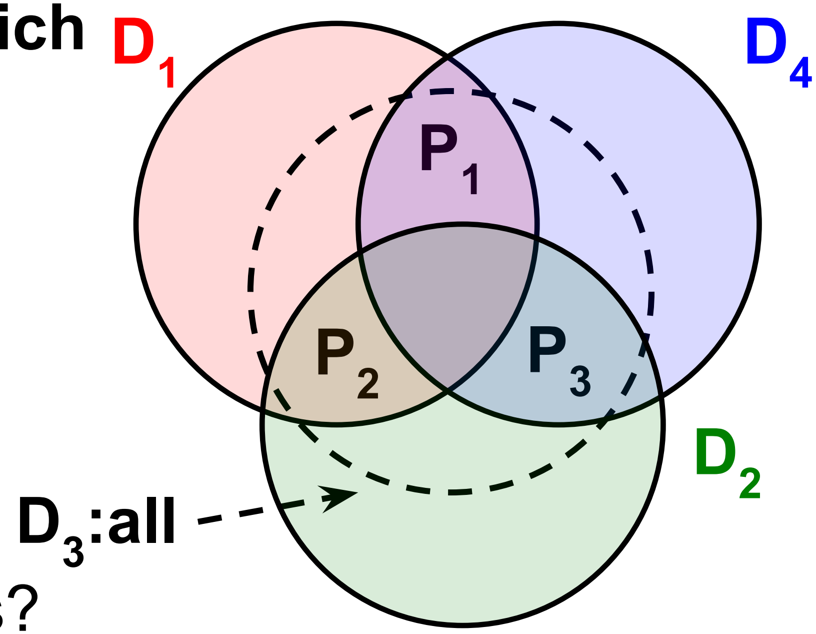
d_{min} = 3: Detect 2 bit errors, correct 1 bit error

Hamming (7, 4): Correcting One Bit Error

- Infer which **corrupt bit** from **which parity checks fail:**

- P_1 and P_2 fail ⇒ Error in **D_1**
- P_2 and P_3 fail ⇒ Error in **D_2**
- P_1, P_2, & P_3 fail ⇒ Error in **D_3**
- P_1 and P_3 fail ⇒ Error in **D_4**

- What if just **one** parity check fails?
  – Then there are multiple errors



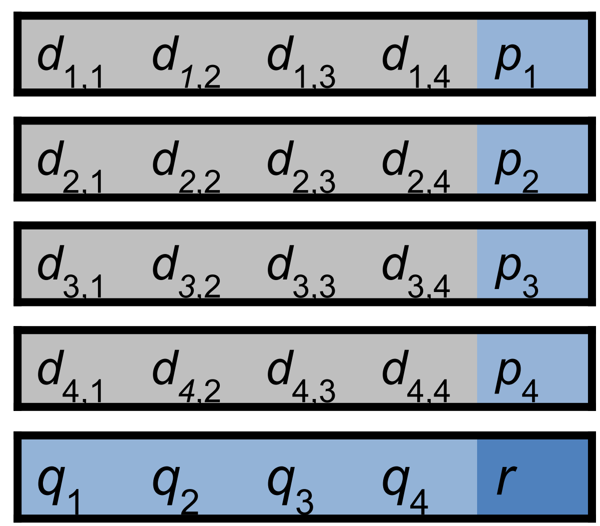
Summary: Higher rate (R = 4/7) code correcting one bit error

Two-dimensional parity

- Break up data into multiple rows
  - Parity bit **across** each row (p_i)
  - Parity bit **down each column** (q_i)
  - Add a parity bit r covering row parities

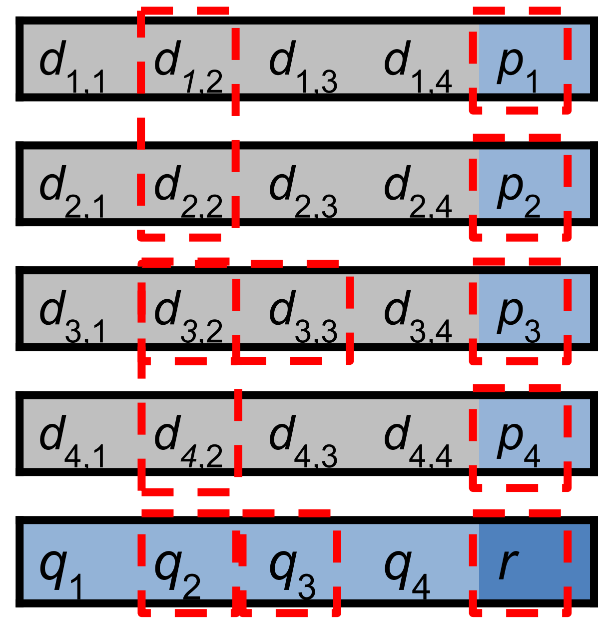
$$p_j = d_{j,1} \oplus d_{j,2} \oplus d_{j,3} \oplus d_{j,4}$$
$$q_j = d_{1,j} \oplus d_{2,j} \oplus d_{3,j} \oplus d_{4,j}$$
$$r = p_1 \oplus p_2 \oplus p_3 \oplus p_4$$

- This example has rate 16/25:



Two-dimensional parity: Properties

- Flip **1 data bit, 3 parity bits** flip
- Flip **2 data bits, ≥ 2 parity bits** flip
- Flip **3 data bits, ≥ 3 parity bits** flip



- Therefore, $\boldsymbol{d_{min}}$ **= 4,** so
  - Can detect ≤ 3 bit errors
  - Can correct single-bit errors (*how?*)
  - d_{min} = 4 because some 4 bit changes that lead to a new codeword, but not 3 or fewer bit changes
  - Single bit errors are corrected by identifying the row/column that don't match up

- 2-D parity detects **most** four-bit errors
  - Example exception: any square of *d* values