Week 3: SECDED coding

Hamming code can detect and correct single-bit errors – for 2 errors they falsely indicate which bits are in error and for > 2 errors they might not even detect an error.

- Single-error c***orrect, double-error detect (SECDED):***
  – commonly used to protect computer memory
  – can correct a single bit error
  – will detect a double bit error (but not correct it)
    • double bit error in computer memory very unlikely
- SECDED coding adds a parity bit to a Hamming code
  – P0 (so it becomes the new LSB)
- So our adding even SECDED to our previous even Hamming example:
  – P0 is 0 in this example to keep even parity overall of the 8 bit total code (including data and hamming code).

Bit Position	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Data/Parity	D4	D3	D2	P3	D1	P2	P1	**P0**
	1	0	1	0	1	0	1	**?**

SECDED: 2 bit error case

- SECDED coding with even parity and 2 errors:
  – if the parity bit is one of the errors, then the Hamming code check bits would show the location of the other error, but the problem is we don't actually know for sure that the parity bit is in error, and so we can't act on the other check bits
  – if the parity bit hadn't been corrupted then both errors are in the Hamming code, which will detect the error but can't be used to correct it
- SECDED is commonly used for Error Correction Code (ECC) memory – workstations and servers have ECC RAM, it is less common on PCs

SECDED, 15 bit code,11 bit data

Easier way to calculate

- e.g. assume we have a Hamming 8-bit (data) code with even parity. We have received code 1010 1110 1001 1111.find and repair the error (if any)

Bit Position	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Parity / Bitmask	D11	D10	D9	D8	D7	D6	D5	P8	D4	D3	D2	P4	D1	P2	P1	P0
Data	1	0	1	0	1	1	1	0	1	0	0	1	1	1	1	1
P0/FFFF	1	0	1	0	1	1	1	0	1	0	0	1	1	1	1	1
P1 / AAAA	1		1		1		1		1		0		1		1	
P2 / CCCC	1	0			1	1			1	0			1	1		
P4 / F0F0	1	0	1	0					1	0	0	1				
P8 / FF00	1	0	1	0	1	1	1	0								

– P0 checks all bits = 1010 1110 1001 1111 , odd P0 ✖
– P1 checks bits 15,13,11,9 7,5,3,1 = 1111 1011, odd , but P1 ✖
– P2 checks bits 15,14,11,10 7,6,3,2 = 1011 101 1, even, so P2 ✔
– P4 checks bits 15,14,13,12 7,6,5,4 = 1010 100 1, even, so P4 = ✔
– P8 checks bits 15,14,13,12 11,10,9,8 = 1010 1110, odd but P8 = 1 ✖
– Simply ADD the bit-value for the incorrect P's
  • Error Position = P0 + P1+P8 = 0+1 + 8 = bit 9 = 1, flip it to zero
  • corrected data = 1010 1100 1001 1111 , data = 101 0110 1001

SECDED 32 bit code, 26 bit data

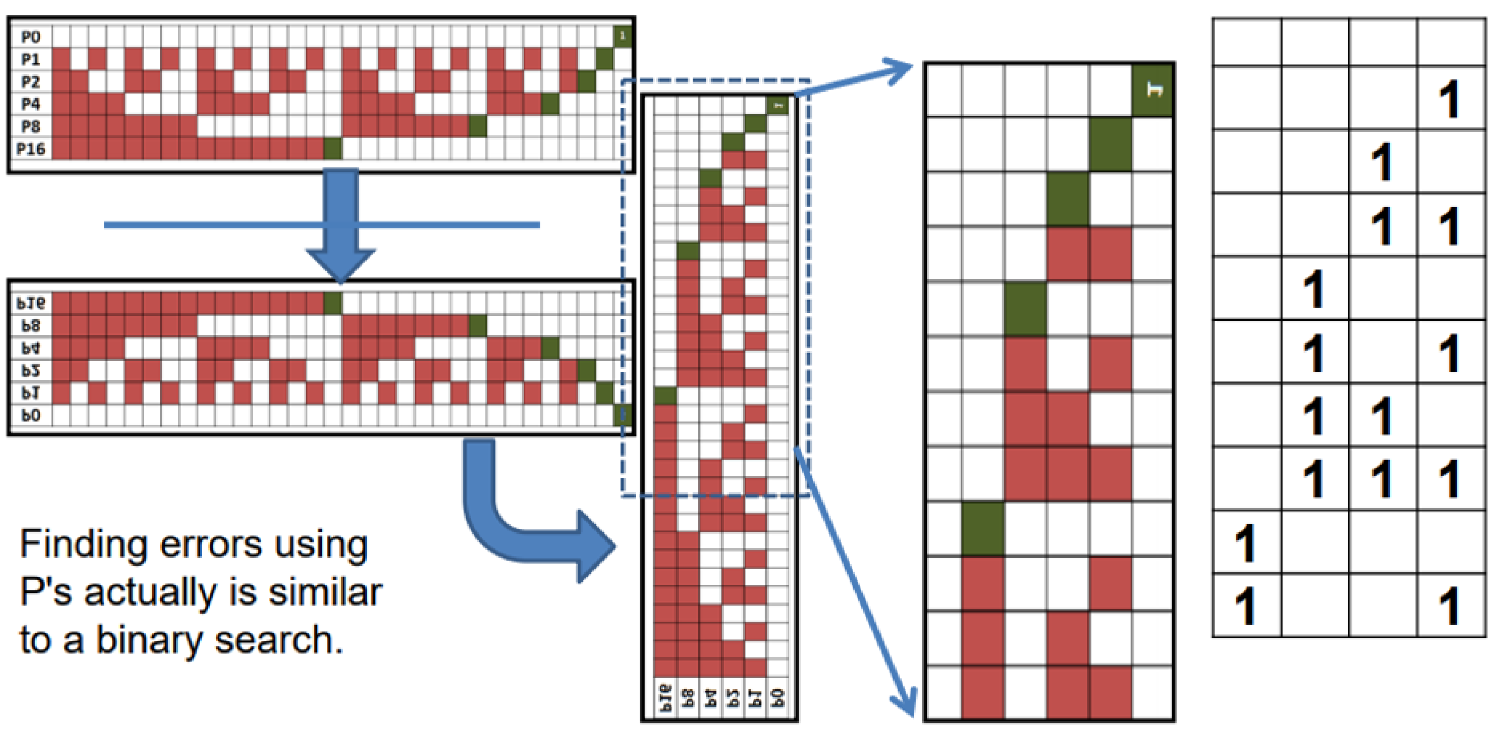
Easier way to calculate

Bit Pos'n																	**6**								**8**				**4**	**3**	**2**	**1**	**0**
Data/Parity	**P0**	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	1	0	0	0	1	0	1	1	1	0	**1**
AAAAAAAA	P1																																
CCCCCCC	P2																																
F0F0F0F0	P4																																
FF00FF00	P8																																
FFFF0000	P16																																

– D = 2D4A AC5D = 0010 1101 0100 1010 1010 1100 0101 1100
– P0 = D & 1 = 1, countbits(D & FFFFFFFF) = 15 (odd) = Error ✖
– P1 = D & 2 = 0, countbits(D & AAAAAAAA) = 8 (even) = Ok ✔
– P2 = D & 4 = 1, countbits(D & CCCCCCCC) = 10 (even) = Ok ✔
– P4 = D & 8 = 1, countbits(D & F0F0F0F0) = 6 (even) = Ok ✔
– P8 = D & 16 = 0, countbits(D & FF00FF00) = 8 (even) = Ok ✔
– P16 = D & 32 = 0, countbits(D & FFFF0000) = 7 (odd) = Error ✖
– Bit-value of incorrect P's = 16 (only P16, not counting P0)
  • So the check bit for p16 is itself in error, so we flip it.
  • Correct data = 2D4B AC5D

SECDED a truth table

- P1-Pn actually correspond to the head (first entry) in each column of a truth table. As the number of bits grows logarithmically with number of values, so does the SECDED number of P's. P0 covers all the bits



SECDED program

```
int16 SECDED_detect_correct(int16 N) {
        // returns N if correct(ed), or 0xFFFF if uncorrectable
        int16 mask[5] = {0xFFFE, 0xAAA8, 0xCCC8, 0xF0E0, oxFE00};
        int16 pPos[5] = {     0,      1,      3,      7,     15};
        int pCalc[5], pStored[5];
        for (int i = 0, bitpos = 0;    i < 5;     i++) {
                pCalc[i] = (countbits(N & mask[i]) & 1);
                pStored[i] = (N & (1 << pPos[i])) > 0 ? 1 : 0;
                if (pCalc[i] != pStored[i]) {    // found an error
                        bitpos += (1 << i);     // (1 << i) is same as 2^i
                }
        }
        boolean error_detected = (bitpos > 0);
        boolean can_correct = (error_detected && (pCalc[0] != pStored[0]));

        int16 newN = N ^ (1 << bitpos);   // Flip the erroneous bit
        if (error_detected)
           if (can_correct)  return newN;      // bit position of error
           else              return 0xFFFF;    // -1 is never a valid code
       else return N;    // corrected or unchanged value
}
```

SECDED coding summary

Summary of SECDED properties

- Assume, for illustration, an even parity SECDED:
  – if no error,
    • parity will be even and Hamming check bits show no error
  – if 1 error,
    • parity will **not** be even, and Hamming check bits will indicate which bit is in error (and we correct it)
  – if 2 errors,
    • parity will be even, but Hamming check bits will indicate an error but not enough info to correct the bits
      – explained further in next slide
  – if >2 errors,
    • SECDED may not detect the error
      – but incredibly unlikely to have >2 errors in memory
- We need 1 check digit for every column in the truth table (+ P0 = m + p +1

Data errors: beyond 2 errors

- Do SECDED codes only detect/correct 1/2 bit?
- Actually, no
  – Hamming detect an ODD number of bits as being in error
  – With P0, SECDED can tell the difference between 0 errors and a non-zero EVEN number of bits in error.
- Where there is a higher probability of errors, we need to use error-tolerant coding. We looked at 1 already
  – Gray codes.
- There are variations of Hamming codes using complicated polynomials which will detect multiple bit errors, but that is beyond this course
- But there is a data tolerance approach which is simpler.
  – This is an extension of the Hamming distance idea and uses hardware correlators.

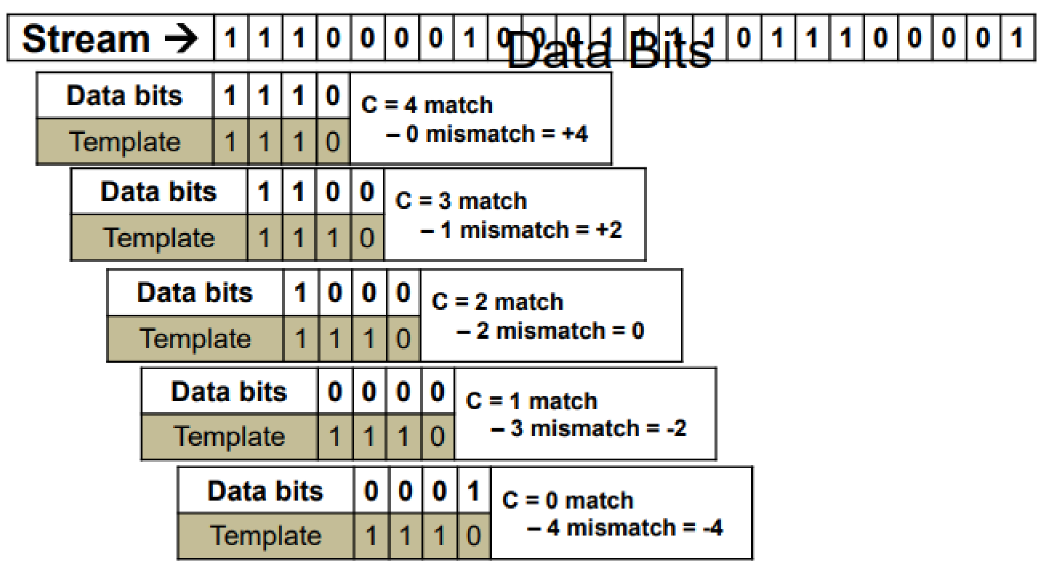
Data error tolerance: using templates

- Suppose if instead of using 1 bit to represent a binary state in a data stream, you use 4 bits.
  – 1110 is a '1' bit, and 0001 is a '0' bit.
  – A typical data stream might be 111000010001111011100001
- We can read these bits using a correlation.
- Binary data correlation is defined as follows:
  Correlation = sum(num of matches) – sum(num of mis-matches)
  – where 'matches' refers to the matches between bit stream bits and the template bits.
- Example

Data Bits→	1	1	1	0	0	0	0	1	0	0	0	1	1	1	1	0	1	1	1	0	0	0	0	1
Mask	1	1	1	0																				

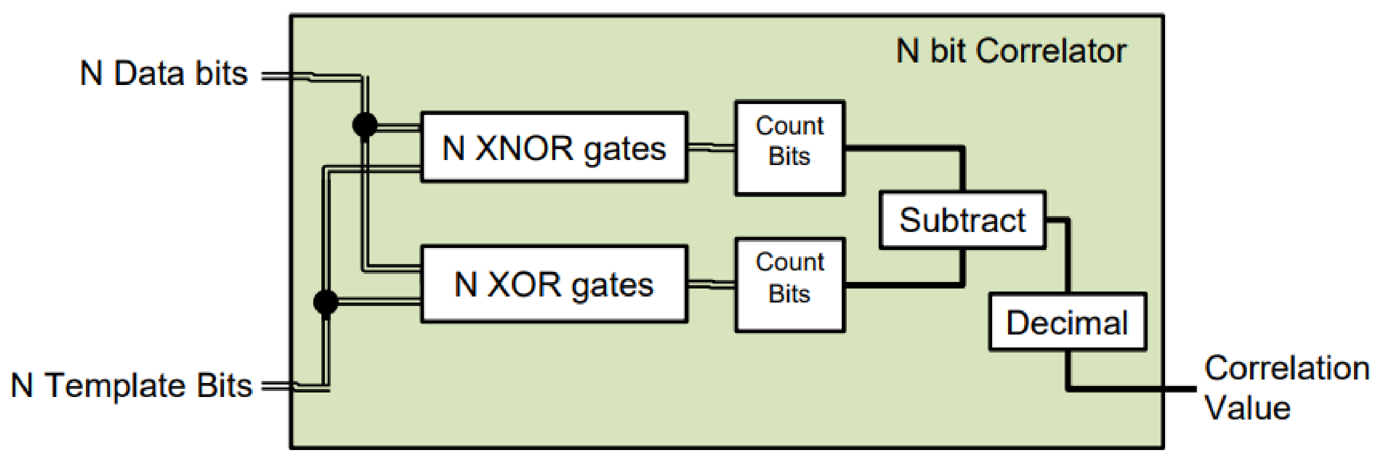
'1' '0' '0' '1' '1' '0'

- Example of Binary Bit Stream Correlation
  – for '1' bit, #matches = 4, #mismatches = 0, Correlation = +4
  – for '0' bit, #matches = 0, #mismatches = 4, Correlation = -4
  – if one of the 4 components of the '1' bit errored, then correlation would be +2 (#matches=3, #mismatches=1)
  – if 2 bits errored, correlation = 0, so we don't know if '1' or '0'

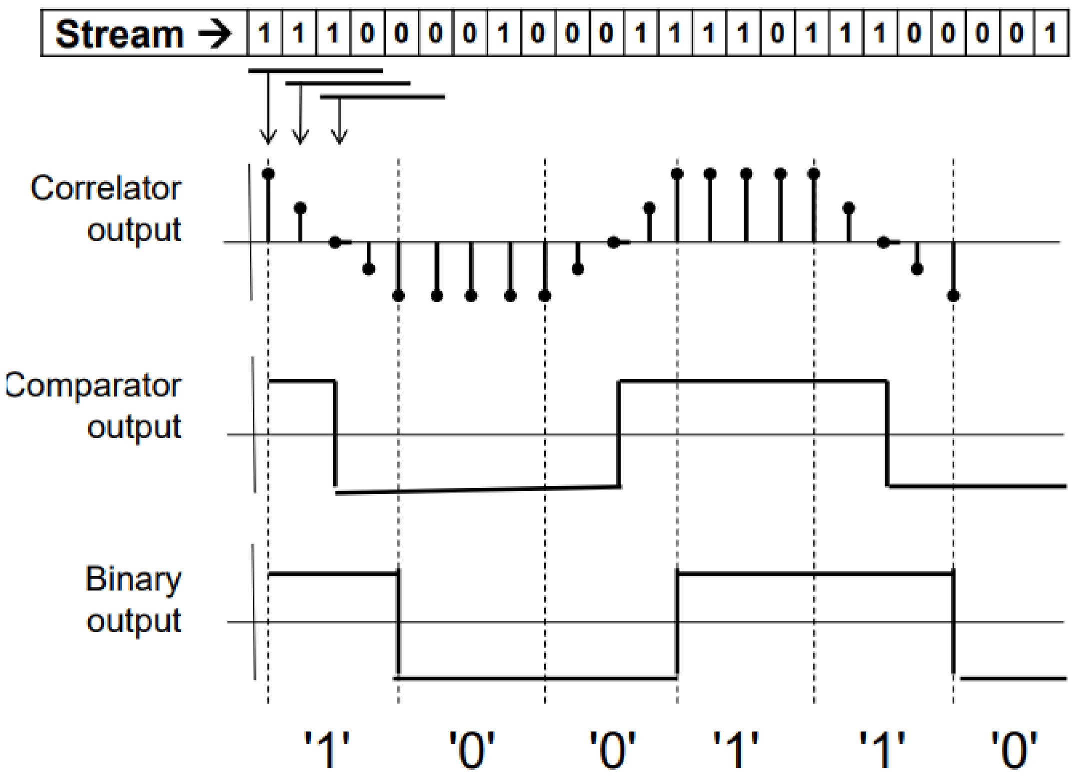


- Example of Binary Bit Stream Correlation
  – for '1' bit, #matches = 4, #mismatches = 0, Correlation = +4
  – for '0' bit, #matches = 0, #mismatches = 4, Correlation = -4
  – if one of the 4 components of the '1' bit errored, then correlation would be +2 (#matches=3, #mismatches=1)
  – if 2 bits errored, correlation = 0, so we don't know if '1' or '0'

Data Bits→	1	1	1	0	0	0	0	1	0	0	0	1	1	1	1	0	1	1	1	0	0	0	0	1



- Binary correlation, for the 4-bit template on data stream.



- So for a 4 bit template, we can allow only up to 2 bit errors, but with larger templates, we can accept more bit errors.
  – Note that not just any binary pattern can be used.
  – Only special patterns have the correlation property
- Suitable template patterns include
  – Barker codes (used in Radar)
    • 1110, 11101, 1110010,11100010010, 1111100110101
  – M-sequences (length = 2n-1)
    • 1110, 1110101, …
  – Gold codes (used in GPS, inter-planetary comms)
  – Many others of length 2n, 2n-1, prime, prime-1….