COSC2473

Data Error Handling

Intro and Gray Codes

Parity and Hamming Codes

<u>SECDED and Templates</u>

SECDED coding

- Hamming code can detect and correct single-bit errors
  - for 2 bit errors it falsely indicates which bit(s) are in error and for > 2 errors it might not even detect an error

- ***<u>Single-error correct, double-error detect (SECDED)</u>*** :
  - commonly used to protect computer memory
  - can correct a single bit error
    - single bit error possible due to signal ‘glitch’
  - will detect a double bit error (but not correct it)
    - double bit error in computer memory very unlikely

SECDED coding

- SECDED coding adds a parity bit to a Hamming code
  - P0 (so it becomes the new LSB)
  - That's it !!

- So we add 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).
  - So for data = 1011, the SECDED code = 10101010_2

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 is not 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, 8 bit code, 4 bit data A

Easier way to calculate

- e.g. assume we have a Hamming 4-bit (data) code with even parity. We have received code 45_{16} = 100 0101_2.
  - find and repair the error (if any)

Orig Data				Code						
Bit Position	7	6	5	4	3	2	1	0	Calc	✗ ✓
Parity | Bitmask	D4	D3	D2	P4	D1	P2	P1	P0	P's	
P0 | FF								?		
P1 | AA							?			
P2 | CC						?				
P4 | F0				?						
Correct Bit(s)										
Correct Data				Code						

SECDED, 8 bit code, 4 bit data A

Easier way to calculate

- e.g. assume we have a Hamming 4-bit (data) code with even parity. We have received code 45_{16} = $0100\ 0101_2$.
  - find and repair the error (if any)

Bit Position	7	6	5	4	3	2	1	0
Parity | Bitmask	D4	D3	D2	P4	D1	P2	P1	P0
P0 | FF	0	1	0	0	0	1	0	1?
P1 | AA	0		0		0		0?	
P2 | CC	0	1			0	1?		
P4 | F0	0	1	0	0?				

P0 Data (45 = 0100 010? Sum(Data)=2, even but P0 = 1 ✘

P1 checks bits 7,5,3,1 Sum(000?) = 0, even so P1 = 0 ✔

P2 checks bits 7,6,3,2 Sum(010?) = 1, odd so P2 = 1 ✔

P4 checks bits 7,6,5,4 Sum(010?)=1, odd but P4 = 0 ✘

- So the error must be in bits 7,6,5,4.
- P2 checks bits 7.6 and P2 is OK, , so it is not bit 6,7
- P1 checks bit 5, and P1 is OK, so it is not bit 5
- error must be bit 4 – corrected code is 0101 0101 = 55_{16}
- corrected data = 0101 = 5.

SECDED, 15 bit code,11 bit data C

Easier way to calculate

- e.g. assume we have a Hamming 8-bit (data) code with even parity. We have received code $30B9_{16} = 0011\ 0000\ 1011\ 1001_2$.
  - find and repair the error (if any)

Orig Data						**Code**												
Bit Position	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Calc**	✗ ✓
Data / Parity	**D11**	**D10**	**D9**	**D8**	**D7**	**D6**	**D5**	**P8**	**D4**	**D3**	**D2**	**P4**	**D1**	**P2**	**P1**	**P0**	**P's**	
P0 | FFFF																		
P1 | AAAA																		
P2 | CCCC																		
P4 | F0F0																		
P8 | FF00																		
Correct Bit(s)																		
Correct Data						**Code**												

SECDED, 15 bit code,11 bit data C

Easier way to calculate

- e.g. assume we have a Hamming 8-bit (data) code with even parity. We have received code $30B9_{16}$ = $0011\ 0000\ 1011\ 1001_2$.
  - 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
P0 | FFFF	0	0	1	1	0	0	0	0	1	0	1	1	1	0	0	1?
P1 | AAAA	0		1		0		0		1		1		1		0?	
P2 | CCCC	0	0			0	0			1	0			1	0?		
P4 | F0F0	0	0	1	1					1	0	1	1?				
P8 | FF00	0	0	1	1	0	0	0	0?								

P0 Data (30B9 = 0011 0000 1011 100?. Sum(Data)=6, even but P0 = 1

P1 checks bits 15,13,11,9 7,5,3,1 Sum(0010 111?)=4 , even so P1 = 0

P2 checks bits 15,14,11,10 7,6,3,2 Sum(0000 101?)=2, even so P2 = 0

P4 checks bits 15,14,13,12 7,6,5, Sum(0011 101?)=4, even but P4 = 1

P8 checks bits 15,14,13,12 11,10,9,8 Sum(0011 000?)=2,even so P8 = 0

- Simply ADD the bit-value for the incorrect P's
  - Error Position = P0 + P4 = 4, so flip b_4 = 1, to 0,
  - **corrected code = 0011 0000 1010 1001 = $30A9_{16}$ for data 001 1000 1011 = $18B_{16}$**

SECDED, 15 bit code,11 bit data C

Easier way to calculate

- e.g. assume we have a Hamming 8-bit (data) code with even parity. We have received code $AE9A_{16} = 1010\ 1110\ 1001\ 1010_2$.
  - 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**
P0 | FFFF	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**0?**
P1 | AAAA	**1**		**1**		**1**		**1**		**1**		**0**		**1**		**1?**	
P2 | CCCC	**1**	**0**			**1**	**1**			**1**	**0**			**1**	**0?**		
P4 | F0F0	**1**	**0**	**1**	**0**					**1**	**0**	**0**	**1?**				
P8 | FF00	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0?**								

P0 Data (AE9A = 1010 1110 1001 1010. Sum(Data)=9, odd but P0 = 0

P1 checks bits 15,13,11,9 7,5,3,1 Sum(1111 101)=6 , even but P1 = 1

P2 checks bits 15,14,11,10 7,6,3,2 Sum(1011 101)=5, odd but P2 = 0

P4 checks bits 15,14,13,12 7,6,5, Sum(1010 100)=3, odd so P4 = 1

P8 checks bits 15,14,13,12 11,10,9,8 Sum(1010 111)=5, odd but P8 = 0

- Simply ADD the bit-value for the incorrect P's
  - Error Position = P0 + P1 + P2 + P8 = 0+1+2+8 = b_{11} = 1, flip it to 0,
  - **corrected code = 1010 0110 1001 1010 = $A69A_{16}$ for data 101 0011 1001 = 539_{16}**

SECDED 32 bit code, 26 bit data D

Easier way to calculate

Orig Data	2D4A AC5C											Code	0010 1101 0100 1010 1010 1101 0101 1100																					Calc
Bit Pos'n					28				24				20				16				12				8				4		P2	P1	P0	P's
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	0	
AAAAAAAA	P1																															?		
CCCCCCCC	P2																														?			
F0F0F0F0	P4																												?					
FF00FF00	P8																								?									
FFFF0000	P16															?																		
Correct Bit(s)																																		
Correct Data												Code																						

D = 2D4A AC5C = 0010 1101 0100 101**0** 1010 110**0** 0101 1**100**

SECDED 32 bit code, 26 bit data D

Easier way to calculate

Bit Pos’n																	16								8				4		2	1	0
Data + Parity	**P0**	0	0	1	0	1	0	0	1	0	1	0	0	1	0	1	1	1	0	1	0	1	1	0	0	0	1	0	1	1	1	0	0
AAAAAAA8	**P1**																															0	
CCCCCCC8	**P2**																														1		
F0F0F0E0	**P4**																												1				
FF00FE00	**P8**																								0								
FFFE0000	**P16**															1																	

D = 294B AC5C = 0010 1001 0100 1011 1010 110**0** 0101 1**100**

P0 = D & 0x01 = 0, count1bits(D & 0xFFFFFFFE) = 15 (odd) = Error ✘

P1 = D & 0x02 = 0, count1bits(D & 0xAAAAAAA8) = 8 (even) = Ok ✓

P2 = D & 0x04 = 0, count1bits(D & 0xCCCCCCC8)= 8 (even) = Error ✘

P4 = D & 0x10 = 1, count1bits(D & 0xF0F0F0E0) = 5 (even) = Ok ✓

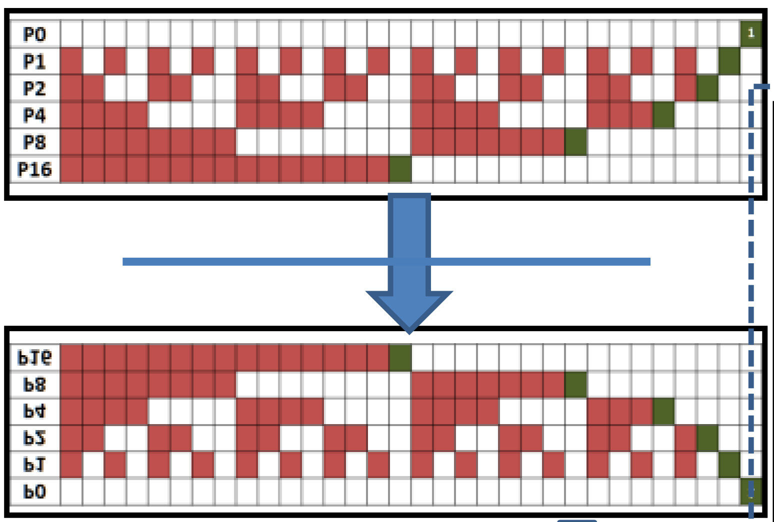
P8 = D & 0x100 = 0, count1bits(D & 0xFF00FE00) = 7 (even) = Error ✘

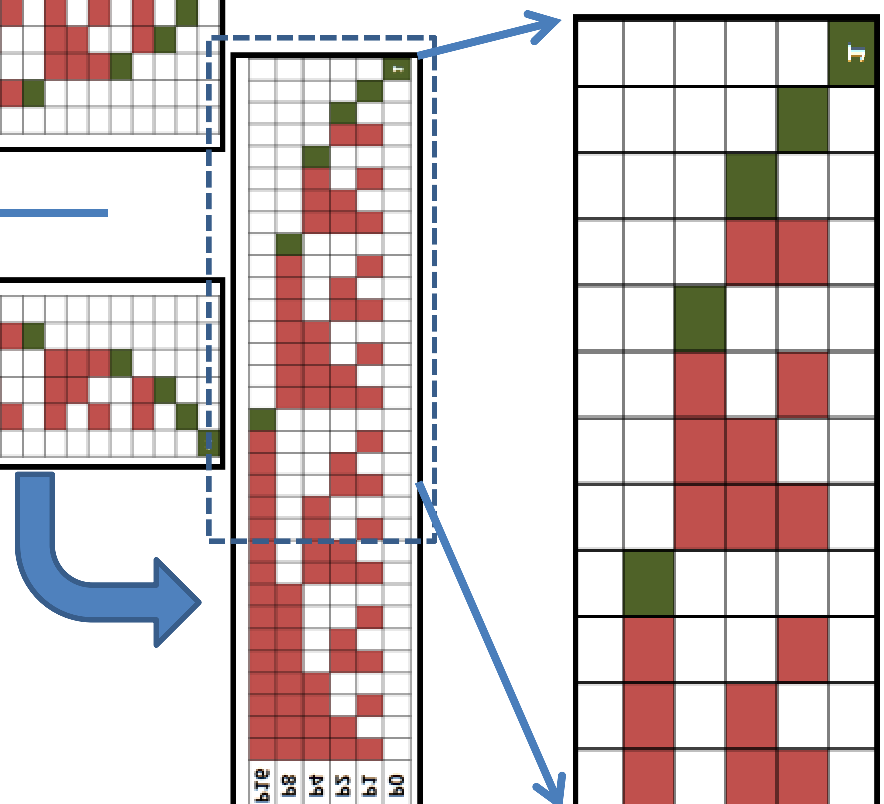
P16 = D & 0x10000=1, count1bits(D & 0xFFFE0000) = 6 (even) = Error ✘

- Bit-value of incorrect P's = 16,8,2 (not counting P0) = 26
  - Correct data = 2D4B AC5D, and without parity = 00 1011 0101 0010 1101 0110 0101

SECDED – a Binary Counter !?

- 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





			1
		1	
		1	1
	1		
	1		1
	1	1	
	1	1	1
1			
1			1

Finding errors using P's actually is similar to a binary search.

SECDED program in Python

```
def SECDED_correct(N, debug=0):
    # returns N if correct(ed), or 0xFFFF if uncorrectable
    mask = [0xFFFFFFFE, 0xAAAAAAA8,0xCCCCCCC8,0xF0F0F0E0,0xFF00FE00,0xfffe0000]
    pPos = [          0,           1,          2,          4,         8,         16]
    pCalc = [0,0,0,0,0,0]
    pStored = [0,0,0,0,0,0]
    bittocorrect = 0
    for i in range(6) :
        pCalc[i] = (count1bits(N & mask[i]) & 31)
        pStored[i] = 1 if (N & (1 << pPos[i])) > 0 else 0
        if (i > 0 and (1&pCalc[i]) != pStored[i]):   # found an error
            bittocorrect += pPos[i]
        if (debug & 1):
            print(i, bittocorrect, pPos[i], pCalc[i], pStored[i])
    error_detected = (bittocorrect > 0)
    can_correct = (error_detected and (1 & pCalc[0] != pStored[0]))
    newN = N ^ (1 << bittocorrect)      # Flip the erroneous bit
    if (debug & 2):
        print("N ",error_detected,can_correct,hex(N),hex(newN))
    if (error_detected):
        if (can_correct):
            return(newN)     #  bit position of error
        else:
            return(-1)       # -1 is never a valid code
    else:
        return(N)            # corrected or unchanged value
```

```
def count1bits(n):
  count = 0
  while n:
    n &= n-1
    count += 1
  return count
```

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
  - **where m = #data bits, p = #check bits, and 2^p >= m+p+1**

Data Errors – Beyond 2 errors

- Do SECDED codes only detect/correct 1 or 2 bits?
- Actually, no
  - Hamming detects 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 and call it a code.
  - Code 1110 is a '1' bit, and code 0001 is a '0' bit.
  - A typical data stream might be 111000010001111011100001
- We can read these bits using a <u>*correlation*</u>.
- <u>***Binary data correlation***</u> is defined as follows:

Correlation = sum(num of matches) – sum(num of <u>mis</u>-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**
Template	1	1	1	0																				
	'1'				'0'				'0'				'1'				'1'				'0'			

Data Error Tolerance: Using Templates

- 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'

Stream →	1	1	1	0	0	0	0	1	0	0	0	1	1	1	1	0	1	1	1	0	0	0	0	1

Data Bits

Data bits	1	1	1	0	C = 4 match – 0 mismatch = +4
Template	1	1	1	0	

Data bits	1	1	0	0	C = 3 match – 1 mismatch = +2
Template	1	1	1	0	

Data bits	1	0	0	0	C = 2 match – 2 mismatch = 0
Template	1	1	1	0	

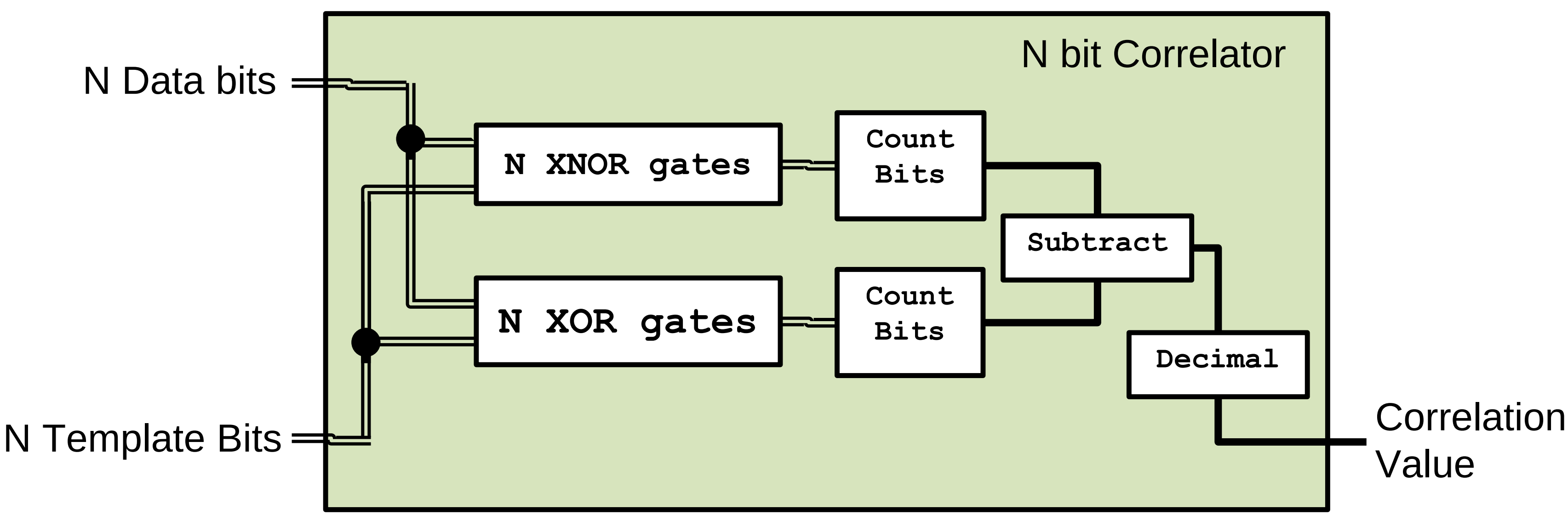
Data bits	0	0	0	0	C = 1 match – 3 mismatch = - 2
Template	1	1	1	0	

Data bits	0	0	0	1	C = 0 match – 4 mismatch = - 4
Template	1	1	1	0	

Data Error Tolerance: Using Templates

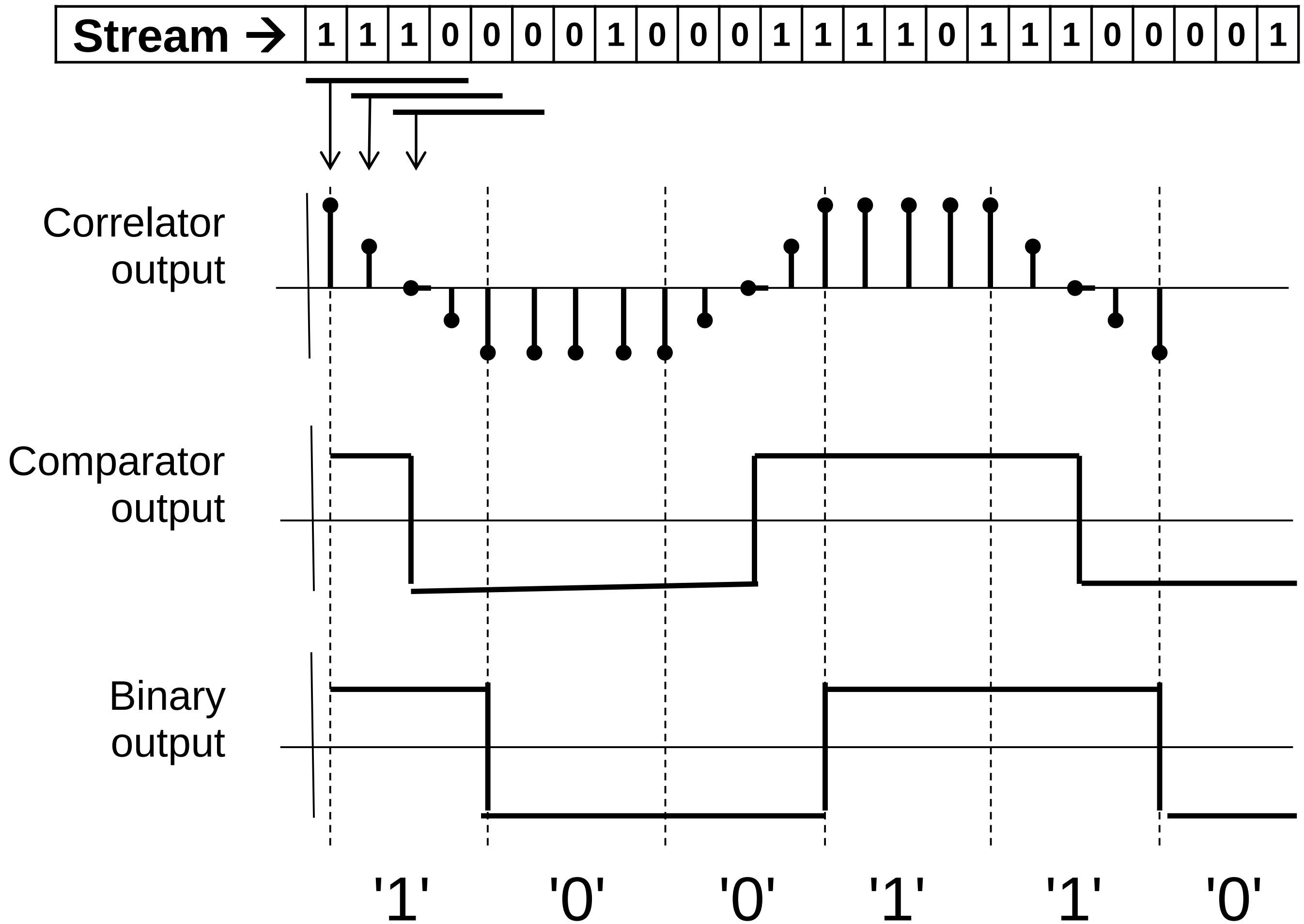
- Example of Binary Bit Stream Correlation
  - for '1' bit, #matches = 4, #mismatches = 0, Correlation = +4
  - for '0' bit, #matches = 0, #mismatches = 4, Correlation = -4
  - if one of the 4 components of the '1' bit errored, then correlation would be +2 (#matches=3, #mismatches=1)
  - if 2 bits errored, 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



Data Error Tolerance: Using Templates

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



Data Error Tolerance: Using Templates

- 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 = 2^n-1), also called LFSR sor PN sequences
    - 1110, 1110101, …
  - Length 1023 Gold codes (used in GPS, inter-planetary comms
  - Many others of length 2^n, 2^n-1, prime, prime-1….

<u>Gold codes are the core of how CDMA works</u>

Data Errors - Summary

- Data Error Tolerance
  - Gray Codes
  - Template Matching
- Data Error Detection
  - Parity
  - Hamming – combine scale, parity and Gray code ideas
- Data Error Correction
  - Hamming
  - SECDED – Combine Hamming with parity to enable detect / correct