Fault-Tolerant Design of Digital Systems

EGE 534

Information Redundancy

Dr. Baback Izadi
Department of Electrical and Computer Engineering and
State University of New York – New Paltz
bai@engr.newpaltz.edu

Outline

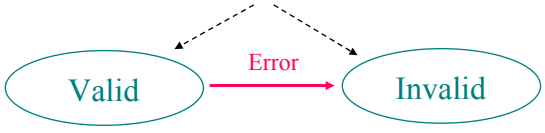
- Error detecting and error correcting codes
- Codes for storage and communication
- Codes for arithmetic operations
- Codes for control units
- Example: Applying the detection techniques used so far
  - A failure resilient node/network controller

Basic Concepts

- Code
  - A mean of representing information using a well defined set of rules
- Codeword
  - A collection of symbols used to represent a data based on specific code
  - Example: BCD code using 0's and 1's
- Valid codeword Vs. invalid codeword
  - Codeword is valid if it adheres to all the rules that defines the code.
- Encoding Process
  - Determining the pertaining codeword from a data item
  - Example: Data item = 7 ➔ 0111
- Decoding Process
  - Recovering the original data from the codeword.
- Separable Code:
  - Original information is appended with the check bits to form the code
- Non separable Code

Basic Concept of Error Detecting/ Correcting

- Error detection code
  - Specific representation allowing error introduced into the code to be detected.
  - n bits ➔ 2^n combinations



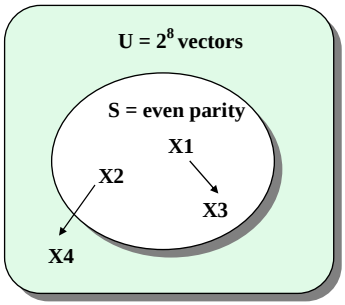
  - Is BCD error detecting code?
- Error correcting code
  - Error detecting initiates reconfiguration
  - Error correction provides fault masking

Fault Detection through Encoding

- At logic level, codes provide means of masking or detection of errors
- Formally, code is a subset S of universe U of possible vectors
- A noncode word is a vector in set U-S

Example:
X1 is a codeword <10010011>
Due to multiple bit error, becomes
X3 = <10011100> **not detectable**

X2 is a codeword, becomes X4 noncode **detectable**



Hamming Distance

- The ***Hamming weight*** of a vector x (e.g., codeword), w(x), is number of nonzero elements of x.
- ***Hamming distance*** between two vectors x and y, d(x,y) is number of bits in which they differ.
- ***Distance of a code*** is a minimum of Hamming distances between all pairs of code words.

Example: x = (1011), y = (0110)

w(x) = 3, w(y) = 2, d(x, y) = 3

Distance Properties

- A code with a distance of 2 can detect a single fault.
- A code with a distance of 3 can correct a single fault **or** detect a double fault.
- To detect all error patterns of *Hamming distance* $\leq$ d, code distance must be $\geq$ d+1
- To correct all error patterns of *Hamming distance* $\leq$ c, code distance must be $\geq 2c + 1$
- To correct c bits, **and** detect d bits, the code distance must be $\geq 2c + d + 1$

Basic Code Operations

- Consider n-bit vectors, space of 2^n vectors
- A subset of 2^n vectors are code words
- Subset called (n, k) code, where fraction k/n is called rate of code
- Addition operation on vectors is bit-wise exclusive-OR

  $X + Y = < x_1 \oplus y_1, x_2 \oplus y_2, \ \ldots, x_n \oplus y_n >$
- Multiplication operation is bitwise AND

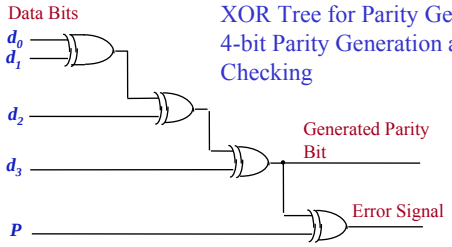
  $cX = < cx_1, cx_2, \ \ldots, cx_n >$

Example

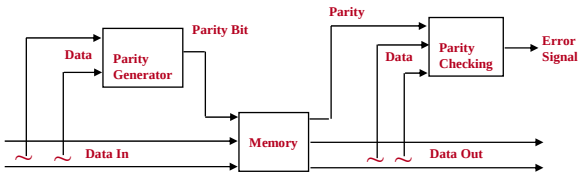
- Parity Codes
- Separable code

Odd and even parity codes for BCD data

Decimal Digit	BCD	BCD odd parity	BCD even parity
0	0000	00001	00000
1	0001	00010	00011
2	0010	00100	00101
3	0011	00111	00110
4	0100	01000	01001
5	0101	01011	01010
6	0110	01101	01100
7	0111	01110	01111
8	1000	10000	10001
9	1001	10011	10010
		Parity Bit	Parity Bit

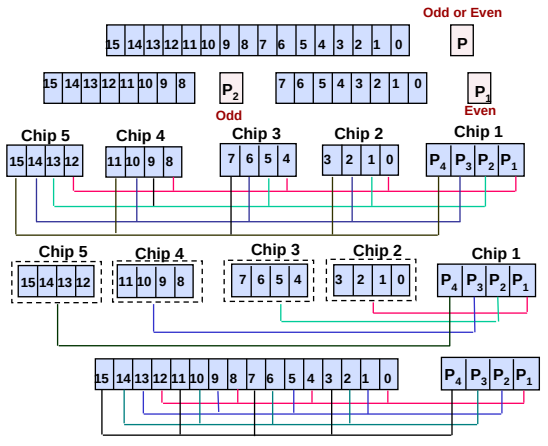


XOR Tree for Parity Generation
4-bit Parity Generation and Checking



Organization of a memory that uses single-bit parity. The parity bit is generated when data is written to memory and checked when data is ready.

Variation of Parity Codes for RAM



Bit-per-word parity

Bit-per-byte parity

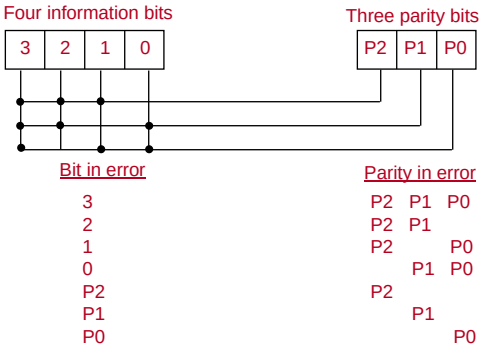
Bit-per-multiple-chips parity

Bit-per-chip parity

Interlaced parity

Overlapping Parity

- Parity groups are formed with each bit appearing in more than one parity group
- Errors can be detected and located
- Erroneous bit can be corrected by a simple complementation



Parity Codes for Memory - Comparison

Parity Code	Advantages	Disadvantages
Bit-per-word: one parity bit per data word	Detects all single-bit errors	Certain errors undetected, e.g., a word, including parity bit becomes all 1s, due to a failure of a bus or a set of data buffers.
Bit-per-byte: each data portion (e.g., a byte) is protected by a separate parity bit; the parity of one group should be even and the parity of the other group should be odd	Detects all-1s and all-0s conditions	Ineffective for multiple errors, e.g., the whole-chip failure
Bit-per-multiple-chips: one bit from each chip is associated with a single parity bit	Detects failure of entire chip	Cannot locate failure of complete chip
Bit-per-chip: each parity bit is associated with one chip of the memory	Detects single-bit errors and identifies chip with erroneous bit	Susceptible to whole-chip failure, i.e., a single chip error can result in multiple bits to be corrupted and this may go undetected.
Interlaced: similar to the bit-per-multiple-chips; must ensure that no two adjacent bits are from the same parity group	Detects errors in adjacent bits	Parity groups are not based on physical organization of the memory
Overlapping:	Error can be detected and corrected	Multiple bits of parity is needed

Parity Prediction in Arithmetic Circuits

- Binary adder
  - Two inputs: $A = (a_{n-1} \ldots a_0\ a_c)$ and $B = (b_{n-1} \ldots b_0\ b_c)$
  - Two operands to be added: $(a_{n-1} \ldots a_0)$ and $(b_{n-1} \ldots b_0)$
  - a_c and b_c are check bits of A and B, respectively
  - Encoded output will be $s = (s_{n-1} \ldots s_0\ s_c)$ where $(s_{n-1} \ldots s_0)$ are determined by the ordinary binary addition of $(a_{n-1} \ldots a_0)$ to $(b_{n-1} \ldots b_0)$ and s_c is the check bit for $(s_{n-1} \ldots s_0)$
  - Then

$$s_c = \sum_{i=0}^{n-1} s_i$$

$$= \sum_{i=0}^{n-1} a_i \oplus \sum_{i=0}^{n-1} b_i \oplus \sum_{i=0}^{n-1} c_i$$

  - Reduces to

$$s_c = a_c \oplus b_c \oplus \sum_{i=0}^{n-1} c_i$$

A four bit example

$A = a_3 a_2 a_1 a_0$ $\quad a_c = a_3 \oplus a_2 \oplus a_1 \oplus a_0$

$B = b_3 b_2 b_1 b_0$ $\quad b_c = b_3 \oplus b_2 \oplus b_1 \oplus b_0$

$S = s_3 s_2 s_1 s_0$ $\quad s_c = s_3 \oplus s_2 \oplus s_1 \oplus s_0$

$s_0 = a_0 \oplus b_0 \oplus c_0$

$s_1 = a_1 \oplus b_1 \oplus c_1$

$s_2 = a_2 \oplus b_2 \oplus c_2$

$s_3 = a_3 \oplus b_3 \oplus c_3$

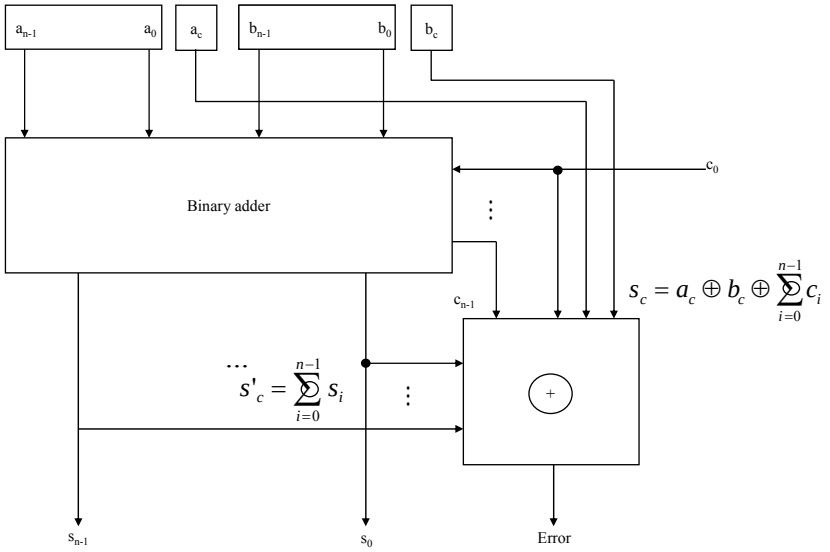
$$s_c = a_c \oplus b_c \oplus \sum_{i=0}^{n-1} c_i$$

$s_c = (a_0 \oplus b_0 \oplus c_0) \oplus (a_1 \oplus b_1 \oplus c_1) \oplus (a_2 \oplus b_2 \oplus c_2) \oplus (a_3 \oplus b_3 \oplus c_3)$

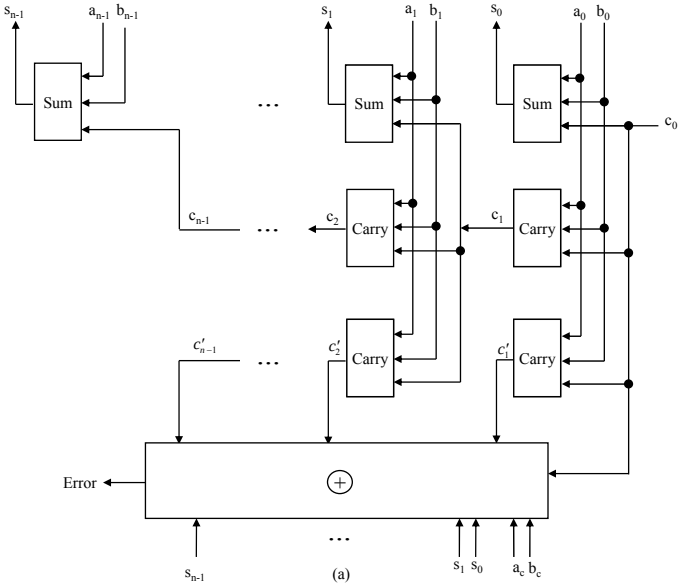
$s_c = (a_3 \oplus a_2 \oplus a_1 \oplus a_0) \oplus (b_3 \oplus b_2 \oplus b_1 \oplus b_0) \oplus (c_3 \oplus c_2 \oplus c_1 \oplus c_0)$

$s_c = a_c \oplus b_c \oplus (c_3 \oplus c_2 \oplus c_1 \oplus c_0)$

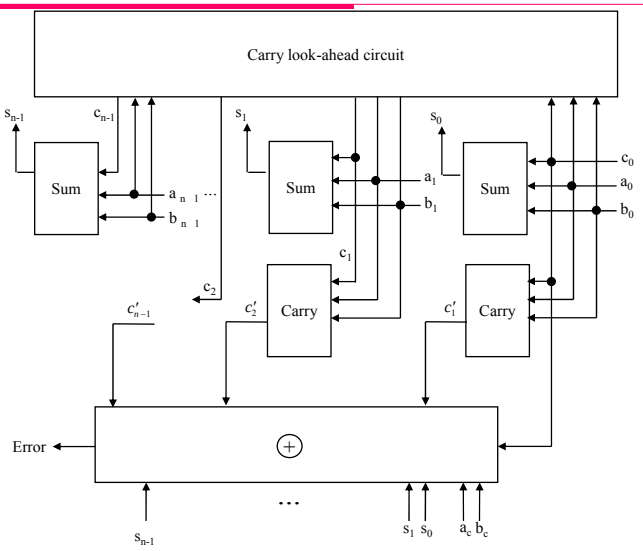
Parity Prediction in Binary Adder



Parity-Checked Binary Adder



Parity-Checked Binary Adder (cont.)



Binary Multiplier

$$\begin{array}{r} a_3a_2\,a_1a_0 \\ \times\ b_3b_2\,b_1b_0 \\ \hline a_3b_0\,a_2b_0\ a_1b_0\ a_0b_0 \\ a_3b_1\,a_2b_1\ a_1b_1\ a_0b_1 \\ a_3b_2\,a_2b_2\ a_1b_2\ a_0b_2 \\ a_3b_3\,a_2b_3\ a_1b_3\ a_0b_3 \\ \hline p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0 \end{array}$$

$$p_0 = a_0b_0$$
$$p_1 = a_0b_1 \oplus a_1b_0$$
$$p_2 = a_0b_2 \oplus a_1b_1 \oplus a_2b_0 \oplus c_{1,1}$$
$$p_3 = a_0b_3 \oplus a_1b_2 \oplus a_2b_1 \oplus a_3b_0 \oplus c_{2,1} \oplus c_{1,2}$$
$$p_4 = a_1b_3 \oplus a_2b_2 \oplus a_3b_1 \oplus c_{3,1} \oplus c_{2,2} \oplus c_{1,3}$$
$$p_5 = a_2b_3 \oplus a_3b_2 \oplus c_{3,2} \oplus c_{2,3} \oplus c_{1,4}$$
$$p_6 = a_3b_3 \oplus c_{3,3} \oplus c_{2,4}$$
$$p_7 = c_{3,4}$$

Example:

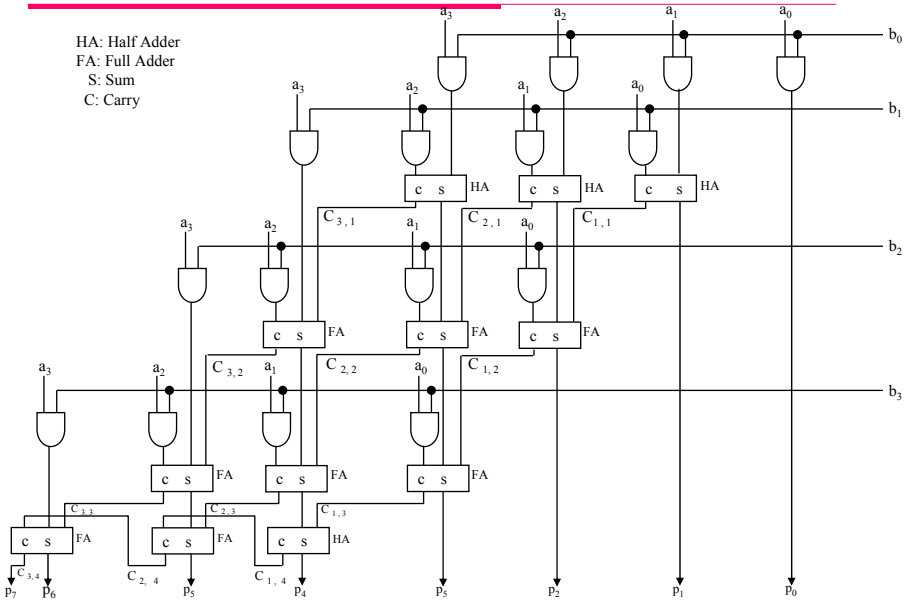
$$\begin{array}{r} 1\ 0\ 1\ 1 \\ \times\ 0\ 1\ 1\ 0 \\ \hline 0\ 0\ 0\ 0 \\ 1\ 0\ 1\ 1 \\ 1\ 0\ 1\ 1 \\ 0\ 0\ 0\ 0 \\ \hline p_7p_6\,p_5p_4p_3p_2p_1p_0 \end{array}$$

Cases for A's and B's		
A Odd $\sum a_i = 1$	A Even 1 $\sum a_i = 0$	A Even 1 $\sum a_i = 0$
B Odd $\sum b_j = 1$	B Odd 1 $\sum b_j = 0$	B Even 1 $\sum b_j = 0$
Odd $a_ib_j = 1$ $\sum p_j = 1$	Even # $a_ib_j = 1$ $\sum p_j = 0$	Even # $a_ib_j = 1$ $\sum p_j = 0$

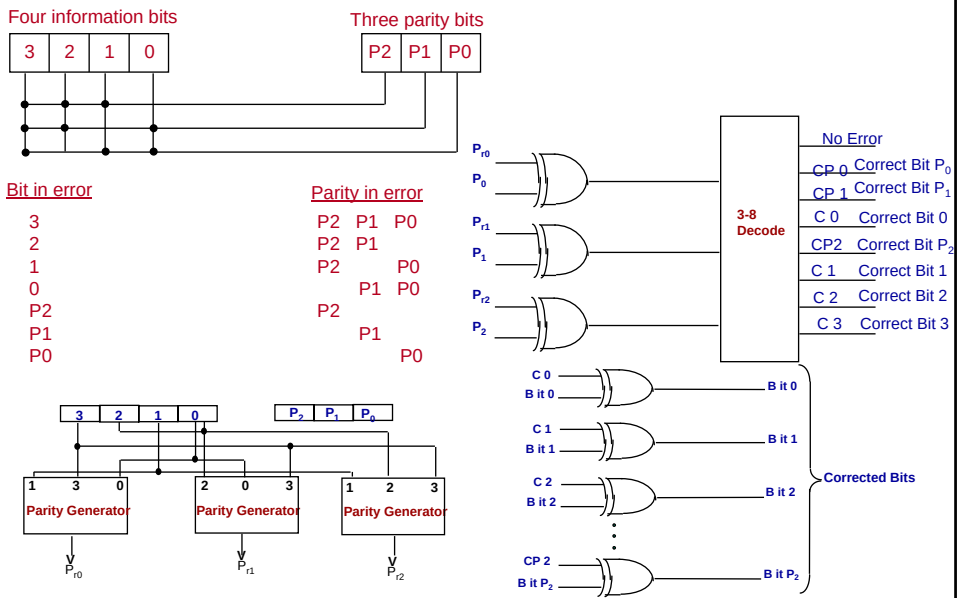
Therefore, denoting the check bit for $(p_7 \ldots p_0)$ by p_c

$$p_c = \sum_{i=0}^{7} p_i$$
$$= \left(\sum_{i=0}^{3} a_i\right)\left(\sum_{i=0}^{3} b_j\right) \oplus \sum_{i=1}^{3}\sum_{j=1}^{4} c_{i,j}$$
$$= a_cb_c \oplus \sum_{i=1}^{3}\sum_{j=1}^{4} c_{i,j}$$

Multiplier Using Array of Full-Half Adders



Error Correction with Overlapped Parity



Hamming Error-Correcting Code

- Require from 10% to 40% redundancy
- Overlapping parity
- The Hamming single-error correcting code uses ***c*** parity check bits to protect ***n*** bits of information:

$$2^c \geq c + n + 1$$

- Example:
  - For four information bits (d_3, d_2, d_1, d_0), need three parity bits (p_2, p_1, p_0)
  - the bits are partitioned into groups as (d_3, d_1, d_0, p_0), (d_3, d_2, d_0, p_1) and (d_3, d_2, d_1, p_2)
  - the grouping of bits can be determined from a list of binary numbers from 0 to 2^n - 1.
  - Each check bit is specified to set the parity, either even or odd, of its respective group

7	6	5	4	3	2	1
d_3	d_2	d_1	p_2	d_0	p_1	p_0

Hamming Error-Correcting Code

7	6	5	4	3	2	1
d_3	d_2	d_1	p_2	d_0	p_1	p_0

$p_2p_1p_0$	Bit in Error
0 0 0	No Error
0 0 1	P_0
0 1 0	P_1
0 1 1	d_0
1 0 0	p_3
1 0 1	d_1
1 1 0	d_2
1 1 1	d_3

Observe that the check bits identifies the position of the bit in error

Parity bits calculation

p2 = XOR of bits (5, 6, 7) = $d_1 \oplus d_2 \oplus d_3$
p1 = XOR of bits (3, 6, 7) = $d_0 \oplus d_2 \oplus d_3$
p0 = XOR of bits (3, 5, 7) = $d_0 \oplus d_1 \oplus d_3$

Parity checking

c2 = XOR of bits (4,5, 6, 7) = $p_2 \oplus d_1 \oplus d_2 \oplus d_3$
c1 = XOR of bits (2,3, 6, 7) = $p_1 \oplus d_0 \oplus d_2 \oplus d_3$
c0 = XOR of bits (1,3, 5, 7) = $p_0 \oplus d_0 \oplus d_1 \oplus d_3$

Example: $d_3d_2d_1d_0$= 0101 ➔ $p_2p_1p_0$= 101
Hence, code = $d_3d_2d_1\mathbf{p_2}d_0\mathbf{p_1p_0}$ = 0101101
Assume a bit error in bit d_0 ➔ 0101001

c3 = $p_2 \oplus d_1 \oplus d_2 \oplus d_3$ = 0
c2 = $p_1 \oplus d_0 \oplus d_2 \oplus d_3$ = 1
c1= $p_0 \oplus d_0 \oplus d_1 \oplus d_3$ = 1

Hence, bit 3 is in error! And needs to be inverted.

Check Bits and Syndromes for Single-Bit Errors

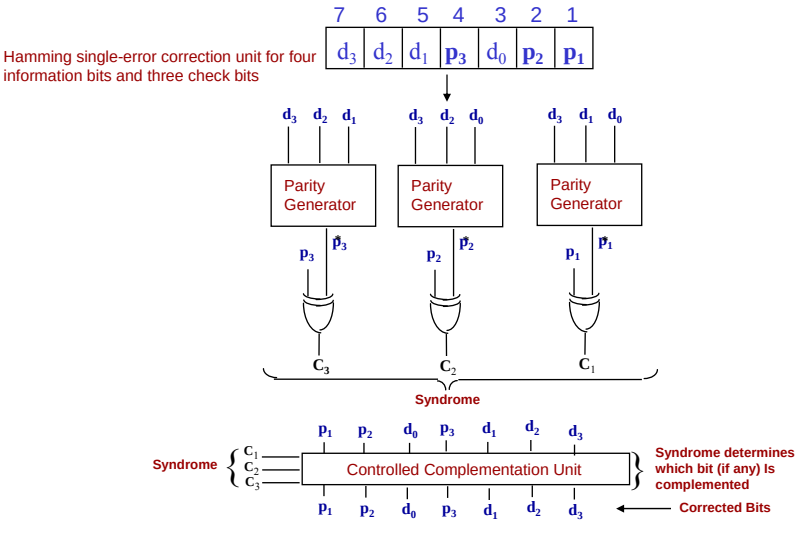
- The original data is encoded by generating a set C_g, of parity bits.
- To check correctness, the encoding process is repeated and a set C_c, of parity bits is generated.
- If C_g and C_c agree, the information is correct.
- If C_g and C_c disagree, the information is incorrect and must be corrected.
- To aid the correction, a *syndrome* is defined:

  The syndrome is a binary word that has 1 in each bit position in which C_g and C_c disagree; the syndrome points directly to the erroneous bit.

7	6	5	4	3	2	1
d_3	d_2	d_1	$\mathbf{p_2}$	d_0	$\mathbf{p_1}$	$\mathbf{p_0}$

Erroneous bits	Check bits affected	Syndromes
d_0	p_0, p_1	011
d_1	p_0, p_2	101
d_2	p_1, p_2	110
d_3	p_0, p_1, p_2	111
p_1	p_0	001
p_2	p_1	010
p_3	p_2	100

Hamming Single-Error Correction Unit



Double-Bit Errors

- Say d_1 and d_0 become faulty,
  - $p_2\ p_1\ p_0 = 111$
  - d_3 is incorrectly inverted.

7	6	5	4	3	2	1
d_3	d_2	d_1	$\mathbf{p_2}$	d_0	$\mathbf{p_1}$	$\mathbf{p_0}$

Erroneous bits	Check bits affected	Syndromes
d_0	p_0, p_1	011
d_1	p_0, p_2	101
d_2	p_1, p_2	110
d_3	p_0, p_1, p_2	111
p_0	p_0	001
p_1	p_1	010
p_3	p_2	100

- Double errors can't be corrected, can it be detected?
- Adding an extra parity bit to check data and overlapping parity bits
- $\underbrace{d_{n-1} \cdots d_1, d_0}$ $\underbrace{p_{k-1} \cdots p_0}$ $\boxed{p}$

Single bit error	Detects error	Detects error
Double bits error	Detects error	Does not detect error

Modified Hamming Code (SEC-DED)

H matrix for Single Error Correction and Double Error Detection

	8	7	6	5	4	3	2	1
	$\mathbf{p_3}$	$\mathbf{d_3}$	$\mathbf{d_2}$	$\mathbf{d_1}$	$\mathbf{p_2}$	$\mathbf{d_0}$	$\mathbf{p_1}$	$\mathbf{p_0}$
$\mathbf{c_2}$	0	1	1	1	1	0	0	0
$\mathbf{c_1}$	0	1	1	0	0	1	1	0
$\mathbf{c_0}$	0	1	0	1	0	1	0	1
$\mathbf{c_3}$	1	1	1	1	1	1	1	1

Check bits computation

P_2 = XOR (5, 6, 7)
P_1 = XOR (3, 6, 7)
P_0 = XOR (3, 5, 7)

P_3 = parity over the first 7 bits of the code word
i.e. $d_3\ d_2\ d_1\ \mathbf{p_2}\ d_0\ \mathbf{p_1}\ \mathbf{p0}$

c_2	c_1	c_0	c_3	
0	0	0	0	**No errors**
x_2	x_1	x_0	1	**Single error in a position ($x_2x_1x_0$)**
y_2	y_1	y_0	0	**Double error**
0	0	0	1	**Error in bit p4**

Syndromes computation

C_2 = XOR (4, 5, 6, 7)
C_1 = XOR (2, 3, 6, 7)
C_0 = XOR (1, 3, 5, 7)

C_3 = parity over all 8 bits of the code word

Single Error Correction and Double Error Detection Hamming Code (SEC-DED) Example

Failure scenarios

	8	7	6	5	4	3	2	1	
	p_4	d_3	d_2	d_1	p_2	d_0	p_1	p_0	
1	0	0	1	1	0	0	1	1	No errors
2	0	0	1	1	0	1	1	1	Single error in position 3
3	0	0	0	1	0	0	1	1	Single error in position 6
4	0	0	1	0	0	0	1	0	Double error
5	1	0	1	1	0	0	1	1	Error in bit p3

Check bits computation

P_2 = XOR (5, 6, 7)
P_1 = XOR (3, 6, 7)
P_0 = XOR (3, 5, 7)

Initial data

d_3 d_2 d_1 d_0
0 1 1 0

	c_3	c_2	c_1	c_0	Corresponding Syndromes
1	0	0	0	0	No errors
2	1	0	1	1	Single error in position 3
3	1	1	1	0	Single error in position 6
4	0	1	0	0	Double error
5	1	0	0	0	Error in bit p3

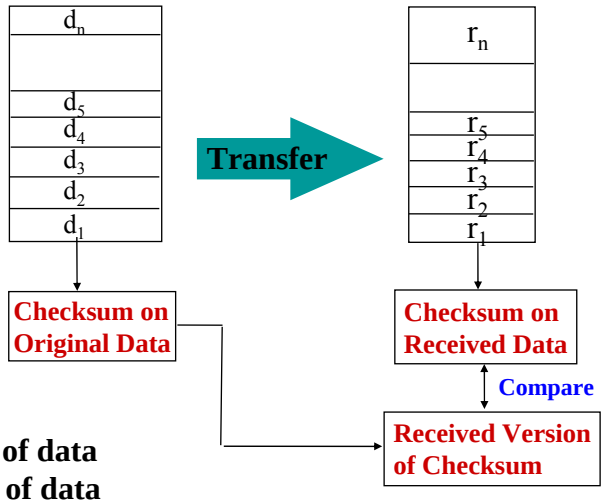
Syndromes computation

C_2 = XOR (4, 5, 6, 7)
C_1 = XOR (2, 3, 6, 7)
C_0 = XOR (1, 3, 5, 7)

C_3 = parity over all 8 bits of the code word

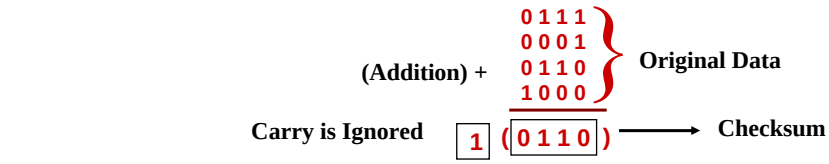
Checksum Codes - Basic Concepts

- The checksum is appended to block data when such blocks are transferred

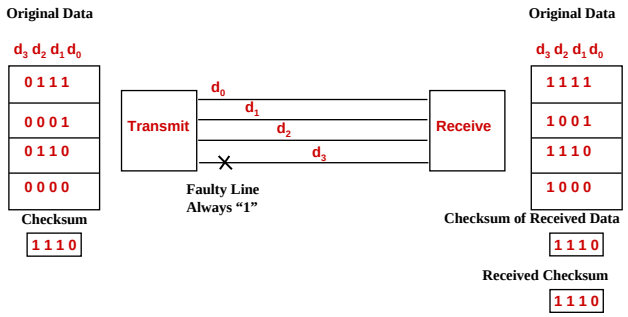


d_i = original word of data
r_i = received word of data

Single Precision Checksums



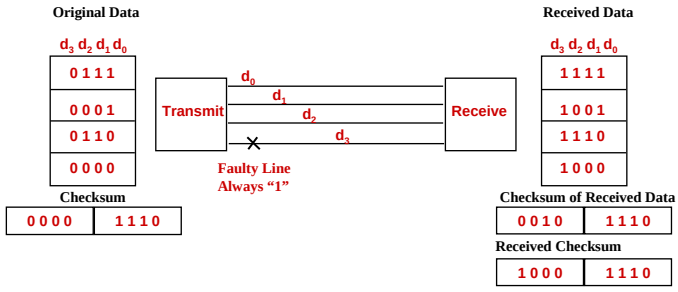
A single-precision checksum is formed by adding the data words and ignoring any overflow



The single-precision checksum is unable to detect certain types of errors. The received checksum and the checksum of the received data are equal, so no error is detected.

Double Precision Checksums

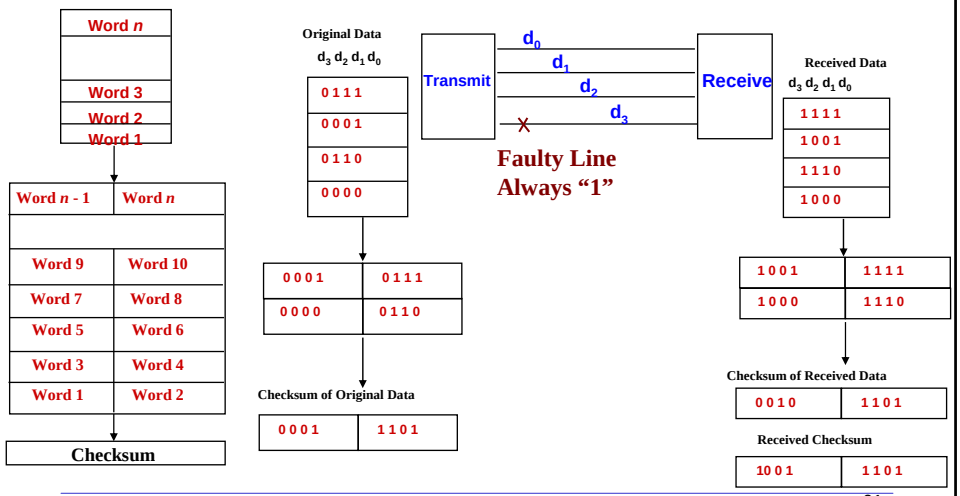
- Compute 2n-bit checksum for a block of n-bit words
- Overflow is still a concern, but it is now overflow from a 2n-bits



The received checksum and the checksum of the received data are not equal, so the error is detected

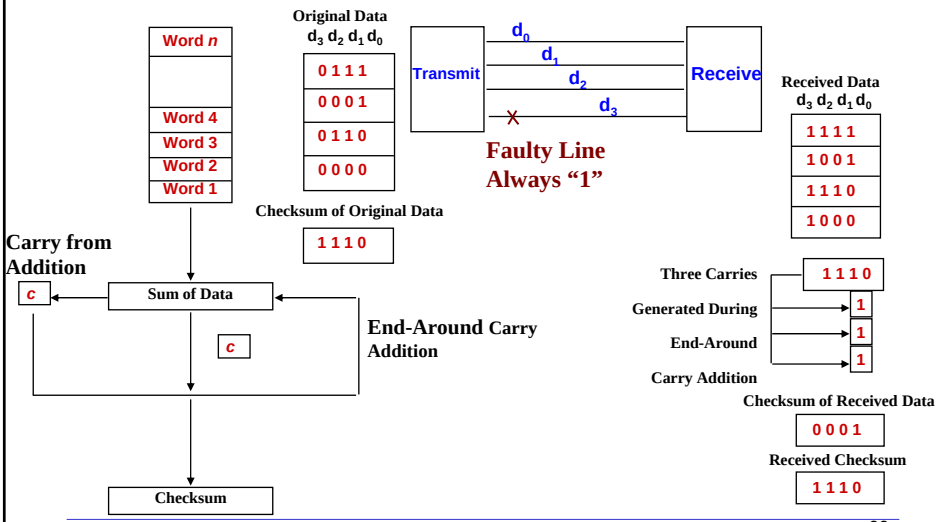
Honeywell Checksums

- Concatenate consecutive words to form double words to create k/2 words of 2n bits; checksum formed over newly structured data



Residue Checksums

The same concept as the single-precision checksum except that the carry bit is not ignored and is added to checksum in an end-around carry fashion



Codes for Storage and Communication
Cyclic Codes

- Cyclic codes are parity check codes with additional property that cyclic shift of codeword is also a codeword
  - if (C_{n-1}, C_{n-1} ... C_1, C_0) is a codeword, (C_{n-2}, C_{n-3}, ... C_0, C_{n-1}) is also a codeword
- Cyclic codes are used in
  - sequential storage devices, e.g. tapes, disks, and data links
  - communication applications
- An (n,k) cyclic code can detect single bit errors, multiple adjacent bit errors affecting fewer than (n-k) bits, and burst transient errors
- Cyclic codes require less hardware, in form of linear feedback shift registers
  - in comparison, parity check codes require complex encoding, decoding circuit using arrays of EX-OR gates, AND gates, etc.

Cyclic Code and Polynomials

- Cyclic codes employ on the representation of data by a polynomial
- If (C_{n-1}, C_{n-2} ... C_1, C_0) is a codeword, its polynomial representation is $C(x)= C_{n-1}\, x^{n-1} + C_{n-2}x^{n-2} + \ldots C_1x + C_0$
- The (n,k) cyclic code is characterized by
  - G(x): the generator polynomial of degree (n-k)
  - D(x): the data polynomial of degree (k-1)
    - d_{k-1}, d_{k-2} ... d_1, d_0 ➔ $D(x)= d_{k-1}\, x^{k-1} + C_{k-2}x^{k-2} + \ldots C_1x + C_0$
    - Similar idea to number representation in a base
      - $(d_3d_2d_1d_0)_r = d_3 \times r^3 + d_2 \times r^2 + d_1 \times r + d_0$
  - V(x): Code polynomial Code word V
    - Code word V = (v_{n-1}, v_{n-2} ... v_1, v_0) of degree (n-1)

Cyclic Code and Polynomials

- Encoding process: $V(x) = D(x) * G(x)$ using modulo 2
  - Code word $V(x)$ is a non-separable code
- Example: for (7,4) code, $G(x) = x^3 + x + 1$
  - Given data word (1011) ➔ $D(x) = x^3 + x + 1$
  - $V(x) = G(x) * D(x) = (x^3 + x + 1)(x^3 + x + 1) =$
    $x^6 + x^4 + x^3 + x^4 + x^2 + x + x^3 + x + 1 = x^6 + x^2 + 1$
  - Hence code word is (1000101)
  - Given data word (1111) ➔ $D(x) = x^3 + x^2 + x + 1$
  - $V(x) = G(x) * D(x) = (x^3 + x^2 + x + 1)(x^3 + x + 1) =$
    $= x^6 + x^5 + x^3 + 1$
  - Hence code word is (1101001)

Basic Operations on Polynomials

- Can multiply or divide one polynomial by another, follow modulo 2 arithmetic, coefficients are 1 or 0, and addition and subtraction are same

Multiplication

$$(x^4 + x^3 + x^2 + 1)(x^3 + x)$$

$$x^7 + x^6 + x^5 + x^3$$
$$+ x^5 + x^4 + x^3 + x$$
$$= x^7 + x^6 + x^4 + x$$

Division

$$x^4 + x^3 + x^2 + 1 \;\overline{)}\; x^5 + x^4 + x^3 + x$$

x — Quotient

$x^5 + x^4$

$x^5 + x^4 + x^3 + x$

$x^3 + x$ — Remainder

- G(x) is a polynomial of degree (n-k) for an (n,k) code, with a unity coefficient in (n-k) term
- G(x) is a factor of $x^n - 1$, i.e., it divides it with zero remainder
  - if a polynomial with degree n-k divides $x^n - 1$, then G(x) generates a cyclic code
  - for (7,4) code, $G(x) = x^3 + x + 1$, can verify g(x) divides $x^7 - 1$

Cyclic Code - Example

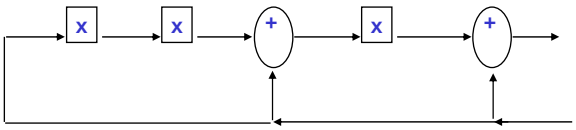
- **Generator polynomial**
  - $G(x) = x^3 + x + 1$ for (7,4) code
- **Data polynomial**
  - $D(x) = d_3x^3 + d_2x^2 + d_1x + d_0$
- **Code polynomial**
  - $V(x) = \nu_6x^6 + \nu_5x^5 + \nu_4x^4 + \nu_3x^3 + \nu_2x^2 + \nu_1x + \nu_0$
- Code distance is 3
  - SEC/DED code

Cyclic codes for 4-bit information words.

Information (d_3, d_2, d_1, d_0)	Code $(\nu_6,\nu_5,\nu_4,\nu_3,\nu_2,\nu_1,\nu_0)$
0000	0000000
0001	0001101
0010	0011010
0011	0010111
0100	0110100
0101	0111001
0110	0101110
0111	0100011
1000	1101000
1001	1100101
1010	1110010
1011	1111111
1100	1011100
1101	1010001
1110	1000110
1111	1001011

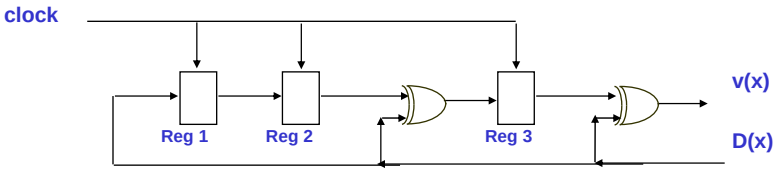
Circuit to Generate Cyclic Code

- Consider blocks labeled X as multipliers, and addition elements as modulo 2



$g(x)=x^3 + x + 1$
$V(x) = [(x^2 +1)x +1] D(x)$

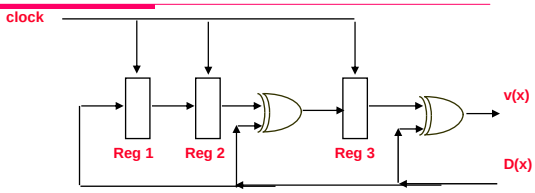
- Another representation is to replace multipliers by storage elements, adders by EX-OR gates



Generation of Code Words

Cyclic codes for 4-bit information words.

Information (d_0, d_1, d_2, d_3)	Code $(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
0000	0000000
0001	0001101
0010	0011010
0011	0010111
0100	0110100
0101	0111001
0110	0101110
0111	0100011
1000	1101000
1001	1100101
1010	1110010
1011	1111111
1100	1011100
1101	1010001
1110	1000110
1111	1001011



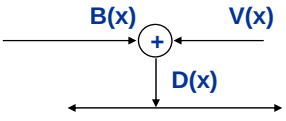
The encoding process

Clock period	Register values 1	2	3	D(x)	V(x)
0	0	0	0	1	1
1	1	0	1	1	0
2	1	1	1	0	1
3	0	1	1	1	0
4	1	0	0	0	0
5	0	1	0	0	0
6	0	0	1	0	1
7	0	0	0		

Data polynomial = $d_0 + d_1x + d_2x^2 + d_3x^3$
Generator polynomial = $1 + x + x^3$
Code polynomial = $v_0 + v_1x + v_2x^2 + v_3x^3 + v_4x^4 + v_5x^5 + v_6x^6$

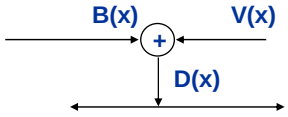
Decoding of Cyclic Codes

- Determine if code word $(r_{n-1}, r_{n-2}, ..., r_1, r_0)$ is valid
- Code polynomial $r(x) = r_{n-1}\, x^{n-1} + r_{n-2}\, x^{n-2} + ... \, r_1\, x + r_0$
- If r(x) is a valid code polynomial, it should be a multiple generator polynomial g(x)
- r(x) = d(x) g(x) + s(x), where **s(x) the syndrome polynomial** should be zero
- Hence, divide r(x) by g(x) and check the remainder whether equal to 0
- D(x) = V(x) / G(x)

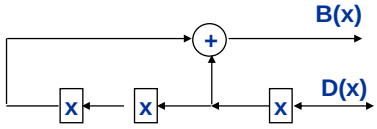


Decoding of Cyclic Codes

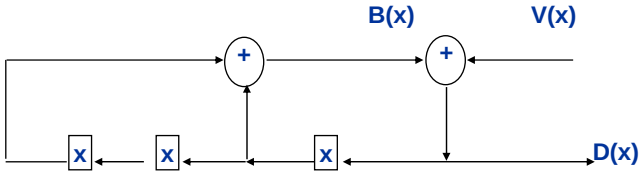
- Example: for (7,4) code, $G(x) = x^3 + x + 1$
- $D(x) = V(x) + B(x)$
- $V(x) = D(x) - B(x)$



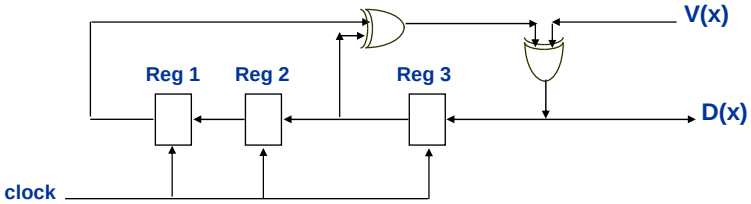
- Since in Modulo 2 addition and subtraction are the same
- $V(x) = D(x) + B(x)$
- $V(x) = D(x)\,(x^3 + x + 1)$
- $B(x) = D(x)\,(x^3 + x)$



Circuits for Decoding



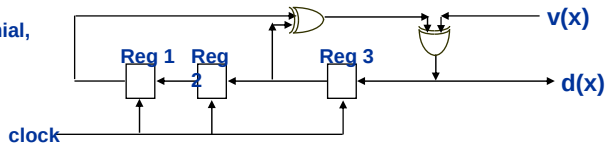
Another representation is to replace multipliers by storage elements and adders by EX-OR gates



Note: **Once the division is completed, the registers contain the value of the syndrome (remainder)**

Example Decoding

Generator polynomial,
$g(x) = x^3 + x + 1$



The decoding process with correct information

Clock period	Register values 1	2	3	V(x)	B(x)	D(x)
0	0	0	0	1	0	1
1	0	0	1	0	1	1
2	0	1	1	1	1	0
3	1	1	0	0	1	1
4	1	0	1	0	0	0
5	0	1	0	0	0	0
6	1	0	0	1	1	0
7	0	0	0			

Syndrome; Code word; Original Information

The decoding process with erroneous information

Clock period	Register values 1	2	3	V(x)	B(x)	D(x)
0	0	0	0	1	0	1
1	0	0	1	0	1	1
2	0	1	1	1	1	0
3	1	1	0	1	1	0
4	1	0	0	0	1	1
5	0	0	1	0	1	1
6	0	1	1	1	1	0
7	1	1	0			

Nonzero Syndrome; Received word

Arithmetic Codes

- Useful to check arithmetic operations
- Parity codes are not preserved under addition, subtraction
- Arithmetic codes can be separate (check bits disjoint from data bits) or nonseparate (combined check and data)
- Several Arithmetic codes
  - *AN codes, Residue codes, Inverse residue codes, Residue Number Systems (RNS)*
- Arithmetic codes must be invariant to a set of arithmetic operations
  - A(b*c) = A(b) * A(c)

AN Codes

- Multiple each data word N with some constant A
- Code is invariant to addition and subtraction, but not to multiplication and division
- Constant A determines the number of extra bits required and error detection capability provided
- $A \neq 2^a$
  - If $A = 2^a$
  - $N = n_{j-1}\, n_{j-2} \ldots n_2\, n_1\, n_0$
  - $A\,N = n_{j-1}\, n_{j-2} \ldots n_2\, n_1\, n_0 \overbrace{00 \ldots 0}^{a}$
  - AN still divisible to A if and bits n_i is in error

Properties of AN Codes

- A valid AN code is 3N code

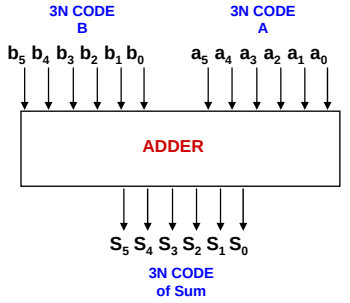
N	3 N
0000	000000
0001	000011
0010	000110
0011	001001

- Operation performed under an AN code can be checked by determining if the result is evenly divisible by A
- 6 + 1 = 7 ➔ 010010 + 000011 = 010101 (21 is divisible by 3)
- If you have an output line s-a-1 i.e. 010111
  - 23 not divisible by 3 and error is detected

Example of 3N Code

Resulting 3N code words for a 4-bit information words

Original Information	3N code word
0000	000000
0001	000011
0010	000110
0011	001001
0100	001100
0101	001111
0110	010010
0111	010101
1000	011000
1001	011011
1010	011110
1011	100001
1100	100100
1101	100111
1110	101010
1111	101101



Normal Operation

A = 0 1 0 0 1 0 (3N Code of 6)
+ B = 0 0 0 0 1 1 (3N Code of 1)
S = 0 1 0 1 0 1 (3N Code of 7)

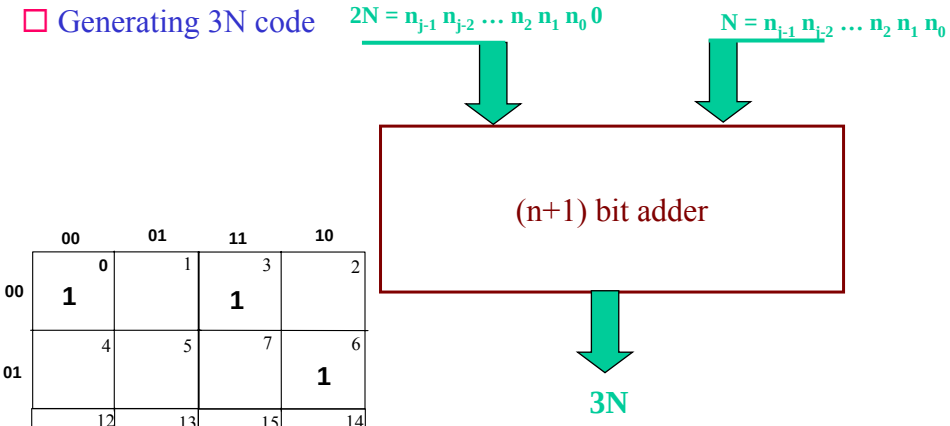
If S_1 is always "1"

A = 0 1 0 0 1 0 (3N Code of 6)
B = 0 0 0 0 1 1 (3N Code of 1)
S = 0 1 0 1 1 1 (Not a valid 3N Code)

Illustration of the error detection capabilities of the 3N arithmetic code. The presence of the fault results in the sum being an invalid 3N code.

Generating AN Codes

☐ Generating 3N code



	00	01	11	10
00	0 **1**	1	3 **1**	2
01	4	5	7	6 **1**
11	12 **1**	13	15 **1**	14
10	8	9 **1**	11	10

☐How to check if a code is a 3N

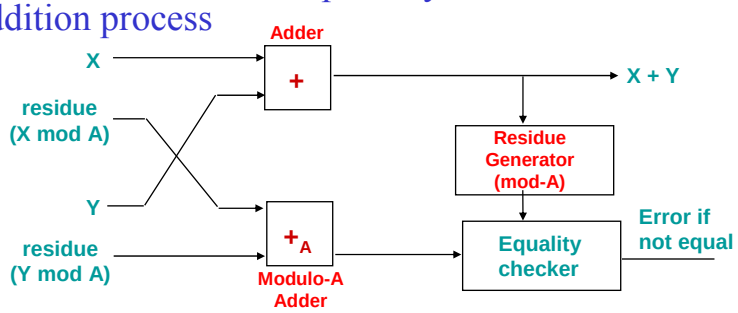
Residue codes

- Separable codes D | R
  - D is data and R is the residue of the data
  - *N = Q m + r* ; r remainder or residue, Q quotient, and m modulus
    - *N = 14 ➔ m = 3, r =2 ➔ 14 = 2 mod (3)*
  - Example: residue of a 4 bit mod 3

Data	residue	Codeword
0000	0	0000 00
0001	1	0001 01
0010	2	0010 10
0011	0	0011 00
0100	1	0100 01

Residue Codes

- Residue of an integer *n* with respect to *A* is: *n mod (A)*
- Residue code are invariant to additions
- *((x mod A) + (y mod A)) mod A = (x+y) mod A*
- Residues can be handled separately from the data during the addition process



- If A is any integer not a power of 2, then Residue Code detects any single-bit error in an arithmetic operation of ADD or SUB

Low Cost Residue Code

- $m = 2^b - 1$ ($b \geq 2$) then code word has n+b bits
- Encoding in low cost residue is done by dividing the information into b bits and add them in mod $(2^b - 1)$.
- Example: b = 2 ➔ $m = 2^2 - 1 = 3$
  - Data = $(167)_{10} = (10100111)_2$

$$\underbrace{\underbrace{10}_{01}\ \underbrace{10}_{01}}_{10}\quad\underbrace{\underbrace{01}_{01}\ \underbrace{11}_{01}}_{10}$$

167 mod 3 = 2

- Low cost residue code provides easy encoding
- Inverse residue code
  - Instead of appending r, append m-r
  - Better detection of repeated-used fault

Residue Number System (RNS)

- Represent a number by a set of relatively prime moduli
- P = [3, 4, 5]
  - $32_{10} = (2\ 0\ 2)_{RNS}$
  - $14_{10} = (2\ 2\ 4)_{RNS}$

$$\begin{array}{rr} 32_{10} & (2\ 0\ 2)_{RNS} \\ +\ 14_{10} & +\ (2\ 2\ 4)_{RNS} \\ \hline 46_{10} & (1\ 2\ 1)_{RNS} \end{array}$$

- Speed advantage – carry-free number system
- Error detection capability

Residue Number System (RNS)

- Redundant moduli provide higher code distance and therefore provides error detection

- Example: P = [3,4]

Number	RNS		RNS (Binay)	
0	0	0	00	00
1	1	1	01	01
2	2	2	10	10
3	0	3	00	11
4	1	0	01	00
5	2	1	10	01
6	0	2	00	10
7	1	3	01	11

- Code distance is one

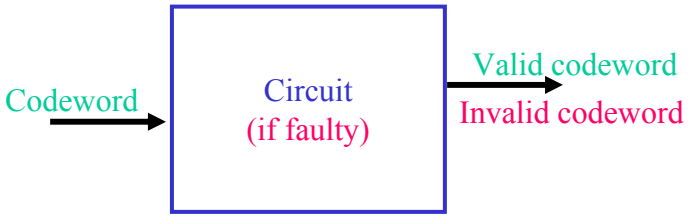
- Example: P = [3, 4, 5]

Number	RNS			RNS (Binay)		
0	0	0	0	00	00	000
1	1	1	1	01	01	001
2	2	2	2	10	10	010
3	0	3	3	00	11	011
4	1	0	4	01	00	100
5	2	1	0	10	01	000
6	0	2	1	00	10	001
7	1	3	2	01	11	010

- Code distance is two
- Tolerates one error

Self-Checking Concept

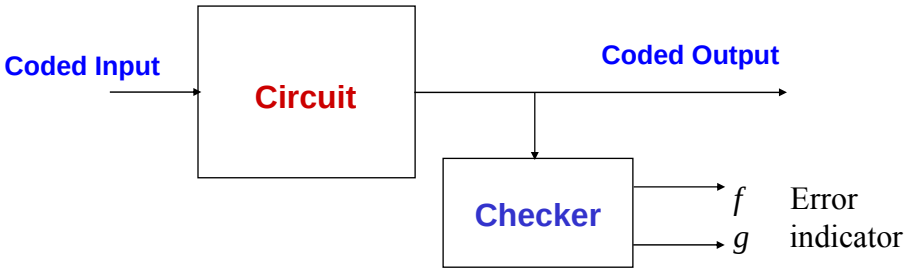
- **Have systems with capabilities of self-checking**
  - Duplication and coding scheme, need to compare outputs of two modules
  - What if the checker fails – single point of failure sceneraio
- Self Checking is the ability to automatically detect the existence of a fault without the need for any externally applied stimulus



Self-Checking Circuits

- **Self-testing circuit** for every fault from a prescribed set there exists at least one valid input code word that will produce an invalid output code word when a single fault is present in the circuit
- **Fault secure circuit** any single fault from a prescribed set results in the circuit either producing the correct code word or producing a non-code word, for any valid input code word
- **Totally self-checking circuit** (TSC)
  - the circuit is both fault secure and self-testing
  - all single faults are detectable by at least one valid code word input, and when a given input combination does not detect the fault, the output is the correct code word output

Self Checking Checker



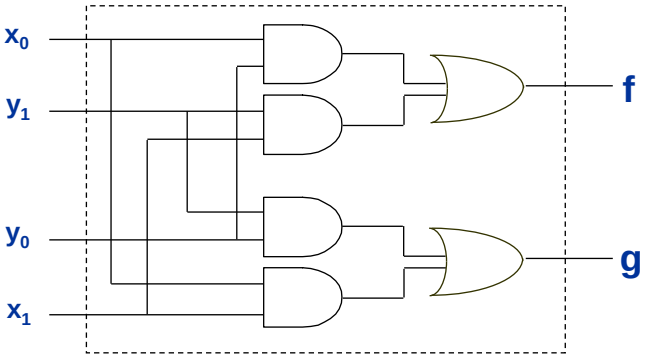
- Two outputs needed to avoid stuck at fault situation
  - If checker is faulty free (valid output)
    - Checker output = 01 or 10
  - If checker is faulty
    - Checker output = 00 or 11

Two Rail Checker

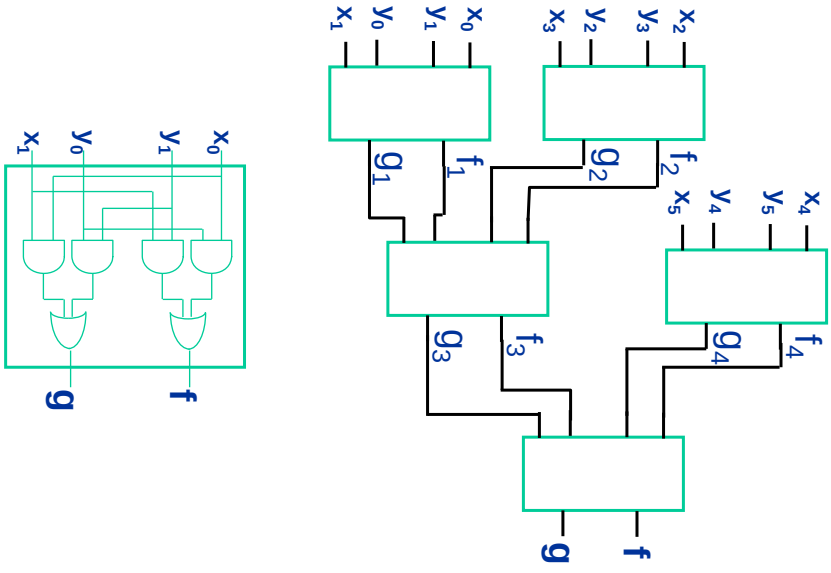
$f = A \oplus B = \bar{A}B + A\bar{B}$
$g = A \oplus B = \bar{A}\bar{B} + AB$

$A = x_0 = y_0$
$B = x_1 = y_1$

$f = x_0y_1 + x_1y_0$
$g = x_0x_1 + y_1y_0$



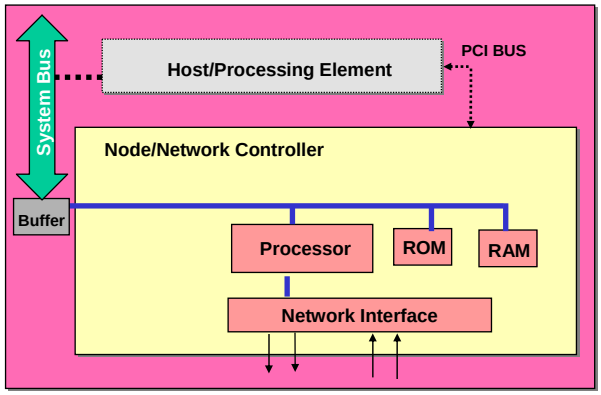
Two Rail Checkers



"Vanilla" Node/Network Controller

Node/network controller

- executes high level protocols (e.g., reliable multicast) and interfaces directly with the network
- host/ processing element can directly access memory of the controller
- the controller can access memory of the host/processing element



Fault Resilient Node/Network Controller

High level protocol engine
(executes high level protocols, e.g., reliable multicast)

- **duplicated**, tightly coupled CPUs
- **data buses compared** on memory accesses and other CPU operations, e.g., write to memory of the the network adapter)
- **memory write-protection** (MASK) on all write operations from the processing element (usually DMA operations)
- **checksum** computation on the data from ROM

Network adapter

- **parity checking** on all data transfers over the bus
- **watchdog** to monitor the node communications with the network
- **control flow checking** and **data audit** to monitor SW errors