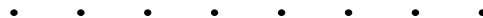




## 2.2 Information Redundancy

- **Idea:** Add redundant information to data to allow
  - Fault detection
  - Fault masking
  - Fault tolerance
- Error detecting codes and error correcting codes



## Information Redundancy

We distinguish:

- Data words → the actual information contents
- Code words → the transmitted information

Codes can be:

- Separable → if the code word contains all original data bits plus additional check bits
- Non-separable → otherwise

Example: ASCII coding of single digit (*separable*)

- Data word: 9
- Code word: 49



## Hamming Distance

- Hamming Distance for a pair of code words:  
The number of bits that are different between the two code words
  - E.g. 0000, 0001 → HD 1
  - E.g. 0100, 0011 → HD 3
- Hamming Distance for a code:  
 $HD(\text{code}) = \text{Min}_{x,y} \{HD(x,y)\}$



## Hamming Distance

- Standard binary code has HD 1
- If a code has HD 2
  - single bit error will result in incorrect code word
  - **detection** of single-bit errors
- If a code has HD 3
  - **detection** of double-bit errors
  - **correction** of single-bit errors
  - since incorrect word will be “closer” to one of the code words (1-bit difference as opposed to 2- or more-bit difference)



⋮

## BCD (Binary Coded Decimals)

- $79_{\text{Dec}}$
- $0111\ 1001_{\text{BCD}}$
- Valid BCD codes:  $0000 \dots 1001$
- Invalid BCD codes:  $1010 \dots 1111$   
→ Some errors can be detected

• HD = ? 1

⋮

## Parity

- Simplest form: odd parity / even parity
- Add 1 bit to data word to achieve odd parity

•  $0100\ 1001\ 0$

•  $0110\ 1001\ 1$

• HD = ? 2

⋮

## Parity

Application:

- Data transmission
- Memory modules with error detection / correction  
set for writing, re-generate&check for reading

⋮

## Parity

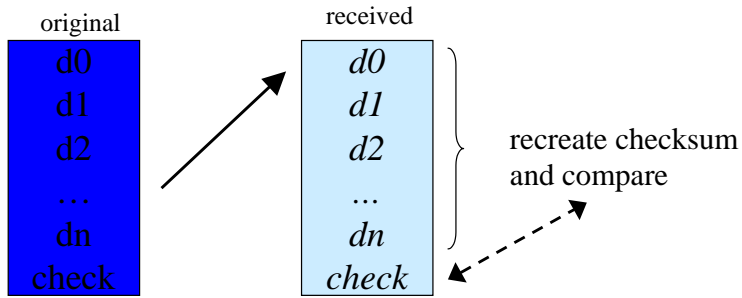
Variations:

- Bit per word  
does not catch memory faults (stuck on 0, stuck on 1)
- Bit per Byte  
with alternating even-odd parity (catches memory faults)
- Bit per multiple Chips  
use 1 bit each per chip for parity, e.g. 8 data chips + 1 parity chip, detects chip failure
- Bit per Chip  
as above, but group several bits from same chip for 1 parity bit
- Interlaced Parity (→ bit per multiple chip)
- Overlapping Parity (→ Hamming codes)

⋮

# Checksums

- A variation of parity, e.g. add modulo 256
- Used for transferring blocks of data



- HD = ? 2

⋮

⋮

# Hamming Code

- d data bits
- c check bits
- $2^c$  different output values of check bits
- number of positions where single-bit error can occur (or no error at all): d+c
- Therefore:  
 $2^c \geq d+c+1$

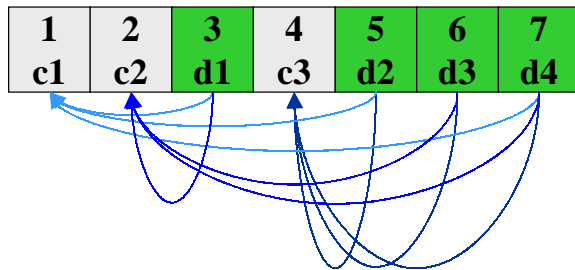
- d = 4 c = ? 3

⋮

⋮

# Hamming Code

- d = 4, c = 3
- Arrange check bits at positions 1, 2, 4, ...

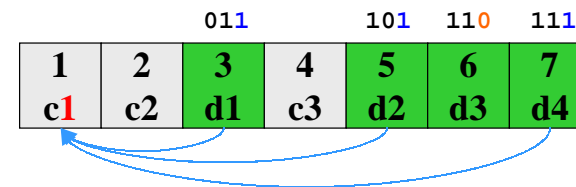


⋮

⋮

# Hamming Code

- Which check bits go with which data bits ?
- Always use the data bits with a "1" in the corresponding position

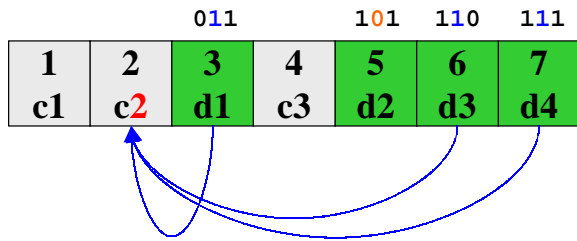


⋮

⋮

# Hamming Code

- Which check bits go with which data bits ?
- Always use the data bits with a “1” in the corresponding position

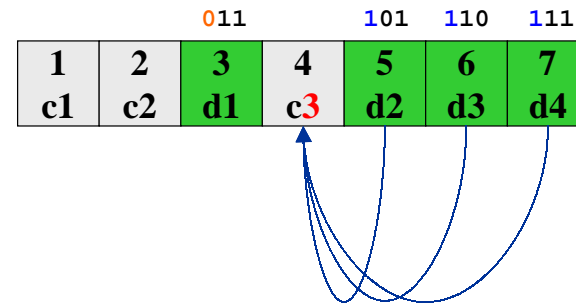


• • • • • • • •

⋮

# Hamming Code

- Which check bits go with which data bits ?
- Always use the data bits with a “1” in the corresponding position



• • • • • • • •

⋮

# Hamming Code

- $c1 = d1 \oplus d2 \oplus d4$
- $c2 = d1 \oplus d3 \oplus d4$
- $c3 = d2 \oplus d3 \oplus d4$

• **HD = ?** 3

- Detection and correction of single-bit errors
- Detection of double-bit errors

• • • • • • • •

⋮

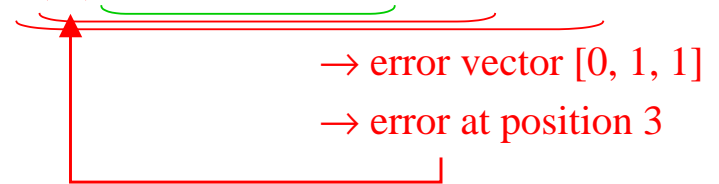
# Hamming Code

Example:

1011 → 0 1 1 0 0 1 1

Error (a):

0 1 ~~0~~ 0 0 1 1 →  $c3=0, c2=0, c1=1$



• • • • • • • •

⋮

## Hamming Code

Example:

1011 → 0 1 1 0 0 1 1

Error (b):

~~0~~1 0 0 0 1 1 →  $c_3=0, c_2=1, c_1=1$

→ error vector [0, 0, 1]  
→ error at position 1

⋮

## m-of-n Codes

- code word length: n bits
- code word has exactly m “1”s
- single-bit error will result in m-1 or m+1 “1”s  
→ error detection
- Sample implementation:
  - i bits data
  - add another i bits to assure code word has i “1”s
  - code bits required in relation to data bits: **100%**
  - **i-of-2i code**

• HD = ? 2

⋮

## Duplication Codes

- Completely duplicate information to build code word
- Example:  
0110 → 0110 0110

• HD = ? 2

⋮

## Berger Code

- Count number of “1”s in data word
- Add complement of result as check bits
- Example (7 data bits → 3 check bits required):  
00111010 → 4 “1”s →  $4 = 100_2$  → complement 011<sub>2</sub>  
Codeword: 00111010011
- Advantage:
  - detects all multiple, uni-directional errors
  - detects stuck-to-0, stuck-to-1

• HD = ? 2


⋮

## Arithmetic Codes

- Useful for checking arithmetic operators
- Operation to be checked needs invariant:  
 $\text{code}(a \otimes b) = \text{code}(a) \otimes \text{code}(b)$
- AN code
  - multiply data word by constant factor
  - invariant to addition/subtraction (not multiplication)
  - simplest version: 3N code


⋮

## Arithmetic Codes

- 3N code
- check validity of “+” operation by comparing:  
 $3N(a) + 3N(b)$  with  $3N(a+b)$   
 $3a + 3b \stackrel{?}{=} 3(a+b)$
- Example:
  - test arithmetic for “5+3”
  - $3N(5) = 15, \quad 3N(3) = 9, \quad 15+9 = 24$
  - $3N(5+3) = 3N(8) = 24$  

⋮

## Arithmetic Codes

- Residue code
- Add “residue” (remainder of division by constant) as check bits
- Separable code
- 3R code (remainder of division by 3, “modulo 3”)
- Example:
  - $14 \rightarrow 14 \text{ MOD } 3 = 2$  as bits: 111010
  - $7 \rightarrow 7 \text{ MOD } 3 = 1$  as bits: 011101
  - sum of data words: 21, 10101, sum of residues MOD 3: 00
  - residue of data word sum: 00 

⋮

## Cyclic Codes

- Code word is rotation-invariant
- Suitable for hardware like disks, tapes, etc.
- Check bits can easily be generated in hardware

⋮

## 2.3 Time Redundancy

### Idea:

- Hardware redundancy and information redundancy require extra hardware for calculation and storing  
→ *higher production cost*
- Instead of using extra hardware use time redundancy  
→ *longer processing time*
  - depends on application and system performance whether feasible
  - better suited for non-time-critical applications

.....

⋮

## 2.3.1 Transient Fault Detection

### Idea:

Perform same computation 2 or more times

- 2 times: fault detection
- 3 or more times: fault correction (by majority vote)

### Problem:

- Even a transient error may corrupt program data completely  
⇒ Transient fault may affect program counter, loop index, etc.
- ⇒ Repetition of previous calculation **not possible**

.....

⋮

## 2.3.2 Permanent Fault Detection

### Idea:

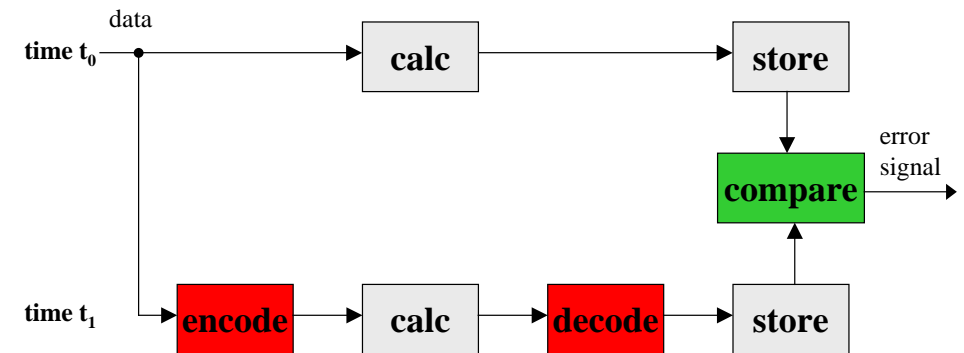
Calculating everything twice will not help, so instead:

1. calculate normal result
2. repeat calculation with encoded data  
(e.g. complement, arithmetic shift, etc.)

.....

⋮

## Permanent Fault Detection



.....

⋮

## Permanent Fault Detection

### Selection of encoding function

- must not alter calculation  
 $f(\text{data}) = \text{decode}(f(\text{encode}(\text{data})))$
- should be able to detect faults
- sample encoding functions:
  - complement (alternating logic)
  - arithmetic shift (rotation)

.....

⋮

## Permanent Fault Detection

### Complement Encoding

Calculate 1+5 with time redundancy

1. Normal:  $0001 + 0101 = 0110$
2. Two's complement:  $1111 + 1011 = 11010$   
Negate result:  $0110$

.....

⋮

## Permanent Fault Detection

### Rotation Encoding

Calculate 1 AND 5 with time redundancy

1. Normal:  $0001 \text{ AND } 0101 = 0001$
2. Rotation left:  $0010 + 1010 = 0010$   
Rotate right result:  $0001$

.....

⋮

## Permanent Fault Detection

- This approach is called **Alternate Logic** and is usually a technique for time redundancy
- Alternate Logic can also be applied to combinatorial circuits (low-level hardware redundancy)
- This requires **self-dual** circuits:  
A circuit is self-dual if  $f(X) = f'(X')$

.....

⋮

# Self-Dual Circuits

Any combinatorial circuit can be transformed into a self-dual circuit by using additional hardware and one additional input variable  $x_{n+1}$  to the existing  $X = (x_1, x_2, \dots, x_n)$

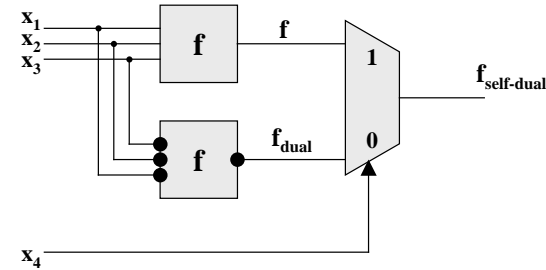
$$f_{\text{self-dual}} = x_{n+1} \cdot f + x_{n+1}' \cdot f_{\text{dual}}$$

with  $f_{\text{dual}} = f'(x_1', x_2', \dots, x_n')$

⋮

⋮

# Self-Dual Circuits



Self-dual means:  $f_{\text{self-dual}}(x_1, x_2, \dots, x_n) = \mathbf{NOT} f_{\text{self-dual}}(x_1', x_2', \dots, x_n')$

With the above diagram, the statement becomes trivial, e.g. assume  $x_4 = 1$

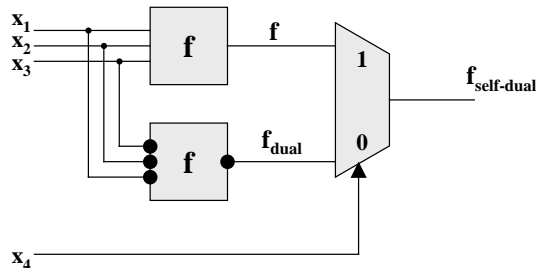
then:  $LHS(x_4 = 1) = f(x_1, x_2, x_3)$

$RHS(x_4 = 0) = \mathbf{NOT}(\mathbf{NOT} f(x_1'', x_2'', x_3'')) = f(x_1, x_2, x_3)$

⋮

⋮

# Self-Dual Circuits



How is a self-dual circuit checked?

Invert all input variables and check whether output bit also flipped.

⋮

⋮

# Self-Checking Circuits

- Self-dual circuits are one method for **self-checking** circuit should detect by itself whether a fault is present
- **In Theory:**
  - *Fault secure:* either output correct data (fault masking) or invalid code word (fault detection)
  - *Self-testing:* there is an input code word that will produce invalid output if a fault is present
  - *Totally self-checking:* fault secure and self-testing
- **In Practice:** good idea, but 100% are not achievable

⋮

⋮

## 2.4 Software Redundancy

- **Idea:** Use extra program code to ensure correctness of calculated result
  - consistency checks
  - capability checks
  - N version
  - N self-checking
  - recovery blocks

⋮

⋮

## Software Redundancy

**“Software does not break as hardware does”**

- Software faults are the result of incorrect design or coding mistakes
- Software faults exists at the first use of the system, it may however take some time until they show as an error.

⇒ Simple duplication of *same* procedure and comparison will **not** work!

⋮

⋮

## Consistency Checks

- For many applications a certain data range for the result is known beforehand
- Check data range, report error if out of bounds
- Examples:
  - temperature calculation (aircon vs. chemical process)
  - percentage calculation [0..1]
  - microprocessor: check for “illegal instruction codes” (can stop “run-away” process)

⋮

⋮

## Capability Checks

Verify that a system possesses the expected capability

- Check processor ALU with sample instructions, compare with known result from ROM
- Check memory by writing&reading sample values e.g. check each byte or use pattern to cover all memory chips
- Check network operation of other processors in multi-processor environment e.g. pass message between all PEs

⋮

⋮

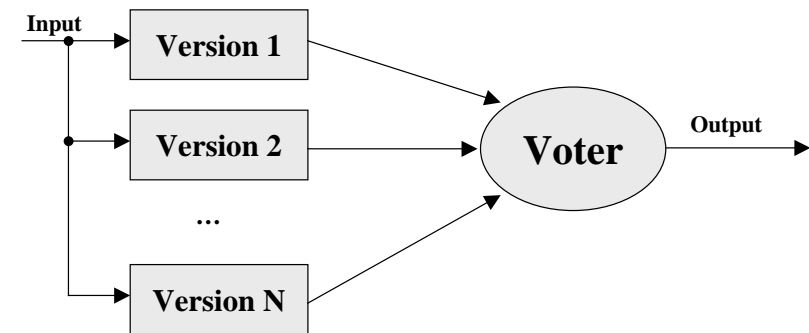
## N Version Programming

- Write N *independent, different* versions of the program
- Versions need to be independent, so a fault will not occur in all modules
- Can tolerate  $(N-1)/2$  faults
- Similar to TMR hardware redundancy

⋮

⋮

## N Version Programming



⋮

⋮

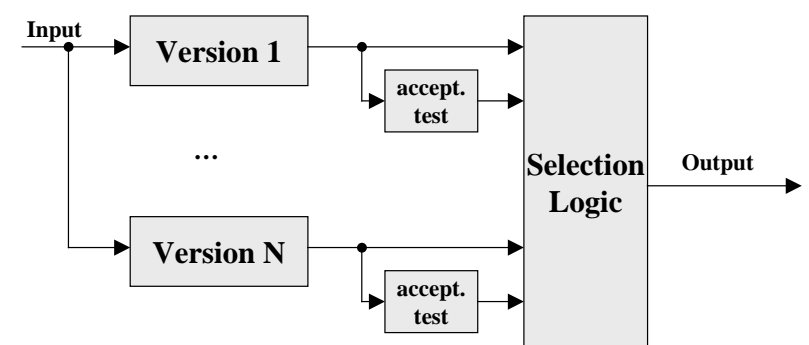
## N Self-Checking Programming

- Write N *unique* versions of the program, each with its own acceptance test
- Selection logic chooses result from one version that passes its acceptance test
- Can tolerate  $N-1$  faults
- Similar approach to “**hot standby**” hardware redundancy

⋮

⋮

## N Self-Checking Programming



⋮

⋮

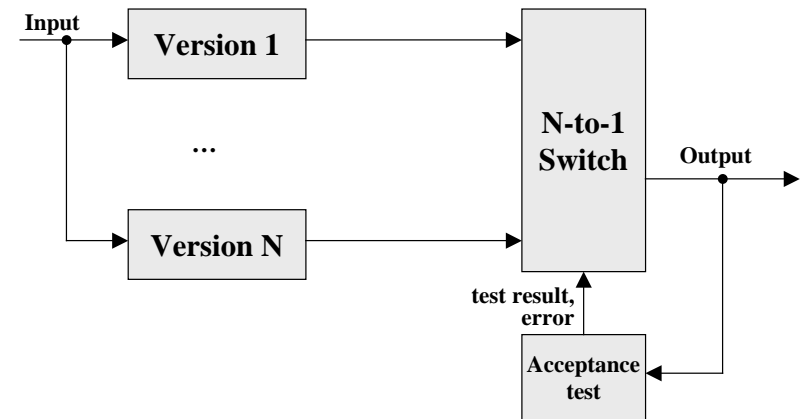
## Recovery Blocks

- Write N unique versions of program plus a **single** acceptance test
- If current program version fails, switch to next version
- System continues with next acceptable version
- System fails if no version passes test
- Can tolerate N-1 faults
- Similar to “**cold standby**” hardware redundancy

⋮

⋮

## Recovery Blocks



⋮

⋮

## Software Redundancy

### Some general remarks

- Software faults are usually much more frequent than hardware faults
- Therefore, if a system is developed with hardware redundancy, it often makes sense to also include software redundancy
- But: Software redundancy is quite expensive because of the man power involved (often more expensive than hardware redundancy)

⋮

⋮

## Software Redundancy

- Different versions need to be implemented by different programmers or teams
  - to avoid duplication of code segments
- Still, software developers tend to make similar mistakes
  - this would violate version independence
- All versions are developed from the same specification
  - a specification flaw cannot be masked by N versions

⋮