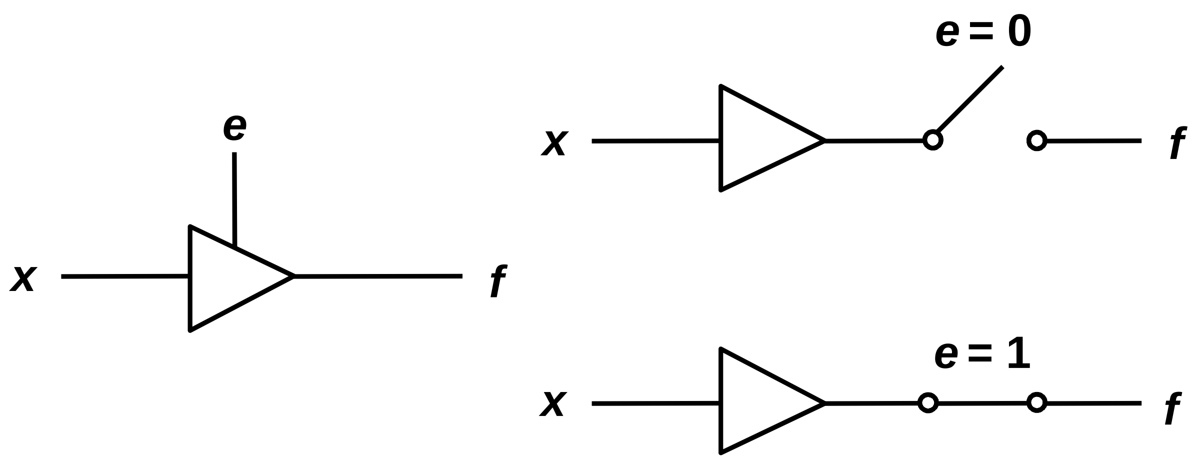
Lecture 6
Combinational Circuit Building Blocks

吳文中

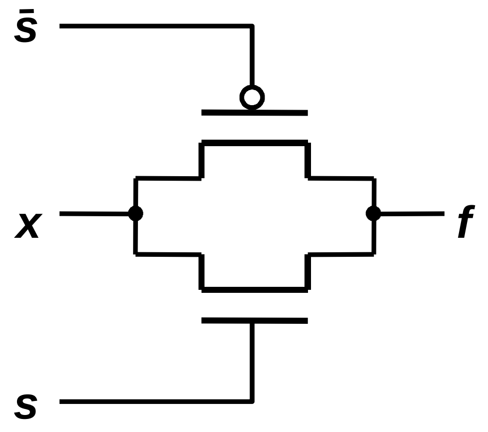
Tri-state-Buffer (3.8)

- Logic value Z, which is called the *high-impedance-state*.
- When e=0 the buffer is completely disconnected from the output *f*



e	*x*	*f*
0	0	Z
0	1	Z
1	0	0
1	1	1

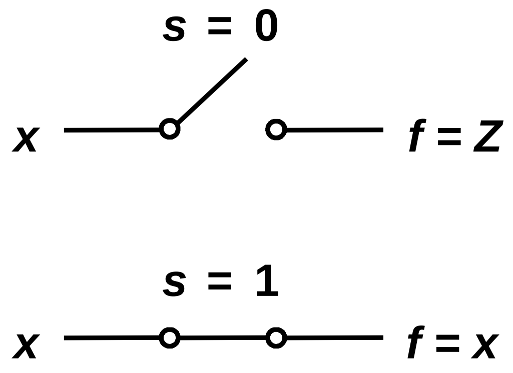
Transmission Gates (3.9)



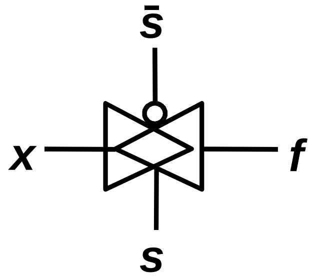
(a) Circuit

s	f
0	Z
1	x

(b) Truth table

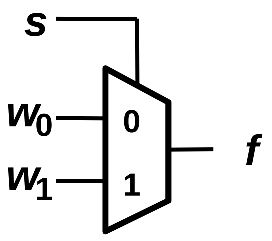


(c) Equivalent circuit



(d) Graphical symbol

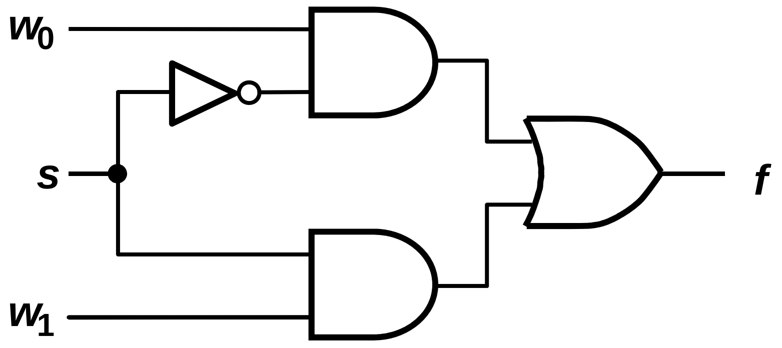
A 2-to-1 Multiplexer (MUX)



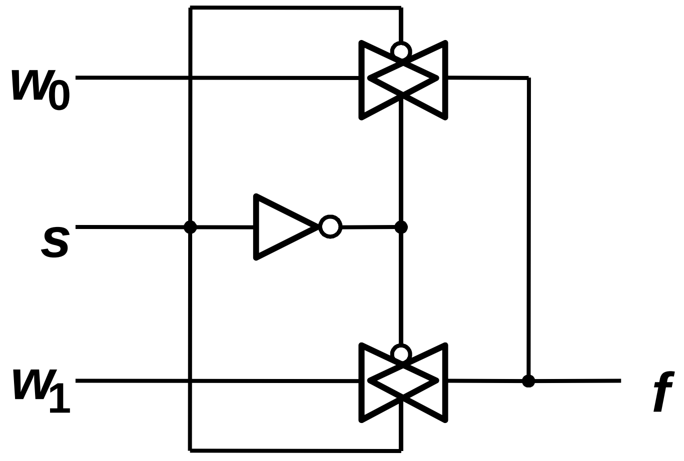
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table



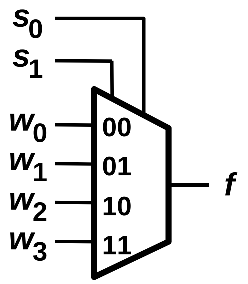
(c) Sum-of-products circuit



(d) Circuit with transmission gates

A 4-to-1 Multiplexer

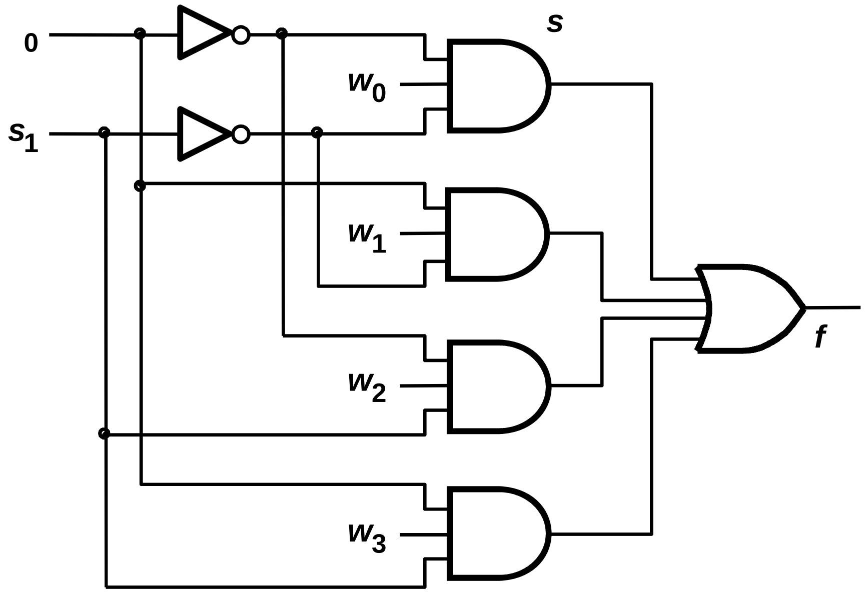
- $f = \overline{s_1}\,\overline{s_0}w_0 + \overline{s_1}s_0w_1 + s_1\overline{s_0}w_2 + s_1s_0w_3$



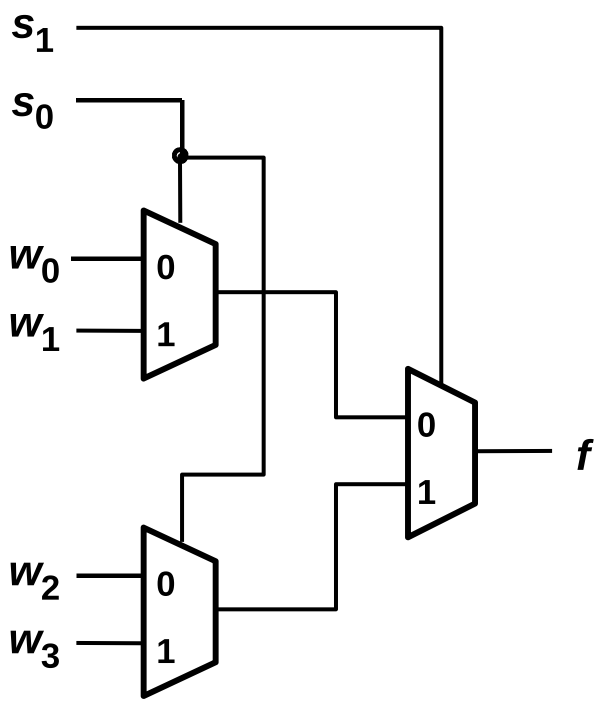
(a) Graphic symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

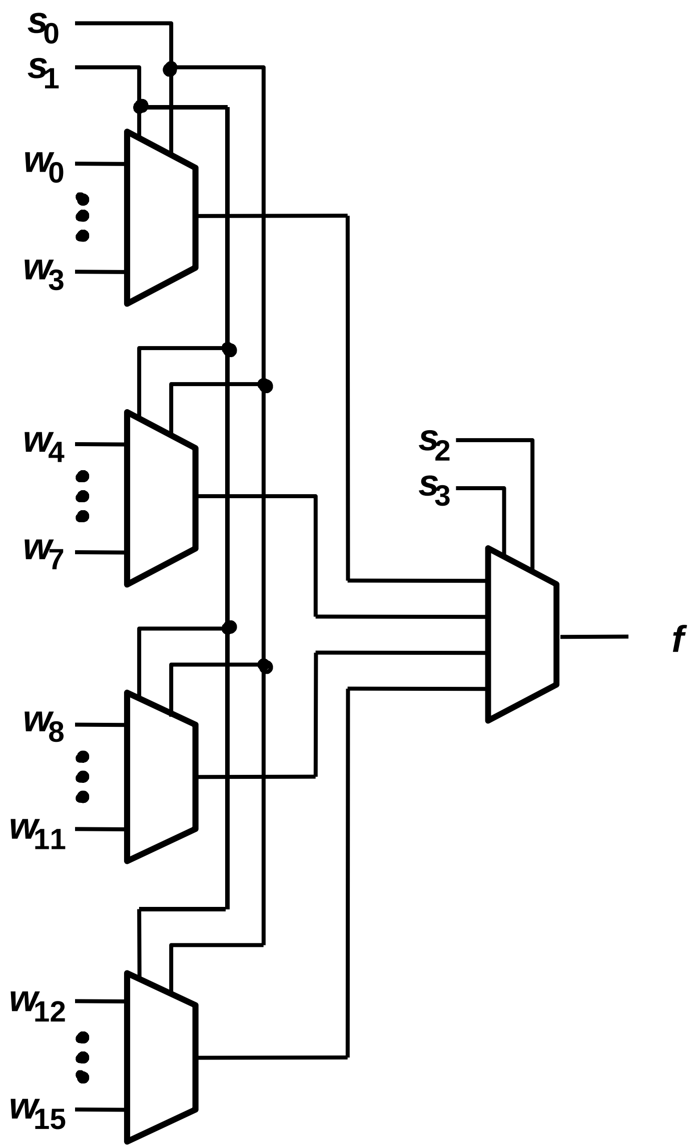


(c) Circuit

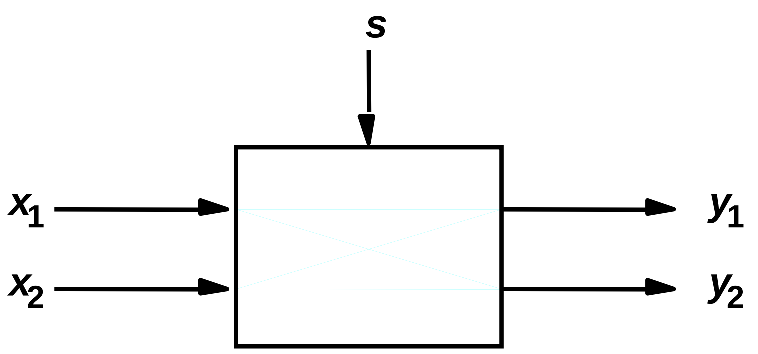


(d) Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

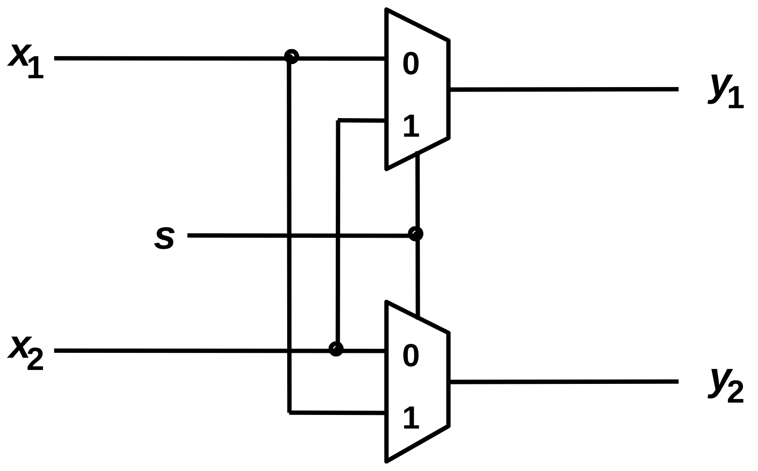
A 16-to-1 Multiplexer



Example 3.2 A 2x2 Crossbar Switch



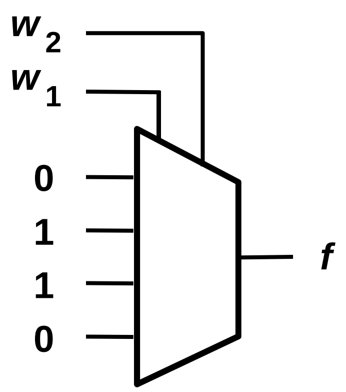
(a) A 2x2 crossbar switch



(b) Implementation using multiplexers

Synthesis of a Logic Function (XOR) using Multiplexers (LUT)

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

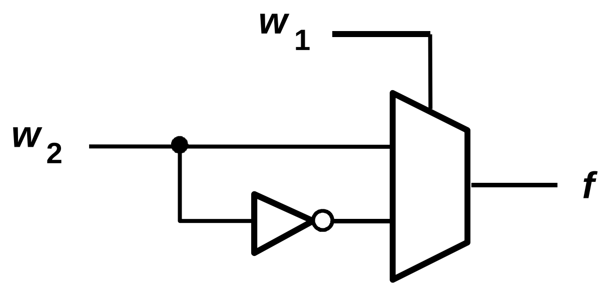


(a) Implementation using a 4-to-1 multiplexer

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\overline{w}_2$

(b) Modified truth table



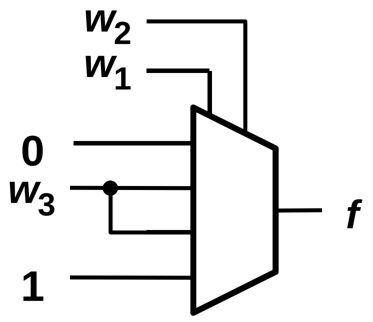
(c) Circuit

Implementation of the Three-input Majority Function using a 4-to-1 Multiplexer

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

(a) Modified truth table



(b) Circuit

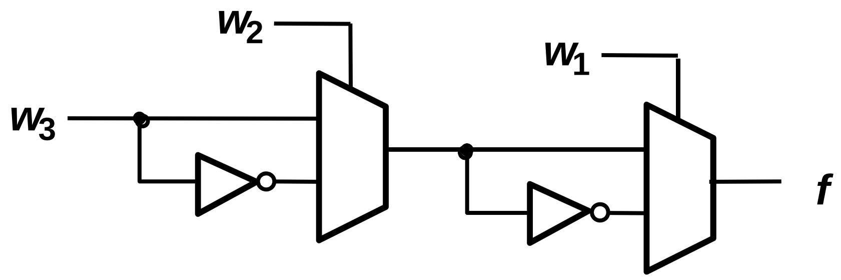
Tree-input XOR Implemented with 2-to-1 Multiplexer

w_1	w_2	w_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$w_2 \oplus w_3$

$\overline{w_2 \oplus w_3}$

(a) Truth table

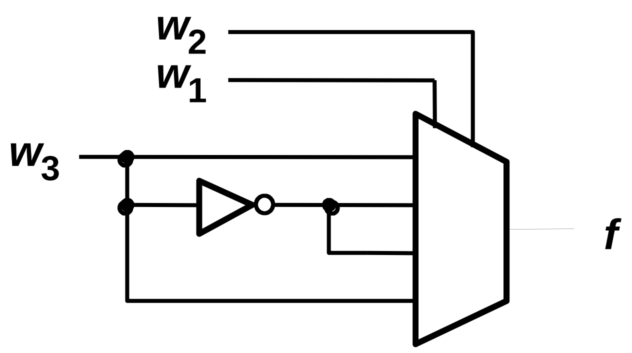


(b) Circuit

Tree-input XOR Implemented with 4-to-1 Multiplexer

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w}_3$
0	1	1	0	
1	0	0	1	$\overline{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



(b) Circuit

Implementation of the Three-input Majority Function using a 2-to-1 Multiplexer

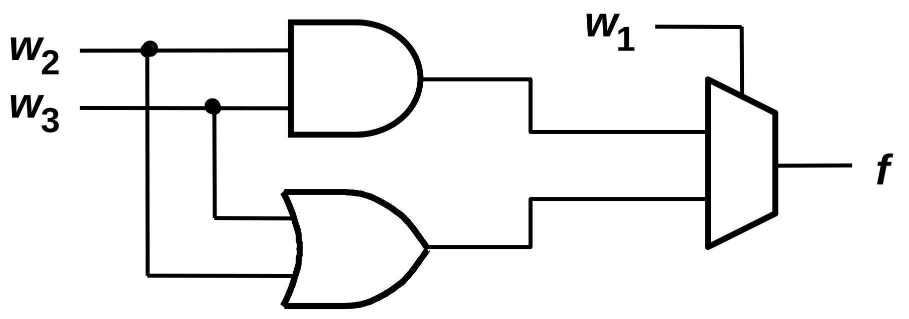
- $f = \overline{w_1}w_2w_3 + w_1\overline{w_2}w_3 + w_1w_2\overline{w_3} + w_1w_2w_3$

$$= \overline{w_1}(w_2w_3) + w_1(\overline{w_2}w_3 + w_2\overline{w_3} + w_2w_3)$$

$$= \overline{w_1}(w_2w_3) + w_1(w_2 + w_3)$$

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	w_2w_3
1	$w_2 + w_3$



(b) Circuit

(b) Truth table

Shannon's Expansion Theorem

- Any Boolean function $f(w_1, \ldots, w_n)$ can be written in the form

$$f(w_1, w_2, \ldots, w_n)$$
$$= \overline{w_1} \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$
$$= \overline{w_1} \cdot f_{\overline{w_1}} + w_1 \cdot f_{w_1}$$

- In general, if the expansion is done with respect to variable w_i, then f_{w_i} denotes

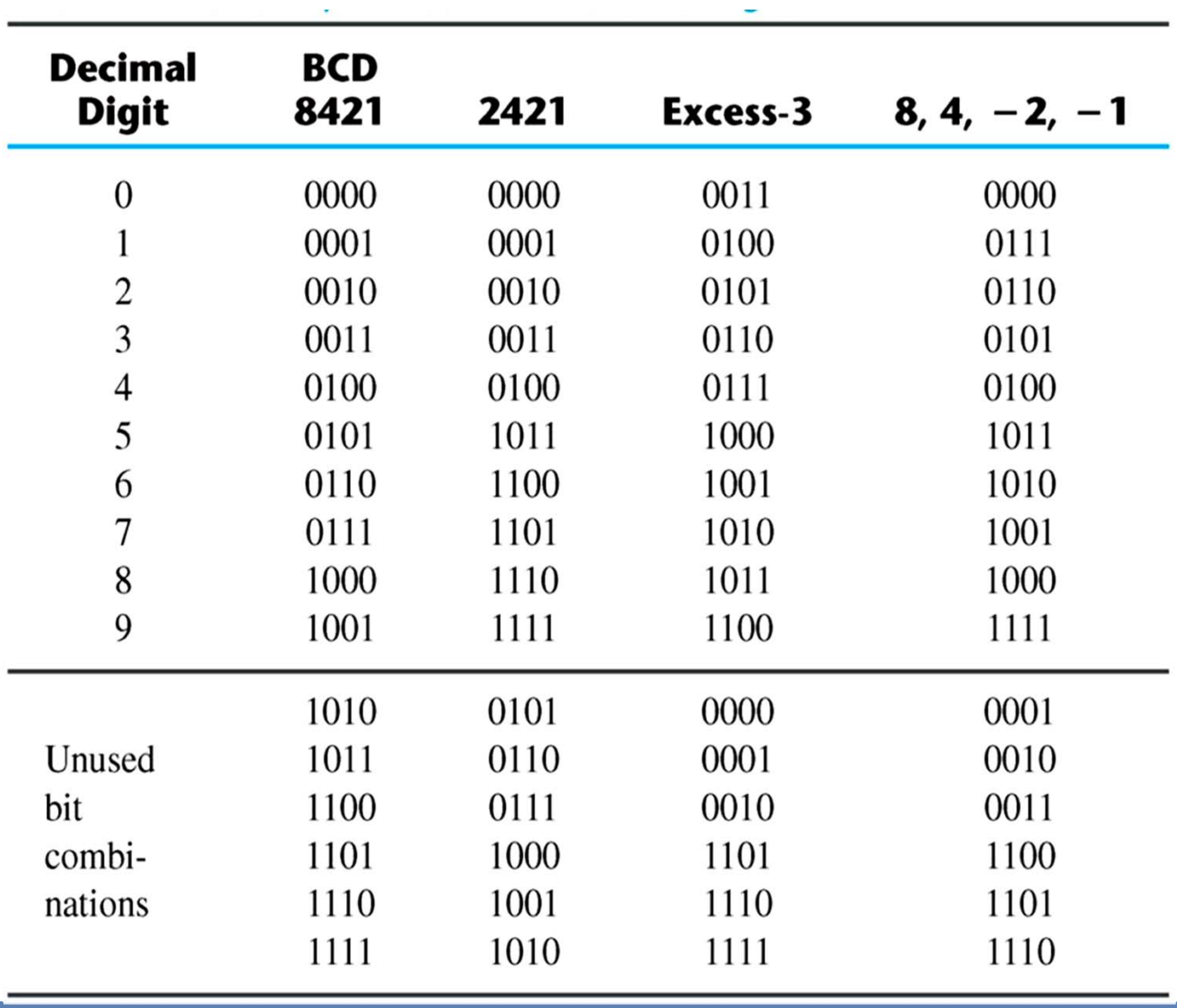
$f(w_1, \ldots, w_{i-1}, 1, w_{i+1}, \ldots, w_n)$,and

$$f = \overline{w_i} \cdot f_{\overline{w_i}} + w_i \cdot f_{w_i}$$

Code and Code Word

- A set of *n*-bit strings in which different bit strings represent different numbers or other things is called a *code.*
- A particular combination of *n* bit-values is called a *code word.*
- A code that uses *n*-bit strings need not contain 2^n valid code words.

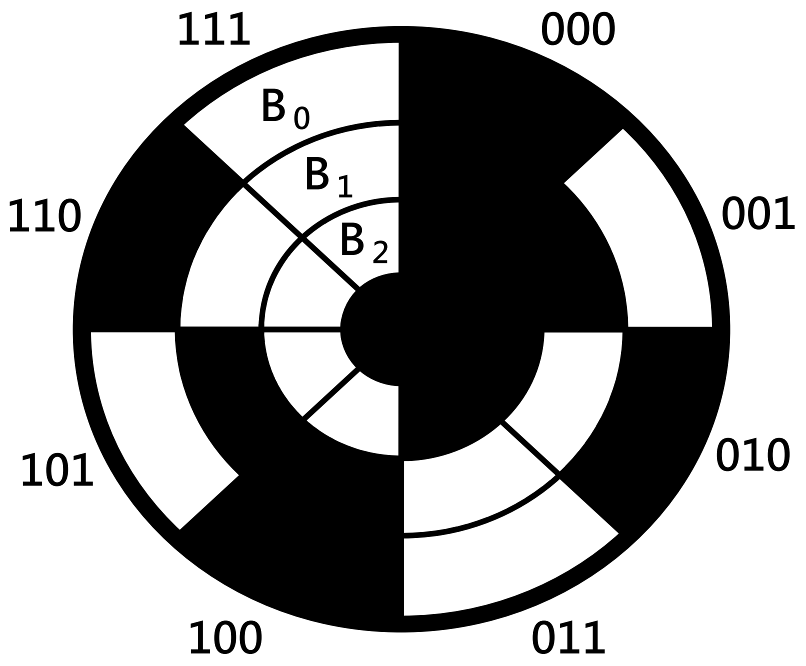
Decimal Codes



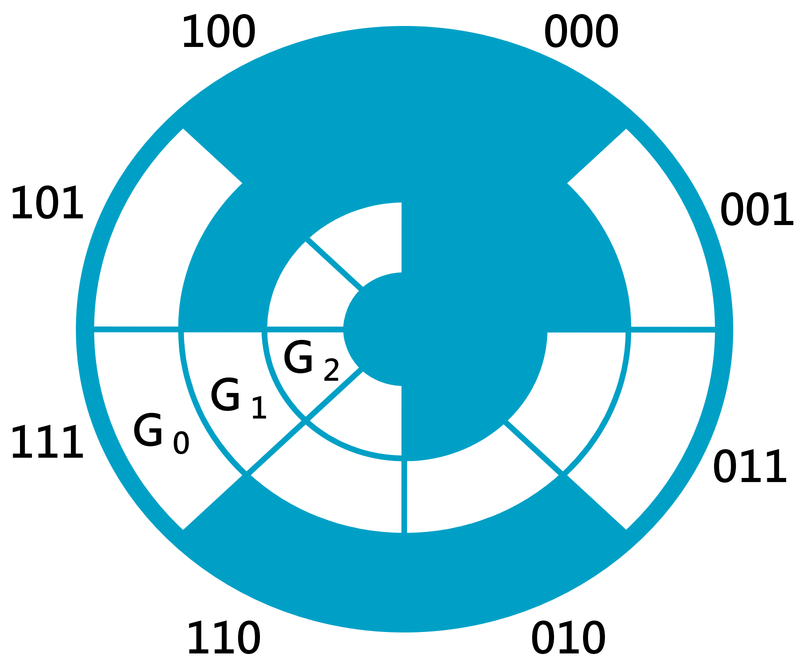
Decimal Digit	BCD 8421	2421	Excess-3	8, 4, −2, −1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused	1011	0110	0001	0010
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1100
nations	1110	1001	1110	1101
	1111	1010	1111	1110

Gray Code

- Advantage: only one it in the code group chages in going from one number to the next.
- Gray code generation: http://en.wikipedia.org/wiki/Gray_code



(a) Binary Code for Positions 0 through 7



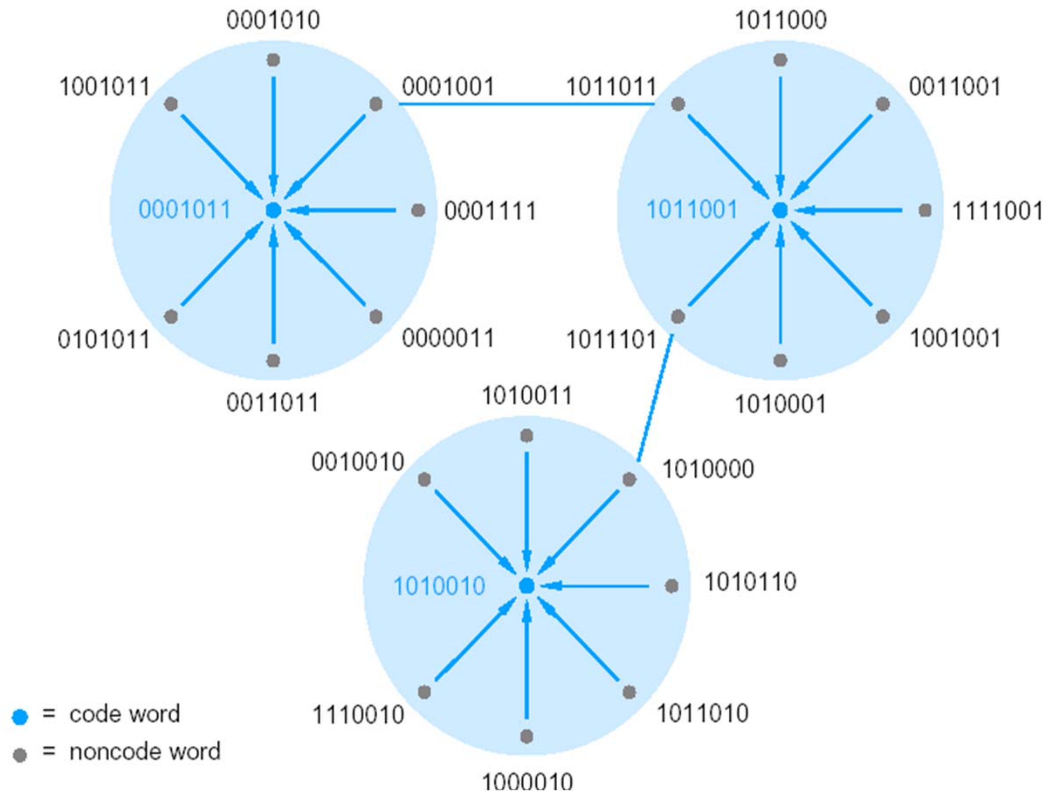
(b) Gray Code for Positions 0 through 7

Error Correction and Detection Codes

- Redundancy (e.g. extra information), in the form of extra bits, can be incorporated into binary code words to detect and correct errors.
- A simple form of redundancy is parity, an extra bit appended onto the code word to make the number of 1’s odd or even. Parity can detect all single-bit errors and some multiple-bit errors.
- A code word has even parity if the number of 1’s in the code word is even.
- A code word has odd parity if the number of 1’s in the code word is odd.

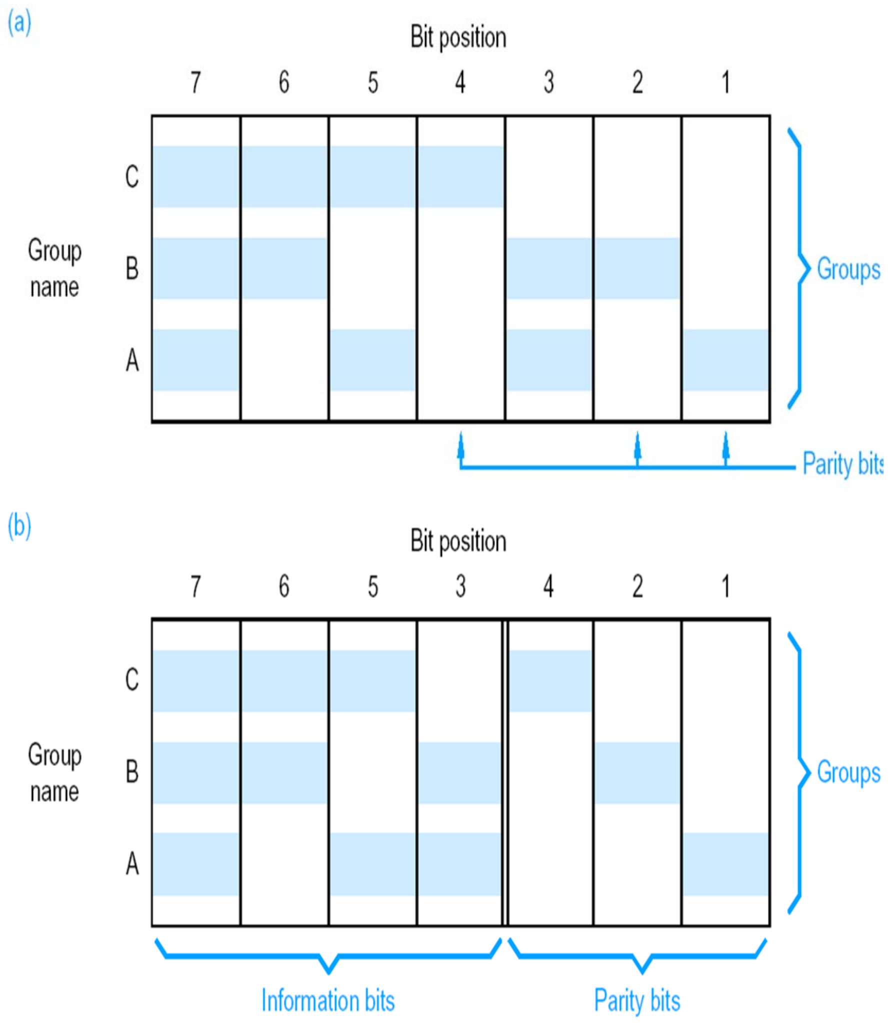
Error-Correcting and Multiple-Error-Detecting Codes

- An example can correct single errors and detect multiple errors.
  - With minimum distance 2c+1 can correct errors that affect up to c bits.
  - If a codes' minimum distance is 2c+d+1, it can be used to correct up to c bits error and detect up to d error bits.

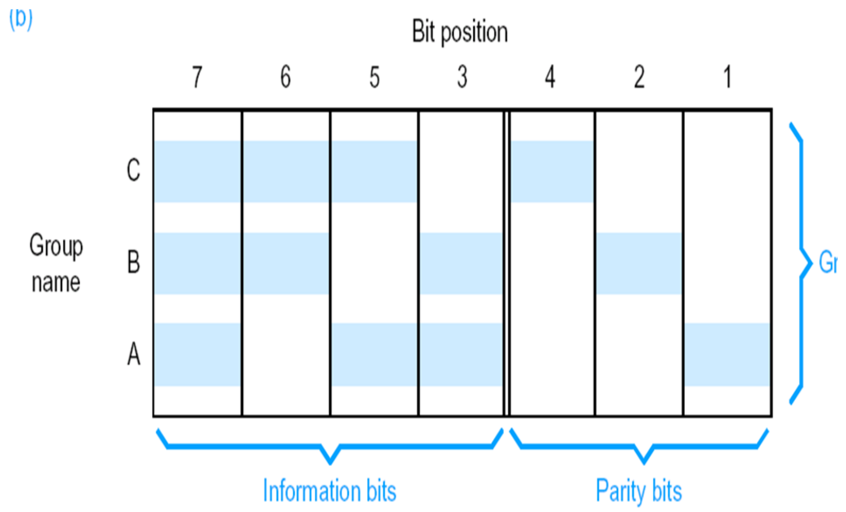


Hamming Code

- In 1950, R.W. hamming described a general method for constructing codes with a minimum distance of 3, now called hamming codes.
  - A (2^i-1)-bit code with i check bits and 2^i-1-i information bits.
  - Any bit position whose number is power of 2 is a check bit, and the remaining positions are information bits.
  - Each check bit is grouped with a subset of the information bits as specified by a *parity-check matrix*.
  - e.g. 0101 011 should be 0001 011



Generation of Parity Bits



$$= \left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}; \quad \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}\begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}; \quad \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Codeword:

0001 011 **0101 101** **1111 111**

Error Detection

- Correct codeword: 0001 011; 0101 101

$$\left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right]\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}; \left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right]\begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

1-bit Error Correction

- 0001 011 $\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow$ 0101 011

$$\left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & \boxed{1} & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = 6$$

$$\boxed{\text{Bit position}: 7 \quad 6 \quad 5 \quad 3 4 \quad 2 \quad 1}$$

- Nonzero number indicates the error bit position.

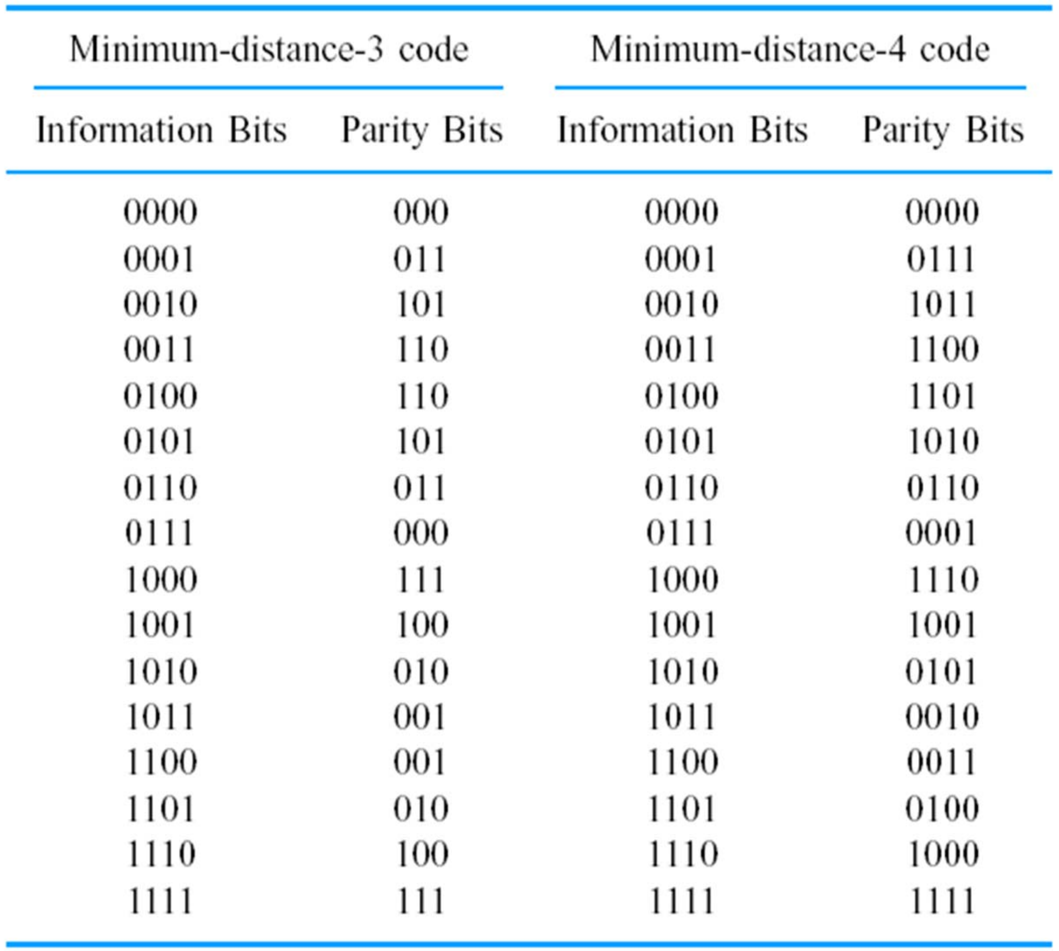
2-bit Error Detection

- 0001 011 → → → → → → → → → → → 0100 011

$$\left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & \boxed{1} & 0 & \boxed{1} & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right] \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = 5$$

- Nonzero number indicates the error occurs, but can’t be corrected back.

Distance-3 and Distance-4 Hamming Code



Minimum-distance-3 code		Minimum-distance-4 code	
Information Bits	Parity Bits	Information Bits	Parity Bits
0000	000	0000	0000
0001	011	0001	0111
0010	101	0010	1011
0011	110	0011	1100
0100	110	0100	1101
0101	101	0101	1010
0110	011	0110	0110
0111	000	0111	0001
1000	111	1000	1110
1001	100	1001	1001
1010	010	1010	0101
1011	001	1011	0010
1100	001	1100	0011
1101	010	1101	0100
1110	100	1110	1000
1111	111	1111	1111

Cyclic-redundancy-check (CRC) Codes

- Appends a few (typically 16 or 32) bits to the end of the bit string for a message and sends out the extended string.
- The receiver then performs a computation which would yield 0 if no bits of the message had been in error; if the result is not 0, then the receiver knows that there has been an error in one or more bits.
- Let M be the message we wish to send, *m* bits long. Let C be a divisor string, *c* bits long. C will be fixed (say hardwired into a serial I/O chip), while M (and *m*) will vary. We must have that:
  - $m > c-1$
  - $c > 1$
  - The first and last bits in C are 1s.

CRC Polynomial and Arithmetic

- 1101 means $x^3 + x^2 + 1$; 1011 means $x^3 + x + 1$
- $(x^3 + x^2 + 1) \times (x^3 + x + 1)$
  $= (x^6 + x^5 + x^3) + (x^4 + x^3 + x) + (x^3 + x^2 + 1)$
  $= x^6 + x^5 + x^4 + 3x^3 + x^2 + x + 1$
  $= x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ (mod 2)
- The polynomial arithmetic is the same as ordinary arithmetic except all carriers are ignored.
- There are only four cases for each bit position: 0+0=0;0+1=1;1+0=1;1+1=0 (no carry)
- The operation is the same as XOR. How about subtraction?
- Is 1010 > 1001?
- 1001 = 1010 + 0011; 1001=1010 - 0011

```
    1101
 x  1011
--------
    1101
   1101x
  0000xx
 1101xxx
--------
 1111111
```

Cyclic-redundancy-check (CRC) Algorithm

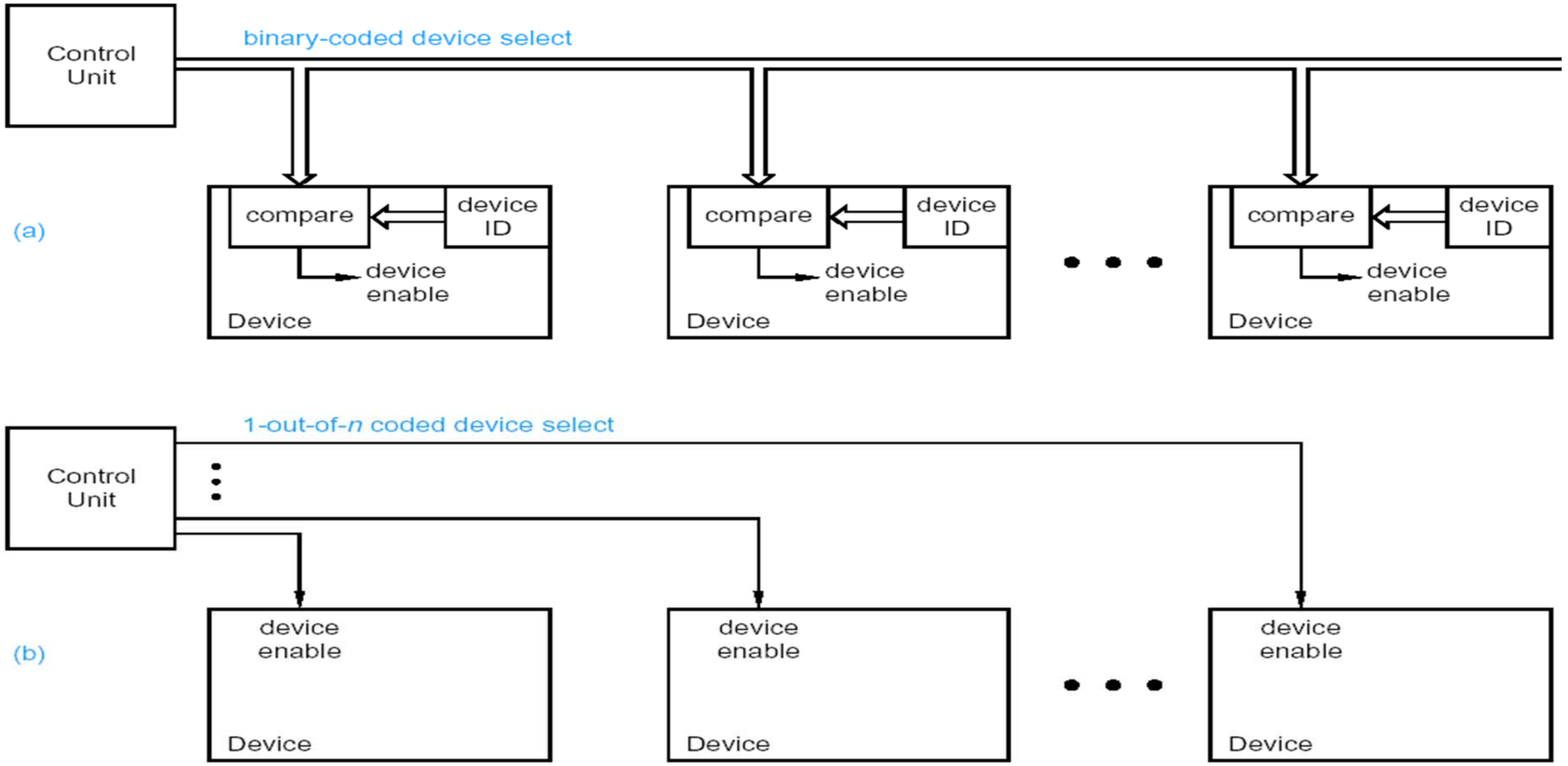
- The CRC field will consist of a string R, *c*-1 bits long. Here is how to generate R, and send the message:
  - 1. Append *c*-1 0s to the right end of M. These are placeholders for where the CRC will go. Call this extended string M'.
  - 2. Divide M' by C, using mod-2 arithmetic. Call the remainder R. Since we are dividing by a *c*-bit quantity, R will be *c*-1 bits long.
  - 3. Replace the *c*-1 appended 0s in M' by R. Call this new string W.
  - 4. Send W to the receiver.

- **e.g. say C = 1011 and M = 1001101. Then we divide as follows:**

```
         1010011
1011 )1001101000
      1011
        1010
        1011
           1100
           1011
            1110
            1011
             101
```

1-out-of-*n* Code

- An *n*-bit code in which valid code words only have one bit equal to 1 and the rest of the bits equal to 0.
- *Inverted 1-out-n* code. only 1 bit 1 and the rest of the

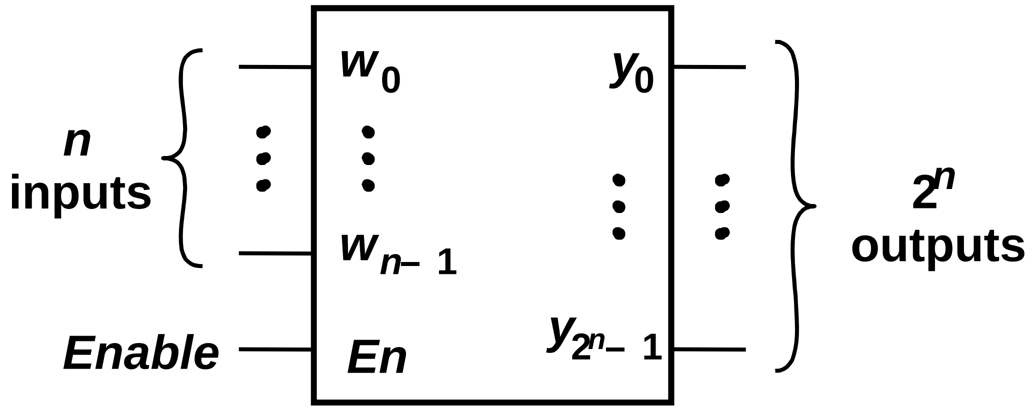


m-out-of-*n* Code

- Generalization of the 1-out-of-*n* code
  - *m* bits equal to 1 and the rest of the bits equal to 0.
- The total number of code words is given by binomial coefficient $\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$
- 8B10B code used in Gigabit Ethernet standard.
  - 10 bits to represent 256 valid code words, or 8 bits worth of data.

Decoder

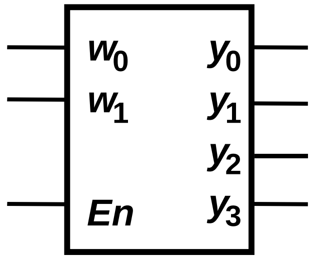
- n-bit binary codes (n inputs) decoded into 2^n 1-out-of-n codes (one-hot encoding, 2^n outputs).
- The output of a binary decoder are one-hot encoded.



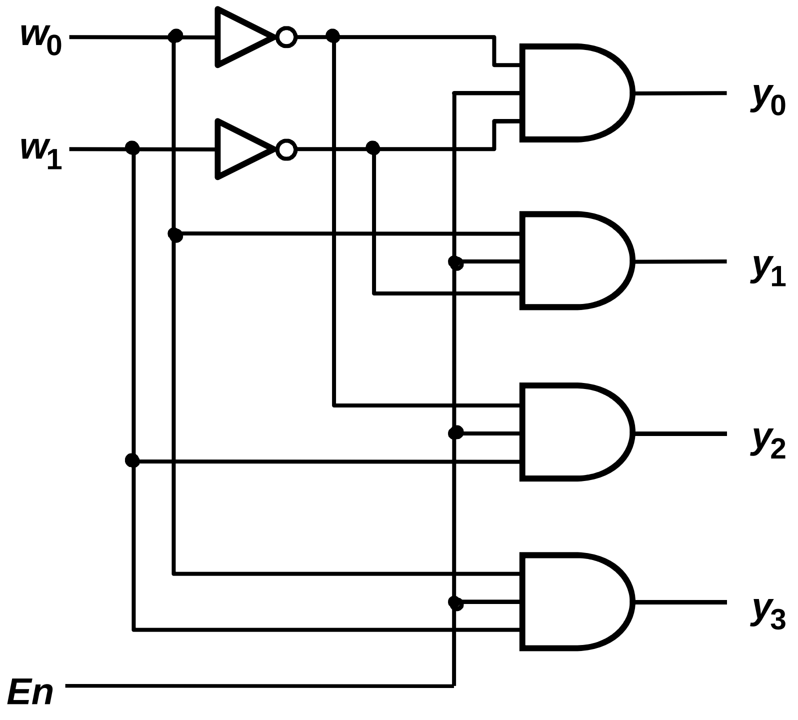
A 2-to-4 Decoder

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table

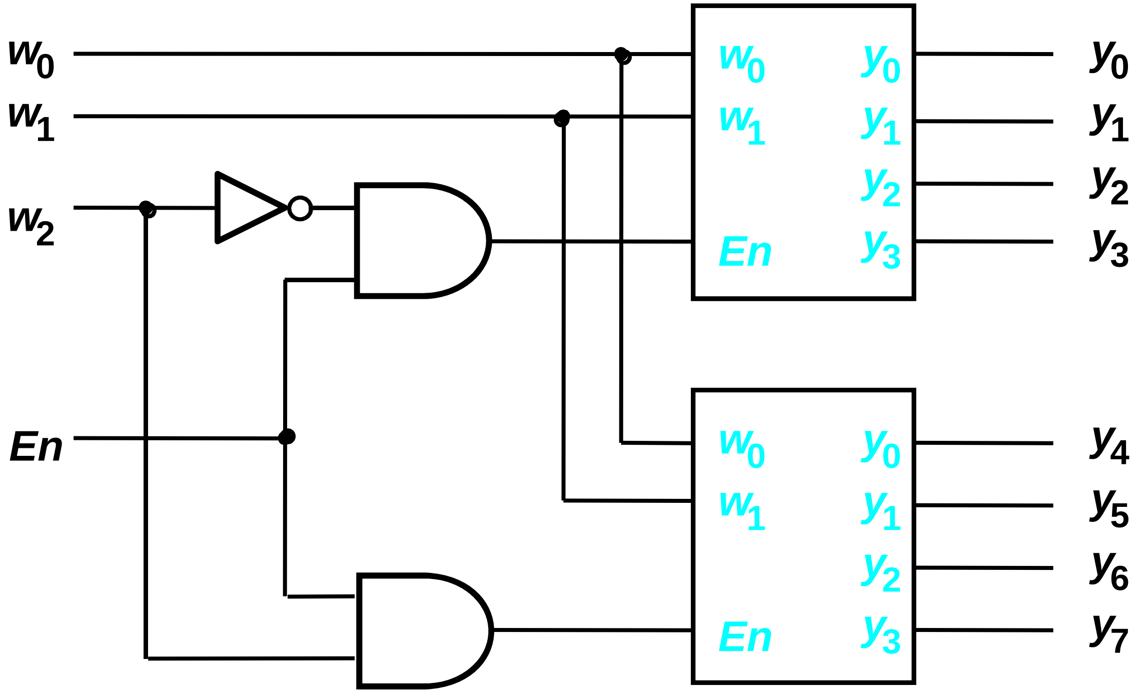


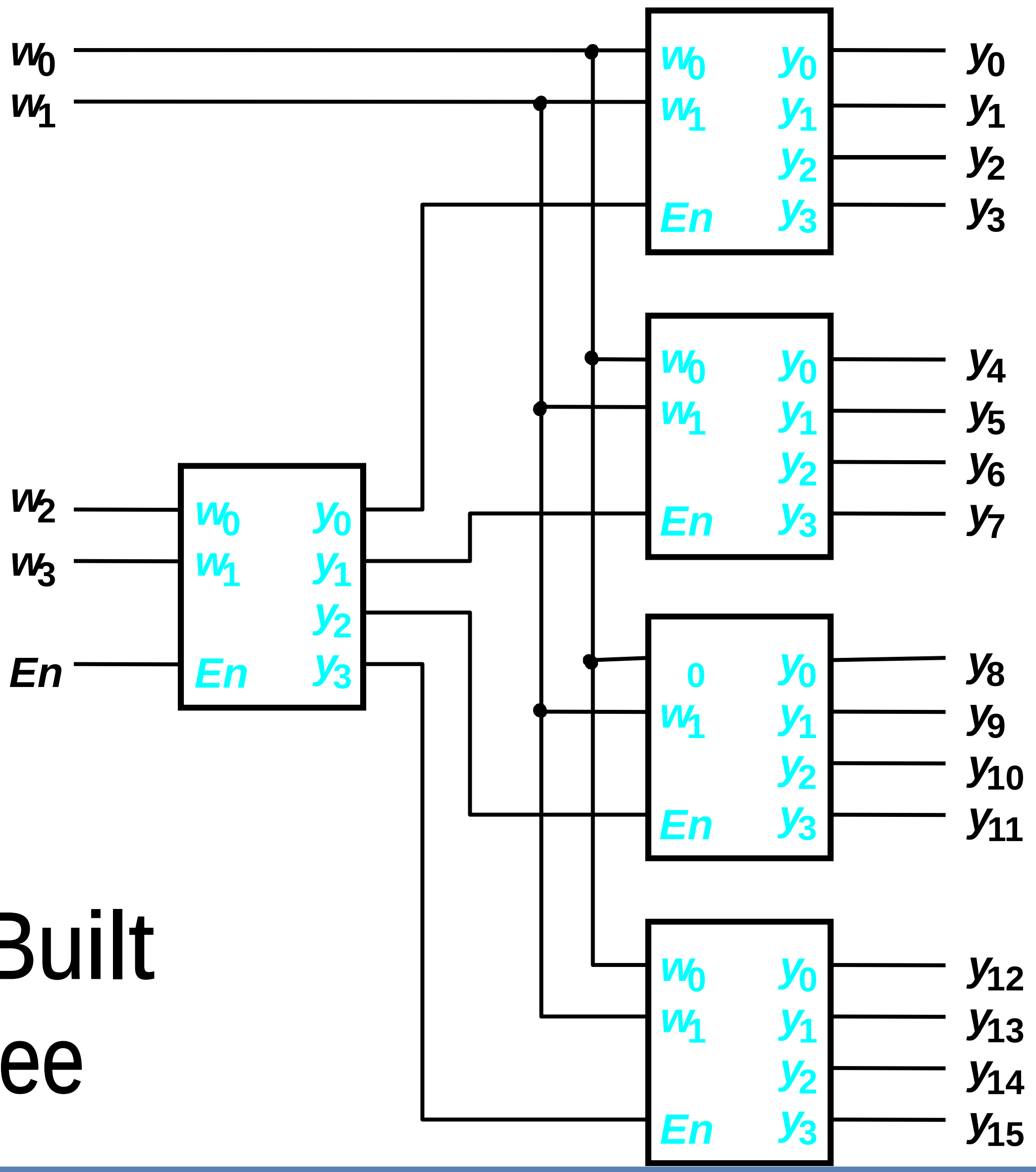
(b) Graphical symbol



(c) Logic circuit

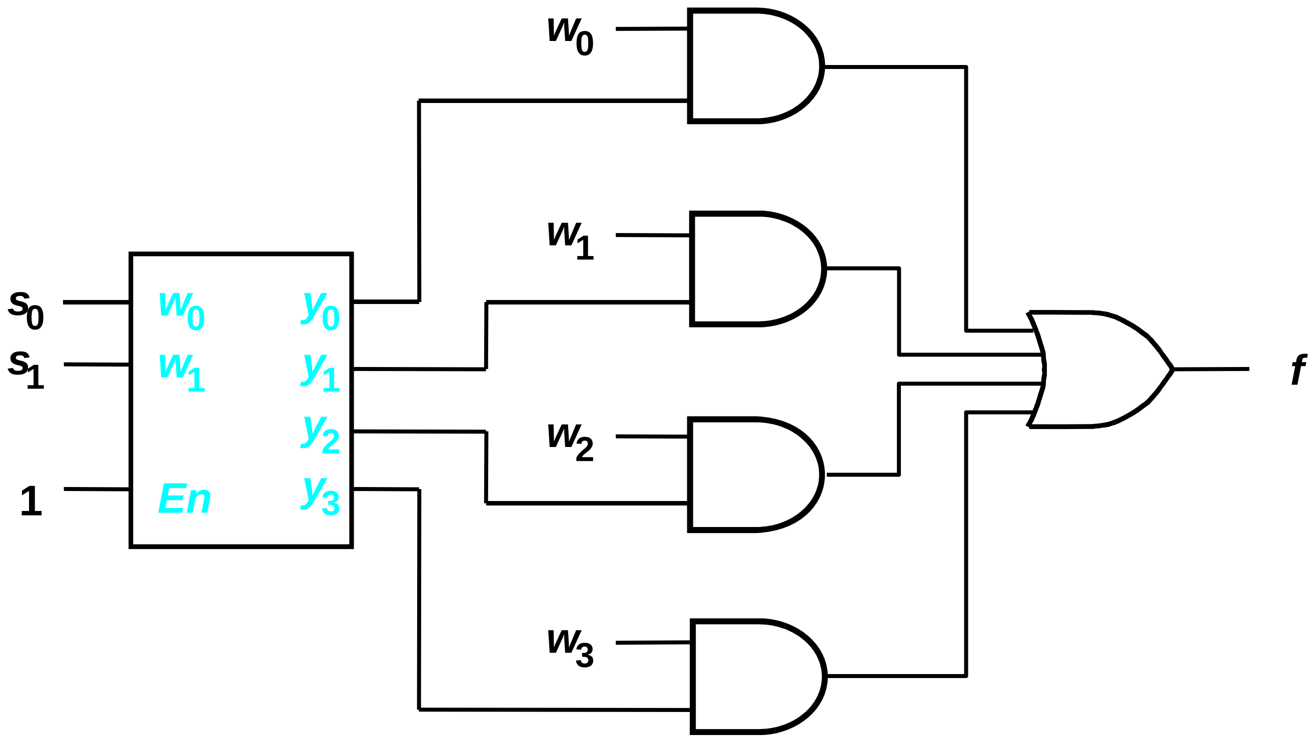
A 3-to-8 Decoder using Two 2-to-4 Decoders



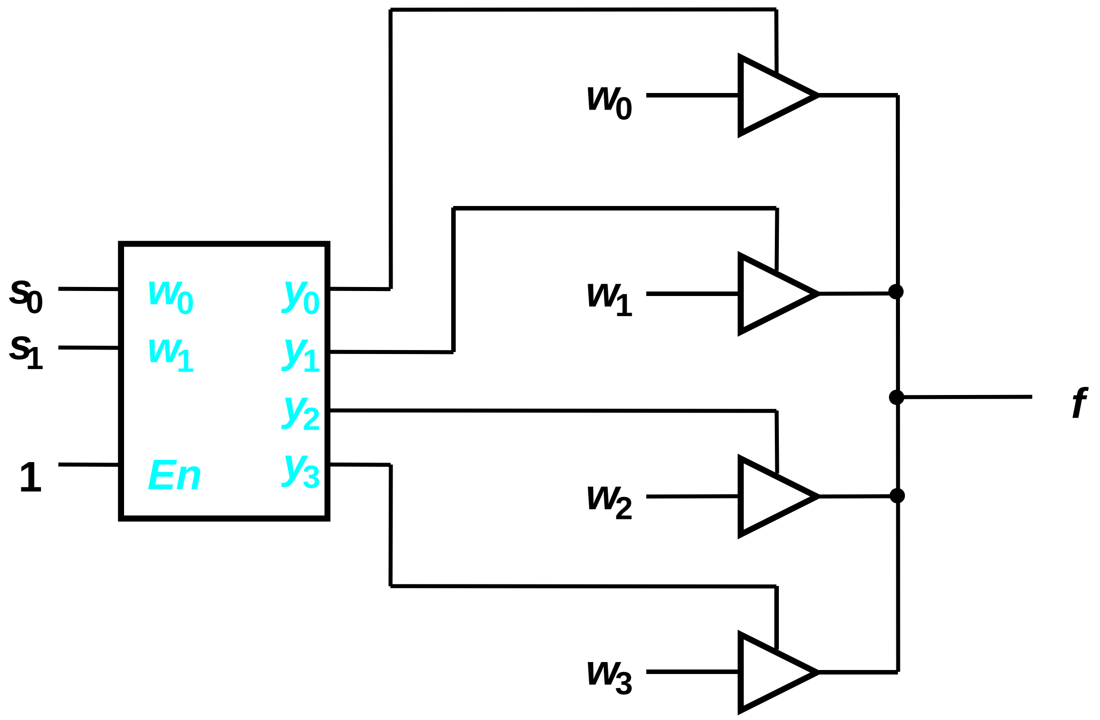


A 4-to-16 Decoder Built using a Decoder Tree

A 4-to-1 Multiplexer Built using a Decoder.



A 4-to-1 Mux built using a Decoder and Tri-state Buffers

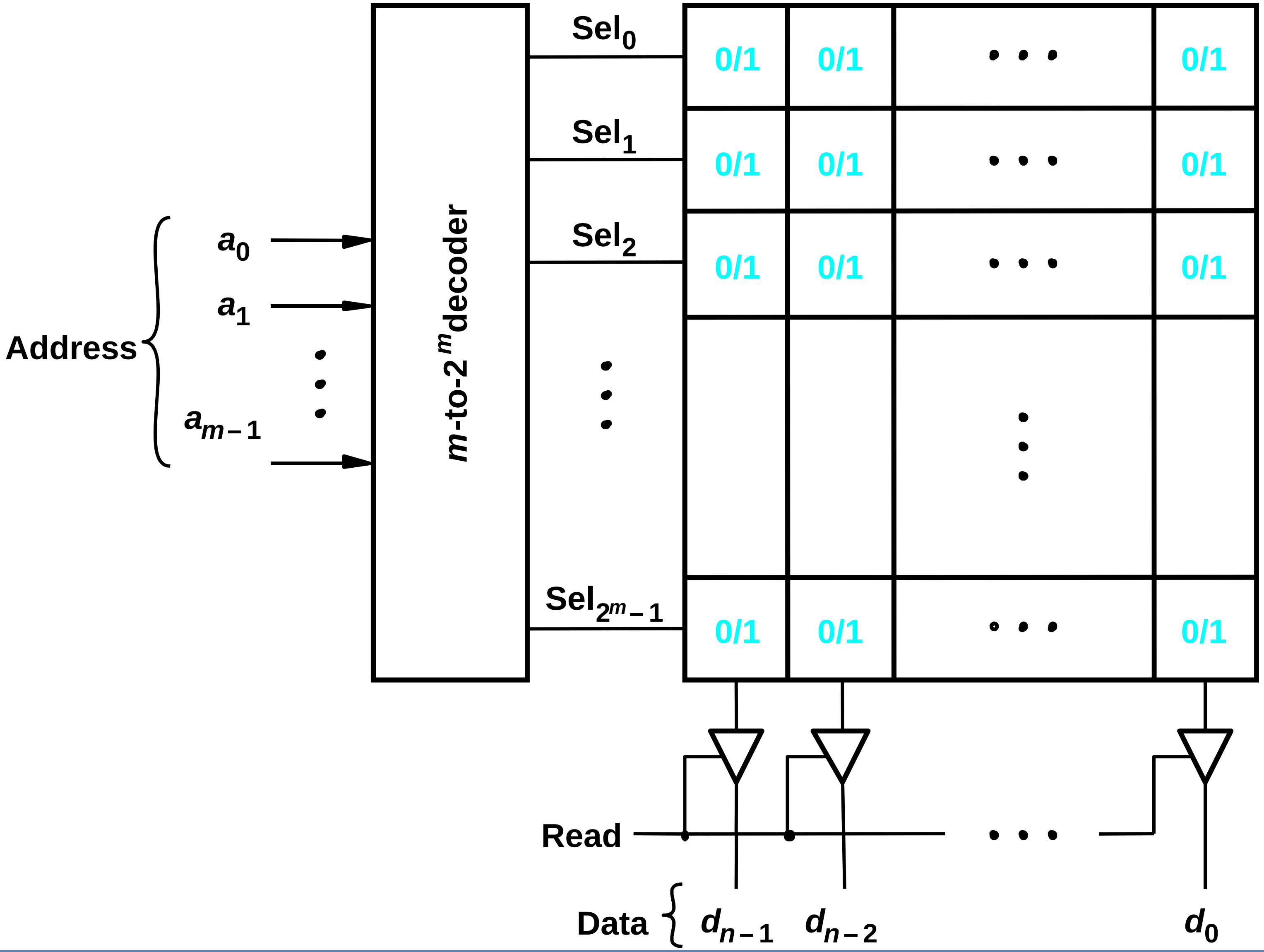


Demultiplexer (DEMUX)

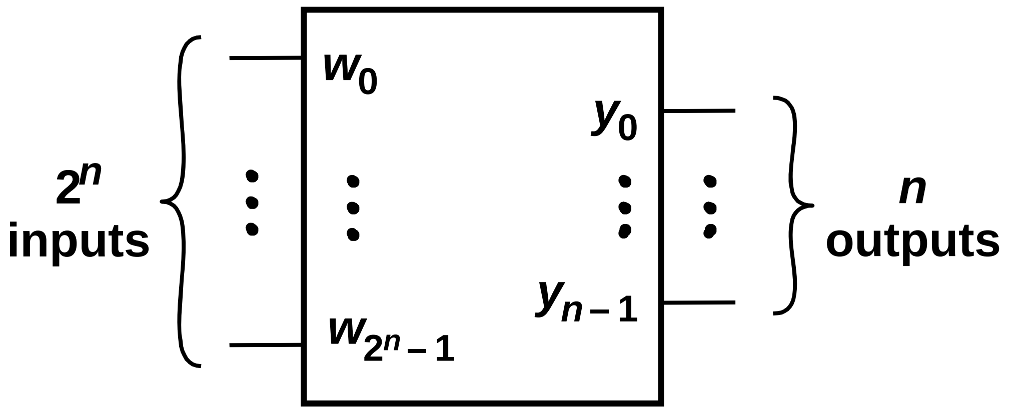
- A 1-to-n demultiplexer performs the opposite function of a n-to-1 multiplexer.
- A n-to-2^n decoder can be used as a 1-to-n demultiplexer, with En as the data input, decoder input as select inputs, and decoder outputs as data output.
- Ex. Implement a 1-to-4 decoder using a 2-to-4 decoder.

In	S_0	S_1	O_0	O_1	O_2	O_3
En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

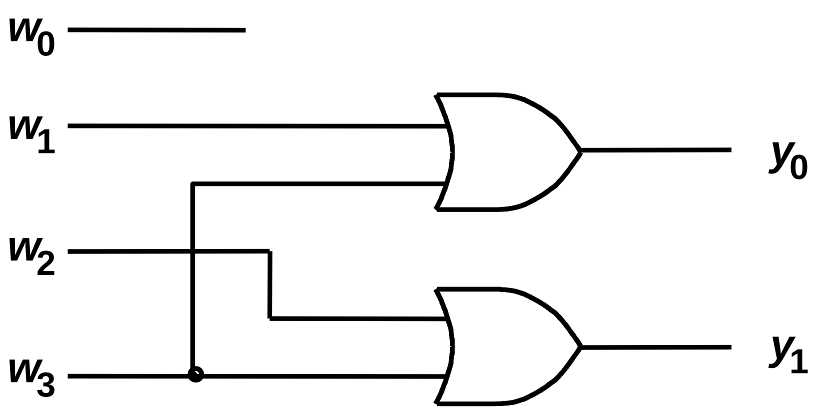
Ex 6.11 Memory Address Decoder



Binary Encoders



w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

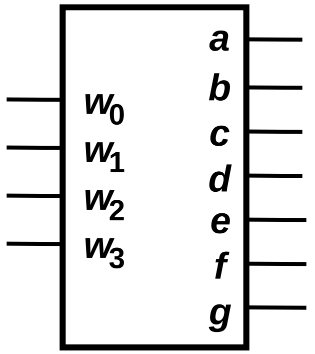


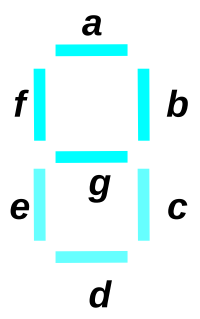
Priority Encoders

- $i_0 = \overline{w_3 w_2 w_1} w_0$
- $i_1 = \overline{w_3 w_2} w_1$
- $i_2 = \overline{w_3} w_2$
- $i_3 = w_3$
- $y_0 = i_1 + i_3$
- $y_1 = i_2 + i_3$
- $z = i_0 + i_1 + i_2 + i_3$

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

BCD-to-7 Segment Display Code Coverter





w_3	w_2	w_1	w_0	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Arithmetic Comparison Circuits

- Define $A = a_3a_2a_1a_0$ and $B = b_3b_2b_1b_0$
- Define $i_k = \overline{a_k \oplus b_k}$
- The comparator's output $AeqB = i_3i_2i_1i_0$
- $AgtB = a_3\overline{b_3} + i_3a_2\overline{b_2} + i_3i_2a_1\overline{b_1} + i_3i_2i_1a_0\overline{b_0}$
- $AltB = \overline{AeqB + AgtB}$