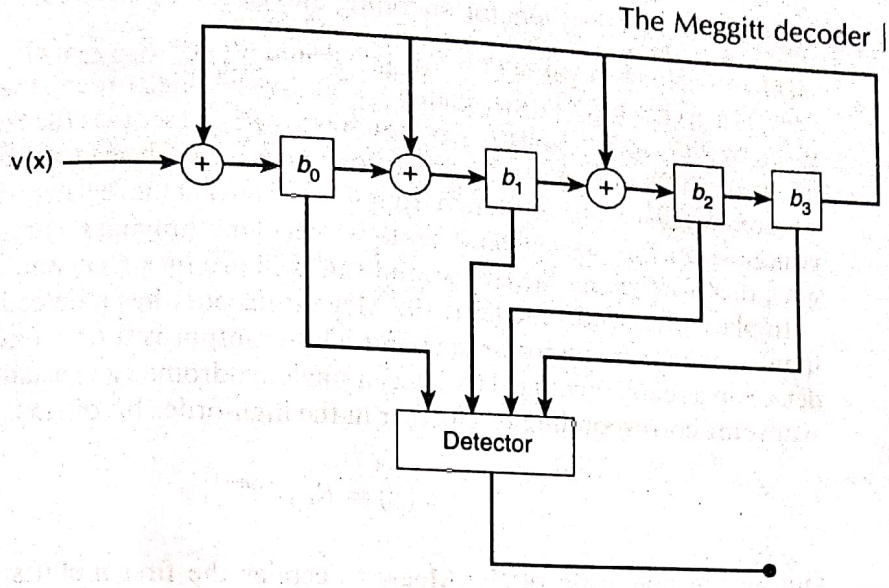


Fig. 4.16 An error-detection decoder for the (7, 3) code with $g(x) = x^4 + x^2 + x + 1$.

stages. At each subsequent shift the output of the stages e_6 and d_6 are added together and fed into the codeword stages. By the 10th shift the erroneous bit and error position have reached the stages d_6 and e_6 respectively, and at the 11th shift the erroneous bit is corrected as it leaves d_6 and enters c_0. Shifts 12, 13, and 14 are required to move the remaining bits into the codeword stages. The final state of the codeword stages is (1 1 0 0 1 0 1) which gives $c(x) = x^6 + x^4 + x + 1$ as required.

A decoder for error detection does not require a syndrome table but just a polynomial-division register, to determine the error syndrome $s(x)$, and a detector to indicate whether or not $s(x)$ is zero. If the detector indicates that $s(x)$ is nonzero then the input $v(x)$ contains errors, otherwise $v(x)$ is a codeword. Consider for example the (7, 3) code, with generator polynomial $g(x) = x^4 + x^2 + x + 1$, used for error detection. Figure 4.16 shows a decoder consisting of a polynomial-division register with its 4 stages feeding into a detector. Here the detector is an OR gate with 4 inputs and whose output is 1 if any of the inputs are 1. If b_0, b_1, b_2, or b_3 are nonzero, after $v(x)$ has entered the register, the detector output will be 1, indicating that $s(x)$ is nonzero and that $v(x)$ therefore contains errors. The detector output is 0 only if b_0, b_1, b_2, and b_3 are 0, that is when $v(x)$ is a codeword polynomial.

4.5 The Meggitt decoder

The Meggitt decoder avoids the need for a syndrome table by computing the error syndromes of all correctable error patterns from a small number of error syndromes. In the case of a single-error-correcting code all the syndromes can be determined from any one error syndrome. Consider a cyclic code with generator polynomial $g(x)$ and let $e(x)$ be a correctable error pattern and $e'(x)$ a 1 bit cyclic shift of $e(x)$ towards high orders. If $s(x)$ and $s'(x)$ are the error syndromes of $e(x)$ and

$e'(x)$ respectively then $s(x) = R_{g(x)}[e(x)]$ and $s'(x) = R_{g(x)}[xs(x)]$. Furthermore, if $e''(x)$ is a cyclic shift of $e'(x)$, again by 1 bit towards high orders, then its syndrome is given by $s''(x) = R_{g(x)}[xs'(x)]$. Continuing this we can see that the error syndromes of all cyclic shifts of $e(x)$ can be obtained from $s(x)$. With regards to the polynomial division register $s'(x)$ is obtained from $s(x)$ by shifting the register once. If the register contains $s(x)$, then a single shift has the effect of multiplying $s(x)$ by x and dividing by $g(x)$, therefore giving $s'(x)$. A second shift will produce $s''(x)$ and so forth.

In place of a syndrome table the Meggitt decoder has a detection circuit whose input is the error syndrome $s(x)$ and whose output is 0 or 1 (see Fig. 4.17). The detection circuit is arranged to detect a single syndrome $s_d(x)$ usually taken to be the syndrome corresponding to an error in the high-order bit of $v(x)$, so that

$$s_d(x) = R_{g(x)}[x^{n-1}].$$

During the operation of the Meggitt decoder the first n shifts are required for entering $v(x)$ into the register and into the stages $d_0, d_1, \ldots, d_{n-1}$. By the end of the nth shift the stages $b_0, b_1, \ldots, b_{n-k-1}$ contain the syndrome $s(x) = R_{g(x)}[v(x)]$ and $d_0, d_1, \ldots, d_{n-1}$ contain $v(x)$. If $s(x) = s_d(x)$ the output of the detector, denoted by s_d, is 1 otherwise it is 0. The output of stage d_{n-1} is added to the output of the detector s_d, so that the input to the stage c_0 is $d_{n-1} + s_d$. If the stage d_{n-1} contains an erroneous bit then $s(x) = s_d(x)$, $s_d = 1$ and the bit will be corrected as it leaves d_{n-1} to enter c_0. At the next shift the error syndrome will be $s'(x) = R_{g(x)}[xs(x)]$ and the process repeats itself. If $s'(x) = s_d(x)$, then $s_d = 1$ and the bit leaving d_{n-1} is inverted as it enters c_0, otherwise the bit is unaltered. This continues until $v(x)$ has left $d_0, d_1, \ldots, d_{n-1}$ and occupies the stages $c_0, c_1, \ldots, c_{n-1}$. A total of $2n$ shifts are required for decoding, where the first n shifts are for determining the error syndrome of $v(x)$ and the remaining n are for error correction.

Figure 4.18 shows a Meggitt decoder for the (7, 4) code with generator polynomial $g(x) = x^3 + x + 1$. Consider again the codeword $c(x) = x^6 + x^4 + x + 1$ incurring the

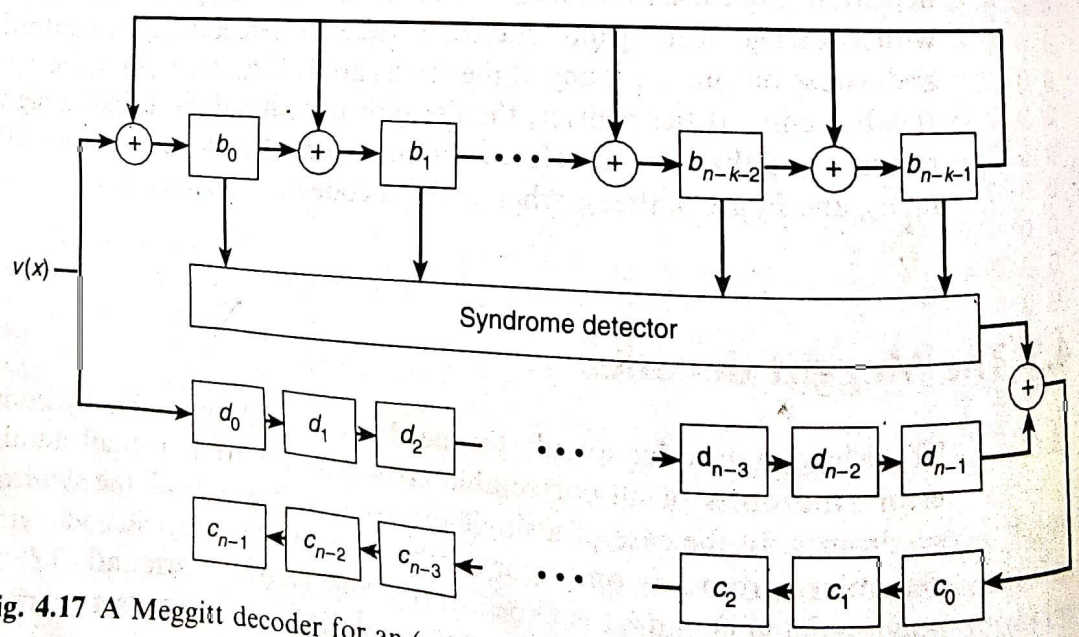


Fig. 4.17 A Meggitt decoder for an (n, k) cyclic code.

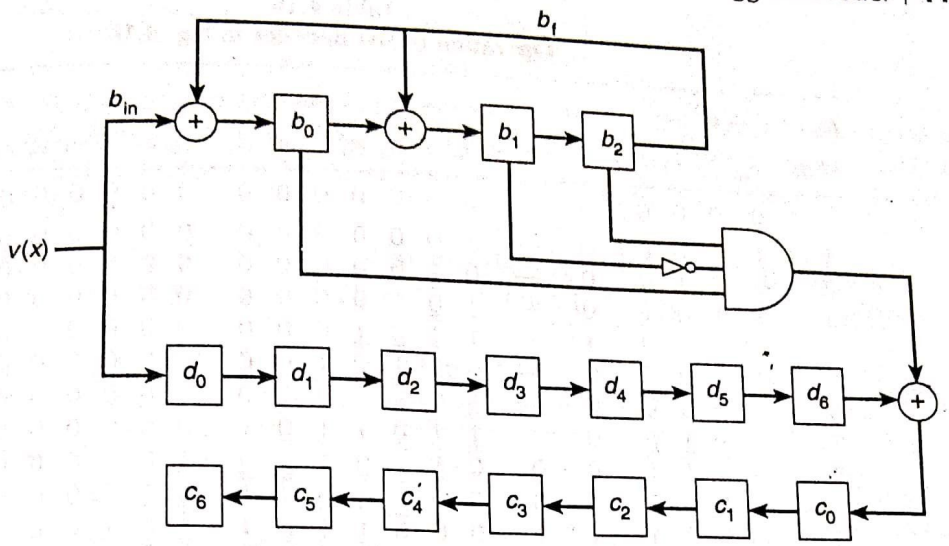


Fig. 4.18 A Meggitt decoder for the (7, 4) cyclic code with $g(x) = x^3 + x + 1$.

error x^3 so giving $v(x) = x^6 + x^4 + x^3 + x + 1$ as the word to be decoded (see Section 4.4 and Table 4.9). The input to the decoder is $(v_6, v_5, v_4, v_3, v_2, v_1, v_0) = (1\,0\,1\,1\,0\,1\,1)$ where v_3 is in error. The detector is arranged to detect the error syndrome of the highest-order bit, so that here we have

$$s_d(x) = R_{g(x)}[x^6]$$

and given the generator polynomial $g(x) = x^3 + x + 1$ this gives

$$s_d(x) = x^2 + 1.$$

Hence the detector needs to detect when $b_0 = 1, b_1 = 0$, and $b_2 = 1$. This is achieved by feeding b_0, b_1, and b_2 into a 3-input AND gate with the b_1 input passing through an inverter. When $b_0 = 1, b_1 = 0$, and $b_2 = 1$ all 3 inputs to the AND gate are 1 and so the gate's output $s_d = 1$. At all other times the gate's output is 0.

The decoder's step-by-step operation is shown in Table 4.10. Fourteen shifts are required for decoding, of which the first 7 are needed to obtain the error syndrome of $v(x)$ and the last 7 for error correction. As before the coefficients of $v(x)$ are underlined to show their progress through the register, and furthermore the erroneous bit is embolded so that its progress can be easily seen. Bits enter the register high-order first, simultaneously feeding into the register and into the stages $d_0, d_1, \ldots, d_6$. By the end of the 7th shift the stages $d_0, d_1, \ldots, d_6$ contain $v_0, v_1, \ldots, v_6$ and $b_0 = 1, b_1 = 1, b_2 = 0$ which give the error syndrome $s(x) = x + 1$. As $s(x) \neq s_d(x)$ the detector output $s_d = 0$ and so v_6 will not be inverted as it leaves d_6 to enter c_0. After 3 more shifts the erroneous bit v_3 occupies d_6 and b_0, b_1, b_2 are 1, 0, 1 respectively, giving the error syndrome $s(x) = x^2 + 1 = s_d(x)$ and $s_d = 1$. Hence at the next shift v_3 is corrected as it is fed from d_6 and into c_0. Finally 3 more shifts are required to move the remaining bits into the codeword stages.

Table 4.10
Operation of the decoder in Fig. 4.18

Input $x^6+x^4+x^3+x+1$																			
Shift	b_{in}	b_0	b_1	b_2	b_f	s_d	d_0	d_1	d_2	d_3	d_4	d_5	d_6	c_0	c_1	c_2	c_3	c_4	c_5	c_6	$s(x)$
	0	0	0	0	0	—	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	—	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	0	0	0	—	1	1	0	0	0	0	0	0	0	0	0	0	0	0	x
2	0	0	1	0	0	—	0	1	1	0	0	0	0	0	0	0	0	0	0	0	x^2+1
3	1	1	0	1	0	—	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	1	—	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	—	0	1	1	0	1	1	0	0	0	0	0	0	0	0	1
6	1	1	0	0	0	—	1	0	1	1	0	1	1	0	0	0	0	0	0	0	$x+1$
7	1	1	1	0	0	0	1	1	0	1	1	0	1	1	0	0	0	0	0	0	x^2+x
8	—	0	1	1	0	0	0	1	1	0	1	1	0	1	1	0	0	0	0	0	x^2+x+1
9	—	1	1	1	1	0	0	0	1	1	0	1	1	1	0	1	0	0	0	0	x^2+1
10	—	1	0	1	1	1	0	0	0	1	1	0	1	1	1	0	1	0	0	0	1
11	—	1	0	0	1	0	0	0	0	0	1	1	0	0	1	1	0	1	0	0	x
12	—	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	1	0	x^2
13	—	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	1	$x+1$
14	—	1	1	0	1	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	

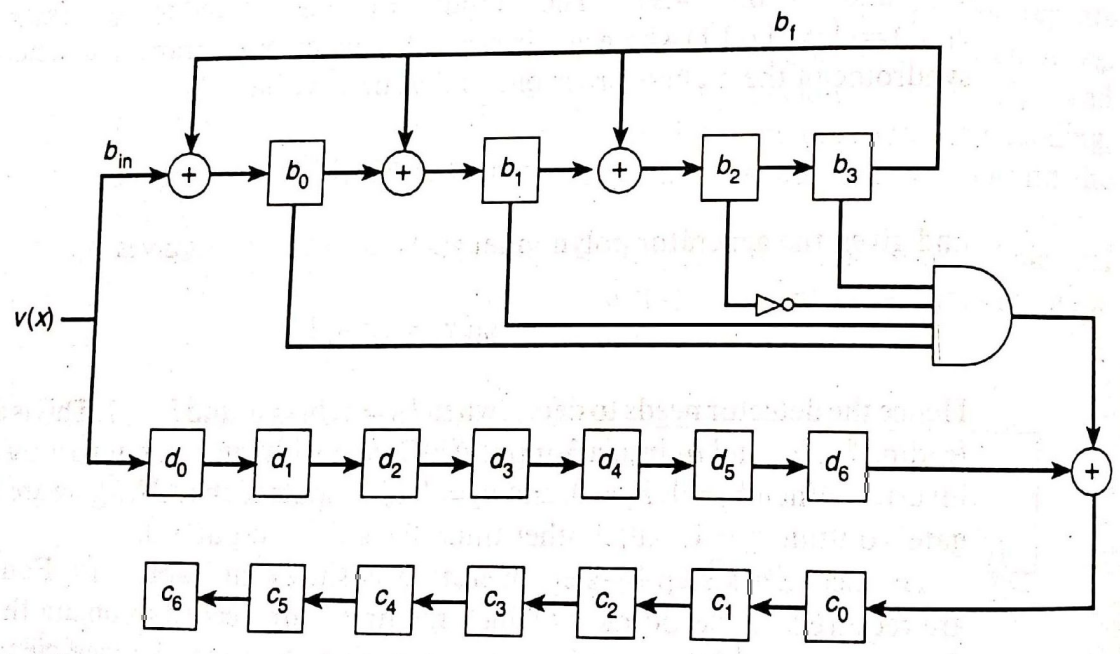


Fig. 4.19 A Meggitt decoder for the (7, 3) cyclic code with $g(x)=x^4+x^2+x+1$.

Example 4.5

Figure 4.19 shows a Meggitt decoder for the single-error correcting (7, 3) code with $g(x)=x^4+x^2+x+1$. Determine the step-by-step operation when $c(x)=x^6+x^5+x^4+x$ incurs the errors (a) $e(x)=1$ and (b) $e(x)=x^6$.

The detector needs to determine when the high-order bit of the input is in error and so it needs to detect when the remainder stages contain the error syndrome of x^6.

This gives

$$s_d(x) = R_{g(x)}[x^6] = x^3 + x + 1$$

as the error syndrome that needs to be detected. Hence the detector needs to detect when $b_0 = 1$, $b_1 = 1$, $b_2 = 0$, and $b_3 = 1$, which can be achieved by feeding b_0, b_1, b_2, and b_3 into a 4-input AND gate with b_2 passing via an inverter (as shown in Fig. 4.19).

(a) When $c(x) = x^6 + x^5 + x^4 + x$ incurs the error $e(x) = 1$ then $v(x) = x^6 + x^5 + x^4 + x + 1$ and the decoder input is (1 1 1 0 0 1 1) as shown in Table 4.11(a).

Table 4.11
Operation of the Meggitt decoder in Fig. 4.19

(a) *Input* $v(x) = x^6 + x^5 + x^4 + x + 1$

Shift	b_{in}	b_0	b_1	b_2	b_3	b_f	s_d	d_0	d_1	d_2	d_3	d_4	d_5	d_6	c_0	c_1	c_2	c_3	c_4	c_5	c_6	$s(x)$
	0	0	0	0	0	0	—	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	—	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
2	1	1	1	0	0	0	—	1	1	0	0	0	0	0	0	0	0	0	0	0	0	$x+1$
3	1	1	1	1	0	0	—	1	1	1	0	0	0	0	0	0	0	0	0	0	0	x^2+x+1
4	0	0	1	1	1	0	—	0	1	1	1	0	0	0	0	0	0	0	0	0	0	x^3+x^2+x
5	0	1	1	0	1	1	—	0	0	1	1	1	0	0	0	0	0	0	0	0	0	x^3+x+1
6	1	0	0	0	0	1	—	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
7	**1**	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1
8	—	0	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	0	0	0	x
9	—	0	0	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	0	0	x^2
10	—	0	0	0	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	0	x^3
11	—	1	1	1	0	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	x^2+x+1
12	—	0	1	1	1	0	0	0	0	0	0	0	1	1	0	0	1	1	1	0	0	x^3+x^2+x
13	—	1	1	0	1	1	1	0	0	0	0	0	0	1	1	0	0	1	1	1	0	x^2+x+1
14	—	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1

The remainder at the 7th shift is $s(x)=1$ which is the error syndrome of $v(x)$. By the 13th shift the erroneous bit occupies the last stage of the delay stages, the remainder stages contain the syndrome $s_d(x)$ and $s_d=1$. Hence at the 14th shift the erroneous bit is corrected as it leaves d_6 and enters c_0.

(b) Here $v(x)=x^5+x^4+x$, the error is detected at the 7th shift and corrected at the 8th as shown in Table 4.11(b). □

We have seen that feeding the input to the high-order side of a polynomial-division register, for an (n, k) code, is equivalent to multiplying the input by x^{n-k}. This is useful when encoding, for multiplication by x^{n-k} is required for generating systematic codewords. At the decoding stage there is no such requirement and once $v(x)$ has entered the register the resulting remainder is $r(x)=R_{g(x)}[x^{n-k}v(x)]$ which is not the error syndrome required. However $r(x)$ still contains information that is solely dependent on errors within $v(x)$. Given a correctable error pattern $e(x)$ there is a unique correspondence between the error pattern and the remainder $r(x)$, and therefore

$$s'(x)=R_{g(x)}[x^{n-k}v(x)] \tag{4.10}$$

is an error syndrome of $v(x)$. The difference between $s'(x)$ and the 'normal' error syndrome $s(x)=R_{g(x)}[v(x)]$ is the correspondence between error syndromes and correctable error patterns. Consider for example the (7, 4) code with $g(x)=x^3+x+1$. The error syndrome $s(x)$ of, say, $e(x)=x^4$ is x^2+x, but using eqn 4.10 gives $s'(x)=1$. Table 4.12 gives the error syndromes $s(x)$ and $s'(x)$ for the (7, 4) code for all the correctable error patterns. From Table 4.12 we can see that the error syndromes $s'(x)$ are the same as $s(x)$ and differ only in their correspondence with the error patterns $e(x)$. Therefore there is no reason why $s'(x)$, instead of $s(x)$, cannot be used as the definition of the error syndrome of $v(x)$.

Consider now a Meggitt decoder for the (7, 4) code with input to the high-order side (see Fig. 4.20). Recall that the detector needs to detect an error in the high-order position x^6, and from Table 4.12 we can see that this means that we need to detect when the error syndrome $s'(x)=x^2$, that is when $b_0=0$, $b_1=0$, and $b_2=1$. A 3-input AND gate can be used for the detector with inverters on the b_0 and b_1 inputs to the gate. Decoding proceeds in the same way as that already described for a Meggitt

Table 4.12
Error syndromes for the (7, 4) code with $g(x)=x^3+x+1$

$e(x)$	$s(x)$	$s'(x)$
1	1	$x+1$
x	x	x^2+x
x^2	x^2	x^2+x+1
x^3	$x+1$	x^2+1
x^4	x^2+x	1
x^5	x^2+x+1	x
x^6	x^2+1	x^2

Note: $s(x)=R_{g(x)}[v(x)]$ and $s'(x)=R_{g(x)}[x^{n-k}v(x)]$

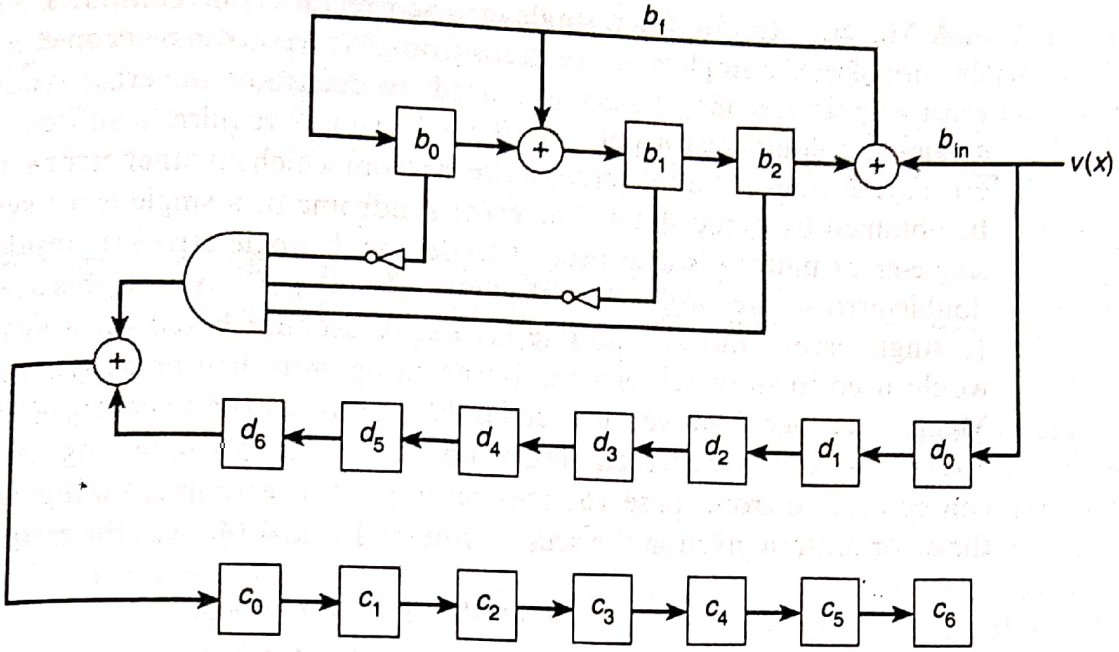


Fig. 4.20 A Meggitt decoder with high-order input for the (7,4) code with $g(x)=x^3+x+1$.

Table 4.13
Operation of the Meggitt decoder in Fig. 4.20

High-order input $v(x)=x^6+x^4+x^3+x+1$

Shift	b_{in}	b_0	b_1	b_2	b_f	s_d	d_0	d_1	d_2	d_3	d_4	d_5	d_6	c_0	c_1	c_2	c_3	c_4	c_5	c_6	$r(x)$
	0	0	0	0	0	—	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	—	1	0	0	0	0	0	0	0	0	0	0	0	0	0	$x+1$
2	0	0	1	1	0	—	0	1	0	0	0	0	0	0	0	0	0	0	0	0	x^2+x
3	1	0	0	1	0	—	1	0	1	0	0	0	0	0	0	0	0	0	0	0	x^2
4	1	0	0	0	0	—	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	—	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0
6	1	1	1	0	1	—	1	0	1	1	0	1	0	0	0	0	0	0	0	0	$x+1$
7	1	1	0	1	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	x^2+1
8	—	1	0	0	1	0	0	1	1	0	1	1	0	1	0	0	0	0	0	0	1
9	—	0	1	0	0	0	0	0	1	1	0	1	1	0	1	0	0	0	0	0	x
10	—	0	0	1	0	1	0	0	0	1	1	0	1	1	0	1	0	0	0	0	x^2
11	—	1	1	0	1	0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	$x+1$
12	—	0	1	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	0	x^2+x
13	—	1	1	1	1	0	0	0	0	0	0	0	1	1	0	0	1	0	1	0	x^2+x+1
14	—	1	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	x^2+x

decoder with input to the low-order side. Table 4.13 shows the operation of the decoder for the input $v(x)=x^6+x^4+x^3+x+1$ (arising from the codeword polynomial $c(x)=x^6+x^4+x+1$ with the error $e(x)=x^3$). Note that 7 shifts are still required to obtain the remainder and 7 to correct the error.

A Meggitt decoder for a single-error-correcting code computes the error syndromes of all the single-error patterns from just one error syndrome. For a double-error-correcting code, the decoder needs to determine the error syndromes of all single- and double-error patterns. To achieve this requires a sufficient number of error syndromes of double-error patterns from which all other error syndromes can be obtained by cyclic shifts. The error syndrome of a single-error syndrome of a single-error pattern is also needed to deal with single errors. Consider the (15, 7) double-error-correcting code with $g(x) = x^8 + x^7 + x^6 + x^4 + 1$, this needs to correct 15 single errors and 105 double errors. A decoder based on a syndrome table would need to store 120 error patterns along with their error syndromes. With a Meggitt decoder, however, we need only to consider the 15 error patterns shown in Table 4.14. All other correctable error patterns and corresponding error syndromes can be derived from these 15 error patterns. For example, taking cyclic shifts of the error pattern given in the second row of Table 4.14 gives the error patterns

$$
\begin{array}{c}
(0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,0) \\
(0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,0\,0) \\
(0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,0\,0\,0) \\
\vdots \qquad\qquad\qquad\qquad \vdots \\
(0\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0) \\
(1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0) \\
(1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1)
\end{array}
$$

which are all the different patterns with two successive bits in error. Each error pattern in Table 4.14 requires its own detection circuit and all error patterns are checked as decoding progresses. The decoder still operates in the manner described

Table 4.14
Syndrome table of a Meggitt decoder for the (15, 7) double-error-correcting code

e	s
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)	(0 0 0 0 0 0 0 1)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1)	(0 0 0 0 0 0 1 1)
(0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1)	(0 0 0 0 0 1 0 1)
(0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1)	(0 0 0 0 1 0 0 1)
(0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1)	(0 0 0 1 0 0 0 1)
(0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1)	(0 0 1 0 0 0 0 1)
(0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1)	(0 1 0 0 0 0 0 1)
(0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1)	(1 0 0 0 0 0 0 1)
(0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1)	(1 1 0 1 0 0 0 0)
(0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1)	(0 1 1 1 0 0 1 0)
(0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1)	(1 1 1 0 0 1 1 1)
(0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1)	(0 0 0 1 1 1 0 0)
(0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1)	(0 0 1 1 1 0 1 1)
(0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1)	(0 1 1 1 0 1 0 1)
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)	(1 1 1 0 1 0 0 1)

for a single-error-correcting code and the same number of shifts $2n$ are required. Table 4.14 can be simplified by taking into account error patterns that can be obtained from cyclic shifts of other patterns. An example of this is the first double error

$$(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1)$$

which when shifted by one bit towards the right gives

$$(1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1)$$

the last entry in the table. Hence either, but not both, of these patterns can be omitted from the table. The 14 double-error patterns can be arranged into 7 pairs, where within each pair the 2 error patterns differ only by a cyclic shift. By taking this into account the Meggitt decoder needs only to detect 8 error syndromes, namely the single-error syndrome and one syndrome for each of the 7 pairs of double-error patterns. The resulting Meggitt decoder requires a simpler detector but an additional n shifts are needed to ensure that error patterns that have been excluded from the table are decoded.

Problems

4.1 Figure 4.21 shows a linear-feedback shift register with 8 stages. Given that the register is initialized with $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (0\ 0\ 1\ 1\ 0\ 1\ 0\ 1)$ determine the state of the register after:
(a) 5 shifts to the left;
(b) a further 4 shifts to the left;
(c) 18 shifts to the right, from the initial state not from the state after (a) or (b).

4.2 Figure 4.22 shows a linear-feedback shift register with high-order input. Determine the contents of the register for the input (1 1 0 0 1 0).

4.3 A linear-feedback shift register has

$$b_f = b_3$$
$$b_3 = b_2 + b_f$$
$$b_2 = b_1$$
$$b_1 = b_0 + b_f$$
$$b_0 = b_{in} + b_f.$$

b_f

Using $\sigma = \boldsymbol{M}^{-1}\boldsymbol{S}$ gives

$$\begin{bmatrix}\sigma_3\\ \sigma_2\\ \sigma_1\end{bmatrix} = \begin{bmatrix}\alpha^9 & \alpha^{13} & \alpha^{10}\\ \alpha^{13} & \alpha^{12} & \alpha^{12}\\ \alpha^{10} & \alpha^{12} & 1\end{bmatrix}\begin{bmatrix}\alpha^3\\ \alpha^5\\ \alpha^5\end{bmatrix} = \begin{bmatrix}\alpha^5\\ \alpha\\ \alpha^{12}\end{bmatrix}$$

and so $\sigma_3 = \alpha^5$, $\sigma_2 = \alpha$, $\sigma_1 = \alpha^{12}$ giving the error-location polynomial

$$\sigma(x) = 1 + \alpha^{12}x + \alpha x^2 + \alpha^5 x^3.$$

Searching $GF(2^4)$ for the roots of $\sigma(x)$ gives α^2, α^{10}, and α^{13} as roots and taking their reciprocals gives α^{13}, α^5, and α^2 as the error-location numbers respectively. The error pattern is therefore $e(x) = x^{13} + x^5 + x^2$ and the codeword polynomial $c(x) = v(x) + e(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$. □

The Peterson–Gorenstein–Zierler decoder forms the basis of decoding algorithms for BCH codes. It is a relatively simple decoder, as the reader should find after working through a few examples. The matrix inversion does however present a problem to the decoder. For a large error-correction limit t, evaluating the resulting determinants can be computationally slow and inefficient. Furthermore, as we shall see later, a second matrix inversion is required when dealing with non-binary codes, and so aggravating the problem. To develop fast decoding algorithms we need to avoid the matrix inversions, this is considered in Sections 7.8 and 7.9.

7.7 Reed–Solomon codes

The codes considered so far have all been binary codes, and we now turn our attention to non-binary codes. At first the idea of a non-binary code may seem rather strange or of little practical use. Information processing, transmission, and storage is usually thought of in terms of a binary representation. Bits are manipulated either individually or in blocks of convenient length, for example as 8-bit words. However, an 8-bit word can be thought of as a single *non-binary symbol* with 256 different values, irrespective of its underlying structure (i.e. the fact that it is really a collection of 8 bits and not a single symbol). Likewise any sequence of r bits can be viewed as a single non-binary symbol that has one of 2^r values. Furthermore symbols need not necessarily be restricted to 2^r values but can be defined for any positive integer.

Non-binary codes are concerned with the detection and correction of errors in symbols. The construction of non-binary codes, along with encoding and decoding techniques, follows directly from that of binary codes. The main difference arises in the need to determine the magnitude of errors and not just the error locations. In binary codes error magnitudes are 1 and it is only necessary to determine the position of errors. Once located error correction is achieved by simply inverting the

erroneous bits. With a non-binary code we first locate the position of the errors and then determine the magnitude of the errors.

Binary codes can be viewed as codes whose symbols have 2 values, 0 and 1, that is the code's symbols lie in $GF(2)$. A non-binary code has its symbols in the field $GF(q)$ where q is a prime number or any power of a prime number. A non-binary (n, k) linear code will have codewords of the form $\boldsymbol{c} = (c_{n-1}, c_{n-2}, \ldots, c_2, c_1, c_0)$ where the codeword components lie in $GF(q)$ and there exists at least one set of k codewords from which all the other codewords can be obtained by linear combinations of the k codewords. A non-binary (n, k) cyclic code can be constructed from a polynomial $g(x)$ of degree $n-k$, where $g(x)$ has its coefficients in $GF(q)$ and divides x^n-1. Note that for binary codes $x^n - 1 = x^n + 1$ and so $g(x)$ can be said to divide the latter if it is to generate a cyclic code.

A t-error-correcting nonbinary BCH code of blocklength $n = q^m - 1$ is a (n, k) cyclic code whose generator polynomial $g(x)$ has its coefficients in $GF(q)$ and roots

$$\beta, \beta^2, \ldots, \beta^{2t}$$

in $GF(q^m)$ an extension field of $GF(q)$. Recall that the generator polynomial of a t-error-correcting binary BCH code is given by the least common multiple LCM of the minimal polynomials $m_i(x)$, over $GF(2)$, of $2t$ consecutive field elements. To construct a non-binary BCH code minimal polynomials over $GF(q)$ are required and the generator polynomial of the code is given by

$$g(x) = \text{LCM}[m_1(x), m_2(x), \ldots, m_{2t}(x)]$$

where now $m_i(x)$ is the minimal polynomial over $GF(q)$ of β^i. If $q=2$ then the minimal polynomials are binary and we obtain the binary BCH codes.

The most important class of non-binary BCH codes are the *Reed–Solomon codes*, which differ from other non-binary codes in that the base field and extension field are taken to be the same. Both the symbols and the generator polynomial roots lie in the field $GF(q)$ and define a Reed–Solomon code with blocklength $n = q - 1$. Here we consider Reed–Solomon codes where $q = 2^m$ and so symbols and roots lie in $GF(2^m)$. A t-error-correcting Reed–Solomon code is a cyclic code whose generator polynomial is the least-degree polynomial that has $\beta, \beta^2, \beta^3, \ldots, \beta^{2t}$ as roots, where β belongs to $GF(2^m)$. The minimal polynomial over $GF(2^m)$ of an element β in $GF(2^m)$ is the factor

$$m_\beta = x + \beta$$

as this is clearly the least-degree polynomial that has β as a root. The generator polynomial of a Reed–Solomon code is therefore

$$g(x) = (x+\beta)(x+\beta^2)(x+\beta^3)\cdots(x+\beta^{2t}). \tag{7.35}$$

Note that there is also no need to take the least common multiple of the factors as all the factors are distinct.

Consider a double-error-correcting Reed–Solomon code over $GF(2^4)$, taking $\beta = \alpha$ gives

$$g(x) = (x+\alpha)(x+\alpha^2)(x+\alpha^3)(x+\alpha^4)$$

and expanding this gives the generator polynomial

$$g(x) = x^4 + \alpha^{13}x^3 + \alpha^6 x^2 + \alpha^3 x + \alpha^{10}. \tag{7.36}$$

Note that the coefficients of $g(x)$ are no longer binary. The code's blocklength is $n = 2^4 - 1 = 15$, and as the degree of $g(x)$ is $r = 4$, then using $n - k = r$ gives $k = 11$ (recall that the degree of the generator polynomial of an (n, k) cyclic code is $n - k$). This is therefore the generator polynomial of a double-error-correcting (15, 11) Reed–Solomon code.

Example 7.7

Construct a single-error-correcting Reed–Solomon code with blocklength 7.

The code is constructed over $GF(2^3)$ as this gives a code with blocklength $n = 2^3 - 1 = 7$. Substituting $t = 1$ and $\beta = \alpha$ in eqn 7.35 gives

$$g(x) = (x+\alpha)(x+\alpha^2) = x^2 + \alpha^4 x + \alpha^3$$

where α is a primitive element in $GF(2^3)$. The information length k is given by $k = n - r$, where $r = 2$ is the degree of $g(x)$ and $n = 7$, and so $k = 5$. This is therefore a single-error-correcting (7, 5) Reed–Solomon code. □

For encoding purposes the Reed–Solomon codes can be treated as cyclic codes or a generator matrix can be constructed from the generator polynomial and the codes can then be treated as linear codes. For example consider the (7, 5) Reed–Solomon code with generator polynomial

$$g(x) = x^2 + \alpha^4 x + \alpha^3 \tag{7.37}$$

and let's construct the systematic codeword for the information word, say, $\boldsymbol{i} = (1\ 0\ \alpha\ \alpha^5\ \alpha^2)$ where α is an element of $GF(2^3)$. The information polynomial corresponding to $\boldsymbol{i}$ is

$$i(x) = x^4 + \alpha x^2 + \alpha^5 x + \alpha^2$$

and multiplying this by $x^{n-k} = x^2$ gives

$$x^2 i(x) = x^6 + \alpha x^4 + \alpha^5 x^3 + \alpha^2 x^2.$$

Recall that to construct systematic codewords we require the remainder of $x^{n-k}i(x)$ divided by $g(x)$. When dividing two non-binary polynomials, care has to be taken to ensure that at each step the coefficients of the highest power of x are the same. The division is a bit more awkward than that of dividing two binary polynomials,

however the principle is the same. Dividing $x^2 i(x)$ by $g(x)$ we get

$$
\begin{array}{r}
x^4 + \alpha^4 x^3 + \alpha^3 x^2 + \alpha^5 x + \alpha^6 \\
x^2 + \alpha^4 x + \alpha^3 \;\overline{)\; x^6 + \alpha x^4 + \alpha^5 x^3 + \alpha^2 x^2} \\
\underline{x^6 + \alpha^4 x^5 + \alpha^3 x^4} \\
\alpha^4 x^5 + x^4 + \alpha^5 x^3 + \alpha^2 x^2 \\
\underline{\alpha^4 x^5 + \alpha x^4 + x^3} \\
\alpha^3 x^4 + \alpha^4 x^3 + \alpha^2 x^2 \\
\underline{\alpha^3 x^4 + x^3 + \alpha^6 x^2} \\
\alpha^5 x^3 + x^2 \\
\underline{\alpha^5 x^3 + \alpha^2 x^2 + \alpha x} \\
\alpha^6 x^2 + \alpha x \\
\underline{\alpha^6 x^2 + \alpha^3 x + \alpha^2} \\
x + \alpha^2
\end{array}
$$

and the remainder is therefore $r(x) = x + \alpha^2$. Adding $r(x)$ to $x^2 i(x)$ gives the codeword polynomial

$$c(x) = x^2 i(x) + r(x) = x^6 + \alpha x^4 + \alpha^5 x^3 + \alpha^2 x^2 + x + \alpha^2$$

which gives the codeword $\boldsymbol{c} = (1\ 0\ \alpha\ \alpha^5\ \alpha^2\ 1\ \alpha^2)$.

Example 7.8

Construct the (15, 13) single-error-correcting Reed–Solomon code and determine the systematic codeword corresponding to $\boldsymbol{i} = (0\ 0\ \alpha\ 0\ 0\ 1\ \alpha^7\ \alpha^2\ 0\ 0\ 1\ \alpha\ \alpha^2)$ where α is a primitive element of $GF(2^4)$.

The generator polynomial is

$$g(x) = (x + \alpha)(x + \alpha^2) = x^2 + \alpha^5 x + \alpha^3.$$

Note that the degree of $g(x)$ is 2, which is consistent with the code's (n, k) parameters, $n - k = 15 - 13 = 2$. The information polynomial corresponding to $\boldsymbol{i}$ is

$$i(x) = \alpha x^{10} + x^7 + \alpha^7 x^6 + \alpha^2 x^5 + x^2 + \alpha x + \alpha^2$$

and dividing $x^2 i(x)$ by $g(x)$ gives the quotient and remainder

$$
\begin{aligned}
q(x) &= \alpha x^{10} + \alpha^6 x^9 + \alpha^{13} x^8 + \alpha^4 x^7 + \alpha^4 x^6 + \alpha^8 x^5 + \alpha^5 x^4 \\
&\quad + \alpha^{14} x^3 + \alpha^{10} x^2 + \alpha^{10} x + \alpha^3 \\
r(x) &= x\alpha^3 + \alpha^6
\end{aligned}
$$

respectively. The codeword polynomial is therefore

$$x^2 i(x) + r(x) = \alpha x^{12} + x^9 + \alpha^7 x^8 + \alpha^2 x^7 + x^4 + \alpha x^3 + \alpha^2 x^2 + \alpha^3 x + \alpha^6$$

which gives the codeword

$$\boldsymbol{c} = (0\ 0\ \alpha\ 0\ 0\ 1\ \alpha^7\ \alpha^2\ 0\ 0\ 1\ \alpha\ \alpha^2\ \alpha^3\ \alpha^6).$$

□

The generator polynomial $g(x)$ of a t-error-correcting Reed–Solomon code has $2t$ linear factors, one for each root $\beta, \beta^2, \ldots, \beta^{2t}$ and the degree of $g(x)$ is therefore $n-k=2t$. Hence the number of parity-check symbols is $2t$ and as such a t-error correcting Reed–Solomon code with blocklength n can be referred to as a $(n, n-2t)$ Reed–Solomon code. Recall that a t-error-correcting code requires a minimum distance of

$$d_{\min} = 2t + 1$$

and so a t-error-correcting Reed–Solomon code has

$$d_{\min} = n - k + 1$$

and therefore the Reed–Solomon codes are maximum-distance codes. Note also that the designed distance, d_0, and minimum distance, $d_{\min}$, of a Reed–Solomon code are the same.

The number of codewords in a non-binary code can be surprisingly large. An (n, k) binary code has 2^k codewords as there are 2^k distinct information words. For example the $(7, 4)$ binary code has 16 codewords. In a t-error-correcting $(n, n-2t)$ Reed–Solomon code over $GF(q)$ each information symbol has q distinct values and there are therefore q^{n-2t} codewords. For example each symbol of the single-error-correcting $(7, 5)$ Reed–Solomon code has 8 distinct values and the code has 32 768 codewords. Clearly decoding algorithms that avoid the use of look-up tables are necessary with codes with such large numbers of codewords.

Decoding Reed–Solomon codes is achieved by first determining the error positions and then the error magnitudes. The methods used for locating errors in binary codes can also be used in Reed–Solomon codes, the only additional theory required is for finding error magnitudes. In a binary code an error pattern of μ errors can be represented by the error polynomial

$$e(x) = x^{p_1} + x^{p_2} + \cdots + x^{p_\mu}$$

where $p_1, p_2, \ldots, p_\mu$ are the error positions. Taking error magnitudes into account, the error polynomial becomes

$$e(x) = y_{p_1} x^{p_1} + y_{p_2} x^{p_2} + \cdots + y_{p_\mu} x^{p_\mu} \tag{7.38}$$

where y_{p_i} is the error magnitude at the position p_i. The decoder input is $v(x) = c(x) + e(x)$ where $c(x)$ is the codeword polynomial incurring the errors. For a t-error correcting code the error syndromes calculated by the decoder are

$$S_i = v(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i)$$

so giving

$$\begin{aligned}
S_1 &= y_{p_1}\alpha^{p_1} + y_{p_2}\alpha^{p_2} + \cdots + y_{p_\mu}\alpha^{p_\mu} \\
S_2 &= y_{p_1}\alpha^{2p_1} + y_{p_2}\alpha^{2p_2} + \cdots + y_{p_\mu}\alpha^{2p_\mu} \\
S_3 &= y_{p_1}\alpha^{3p_1} + y_{p_2}\alpha^{3p_2} + \cdots + y_{p_\mu}\alpha^{3p_\mu} \\
&\vdots \\
S_{2t} &= y_{p_1}\alpha^{2tp_1} + y_{p_2}\alpha^{2tp_2} + \cdots + y_{p_\mu}\alpha^{2tp_\mu}.
\end{aligned} \tag{7.39}$$

Recall that for binary codes we defined the error-location number $X_i = \alpha^{p_i}$. Here we define an additional term $Y_i = y_{p_i}$ known as the *error magnitude* of the ith error-location number, and in doing so we can express the syndrome equations in the more convenient form

$$\begin{aligned} S_1 &= Y_1X_1 + Y_2X_2 + \cdots + Y_\mu X_\mu \\ S_2 &= Y_1X_1^2 + Y_2X_2^2 + \cdots + Y_\mu X_\mu^2 \\ S_3 &= Y_1X_1^3 + Y_2X_2^3 + \cdots + Y_\mu X_\mu^3 \\ &\vdots \\ S_{2t} &= Y_1X_1^{2t} + Y_2X_2^{2t} + \cdots + Y_\mu X_\mu^{2t}. \end{aligned} \tag{7.40}$$

We have already seen that for a binary code $S_2 = S_1^2$, this though does not apply to non-binary codes. Taking S_1^2 gives

$$\begin{aligned} S_1^2 &= (Y_1X_1 + Y_2X_2 + \cdots + Y_\mu X_\mu)^2 \\ &= Y_1^2X_1^2 + Y_2^2X_2^2 + \cdots + Y_\mu^2X_\mu^2 \neq S_2. \end{aligned}$$

Likewise we can show that $S_4 \neq S_2^2$ and clearly for a non-binary code

$$S_{2i} \neq S_i^2$$

and so the $2t$ error syndromes need to be individually evaluated.

Equations 7.40 consist of $2t$ equations with μ unknown error magnitudes, along with known error syndromes and error-location numbers, which can be solved to give the error magnitudes. Before addressing eqns 7.40 for any value of t we first consider a single-error-correcting code. The decoder of a single-error correcting code determines two syndromes S_1 and S_2. From eqn 7.40 setting $t = 1$ and $\mu = 1$ (as the maximum number of correctable errors is 1) gives

$$\begin{aligned} S_1 &= Y_1X_1 \\ S_2 &= Y_1X_1^2 \end{aligned} \tag{7.41}$$

and dividing S_2 by S_1 gives

$$\frac{S_2}{S_1} = \frac{Y_1X_1^2}{Y_1X_1} = X_1.$$

Substituting $X_1 = S_2/S_1$ into the first expression in eqns 7.41 gives $S_1 = Y_1(S_2/S_1)$ and so $Y_1 = S_1^2/S_2$. Therefore the error-location number X_1 and error magnitude Y_1 of a single-error correcting Reed–Solomon code are given by

$$\begin{aligned} X_1 &= S_2/S_1 \\ Y_1 &= S_1^2/S_2. \end{aligned} \tag{7.42}$$

Note that if we let $S_2 = S_1^2$, then eqns 7.42 give

$$\begin{aligned} X_1 &= S_1^2/S_1 = S_1 \\ Y_1 &= S_1^2/S_1^2 = 1 \end{aligned}$$

which are the correct error-location number and error magnitude for a single-error-correcting binary code.

Example 7.9

Consider the (7, 5) single-error-correcting Reed–Solomon code. Given that $v = (0\ 1\ \alpha^5\ \alpha^2\ 1\ \alpha^6\ \alpha^3)$, where α is an element of $GF(2^3)$, corresponds to a codeword c with a single error, determine the position and magnitude of the error and the codeword c.

The polynomial corresponding to c is

$$v(x) = x^5 + \alpha^5 x^4 + \alpha^2 x^3 + x^2 + \alpha^6 x + \alpha^3$$

and in $GF(2^3)$ the error syndromes are

$$S_1 = v(\alpha) = \alpha$$
$$S_2 = v(\alpha^2) = \alpha^3.$$

Using eqns 7.42 gives

$$X_1 = S_2/S_1 = \alpha^3/\alpha = \alpha^2$$
$$Y_1 = S_1^2/S_2 = \alpha^2/\alpha^3 = \alpha^6$$

giving an error location of x^2 and error magnitude α^6. The error incurred by c is therefore $\alpha^6 x^2$ and so the codeword polynomial is

$$\begin{aligned} c(x) &= v(x) + \alpha^6 x^2 \\ &= x^5 + \alpha^5 x^4 + \alpha^2 x^3 + (1 + \alpha^6)x^2 + \alpha^6 x + \alpha^3 \\ &= x^5 + \alpha^5 x^4 + \alpha^2 x^3 + \alpha^2 x^2 + \alpha^6 x + \alpha^3 \end{aligned}$$

giving $c = (0\ 1\ \alpha^5\ \alpha^2\ \alpha^2\ \alpha^6\ \alpha^3)$. □

A decoder for a t-error-correcting Reed–Solomon code determines the number of errors μ and the error-location numbers $X_1, X_2, \ldots, X_\mu$ using any technique that can be used for a binary BCH code. Once the error-location numbers have been found, the μ error magnitudes can be obtained by solving the first μ equations in eqns 7.40 for $Y_1, Y_2, \ldots, Y_\mu$. Note that the error syndromes, given by eqns 7.40, are nonlinear functions of the error-location numbers, but linear functions of the error magnitudes. Hence the error magnitudes can be obtained from the syndrome equations by using a standard matrix inversion method. Defining the column vectors $\boldsymbol{S}$ and $\boldsymbol{Y}$ as

$$\boldsymbol{S} = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \\ S_\mu \end{bmatrix} \tag{7.43}$$

$$\boldsymbol{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_\mu \end{bmatrix} \tag{7.44}$$

and the matrix X as

$$X = \begin{bmatrix} X_1 & X_2 & X_3 & \dots & X_\mu \\ X_1^2 & X_2^2 & x_3^2 & \dots & X_\mu^2 \\ X_1^3 & X_2^3 & X_3^3 & \dots & X_\mu^3 \\ \vdots & & & & \vdots \\ X_1^\mu & X_2^\mu & X_3^\mu & \dots & X_\mu^\mu \end{bmatrix} \tag{7.45}$$

then eqns 7.40 can be written as

$$S = XY.$$

Note that the column vector S given by eqn 7.43 is not the same as S (eqn 7.32) defined for use in the Peterson–Gorenstein–Zierler decoder, for convenience the same notation is used, this should not cause confusion. The matrix X and column vector S are known terms, and so Y can be found by inverting $S = XY$ to give $Y = X^{-1}S$. Therefore the error magnitudes are given by

$$Y = X^{-1}S. \tag{7.46}$$

Note that X cannot be singular because μ nonzero and distinct errors are already known to exist.

Finding error magnitudes is simpler than we may have at first expected. The nonlinear problem faced when determining the error locations does not arise. Instead the error magnitudes are linearly related to the error syndromes and the known error-location numbers. However, as before, we face the computationally inefficient process of matrix inversion. Later we shall see how this matrix inversion can be circumvented (see Section 7.9). Decoding Reed–Solomon codes can be summarized as follows:

1. Find the number of errors μ and error-location numbers $X_1, X_2, \dots, X_\mu$ by using any technique suitable to binary BCH codes.
2. From the error-location numbers construct the matrix X and determine its inverse X^{-1}.
3. The error magnitudes $Y_1, Y_2, \dots, Y_\mu$ are then given by $Y = X^{-1}S$, where S is the column vector constructed from the error syndromes $S_1, S_2, S_3, \dots, S_\mu$.

The example that follows considers decoding a (15, 9) triple-error-correcting Reed–Solomon code. The approach used is based on the Peterson–Gorenstein–Zierler decoder with two matrix inversions, one for the error-location numbers and the other for the error magnitudes. Consider a codeword polynomial $c(x)$, belonging to the triple-error-correcting Reed–Solomon (15, 9) code, that has incurred 3 errors so giving

$$v(x) = \alpha^3 x^{12} + x^8 + \alpha^{10} x^7 + \alpha^2 x^5 + \alpha^8 x^4 + \alpha^{14} x^3 + \alpha^6.$$

Over $GF(2^4)$ the error syndromes are

$$S_1 = v(\alpha) = \alpha^{15} + \alpha^8 + \alpha^{17} + \alpha^7 + \alpha^{12} + \alpha^{17} + \alpha^6 = \alpha^6$$
$$S_2 = v(\alpha^2) = \alpha^{27} + \alpha^{16} + \alpha^{24} + \alpha^{12} + \alpha^{16} + \alpha^{20} + \alpha^6 = 0$$
$$S_3 = v(\alpha^3) = \alpha^{39} + \alpha^{24} + \alpha^{31} + \alpha^{17} + \alpha^{20} + \alpha^{23} + \alpha^6 = \alpha^{14}$$
$$S_4 = v(\alpha^4) = \alpha^{51} + \alpha^{32} + \alpha^{38} + \alpha^{22} + \alpha^{24} + \alpha^{26} + \alpha^6 = \alpha^{11}$$
$$S_5 = v(\alpha^5) = \alpha^{63} + \alpha^{40} + \alpha^{45} + \alpha^{27} + \alpha^{28} + \alpha^{29} + \alpha^6 = \alpha^{14}$$
$$S_6 = v(\alpha^6) = \alpha^{75} + \alpha^{48} + \alpha^{52} + \alpha^{32} + \alpha^{32} + \alpha^{32} + \alpha^6 = \alpha^9.$$

The decoder first assumes that the maximum number of correctable errors, 3, have occurred and constructs the matrix

$$\boldsymbol{M} = \begin{bmatrix} S_1 & S_2 & S_3 \\ S_2 & S_3 & S_4 \\ S_3 & S_4 & S_5 \end{bmatrix} = \begin{bmatrix} \alpha^6 & 0 & \alpha^{14} \\ 0 & \alpha^{14} & \alpha^{11} \\ \alpha^{14} & \alpha^{11} & \alpha^{14} \end{bmatrix}.$$

Evaluating the determinate of $\boldsymbol{M}$ gives

$$\det(\boldsymbol{M}) = \alpha^6 \begin{vmatrix} \alpha^{14} & \alpha^{11} \\ \alpha^{11} & \alpha^{14} \end{vmatrix} + 0 \begin{vmatrix} 0 & \alpha^{11} \\ \alpha^{14} & \alpha^{14} \end{vmatrix} + \alpha^{14} \begin{vmatrix} 0 & \alpha^{14} \\ \alpha^{14} & \alpha^{11} \end{vmatrix}$$
$$= \alpha^6(\alpha^{13} + \alpha^7) + \alpha^{14}(\alpha^{14}\alpha^{14}) = \alpha^{11} + \alpha^{12} = 1$$

Hence $\det(\boldsymbol{M}) \neq 0$ and so the decoder assumes that 3 errors have occurred (which we know is correct). The inverse of $\boldsymbol{M}$ is

$$\boldsymbol{M}^{-1} = \frac{\text{adj}(\boldsymbol{M})}{\det(\boldsymbol{M})} = \begin{bmatrix} \alpha^5 & \alpha^{10} & \alpha^{13} \\ \alpha^{10} & \alpha^7 & \alpha^2 \\ \alpha^{13} & \alpha^2 & \alpha^5 \end{bmatrix}.$$

The coefficients of the error-location polynomial are given by

$$\begin{bmatrix} \sigma_3 \\ \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^5 & \alpha^{10} & \alpha^{13} \\ \alpha^{10} & \alpha^7 & \alpha^2 \\ \alpha^{13} & \alpha^2 & \alpha^5 \end{bmatrix} \begin{bmatrix} \alpha^{11} \\ \alpha^{14} \\ \alpha^9 \end{bmatrix} = \begin{bmatrix} \alpha^{16} + \alpha^{24} + \alpha^{22} \\ \alpha^{21} + \alpha^{21} + \alpha^{11} \\ \alpha^{24} + \alpha^{16} + \alpha^{14} \end{bmatrix} = \begin{bmatrix} \alpha^4 \\ \alpha^{11} \\ 1 \end{bmatrix}$$

and so $\sigma_1 = 1$, $\sigma_2 = \alpha^{11}$ and $\sigma_3 = \alpha^4$ giving the error-location polynomial

$$\sigma(x) = 1 + \sigma_1 x + \sigma_2 x^2 + \sigma_3 x^3 = 1 + x + \alpha^{11}x^2 + \alpha^4 x^3.$$

Searching $GF(2^4)$ for the roots of $\sigma(x)$ shows that $x = \alpha^3$, α^9, and α^{14} are roots, and taking the reciprocals of the roots gives the error-location numbers $X_1 = \alpha^{12}$,

$X_2 = \alpha^6$, and $X_3 = \alpha$ respectively. To find the error magnitudes we construct

$$\boldsymbol{X} = \begin{bmatrix} X_1 & X_2 & X_3 \\ X_1^2 & X_2^2 & X_3^2 \\ X_1^3 & X_2^3 & X_3^3 \end{bmatrix} = \begin{bmatrix} \alpha^{12} & \alpha^6 & \alpha \\ \alpha^9 & \alpha^{12} & \alpha^2 \\ \alpha^6 & \alpha^3 & \alpha^3 \end{bmatrix}.$$

The determinant of $\boldsymbol{X}$ is $\det(\boldsymbol{X}) = \alpha^2$ and its inverse is

$$\boldsymbol{X}^{-1} = \begin{bmatrix} \alpha^8 & \alpha^{12} & \alpha \\ \alpha^7 & \alpha^7 & \alpha^9 \\ \alpha^8 & \alpha^9 & \alpha^5 \end{bmatrix}.$$

Using $\boldsymbol{Y} = \boldsymbol{X}^{-1}\boldsymbol{S}$ the error magnitudes Y_1, Y_2, and Y_3 are given by

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} \alpha^8 & \alpha^{12} & \alpha \\ \alpha^7 & \alpha^7 & \alpha^9 \\ \alpha^8 & \alpha^9 & \alpha^5 \end{bmatrix} \begin{bmatrix} \alpha^6 \\ 0 \\ \alpha^{14} \end{bmatrix} = \begin{bmatrix} \alpha^{14} + \alpha^{15} \\ \alpha^{13} + \alpha^{23} \\ \alpha^{14} + \alpha^{19} \end{bmatrix} = \begin{bmatrix} \alpha^3 \\ \alpha^3 \\ \alpha^9 \end{bmatrix}$$

and so $Y_1 = \alpha^3$, $Y_2 = \alpha^3$, and $Y_3 = \alpha^9$. The error-location numbers $X_1 = \alpha^{12}$, $X_2 = \alpha^6$, and $X_3 = \alpha$ correspond to errors in positions x^{12}, x^6, and x respectively, the error pattern is therefore

$$e(x) = \alpha^3 x^{12} + \alpha^3 x^6 + \alpha^9 x$$

and adding this to $v(x)$ gives the codeword polynomial

$$c(x) = x^8 + \alpha^{10}x^7 + \alpha^3 x^6 + \alpha^2 x^5 + \alpha^8 x^4 + \alpha^{14} x^3 + \alpha^9 x + \alpha^6.$$

Example 7.10

A codeword $c(x)$ belonging to the triple-error-correcting (15, 9) Reed–Solomon code incurs errors so giving

$$v(x) = x^{10} + \alpha^3 x^8 + \alpha^{11} x^7 + \alpha^8 x^6 + \alpha^6 x^5 + \alpha^4 x^4 + \alpha^5 x^2 + \alpha^9 x + \alpha^6$$

determine $c(x)$.

The error syndromes corresponding to $v(x)$ are

$$S_1 = \alpha^4, S_2 = 1, S_3 = \alpha^{10}, S_4 = \alpha^7, S_5 = 0, S_6 = \alpha^{14}$$

over $GF(2^4)$. Taking $\mu = 3$ and substituting the error syndromes into eqn 7.31 gives

$$\boldsymbol{M} = \begin{bmatrix} S_1 & S_2 & S_3 \\ S_2 & S_3 & S_4 \\ S_3 & S_4 & S_5 \end{bmatrix} = \begin{bmatrix} \alpha^4 & 1 & \alpha^{10} \\ 1 & \alpha^{10} & \alpha^7 \\ \alpha^{10} & \alpha^7 & 0 \end{bmatrix}$$

the inverse of which is found to be

$$\boldsymbol{M}^{-1} = \begin{bmatrix} 1 & \alpha^3 & \alpha^{14} \\ \alpha^3 & \alpha^6 & 1 \\ \alpha^{14} & 1 & \alpha^4 \end{bmatrix}.$$

$$\boldsymbol{S} = \begin{bmatrix} S_4 \\ S_5 \\ S_6 \end{bmatrix} = \begin{bmatrix} \alpha^7 \\ 0 \\ \alpha^{14} \end{bmatrix}$$

and substituting $\boldsymbol{M}^{-1}$ and $\boldsymbol{S}$ into eqn 7.34 gives the coefficients of the error-location polynomial

$$\begin{bmatrix} \sigma_3 \\ \sigma_2 \\ \sigma_1 \end{bmatrix} = \boldsymbol{M}^{-1}\boldsymbol{S} = \begin{bmatrix} 1 & \alpha^3 & \alpha^{14} \\ \alpha^3 & \alpha^6 & 1 \\ \alpha^{14} & 1 & \alpha^4 \end{bmatrix} \begin{bmatrix} \alpha^7 \\ 0 \\ \alpha^{14} \end{bmatrix} = \begin{bmatrix} \alpha^5 \\ \alpha^{11} \\ \alpha^2 \end{bmatrix}.$$

Therefore the error-location polynomial is

$$\sigma(x) = 1 + \alpha^2 x + \alpha^{11} x^2 + \alpha^5 x^3$$

the roots of which are α^5, α^8, α^{12} which correspond to error-location numbers $X_1 = \alpha^{10}$, $X_2 = \alpha^7$, and $x_3 = \alpha^3$ respectively. To determine the error magnitudes we construct

$$\boldsymbol{X} = \begin{bmatrix} X_1 & X_2 & X_3 \\ X_1^2 & X_2^2 & X_3^2 \\ X_1^3 & X_2^3 & X_3^3 \end{bmatrix} = \begin{bmatrix} \alpha^{10} & \alpha^7 & \alpha^3 \\ \alpha^5 & \alpha^{14} & \alpha^6 \\ 1 & \alpha^6 & \alpha^9 \end{bmatrix}$$

and its inverse

$$\boldsymbol{X}^{-1} = \begin{bmatrix} \alpha^{12} & \alpha^6 & \alpha^2 \\ \alpha^{11} & \alpha^{10} & \alpha^{13} \\ \alpha^{13} & \alpha^2 & \alpha^{11} \end{bmatrix}$$

along with

$$\boldsymbol{S} = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} \alpha^4 \\ 1 \\ \alpha^{10} \end{bmatrix}$$

Using eqn 7.46 the error magnitudes are given by

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} \alpha^{12} & \alpha^6 & \alpha^2 \\ \alpha^{11} & \alpha^{10} & \alpha^{13} \\ \alpha^{13} & \alpha^2 & \alpha^{11} \end{bmatrix} \begin{bmatrix} \alpha^4 \\ 1 \\ \alpha^{10} \end{bmatrix} = \begin{bmatrix} 1 \\ \alpha^4 \\ \alpha^6 \end{bmatrix}$$

and so $Y_1 = 1$, $Y^2 = \alpha^4$, and $Y_3 = \alpha^6$. The error-location numbers $X_1 = \alpha^{10}$, $X_2 = \alpha^7$, and $X_3 = \alpha^3$ correspond to errors in positions x^{10}, x^7, and x^3 respectively, the error pattern is therefore

$$e(x) = x^{10} + \alpha^4 x^7 + \alpha^6 x^3$$

and adding this to $v(x)$ gives

$$c(x) = \alpha^3 x^8 + \alpha^{13} x^7 + \alpha^8 x^6 + \alpha^6 x^5 + \alpha^4 x^4 + \alpha^6 x^3 + \alpha^5 x^2 + \alpha^9 x + \alpha^6$$

as the required codeword polynomial. □

7.8 The Berlekamp algorithm

The Peterson–Gorenstein–Zierler decoder is fundamental to decoding BCH codes, it overcomes the problem of solving the nonlinear syndrome equations by the use of an error-location polynomial whose coefficients can be obtained by standard linear matrix-inversion methods. This though is the weak link in the decoder, for matrix inversion requires excessive computation, especially for large matrices. In particular for non-binary codes for which a second matrix inversion is required to obtain the error magnitudes. As such the Peterson–Gorenstein–Zierler decoder is quite inefficient and whilst it is of prime importance in illustrating the principles of decoding BCH codes it is, nevertheless, of limited practical use.

The Berlekamp algorithm is a fast and efficient algorithm for decoding BCH codes. Like the Peterson–Gorenstein–Zierler decoder it uses the error-location polynomial, but it avoids the need for matrix inversion when determining the polynomial coefficients. The algorithm is generally considered to be significantly more complex than the Peterson–Gorenstein–Zierler decoder. However its complexity lies mainly in the proof of the algorithm, which we omit.

The algorithm uses an iterative technique to find an error-location polynomial $\sigma(x)$ whose coefficients satisfy Newton's identities, as given by eqns 7.23 and 7.24. The algorithm starts by finding a polynomial $\sigma^{(1)}(x)$, whose coefficients satisfy the first of Newton's identities. A suitable set of initial conditions is required to achieve this, otherwise the algorithm may fail to carry out the required number of iterations. The polynomial $\sigma^{(1)}(x)$ must not only satisfy the first identity, but it must be the polynomial of least degree that meets the requirement. Note that the superscript 1 in $\sigma^{(1)}(x)$ is enclosed in parenthesis to avoid any possible ambiguity with powers of $\sigma(x)$. Next a polynomial $\sigma^{(2)}(x)$ is found that satisfies the first and second identities, again the polynomial must be the polynomial of least degree that meets the requirement. To find $\sigma^{(2)}(x)$ we first check if $\sigma^{(1)}(x)$ meets the requirement, if it does then we let $\sigma^{(2)}(x) = \sigma^{(1)}(x)$. Otherwise $\sigma^{(1)}(x)$ is modified by adding a suitable correction term such that the resulting polynomial has coefficients that satisfy the first two equations of Newton's identities. The modification to $\sigma^{(1)}(x)$ must ensure that $\sigma^{(2)}(x)$ is the polynomial of least degree that meets the requirements. The process continues iteratively. Check to see if $\sigma^{(2)}(x)$ satisfies the first three identities, if it does then $\sigma^{(3)}(x) = \sigma^{(2)}(x)$, otherwise modify $\sigma^{(2)}(x)$ to obtain $\sigma^{(3)}(x)$. Then generate $\sigma^{(4)}(x)$, $\sigma^{(5)}(x)$, ... and so forth, until a polynomial $\sigma^{(2t)}(x)$, with coefficients satisfying all the Newton's identities, is obtained. The error-location polynomial is then

$$\sigma(x) = \sigma^{(2t)}(x)$$

and the error-location numbers are found in the usual manner of finding the inverse roots of $\sigma(x)$.

Convolutional codes

8

We have seen that with an (n, k) block code, codewords are constructed independently of each other. For a binary code the n bit output of an encoder depends solely on the k bits entering the encoder. The $n - k$ parity bits of a codeword depend on the k information bits that enter the encoder, each information bit affects only one codeword. Convolutional codes differ from block codes in that the encoder output is constructed not from a single input but also using some of the previous encoder inputs. Clearly memory is required to achieve this, to store inputs for further use. A convolutional code that at a given time generates n outputs from k inputs and m previous inputs is referred to as an (n, k, m) *convolutional code*.

8.1 Convolution

As their name implies, convolutional codes are based on a convolution operation and it is useful to take a look at this before considering convolutional codes. The mathematical operation of convolution can be applied to analogue functions and to discrete functions, here we address only discrete convolution. Figure 8.1 shows a shift register consisting of 4 *stages* which feed into a modulo-2 adder via the links g_1, g_2, g_3, and g_4, each stage is a 1-bit storage device. The input to the first stage also feeds into the modulo-2 adder, via the link g_0. The terms g_0, g_1, g_2, g_3, and g_4 have values 0 or 1 if the link is absent or present respectively. If all the links are present, then $g_0 = g_1 = g_2 = g_3 = g_4 = 1$. The shift register shown in Fig. 8.1 is an example of a *linear feed-forward shift register* as bits are fed towards the output and not back into the shift register.

Consider the sequence of bits

$$\boldsymbol{u} = (u_0\ u_1\ u_2 \ldots)$$

entering the shift register shown in Fig. 8.1, and let

$$\boldsymbol{v} = (v_0\ v_1\ v_2 \ldots)$$

be the sequence of bits leaving the modulo-2 adder as $\boldsymbol{u}$ enters. The sequences $\boldsymbol{u}$ and $\boldsymbol{v}$ are referred to as the *input sequence* and the *output sequence* respectively. If we now

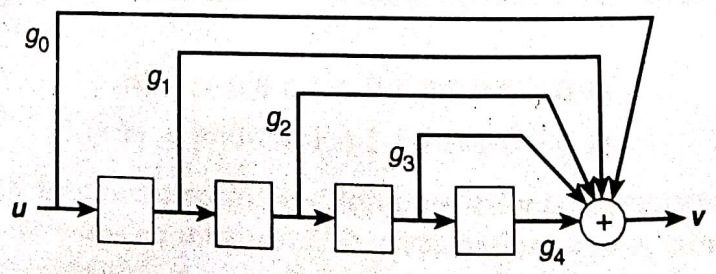


Fig. 8.1 Linear-feedforward shift register with 4 stages.

assume that not all the links in Fig. 8.1 are necessarily present, so that g_0, g_1, g_2, g_3, and g_4 are 0 or 1, then once u_0 enters the register, the output from the modulo-2 adder will be $v_0 = u_0 g_0$ since all the other inputs to the adder are zero. If $u_0 = 0$, or if the link g_0 is absent, so that $g_0 = 0$, then $v_0 = 0$, otherwise $v_0 = 1$. When u_1 enters the register, the bit u_0 moves into the second stage. If $g_1 = 1$, then u_0 is fed into the adder and likewise, if $g_0 = 1$, then u_0 is also fed into the adder. The output from the adder is therefore $v_1 = u_1 g_0 + u_0 g_1$. At the next input the adder output is $v_2 = u_2 g_0 + u_1 g_1 + u_0 g_2$ followed by $v_3 = u_3 g_0 + u_2 g_1 + u_1 g_2 + u_0 g_3$ and $v_4 = u_4 g_0 + u_3 g_1 + u_2 g_2 + u_1 g_3 + u_0 g_4$. In summary the register output is:

$$\begin{aligned} v_0 &= u_0 g_0 \\ v_1 &= u_1 g_0 + u_0 g_1 \\ v_2 &= u_2 g_0 + u_1 g_1 + u_0 g_2 \\ v_3 &= u_3 g_0 + u_2 g_1 + u_1 g_2 + u_0 g_3 \\ v_4 &= u_4 g_0 + u_3 g_1 + u_2 g_2 + u_1 g_3 + u_0 g_4. \end{aligned} \tag{8.1}$$

Note that the first bit u_0 to enter the register has affected all 5 outputs. As the next bit u_5 enters the shift register u_0 will cease to contribute to the output sequence. In a register with m stages each input contributes towards $m+1$ outputs.

Example 8.1

Consider the shift register shown in Fig. 8.1 and let $g_0 = 1$, $g_1 = 1$, $g_2 = 0$, $g_3 = 0$ and $g_4 = 1$. Determine the output sequence for the input sequence $\boldsymbol{u} = (1\ 1\ 0\ 1\ 0\ 1\ \ldots)$.

The first 5 inputs are $u_0 = 1$, $u_1 = 1$, $u_2 = 0$, $u_3 = 1$, and $u_4 = 0$. Using eqns 8.1 gives

$$\begin{aligned} v_0 &= 1 \cdot 1 = 1 \\ v_1 &= 1 \cdot 1 + 1 \cdot 1 = 0 \\ v_2 &= 0 \cdot 1 + 1 \cdot 1 + 1 \cdot 0 = 1 \\ v_3 &= 1 \cdot 1 + 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 = 1 \\ v_4 &= 0 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 0 + 1 \cdot 1 = 0 \end{aligned}$$

as the first 5 bits of the output sequence. The 6th bit is given by

$$\begin{aligned} v_5 &= u_5 g_0 + u_4 g_1 + u_3 g_2 + u_2 g_3 + u_1 g_4 \\ &= 1 \cdot 1 + 0 \cdot 1 + 1 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 \\ &= 0. \end{aligned}$$

The output sequence is therefore $(1\ 0\ 1\ 1\ 0\ 0\ \ldots)$. □

For a finite input sequence, we need to ensure that the last input feeds through the register and appears at the output. To achieve this an additional m zero inputs are required. Consider Fig. 8.1 again, with the input sequence $\boldsymbol{u} = (u_0\, u_1\, u_2\, u_3\, u_4\, u_5\, u_6\, u_7)$. When u_7 enters the register the output is

$$v_7 = u_7 g_0 + u_6 g_1 + u_5 g_2 + u_4 g_3 + u_3 g_4.$$

Assuming now an additional 4 inputs u_8, u_9, u_{10}, and u_{11} then there are a further 4 outputs

$$\begin{aligned} v_8 &= u_8g_0 + u_7g_1 + u_6g_2 + u_5g_3 + u_4g_4 \\ v_9 &= u_9g_0 + u_8g_1 + u_7g_2 + u_6g_3 + u_5g_4 \\ v_{10} &= u_{10}g_0 + u_9g_1 + u_8g_2 + u_7g_3 + u_6g_4 \\ v_{11} &= u_{11}g_0 + u_{10}g_1 + u_9g_2 + u_8g_3 + u_7g_4 \end{aligned}$$

and setting $u_8 = u_9 = u_{10} = u_{11} = 0$ gives

$$\begin{aligned} v_8 &= u_7g_1 + u_6g_2 + u_5g_3 + u_4g_4 \\ v_9 &= u_7g_2 + u_6g_3 + u_5g_4 \\ v_{10} &= u_7g_3 + u_6g_4 \\ v_{11} &= u_7g_4. \end{aligned}$$

All the 7 inputs have now left the shift register and the output sequence is completed. Any further zero inputs will give a zero output. By feeding in the additional m 0 bits the register is said to be cleared or returned to its *zero state*. The m 0 bits can be appended to the input sequence $\boldsymbol{u}$ but should not be confused with the information bits.

Example 8.2

Figure 8.2 shows a shift register with 3 stages and $g_0 = 1$, $g_1 = 0$, $g_2 = 1$, and $g_3 = 1$. Determine the output sequence given the input sequence $\boldsymbol{u} = (1\ 0\ 0\ 1)$.

As $\boldsymbol{u} = (u_0\, u_1\, u_2\, u_3)$ enters the shift register, the output from the adder is

$$\begin{aligned} v_0 &= u_0g_0 \\ v_1 &= u_1g_0 + u_0g_1 \\ v_2 &= u_2g_0 + u_1g_1 + u_0g_2 \\ v_3 &= u_3g_0 + u_2g_1 + u_1g_2 + u_0g_3. \end{aligned}$$

Substituting the values of g_0, g_1, g_2, and g_3 gives

$$\begin{aligned} v_0 &= u_0 \\ v_1 &= u_1 \\ v_2 &= u_2 + u_0 \\ v_3 &= u_3 + u_1 + u_0 \end{aligned}$$

and for $u_0 = 1$, $u_1 = 0$, $u_2 = 0$, and $u_3 = 1$, we get $v_0 = 1$, $v_1 = 0$, $v_2 = 1$, and $v_3 = 0$. To return the register to its zero state we need another 3 inputs, $u_4 = u_5 = u_6 = 0$. This gives

$$\begin{aligned} v_4 &= u_4g_0 + u_3g_1 + u_2g_2 + u_1g_3 = 0 \\ v_5 &= u_5g_0 + u_4g_1 + u_3g_2 + u_2g_3 = 1 \\ v_6 &= u_6g_0 + u_5g_1 + u_4g_2 + u_3g_3 = 1. \end{aligned}$$

The output sequence is therefore $\boldsymbol{v} = (v_0\, v_1\, v_2\, v_3\, v_4\, v_5\, v_6) = (1\,0\,1\,0\,0\,1\,1)$. □

Returning now to eqns 8.1, for the shift register shown in Fig. 8.1, given the input u_j the output from the modulo-2 adder is

$$v_j = u_jg_0 + u_{j-1}g_1 + u_{j-2}g_2 + u_{j-3}g_3 + u_{j-4}g_4$$

or more concisely

$$v_j = \sum_{r=0}^{4} u_{j-r}g_r$$

where $u_{j-r} = 0$ if $j < r$ (this defines $u_{-1} = u_{-2} = u_{-3} = u_{-4} = 0$). For a shift register with m stages and links $g_0, g_1, g_2, \ldots, g_m$ feeding into a modulo-2 adder, the output from the adder is given by

$$v_j = \sum_{r=0}^{m} u_{j-r}g_r \tag{8.2}$$

where again $u_{j-r} = 0$ if $j < r$. Equation 8.2 shows that each output v_j is a *convolution* of $(m+1)$ inputs with $g_0, g_1, \ldots, g_m$. We can think of convolution as a process in which a *window* of $(m+1)$ bits are multiplied by $g_0, g_1, \ldots, g_m$ and added together. The window is then slid along the input sequence by 1 bit and multiplication and summation repeated. The process continues until the window has covered the entire input sequence. Equation 8.2 can be expressed in terms of the input and output sequences as

$$\boldsymbol{v} = \boldsymbol{u} * \boldsymbol{g} \tag{8.3}$$

where the operation $*$ denotes convolution and $\boldsymbol{g} = (g_0\, g_1 \ldots g_m)$ is referred to as a *generator sequence*. The generator sequence can be obtained by inspection of the shift register or alternatively by considering the *impulse response* when the input sequence is a 1 followed by m 0s, i.e. $\boldsymbol{u} = (1\,0\,0\,0 \ldots 0)$. For example, consider again Fig. 8.1 this time with input sequence $\boldsymbol{u} = (u_0\, u_1\, u_2\, u_3\, u_4)$ where $u_0 = 1$ and $u_1 = u_2 = u_3 = u_4 = 0$. Substituting u_0, u_1, u_2, u_3, and u_4 into eqns 8.1 gives

$$\begin{aligned} v_0 &= 1g_0 = g_0 \\ v_1 &= 0g_0 + 1g_1 = g_1 \\ v_2 &= 0g_0 + 0g_1 + 1g_2 = g_2 \\ v_3 &= 0g_0 + 0g_1 + 0g_2 + 1g_3 = g_3 \\ v_4 &= 0g_0 + 0g_1 + 0g_2 + 0g_3 + 1g_4 = g_4. \end{aligned}$$

Hence the output sequence is

$$v = (v_0\, v_1\, v_2\, v_3\, v_4) = (g_0\, g_1\, g_2\, g_3\, g_4) = g$$

and therefore the impulse response gives the generator sequence g. The generator sequence g should not be interpreted as the generator sequence of the shift register but rather of the output from the modulo-2 adder. A line of stages feeding into more than one modulo-2 adder will have a generator sequence for each adder. Example 8.3 below illustrates this.

Example 8.3

Figure 8.3 shows a shift register with 3 stages feeding into two modulo-2 adders. The output v_1 does not have an input directly from u nor from the first stage, but has inputs from the second and third stages. Hence the generator sequence of v_1 is $g_1 = (0\ 0\ 1\ 1)$. Likewise we can see that the generator sequence for v_2 is $g_2 = (1\ 0\ 1\ 1)$. Using eqn 8.2 gives

$$\begin{aligned}
v_0 &= u_0g_0 + u_{-1}g_1 + u_{-2}g_2 + u_{-3}g_3\\
v_1 &= u_1g_0 + u_0g_1 + u_{-1}g_2 + u_{-2}g_3\\
v_2 &= u_2g_0 + u_1g_1 + u_0g_2 + u_{-1}g_3\\
v_3 &= u_3g_0 + u_2g_1 + u_1g_2 + u_0g_3\\
v_4 &= u_4g_0 + u_3g_1 + u_2g_2 + u_1g_3
\end{aligned}$$

for an input $u = (u_0\, u_1\, u_2\, u_3\, u_4)$ and substituting the values of g_1 and g_2 we get

$$\begin{aligned}
v_0 &= 0\\
v_1 &= 0\\
v_2 &= u_0\\
v_3 &= u_1 + u_0\\
v_4 &= u_2 + u_1
\end{aligned}$$

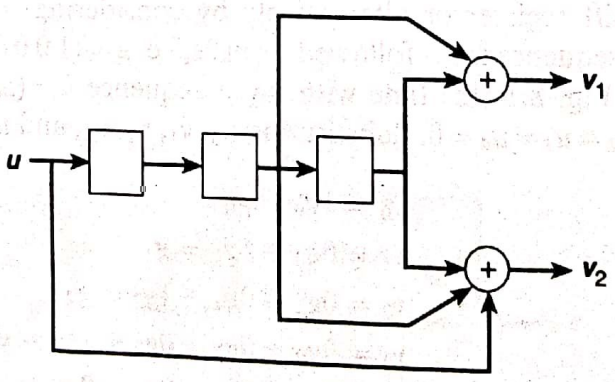
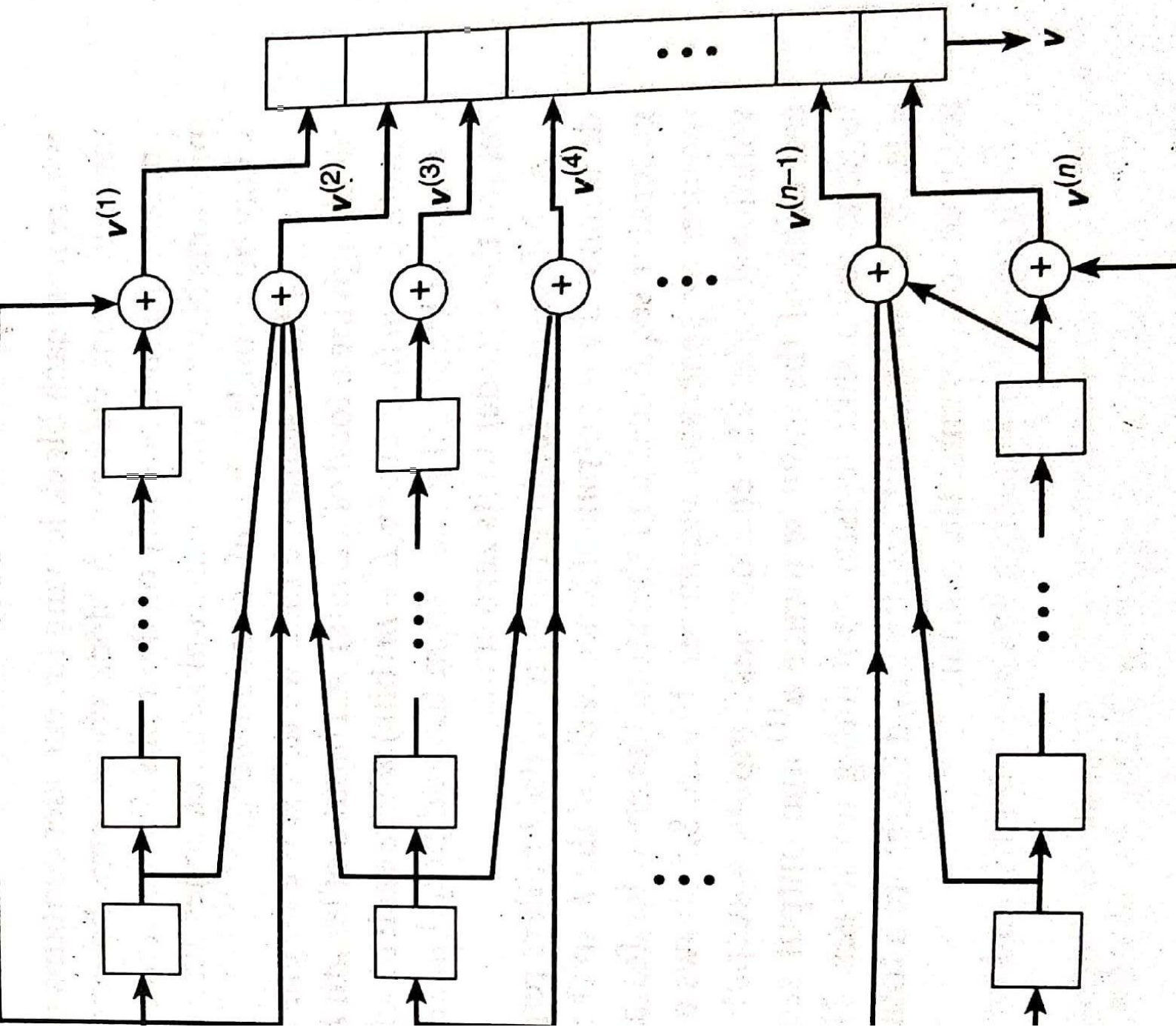


Fig. 8.3 Register with 2 outputs.



adders whose inputs are taken from various stages (depending upon the particular code). Each bit in the encoder can affect any of the n outputs, and given that a bit can stay in the encoder for up to $m+1$ inputs then each bit can affect up to $n(m+1)$ output bits. This measure of the extent to which an input bit affects the output is referred to as the *constraint length*.

The input sequence to the ith shift register is denoted by

$$\boldsymbol{u}^{(i)} = (u_0^{(i)} u_1^{(i)} u_2^{(i)} \ldots) \tag{8.4}$$

and bits are fed into their respective shift register one bit at a time. Viewing the encoder on the whole, bits enter in blocks of k bits at a time, and we can express the input sequence to the encoder as

$$\boldsymbol{u} = (u_0^{(1)} u_0^{(2)} \ldots u_0^{(k)}, u_1^{(1)} u_1^{(2)} \ldots u_1^{(k)}, \ldots). \tag{8.5}$$

Here the first input to the encoder is the block of k bits $u_0^{(1)} u_0^{(2)} \ldots u_0^{(k)}$ followed by the block $u_1^{(1)} u_1^{(2)} \ldots u_1^{(k)}$ and so forth. The output sequence of the jth modulo-2 adder is

$$\boldsymbol{v}^{(j)} = (v_0^{(j)} v_1^{(j)} v_2^{(j)} \ldots) \tag{8.6}$$

and again viewing the encoder on the whole the output sequence is

$$\boldsymbol{v} = (v_0^{(1)} v_0^{(2)} \ldots v_0^{(n)}, v_1^{(1)} v_1^{(2)} \ldots v_1^{(n)}, \ldots) \tag{8.7}$$

where now each block leaving the encoder contains n bits. This may sound like the structure that we have in block codes, here however an n-bit block leaving the encoder depends not only on the k-bits that entered the encoder but also on up to m previous blocks. This is quite unlike block codes where each n-bit codeword depends solely on a single k-bit information word.

If $\boldsymbol{u}$ is a finite sequence then we define the *length* L of $\boldsymbol{u}$ as the number of k-bit blocks (giving a total of kL bits). An input $\boldsymbol{u}$ of length L gives an output sequence $\boldsymbol{v}$ of length $L+m$ (a total of $n(L+m)$ bits) where each block of $\boldsymbol{v}$ contains n bits and where the last m blocks of $\boldsymbol{v}$ arise from the additional m zero inputs that are required to return the encoder to its zero state.

A shift register feeding into n modulo-2 adders requires n generator sequences to determine the n outputs. Hence each of the k shift registers in Fig. 8.4 requires n generator sequences and therefore the encoder for an (n, k, m) convolutional code requires nk generator sequences. Figure 8.5 shows an encoder for the $(4, 3, 2)$ convolutional code. The encoder has 3 inputs, 4 outputs and memory order $m=2$. We can think of the input sequence $\boldsymbol{u}^{(1)}$ and output sequence $\boldsymbol{v}^{(1)}$ as being connected together by a shift register containing no stages. Hence the encoder can be considered as having 3 shift registers and therefore a total of 12 generator sequences are required to determine the output.

Each generator sequence has $m+1$ terms, and we let

$$\boldsymbol{g}^{(i,j)} = (g_0^{(i,j)} g_1^{(i,j)} \ldots g_m^{(i,j)})$$

denote the generator sequence connecting the ith input to the jth output. Then by considering the impulse response of each shift register or by inspecting the

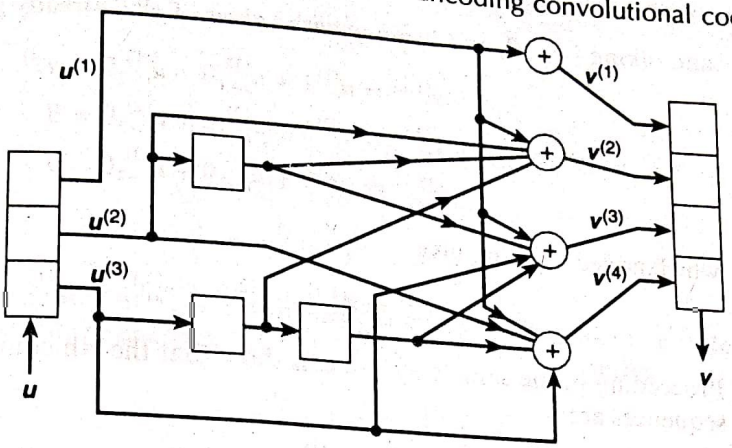


Fig. 8.5 Encoder for the (4, 3, 2) convolutional code.

connections to each adder we find that

$$\begin{aligned} g^{(1,1)} &= (1\ 0\ 0),\ g^{(1,2)} = (1\ 0\ 0),\ g^{(1,3)} = (1\ 0\ 0),\ g^{(1,4)} = (1\ 0\ 0) \\ g^{(2,1)} &= (0\ 0\ 0),\ g^{(2,2)} = (1\ 1\ 0),\ g^{(2,3)} = (0\ 1\ 0),\ g^{(2,4)} = (1\ 0\ 0) \\ g^{(3,1)} &= (0\ 0\ 0),\ g^{(3,2)} = (0\ 1\ 0),\ g^{(3,3)} = (1\ 0\ 1),\ g^{(3,4)} = (1\ 0\ 1). \end{aligned} \tag{8.8}$$

In Section 8.1 we saw that given a generator sequence g, the output sequence v for an input sequence u is $v = u * g$ (eqn 8.3). For the encoder shown in Fig. 8.5, each output sequence has contributions from up to 3 inputs and therefore eqn 8.3 has to be modified to take the additional inputs into account. This is achieved by adding the term $u^{(i)} * g^{(i,j)}$ term to $v^{(j)}$ for each input $u^{(i)}$. For convenience we define

$$v^{(i,j)} = u^{(i)} * g^{(i,j)} \tag{8.9}$$

as the jth output arising from the ith input. The rth component of $v^{(i,j)}$ is

$$v_r^{(i,j)} = u_r^{(i)} g_0^{(i,j)} + u_{r-1}^{(i)} g_1^{(i,j)} + u_{r-2}^{(i)} g_2^{(i,j)}. \tag{8.10}$$

The jth output sequence can be expressed as

$$v^{(j)} = v^{(1,j)} + v^{(2,j)} + v^{(3,j)} \tag{8.11}$$

which has

$$v_r^{(j)} = v_r^{(1,j)} + v_r^{(2,j)} + v_r^{(3,j)} \tag{8.12}$$

as its rth component. From eqns 8.10 and 8.12 the components of each output sequence can be determined. Consider first the components of the output sequence $v^{(1)}$. The rth component of $v^{(1)}$ arising from the ith input is

$$v_r^{(i,1)} = u_r^{(i)} g_0^{(i,1)} + u_{r-1}^{(i)} g_1^{(i,1)} + u_{r-2}^{(i)} g_2^{(i,1)}$$

and taking $i = 1$, 2, and 3, along with values of $g^{(i,j)}$ already given, we obtain

$$v_r^{(1,1)} = u_r^{(1)}1 + u_{r-1}^{(1)}0 + u_{r-2}^{(1)}0 = u_r^{(1)}$$
$$v_r^{(2,1)} = u_r^{(2)}0 + u_{r-1}^{(2)}0 + u_{r-2}^{(2)}0 = 0$$
$$v_r^{(3,1)} = u_r^{(3)}0 + u_{r-1}^{(3)}0 + u_{r-2}^{(3)}0 = 0$$

which added together give

$$v_r^{(1)} = v_r^{(1,1)} + v_r^{(2,1)} + v_r^{(3,1)} = u_r^{(1)}.$$

Proceeding in the same way, we can show that the rth component of all 4 output sequences are

$$\begin{aligned} v_r^{(1)} &= u_r^{(1)} \\ v_r^{(2)} &= u_r^{(1)} + u_r^{(2)} + u_{r-1}^{(2)} + u_{r-1}^{(3)} \\ v_r^{(3)} &= u_r^{(1)} + u_{r-1}^{(2)} + u_r^{(3)} + u_{r-2}^{(3)} \\ v_r^{(4)} &= u_r^{(1)} + u_r^{(2)} + u_r^{(3)} + u_{r-2}^{(3)} \end{aligned} \tag{8.13}$$

and the output sequence leaving the encoder is

$$\boldsymbol{v} = (v_0^{(1)} v_0^{(2)} v_0^{(3)} v_0^{(4)}, v_1^{(1)} v_1^{(2)} v_1^{(3)} v_1^{(4)}, v_2^{(1)} v_2^{(2)} v_2^{(3)} v_2^{(4)} \ldots).$$

Example 8.4

Determine the output sequence from the (4, 3, 2) convolutional encoder, shown in Fig. 8.5, given the input sequences $\boldsymbol{u}^{(1)} = (1\ 0\ 1)$, $\boldsymbol{u}^{(2)} = (1\ 1\ 0)$, and $\boldsymbol{u}^{(3)} = (0\ 1\ 1)$.

The input sequences are of the form $\boldsymbol{u} = (u_0\, u_1\, u_2)$ with $u_r = 0$ if $r < 0$. We start by taking $r = 0$, then from eqns 8.13

$$\begin{aligned} v_0^{(1)} &= u_0^{(1)} \\ v_0^{(2)} &= u_0^{(1)} + u_0^{(2)} \\ v_0^{(3)} &= u_0^{(1)} + u_0^{(3)} \\ v_0^{(4)} &= u_0^{(1)} + u_0^{(2)} + u_0^{(3)} \end{aligned}$$

and substituting $u_0^{(1)} = 1$, $u_0^{(2)} = 1$, $u_0^{(3)} = 0$ gives $v_0^{(1)} = 1$, $v_0^{(2)} = 0$, $v_0^{(3)} = 1$, and $v_0^{(4)} = 0$.

Next take $r = 1$, this gives

$$\begin{aligned} v_1^{(1)} &= u_1^{(1)} = 0 \\ v_1^{(2)} &= u_1^{(1)} + u_1^{(2)} + u_0^{(2)} + u_0^{(3)} = 0 \\ v_1^{(3)} &= u_1^{(1)} + u_0^{(2)} + u_1^{(3)} = 0 \\ v_1^{(4)} &= u_1^{(1)} + u_1^{(2)} + u_1^{(3)} = 0 \end{aligned}$$

and for $r=2$

$$\begin{aligned} v_2^{(1)} &= u_2^{(1)} = 1 \\ v_2^{(2)} &= u_2^{(1)} + u_2^{(2)} + u_1^{(2)} + u_1^{(3)} = 1 \\ v_2^{(3)} &= u_2^{(1)} + u_1^{(2)} + u_2^{(3)} + u_0^{(3)} = 1 \\ v_2^{(4)} &= u_2^{(1)} + u_2^{(2)} + u_2^{(3)} + u_0^{(3)} = 0. \end{aligned}$$

Because the inputs are finite sequences and $m=2$ we need to consider 2 more 0 inputs to return the encoder back to the zero state. This requires

$$\begin{aligned} u_3^{(1)} = u_3^{(2)} = u_3^{(3)} &= 0 \\ u_4^{(1)} = u_4^{(2)} = u_4^{(3)} &= 0. \end{aligned}$$

For $r=3$ we get

$$\begin{aligned} v_3^{(1)} &= u_3^{(1)} = 0 \\ v_3^{(2)} &= u_3^{(1)} + u_3^{(2)} + u_2^{(2)} + u_2^{(3)} = 1 \\ v_3^{(3)} &= u_3^{(1)} + u_2^{(2)} + u_3^{(3)} + u_1^{(3)} = 1 \\ v_3^{(4)} &= u_3^{(1)} + u_3^{(2)} + u_3^{(3)} + u_1^{(3)} = 1 \end{aligned}$$

and $r=4$ gives

$$\begin{aligned} v_4^{(1)} &= u_4^{(1)} = 0 \\ v_4^{(2)} &= u_4^{(1)} + u_4^{(2)} + u_3^{(2)} + u_3^{(3)} = 0 \\ v_4^{(3)} &= u_4^{(1)} + u_3^{(2)} + u_4^{(3)} + u_2^{(3)} = 1 \\ v_4^{(4)} &= u_4^{(1)} + u_4^{(2)} + u_4^{(3)} + u_2^{(3)} = 1. \end{aligned}$$

Therefore the output sequence is

$$\boldsymbol{v} = (1\,0\,1\,0,\ 0\,0\,0\,0,\ 1\,1\,1\,0,\ 0\,1\,1\,1,\ 0\,0\,1\,1). \qquad \square$$

The rth component of the output sequence, given by eqns 8.13, can be obtained directly from the encoder shown in Fig. 8.5. Let's assume that the input sequence is at the rth component, so that $u_r^{(1)}$, $u_r^{(2)}$, and $u_r^{(3)}$ are the inputs, then the 4 outputs can be determined by inspecting the encoder. The output $\boldsymbol{v}^{(1)}$ has only one input to its modulo-2 adder, namely that which arrives directly from $\boldsymbol{u}^{(1)}$ and so $v_r^{(1)} = u_r^{(1)}$, as given by eqns 8.13. The output $\boldsymbol{v}^{(2)}$ has:

(1) inputs directly from $\boldsymbol{u}^{(1)}$, so contributing $u_r^{(1)}$ to $\boldsymbol{v}^{(2)}$;
(2) inputs directly from $\boldsymbol{u}^{(2)}$, which contributes $u_r^{(2)}$;
(3) a contribution from $\boldsymbol{u}^{(2)}$ that is delayed by 1 stage and therefore contributes $u_{r-1}^{(2)}$;
(4) a contribution from $\boldsymbol{u}^{(3)}$ that is also delayed by 1 stage and therefore contributes $u_{r-1}^{(3)}$ to $\boldsymbol{v}^{(2)}$.

Adding together the four contributions gives $u_r^{(1)} + u_r^{(2)} + u_{r-1}^{(2)} + u_{r-1}^{(3)}$ as the rth component of $v^{(2)}$, which again is in agreement with that given by eqns 8.13. Likewise we can verify that $v_r^{(3)}$ and $v_r^{(4)}$ are as given by eqns 8.13.

As with blockcodes the error-control properties of convolutional codes depend on the distance characteristics of the resulting encoded sequences. However with convolutional codes there are several minimum distance measures, the most important measure being the minimum *free distance* defined as the minimum distance between any two encoded sequences. If two sequences v_1 and v_2 are of different length then zeros are added to the shorter corresponding input sequence u_1 or u_2 so that the sequences are the same length. Hence the definition of free distance includes all sequences and not just sequences with the same length. Convolutional codes are linear codes and therefore the sum of two encoded sequences v_1 and v_2 gives another encoded sequence v_3 where the weight of v_3 equals the distance between v_1 and v_2, and $v_3 \neq 0$ if $v_1 \neq v_2$. Hence the free distance of a convolutional code is given by the minimum-weight sequence of any length produced by a nonzero input sequence.

8.3 Generator matrices for convolutional codes

The use of convolution to derive the output sequences is rather tedious and as might be expected, the use of matrices can help to simplify encoding. We have seen that the output sequence $v = u * g$, where u is an input sequence, g is a generator sequence and $*$ is the convolution operation. This equation can be expressed in matrix form, and the convolution operation replaced by matrix multiplication, by defining a *generator matrix* G such that

$$v = uG. \tag{8.14}$$

The matrix G is constructed from the components of the generator sequences and for an (n, k, m) convolutional code

$$G = \begin{bmatrix} G_0 & G_1 & G_2 & \dots & G_m & 0 & 0 & \dots \\ 0 & G_0 & G_1 & G_2 & \dots & G_m & 0 & \dots \\ 0 & 0 & G_0 & G_1 & G_2 & \dots & G_m & \dots \\ \cdot & & & & & \dots & & \\ \cdot & & & & & & \dots & \\ \cdot & & & & & & & \dots \end{bmatrix} \tag{8.15}$$

where G_r is the k by n matrix

$$G_r = \begin{bmatrix} g_r^{(1,1)} & g_r^{(1,2)} & \dots & g_r^{(1,n)} \\ g_r^{(2,1)} & g_r^{(2,2)} & \dots & g_r^{(2,n)} \\ \vdots & & & \vdots \\ g_r^{(k,1)} & g_r^{(k,2)} & \dots & g_r^{(k,n)} \end{bmatrix} \tag{8.16}$$

and **0** is a k by n zero matrix. Each row in $\boldsymbol{G}$ is obtained from the previous row by shifting all the matrices one place to the right. If $\boldsymbol{u}$ has finite length L then $\boldsymbol{G}$ has L rows and $L+m$ columns. On substituting eqn 8.16 into 8.15 the matrix $\boldsymbol{G}$ has kL rows and $n(L+m)$ columns. For the (4, 3, 2) code already considered, we have

$$\boldsymbol{G}_r = \begin{bmatrix} g_r^{(1,1)} & g_r^{(1,2)} & g_r^{(1,3)} & g_r^{(1,4)} \\ g_r^{(2,1)} & g_r^{(2,2)} & g_r^{(2,3)} & g_r^{(2,4)} \\ g_r^{(3,1)} & g_r^{(3,2)} & g_r^{(3,3)} & g_r^{(3,4)} \end{bmatrix}.$$

Using the generator sequences already given for the (4, 3, 2) code (see section 8.2) we get:

$$\boldsymbol{G}_0 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$\boldsymbol{G}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{G}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

The generator matrix for the (4, 3, 2) convolutional code is therefore

$$\boldsymbol{G} = \begin{bmatrix} 1\,1\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 & \mathbf{0} & \mathbf{0}\ldots \\ 0\,1\,0\,1 & 0\,1\,1\,0 & 0\,0\,0\,0 & & \\ 0\,0\,1\,1 & 0\,1\,0\,0 & 0\,0\,1\,1 & & \\ \mathbf{0} & 1\,1\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 & \mathbf{0}\ldots \\ & 0\,1\,0\,1 & 0\,1\,1\,0 & 0\,0\,0\,0 & \\ & 0\,0\,1\,1 & 0\,1\,0\,0 & 0\,0\,1\,1 & \\ \mathbf{0} & \mathbf{0} & 1\,1\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 \\ & & 0\,1\,0\,1 & 0\,1\,1\,0 & 0\,0\,0\,0 \\ & & 0\,0\,1\,1 & 0\,1\,0\,0 & 0\,0\,1\,1 \\ \cdot & \cdot & & \cdots & \\ \cdot & \cdot & & \cdots & \\ \cdot & \cdot & & & \cdots \end{bmatrix} \qquad (8.17)$$

where **0** is a 3 by 4 zero matrix. Once the generator matrix is established finding the output sequence for a given input sequence is straightforward. Reconsider Example 8.4 in which the output sequence for the input sequences $\boldsymbol{u}^{(1)} = (1\ 0\ 1)$, $\boldsymbol{u}^{(2)} = (1\ 1\ 0)$ and $\boldsymbol{u}^{(3)} = (0\ 1\ 1)$ was found using convolution. The input sequence to the register is

$$\begin{aligned} \boldsymbol{u} &= (u_0^{(1)} u_0^{(2)} u_0^{(3)}, u_1^{(1)} u_1^{(2)} u_1^{(3)}, u_2^{(1)} u_2^{(2)} u_2^{(3)}) \\ &= (1\ 1\ 0, 0\ 1\ 1, 1\ 0\ 1). \end{aligned}$$

The generator matrix G is the 9 by 20 matrix

$$G = \begin{bmatrix} 1\,1\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,0\,0\,0 \\ 0\,1\,0\,1 & 0\,1\,1\,0 & 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,0\,0\,0 \\ 0\,0\,1\,1 & 0\,1\,0\,0 & 0\,0\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 1\,1\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,1\,0\,1 & 0\,1\,1\,0 & 0\,0\,0\,0 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,1\,1 & 0\,1\,0\,0 & 0\,0\,1\,1 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,0\,0 & 1\,1\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,1\,0\,1 & 0\,1\,1\,0 & 0\,0\,0\,0 \\ 0\,0\,0\,0 & 0\,0\,0\,0 & 0\,0\,1\,1 & 0\,1\,0\,0 & 0\,0\,1\,1 \end{bmatrix}$$

and using $v = uG$ gives

$$v = (1\,0\,1\,0, 0\,0\,0\,0, 1\,1\,1\,0, 0\,1\,1\,1, 0\,0\,1\,1)$$

which is in agreement with the answer obtained in Example 8.4.

8.4 Generator polynomials for convolutional codes

Whilst the use of generator matrices is an improvement over the use of convolution, it is still nevertheless awkward due to the large size of the generator matrices. However, encoding can be further simplified through the use of *generator polynomials*. Consider the input sequence $u = (u_0\, u_1\, u_2 \ldots)$, it can be represented by the polynomial

$$u(D) = u_0 + u_1 D + u_2 D^2 + \cdots \tag{8.18}$$

where D is the *unit-delay operator* and represents a delay of 1 bit, D^2 represents a 2-bit delay and so forth. In eqn 8.18 u_1 is interpreted as being delay by 1 bit relative to u_0, that is, it arrives 1 bit later than u_0. The bit u_2 is delayed by 2 bits relative to u_0 and 1 bit relative to u_1. Given an output sequence $v = (v_0\, v_1\, v_2 \ldots)$ we can likewise write

$$v(D) = v_0 + v_1 D + v_2 D^2 + \cdots \tag{8.19}$$

The polynomials $u(D)$ and $v(D)$ are referred to as the *input polynomial* and *output polynomial* respectively. If $v(D)$ is the output arising from $u(D)$ then

$$v(D) = u(D)g(D) \tag{8.20}$$

where $g(D)$ is a *generator polynomial*. The product $u(D)g(D)$ is constructed using polynomial multiplication subject to modulo-2 arithmetic.

Example 8.5

Given the arbitrary polynomials $u(D) = 1 + D^2 + D^3$ and $g(D) = 1 + D^3$ then

$$\begin{aligned} v(D) &= (1 + D^2 + D^3)(1 + D^3) \\ &= 1 + D^3 + D^2 + D^5 + D^3 + D^6 \\ &= 1 + D^2 + D^5 + D^6. \end{aligned}$$

□

The generator polynomial can be determined directly from the encoder, in the same way as the generator sequences are determined. Each link to a modulo-2 adder contributes a D^r term to the generator polynomial where r is the number of stages that a bit has to pass through to arrive at the adder. The rth component of the generator sequence $\boldsymbol{g} = (g_0\, g_1\, g_2 \cdots g_m)$ is 0 or 1 depending on whether, after the rth stage there is a link to the adder, and therefore the rth component in the generator polynomial can be written as $g_r D^r$. Hence, for m stages, the generator polynomial can be expressed as

$$g(D) = g_0 + g_1 D + g_2 D^2 + \cdots + g_m D^m. \tag{8.21}$$

Example 8.6

Determine the generator polynomial of the shift register shown in Fig. 8.2. Hence find the output sequence given the input sequence $\boldsymbol{u} = (1\ 0\ 0\ 1)$.

We have already seen that the generator sequence of the encoder shown in Fig. 8.2 is $\boldsymbol{g} = (1\ 0\ 1\ 1)$ and the generator polynomial is therefore

$$g(D) = g_0 + g_1 D + g_2 D^2 + g_3 D^3 = 1 + D^2 + D^3.$$

The input polynomial corresponding to the input sequence $\boldsymbol{u} = (1\ 0\ 0\ 1)$ is

$$u(D) = u_0 + u_1 D + u_2 D^2 + u_3 D^3 = 1 + D^3$$

and the output polynomial is therefore

$$\begin{aligned} v(D) &= u(D)g(D) \\ &= (1 + D^3)(1 + D^2 + D^3) \\ &= 1 + D^2 + D^5 + D^6 \end{aligned}$$

giving the output sequence

$$\boldsymbol{v} = (v_0\ v_1\ v_2\ v_3\ v_4\ v_5\ v_6) = (1\ 0\ 1\ 0\ 0\ 1\ 1).$$

The results agree with that obtained using convolution in Example 8.2. □

An (n, k, m) convolutional code has k generator polynomials for each of the n output sequences and therefore a total of nk generator polynomials. Let

$$g^{(i,j)}(D) = \sum_{r=0}^{m} g_r^{(i,j)} D^r \tag{8.22}$$

be the generator polynomial for the jth output arising from the ith input, where the polynomial coefficients $g_r^{(i,j)}$ are the components of the generator sequence $\boldsymbol{g}^{(i,j)}$. The (4, 3, 2) encoder shown in Fig. 8.5 has 12 generator polynomials of the form

$$g^{(i,j)}(D) = g_0^{(i,j)} + g_1^{(i,j)} D + g_2^{(i,j)} D^2.$$

Taking $i = 1, 2$, and 3, and $j = 1, 2, 3$, and 4, along with the generator sequences given in Section 8.2, gives

$$g^{(1,1)}(D) = 1,\ g^{(1,2)}(D) = 1,\ g^{(1,3)}(D) = 1,\ g^{(1,4)}(D) = 1$$
$$g^{(2,1)}(D) = 0,\ g^{(2,2)}(D) = 1 + D,\ g^{(2,3)}(D) = D,\ g^{(2,4)}(D) = 1$$
$$g^{(3,1)}(D) = 0,\ g^{(3,2)}(D) = D,\ g^{(3,3)}(D) = 1 + D^2,\ g^{(3,4)}(D) = 1 + D^2.$$

The jth output polynomial is a linear combination of the contributions from each of the k inputs and is given by

$$v^{(j)}(D) = \sum_{i=1}^{k} u^{i}(D) g^{(i,j)}(D). \tag{8.23}$$

Example 8.7
Consider again the (4, 3, 2) code with input sequence $\boldsymbol{u}^{(1)} = (1\ 0\ 1)$, $\boldsymbol{u}^{(2)} = (1\ 1\ 0)$, and $\boldsymbol{u}^{(3)} = (0\ 1\ 1)$. The corresponding input polynomials are

$$u^{(1)}(D) = 1 + D^2$$
$$u^{(2)}(D) = 1 + D$$
$$u^{(3)}(D) = D + D^2.$$

Using eqn 8.23 and the generator polynomials $g^{(i,j)}(D)$ already derived, the first output polynomial is

$$\begin{aligned} v^{(1)}(D) &= u^{(1)}(D)g^{(1,1)}(D) + u^{(2)}(D)g^{(2,1)}(D) + u^{(3)}(D)g^{(3,1)}(D) \\ &= (1 + D^2)1 + (1 + D)0 + (D + D^2)0 \\ &= 1 + D^2 \end{aligned}$$

and the second

$$\begin{aligned} v^{(2)}(D) &= u^{(1)}(D)g^{(1,2)}(D) + u^{(2)}(D)g^{(2,2)}(D) + u^{(3)}(D)g^{(3,2)}(D) \\ &= (1 + D^2)1 + (1 + D)(1 + D) + (D + D^2)D \\ &= D^2 + D^3. \end{aligned}$$

Likewise $v^{(3)}(D)$ and $v^{(4)}(D)$ can be determined so giving

$$v^{(1)}(D) = 1 + D^2$$
$$v^{(2)}(D) = D^2 + D^3$$
$$v^{(3)}(D) = 1 + D^2 + D^3 + D^4$$
$$v^{(4)}(D) = D^3 + D^4.$$

□

The output polynomial $v(D)$ has contributions from $v^{(1)}(D)$, $v^{(2)}(D), \ldots, v^{(n)}(D)$ suitably delayed to take into account the order in which bits leave the encoder and it can be shown that

$$v(D) = \sum_{j=1}^{n} D^{j-1} v^{(j)}(D^n). \tag{8.24}$$

The components of $\boldsymbol{v}$ are then given by the coefficients of $v(D)$.

Example 8.8

The output polynomial $v(D)$ for the $(4, 3, 2)$ code, given by eqn 8.24 is

$$v(D) = v^{(1)}(D^4) + Dv^{(2)}(D^4) + D^2v^{(3)}(D^4) + D^3v^{(4)}(D^4)$$

and using $v^{(1)}(D)$, $v^{(2)}(D)$, $v^{(3)}(D)$, and $v^{(4)}(D)$ obtained in Example 8.7 gives

$$\begin{aligned} v(D) &= (1 + D^8) + D(D^8 + D^{12}) + D^2(1 + D^8 + D^{12} + D^{16}) + D^3(D^{12} + D^{16}) \\ &= 1 + D^2 + D^8 + D^9 + D^{10} + D^{13} + D^{14} + D^{15} + D^{18} + D^{19}. \end{aligned}$$

Expressing this as the vector

$$\boldsymbol{v} = (v_0\ v_1\ v_2\ v_3,\ v_4\ v_5\ v_6\ v_7,\ v_8\ v_9\ v_{10}\ v_{11},\ v_{12}\ v_{13}\ v_{14}\ v_{15},\ v_{16}\ v_{17}\ v_{18}\ v_{19})$$

gives

$$\boldsymbol{v} = (1\ 0\ 1\ 0,\ 0\ 0\ 0\ 0,\ 1\ 1\ 1\ 0,\ 0\ 1\ 1\ 1,\ 0\ 0\ 1\ 1).$$

This therefore is the output given the inputs $\boldsymbol{u}^{(1)} = (1\ 0\ 1)$, $\boldsymbol{u}^{(2)} = (1\ 1\ 0)$, and $\boldsymbol{u}^{(3)} = (0\ 1\ 1)$ and is the same as that obtained before, using convolution and using matrices. □

The output polynomial $v(D)$ can be expressed directly in terms of the input polynomials, substituting eqn 8.23 into eqn 8.24 gives

$$v(D) = \sum_{i=1}^{k} u^{(i)}(D^n)g^{(i)}(D) \tag{8.25}$$

where

$$g^{(i)}(D) = \sum_{j=1}^{n} D^{j-1}g^{(i,j)}(D^n) \tag{8.26}$$

is the generator polynomial of the ith input summed over all n outputs.

Example 8.9

The $(4, 3, 2)$ code has generator polynomials $g^{(1)}(D)$, $g^{(2)}(D)$, and $g^{(3)}(D)$ representing the individual contributions of the three inputs to the total output from the encoder. From eqn 8.26 we get

$$\begin{aligned} g^{(1)}(D) &= g^{(1,1)}(D^4) + Dg^{(1,2)}(D^4) + D^2g^{(1,3)}(D^4) + D^3g^{(1,4)}(D^4) \\ g^{(2)}(D) &= g^{(2,1)}(D^4) + Dg^{(2,2)}(D^4) + D^2g^{(2,3)}(D^4) + D^3g^{(2,4)}(D^4) \\ g^{(3)}(D) &= g^{(3,1)}(D^4) + Dg^{(3,2)}(D^4) + D^2g^{(3,3)}(D^4) + D^3g^{(3,4)}(D^4) \end{aligned}$$

and substituting the generator polynomials gives

$$\begin{aligned} g^{(1)}(D) &= 1 + D + D^2 + D^3 \\ g^{(2)}(D) &= D + D^3 + D^5 + D^6 \\ g^{(3)}(D) &= D^2 + D^3 + D^5 + D^{10} + D^{11}. \end{aligned}$$

If we once again consider the inputs $\boldsymbol{u}^{(1)} = (1\ 0\ 1)$, $\boldsymbol{u}^{(2)} = (1\ 1\ 0)$, and $\boldsymbol{u}^{(3)} = (0\ 1\ 1)$, so that $u^{(1)}(D) = 1 + D^2$, $u^{(2)}(D) = 1 + D$ and $u^{(3)}(D) = D + D^2$, then using eqn 8.25 gives

$$\begin{aligned} v(D) &= u^{(1)}(D^4)g^{(1)}(D) + u^{(2)}(D^4)g^{(2)}(D) + u^{3}(D^4)g^{(3)}(D) \\ &= (1 + D^8)(1 + D + D^2 + D^3) + (1 + D^4)(D + D^3 + D^5 + D^6) \\ &\quad + (D^4 + D^8)(D^2 + D^3 + D^5 + D^{10} + D^{11}) \\ &= 1 + D^2 + D^8 + D^9 + D^{10} + D^{13} + D^{14} + D^{15} + D^{18} + D^{19} \end{aligned}$$

as obtained in Example 8.8. □

8.5 Graphical representation of convolutional codes

Convolutional codes can be represented graphically using tree, trellis and state diagrams, all of which show the state of a register and the encoder output for all possible inputs. The *state* of a register is defined as the contents of the stages at a given point during encoding. The diagrams provide an interesting way of looking at the structure and encoding of convolutional codes, and furthermore, the trellis diagram plays an important role when decoding using the Viterbi algorithm (considered in Section 8.6).

We first consider *state diagrams*. Figure 8.6 shows an encoder for a (2, 1, 2) convolutional code. At any point during encoding, the encoder can be in one of the 4 states 00, 01, 10 and 11, referred to as states S_0, S_1, S_2, and S_3 respectively. The states are shown in Fig. 8.7 as *nodes* (filled circles) connected together by *branches* (solid or dashed lines) that represent transitions from one state to another, depending upon whether the input is 1 (solid line) or 0 (dashed line). The output occurring with each transition is shown next to the relevant branch. The state diagram gives a complete description of encoding in that for a given input the output and the next state of the encoder can be determined. To arrive at the state diagram we need to consider the operation of the encoder, taking in to account all possible transitions.

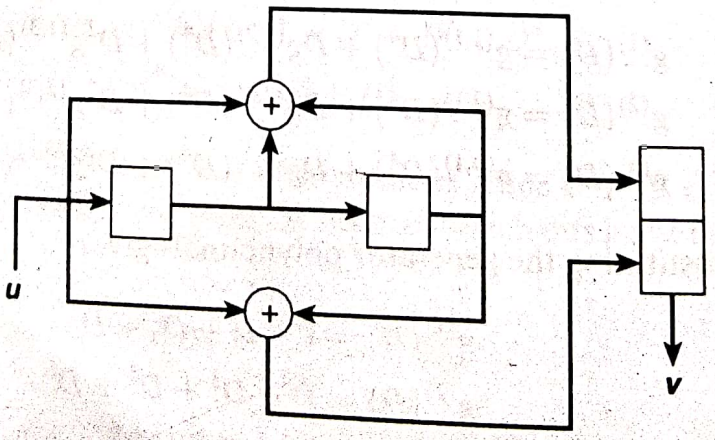


Fig. 8.6 A (2, 1, 2) convolutional encoder.

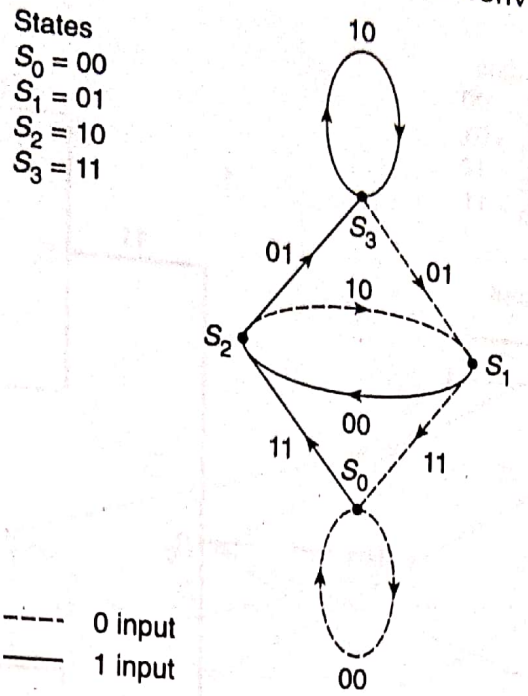


Fig. 8.7 State diagram for the (2, 1, 2) convolutional code.

Referring to Fig. 8.7 we assume that encoding starts at S_0, that is at the zero state with the stages at 00. Whilst the input stays at 0, transitions occur to the same state S_0 and the output is 00. At the first nonzero input, the stage contents change to 10 and the transition is to S_2 giving an output of 11. If the next input is also a 1, then the stage contents change to 11, the transition is to S_3 and the output is 01. Otherwise a 0 input gives a transition to S_1 and an output of 10. The state diagram is completed by considering transitions from S_3 and S_1 for inputs 0 and 1. Given an arbitrary input $\boldsymbol{u} = (u_0\, u_1\, u_2 \ldots)$ the output can be determined by following the transitions through the state diagram.

Example 8.10

Consider the input sequence $\boldsymbol{u} = (1\ 1\ 0\ 1\ 0\ 0)$. Referring to the state diagram in Fig. 8.7, and starting from the state S_0 we get

Input 1, transition to S_2 giving an output 11
Input 1, transition to S_3 giving an output 01
Input 0, transition to S_1 giving an output 01
Input 1, transition to S_2 giving an output 00
Input 0, transition to S_1 giving an output 10
Input 0, transition to S_0 giving an output 11

The output sequence is therefore $\boldsymbol{v} = (11, 01, 01, 00. 10, 11)$ □

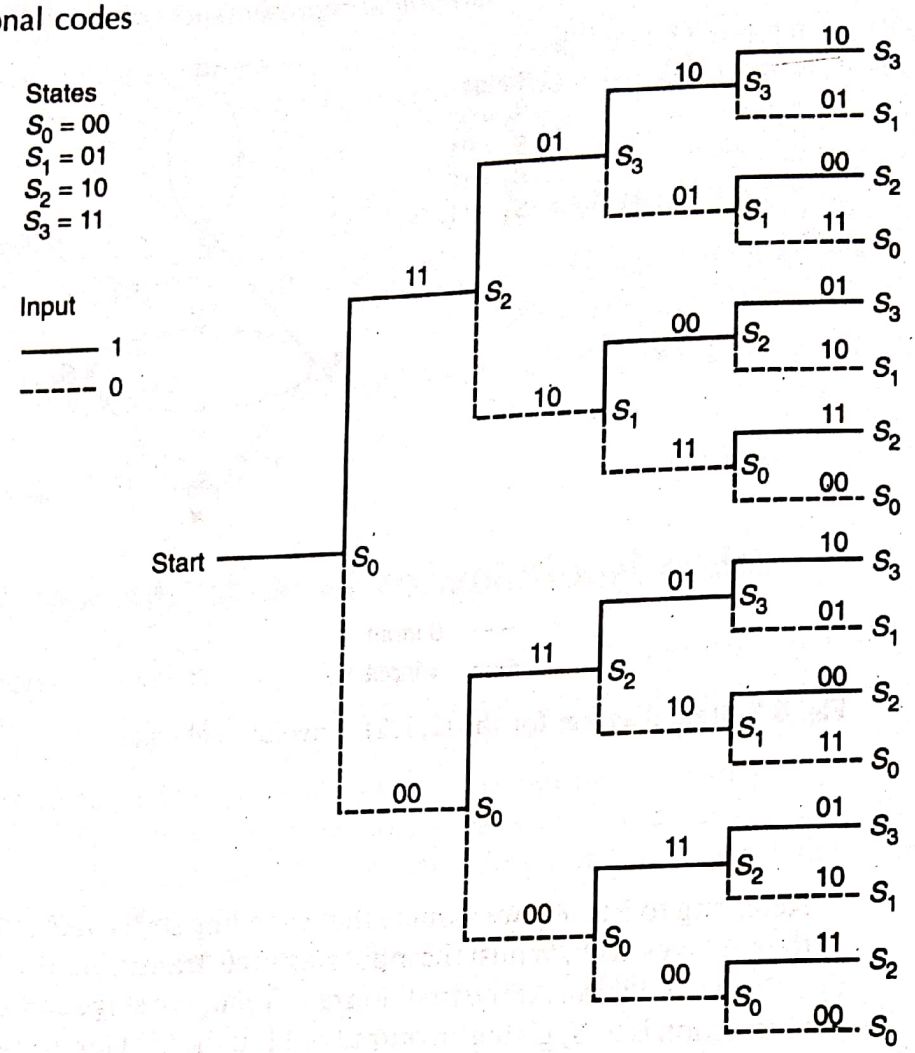


Fig. 8.8 Tree diagram for the (2, 1, 2) convolutional code.

Figure 8.8 shows a *tree diagram* for the (2, 1, 2) code. This again shows transitions from state to state, with corresponding outputs, for all inputs. The state and tree diagrams differ in that the latter shows the evolution of encoding with time. In the state diagram there is no way of knowing how a state was arrived at, from any specified state it is only possible to determine the next state and output for a given input. In the tree diagram moving from left to right represents the flow of time as the input sequence enters the encoder. At any node transitions can only occur to one of the two nodes to the right of the node. The output at each transition is shown above the horizontal portion of each branch. Encoding starts at the far left at S_0 and progresses from left to right. The tree diagram shown in Fig. 8.8 has been truncated after the 4th input, but theoretically the tree extends to the right infinitely.

A tree diagram gives an interesting graphical view of a convolutional code but is of limited practical use due to its size. However tree diagrams have a repetitive structure that allow the diagrams to be simplified whilst maintaining the basic feature of showing how encoding progresses with time. The resulting diagrams are known as *trellis diagrams* and Fig. 8.9 shows the trellis diagram for the (2, 1, 2) code. The trellis

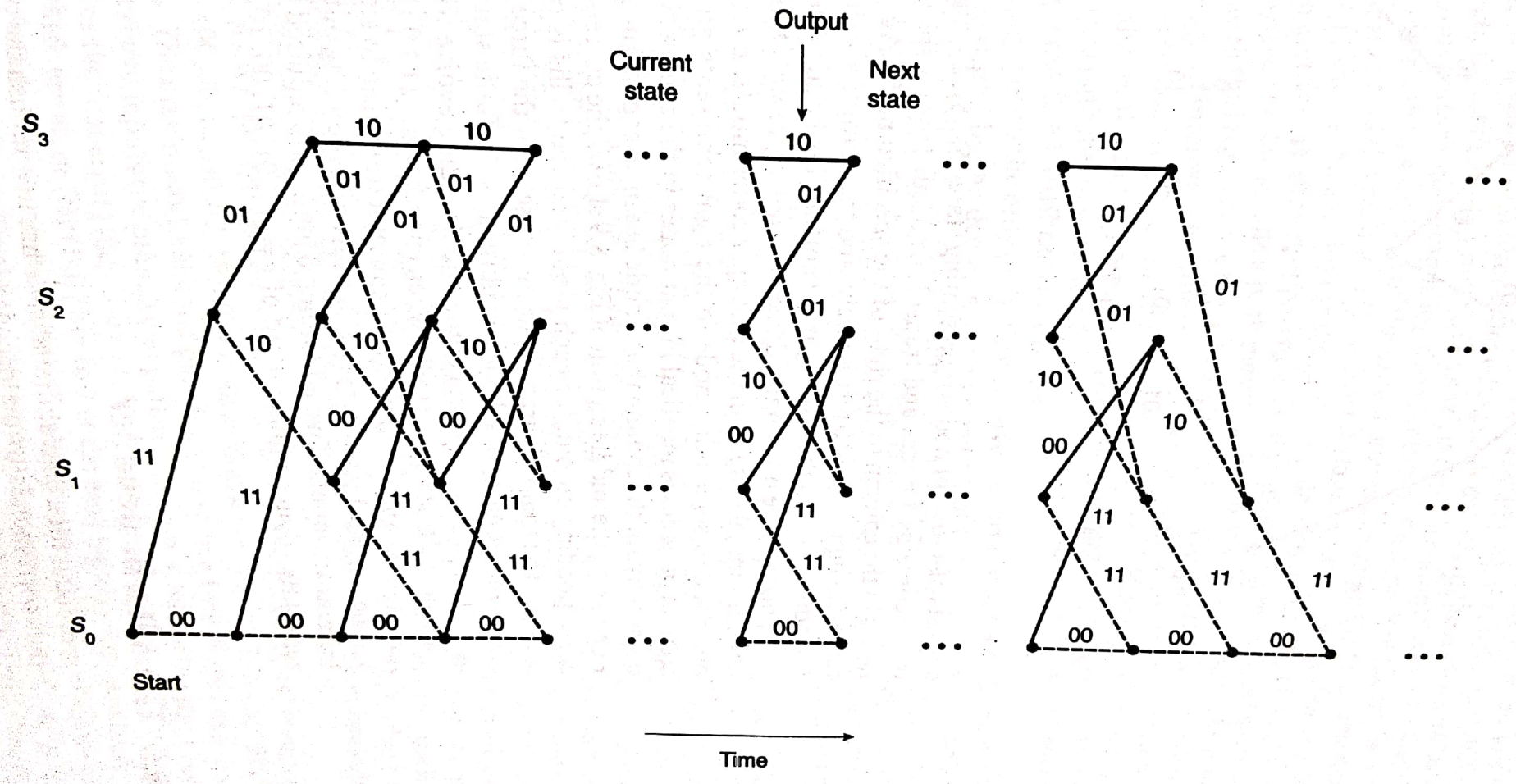


Fig. 8.9 Trellis diagram for the (2, 1, 2) convolutional code.

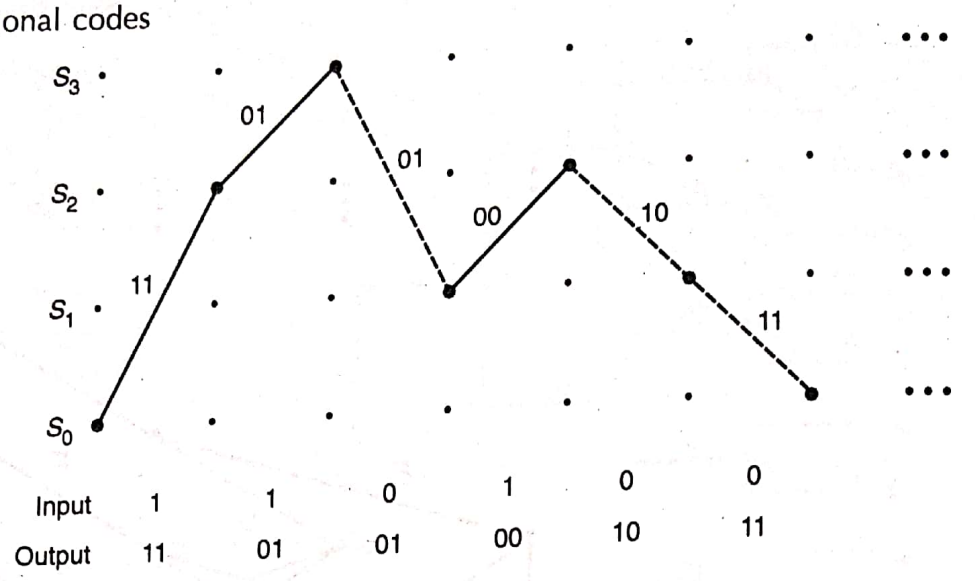


Fig. 8.10 Trellis when encoding $\boldsymbol{u}=(1\ 1\ 0\ 1)$.

consists of 4 rows of nodes representing states S_0, S_1, S_2, and S_3. Each column in the trellis contains 4 nodes and transitions occur from a specified node to one of two nodes in the column to the right of the specified node. Again solid and dashed lines represent inputs 1 and 0 respectively, and the outputs are shown next to each branch. Encoding starts at the state S_0 at the bottom left-hand node. On the left of the trellis, some nodes have been omitted because they represent states that cannot be reached from the initial state S_0 in the first m encoder inputs. Every path in the trellis, from one node to some other node to the right represents an encoded sequence. Each column of nodes is identical in that nodes in the same row have the same branches irrespective of the column that a node lies in, except at the start and end of the trellis. For clarity, the middle section of Fig. 8.9 shows the current state and next state, with corresponding output, for each input, and the trellis is obtained by repeating this section. The trellis diagram is the most useful of the three graphical representations of convolutional codes. It clearly illustrates the finite states of a convolutional code as well as showing how encoding and decoding evolve with time.

Figure 8.10 shows the trellis when encoding $\boldsymbol{u}=(1\ 1\ 0\ 1)$. The encoder starts at the zero state S_0 and the first input 1 causes a transition to state S_2 and an output of 11. The next input 1 gives the state S_3 with corresponding output 01 and so forth for the 3rd and 4th inputs. Two additional 0 inputs are required to return the encoder to its zero state, giving a final output of $\boldsymbol{v}=(11, 01, 01, 00, 10, 11)$.

8.6 The Viterbi decoder

A decoder for a convolutional code has to establish the path through the code trellis that corresponds to the decoder input $\boldsymbol{w}$. The Viterbi algorithm is a maximum-likelihood decoder that determines the path that $\boldsymbol{w}$ most likely corresponds to. This path is then taken to be the encoder output $\boldsymbol{v}$, from which the encoder input $\boldsymbol{u}$ can be

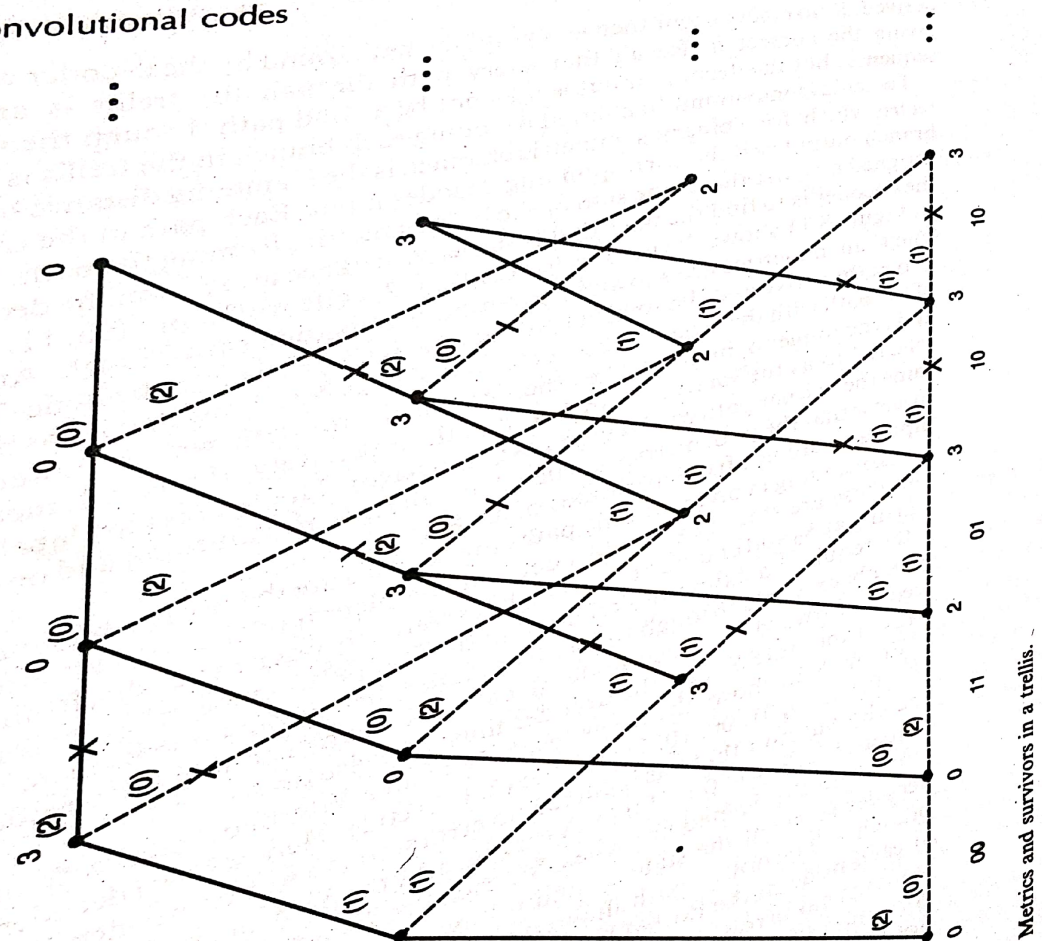
derived. If no errors occur then $w = v$ and the path found by the decoder will be v, so giving the correct u. Recall that every path through the trellis is an encoded sequence, but the decoder input w may not be a valid path through the trellis.

To achieve maximum-likelihood decoding each branch in the trellis is assigned a *metric* which, for a binary-symmetric channel, is the Hamming distance between the branch output and the corresponding decoder input. Each path in the trellis is also assigned a metric that is the sum of the branch metrics forming the path. The aim of the decoder is to find the path that is the least distance away from the decoder input w. Figure 8.11 shows the metrics for the (2, 1, 2) code with input (00, 11, 01, 10, 10). The branch metrics are shown in parenthesis alongside each branch. At each node two paths arrive and the smallest path metric is shown next to the node. The branch of the path with the larger metric is marked with an **X** to show that it has been deleted and is no longer of interest. The other branch and the path, with the lower metric, are referred to as the *survivor* branch and path respectively. If the path metrics are the same then either path can be taken as the survivor. The node is then labelled with a **T** to show that the path metrics are tied. Note that metric values depend on the decoder input and are not fixed attributes of a trellis.

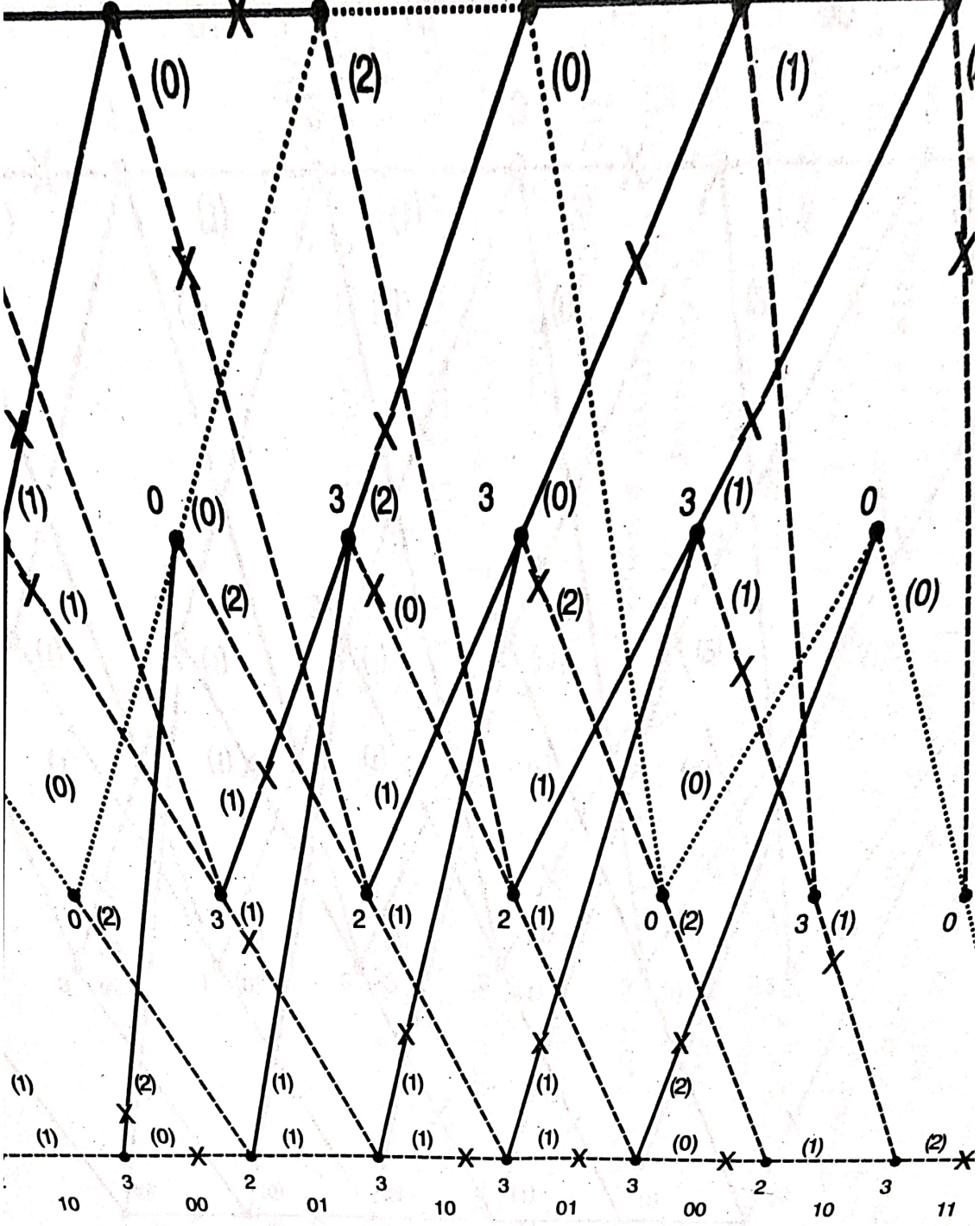
As decoding evolves the only paths that survive are those with the smallest metrics and these are stored by the decoder. On completion of the decoder input, the path starting at S_0 and ending at S_0 going backwards through the trellis, from right to left, is the required maximum-likelihood path. This path can be uniquely followed when going backwards through the trellis because there is only one branch, the survivor, feeding forward into each node. Recall that when encoding the encoder starts and ends at the state S_0 and the decoder must likewise do so.

Figure 8.12 shows the trellis resulting from a Viterbi decoder with input $w = (11, 10, 00, 01, 10, 01, 00, 10, 11)$ for the (2, 1, 2) code. Here w is the decoder input for an encoder output v that has incurred no errors, so $v = w$. Furthermore v is the encoded sequence for $u = (1\,0\,1\,1\,1\,0\,1)$. As in Fig. 8.11 the path and branch metrics are shown along with the deleted branches, for each decoder input. At the end of the input the decoder starts at the state S_0 and follows the path backwards along the surviving branches. The dotted path shows the resulting maximum-likelihood path. Note that at each node in the dotted path the path metric is zero, this is to be expected as w contains no errors. To determine the decoder output, that the constructed path corresponds to, we need to know the decoder output at each branch, for clarity this is not shown in Fig. 8.12. However, referring to Fig. 8.9 we find that the path in Fig. 8.12 gives $v = (11, 10, 00, 01, 10, 01, 00, 10, 11)$ and $u = (1\,0\,1\,1\,1\,0\,1)$, where the last two 0s of u (for returning the encoder to its zero state) have been excluded. Decoding has therefore been successful.

Figure 8.13 shows the trellis when $v = (11, 10, 00, 01, 10, 01, 00, 10, 11)$ incurs the 2-bit error $e = (00, 00, 01, 00, 00, 10, 00, 00, 00)$ so that $w = v + e = (11, 10, 01, 01, 10, 11, 00, 10, 11)$. The maximum-likelihood path is again shown by a dotted line. Note that for the first two inputs the branch and path metrics are 0, but on the third input (which contains the first error) they rise to 1. The path metric stays at 1 for the next two inputs (which are error free) but rises to 2 at the next input (which contains the second error). As there are no more errors the final path metric is 2, which means that all other paths through the trellis are at a distance of 2 or greater away from w. The dotted path shown in Fig. 8.13 is the correct path for v (see Fig. 8.12) and therefore decoding has been successful, despite the incurred errors.



Metrics and survivors in a trellis.



oder input with no errors.

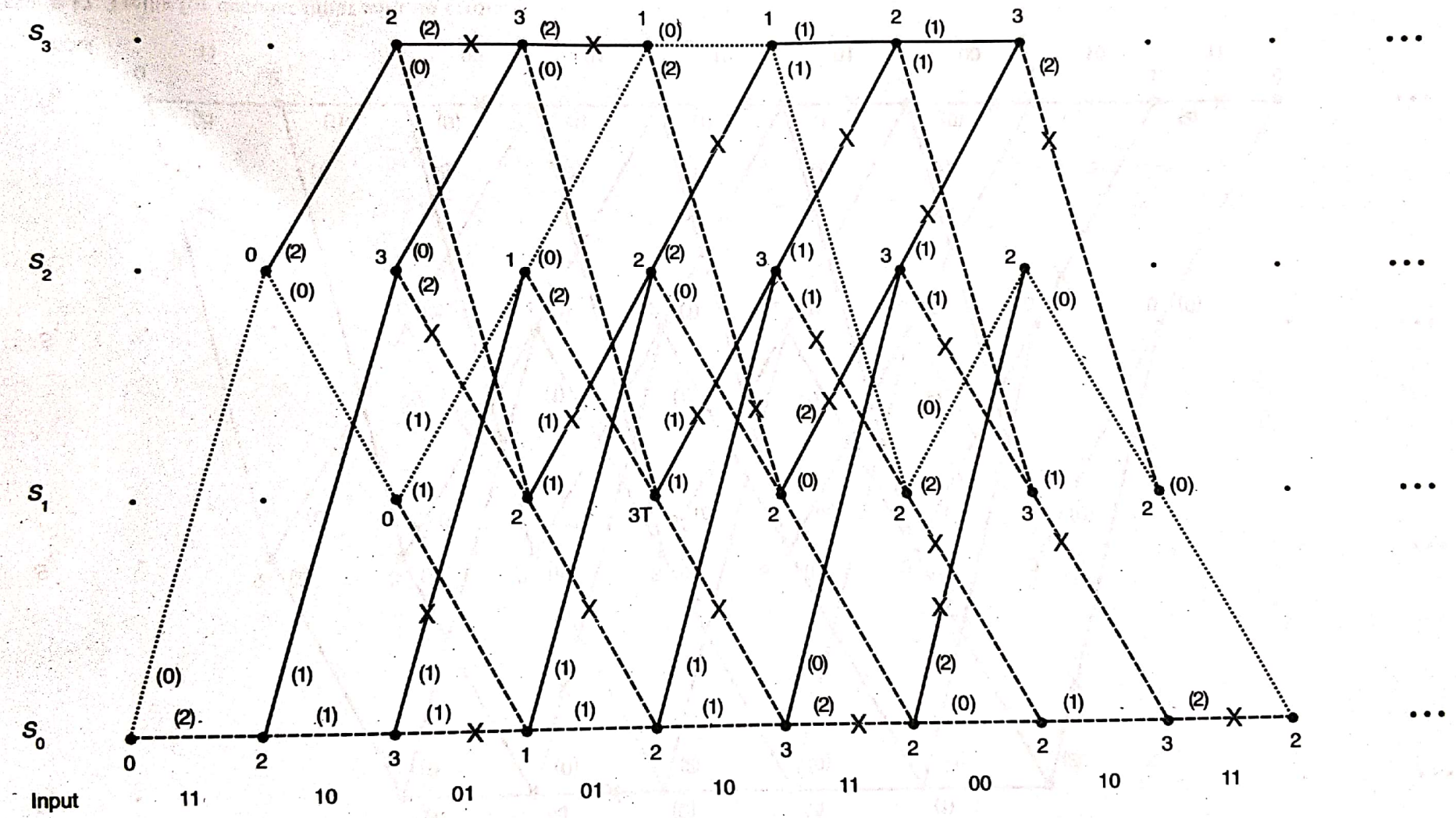


Fig. 8.13 Trellis for decoder input with 2 errors.