Convolution Codes

1.0 Prologue: Convolutional codes, why should complicate our lives with them

People used to send voice waveforms in electrical form over a twisted pair of wires. These telephone voice signals had a bandwidth of 4KHz. If the channel polluted the signal with a bit of noise, the only thing that happened was that the conversation got a bit noisier. As technology developed, we digitized the voice signals at 8000 samples per second (twice the highest frequency to prevent aliasing) and transmitted the bits. If noise corrupted a few bits, the corresponding sample value(s) would be slightly wrong or very wrong depending on whether the bad bits were near the most-significant-bit or least-significant-bit. The conversation sounded noisier, but were still discernible. Someone saying “cat” will not be thought to have said “dog,” and probably would not even be thought to have said “caught.”

When people started to send data files rather than voice, corrupted bits became more important. Even one wrong bit could prevent a program from running properly. Say the noise in a channel was low enough for the probability of a bad bit to be $1x10^{-9}$ i.e. the chances of a bit being correct is 0.999999999 (nine 9’s). The chances of 1000 bits being all correct is 0.999999 (six 9’s) and the chances of 10^6 bits being all correct is 0.999 (three 9’s). A 1 megabyte file ($8x10^6$ bits) has almost a 1% chance of being corrupted. The reliability of the channel had to be improved.

The probability of error can be reduced by transmitting more bits than needed to represent the information being sent, and *convolving* each bit with neighbouring bits so that if one transmitted bit got corrupted, enough information is carried by the neighbouring bits to estimate what the corrupted bit was. This approach of transforming a number of *information* bits into a larger number of *transmitted* bits is called *channel coding*, and the particular approach of convolving the bits to distribute the information is referred to as *convolution coding*. The ratio of information bits to transmitted bits is the *code rate* (less than 1) and the number of information bits over which the convolution takes place is the *constraint length*.

For example, suppose you *channel encoded* a message using a *convolution code*. Suppose you transmitted 2 bits for every information bit (*code rate*=0.5) and used a *constraint length* of 3. Then the coder would send out 16 bits for every 8 bits of input, and each output pair would depend on the present and the past 2 input bits (*constraint length* =3). The output would come out at twice the input speed.

Since information about each input bit is spread out over 6 transmitted bits, one can usually reconstruct the correct input even with several transmission errors.

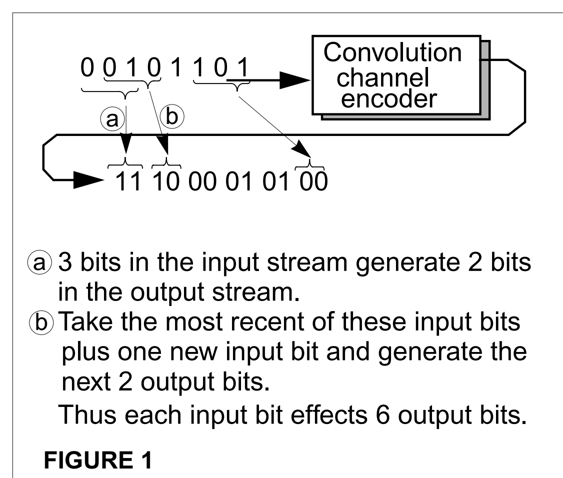


FIGURE 1

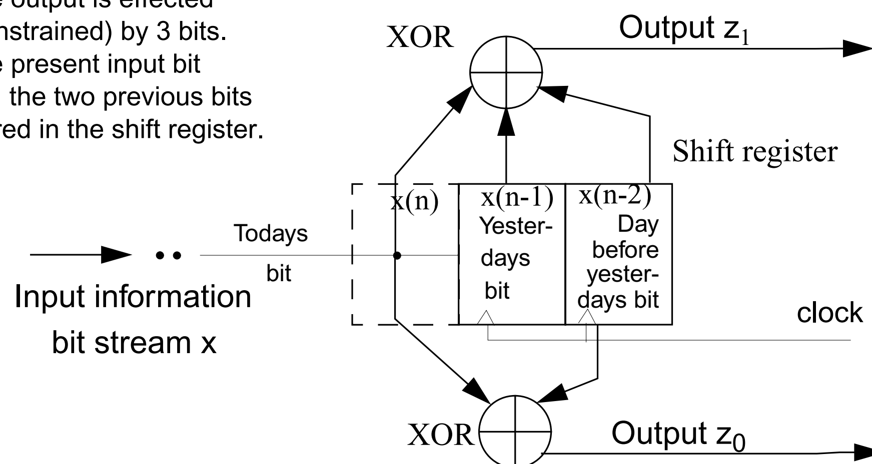
The need for coding is very important in the use of cellular phones. In this case, the “channel” is the propagation of radio waves between your cell phone and the base station. Just by turning your head while talking on the phone, you could suddenly block out a large portion of the transmitted signal. If you tried to keep your head still, a passing bus could change the pattern of

bouncing radio waves arriving at your phone so that they add destructively, again giving a poor signal. In both cases, the SNR suddenly drops deeply and the bit error rate goes up dramatically. So the cellular environment is extremely unreliable. If you didn't have lots of redundancy in the transmitted bits to boost reliability, chances are that digital cell phones would not be the success they are today. As an example, the first digital cell system, Digital Advance Mobile Phone Service (D-AMPS) used convolution coding of rate 1/2 (i.e. double the information bit rate) and constraint length of 6. Current CDMA-based cell phones use spread-spectrum to combat the unreliably of the air interface, but still use convolution coding of rate 1/2 in the downlink and 1/3 in the uplink (constraint length 9). What CDMA is, is not part of this lab. You can ask the TA if you are curious.

2.0 Example of Convolution Encoding

FIGURE 2

The constraint length is 3.
The output is effected
(constrained) by 3 bits.
The present input bit
and the two previous bits
stored in the shift register.



This is a convolution encoder of *code rate* 1/2 This means there are two output bits for each input bit. Here the output bits are transmitted one after another, two per clock cycle.

The output $z_1 = x(n) \oplus x(n-1) \oplus x(n-2)$.

Here x(n) is the present input bit, x(n-1) was the previous (yesterdays) bit, etc.

The output $z_0 = x(n) \oplus x(n-2)$.

The input connections to the XORs can be written as binary vectors Gz_1=[1 1 1] and Gz_0=[1 0 1] are known as the *generating vectors* or *generating polynomials* for the code. They indicate where the taps are drawn off the register into each XOR gate.

2.1 The Encoder as a Finite-State Machine

The correlation encoder can be described as a Mealy machine. The state is the two bits in the shift register.

Let the first input bit to the shift register be x(n) = 1, and let the flip-flops be reset to zero so x(n-1)=x(n-2)=0 . Then:-

State= 00 = S_{00} = {x(n-1),x(n-2)}
Output z={z_1,z_0}
$z_1 = x(n-2) \oplus x(n-1) \oplus x(n)$
$= 1 \oplus 0 \oplus 0 = 1$
$z_0 = x(n-2) \oplus x(n)$
$= 1 \oplus 0 = 1$
z={z_1,z_0}= 11

After the clock, state bit x(n-1)=0 will shift right into x(n-2), the input x(n)=1 will shift left into x(n-1), and the next state will be 01 = S_{10}.

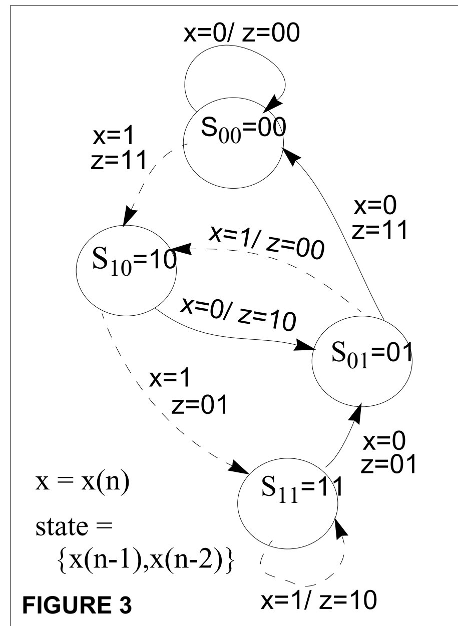


FIGURE 3

2.2 The Trellis Encoding Diagram

To get the trellis diagram, squash the state diagram so S_{00}, S_{10}, S_{01} and S_{11} are in a vertical line. This line represents the possible states at time t=n (now). Make time the horizontal axis. Put in another line of states to show the possible states at t=n+1.

Then add the transitions to the diagram. Make the them all go from states @ t=n to states @ t=n+1. Thus the self loop at state S_{00} in the state graph becomes the horizontal line from S_{00}@t=n to S_{00} @t=n+1 in the trellis diagram.

The complete trellis diagram extends past t=1 to as many time steps as are needed.

Suppose the encoder is reset to state S_{00}=00, and the input is 1,0,1,1,0,0. By following the trellis one sees that

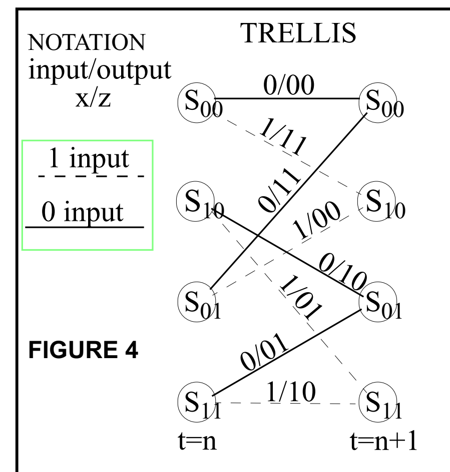


FIGURE 4

the output is 11 10 00 01 01 11. Also it passes through states S_{00}, S_{10}, S_{01}, S_{10}, S_{11}, S_{01} ending in S_{00} @ t=6.

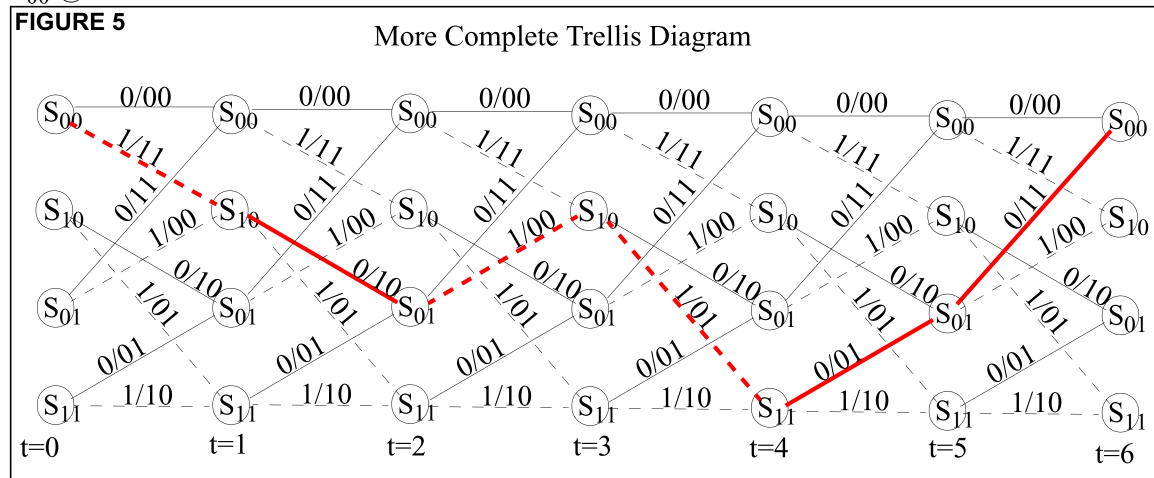


FIGURE 5 More Complete Trellis Diagram

2.3 Lab and Problem Rules

The Convolution encoder/Viterbi decoder design problem w**ill be done by a groups of three persons.**

The number of exercises is (usually) divisible by three so one person in each group can do every third problem and thus do one-third of the exercises. The three are to be submitted together with the name of the person doing each part attached to the part.

Five marks will be assigned for each persons questions and will be given to the person in the group answering the question. Two person groups should take turns answering the odd questions which will be assigned three marks. This applys to temporary two person groups, groups where one member goofs off. Another member can get three extra marks by doing his/her questions.

One common mark will be assigned to coordination within the group. Do they use common symbols? Do they hand the assignments in at the same time attached together? Do they refer to the other questions where appropriate? Violation of any one of the above may cost each group member his/her common mark.

All members of the group are responsible for knowing how to do each exercise. Related problems will appear on examinations.

The problems and labs have subtle, and also not so subtle, changes from last year. One way to lose marks quickly is to submit answers taken from last year. The penalty will be zero for the question(s) invoved and a 75% reduction in the mark of the whole group.

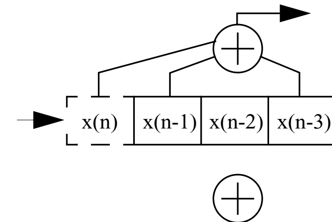
2.4 First Exercise: A Convolution Encoder.

1. Problem: Encoding a number
   - Take the last 4 digits in your student (the least significant digits).
   - Convert them to a hexadecimal number (Matlab has a function dec2hex).
   - Convert the hexadecimal number to binary (12 to 16 bits).
   - Use this as data for the encoder below. Feed in the least significant bit first. Also reset the shift register to 00 before you start.

- Calculate the output bits and states when one encodes these bits using a *code rate* 1/2, *constraint length* 3 encoder with *generating vectors* [111] and [101]. Tabulate how the state and output values change with each clock cycle.

Clock cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
input bit																
shift reg (state)	00															
Output [z_1,z_0]																

2. Problem: Draw the circuit for an encoder which has:
   a *code rate* =1/2,
   constraint length of 4,
   generating vectors Gz_1=[1101] and Gz_0[1111], where the "0" means no connection to x(n-2).



3. Problem: Draw the state graph for the above constraint length 4 encoder. Draw the first 3 time steps of the trellis diagram for the above constraint length 4 encoder.

2.5 First Lab: Design a Convolution Encoder in Verilog

The constraint-length 4 encoder circuit can be coded as a finite-state machine or as a shift register.

```
reg  [2:0]  state;  // (shift-reg length)=(constraint length -1)
    always @(posedge clk ... ) begin
        if(rst) ...
        else ... state <=state >> 1; // Right shift 1 position
        end
```

The test bench is not part of the circuit. It supplies the input signals and may compare output signals with those that are expected. Test benches are easier to write and use if they are *synchronous*. This means they always send out signals slightly after the clock (**or at least never at the same time as the clock**). It also means a writing style with few #n delay.

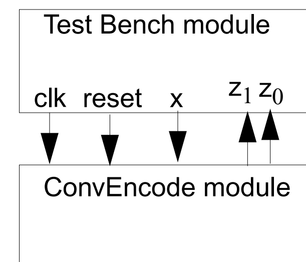


FIGURE 6

```
    initial begin
. . .   #11  x=1;
        #10  x=0;
        #10  x=1;
        #20  x=0;
          . . .
```

Poor

2.5.1 A Synchronous Test Bench

In a synchronous test bench, the test signals are timed by @(posedge clk) statements rather than each having its own timing. There are only two delays here, one to set the clock period, and the other to delay the input signal x so it's changes are obviously past the clock edge.

Note there are things not included here, like reset.

```
module SyncTestBench;
    reg [7:0] data;  //Fill this with the data stream to be encoded.
                     // Note the first bit to go out is on the right
```

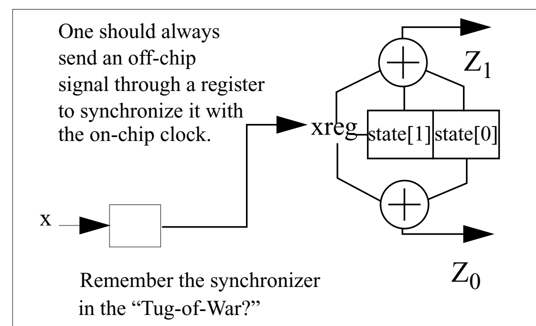
```
reg x, clk;
integer I,Cntr;   // Use integers for test-bench counter indexes
initial
  begin
    I=0;
    data=8'b10101101; // Underscore has no meaning except
                              // to visually space the bits.
    clk=0;
    forever  #5 clk=~clk; end
  end
// send in a new value of x every clock cycle
  always @(posedge clk)
       begin
            if (I==8) $finish; // Stop the simulation when one runs out of data.
  // The #1 makes x change 1ns after the clock and never on the clock edge.
  // The nonblocking symbol "<=" on I ensures that any other clocked module using
  // I will grab the same I as this procedure, that is before I is updated to I+1.
            x<= #1 data[I];
            I<=I+1;
       end
  endmodule
```

For the constraint length 3 system, you must have the test bench automatically compare your answer with the result you obtained from your student number.

1. Write a finite-state machine encoder for the *constraint length* 3 system. Draw the simplified circuit diagram of the FSM implementation.

```
always @( state or  xreg)
     case(state)
     2'b00: nxtstate =...;
     ...
     endcase

 always @(posedge clk or negedge rst)
 // Most library flip-flops are negedge
if(!rst) state <= 0;
...
state <=nxtstate;
```



2. Write a shift-register based encoder for the *constraint length* 4 system (Sect 2.3 prob 2). Generating vectors Gz_1=[1101] and Gz_0=[1111]. Draw a simplified circuit diagram for the shift-register based implementation. Note that even though the circuit performs the same ultimate function, the diagrams questions 1 and 2 should look quite different.

```
always @(posedge clk  or negedge reset)
  . . .
  state <=state>> 1; // right shift 1 position
  state[2] <= xreg;  // Overwrite  the zero shifted in on the previous line.
```

3. Write a synchronous test bench so the two encoder modules can be simulated. Have the test bench automatically compare your answer with the desired result
   a) Check the constraint-length 3 encoder using the data $00001101001011_{\text{first-bit}}$
   b) Check the constraint-length 4 encoder with the same data.

Ans: Constraint length 3 encoded data-
11,01,01,00,10,11,11,10,00,01,01,11,00,00,00,00.
Ans: Constraint length 4 encoded data-
11,00,10,01,00,01,00,11,10,11,10,10,11,00,00,00,00,

You must run the test-bench against your encoders and include the self-checking log file and waveform in the report.

4. When you design the decoder, you will probably find you need a clock six times faster than that used for the encoder. In this situation, you should use enable signals that do not have the critical timing needed for a second clock. Rewrite the *constraint length* 3 decoder to allow shifting only when enabled. Make the necessary changes to the simplified hardware diagram.

```
always @(posedge clk  or  negedge reset)
  . . .
 if(shift_en)
     begin state <=state>> 1; // Right shift 1 position
          state[1] <= xreg;  // Overwrite  the zero shifted in on the previous line.
     end
```

5. Generate a signal in the encoder module that pulses every 6 clock cycles to use as shift_en. Draw the simplified logic diagram of this counter circuit which produces the shift_en.

```
assign  shift_en = (cntr= =5);
assign (ctr==5) ? nxtcntr=0 : nxtcntr=cntr +1;

always @(posedge clk  . . .
  . . .
     cntr <=nxtcntr;
```

Also fix your test bench so it now only changes x every 6 clock cycles. Have the test bench drill down through the module hierarchy to get the *shift_en* signal.

```
assign shift_enTB = TopModule.Encoder.shift_en;
```

This gets the signal, for simulation only, without adding an unwanted pin in TopMod.

The output of the encoder should be serial, if it is to be transmitted. Generate a mux and its control signal, **Z_1select**, that will send out Z_1 for 3 clock cycles, half the length of the x input bit, and Z_0 for the next three clock cycles. This will allow two serial output bits for every input bit.

```
assign Z1select= (count= = 0 | count = = 1| count=2);
```

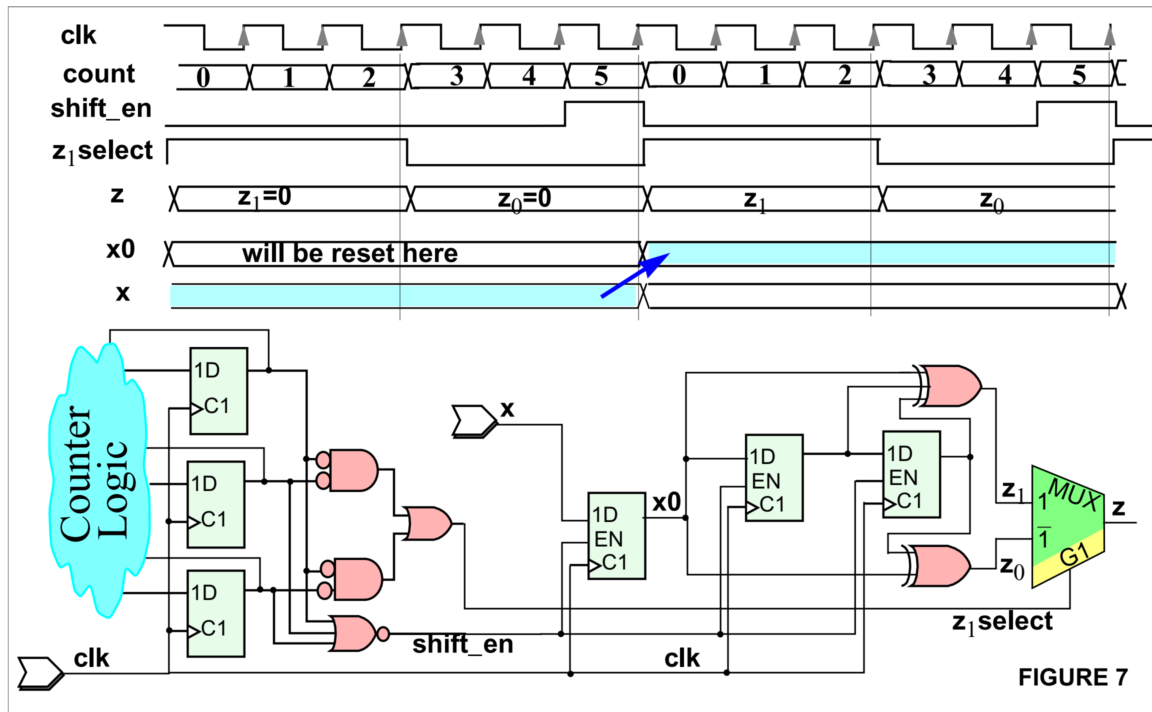


FIGURE 7

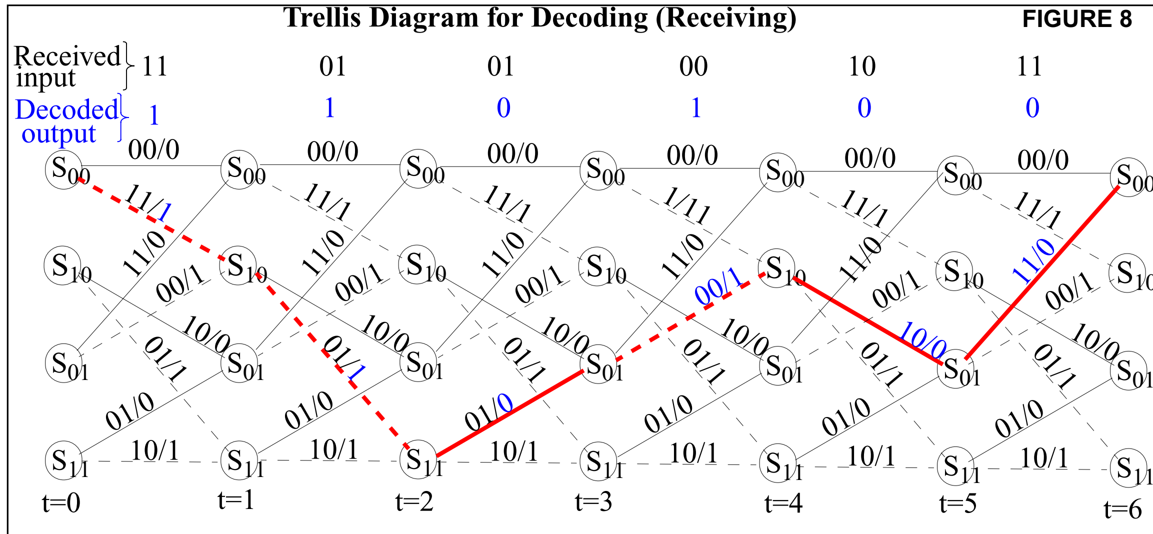
3.0 Convolution Decoder

The next part of the project will be to design a convolution decoder to retrieve the information bits from the transmitted bits. It should succeed even in the presence of some errors in the transmitted bits. The method we will use is called a *Viterbi decoder.*

3.1 Decoding Using the Trellis Diagram

Consider a decoder that receives the transmitted signal 11 01 01 00 10 11 going from t=0 to t=6. Assume the trellis was reset to state S_{00} (00) at the start. Knowing that we started in S00, if we receive 11 we know that a one was the original input and that we should now be in state S10. Similarly, if a 0 was the original input, we should get a 00 from the channel and stay in state 00. If there was an error on the channel, we would get either 01 or 10. In this case we aren't sure yet which state to go to, but we penalize each possibility and continue on our way. After enough bits are collected, we go back and look at the 'most likely sequence' of bits. This is how we get the error correction out of the circuit. One goes through the trellis as in the encoder, only for decoding the numbers written over the branches, the numbers are written as the decoder-input/decoded-output. Thus the first input, 11, gives a decoded output of 1 and takes the machine to state S_{10}.

At t=1 in state S_{10}, the next input 01 causes a 1 output and a change to state S_{11}.

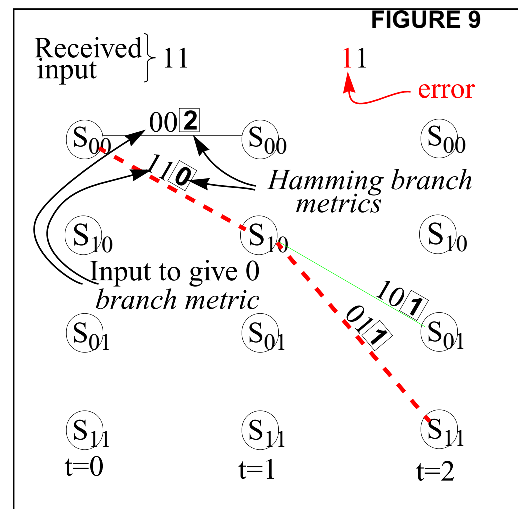


3.1.1 The Hamming Distance (Metric)

This distance is used to show how far apart two binary numbers are. Compare the bits in the same positions in the two numbers. The number of positions that are different is the Hamming distance (h).

Thus 11and 01 are distance 1 apart (h=1),
1001001 and 1001010 are distance 2 apart (h=2).

Very shortly we will call this Hamming distance 'h', the *Hamming branch metric* or *branch metric.*



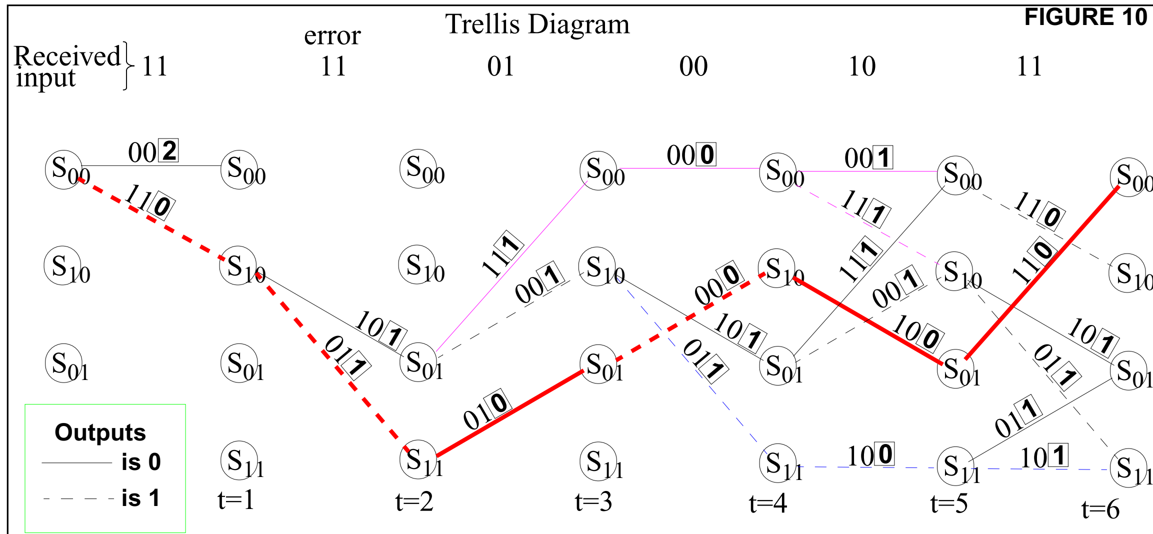
Applying the Hamming Distance to Decoding

Suppose the first four received bits have an error so instead of 11 01, one receives 11 11, as in Fig. 8

For the first input 11, there are two choices leaving state S_{00}, one for input 11**0** and the other for input 00**2**. The number in the box is the Hamming distance between the received input bits and the bits required for the state transition. It is clear one should make the transition from $S_{00} \Rightarrow S_{10}$.

The next input has an error. Note there are no 11 or 00 paths leaving state S_{10}. Both possible paths, 10 or 01, are at Hamming distance 1. At this time either transition looks equally likely, but wait!

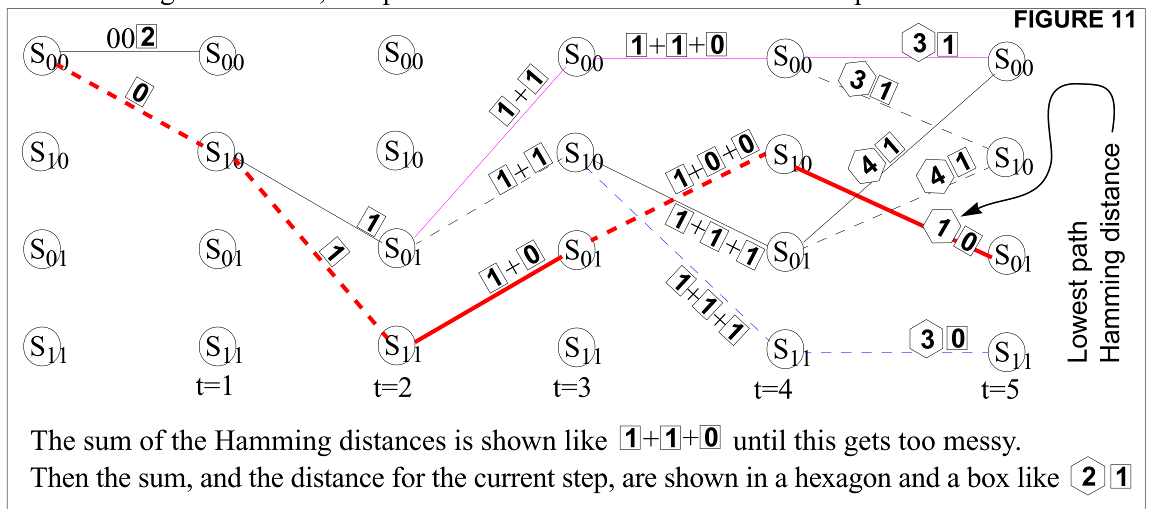
FIGURE 10



At t=2, if one starts from S_{11}, then h=0 for the (red) path to state S_{01}. However if one starts from S_{01} one has h=1 for either the path to S_{00} or to S_{10}.
Thus at t=1 the choice between the path $S_{10} \Rightarrow S_{01}$ and $S_{10} \Rightarrow S_{11}$ was not obvious. However at t=2, the choice is clearer. We should chose a path through the trellis based on the *path Hamming distance* or *path metric*, which is the sum of the Hamming distances as one steps along a path through the trellis.

Figure 11 shows how the Hamming distances sum as one goes down various paths through the trellis diagram. At t=5, one path has a total distance of 1 from the input data. The others have

FIGURE 11



The sum of the Hamming distances is shown like 1+1+0 until this gets too messy.
Then the sum, and the distance for the current step, are shown in a hexagon and a box like 2 1

a distance of 3 or 4. Thus the most likely path is $S_{00} \Rightarrow S_{10} \Rightarrow S_{11} \Rightarrow S_{01} \Rightarrow S_{10} \Rightarrow S_{01}$ with a *path distance* of 1, and the corresponding output data is 11010 (Recall ——— trellis edges represent a receiver output of 0, and - - - - - edges represent an output of 1).

3.1.2 Metrics

A metric is a measure of something. The more general name for what we called the *Hamming distance* is *branch metric*, and for the *path Hamming distance* is *path metric*. One does not have to use the the Hamming distance as a measure. In decoders where the input is an analog signal, the distance between the actual and expected voltage may be measured, and the sum of the squares of the errors might be used for the branch metric.

References:

Bernard Sklar, *Digital Communications Fundamentals and Applications*.

B.P.Lathe, *Modern Digital and Analog Communication Systems*, Holt, Rinehart & Winston, 1989

Prof. Ian Marsland, http://www.sce.carleton.ca/courses/94554/

Click on "Convolutional Codes." You will need Ghostview (gv) to read the Postscript file.

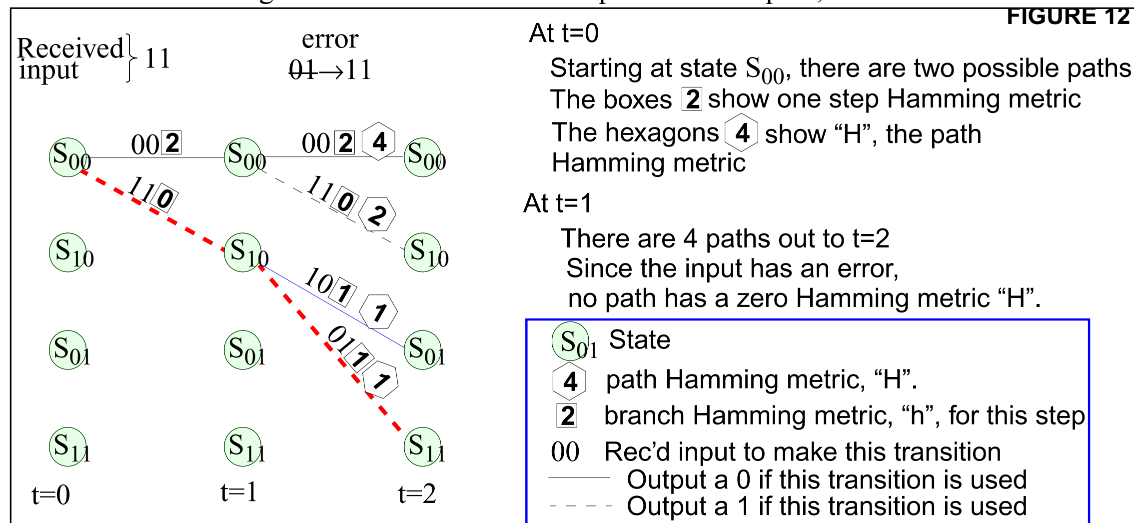
4.0 The Viterbi Decoder

The decoding example shown above has to follow four paths through the trellis, and remember them for future decisions. For larger decoders, such the cell phone ones with constraint lengths (shift-register lengths + 1) of 6 and 9, the number of paths can get quite large (32 and 256).

Viterbi developed an algorithm 1n 1967, which allows the many paths to be discarded without tracking them to completion. He noticed that if two paths merge to one state, only the one with the smaller *path Hamming metric*, “H,” need be remembered. The other one can never be part of the most likely path.

This means that with the *constraint length* 3 (shift-register length 2) system in the previous examples only have to remember 4 paths. In general a *constraint length* K system will have to remember 2^{K-1} paths. In theory, the path length should go for the length of the message in order to get the true maximum likelihood path. However it turns out that path lengths of 4 to 5 times the constraint length can almost always give the best path.

The next few figures show how the decoder picks the best path, even when there are errors.



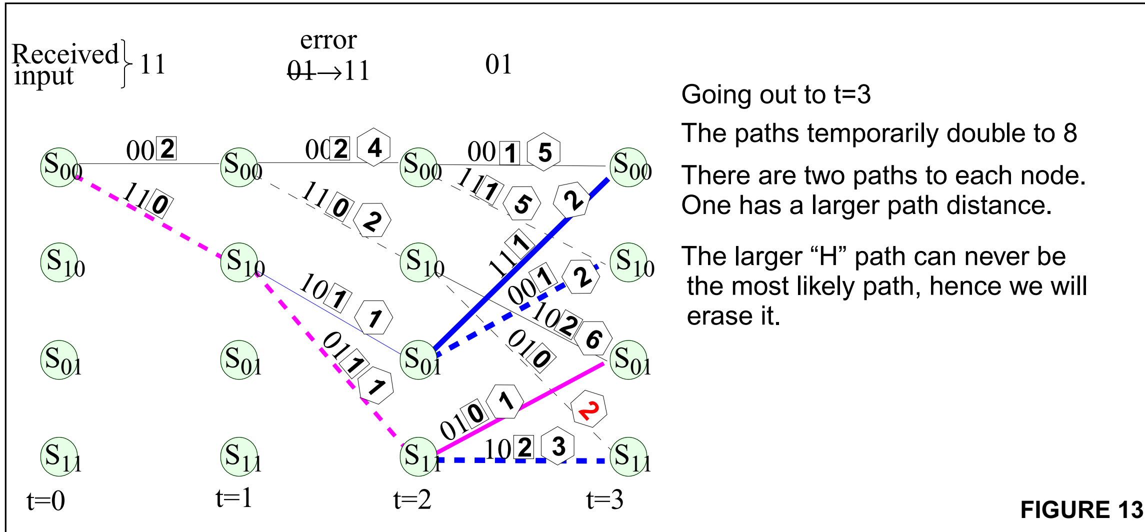


FIGURE 13

FIGURE 14

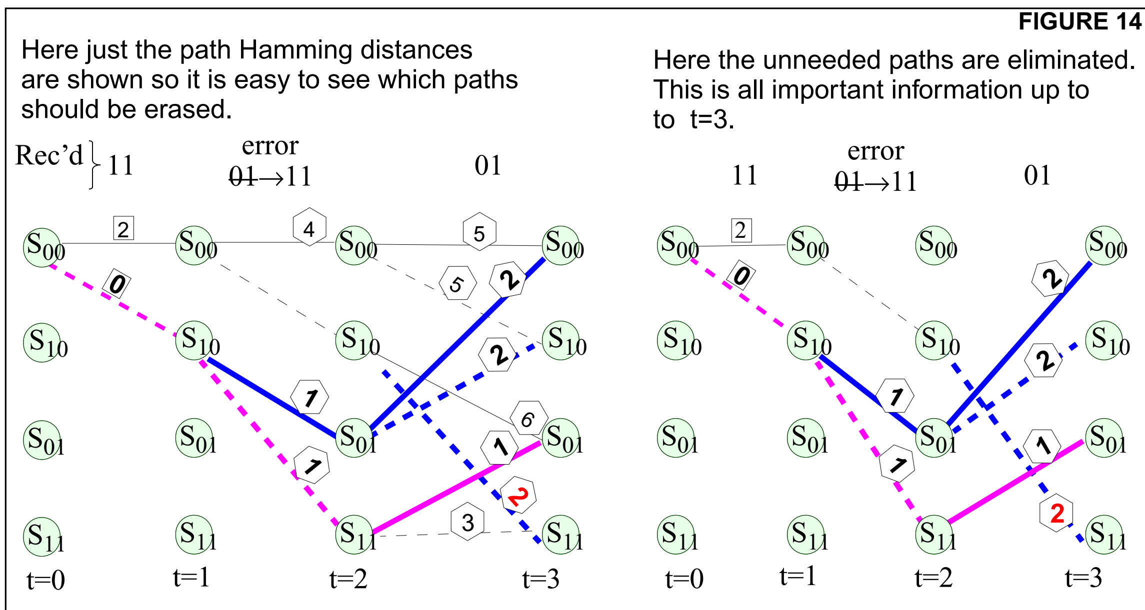
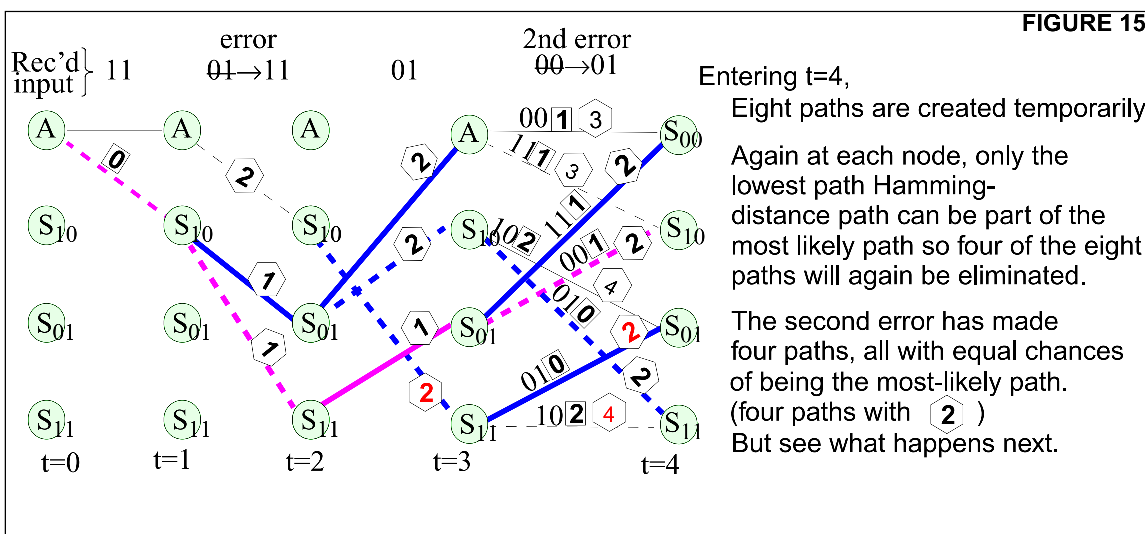


FIGURE 15



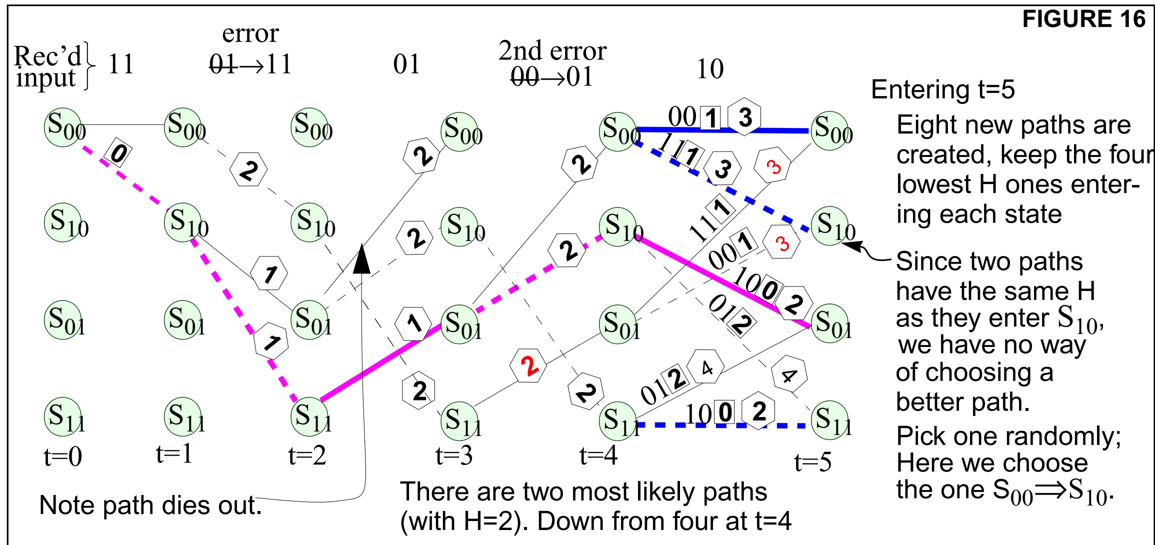
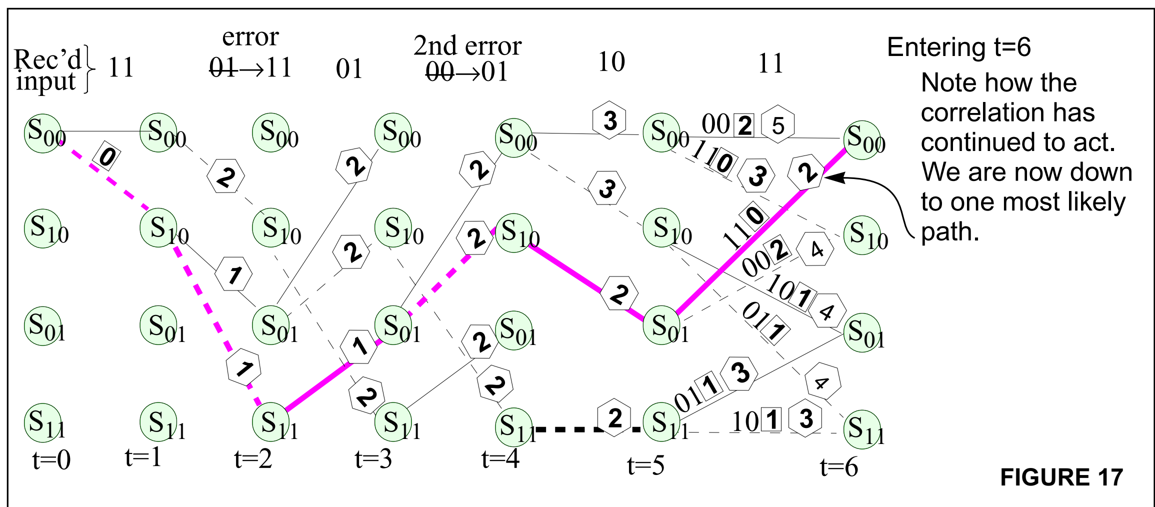


FIGURE 16
Rec'd input
11
error
01→11
01
2nd error
00→01
10
Entering t=5
Eight new paths are created, keep the four lowest H ones entering each state
Since two paths have the same H as they enter S10, we have no way of choosing a better path.
Pick one randomly; Here we choose the one S00⇒S10.
Note path dies out.
There are two most likely paths (with H=2). Down from four at t=4
t=0
t=1
t=2
t=3
t=4
t=5


Rec'd input
11
error
01→11
01
2nd error
00→01
10
11
Entering t=6
Note how the correlation has continued to act. We are now down to one most likely path.
t=0
t=1
t=2
t=3
t=4
t=5
t=6
FIGURE 17

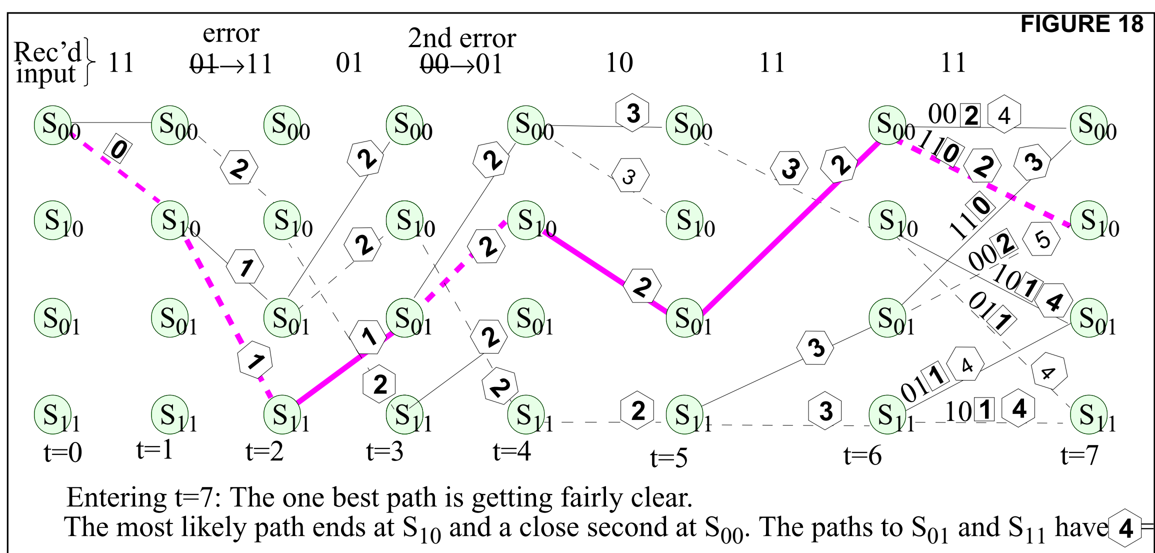
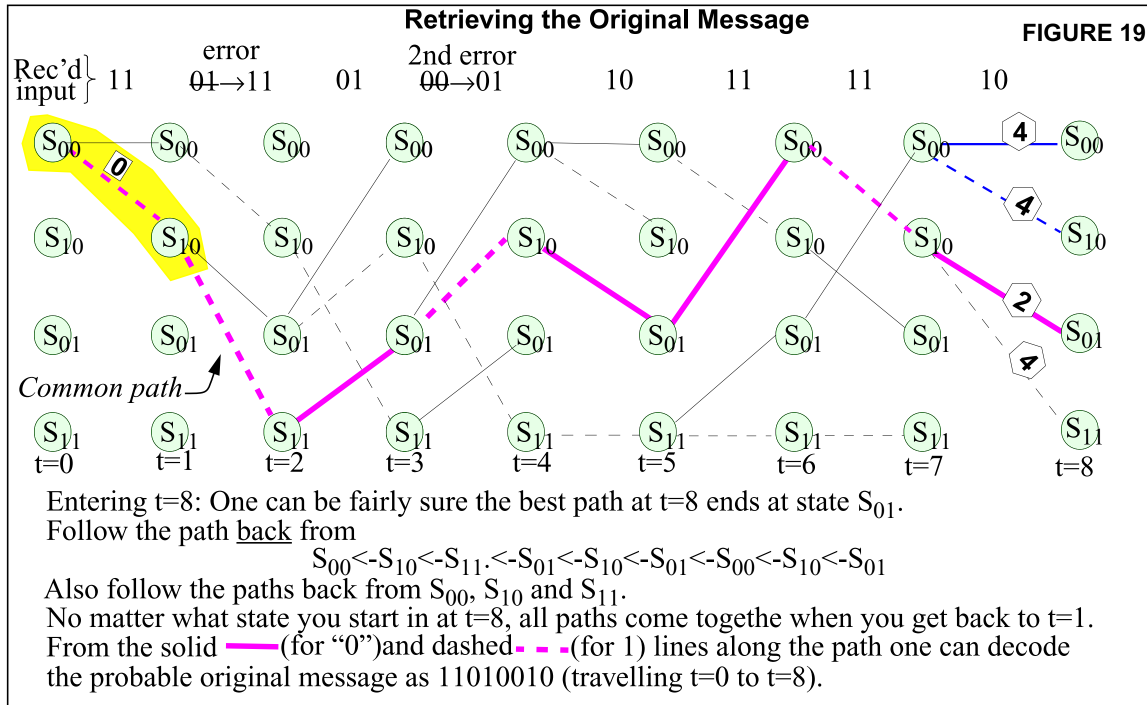


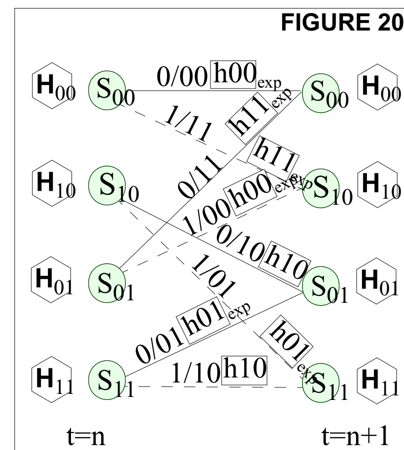
FIGURE 18
Rec'd input
11
error
01→11
01
2nd error
00→01
10
11
11
t=0
t=1
t=2
t=3
t=4
t=5
t=6
t=7
Entering t=7: The one best path is getting fairly clear.
The most likely path ends at S10 and a close second at S00. The paths to S01 and S11 have 4



As illustrated above, the Viterbi decoder can decode a convolution code in the presence of some errors.

If two branches entering a state have equal "H," then the code is unable to tell if one path is more likely than another. Pick one path at random.

4.1 Exercise 2: Add-Compare-Select Design



The circuit to add H+h, compare H+h on the two paths, and select the smaller path metric, is called the add-compare-select circuit.

Problem Prolog: Using the algorithm

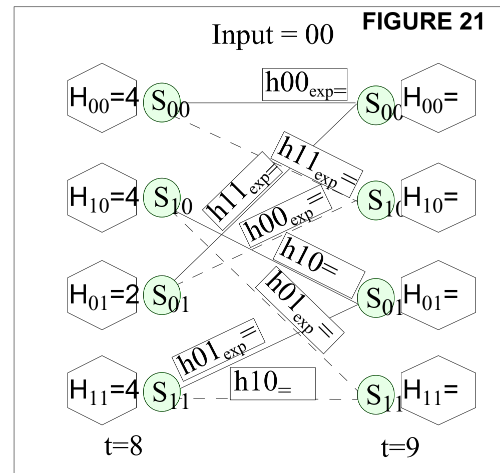
A typical step in the trellis decoder is shown.
The *path Hamming metrics* H at each trellis step are H_{00}, H_{10}, H_{01}, and H_{11}.
The *branch Hamming metric* h for each edge are given subscripts matching the input which makes them 0.
Thus the edge from S_{00} to S_{00} , and S_{01} to S_{10} both use the symbol $h00_{exp}$, meaning 00 is expected from the channel if this branch is taken. If the input is 00, $h00_{exp}$=0, If the input is 10 or 01, $h00_{exp}$=1, if the input is 11, $h00_{exp}$=2.

Pseudocode

This is Verilog in which the syntax is not critical. For example *begin, end* and semicolons may be omitted if the meaning is clear to the reader. In pseudocode the comments are often more important than the code.

1. Problem

a) Starting at t=8 with an input of 00, as in Figure 21, calculate and fill in the values of h_{ij} and hence the H_k for t=9.



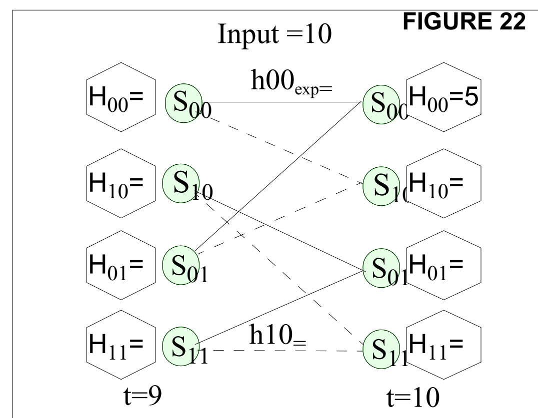
b) Write pseudocode to calculate the branch Hamming metrics for each step.
Let the two input bits z_1, z_0.
Let the Hamming metrics associated with the eight trellis edges for this step be $h00_{exp}$, $h01_{exp}$, etc.

Calculate these metrics using a case statement:

```
case ({z1, z0})
    2'b00 : begin h00exp=0;   ...   h11exp=2; end
    2'b01 : ...
```

2. **Problem**:

a) Starting at t=9, using the H_k from Prob 1, step a) and input 10, calculate and fill in Figure 22.

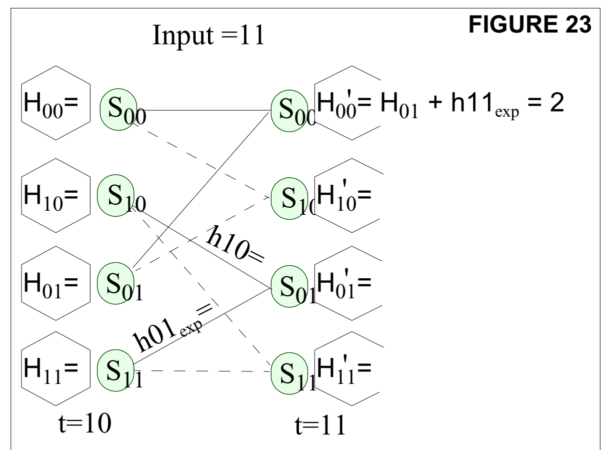


b) Use Boolean algebra to calculate them as 2-bit binary numbers. i.e 2=10, 1=01 and 0=00.
Use **reg /*wire*/** [1:0] $h00_{exp}$, $h01_{exp}$, h10, $h11_{exp}$;
Example:

$h00_{exp}[1] = y\&x$; $h00_{exp}[0] = y\hat{}x$;

3. Problem

a) Start at t=10, with input 11 and use the H_k from Prob. 2, step a). Let the new H_k at t=11 be written with a prime i.e. H_{00}' H_{10}' H_{01}' and H_{11}'. Fill in Figure 23 but put in an expression, as well as a number, for each H_k'. This has already been done for H_{00}'
Only the better path is written here.



b) Write pseudocode to update the H_k in going from step t=n to step t=n+1.

Use *if* statements to calculate the H_k' to be associated with the four states at t=n+1. Use H_{00}nxt instead of H_{00}' since Verilog cannot handle primes.

```
if (H00+h00exp < H01+h11exp) begin H00nxt= H00+h00exp; end   else ...
```

c) The flip-flop procedure.

Write a procedure to clock the flip-flops and replace the old Hs with the new ones. Combinational logic in parts 2 and 3 calculated the D inputs for the flip-flops. For example:-

```
always @(posedge clk
       H00 <= H00nxt ....
```

Don’t put combinational logic in a flip-flop procedure, and don’t forget a reset.

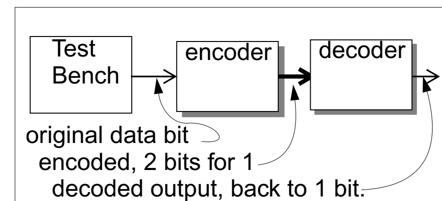
4. **Problem:** When is the output correct?

Experience has shown that all backward paths converge into one if one traces them back 4 or 5 times the *constraint length*. Using the paths in Figure 19, you will find that if one traces back far enough it does not matter which path one follows.

a) Take a copy of Figure 19. Start at t=8; start at each state in turn and colour backwards until you reach t=0 or until you hit previous colouring. At what time (t=?) do the paths all converge?

5. **Problem**: When is the output correct?

Look at Figure 15 in which the ending time is t=4.



a) Using data available at t=4 could you say, with confidence, what the original data bit was between t=0 and t=1? Why not?

b) Take a copy of Figure 18. Start at t=7; start at each state in turn and colour backwards until you reach t=0 or until you hit previous colouring. At what time (t=?) do the paths all converge?

c) Follow the trellis backwards, and from the information in the trellis find the most probable original data. Write out the message in the correct order with the earliest (t=0) bit on the left.

6. **Problem**: Finding the original data from the state.

a) The states can be placed in two sets, depending on their most-significant bit.
If one is in a state with the most-significant bit=0 (S_{00} or S_{01}), what was the original data in the previous step?
If one is in a state with the most-significant bit=1 (S_{10} or S_{11}), what was the original data in the previous step?
Recall that the state of the decoder mirrors the state of the encoder, which is a shift-register. Write pseudocode to send out the proper output bit based on the state during the trace back.

```
reg [1:0] state
if (state[1]) output= ...   // Make this more exact.
```

7. **Problem:** Use Figure 24 only.

a) If the decoder was in state S_{01} at t=3, what was the original data (before encoding) between t=2 and t=3? (The obvious answer is right.)
If it was in state S_{00} at t=3, what was the original data between t=2 and t=3?
If it was in state S_{10} at t=3, what was the original data between t=2 and t=3?
If it was in state S_{11} at t=3, what was the original data between t=2 and t=3?

b)

8. **Problem:** How to backtrack.

Figure 25 is the same as Figure 19 except the numbers are all removed. It still contains enough information to trace back from t=8.

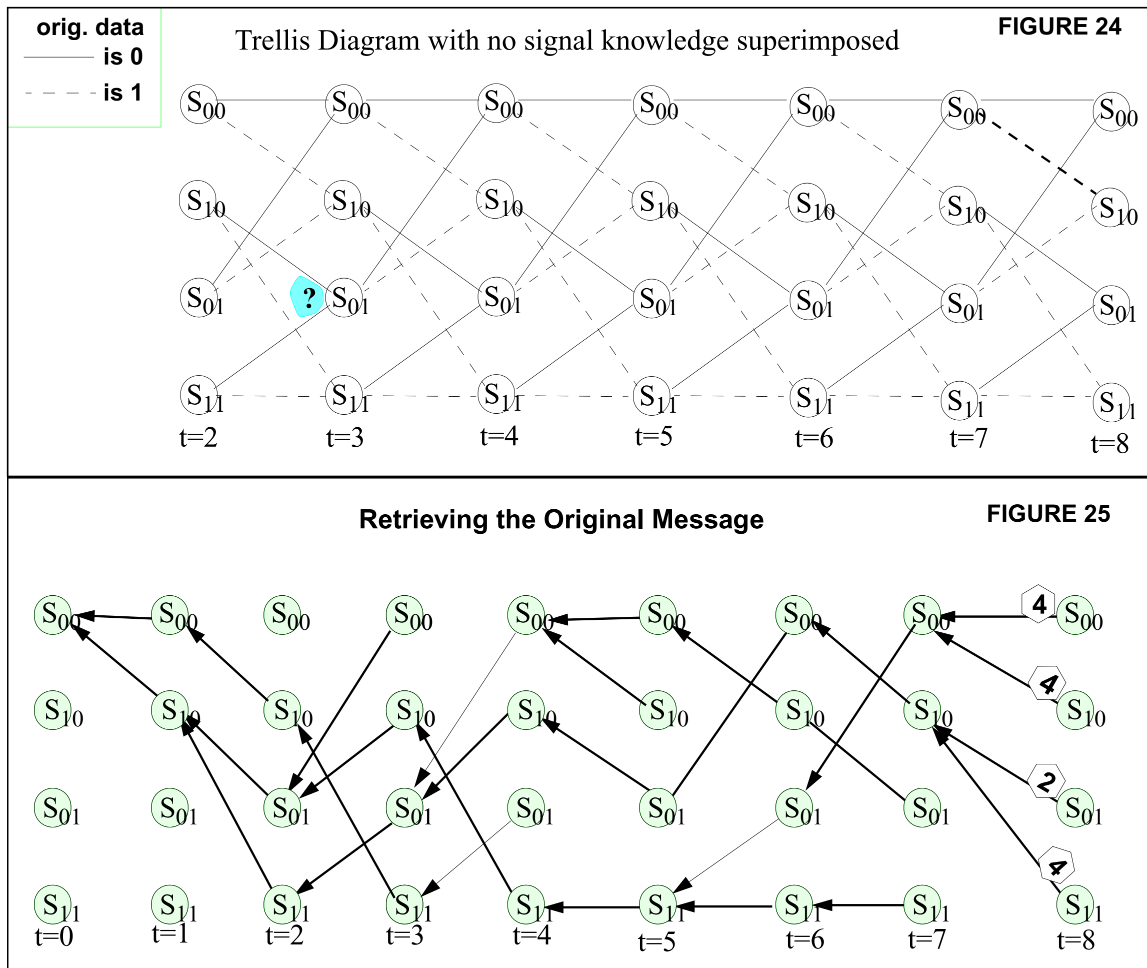


Figure 24 shows a trellis decoder only. It gives no information about the data. Figure 25 shows paths, but when you trace back to the area between t=2 and t=3 you cannot tell from the figure what the data was. However the those who did questions Prob: 6 or 7 can tell you.

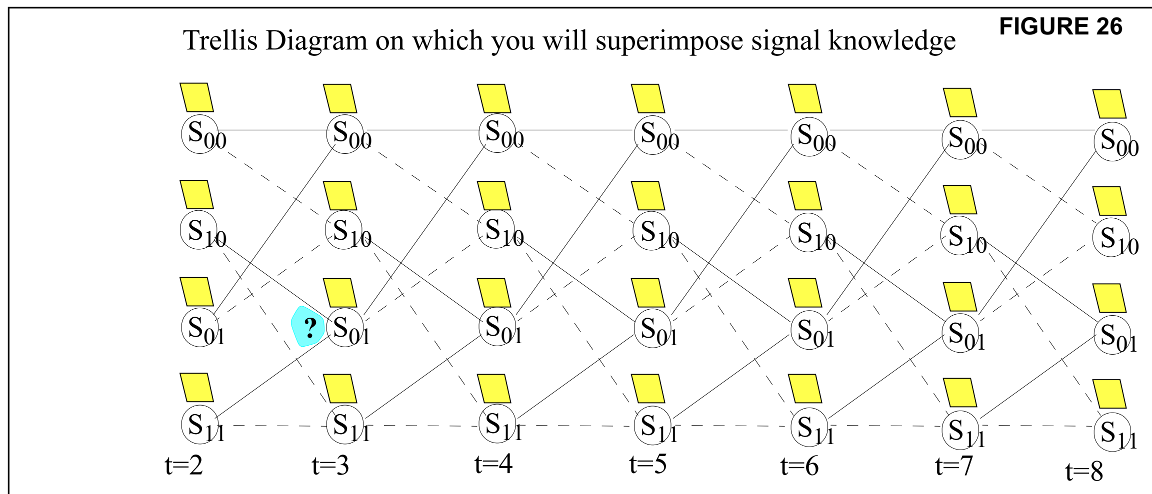
Figure 26 is the same as Figure 24 except some little parallelograms have been drawn associated with each state in each time step.

Fill a minimum of information in each parallelogram. This information would allow your lab partner two to back-trace knowing that H_{01} had the minimum path Hamming metric at t=8. Thus by looking only at Figure 26 and starting at state S_{01} at time t=8, one should be able to tell what the original bit was between t=2 and t=3. You may establish some conventions like a 1 in the state S_{01} box means.... However they must be independent of the data.

a) Using Figure 26, fill in the boxes at t=3 if you have not done so already, so that one can determine the original data bits between t=2 and t=3, and also between t=1 and t=2.
You should hand in the filled in Figure 25 and your list of conventions.

b) How many bits per step must be stored to allow for backtracking and extraction of the original data?

9. **Problem**: In communications *latency* is the term for the time difference between the time the input signal was received and the output signal is sent out. *Throughput* is the number of input signals that can pe processed per second. The point of this problem is to show it does not matter how long it takes to decode the data as long as you can keep up with the input.



c) If a decoder had to wait until all paths converged before it had confidence it could send out a correct output, what would the latency be in clock cycles?
There are two answers for c):
(i) What latency would one get based only on what was needed for the data stream as shown in Figure 19 and 17?
(ii) What was the latency, mentioned earlier in these notes, that experience has shown gives the most likely bit for almost all cases?

d) If the decoder delays the signal by 20 clock cycles, *latency*, would anyone care assuming:[1]

- The signal was a digitized phone conversation? (i) clock =1MHz, (ii) clock = 100Hz. People get annoyed if the round trip delay in a telephone is over 250 ms.
- The signal was a www page?
- The signal was a digital TV signal?

e) If the decoder could not take in the next input until it had spent 12 or 15 clock cycles processing the previous data, would this affect the *throughput*? Would this matter for the applications in d)?

5.0 Extracting The Original Data.

Consider Figure 27. The path with the lowest path Hamming metric (H), starts at state S_{01} at t=8 with H=2. Backing up would take the path to S_{10} at t=7. The edge is a solid line which seems to say the original data was 0. Unfortunately we can't be sure of this. Because of the convolution code, this path's H of 2 could increase in the next few cycles and another path might get the lowest H.

However if one goes back to t=2 and travels ahead in time, only paths that start at S_{11} or S_{01} make it all the way to t=8. The others die out. Only the path from S_{11} has H=2 at t=8, thus we are fairly sure the edge from S_{11} at t=2 to S_{01} at t=3 is on the most likely path and the original data between these two clock edges was 0 (a solid line is 0, a broken line is 1).

This illustrates why we waited six cycles here before sending out the output. At time t=8, we can be somewhat confident that the "0" data at t=2 is the most likely. In general one would wait twice that time to be very sure.

1. People get annoyed if the round trip delay in a telephone is over 250 ms.

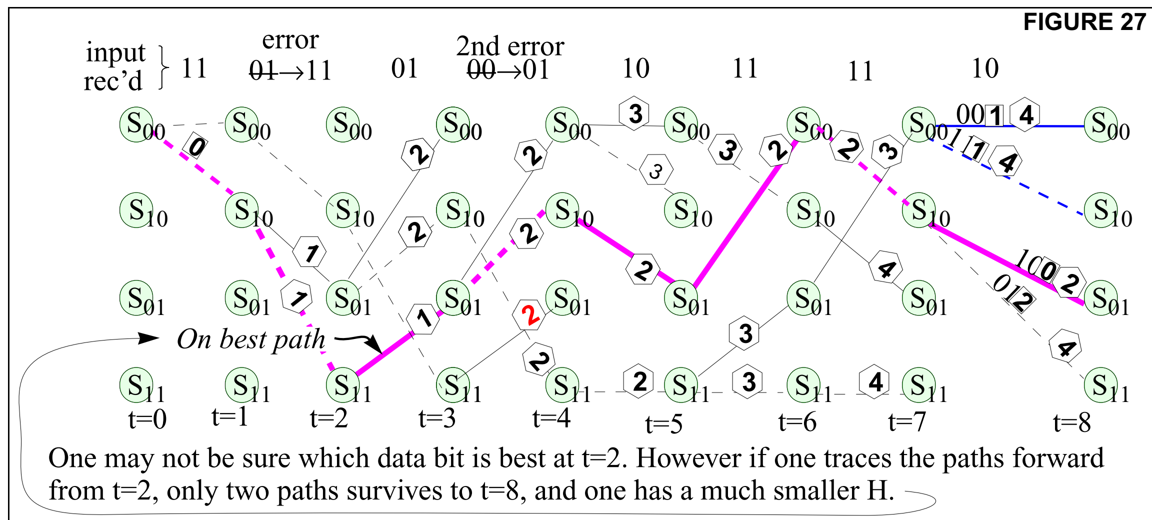


FIGURE 27

One may not be sure which data bit is best at t=2. However if one traces the paths forward from t=2, only two paths survives to t=8, and one has a much smaller H.

5.1 Trace Back In More Detail

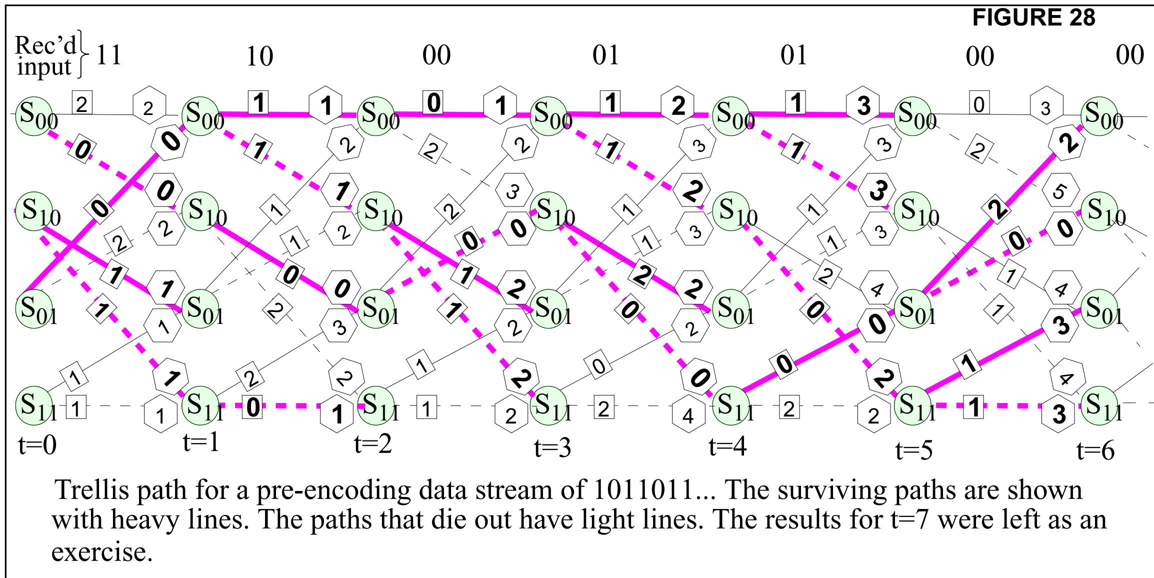
Tracing back is a long process if the full trace is done every data cycle. The back trace can be done only if the clock runs several times faster than the data rate. To trace back 15 cycles to find each output bit, means that the input data rate must be no more than clock/16. One input cycle, followed by fifteen trace-back cycles. It turns out one can increase the data rate up as high as clk/2, but that will come later.

Figure 28 shows the trellis after decoding a 11,10,00,01,01,00,00 input stream.

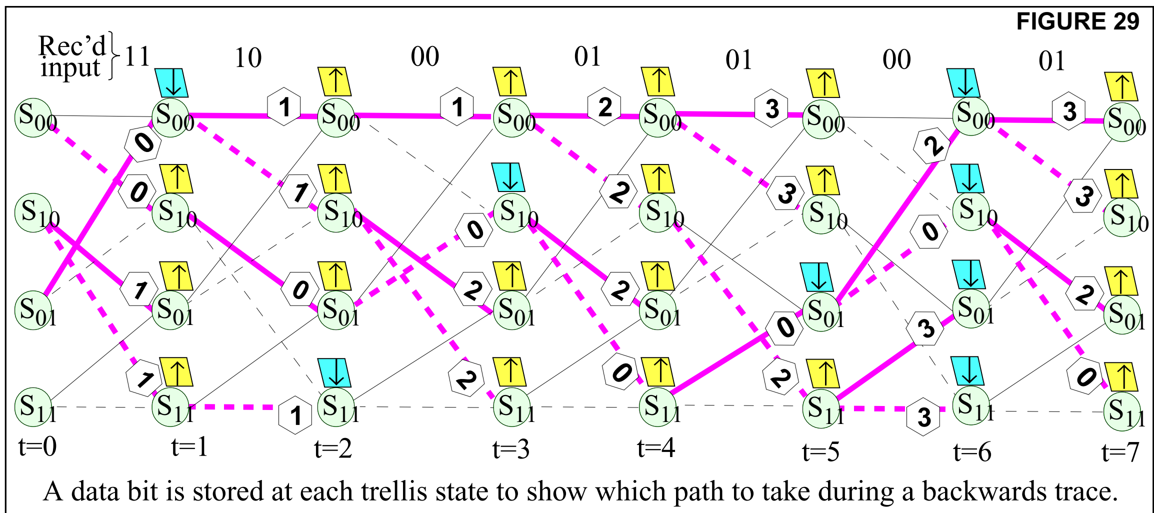
Figure 29 shows how storing one bit, which shows whether to take the upper or the lower path during backtrace, will allow one to reconstruct the trellis.

Figure 30 shows all the surviving paths. If one traces any path back from t=7, one will reach S_{10} at t=2. Since all the back traces converge, one has confidence that the value of the data originally generated between t1 and t2, was one (dashed lines represent a one). This example converged quickly, other examples may take longer.

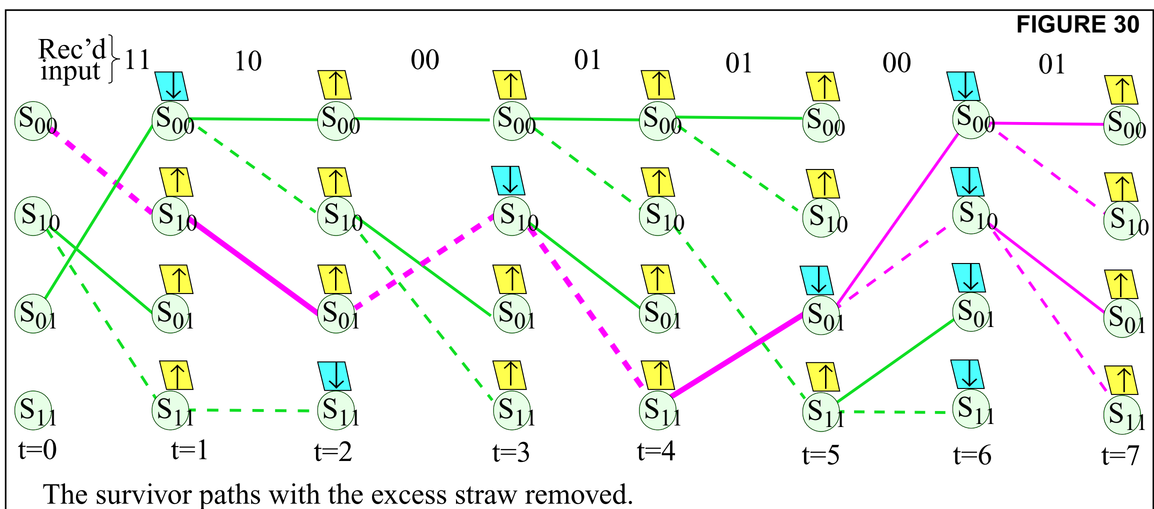
Also note from Figure 29, or from Problem: 6 or Problem: 7 that states with 1 as the most-significant bit (MSB) have only dashed lines (ones) entering them, and states with 0 as the MSB solid lines (zeros) entering them. This means that at the end of the traceback, the data was ”0” if the MSB of the state is 0, and “1” if the MSB of the state is a 1.



Trellis path for a pre-encoding data stream of 1011011... The surviving paths are shown with heavy lines. The paths that die out have light lines. The results for t=7 were left as an exercise.



A data bit is stored at each trellis state to show which path to take during a backwards trace.

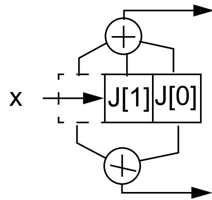


The survivor paths with the excess straw removed.

5.1.1 The Trellis Butterfly.

For all rate 1/2 trellises, one can find a small picture which describes the trellis completely. The picture looks something like a butterfly.

Though it isn't really neccessary, some like to have a formula to express how bits of the state change with different transitions. Note that you if you think of the state as the contents of a shift register, the formula isn't really very informative - it just formalizes how the bits flip.



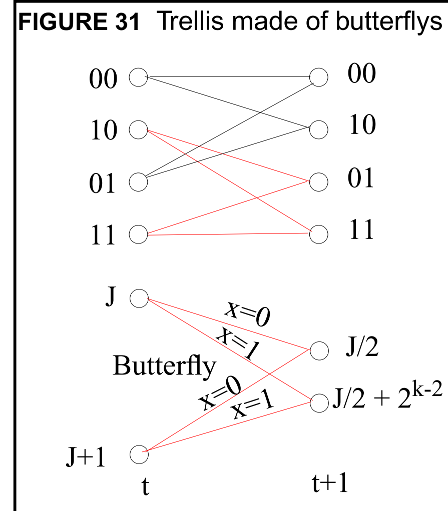


FIGURE 31 Trellis made of butterflys

Then going from J to J/2 represents a right shift, and shifting in an x of 0. Going J to J/2 + 2^{k-2} represents a right shift, but shifting x=1 into the flip flops.

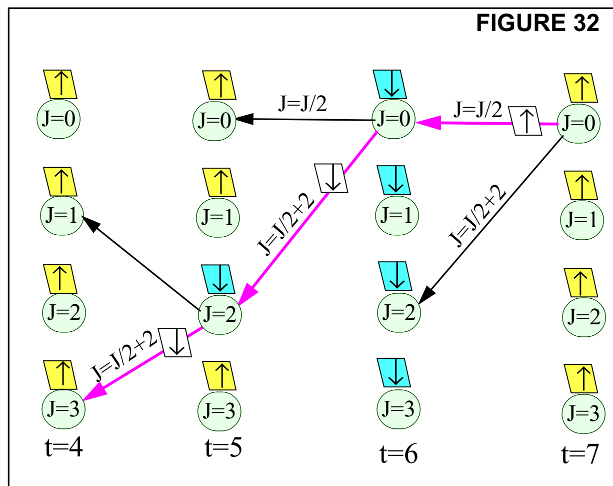


FIGURE 32

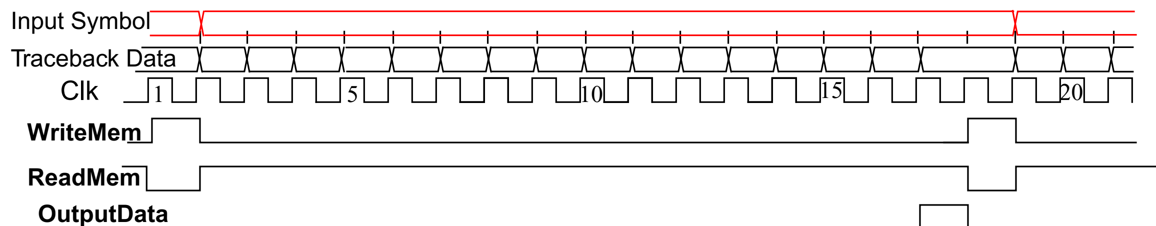
For a constraint length k=3, 2^{k-2}=2.

Figure 32shows how to travel backwards through the trellis using the bits stored during each time step to determine whether to take the upward or downward path. Here we start at state J=0. Knowing that one is in state J allows the two paths to be calculated on the fly.

5.1.2 Timing for the simple decoder

The simplest decoder will have the data input at 1/17th of the clock rate. It will do an add-compare-select on that data and store the "camefrom" bits in memory 1 out of 16 cycles. The rest of the time it is doing the backtrace. That is it will write 1 bit, and then back trace 15 bits to be sure it has found the correct path. Then it will backtrace 1 more bit which it will use as output, before it processes the next data input.

One will need control signals as shown.



This will be slow because the throughput will be 1/17 of the input symbol rate (a symbol here is two-bits).

5.2 Summary of the Design

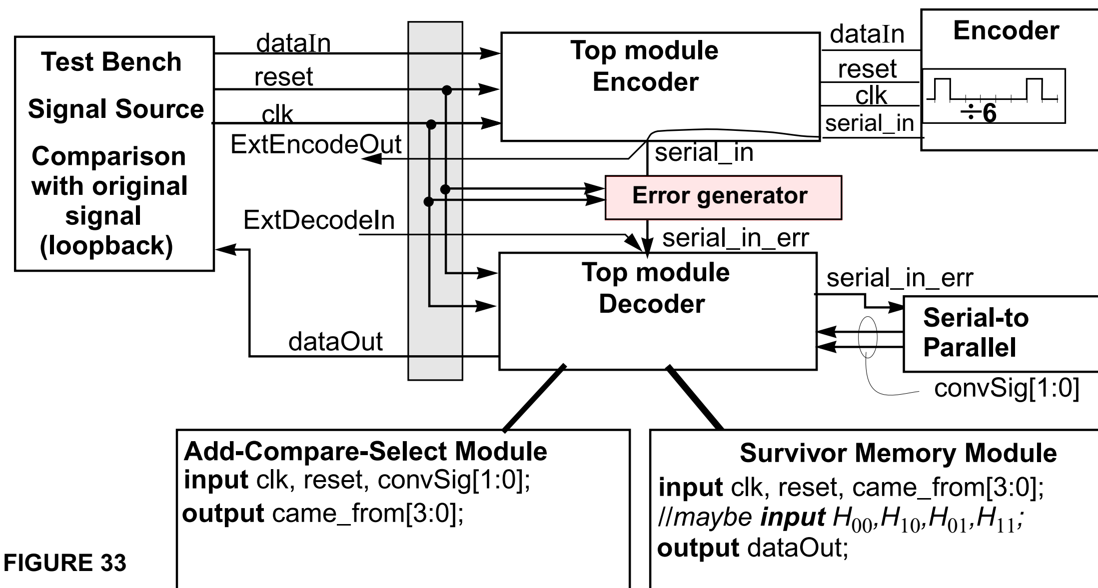


FIGURE 33

Figure 33 shows one way to do the Verilog design. The top modules only collect signals and pass them on to the lower level modules. When an ASIC is built the arguments for the top module are the pins of the ASIC. If this circuity was to all be built on one chip, one would put on a top-top or wrapper module where the shaded box is to define the pins.

The error generator is used for testing. To test the circuit, connect the encoder to the decoder through the error generator, and see if the dataOut equals the dataIn. Under normal (not test) operation, the encoder would need an output lead to the outside world and the decoder would need an input lead from outside.

5.3 Second Lab: Start of Convolution Decoder in Verilog

Now we will consider the overall project. You should be able to design each block in the block diagram except the *Survivor Memory* block which will not be done until Exercise/Lab 3. For the initial circuit, you do not need to do a trace back. Just send out as correct, the data from the state with the lowest H. Of course this will not have any error correction.

You should consider these concepts:

a) We will run the encoder and decoder from a common clock. However the decoder will need to run faster that the encoder shift register, likely by six times.
In that case you should generate a pulse which comes every six clock cycles which shifts the encoder register.

b) Your design will be a rate=1/2, constraint length=3; Gz_1=[111],Gz_0=[101] decoder.

c) To make your code more reusable you may want to **parameterize** your design. One can do this easily in Verilog for some values. For others it may be too much trouble.

- For example, at this time, you do not know the register lengths needed in the Add-Compare-Select unit. Parameterizing them makes it easy to adjust their length later.

- One can use **parameter** which must be defined inside the module. This is good for local parameters.

```
module decode
   parameter size=9;
   . . .
   reg ACS[size-1, 0];
```

- One could use macros where the definition needs only to be compiled before it is used. This is good for global definitions.

```
`define  size  9;
module decode
   . . .
   reg ACS[`size -1,0];
```

d) Considerable emphasis will be given to testing. One simple test is a loopback test where the decoded output is sent to the test bench where it is compared with the original input.

e) The Error Generator block is necessary if you want to simulate to the error correction properties. First errors will be generated part of test bench so it is only useful during simulation. Later you might consider making it part of the loopback test so it can be used for testing in the field.

What to do for the lab

1. Draw a block diagram somewhat like that of Figure 33. [2]However make it bigger and show the arguments passed to all modules. If a module will be longer than a page of code, try to divide it.

2. Write Verilog for the *Decoder Top Module* and *Encoder Top Module* (already done?) Draw a simplified hardware diagram for these modules.

3. Recall that the encoder uses a counter and decode its output to shift the register and multiplex the output according to the following enable/control signals.

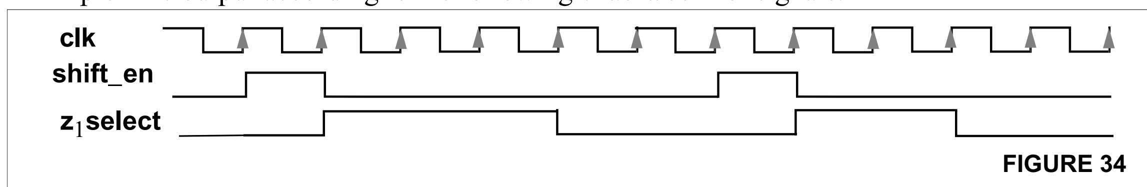


FIGURE 34

- It provides a pulse every six clock cycles to enable the shift register.
- It provides a divide by three signal to change the output data bit every 3 clock cycles.

   Considering the sequence of the input stream, design a s*erial to parallel* module which will capture z1 and z0 at an appropriate time and present them to the decoder as a group, convsig[1:0], ready for decoding. Draw a waveform diagram for the

2. We like originality in block diagrams as long as you can give a reason for changes.

control signals that you will generate and when things move around in the serial-to-parallel register.

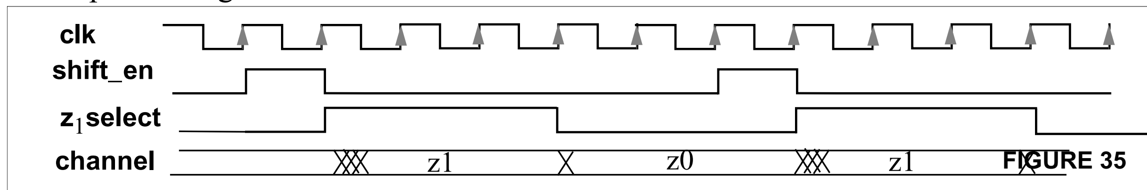


FIGURE 35

4. Write the Verilog code for the *serial to parallel* module.
   Draw a simplified hardware diagram for the circuit.

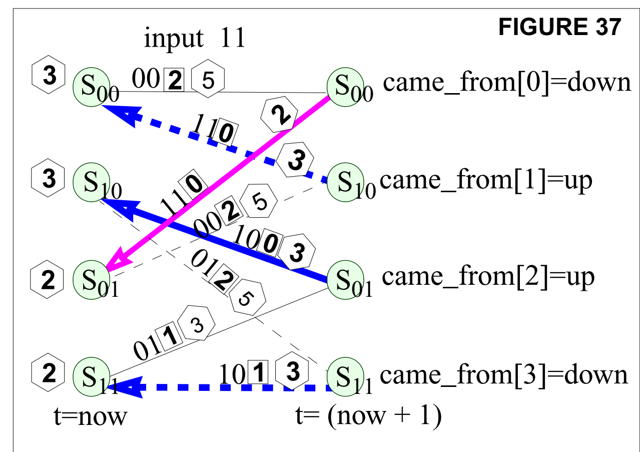
5. Write the *Add-Compare-Select* module. (See Exercise 2.)Draw a simplified hardware diagram for the circuit.

6. Modify the *Test Bench* to handle the decoder and the encoder. This should include a loop-back test which compares the *dataIn* (*x* on the right) with *dataOut*.
   You will have a delay (latency) between *dataIn* and *dataOut*. At the start *dataOut* will be the bit corresponding to the present lowest H (path Hamming metric) so the latency will be only that of the serial-to-parallel converter. Later you will want to add a latency of 4 to 5 times the *constraint length*. Include the resultant simulation log and waveform.

```
// send in a new x every clock cycle
always @(posedge clk)
  begin
     if (I==9) $finish;
      x<= #1 data[I];
      if(I>latncy)   y<=#1 data[I-latncy];
     I<=I+1;
  end
 assign err=(y!=dataOut)
```
FIGURE 36

7. Add to Add_Compare_Select module to generate a four 1-bit signal called *came_from*. These signals indicate whether the trellis lines leading back from the next states to the present-time-step states, came from a higher state or a lower state.

   Thus it would show whether the next state S_{00} came from the present S_{00} (up) or S_{01} (down). Figure 37 shows these bits and their meaning. These four came_from signals will be sent to the survivor-path memory-module which will be written later.



   Add this logic to the simplified diagram from question 6.

8. Consider the Error Generator. For this question treat it like a test bench so you can use nonsynthesizable constructs like **$random**. This gives a new random integer every time it is called. Randy will be this random integer truncated to 5 bits. A competely random 5-bit number will, on average, be 01110 (or any single value) one time out of 32. However once in a while it might be 01110 twice in a row. Write a random number

FIGURE 38
```
reg [4:0] randy;
always @(posedge clk) begin

//randy gets the 5 lsb of $random
     randy <= $random;
     if (randy = = 4'b01110)
          serial_in_err<= ~serial_in;
```

generator such as in Figure 38 to produce an error, on average, every 8 data bits across the channel. Be carefull, this does NOT mean every 8 clock cycles. You will need to advance the random number generator in a similar way as the shift-register in the encoder. Keep a count of every time you generate an error in the channel.

9. Write a pseudorandom generator as was used in 97.350 to replace $**random** in question 8. There is a lot about pseudorandom generators in the notes. If you use such a generator, you must make its period much longer than 31 or your errors will be periodic. Supposedly random errors that come every 31 bits are not a good test. There was a question about this circuit on the Winter 2001 final which is available on the web. Include a counter in the random error generator to count the number of errors introduced into the channel. Again, make sure you advance the error generator only as appropraite, not every clock cycle.