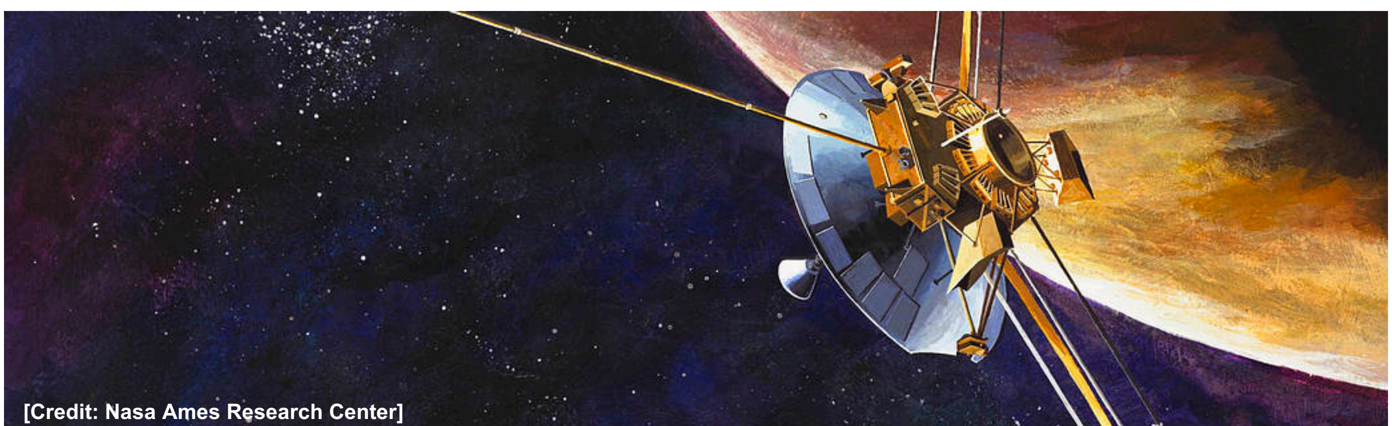
Convolutional Codes

COS 463: Wireless Networks
Lecture 9
Kyle Jamieson

[Parts adapted from H. Balakrishnan]

Convolutional Coding: Motivation

- So far, we've seen block codes
- **Convolutional Codes:**
  - **Simple design,** especially at the transmitter
  - **Very powerful error correction** capability (used in NASA Pioneer mission deep space communications)



Convolutional Coding: Applications

- **Wi-Fi** (802.11 standard) and **cellular networks** (3G, 4G, LTE standards)
- Deep space **satellite communications**
- Digital Video Broadcasting (**Digital TV**)
- **Building block** in more advanced codes (**Turbo Codes**), which are in turn used in the above settings

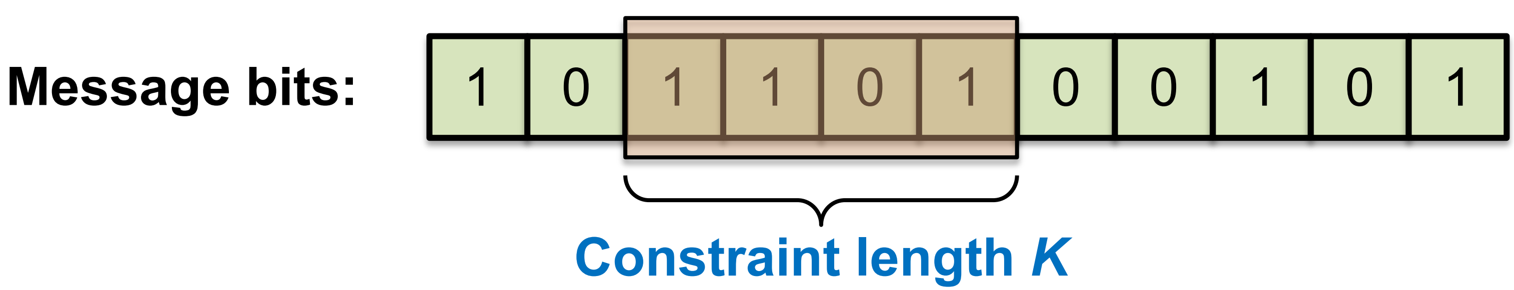
Today

1. **Encoding data using convolutional codes**
   - How the encoder works
   - Changing code rate: Puncturing

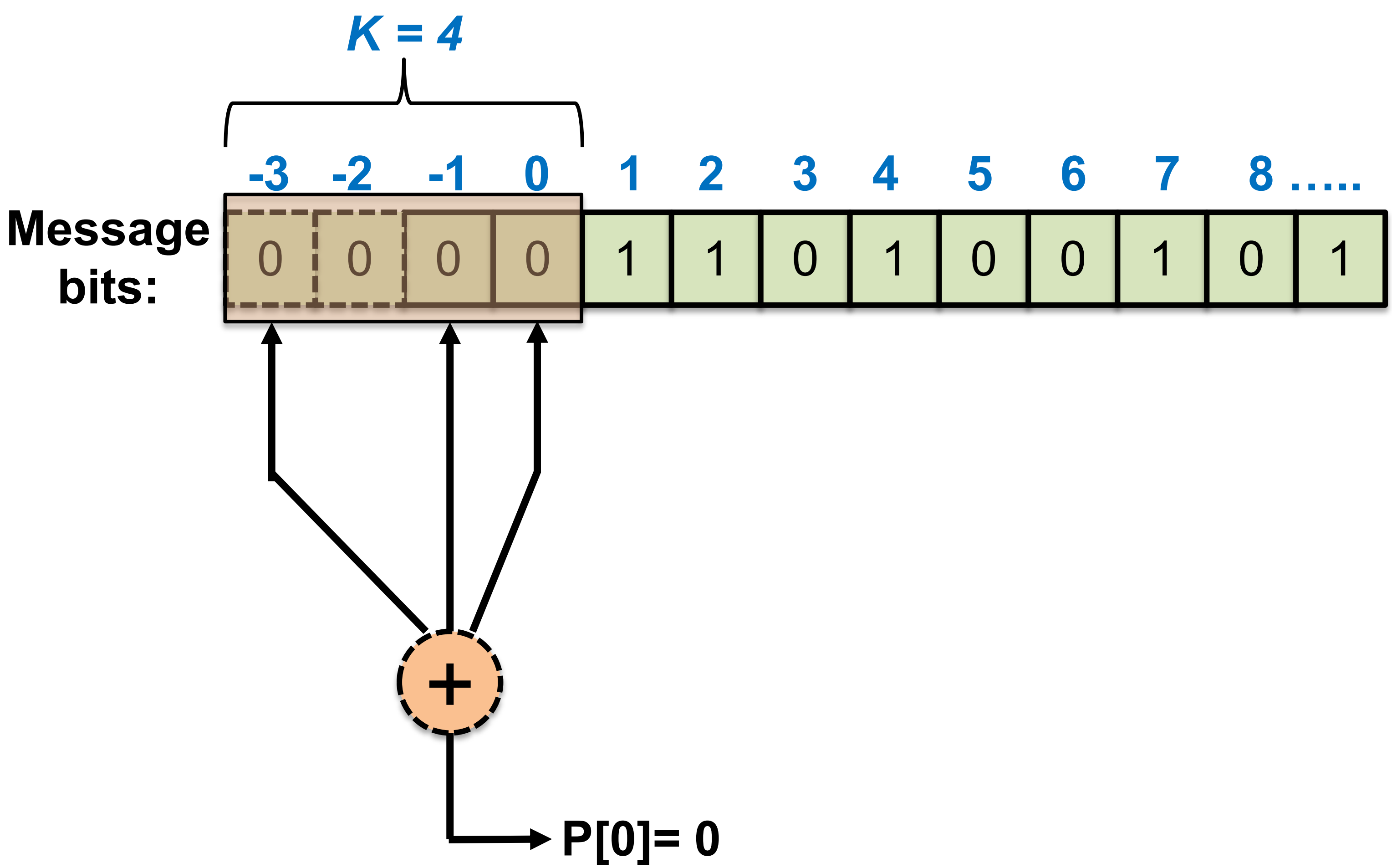
2. Decoding convolutional codes: Viterbi Algorithm

Convolutional Encoding

- Don't send message bits, send **only parity bits**
- Use a **sliding window** to select which message bits may participate in the parity calculations

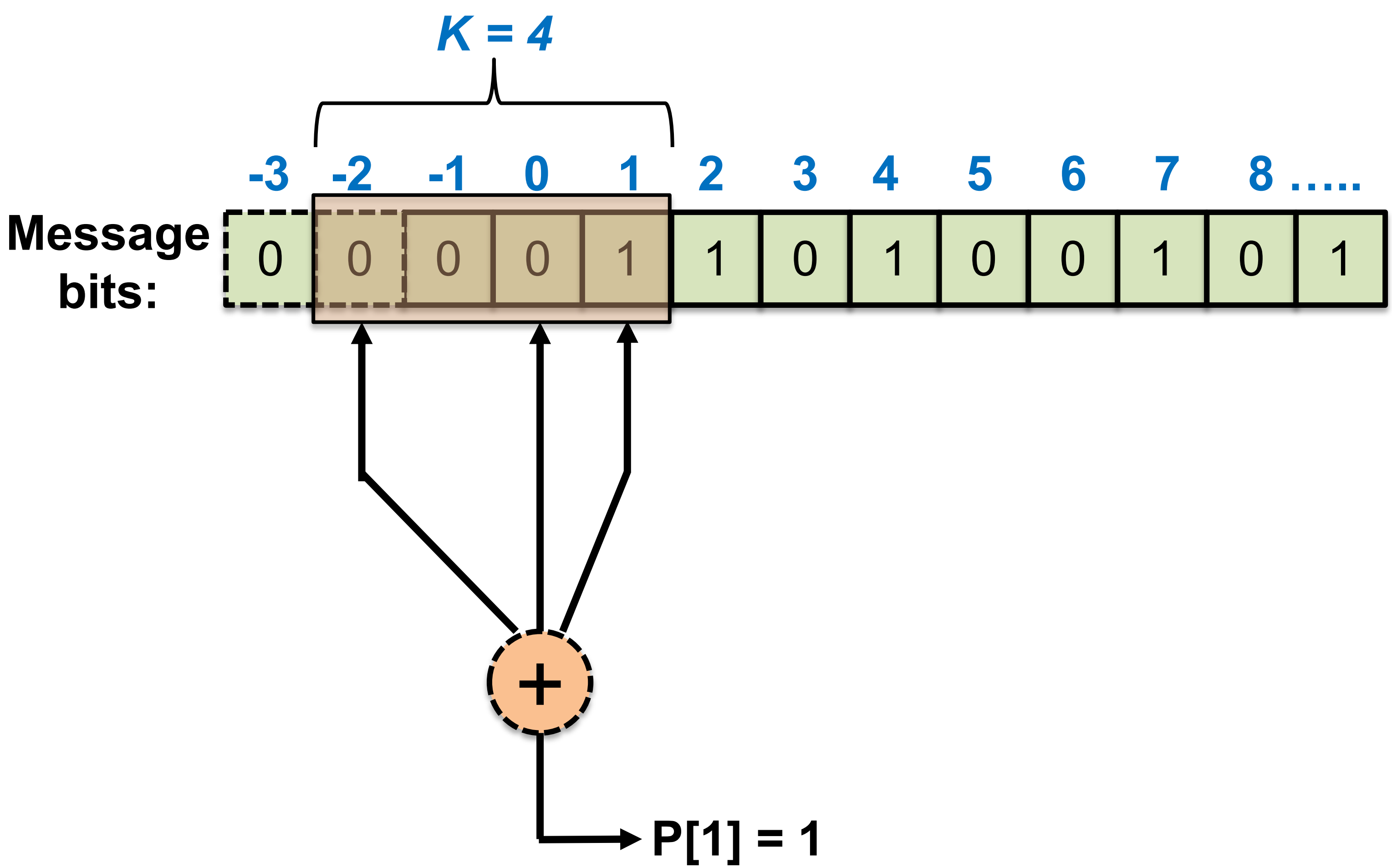


Sliding Parity Bit Calculation



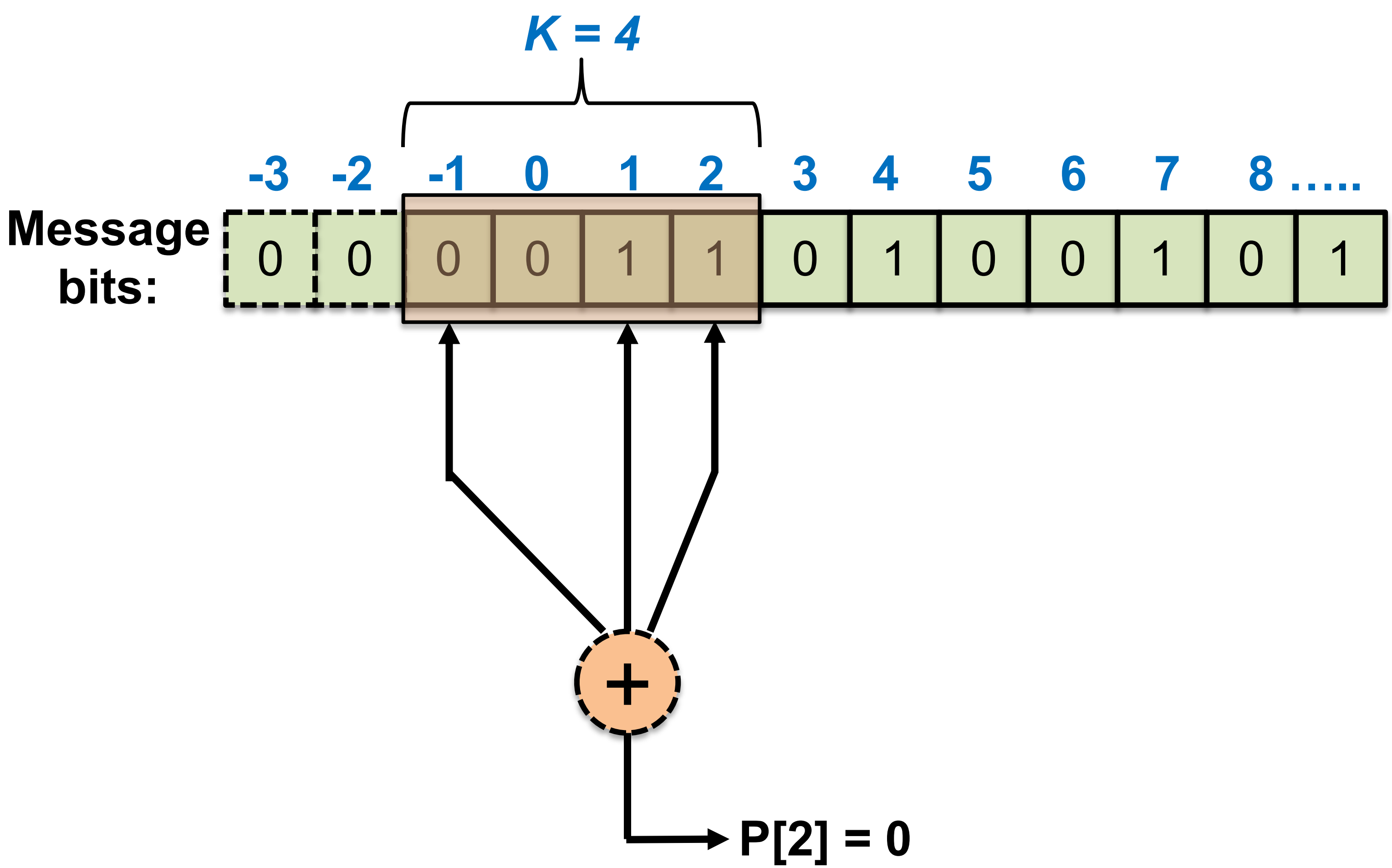
- **Output: 0**

Sliding Parity Bit Calculation



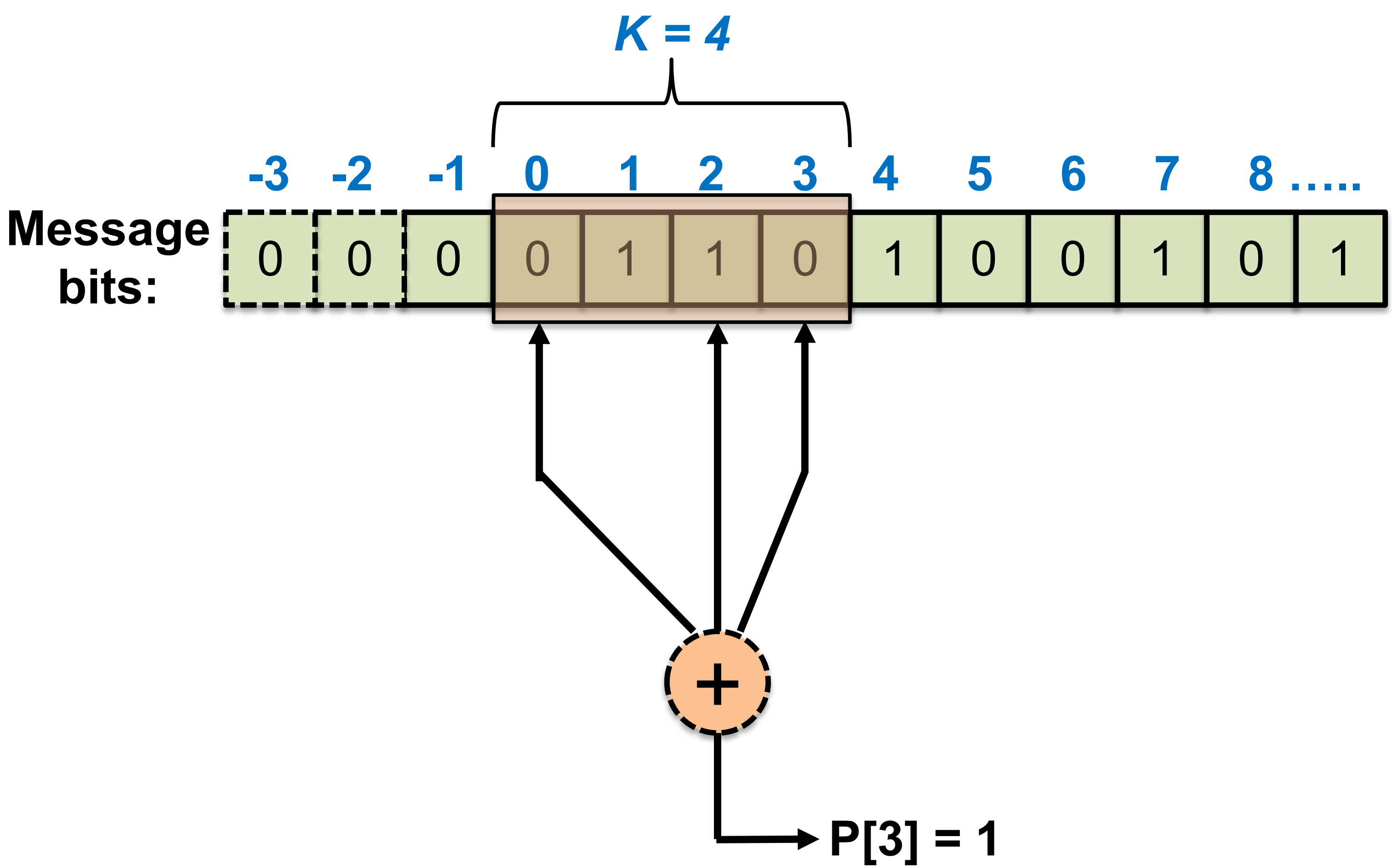
- **Output: 01**

Sliding Parity Bit Calculation



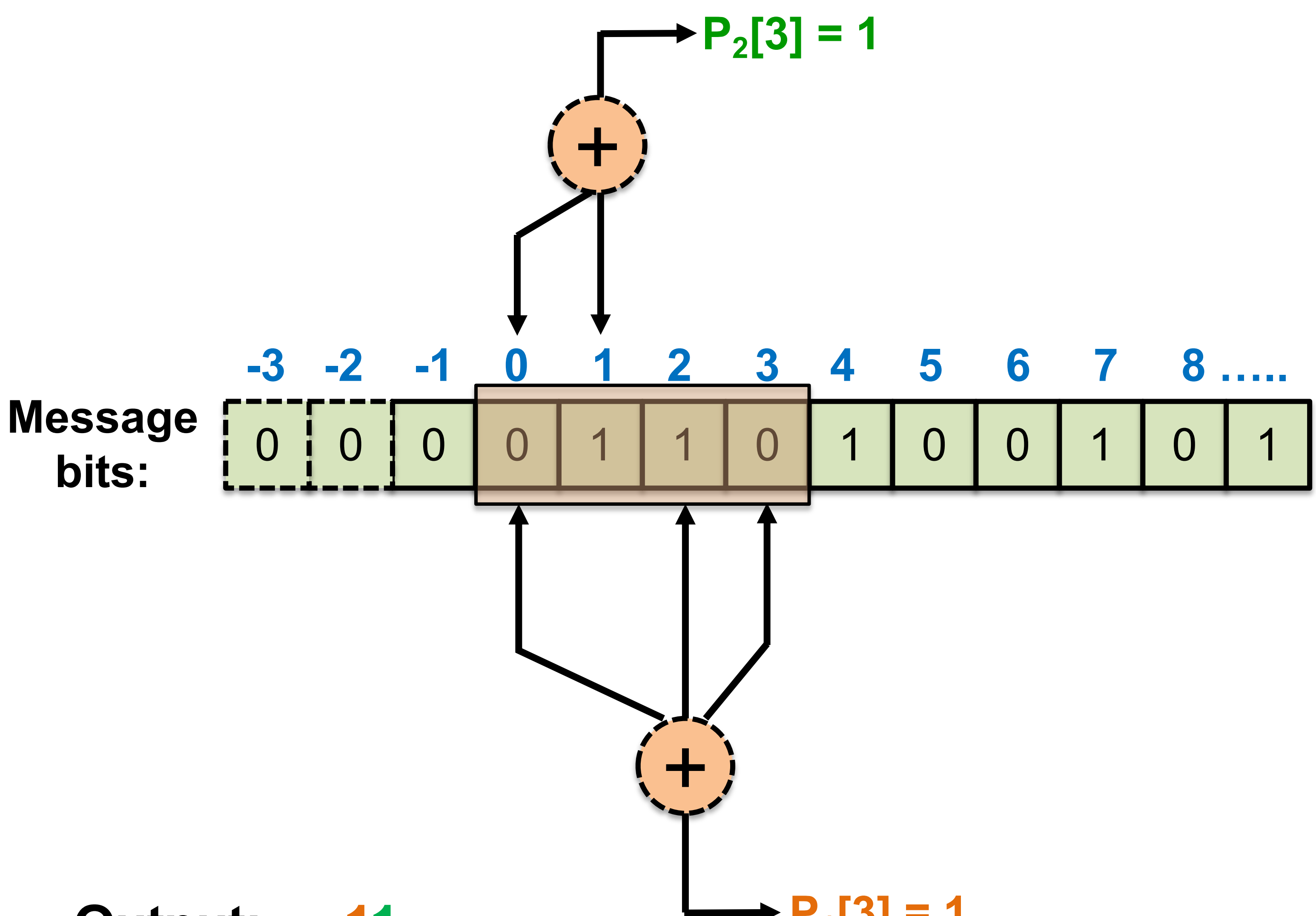
- **Output: 010**

Sliding Parity Bit Calculation



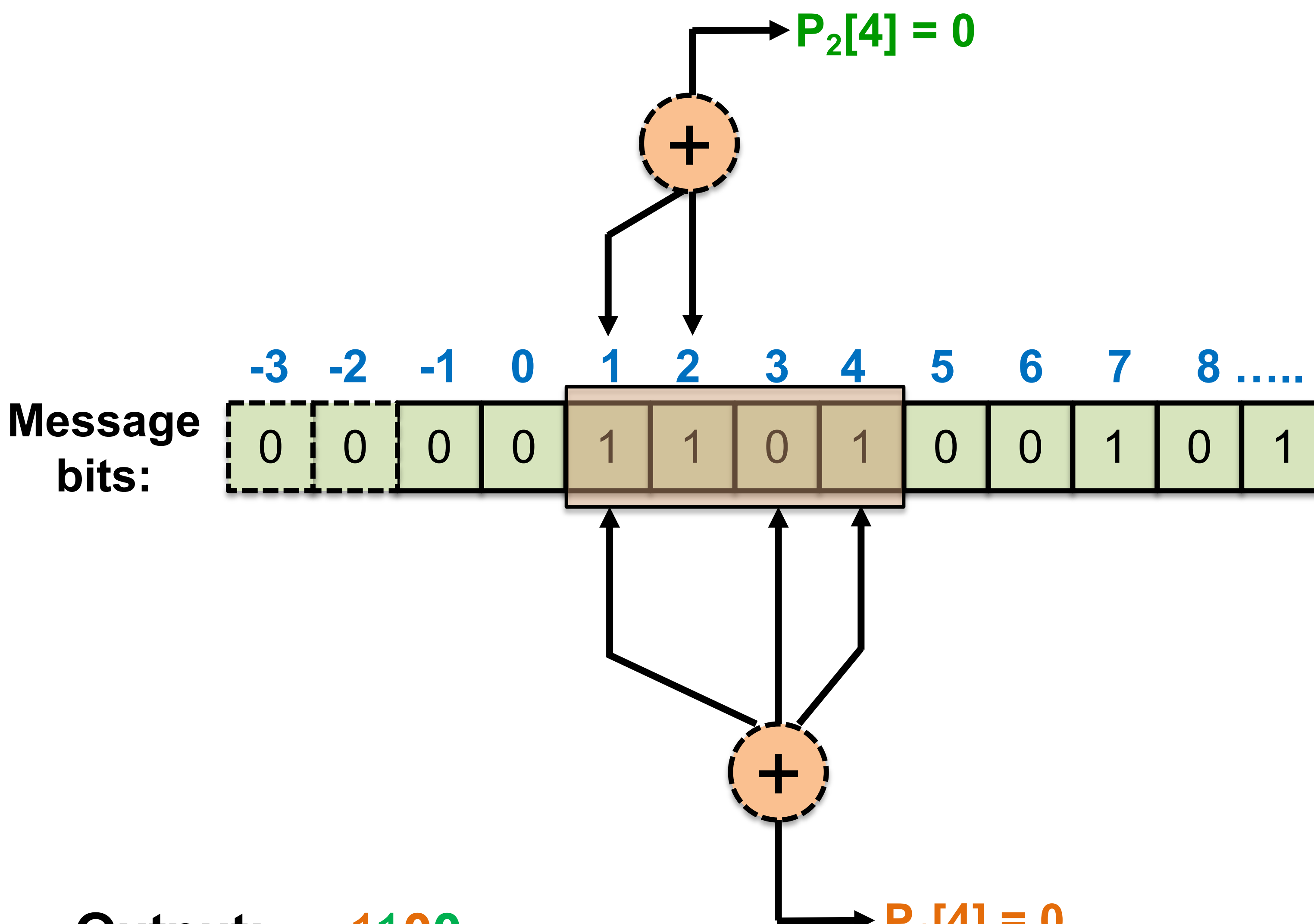
- **Output: 0100**

Multiple Parity Bits



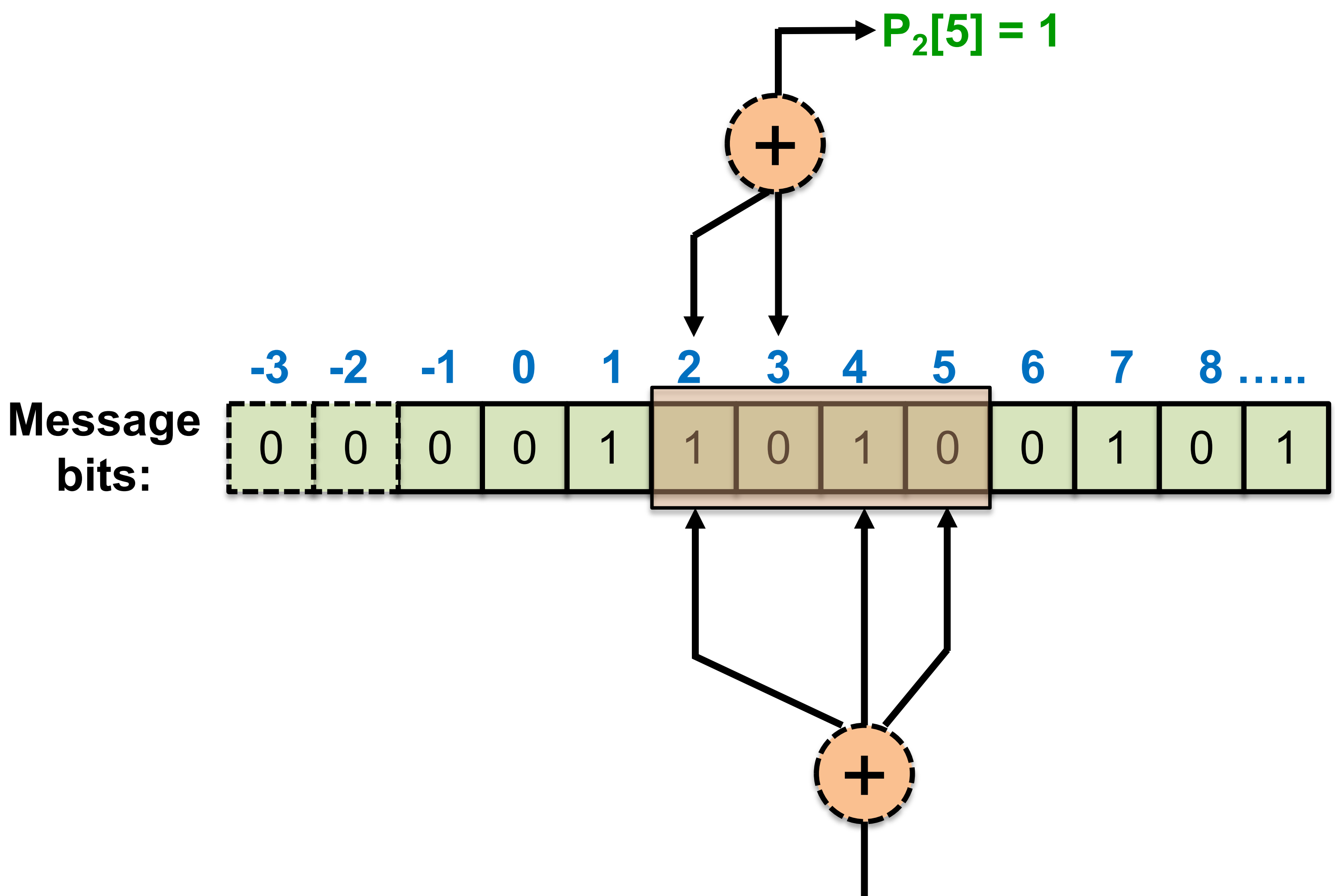
- **Output: ….11**

Multiple Parity Bits



- **Output: ….1100**

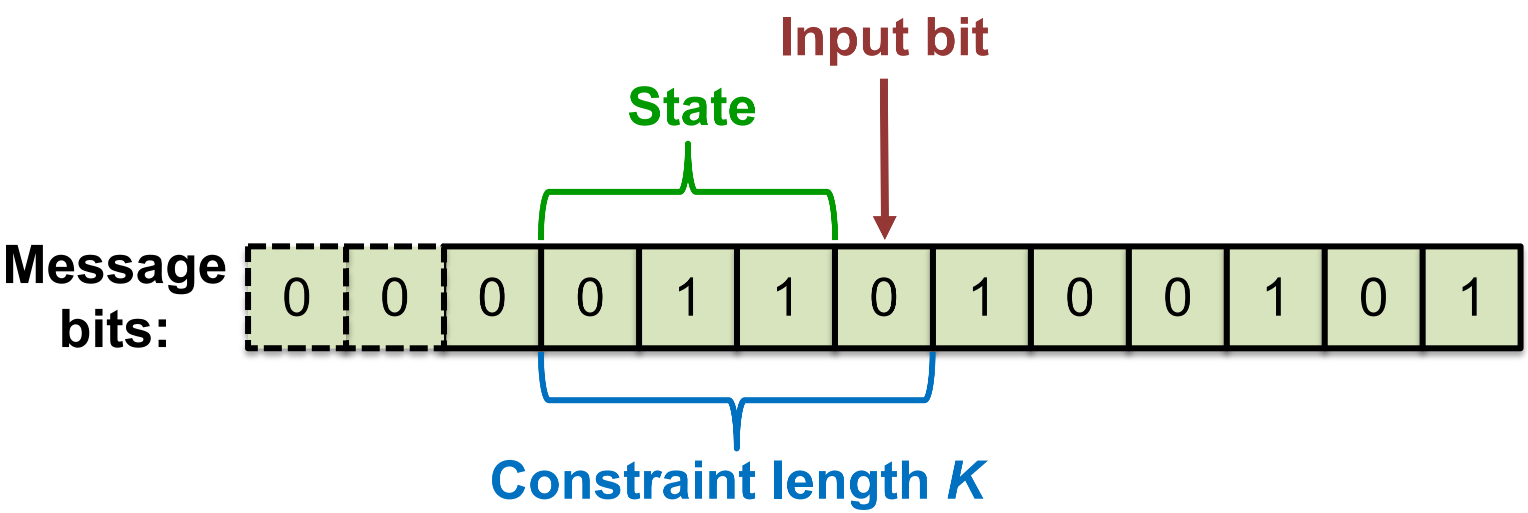
Multiple Parity Bits



- **Output: ….**110001

Encoder State

- **Input bit** and **K-1 bits of current state** determine state on next clock cycle
  - Number of states: 2^{K-1}



Constraint Length

- ***K*** is the **constraint length of the code**
- **Larger K:**
  - **Greater redundancy**
  - **Better error correction possibilities** (usually, not always)

Transmitting Parity Bits

- **Transmit the parity sequences, not the message itself**
  - Each message bit is "**spread across**" K bits of the output parity bit sequence
  - If using **multiple generators**, **interleave** the bits of each generator
    - *e.g.* (two generators):

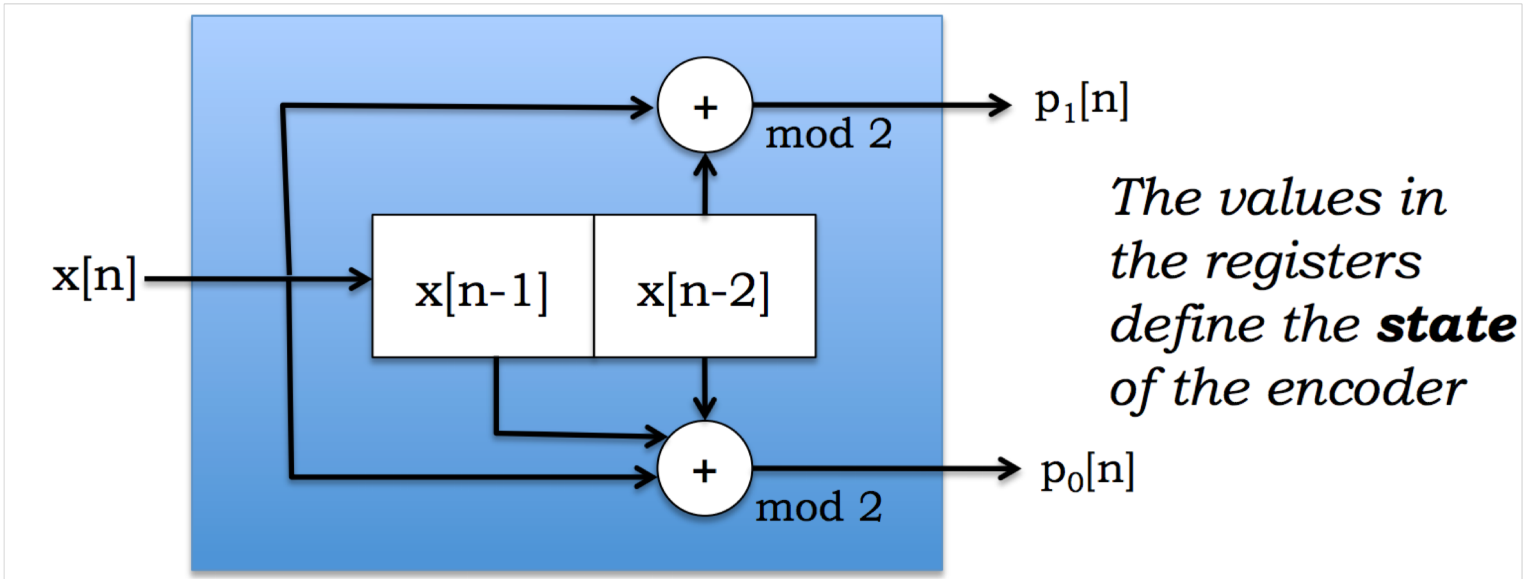
$$p_0[0], p_1[0], p_0[1], p_1[1], p_0[2], p_1[2]$$

Transmitting Parity Bits

- **Code rate** is 1 / #_of_generators
  - *e.g.*, 2 generators → rate = ½

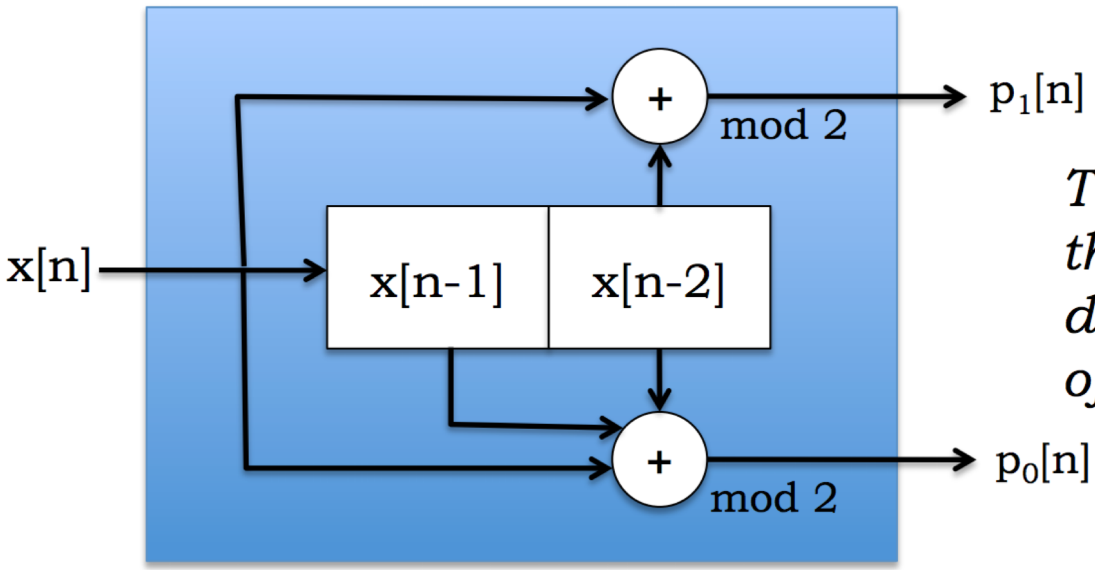
- **Engineering tradeoff**:
  - More generators **improves bit-error correction**
    - But **decreases rate of the code** (the number of message bits/s that can be transmitted)

Shift Register View



- **One message bit x[n] in, two parity bits out**
  - **Each timestep:** message bits **shifted right** by one, the **incoming bit** moves into the **left-most register**

Equation View



The values in the registers define the ***state*** *of the encoder*

0th stream: $p_0[n] = x[n] + x[n-1] + x[n-2] \pmod 2$

1st stream: $p_1[n] = x[n] + x[n-2] \pmod 2$

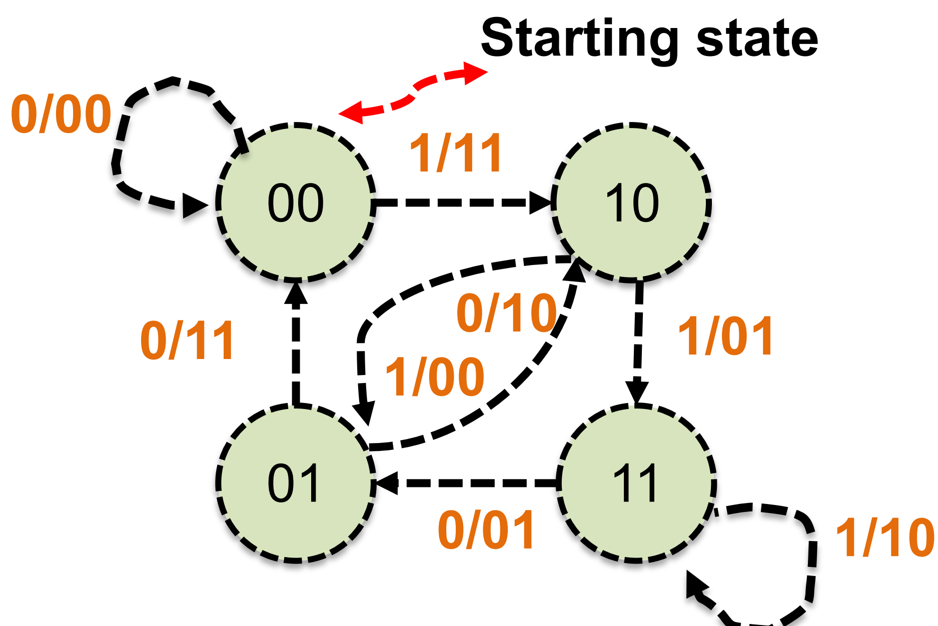
Today

1. **Encoding data using convolutional codes**
   - **Encoder state machine**
   - Changing code rate: Puncturing

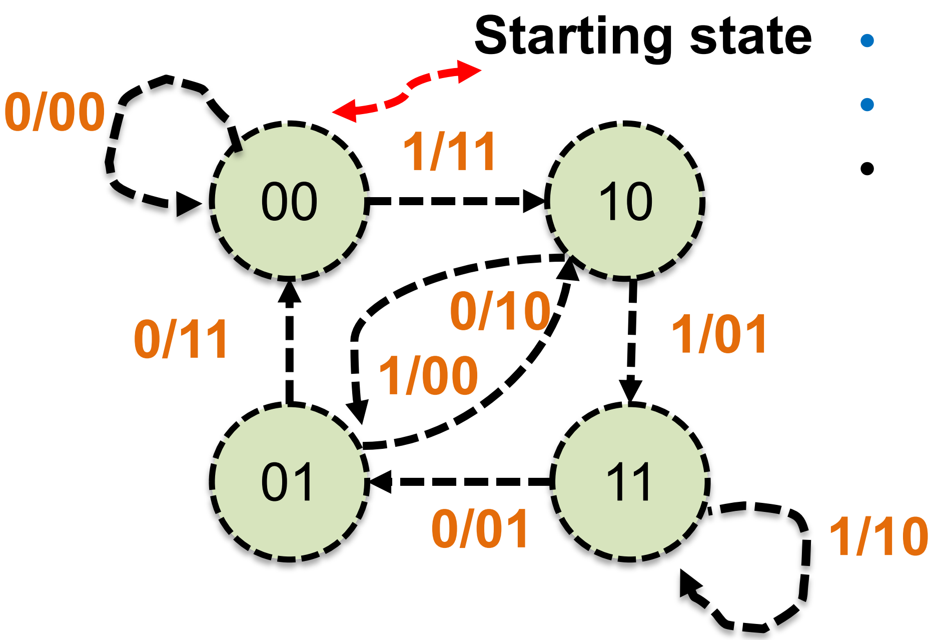
2. Decoding convolutional codes: Viterbi Algorithm

State Machine View

- Example: **K = 3, code rate = ½, convolutional code**
  - There are 2^{K-1} states
  - **States** labeled with **(x[n-1], x[n-2])**
  - **Arcs** labeled with **$x[n]/p_0[n]p_1[n]$**
  - **Generator: g_0 = 111, g_1 = 101**
  - **msg** = 101100



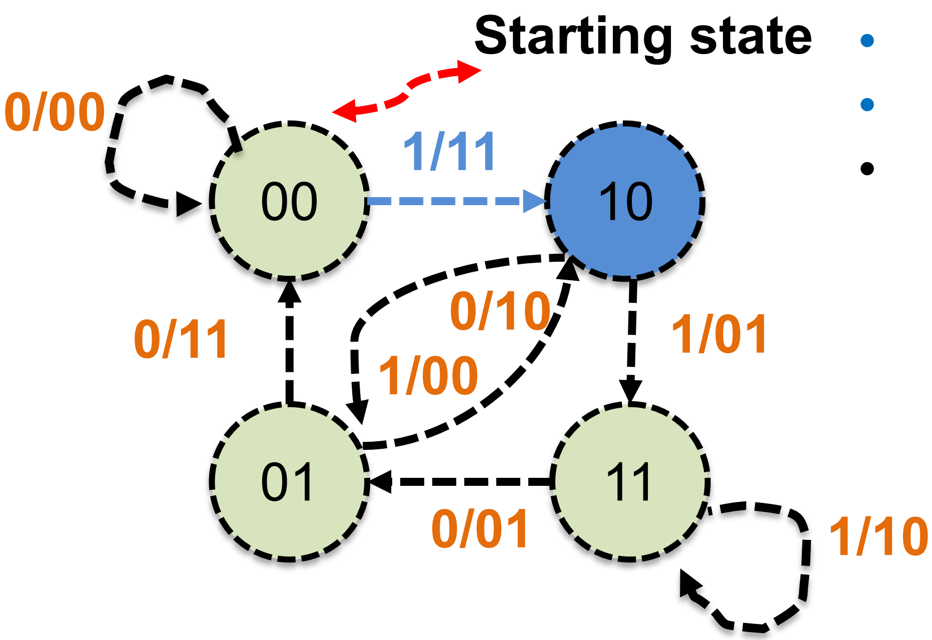
State Machine View



- $P_0[n] = (1*x[n] + 1*x[n-1] + 1*x[n-2]) \bmod 2$
- $P_1[n] = (1*x[n] + 0*x[n-1] + 1*x[n-2]) \bmod 2$
- **Generators**: $g_0 = 111$, $g_1 = 101$

- **msg** = 101100
- **Transmit**:

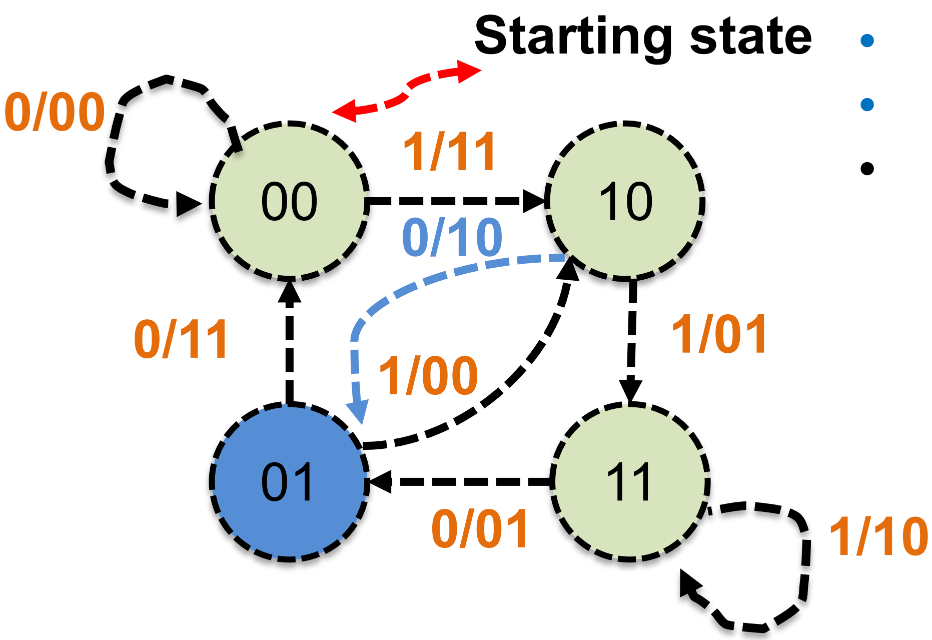
State Machine View



- $P_0[n]$ = 1*1 + 1*0 + 1*0 mod 2
- $P_1[n]$ = 1*1 + 0*0 + 1*0 mod 2
- **Generators**: g_0 = 111, g_1 = 101

- **msg** = **1**01100
- **Transmit**: **11**

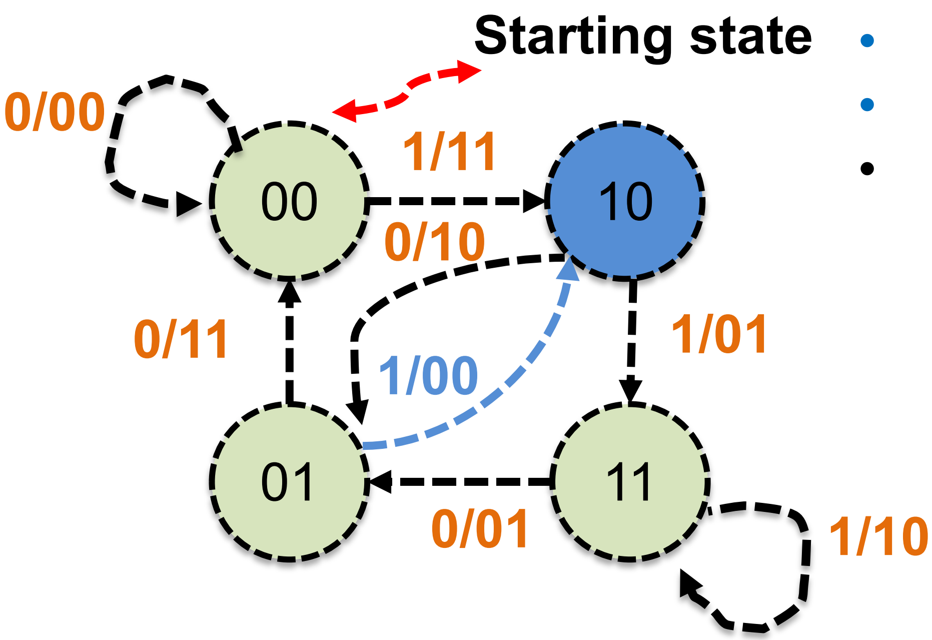
State Machine View



- $P_0[n]$ = 1*0 + 1*1 + 1*0 mod 2
- $P_1[n]$ = 1*0 + 0*1 + 1*0 mod 2
- **Generators**: g_0 = 111, g_1 = 101

- **msg** = 1**0**1100
- **Transmit**: 11 **10**

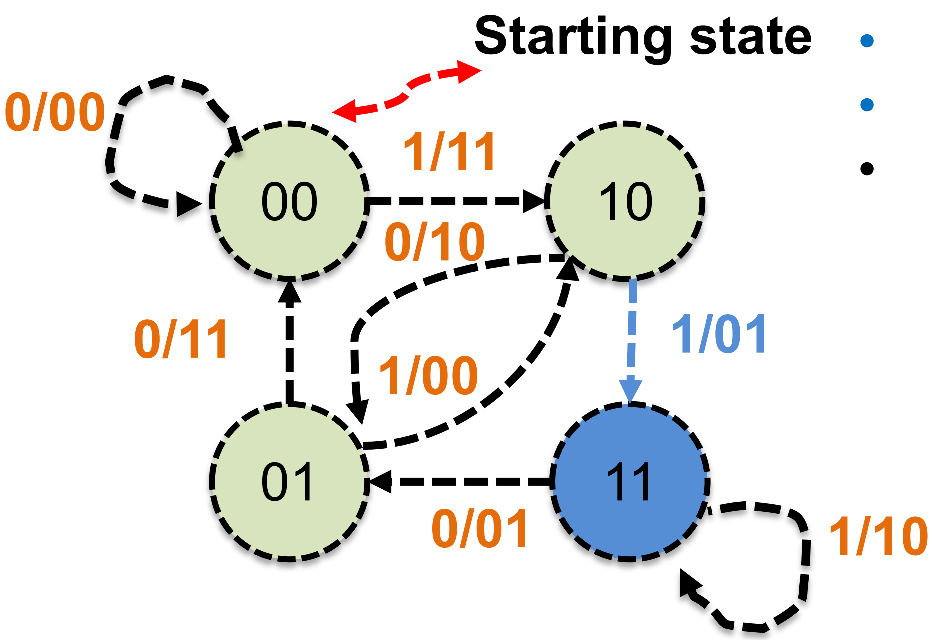
State Machine View



- $P_0[n]$ = 1*1 + 1*0 + 1*1 mod 2
- $P_1[n]$ = 1*1 + 0*0 + 1*1 mod 2
- **Generators**: g_0 = 111, g_1 = 101

- **msg** = 10**1**100
- **Transmit**: 11 10 **00**

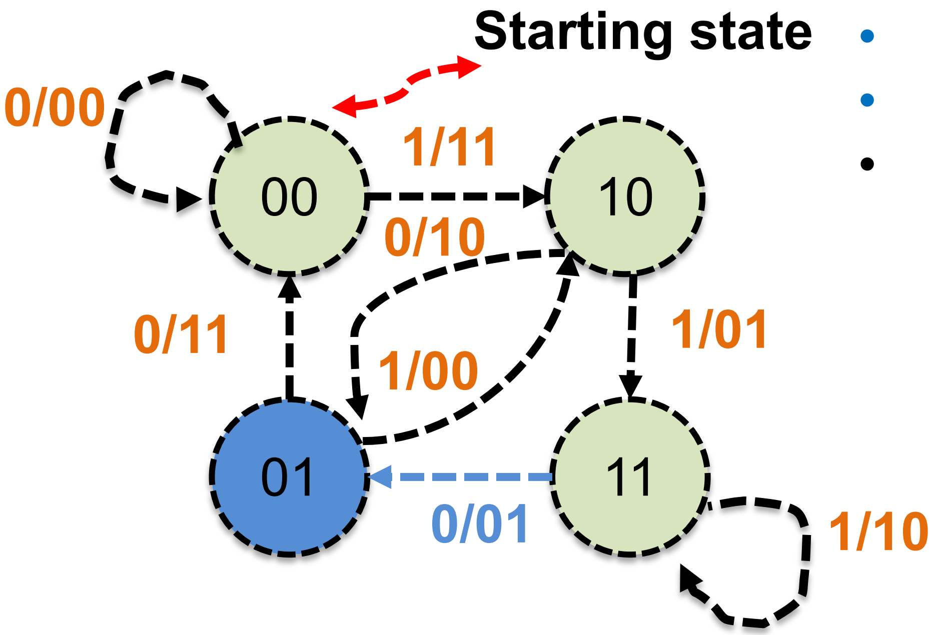
State Machine View



- $P_0[n] = 1*1 + 1*1 + 1*0$
- $P_1[n] = 1*1 + 0*1 + 1*0$
- **Generators**: $g_0 = 111$, $g_1 = 101$

- **msg** = 101**1**00
- **Transmit**: 11 10 00 **01**

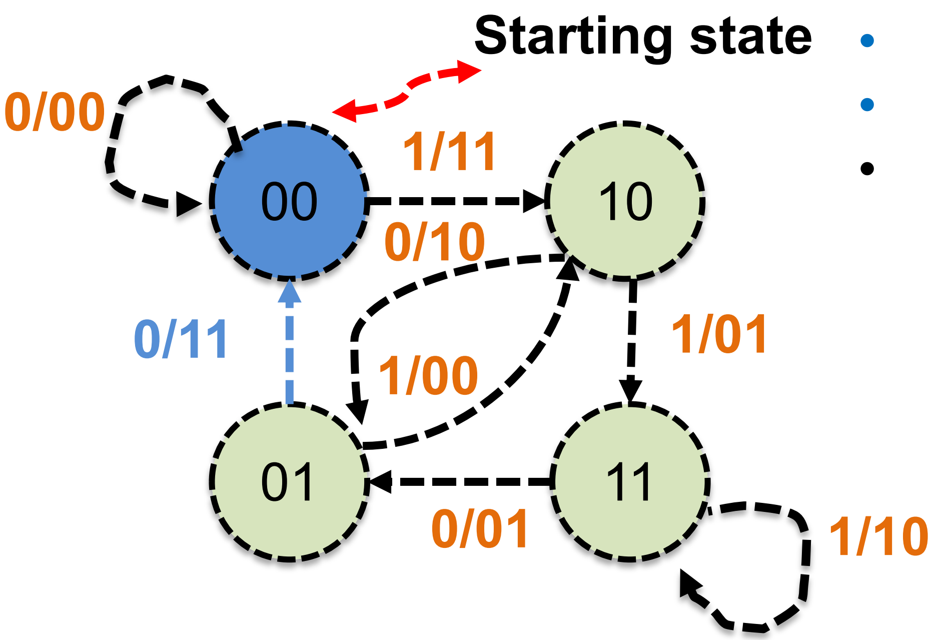
State Machine View



- $P_0[n] = 1*0 + 1*1 + 1*1$
- $P_1[n] = 1*0 + 0*1 + 1*1$
- **Generators**: $g_0 = 111$, $g_1 = 101$

- **msg** = 1011**00**
- **Transmit**: 11 10 00 01 **01**

State Machine View



- $P_0[n] = 1*0 + 1*0 + 1*1$
- $P_1[n] = 1*0 + 0*0 + 1*1$
- **Generators**: $g_0 = 111$, $g_1 = 101$

- **msg** = 10110**0**
- **Transmit**: 11 10 00 01 01 **11**

Today

1. **Encoding data using convolutional codes**
   - How the encoder works
   - **Changing code rate: Puncturing**

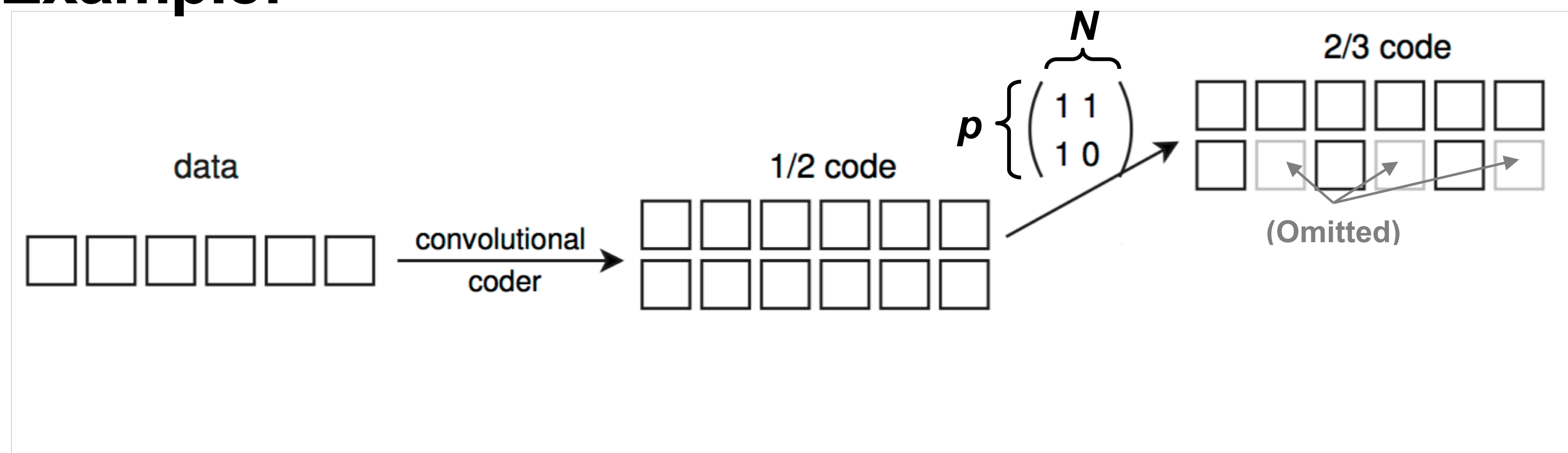
2. Decoding convolutional codes: Viterbi Algorithm

Varying the Code Rate

- *How to* ***increase the rate*** *of a convolutional code?*

- Transmitter and receiver agree on coded bits to **omit**
  - ***Puncturing table*** indicates which bits to include (**1**)
    - Contains ***p*** rows (one per parity equation), ***N*** columns

- **Example:**



Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **With Puncturing:**

$$\boldsymbol{P_1} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Puncturing table

Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **With Puncturing matrix:**

$$P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

5 out of 8 bits are retained

Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **With Puncturing matrix:**

$$P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

- **Punctured, coded bits:**

0
0

Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **With Puncturing matrix:**

$$P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

- **Punctured, coded bits:**

0	0
0	

Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **With Puncturing matrix:**

$$P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

- **Punctured, coded bits:**

0	0	1
0		

Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **With Puncturing matrix:**

$$P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

- **Punctured, coded bits:**

0	0	1	
0			1

Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **With Puncturing matrix:**

$$P_1 = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

- **Punctured, coded bits:**

0	0	1		1
0			1	1

Punctured convolutional codes: example

- **Coded bits =**

0	0	1	0	1
0	0	1	1	1

- **Punctured, coded bits:**

0	0	1		1
0			1	1

- Punctured rate is **increased** to: **R = (1/2) / (5/8) = 4/5**

Stretch Break and Question [MIT 6.02 Chp. 8, #1]

- Consider a convolutional code whose parity equations are:

$$p_0 = x[n] + x[n-1] + x[n-3]$$
$$p_1 = x[n] + x[n-1] + x[n-2]$$
$$p_2 = x[n] + x[n-2] + x[n-3]$$

1. What's the rate of this code? How many states are in the state machine representation of this code?

2. To increase the rate of the given code, 463 student Lem E. Tweakit punctures it with the following puncture matrix:

$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$. What's the rate of the resulting code?

Today

1. Encoding data using convolutional codes

2. **Decoding convolutional codes: Viterbi Algorithm**
   – Hard decision decoding
   – Soft decision decoding

Motivation: The Decoding Problem

- Received bits:
  000101100110

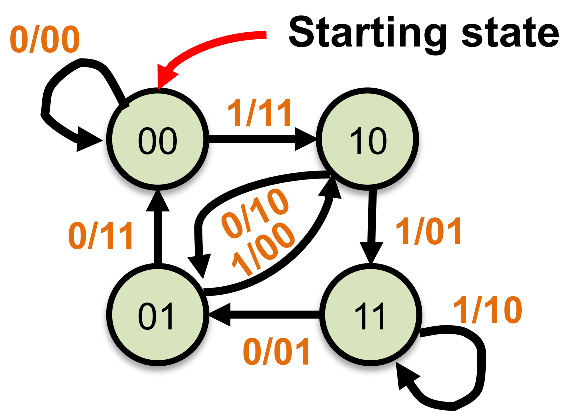
- Some errors have occurred

- *What's the 4-bit message?*

- **Most likely: 0111**
  - Message whose coded bits is **closest to received bits** in Hamming distance

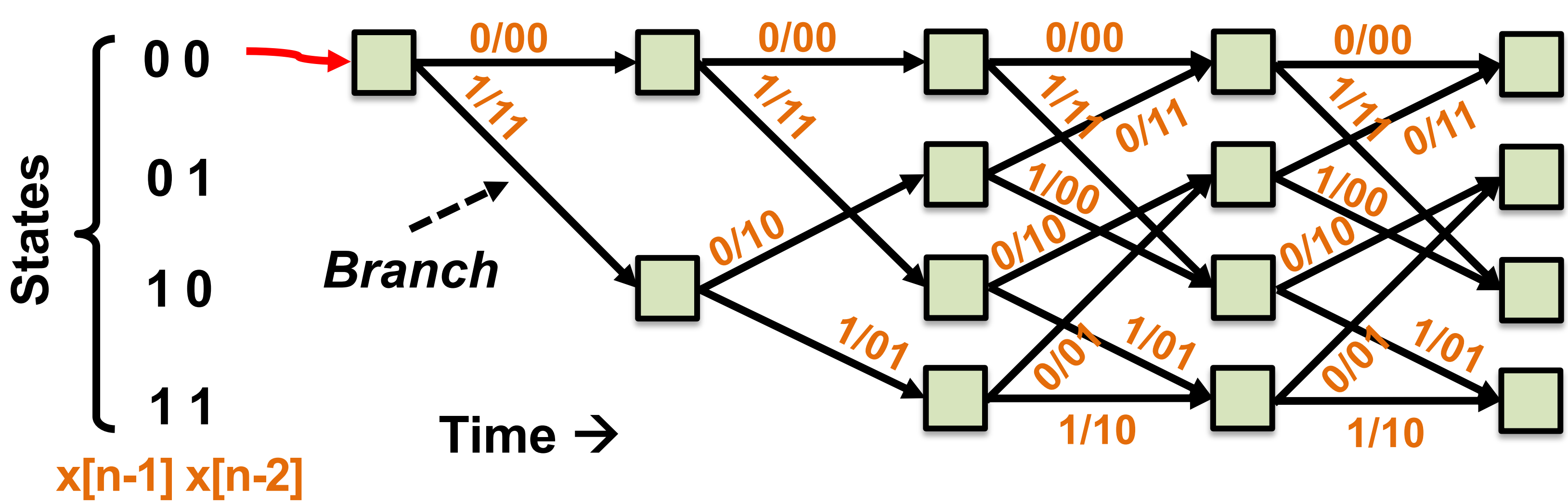
Message	Coded bits	Hamming distance
0000	000000000000	5
0001	000000111011	6
0010	000011101100	4
0011	000011010111	...
0100	001110110000	
0101	001110001011	
0110	001101011100	
0111	001101100111	**2**
1000	111011000000	
1001	111011111011	
1010	111000101100	
1011	111000010111	
1100	110101110000	
1101	110101001011	
1110	110110011100	
1111	110110100111	

The Trellis



- **Vertically,** lists encoder **states**
- **Horizontally,** tracks **time steps**
- ***Branches*** connect states in successive time steps

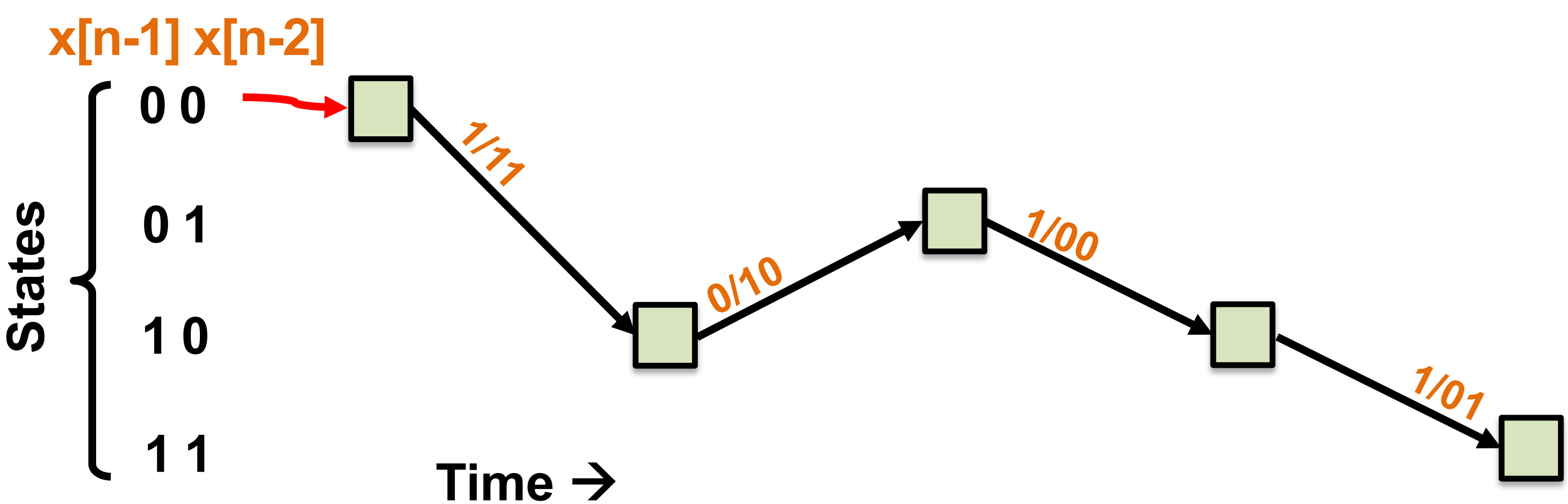
Trellis:



The Trellis: Sender's View

- **At the sender, transmitted bits** trace a unique, single ***path* of branches through the trellis**
  - *e.g.* transmitted data bits **1 0 1 1**

- **Recover transmitted bits ⟺ Recover path**



Viterbi algorithm


Andrew Viterbi (USC)

- **Want**: Most likely **sent bit sequence**

- Calculates **most likely path** through **trellis**

1. ***Hard input* Viterbi algorithm:** Have **possibly-corrupted** encoded **bits**, after reception

2. ***Soft input* Viterbi algorithm:** Have **possibly-corrupted likelihoods** of each bit, after reception
   – *e.g.*: “this bit is 90% likely to be a 1.”

Viterbi algorithm: Summary

- ***Branch metrics*** score **likelihood of each trellis branch**

- At any given time there are **$2^{K\text{-}1}$ most likely messages** we're tracking (one for each state)
  - One message ↔ one trellis path
  - ***Path metrics*** score **likelihood of each trellis path**

- **Most likely message** is the one that produces the **smallest path metric**

Today

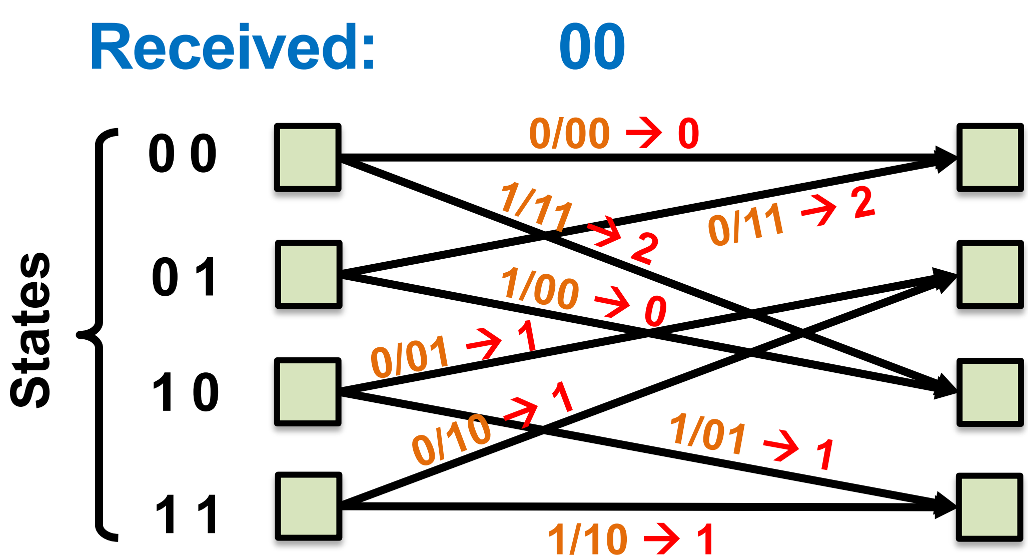
1. Encoding data using convolutional codes

2. **Decoding convolutional codes: Viterbi Algorithm**
   - **Hard input decoding**
   - Soft input decoding

Hard-input branch metric

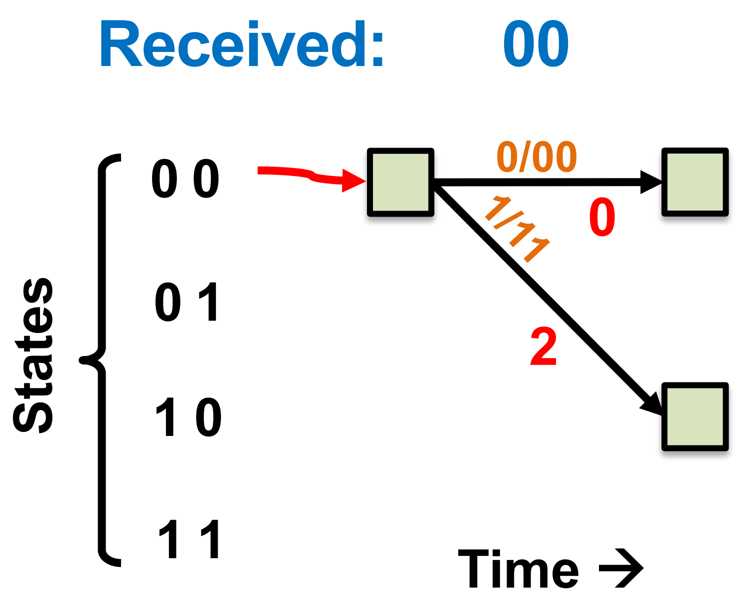
- Hard input → **input is bits**

- **Label every branch** of trellis with branch metrics
  - ***Hard input Branch metric:*** **Hamming Distance** between **received** and **transmitted** bits



Hard-input branch metric

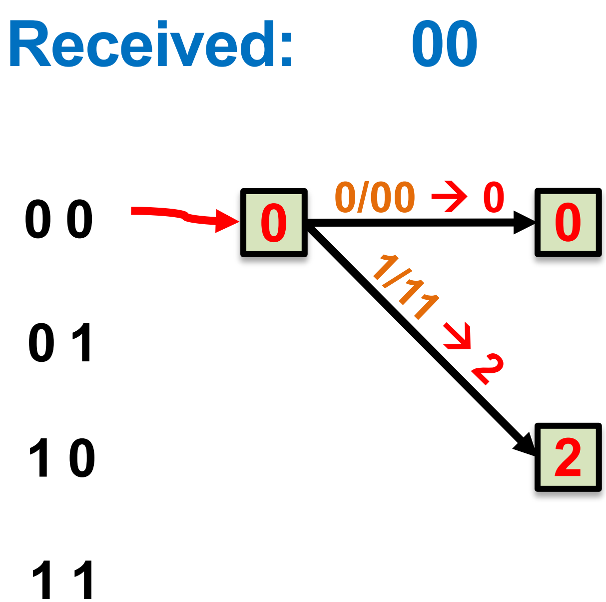
- Suppose we know encoder is in **state 00,** **receive bits: 00**



Hard-input path metric

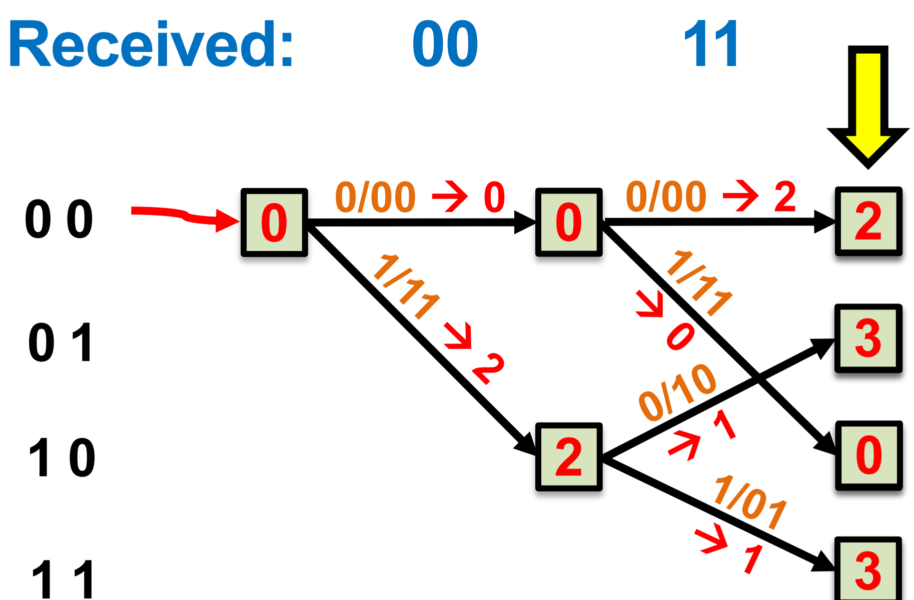
- ***Hard-input path metric:*** **Sum** Hamming distance between **sent** and **received bits along path**

- Encoder is initially in **state 00, receive bits: 00**



Hard-input path metric

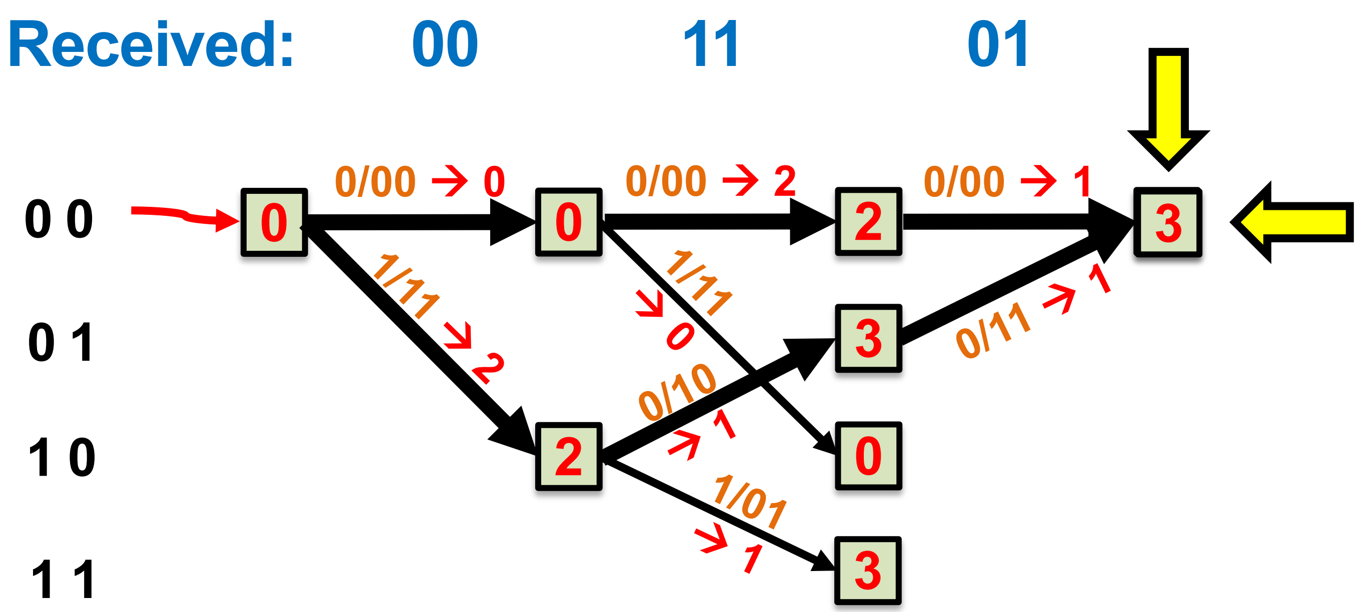
- Right now, each state has a **unique predecessor** state
- Path metric: Total bit errors **along path ending at state**
  - Path metric of **predecessor** + **branch metric**



Hard-input path metric

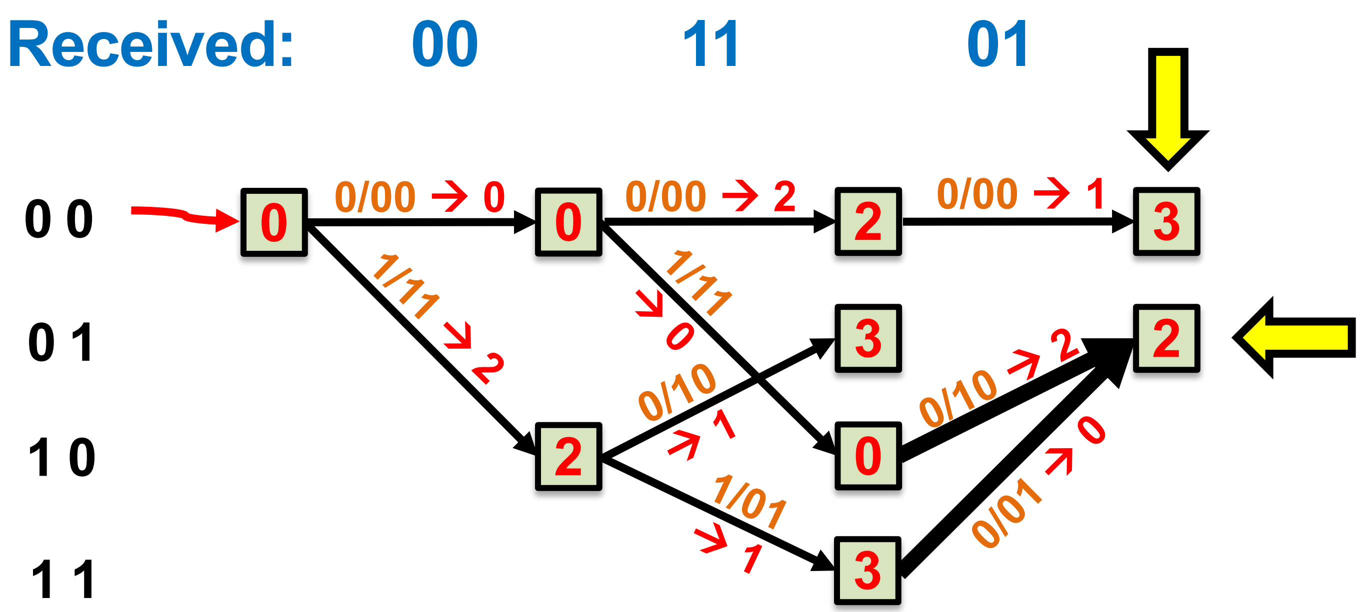
- Each state has **two predecessor states, two** ***predecessor paths*** (which to use?)

- **Winning** branch has **lower** path metric (**fewer** bit errors): ***Prune*** **losing** branch



Hard-input path metric

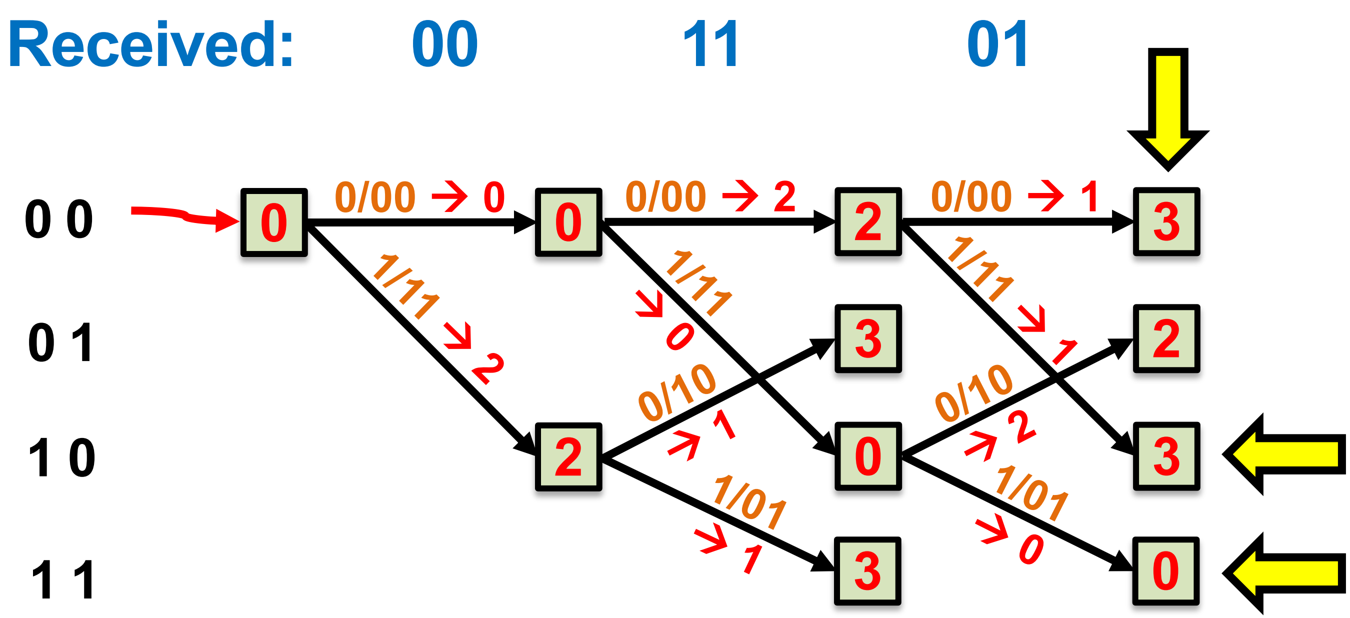
- Prune losing branch **for each state** in trellis



Pruning non-surviving branches

- ***Survivor path*** begins at each state, traces **unique path** back to **beginning** of trellis
  - **Correct path** is one of **four** survivor paths

- Some branches are not part of any survivor: **prune them**

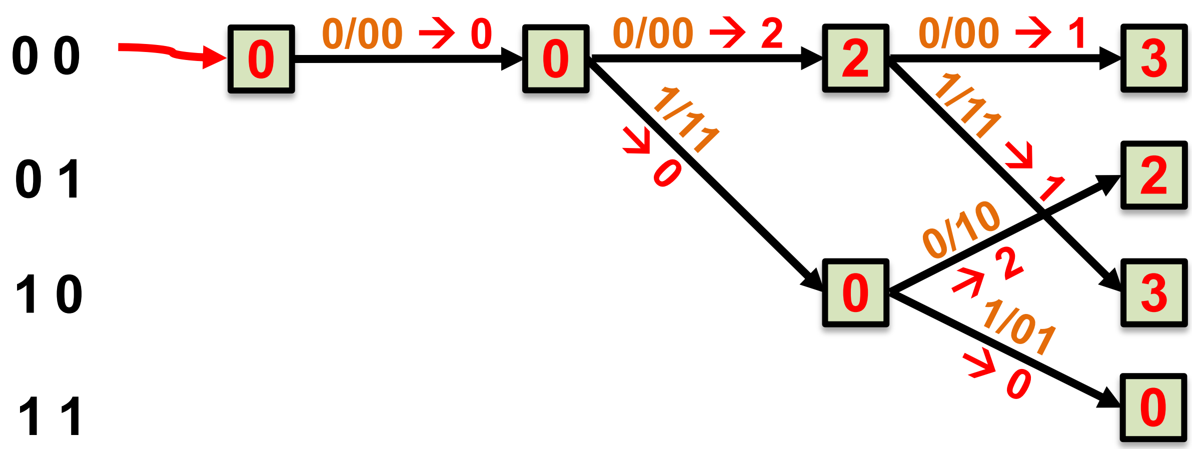


Making bit decisions

- When **only one branch remains** at a stage, the Viterbi algorithm **decides** that branch's **input bits:**

Received: 00 11 01

Decide: 0

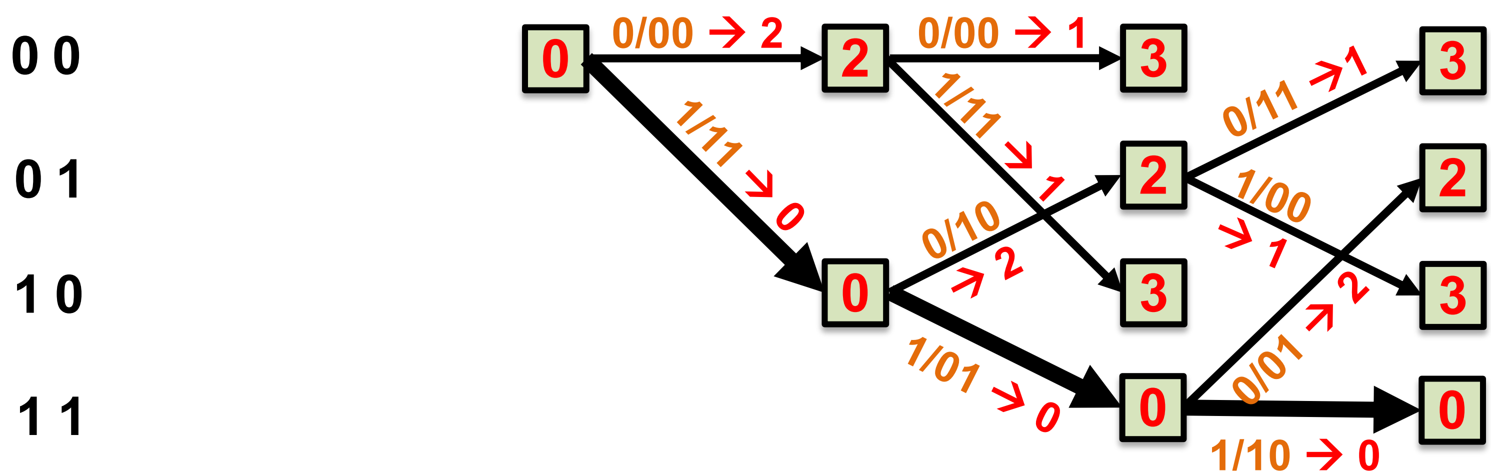


End of received data

- **Trace back** the survivor with **minimal path metric**
- Later stages **don’t get benefit** of future error correction, had data not ended

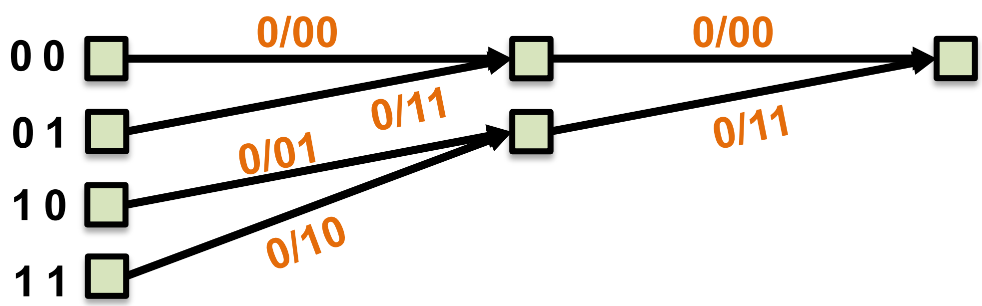
Received: 00 11 01 10

Decide: 0 1 1 1



Terminating the code

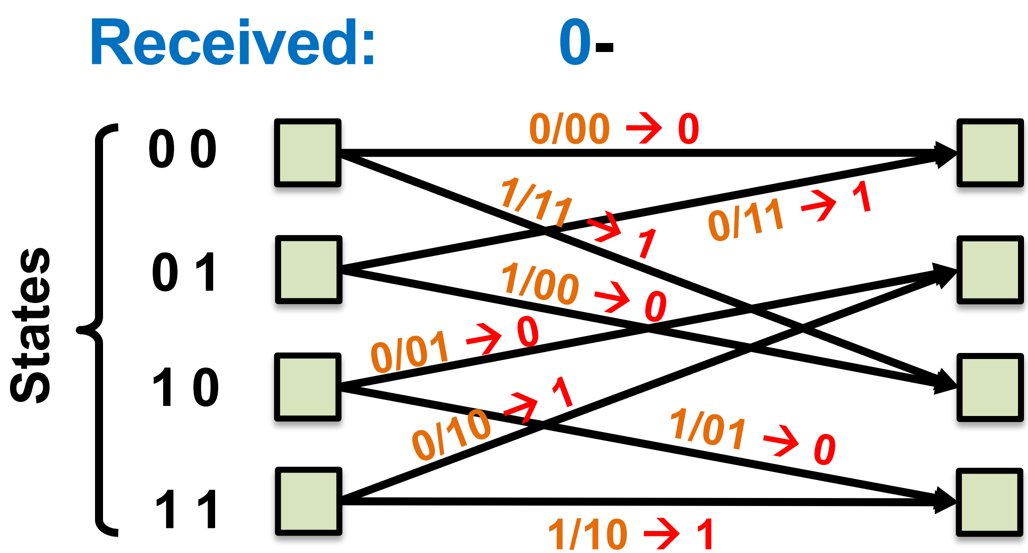
- **Sender** transmits **two 0 data bits** at end of data
- **Receiver** uses the following trellis at end:



- After termination only one trellis survivor path remains
  - Can make better bit decisions at end of data based on this sole survivor

Viterbi with a Punctured Code

- Punctured bits are never transmitted
- Branch metric measures dissimilarity only between **received and transmitted unpunctured bits**
  - Same path metric, same Viterbi algorithm
  - Lose some **error correction capability**



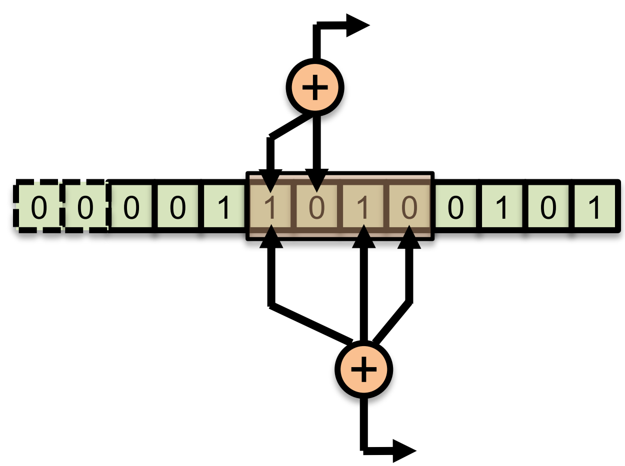
Today

1. Encoding data using convolutional codes

2. **Decoding convolutional codes: Viterbi Algorithm**
   - **Hard input decoding**
     - **Error correcting capability**
   - Soft input decoding

How many bit errors can we correct?

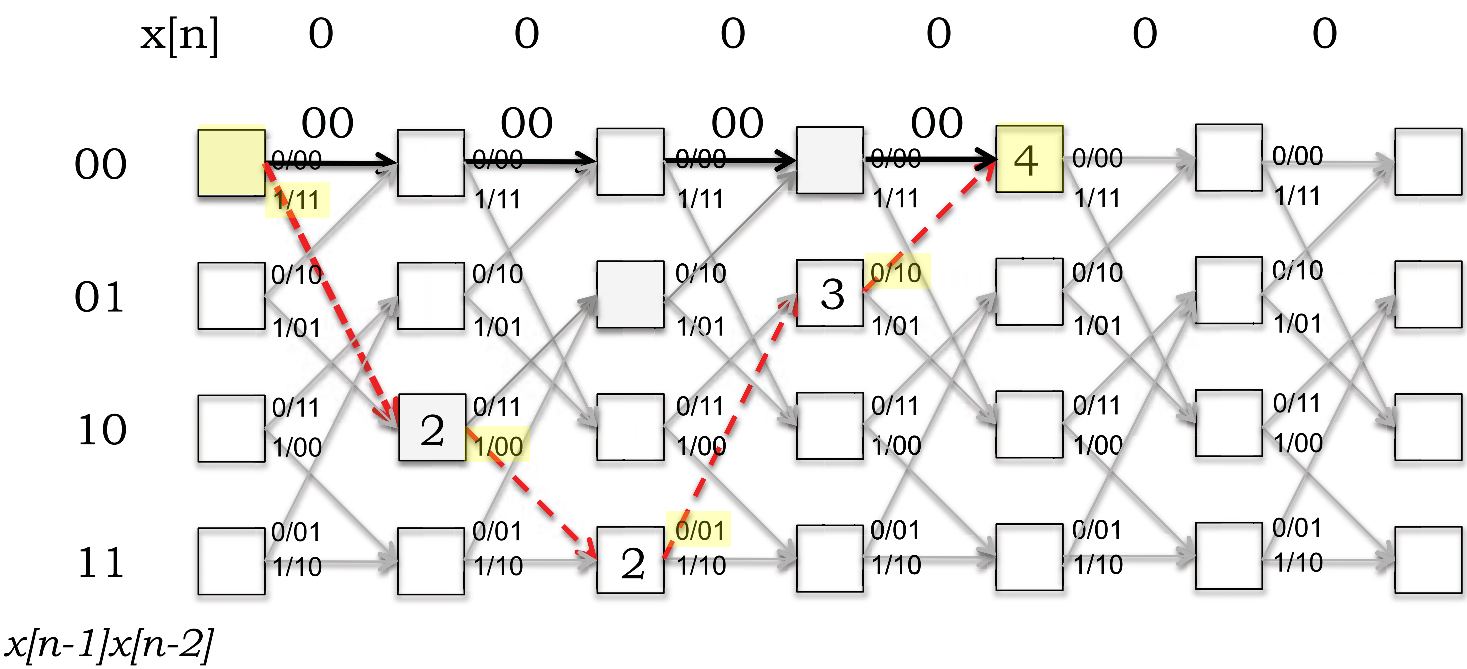
- Think back to the encoder; **linearity property:**
  - Message m_1 → Coded bits c_1
  - Message m_2 → Coded bits c_2
  - Message $m_1 \oplus m_2$ → Coded bits $c_1 \oplus c_2$



- So, d_{min} = minimum distance between **000...000** codeword and **codeword with fewest 1s**

Calculating d_{min} for the convolutional code

- Find path with **smallest non-zero path metric** going from **first 00** state to a **future 00 state**
- Here, d_{min} = 4, so can correct **1 error in 8 bits:**



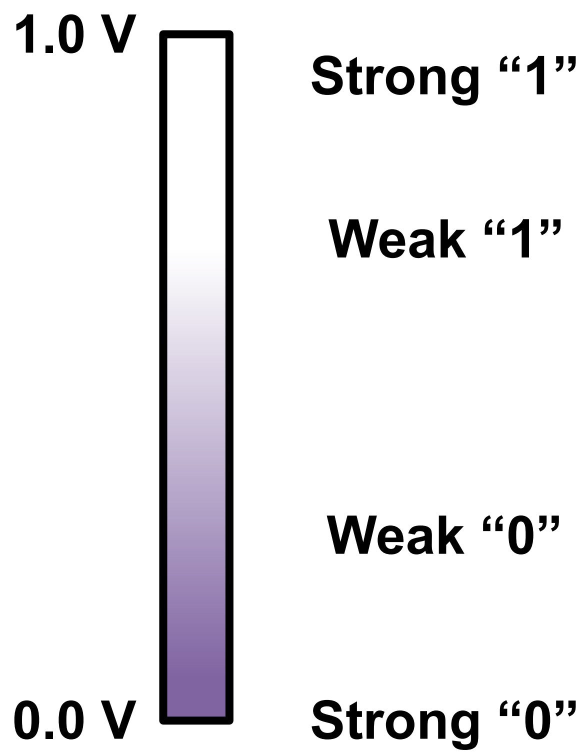
Today

1. Encoding data using convolutional codes
   – Changing code rate: Puncturing

2. **Decoding convolutional codes: Viterbi Algorithm**
   – Hard input decoding
   – **Soft input decoding**

Model for Today

- **Coded bits** are actually continuously-valued "voltages" between **0.0 V and 1.0 V:**



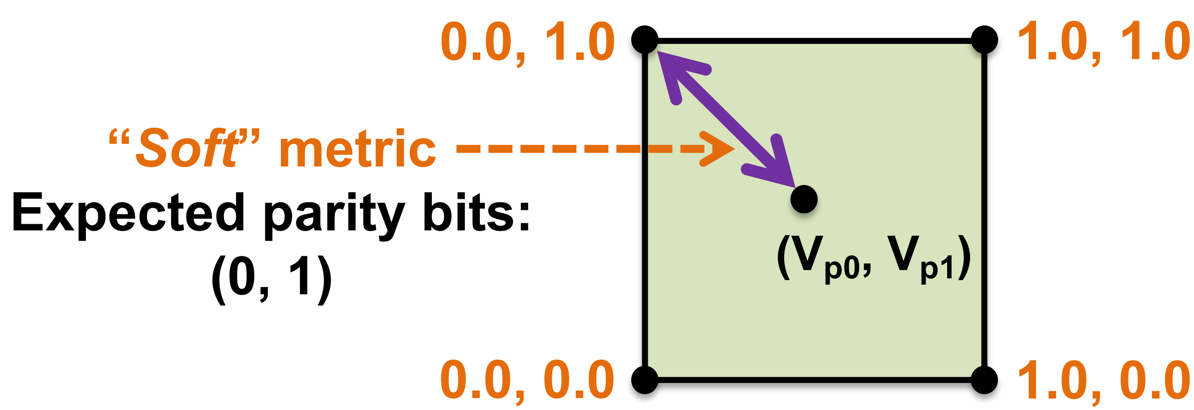
On Hard Decisions

- Hard decisions digitize each voltage to "**0**" or "**1**" by comparison against ***threshold voltage* 0.5 V**
  - **Lose information** about how "good" the bit is
    - Strong "1" (0.99 V) **treated equally to** weak "1" (0.51 V)
- **Hamming distance** for branch metric computation
- But **throwing away information** is almost never a good idea when making decisions
  - *Find a **better branch metric** that **retains information** about the received voltages?*

Soft-input decoding

- **Idea**: Pass received voltages to decoder before digitizing
  - **Problem:** Hard branch metric was Hamming distance

- *"Soft" branch metric*
  - **Euclidian distance** between received voltages and voltages of expected bits:



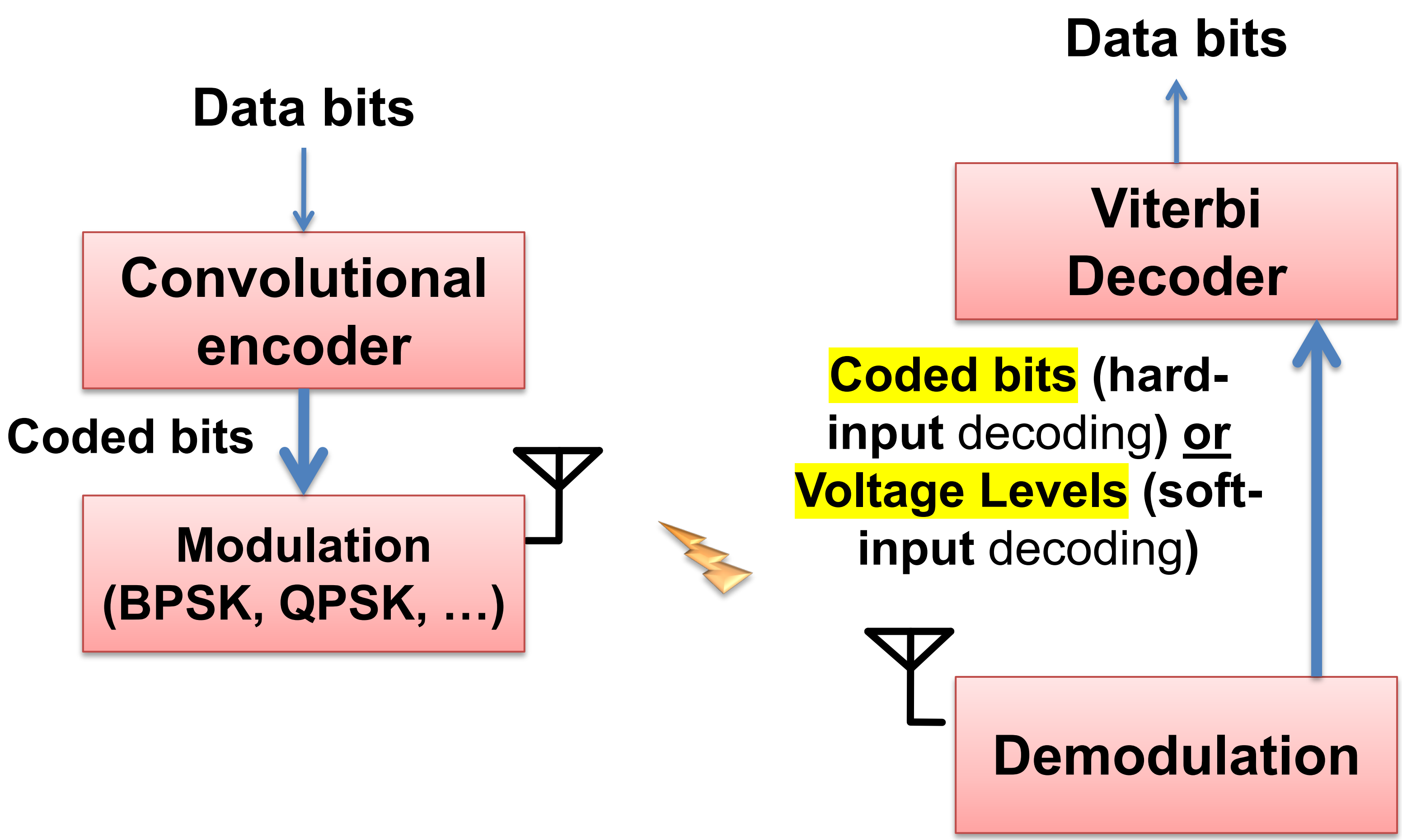
Soft-input decoding

- **Different** branch metric, hence **different** path metric
  - **Same** path metric computation

- **Same** Viterbi algorithm

- **Result:** Choose **path** that minimizes sum of squares of Euclidean distances between received, expected voltages

Putting it together: Convolutional coding in Wi-Fi



Thursday Topic:
Rateless Codes

Next week's Precepts:
Lab 2