BUS ENCODING TO PREVENT CROSSTALK DELAY

by

Bret Victor

Masters Thesis: The Director's Cut

May, 2001

College of Engineering

University of California, Berkeley

94720

Acknowledgments

A number of people contributed directly to the making of this project, and to them I am very grateful. I owe thanks to my advisor Professor Keutzer for supporting this work, as well as his lecture on the crosstalk problem which originally sparked the whole idea; Professor Brayton for reading and commenting on this report, as well as getting me interested in CAD in the first place; Professor Effros at Caltech for teaching me all I know (and much I've forgotten) about information theory and coding; and Brian Limketkai for his TEX template and advice, without which I would still be cursing loudly and ineffectually at the Microsoft Equation Editor.

I have noticed that many people use the Acknowledgements section as a general forum to name those who have played a meaningful role in their lives, if not the project specifically. A "shout-out to the homies", if you will. I originally resisted this idea, because I do not consider this project to be one of my more significant creations, certainly not worthy of dedication. However, I realized that although this may be a minor work, it represents a major event in my life—the end of my schooling, at least for now. I felt I shouldn't pass up this given opportunity to formally thank those who have helped me reach this juncture.

I owe thanks to Andy for good times, music and pie, and the constant willingness to indulge even the strangest of ideas; André for selflessness, always being aware of what the truly important things are, and the bits of wisdom that occasionally hit their mark; Brian for always being there to talk or just blather nonsensically; Pete for friendship while giving me a chance to feel like a mentor; and the Shumpound gang (Donna, Chris, Shumway, John, Albert, and Michele) for their tolerance.

Foremost and above all, I would like to thank my parents and Joanne for their love and support throughout the years. You have made me who I am.

Contents

Theorems and Lemons

Figures

Tables

Chapter 1

Introduction

In 1948, Claude Shannon published a paper that changed the world.

"A Mathematical Theory of Communication" is generally considered to have created the field of information theory. Shannon stunned the communication theory community with his proofs about compression and channel capacity, many of which were in contrast to generally accepted engineering principles. Almost all modern communication and information-processing systems, from computer hard drives to telephone networks to medical imaging machines, are based upon Shannon's ideas. Without Shannon, there would be no internet. Every step of an internet link, from the physical data transmission layer to the software that draws an image on your screen, owes its existence to Shannon's 1948 paper.

But, an actual examination of this paper reveals something interesting. The paper does not describe how to build a modem, or set up a wireless communication network. It does not explain how to compress an image or correct errors in data transmission. It does not even describe a method for generating a code that applies its concepts. In fact, it could be said that Shannon's paper does not say how to do *anything at all*.

How is this possible? How can one of the greatest papers ever written, one that revolutionized its field and so many others, not contain any practical information?

Shannon developed a theory. He invented some new concepts, such as *entropy* and *channel capacity*, and derived a mathematical framework around them. He added rigor and mathematical formalism to his subject in a way which nobody had ever considered. And he derived fundamental limits. Instead of just designing another communication system, he found the laws which all communication systems obey. In doing so, he told the engineers of the time that they could build things which they hadn't even thought were possible. Shannon changed the way people thought.

I am not Shannon, and the document that you are holding right now cannot claim nearly the scope nor import of Shannon's 1948 opus. Nevertheless, I like to think that it draws some inspiration from Shannon's philosophy. This philosophy can be stated as: "The key to understanding a new subject is developing a rigorous mathematical framework." Or, more simply: "Prove your theorems before you build your circuits."

This paper, like Shannon's, is about fundamental limits. Although there is a small amount of practical information, such as an actual design example, the primary thrust of this work is theoretical. This work represents, to the best of my knowledge, the exploration of a novel concept, and I feel that it is vital to have a solid theoretical grounding before attempting a practical design. Otherwise, you are designing blindly, never knowing if what you are doing is optimal or even possible.

With that said, it should be mentioned that I've always hated proofs. Throughout my education, I have loathed (to the point of occasionally skipping) homework problems whose first word was the dreaded imperative: "Prove". My natural tendency has always been to take a few data points, find a pattern, and throw my faith in the simplicity of nature. But, for the reasons stated above, I felt that this topic deserved special treatment, so I made an effort to revive my well-atrophied senses of rigor and formality.*

As with any description of new concepts to the uninitiated, if the material herein

* Perhaps, after this disclosure, the reader will forgive any apparent hand-waving in the proofs to come.

seems at all simple or obvious, it means only that I have succeeded in explaining it well. The process of discovery was long and uncertain, and how to get from one step to the next, or even figure out what the next step was, often was unclear. The road ahead was always foggy. Pages of mathematical derivations would lead nowhere; computer experiments would spit out data that had no apparent form or pattern whatsoever. But, all in all, the work was enormously enjoyable. There may have been dark and uncertain times, but the breakthroughs were epiphanous. They made it all worthwhile.

It is my hope that I can share some of this excitement with the reader. Although I certainly don't expect you to tear around the house screaming "It's Fibonacci!" as I may have allegedly done,* I do hope to impart some of the pleasure of finding mathematical beauty and order in the most unexpected places.

The remainder of this report is laid out as follows. Chapter 2 discusses why the topic is relevant, and provides an analysis of the physical behavior that is responsible for it. It also describes some of the other techniques that are currently being used. Once the background information and motivation have been established, it's on to the fun stuff! Chapter 3 provides an overview of the coding process and sets the scene for the theory that follows. Chapters 4, 5, and 6 develop the theory behind self-shielding codes. These chapters are named after the type of code discussed therein ("Unpruned Code with Memory", "Pruned Code with Memory", "Memoryless Code"), but the reader will notice that each of these types of codes requires the development of a distinct branch of the theory. Perhaps the chapter subtitles ("Counting the Neighbors", "Inspecting the Neighbors", "Connecting the Neighbors") give a better hint as to the true content of the chapters. Chapter 7 focuses on practical aspects of the coding process, discussing issues related to implementation. An example design is also presented. Finally, concluding thoughts and ideas for future work are given in Chapter 8.

* I wasn't really screaming.

Chapter 2

Motivation

2.1 Introduction

There was a time, not long ago, when IC designers didn't worry about interconnect delay. They focused on the speed of their transistors, and gates made out of those transistors, and logic blocks made out of those gates. The delay through these active components was calculated carefully and closely scrutinized. The metal that connected the components, on the other hand, was considered to be as ideal as the wire on the schematic diagram, and its effects were neglected. At the time, this attitude was acceptable, because in comparison to the delay through logic, interconnect delay was indeed negligible.

In modern semiconductor processes, sadly, this is no longer the case. For reasons that will be discussed, interconnect now plays a significant and sometimes prominent role in determining the speed of a circuit. Furthermore, this trend will only continue. If technology continues developing at its current rapid pace, techniques for analyzing interconnect delay and dealing with it will become not just attractive, but vital.

Parameter	Relation	Scaling
W, L, t_{ox}		$1/S$
V_{DD},V_T		$1/U$
C_{ox}	$1/t_{ox}$	S
C_{gate}	$C_{ox}WL$	$1/S$
k_n,k_p	$\mu C_{ox}W/L$	S
I_{sat}	$C_{ox}WV$	$1/U$
R_{on}	V/I_{sat}	1
intrinsic delay	$R_{on}C_{gate}$	$1/S$
power	$I_{sat}V$	$1/U^2$
power density	P/WL	S^2/U^2

Table 2.1 Variation of Device Parameters with Scaling (from [2]).

2.2 Interconnect Delay

This section will provide a brief and largely qualitative analysis of the issues responsible for the increasing significance of interconnect delay. This material is basically a summary of the excellent and thorough analysis presented in [2]; the interested reader is encouraged to look there (or [3], if necessary) for the complete treatment.

It is well recognized that technology, as quantified by the integration density and computational complexity of digital circuit designs, is improving at an extremely rapid pace. The much-touted Moore's Law predicts that these metrics double every eighteen months. There are two primary factors responsible for this technology growth—the scaling of device geometries and the increase in die size. Scaling refers to the shrinking of transistors with every process generation. Having smaller devices implies being able to fit more devices onto a chip, and thus more complex designs are realizable. But scaling results in a number of other effects related to speed and power consumption, as we will see momentarily. Die size refers to the physical area of the chip, and this number has been growing throughout the years. Bigger chips also lead directly to an increase in the amount of devices available and thus the design complexity.

Table 2.1 shows how some important device parameters vary with scaling. The data in the table is derived for short-channel CMOS devices, which constitute the majority of devices on a modern chip. We assume that all device dimensions are scaled down by the same factor S, and the supply voltage is scaled down by a factor U. Voltage scaling must be considered as a separate parameter, because not scaling the voltage leads to a dramatic rise in power consumption, whereas scaling the voltage at the same rate as the geometry results in generations of incompatible systems, as well as dangerously low threshold voltages and other circuit problems. The most important entry of the table for our purposes is the "intrinsic delay", which represents the time needed for a transistor to charge the gate capacitance of a fanout transistor. Overall circuit delay is roughly proportional to this number, and we see that it scales as $1/S$. This very effect is largely responsible for the massive increase in circuit speeds over the last ten years.

However, this only considers the speed of the active devices. The problem is that scaling affects interconnect delay in quite a different manner. Central to this problem is the existence of two different types of interconnect. Wires used inside a block of logic are called *local* interconnect, and their dimensions scale with everything else in the block. On the other hand, the wires that connect blocks on a chip are referred to as *global* interconnect, and their *length does not scale*. The reason for this is that on-chip blocks, while becoming more dense, are not necessarily moving closer together. In fact, with increasing die size, there may be blocks that need to communicate that are even further apart than was previously possible. Thus, the lengths of global interconnect wires are actually getting longer.

The effects of this phenomenon are shown in Table 2.2. S is again the geometry scaling factor, and S_L is the scaling factor for the length of the global wires, which is less than one because these lengths are growing, not shrinking. We see that the delay of a wire, which can be represented to first order by an RC time constant, is constant for local wires and actually increasing for global wires! [2] gives values of $S = 1.15$ and $S_L = 0.94$, which predict a 50% increase in global wire delay per year.

Parameter	Relation	Local Wire	Global Wire
W, H, t		$1/S$	$1/S$
L		$1/S$	$1/S_L$
C	LW/t	$1/S$	$1/S_L$
R	L/WH	S	S^2/S_L
RC	L^2/Ht	1	S^2/S_L^2

Table 2.2 Variation of Wire Parameters with Scaling (from [2]).

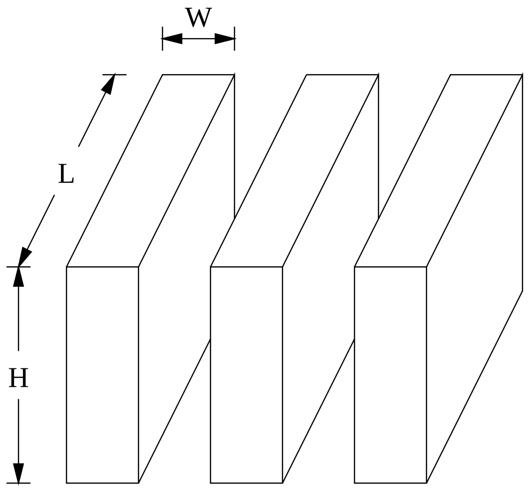


Figure 2.1 Illustration of Dimensions of Modern Global Interconnect.

This is in stark contrast to the decreasing delay trends followed by active devices, and explains why scaling and increasing die size are bringing the role of interconnect in circuit delay into the forefront.

This scaling approach clearly can lead to unacceptable results. Therefore, global wires are often scaled differently from the rest of the geometry. The width is generally scaled more slowly than S, and the thickness, or height, of the higher metal layers where global interconnect is typically routed is also scaled slowly or sometimes not at all. This reduced scaling attempts to preserve the cross-sectional area of the wires, and therefore prevent the detrimental increase in resistance responsible for the results in Table 2.2.

However, the reduced and non-uniform scaling of global wires causes two very important negative effects. The first is scarcity. Designs are becoming increasingly more complex, but if the pitch of global wires is not scaled, there is no increasing supply of interconnect to ship around the increasing amount of data. Thus, global routing resources are at a premium, and these wires cannot be wasted. The second effect is that, with scaling of wire thickness halted, the wires begin to take on a strange shape, as shown in Figure 2.1. Many modern designs have global wires which are taller than they are wide. Placing two of these wires near each other results in a structure which bears an uncomfortable resemblance to a parallel-plate capacitor. Capacitance between signal wires leads to crosstalk.

2.3 Crosstalk Analysis

Crosstalk is the term used to refer to a signal affecting another signal from which it should be isolated. Such unintentional coupling of signals can occur though a number of means. Current loops can create magnetic fields or even radiation that can be picked up inductively by remote wire loops. The finite resistance and inductance of the power supply wires can lead to signals coupling through the supply rails. But the most common source of crosstalk is capacitively coupled wires.

Every piece of metal in a circuit, unless it is completely enclosed in a grounded metal box, has a capacitive connection to every other piece of metal in the circuit. The specific value of the capacitance is dependent on a number of factors. Capacitance is inversely proportional to distance, so pieces that are further away have a smaller capacitance. If a grounded piece of metal is inserted between two pieces, most of the electric field lines from the two pieces will terminate on the grounded piece, and the capacitance will be greatly reduced.* A complete analysis of the capacitance between every wire in a circuit is simply intractable, even for almost trivially small circuits.

* On the other hand, if the piece inserted in the middle is not grounded, but floating, the capacitance between the two pieces will actually increase slightly, because the electric field lines can take a small shortcut and travel through the metal (dielectric constant effectively infinite) instead of through air (dielectric constant of one) or whatever the surrounding medium is. This effect has surprised many an engineer who did not study physics hard enough.

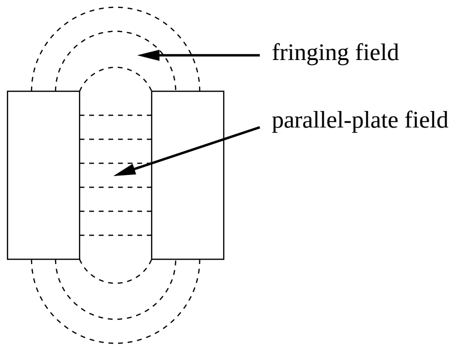


Figure 2.2 Electric Field Lines Between Cross-Coupled Wires.

Therefore, analysis of cross-coupled capacitances usually only considers neighboring wires.

Figure 2.2 shows two wires and their cross-coupling electric field lines. Notice that there are two main components to the field. The lines that go straight across, from the side of one wire to the other, are called the *parallel plate* field, and the lines that curve around from non-facing sides are called the *fringing field*. The parallel plate capacitance is typically easy to calculate:

$$C = \frac{\varepsilon_r \varepsilon_0 A}{d} \tag{2.1}$$

where A is the area of the plates, d is the distance between them, ε_0 is the permitivity of free space, and ε_r is the dielectric constant, or relative permitivity, of the material between the plates. On the other hand, the fringing capacitance is strongly dependent on the geometry of the wires and is usually only approximated, or even neglected when possible. The important thing to notice for the subject at hand is that traditionally-sized flat wires are mostly restricted to coupling through fringing capacitance, but the oddly-scaled global wires, as described above, can couple strongly through parallel-plate capacitance.

Now that the source of crosstalk has been discussed, we will take a look at its effects. The consequences of crosstalk are different for each circuit style. In analog circuit design, crosstalk can be modeled as a noise source. The circuit must be designed robust enough to meet the specifications even in the presence of this noise.

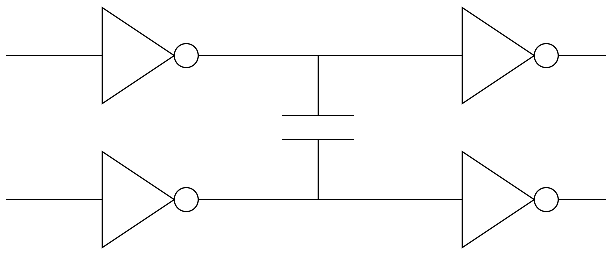


Figure 2.3 Digital Circuit with Crosstalk.

In the digital realm, crosstalk can lead to malfunctioning systems in design styles that are sensitive to spurious glitches on wires, such as asynchronous logic and precharge or "domino" logic. For standard CMOS synchronous digital designs, which is the focus in this work, crosstalk manifests itself as a delay.

The clearest way to see this is to plunge right into some circuit analysis. The circuit shown in Figure 2.3 represents the type of situation that we are concerned with. Two digital gates drive two other digital gates, with a cross-coupled capacitance present between the two wires. We can linearize this circuit by replacing the driving gates with their Thévenin equivalent voltage sources and series resistances, and the fanout gates by their load capacitance. Characterizing the fanout with a capacitance is quite accurate, but linearizing the driver is more approximate, since transistors are not linear devices. However, it is a fairly good approximation, especially for the first half of a transition. ([2] discusses the validity of the "R_{on} transistor model".) The most serious simplification is ignoring the distributed-rc nature of the wires and using lumped resistors and capacitors. However, any model that takes this into account almost certainly cannot be subjected to hand analysis. We must use a small number of lumped elements in order to keep the analysis tractable.

Figure 2.4 shows the circuit that we will analyze. We will first consider the behavior when one wire makes a low-to-high transition and the other wire makes a high-to-low transition. This is expressed by the following initial conditions:

$$\begin{aligned} V_1(0) &= 0, \qquad & V_1(\infty) &= V_{DD} \\ V_2(0) &= V_{DD}, \qquad & V_2(\infty) &= 0 \end{aligned} \tag{2.2}$$

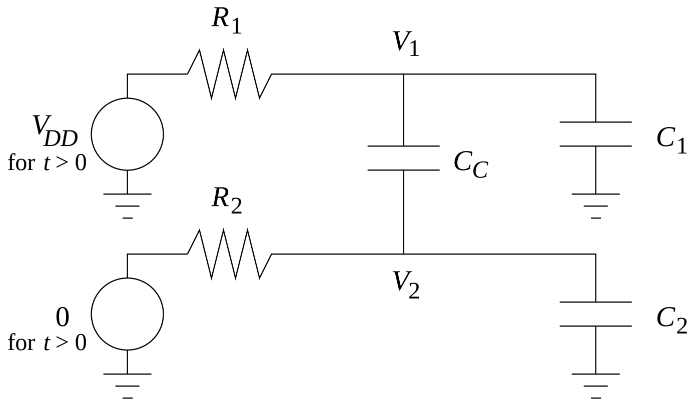


Figure 2.4 Linearized Model of Circuit with Crosstalk.

Defining I_1 to be the current traveling from left to right across R_1, and defining I_2 similarly for the current across R_2, we can write:

$$I_1 = \frac{V_{DD} - V_1}{R_1} = C_1 \frac{dV_1}{dt} + C_C \frac{d(V_1 - V_2)}{dt} \tag{2.3}$$

$$I_2 = \frac{-V_2}{R_2} = C_2 \frac{dV_2}{dt} + C_C \frac{d(V_2 - V_1)}{dt} \tag{2.4}$$

We can rewrite the above equations as

$$R_1(C_1 + C_C)\frac{dV_1}{dt} + V_1 = R_1 C_C \frac{dV_2}{dt} + V_{DD} \tag{2.5}$$

$$R_2(C_2 + C_C)\frac{dV_2}{dt} + V_2 = R_2 C_C \frac{dV_1}{dt} \tag{2.6}$$

There are a number of methods for solving a system of differential equations such as this one. But probably the easiest method is to simply guess at a general form of the solution, plug it in, and try to evaluate coefficients.[*] Every electrical engineer knows that a simple RC circuit, such as what we would have in Figure 2.4 if we removed C_C, has a solution of the form

$$V(t) = k_c + ke^{-t/RC} \tag{2.7}$$

We might guess that our solution has a similar form, but with an additional time constant because of the coupling capacitor. Furthermore, the way the two equations

[*] This tends to be the easiest method only if you are a good guesser.

are related suggests that the pair of time constants must be the same for both $V_1(t)$ and $V_2(t)$. Thus, we will try:

$$V_1(t) = k_{c1} + k_1 e^{-t/a} + k_{1b} e^{-t/b} \tag{2.8}$$

$$V_2(t) = k_{c2} + k_2 e^{-t/a} + k_{2b} e^{-t/b} \tag{2.9}$$

The first step is to eliminate some of the coefficients by applying the initial conditions in (2.2). Evaluating the above expressions at $t = 0$ and $t = \infty$ and equating to the proper initial conditions gives us:

$$\begin{aligned} k_{c1} = V_{DD}, &\qquad k_{c1} + k_1 + k_{1b} = 0 \\ k_{c2} = 0, &\qquad k_{c2} + k_2 + k_{2b} = V_{DD} \end{aligned} \tag{2.10}$$

We can therefore rewrite (2.8) and (2.9) as

$$V_1(t) = V_{DD} + k_1 e^{-t/a} - (k_1 + V_{DD}) e^{-t/b} \tag{2.11}$$

$$V_2(t) = k_2 e^{-t/a} - (k_2 - V_{DD}) e^{-t/b} \tag{2.12}$$

We now plug these guesses for $V_1(t)$ and $V_2(t)$ into (2.5) and (2.6). The resulting mess is not shown, but the astute reader will notice that, after cancelling out some constant terms, all terms in the result will be proportional to either $e^{-t/a}$ or $e^{-t/b}$. Therefore, each differential equation produces two equations relating our coefficients, one from equating the $e^{-t/a}$ terms and one from equating the $e^{-t/b}$ terms. After some algebraic manipulation, these equations can be written as follows:

$$\begin{aligned} &\textit{from } e^{-t/a} \textit{ terms in (2.5):} &\quad a &= R_1\left(C_1 + \left(1 - \frac{k_2}{k_1}\right)C_C\right) \\ &\textit{from } e^{-t/a} \textit{ terms in (2.6):} &\quad a &= R_2\left(C_2 + \left(1 - \frac{k_1}{k_2}\right)C_C\right) \\ &\textit{from } e^{-t/b} \textit{ terms in (2.5):} &\quad b &= R_1\left(C_1 + \left(1 - \frac{k_2 - V_{DD}}{k_1 + V_{DD}}\right)C_C\right) \\ &\textit{from } e^{-t/b} \textit{ terms in (2.6):} &\quad b &= R_2\left(C_2 + \left(1 - \frac{k_1 + V_{DD}}{k_2 - V_{DD}}\right)C_C\right) \end{aligned} \tag{2.13}$$

We now have four equations and four unknowns (a, b, k_1, and k_2), so this system possibly has a solution. Equating the two equations for a, equating the two equations for b, and then combining the results gives us the following relation between k_1 and k_2:

$$k_1 = k_2 \frac{R_2(C_2 + C_C) - R_1(C_1 + 3C_C)}{R_2(C_2 + 3C_C) - R_1(C_1 + C_C)} - V_{DD} \frac{R_2(C_2 + 2C_C) - R_1(C_1 + 2C_C)}{R_2(C_2 + 3C_C) - R_1(C_1 + C_C)} \quad \textbf{(2.14)}$$

After this point, it gets ugly. The system can indeed be solved, but the result is a dense mass of parameters and square roots which is entirely too complex for our purposes.*

In an attempt to make the result manageable, we will make an assumption. Looking at Figure 2.3, we might imagine that most of the time, the two driving gates will be about equal in strength, and the two fanout gates will provide about the same loading capacitance. In terms of our circuit model, these assumptions translate to:

$$R = R_1 = R_2 \quad \textbf{(2.15)}$$

$$C = C_1 = C_2 \quad \textbf{(2.16)}$$

These assumptions clean up the equations enormously. Looking back at (2.14), we see that it immediately collapses to

$$k_1 = -k_2 \quad \textbf{(2.17)}$$

Plugging this information into (2.13), we find a surprising result:

$$a = RC + 2RC_C$$
$$b = RC + 2RC_C \quad \textbf{(2.18)}$$

When we set the resistances and capacitances equal, a and b are the same, and there is only one time constant. Therefore, our final solution is

$$V_1(t) = V_{DD}\left(1 - e^{-t\,/\,RC+2RC_C}\right) \quad \textbf{(2.19)}$$

$$V_2(t) = V_{DD}\, e^{-t\,/\,RC+2RC_C} \quad \textbf{(2.20)}$$

* Not complex in the mathematical sense. As long as the resistances and capacitances are non-negative, this circuit will not oscillate.

Before we examine this result, we will solve another case so we have something to compare with. The preceding equations describe the situation when one wire makes a low-to-high transition and the other makes a high-to-low transition. Now, we will derive the result for when the first wire makes a low-to-high transition and the other attempts to stay at a constant voltage. The voltages that are being solved for will be renamed V_{s1} and V_{s2} in order to keep them distinct from the previous results. The differential equations (2.5) and (2.6) still apply, but we now have slightly different initial conditions:

$$\begin{aligned} V_{s1}(0) &= 0, \qquad V_{s1}(\infty) = V_{DD} \\ V_{s2}(0) &= 0, \qquad V_{s2}(\infty) = 0 \end{aligned} \tag{2.21}$$

Evaluating our guesses in (2.8) and (2.9) with these conditions gives us the following:

$$V_{s1}(t) = V_{DD} + k_1 e^{-t/a} - (V_{DD} + k_1)e^{-t/b} \tag{2.22}$$

$$V_{s2}(t) = k_2 e^{-t/a} - k_2 e^{-t/b} \tag{2.23}$$

Notice that (2.23) is subtly different from (2.12). We plug these equations into (2.5) and (2.6), and after a similar procedure as above, including making the same assumption, we come out with a solution for the coefficients:

$$\begin{aligned} a &= RC \\ b &= RC + 2RC_C \\ k1 = k2 &= \frac{1}{2}V_{DD} \end{aligned} \tag{2.24}$$

Now, there are indeed two different time constants, and we no longer have to feel quite so silly for our guesses in (2.8) and (2.9). The final result is:

$$V_{s1}(t) = V_{DD}\left(1 - \frac{1}{2}e^{-t/RC+2RC_C} - \frac{1}{2}e^{-t/RC}\right) \tag{2.25}$$

$$V_{s2}(t) = \frac{V_{DD}}{2}\left(e^{-t/RC+2RC_C} - e^{-t/RC}\right) \tag{2.26}$$

We now have solutions for a wire transitioning next to an oppositely transitioning wire and transitioning next to a steady signal. These two solutions, $V_1(t)$ and $V_{s1}(t)$,

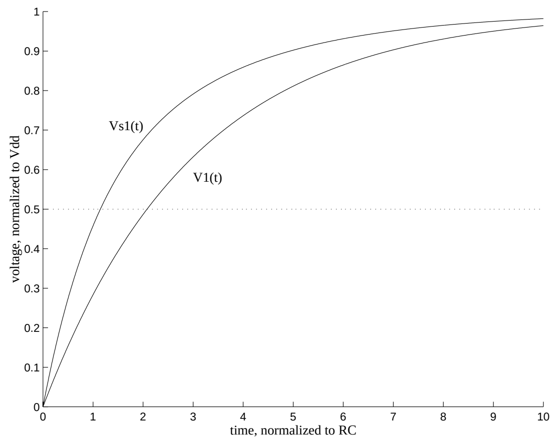


Figure 2.5 $C_C = C$.

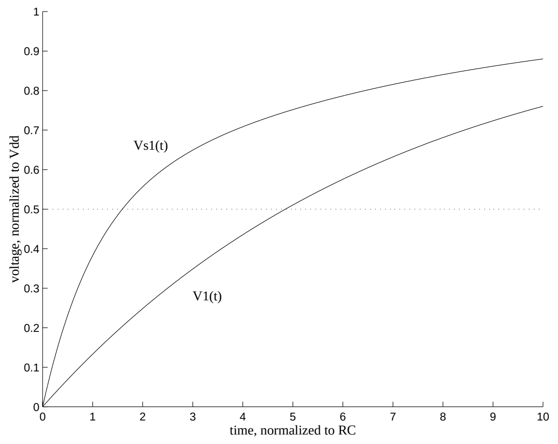


Figure 2.6 $C_C = 3C$.

are plotted in Figures 2.5 and 2.6 for two values of C_C. Clearly, the activity on the other wire is significantly affecting the rise time of the signal. In order to quantify this, we can attempt to calculate the time at which each signal crosses the halfway point $V_{DD}/2$. We denote this time as t_1 and t_{s1} for the two signals respectively. That is,

$$V_1(t_1) = V_{s1}(t_{s1}) = \frac{1}{2}V_{DD} \qquad \textbf{(2.27)}$$

Because $V_1(t)$ has only one time constant, the calculation is easy:

$$t_1 = \ln(2)\,(RC + 2RC_C) \qquad \textbf{(2.28)}$$

Unfortunately, there is no closed-form solution for t_{s1}. However, in this age of powerful computers, that does not stop us from calculating it numerically and making a plot. Figure 2.7 plots t_1/t_{s1} versus C_C/C, or the ratio of the delays versus the relative value of the coupling capacitance, and is probably the most important plot in the chapter. We see that the delay ratio goes almost linearly with the relative coupling capacitance. The qualitative conclusion that we can draw from all of this is that when the coupling capacitance becomes comparable to the load capacitance, two wires that attempt to transition in opposite directions receive a stiff penalty in delay time. We call this penalty the *crosstalk delay*, and the next 100 pages will be dedicated to finding a design approach to prevent it.

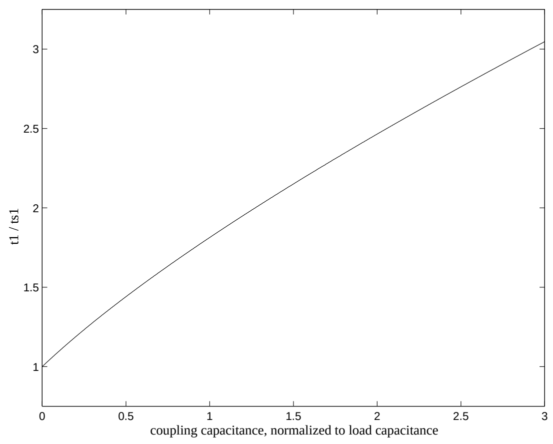


Figure 2.7 t_1/t_{s1} versus C_C/C.

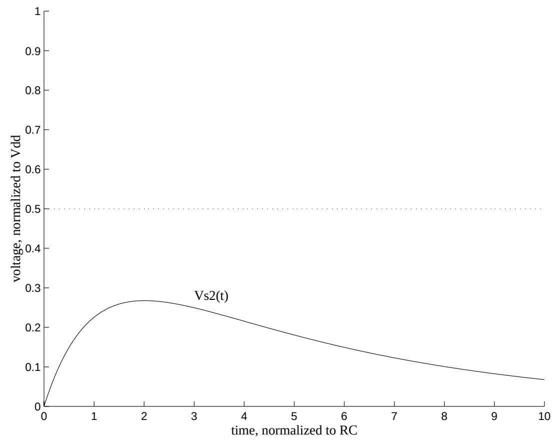


Figure 2.8 $V_{s2}(t)$ for $C_C = 2C$.

Before we move on, it is interesting to take a look at $V_{s2}(t)$, which describes the behavior of the so-called steady wire. Examining Figure 2.8, we see that this voltage is clearly affected by the transition on the other wire. However, this phenomenon usually creates no problems in a digital circuit, unless the height of the pulse rises above the digital switching threshold. In our model, V_{s2} can never reach $V_{DD}/2$ for any value of C_C, although it's possible that it could with a more complicated model.

2.4 Combating Crosstalk

In a synchronous circuit, the clock speed is limited by the slowest block of logic between sequential elements, which is known as the *critical path*. If crosstalk delay occurs on this critical path, the clock speed must be lowered in order to accommodate it, and therefore the entire circuit slows down. The recent years have seen an increase in the value of C_C/C for reasons explained in Section 2.2, and thus there is currently considerable interest in methods for preventing or dealing with crosstalk.

In the literature there are a number of techniques designed for combating crosstalk. Many of them are designed for minimizing crosstalk delay within a datapath or logic block. With the appropriate models and approximations, it is possible to incorporate crosstalk analysis into static timing analysis, and devise a routing scheme that uses this information to minimize the critical path with crosstalk considered.

Such schemes are given in [4], [5], and [6].

However, as we saw in Section 2.2, the primary source of interconnect delay problems is not local wires, but global wires. Creative routing schemes within a logic block will not solve the problem of crosstalk on a long cross-chip bus. As scaling continues and the delay across global wires increases, it is increasingly likely that the critical path will lie on a long bus.

[7] and [8] give some techniques for avoiding crosstalk delay on a long bus. One technique is skewing the timing of signals on the bus. This involves introducing an intentional delay into the drivers on every other wire. The result is that wires are active only when their neighbors are quiet. The problem with this approach, other than the intrinsic delay of the skewing, is that it requires careful technology-dependent circuit design, and brings up timing issues that are susceptible to process variation. It would be preferable to consider a method that fits better into a synchronous framework.

One such technique is bus interleaving. Two buses are routed such that each wire on a bus is adjacent only to wires on the other bus. If the two buses are mutually exclusive, perhaps a read and a write port for example, crosstalk delay will never occur. This is an excellent technique, but it requires mutually exclusive buses. For a bus that can transition on any and every clock cycle, this can't be used.

Another technique that is mentioned is precharging. On one phase of the clock, all of the wires on the bus are driven to a high value. On the next phase, the appropriate wires are driven low. This avoids crosstalk delay, but wastes an enormous amount of energy. The capacitance on a long bus is large enough that precharging it on every clock cycle may incur detrimental costs in power consumption. It is usually not an option.

Probably the most common technique is simply using large repeaters to drive the crosstalk capacitance through brute force [9]. This is somewhat equivalent to reducing the value of R in the previous section's analysis, and does result in a reduction in the delay. The problem with this method is that it is also power-hungry, and every repeater requires the bus to make its way from the high metal layers down to the

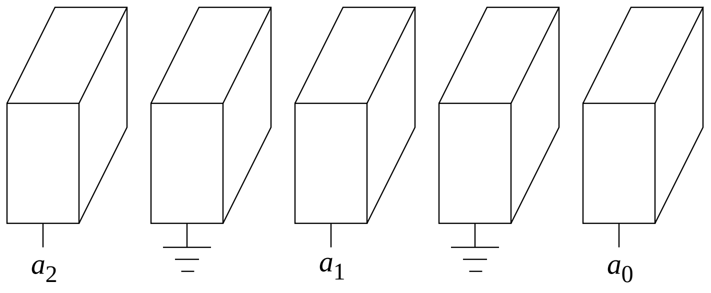


Figure 2.9 Illustration of Shielding.

active layer and back through stacks of vias. It also requires technology-dependent tuning, is susceptible to process variation, and is a rather inelegant solution.

The simplest and most effective technique for preventing crosstalk delay is shielding. This involves putting a power supply wire, either ground or V_{DD}, between every wire on the bus, as shown in Figure 2.9. These constant-voltage wires act as electromagnetic shields, and prevent activity on one signal wire from significantly coupling over to another signal wire. The only problem with shielding is that it wastes wires. When the wires in question are the scarce, unscaled wires reserved for global interconnect, it can be difficult to justify doubling the channel width. Nevertheless, this technique has been used in some high-speed designs where crosstalk delay on the bus would have limited the clock speed.

The next six chapters will discuss a novel alternative to these techniques. This new method is based on the observation that, although interconnect delay is increasing, logic delay is still decreasing rapidly. As processing power becomes cheaper, both in terms of area and speed, it makes more and more sense to solve problems with logic instead of circuit tricks. Instead of trying to creatively route the wires, or play with timing, or overpower the crosstalk, we can actually consider *changing the data itself*. If we change the data in the right way, we can eliminate crosstalk delay by design. This is called encoding.

Chapter 3

Self-Shielding Codes

3.1 Introduction

Coding is the art of making the data fit the channel. A data compression code, for instance, is used when the channel is assumed to be slow or costly. Data compression works by finding and removing redundancy in the data, condensing it into the smallest representation that still contains enough of the original information. This minimizes channel use when the data is transmitted. An error correcting code, on the other hand, is used when the channel is assumed to be noisy. It introduces intentional redundancy into the data, so that the original information can be recovered even if there are errors in transmission. On the surface, it appears that these two types of codes, one designed to remove redundancy, the other to add it, are performing opposite tasks. However, both codes are both working toward the same goal—packaging information so it meets the constraints of the channel. The type of code that we explore here will be no different.

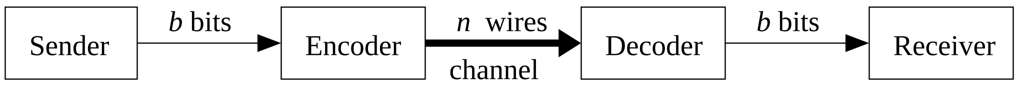


Figure 3.1 Model of Communication Chain.

3.2 Modeling the Communication Chain

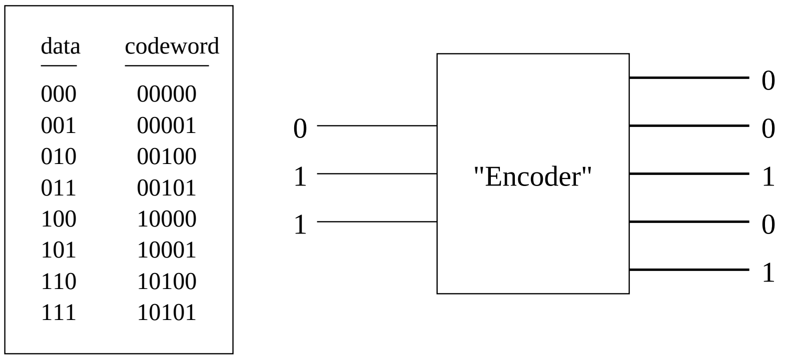
At the highest level, coding simply isolates data from the communication medium. In much the same manner as a snacker uses two slices of bread to avoid eating gobs of peanut butter by hand, the encoder and decoder form a channel sandwich, preventing raw data from being sent over the channel.

This is illustrated in Figure 3.1. The block diagram can be interpreted as follows: The sender presents the encoder with a b-bit data word. The encoder then drives the channel, which consists of an array of n wires. The decoder reads the channel and presents the receiver with the same b-bit data word that was given by the sender.

We must put some additional stipulations on this model before we develop our codes. Because we are modeling a generic bus, we assume that the sender can send a different data word on each clock cycle, and all of the 2^b possible words are valid. The model specifies nothing about the latencies of the encoder and decoder, as long as the throughput is one data word per clock cycle. However, it is implied that the propagation delay across the channel is long or even critical, and this is why we are encoding it in the first place.

3.3 The Fundamental Rule

Consider the simplest anti-crosstalk technique discussed in Chapter 2—shielding. This merely consists of placing a grounded "shield" wire between every data wire in order to electromagnetically isolate the signals. But we can, in fact, think of shielding as a code, and represent it with our model in Figure 3.1. The channel width n is $2b - 1$. The encoder encodes a 0 bit in the data word with a $\{0\,0\}$ signal on two channel wires, and a 1 bit with a $\{1\,0\}$ signal. The decoder performs the reverse mapping, which is effectively the same as examining every other wire. This is illustrated in Figure 3.2.



data	codeword
000	00000
001	00001
010	00100
011	00101
100	10000
101	10001
110	10100
111	10101

Figure 3.2 Shielding Viewed As A Code.

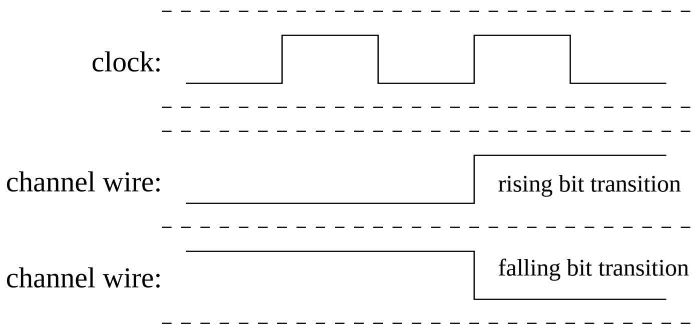


Figure 3.3 Timing Diagram of the Undesired Event.

This is a code, and according to Section 3.1, every code exists in order to meet a channel constraint. What is the channel constraint in this case? In Section 2.3, we learned that “crosstalk delay”, which could potentially limit the clock speed, arises when signals on two adjacent wires simultaneously transition in opposite directions. This situation is depicted in Figure 3.3. Studying the shielding code, we see that such an event is forbidden by design. By eliminating the possibility of any transition whatsoever on every other wire, there can never be a rising transition next to a falling one, and thus crosstalk delay cannot occur.

Shielding is a “code” to overcome this channel constraint, but it is not necessarily a good one. In Section 2.2, we learned that wires at higher-level metal layers, where such a bus would normally be routed, are scarce. Each wire in the channel consumes expensive routing resources. Therefore, it makes sense to take channel width to be

codeword at time 1:	0010	0000	0100	0100	0010
	↓	↓	↓	↓	↓
codeword at time 2:	0110	1111	0001	0010	0100
	valid	*valid*	*valid*	**invalid**	**invalid**

Figure 3.4 Examples of Valid and Invalid Transitions.

our cost metric. Doing so, we find that this code performs rather poorly—it requires almost twice as many wires as data bits. One might guess that better encodings exist, codes that meet the same channel constraint, but require fewer wires. One would be right.

We call such codes "self-shielding" or "crosstalk-immune". These are codes that fit the communication model described in Section 3.2, plus abide by the following Fundamental Rule. The Fundamental Rule represents our channel constraint, and this simple statement is responsible for every theorem and derivation in this work.

Fundamental Rule of Self-Shielding Codes: *For any two codewords W_1 and W_2, W_2 may not be placed on the channel immediately following W_1 if doing so would cause a rising bit transition adjacent to a falling bit transition.*

By design, a properly implemented self-shielding code can never cause crosstalk delay on the channel. Figure 3.4 gives some examples of valid and invalid transitions.

3.4 Coding Terminology

Consider again the block diagram of the communication chain in Figure 3.1. We say that the data words provided by the sender are represented by *symbols*. The use of the word *symbol* emphasizes that the specific values of the data words are irrelevant to the code. In our case, the only property of the data words that we care about is their quantity, because this determines the size of the symbol set.* Representing data words with symbols frees us from having to consider specific data values during

* Other encoding schemes, such as those used for data compression, may care about the statistical properties of the symbol set as well.

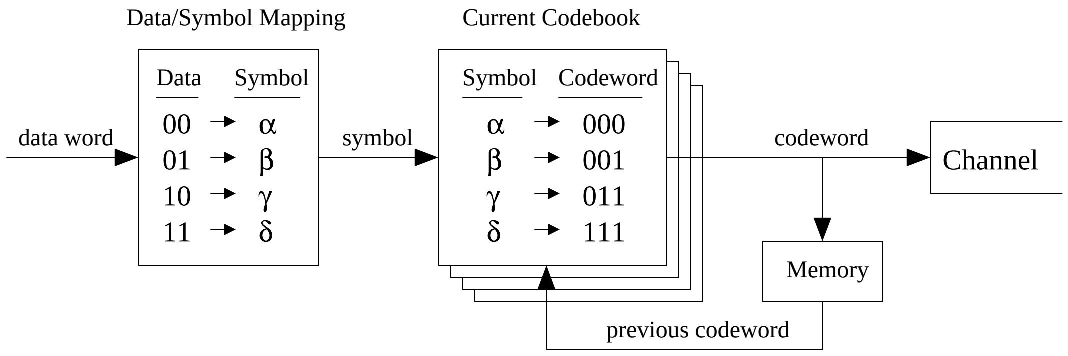


Figure 3.5 Illustration of Coding Terminology.

analysis. As long as we are able to express any symbol at any time, we know we have a valid code. The mapping between symbols and actual data words occurs in the implementation phase, and may determine the efficiency of the implementation but not the overall properties of the code.

We say that the values placed on the channel are called *codewords*. The mapping between symbols and codewords is called a *codebook*. Although we can abstract the data words with symbols, we must consider the specific values of the codewords, because the sequence of codewords placed on the channel must obey the channel constraint. It is possible that the codebook, and thus the active set of codewords, changes with time. If this is the case, the code is said to have *memory*. If the codebook is fixed, then the code is *memoryless*. Figure 3.5 illustrates this terminology.

Specific to self-shielding coding is a restriction on which codewords are allowed to follow which, as dictated by the Fundamental Rule. We say that a codeword is *connected* to another codeword if it is valid to transition from one to the other. That is, if two codewords are connected, then one may placed on the channel immediately following the other. It will aid our analysis to import some terminology from graph theory. We can form a graph with the codewords as vertices and the connections as edges. The graph is undirected because the connection relation is symmetric. We can then say that the *neighbor set* of a codeword is the set of codewords that it is connected to, and the *degree* of a codeword is the size of its neighbor set. Note that every codeword is connected to itself, because a channel that does not transition at

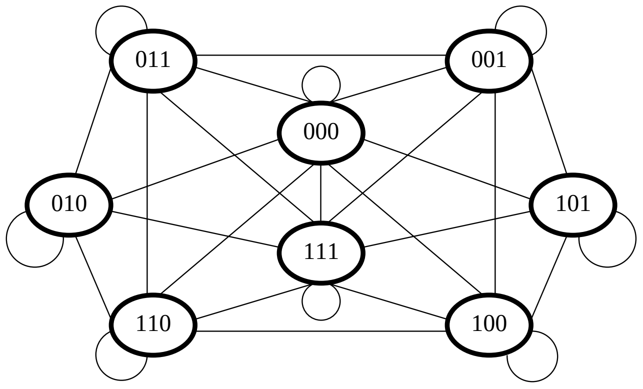


Figure 3.6 Connection Graph for $n = 3$.

all certainly abides by the Fundamental Rule. Thus, every codeword has itself as a neighbor. The connection graph for $n = 3$ is shown in Figure 3.6.

Chapter 4

Unpruned Code With Memory

(Counting the Neighbors)

4.1 Introduction

Now that the Fundamental Rule has been defined, we must analyze its implications in an attempt to determine the properties of self-shielding codes. In this chapter, the rudiments of self-shielding coding theory will be developed as we derive the performance of a simple code. The theory developed in this chapter will be concerned only with the size of codewords' neighbor sets, not with their contents. The code type that this theory applies to is the unpruned code with memory.

Although the cost metric as stated in Section 3.3 is the channel width n given a particular symbol set size 2^b, it is more convenient for our analysis to examine it from the opposite viewpoint. That is, given a channel of n wires, we determine the minimum number of symbols that can be expressed at any time. The base-2 logarithm of this number is the maximum data width b, and we use this as our performance metric.

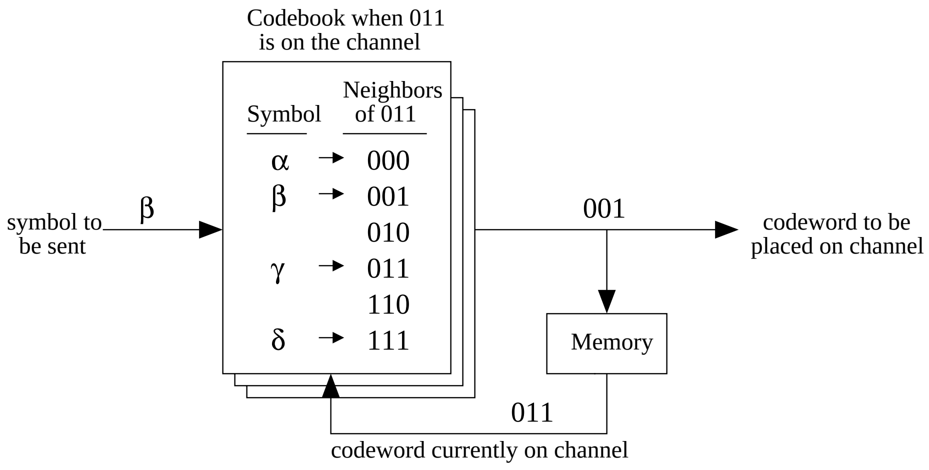

Codebook when 011 is on the channel
Symbol
Neighbors of 011
α → 000
β → 001
010
γ → 011
110
δ → 111
symbol to be sent
β
001
codeword to be placed on channel
Memory
011
codeword currently on channel

Figure 4.1 Example of Codebooks and Memory.

In order for a symbol to be expressible, it must be mapped to a codeword in the codebook. However, there is a restriction on which codewords can be in the codebook at a given time. Because, by definition, only neighbors of the codeword currently on that channel are allowed to follow it, these neighbors are the only codewords that can be in the current codebook. If a symbol is mapped to one of these neighbors and that neighbor is then placed on the channel, that symbol is expressed and the data word that the symbol represents is effectively transmitted. Because the neighbor set is different for each codeword, it is possible that the codebook will change every time a new codeword is placed on the channel. That is, the symbol that is expressed by a codeword on the channel may depend on what the previous codeword was. This simply means that the encoding has memory. An example is illustrated in Figure 4.1.

Thus, the maximum number of symbols that can be expressed at a particular time is equal to the degree of the codeword on the channel. If we want to express more symbols than this, there simply aren't enough neighbors to go around. For the unpruned code with memory, we will make no assumptions about the contents of the codebooks. Thus, we must be prepared for any of the 2^n possible codewords to be on the channel at any time. This means that we must choose b such that the degree of every possible codeword is at least 2^b. Therefore, to determine the maximum performance of this code, we must find the codeword with the smallest degree, since it will be the limiting factor in determining b.

However, calculating the degree of a codeword is not a trivial matter. Imagine if there were no Fundamental Rule. We could just say that every codeword has degree 2^n, because any codeword could transition to any other. But this trivial calculation is based on an implicit assumption—that the bits can transition independently of one another. Each of the n bits can go to either of 2 states, so we have a total of 2^n possible transitions. However, with the Fundamental Rule in place, we suddenly have a dependency between adjacent bit transitions. For example, if we choose to raise a particular 0 bit in the codeword, then any adjacent 1 bits must stay, although adjacent 0 bits may either rise or stay. The decision for them affects their adjacent bits, and so on across the codeword. This may seem like a complicated situation, but with the appropriate mathematical formalization, we can get it all under control. In the following theorems, we will derive the method for calculating the degree of any codeword, and we will use this to find the limiting codeword and its degree.

4.2 Degree of Class 1 Codewords

Definition: A *class 1 codeword* is a codeword with alternating 0 and 1 bits. For example, 01010 and 10101 are 5-bit class 1 codewords.

Definition: d_n is the degree of a class 1 codeword that is n bits wide.

Theorem 4.1: *d_n are Fibonacci numbers. Specifically,*

$$d_n = F_{n+2} \tag{4.1}$$

where F_n is the classical Fibonacci sequence $\{1, 1, 2, 3, 5, 8, 13, \ldots\}$.

Proof: Consider, without loss of generality, a class 1 codeword of n bits that begins with a 0 bit. In a valid codeword transition, this first bit can either stay or rise, but this choice affects the allowable transitions of the second bit. If the first bit stays, the second bit is free to stay or fall with no restrictions. The second through nth bits form a class 1 codeword of width $n-1$, and thus can realize d_{n-1} transitions.

$$\boxed{\begin{array}{ccccc} \begin{bmatrix}\text{neighbors of}\\ 010101\end{bmatrix} & = & 0\begin{bmatrix}\text{neighbors of}\\ 10101\end{bmatrix} & + & 11\begin{bmatrix}\text{neighbors of}\\ 0101\end{bmatrix} \\ d_6 & = & d_5 & + & d_4 \end{array}}$$

Figure 4.2 Example of Recursion of d_n.

However, if the first bit rises, then the second bit is forced to stay, because it would violate the Fundamental Rule if it were allowed to fall. The third bit is then free to stay or rise without restrictions. The third through nth bits form a class 1 codeword of width $n-2$, and thus can realize d_{n-2} transitions. The total number of transitions d_n is the sum of the two cases:

$$d_n = d_{n-1} + d_{n-2} \tag{4.2}$$

This is illustrated in Figure 4.2. This is the same recurrence relation obeyed by the Fibonacci sequence. But in order to show that d_n are in fact Fibonacci numbers, we need to establish initial conditions. Two initial conditions are needed, because (4.2) is a second-order equation. A 1-bit class 1 codeword is “0”. It can transition to two codewords: “0” and “1”. A 2-bit class 1 codeword is “01”. It can transition to “00”, “01”, or “11”, but *not* “10”. We see that $d_1 = 2$ and $d_2 = 3$. These are in fact Fibonacci numbers, F_3 and F_4 respectively. Therefore, $d_n = F_{n+2}$. ❑

Corollary:

$$d_n = \frac{1}{\phi + 1/\phi}\left(\phi^{n+2} - (-\phi)^{-(n+2)}\right), \qquad \phi = \frac{1+\sqrt{5}}{2} \tag{4.3}$$

$$d_n = \begin{cases} \frac{2}{\phi+1/\phi}\cosh\left((n+2)\ln(\phi)\right), & \textit{odd } n \\ \frac{2}{\phi+1/\phi}\sinh\left((n+2)\ln(\phi)\right), & \textit{even } n \end{cases} \tag{4.4}$$

Proof: If we rewrite (4.2) as

$$d_{n+2} - d_{n+1} - d_n = 0 \tag{4.5}$$

we see that it is a second-order homogeneous difference equation with constant coefficients. We solve by making the substitution $d_n = a^n$ in (4.5), and combining the

n	d_n	$\log_2(d_n)$
1	2	1.00
2	3	1.58
3	5	2.32
4	8	3.00
5	13	3.70
6	21	4.39
7	34	5.09
8	55	5.78
9	89	6.48

Table 4.1 Degrees of Some Class 1 Codewords.

two solutions for a using linear superposition. This results in a solution of the form

$$d_n = c_1 a_1^n + c_2 a_2^n\,. \qquad \textbf{(4.6)}$$

We find that

$$a_1 = \frac{1+\sqrt{5}}{2}, \qquad a_2 = \frac{1-\sqrt{5}}{2}\,. \qquad \textbf{(4.7)}$$

To find c_1 and c_2, we need two initial conditions. Using $d_1 = 2$ and $d_2 = 3$ as above, we find

$$c_1 = \frac{a_1^2}{\sqrt{5}}, \qquad c_2 = -\frac{a_2^2}{\sqrt{5}}\,. \qquad \textbf{(4.8)}$$

The numbers in (4.7) are in fact rather famous. a_1 is called the *golden ratio*, and is commonly written as ϕ. a_2 is the negative reciprocal of ϕ as well as its conjugate (as well as $1-\phi$), and is sometimes referred to as the *silver ratio* and written as ϕ_C.* Applying (4.7) and (4.8) to (4.6), we can write the result as either (4.3) or (4.4) ❑

Some values of d_n are shown in Table 4.1.

* ϕ has a tendency to turn up in all sorts of places, from the growth patterns of plants to the dimensions of the Greek Parthenon to the proportions in the paintings of Leonardo Da Vinci. The famous opening bars of Beethoven's Fifth are repeated at exactly the $1/\phi$ point in the symphony. There is much mathematical folklore associated with ϕ, and much of it is intertwined with Fibonacci folklore because of their close association. There are also those who claim that ϕ has been over-romanticized, and its ubiquity is all a big coincidence.

4.3 Degree of General Codewords

Definition: An *independent boundary* in a codeword occurs between two adjacent bits of the same value. A *dependent boundary* occurs between two adjacent bits of different values.

For example, the codeword 0011 has three boundaries, and they are independent, dependent, and independent respectively.

Definition: A *section* of a codeword is one of the pieces that would result if the codeword were to be split at its independent boundaries.

For example, the codeword 10110100 has three sections: 101, 1010, and 0. Notice that each section, if isolated, would be considered a class 1 codeword.

Definition: The *class* of a codeword is equal to the number of sections, or the number of independent boundaries plus one.

Definition: $d_{\{n_1,n_2,\ldots,n_c\}}$, where c is the class, denotes the degree of a codeword with sections of width n_1, n_2, etc.

In general, it is often handy to describe a codeword with an ordered set of c elements instead of explicitly giving its bit pattern. That is, the codeword 10110100 may be written as $\{3,4,1\}$. Notice that $\{3,4,1\}$ refers to the bitwise inverse as well, the codeword 01001011. This ambiguity is typically acceptable because codewords and their bitwise inverses tend to have exactly the same properties. If we should want to distinguish the two codewords, we may refer to the codeword's *polarity*. Other notational conventions include using $\{n_1,n_2,\ldots\}^k$ to refer to the given sequence of sections repeated k times, and $\{n_1,n_2,\ldots\}\{m_1,m_2,\ldots\}$ to refer to the concatenation of the codewords: $\{n_1,n_2,\ldots,m_1,m_2,\ldots\}$.

Theorem 4.2: *The set of transitions that a section is allowed to make is not restricted or affected by the transitions chosen for the other sections in the codeword.*

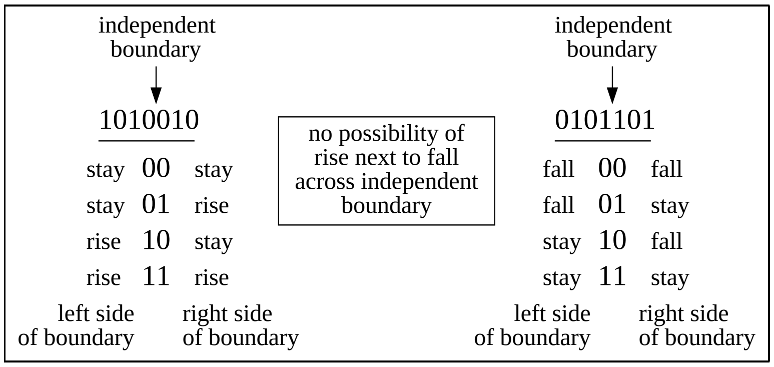


Figure 4.3 Independent Boundaries.

Proof: Two adjacent sections, by definition, are separated by an independent boundary, meaning that there are two adjacent bits of the same value. No transition can be chosen for these two bits that violates the Fundamental Rule. If they are both 0 bits, then they can both stay, both rise, or one can rise and the other stay. If they are both 1 bits, then they can both stay, both fall, or one can fall and the other stay. But in no case can there be a rising transition next to a falling one. (See Figure 4.3.) Since the Fundamental Rule is the only restriction on bit transitions and only applies to adjacent bits, the bit transitions on one side of an independent boundary cannot restrict the transitions on the other side. Thus, the sets of transitions for each section are independent of one another. ❑

Remark: This theorem demonstrates the origin of the term *independent boundary*. It is impossible to violate the Fundamental Rule across this boundary, so transitions on one side are independent from those on the other.

Theorem 4.3: *The degree of any codeword is equal to*

$$d_{\{n_1,n_2,\ldots,n_c\}} = \prod_{i=1}^{c} d_{n_i} \tag{4.9}$$

where c is the codeword class and n_i is the number of bits in the ith section.

Proof: By definition, each section, if isolated, would be a class 1 codeword, and so by Theorem 4.1, it could make d_{n_i} valid transitions. By Theorem 4.2, each section

$$\begin{array}{ccccccccccc} 10100100 & \longrightarrow & 1010 & \mid & 010 & \mid & 0 & & & & \\ d_{\{4,3,1\}} & = & d_4 & \times & d_3 & \times & d_1 & = & 8 \times 5 \times 2 & = & 80 \end{array}$$

Figure 4.4 Example of Calculation of d_W.

$$x = 5, \quad y = 3$$

$$\text{class 1: } \overbrace{01010101}^{d_{x+y}} \qquad \text{class 2: } \overbrace{01010}^{d_x} \overset{\times}{\vphantom{0}} \overbrace{010}^{d_y}$$

Figure 4.5 Illustration of d_{x+y} versus $d_x\, d_y$.

can transition independently of one another, so they are effectively isolated. Any valid transition in one section can be matched with any valid set of transitions in the other sections. Thus, the total number of valid transitions in the complete codeword is the product of the degrees of each section. ❑

The above formula is illustrated in Figure 4.4. We now know how to calculate the degree of any codeword. It remains to find the codeword with the lowest degree, because it is responsible for determining the performance of our code. In the following theorem, we will prove that the limiting codeword which we seek is none other than the class 1 codeword.

Theorem 4.4: *For a given codeword width n, the degree of any codeword in class $c > 1$ is greater than d_n.*

Proof: We begin by proving this for $c = 2$. Using Theorem 4.3, the proposition can be stated mathematically as:

$$d_{x+y} < d_x d_y \qquad \text{for } x > 0,\, y > 0 \tag{4.10}$$

where $x + y = n$. This is illustrated in Figure 4.5. Restating (4.10) in terms of (4.3), we get

$$\left(\phi + \frac{1}{\phi}\right)\left(\phi^{x+y+2} - (-\phi)^{-x-y-2}\right) < \left(\phi^{x+2} - (-\phi)^{-x-2}\right)\left(\phi^{y+2} - (-\phi)^{-y-2}\right). \tag{4.11}$$

Multiplying this out and shuffling terms gives us

$$\phi^{x+y+4} - \phi^{x+y+3} - \phi^{x+y+1} + (-\phi)^{-x-y-4} - (-\phi)^{-x-y-3} - (-\phi)^{-x-y-1} > (-1)^y \phi^{x-y} + (-1)^x \phi^{y-x} \,. \quad \textbf{(4.12)}$$

Because ϕ^k is a solution to (4.5), we know that

$$\phi^k = \phi^{k-1} + \phi^{k-2} \,. \quad \textbf{(4.13)}$$

Transposing and resubstituting (4.13) a few times leads to the identity

$$\phi^k = \phi^{k+4} - \phi^{k+3} - \phi^{k+1} \,. \quad \textbf{(4.14)}$$

Because $(-\phi)^{-k}$ solves (4.5) as well, we can similarly derive that

$$(-\phi)^{-k} = (-\phi)^{-k-4} - (-\phi)^{-k-3} - (-\phi)^{-k-1} \,. \quad \textbf{(4.15)}$$

Using these identities with (4.12) leads to

$$\phi^{x+y} + (-\phi)^{-x-y} > (-1)^y \phi^{x-y} + (-1)^x \phi^{y-x} \,. \quad \textbf{(4.16)}$$

Moving all terms to the left side and factoring gives us

$$\big(\phi^x - (-\phi)^{-x}\big)\big(\phi^y - (-\phi)^{-y}\big) > 0 \,. \quad \textbf{(4.17)}$$

(4.17) is true if $\big(\phi^k - (-\phi)^{-k}\big)$ is positive for both $k = x$ and $k = y$.* If k is odd, this expression is $\big(\phi^k + \phi^{-k}\big)$, which is positive for all k. If k is even, the expression is $\big(\phi^k - \phi^{-k}\big)$, which is true for all $k > 0$. Since both x and y are positive, this is always the case. Thus $d_{x+y} < d_x d_y$.

For classes $c > 2$, this inequality can be applied iteratively:

$$d_{x+y+z} < d_{x+y} d_z < d_x d_y d_z \quad \textbf{(4.18)}$$

and so on. Thus the degree of any codeword in class $c > 1$ is greater than the degree of a class 1 codeword of the same width. ❑

* Or negative for both, of course. But that's not the case here.

4.4 Class Properties

Armed with the previous theorems, we can go ahead and prove a number of interesting properties about codeword classes. We will find the minimum degree, maximum degree, and size of each class for a given n.

Theorem 4.5: *For a given codeword width n and class c, the set of codewords with the smallest degree in the class is the set of codewords with the most 2-bit sections. (Class Minimum)*

Proof: For $c \leq \frac{n}{2}$, the codewords with the most 2-bit sections are $\{2, 2, 2, \ldots, n-2c+2\}$ and its permutations. We can also write this as $\{2\}^{c-1}\{n-2c+2\}$. For $c > \frac{n}{2}$, the codewords have $(n-c)$ 2-bit sections and $(2c-n)$ 1-bit sections. That is, $\{2\}^{n-c}\{1\}^{2c-n}$ and its permutations. In both cases, it is possible to generate all other codewords with the same n and c by starting with one of these codewords and repeatedly replacing a 2-bit section and an m-bit section with a $(2+k)$-bit section and a $(m-k)$-bit section. That is, any other codeword can be formed by taking k bits from some section and moving them to a 2-bit section, some number of times. Each time this shift is made, the degree of the codeword changes by a factor of

$$\frac{d_{2+k}d_{m-k}}{d_2 d_m} \tag{4.19}$$

We will prove that for all numbers positive and $m-k \neq 2$, this factor is greater than 1. That is, the degree increases when the shift is made, unless the shift amounts to simply swapping the sections. We can write this as the inequality

$$d_{2+k}d_{m-k} > d_2 d_m \tag{4.20}$$

and then restate it using (4.3):

$$\left(\phi^{k+4} - (-\phi)^{-k-4}\right)\left(\phi^{m-k-2} - (-\phi)^{-m+k+2}\right) > \left(\phi^4 + \phi^{-4}\right)\left(\phi^{m+2} - (-\phi)^{-m+2}\right). \tag{4.21}$$

Multiplying, canceling common terms, and playing with powers of -1 gives us

$$\phi^{m-k-2}(-\phi)^{-k} + (-\phi)^{-m+k+2}\phi^k < \phi^{m-2} + (-\phi)^{-m+2}. \tag{4.22}$$

Moving terms to the left side and factoring results in

$$\left(\phi^{k} - (-\phi)^{-k}\right)\left(\phi^{m-k-2} - (-\phi)^{-m-k+2}\right) > 0\,. \tag{4.23}$$

This inequality is in a similar form as (4.17), and the analysis of (4.17) is applicable here. We find, for the cases that we are interested in, the inequality is true for $k > 0$ and either $m-k > 2$ or $(m-k)$ odd. The only time that a positive $m-k$ can violate both clauses of the second condition is when it is 2. Thus, for all numbers positive and $m-k \neq 2$, the inequality is true. Thus, the bit shift always results in the degree increasing. Because any other codeword in the class can be formed by starting with a codeword with the most 2-bit sections and applying enough shifts, the codeword with the most 2-bit sections must have the lowest degree in the class. ❑

Corollary: *For a given codeword width n and class c, the smallest degree in the class is:*

$$\begin{aligned} 3^{c-1}d_{n-2c+2} &\qquad \text{for } 1 \le c \le \tfrac{n}{2} \\ 2^{2c-n}3^{n-c} = 2^{n}\left(\tfrac{3}{4}\right)^{n-c} &\qquad \text{for } \tfrac{n}{2} \le c \le n \end{aligned} \tag{4.24}$$

Proof: Applying Theorem 4.3 to Theorem 4.5 gives the desired result. ❑

Theorem 4.6: *For a given codeword width n and class c, the set of codewords with the largest degree in the class is the set of codewords with the most 1-bit sections. (Class Maximum)*

Proof: The codewords with the most 1-bit sections are the permutations of $\{1, 1, 1, \ldots, n-c+1\}$, or $\{1\}^{c-1}\{n-c+1\}$. With a method similar to that described in Theorem 4.5, all other codewords with the same n and c and be generated by repeatedly replacing a 1-bit section and an m-bit section with a $(1+k)$-bit section and an $(m-k)$-bit section. When this replacement is made, the degree changes by a factor of

$$\frac{d_{1+k}d_{m-k}}{d_1 d_m}\,. \tag{4.25}$$

We will prove that for all numbers positive and $m - k \neq 1$, this factor is less than 1. That is, the degree decreases when this change is made, unless it amounts to simply swapping sections. We write this as the inequality

$$d_{1+k} d_{m-k} < d_1 d_m \tag{4.26}$$

and restate it using (4.3):

$$\left(\phi^{k+3} - (-\phi)^{-k-3}\right)\left(\phi^{m-k-2} - (-\phi)^{-m+k+2}\right) < \left(\phi^4 + \phi^{-3}\right)\left(\phi^{m+2} - (-\phi)^{-m+2}\right). \tag{4.27}$$

Multiplying and canceling common terms gives us

$$\phi^{m-k-1}(-\phi)^{-k} + (-\phi)^{-m+k+1}\phi^k < \phi^{m-1} + (-\phi)^{-m+1}. \tag{4.28}$$

Moving terms to the left side and factoring results in

$$\left(\phi^k - (-\phi)^{-k}\right)\left(\phi^{m-k-1} - (-\phi)^{-m-k+1}\right) > 0. \tag{4.29}$$

Remarkably, (4.29) is almost identical to (4.23), even though the inequality started out in the other direction. (4.29) is true for $k > 0$ and $m - k > 1$. Thus, the bit shift always results in the degree decreasing, as long as $m - k \neq 1$. Because any other codeword in the class can be formed by starting with a codeword with the most 1-bit sections and applying enough shifts, the codeword with the most 1-bit sections must have the highest degree in the class. ❑

Corollary: *For a given codeword width n and class c, the largest degree in the class is:*

$$2^{c-1} d_{n-c+1} \tag{4.30}$$

Proof: Applying Theorem 4.3 to Theorem 4.6 gives the desired result. ❑

Remark: We can use (4.24) and (4.30) to calculate the range of degrees in each class for a given n, as shown in Table 4.2. Doing so reveals a somewhat surprising result. For $n \geq 8$, there are overlaps in the ranges. There is some c for which the

c: n	1	2	3	4	5	6	7	8	9	10	11
1	2										
2	3	4									
3	5	6	8								
4	8	9-10	12	16							
5	13	15-16	18-20	24	32						
6	21	24-26	27-32	36-40	48	64					
7	34	39-42	45-52	54-64	72-80	96	128				
8	55	63-68	72-84	81-104	108-128	144-160	192	256			
9	89	102-110	117-136	135-168	162-208	216-256	288-320	384	512		
10	144	165-178	189-220	216-272	243-336	324-416	432-512	576-640	768	1024	
11	233	267-288	306-356	351-440	405-544	486-672	648-832	864-1024	1152-1280	1536	2048

Table 4.2 Range of Degrees in Each Class.

maximum degree in class c is greater than the minimum degree in class $c+1$. Or in other words, there are codewords which have a higher degree than some codewords in a higher class. It is important not to assume that degrees are strictly ordered by class, because for $n \geq 8$, they aren't.[*]

Theorem 4.7: *For a given codeword width n, the number of codewords in a class c is:*

$$\frac{2(n-1)!}{(c-1)!(n-c)!} \tag{4.31}$$

(Class Size)

Proof: In a codeword of width n, there are $n-1$ bit boundaries. If the class is c, then $c-1$ of these boundaries must be independent. The number of ways of distributing $c-1$ objects over $n-1$ positions is

$$\binom{n-1}{c-1} \tag{4.32}$$

For each distribution of independent boundaries, there are two codewords, one of each polarity. Once the first bit of the codeword is chosen, the independent/dependent

[*] Because the strict ordering of degrees by class is such a natural assumption to make, the author spent a regrettable amount of time in the early stages of this research trying to prove it. The lesson here is to be as thorough as possible with experimentation before attempting a proof. The other lesson, perhaps, is that proving something that isn't actually true can be really quite difficult.

boundary distribution determines the rest of the bits, but the first bit can be chosen to be either 0 or 1. Thus, the total number of codewords in the class is

$$2\binom{n-1}{c-1} \tag{4.33}$$

which is equivalent to (4.31). ❑

Remark: Notice that distributing $c-1$ independent boundaries is equivalent to distributing $n-c$ dependent boundaries. Thus, class sizes show a symmetry, with class c having the same number of members as class $n-c+1$.

4.5 Results

With the theory developed in Sections 4.2 and 4.3, and Theorem 4.4 in particular, we are able to state the maximum performance of the unpruned code with memory. Note that we have not designed this code, nor even defined a set of codewords. The only specifications were that it must obey the Fundamental Rule, and it must allow at least 2^b symbols to be expressed when any possible n-bit value is on the channel. From these two specifications alone, we can determine that the maximum number of data bits that can be transmitted over a crosstalk-immune n-wire channel is:

$$b = \log_2(d_n) \tag{4.34}$$

It can be seen in (4.3) that d_n is asymptotically proportional to ϕ^n. Thus, adding a wire to the channel increases the maximum data width by about $\log_2(\phi)$, or 0.69 bits. This can be compared to simple shielding, which has an asymptotic performance of only 0.5 bits per channel wire.

Figure 4.6 plots the performance of this code, and Table 4.3 lists the channel widths required to transmit data of various widths. We see that a 32-bit bus could be implemented with 46 wires. This compares very favorably to a simple shielding scheme which would require 63 wires. If we conceptually consider $n-b$, channel width minus data bits, to be the number of "extra wires" required to eliminate crosstalk delay, shielding uses 31 extra wires, whereas coding uses only 14.

But, we can do even better than this.

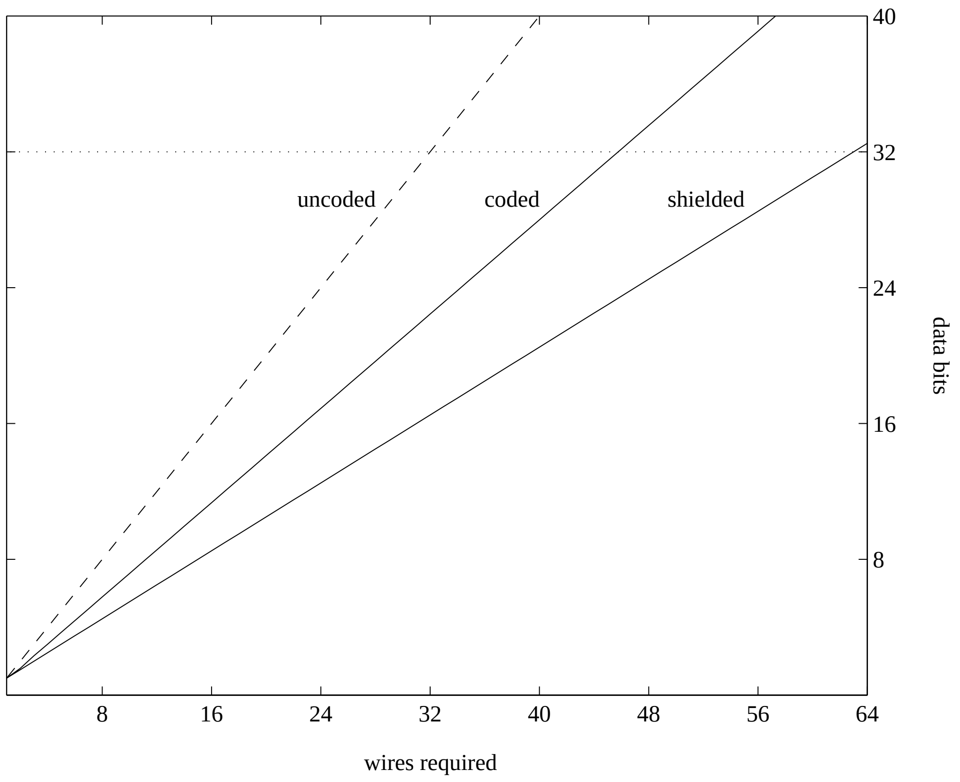


Figure 4.6 Performance of Unpruned Code.

bits	wires required		bits	wires required	
	coded	*shielded*		*coded*	*shielded*
1	1	1	17	25	33
2	3	3	18	26	35
3	4	5	19	28	37
4	6	7	20	29	39
5	7	9	21	30	41
6	9	11	22	32	43
7	10	13	23	33	45
8	12	15	24	35	47
9	13	17	25	36	49
10	15	19	26	38	51
11	16	21	27	39	53
12	17	23	28	41	55
13	19	25	29	42	57
14	20	27	30	43	59
15	22	29	31	45	61
16	23	31	32	46	63

Table 4.3 Performance of Unpruned Code.

Chapter 5

Pruned Code With Memory

(Inspecting the Neighbors)

5.1 Introduction

In the previous chapter, we found the maximum performance of an unpruned code with memory. That code was called "unpruned" because no assumptions were made about which codewords were allowed on the channel. We had to be prepared to transition from any possible n-bit value, and thus our performance was limited by the codeword with the smallest degree. This codeword turned out to be the class 1 codeword.

However, if we ensure that our code never transitions *to* a class 1 codeword, then we know that there will never be a class 1 codeword on the channel. Thus, we won't have to worry about transitioning *from* a class 1 codeword, and the code performance will no longer be limited by d_n. In other words, we simply throw the class 1 codewords out of the codebooks.

Although this raises the limiting degree, and hence the performance of our code, it also has the effect of decreasing the degrees of some of the other codewords. Specifically, the degrees of all codewords in the class 1 codewords' neighbor sets will decrease, because these codewords are no longer allowed to transition to a class 1 code. But this won't affect the overall performance of the code, unless one of these codewords is the new limiting factor.

There is no reason to stop with class 1 codes, though. We can continue to find and remove the codewords with the limiting degrees, continuing until the limiting degree is as high as possible. This is the basis of the codeword pruning algorithms, which we will study in this chapter. Inherent in understanding the pruning process is the need to take a look inside the codewords' neighbor sets, and see which codewords they are connected to. Unlike the previous chapter, which dealt only with the size of the neighbor set, the theory developed in this chapter will examine the neighbors themselves.

5.2 Optimal Pruning

Consider the following algorithm for pruning the codeword set.

Algorithm 5.1: *Optimal Codeword Pruning Algorithm*

```
While there are valid codewords left:
    Find the set of valid codewords with the lowest degree.
    For each codeword $W$ in the set:
        Remove $W$ from the set of valid codewords
        Decrement the degree of each codeword in $W$'s neighbor set.
```

As this algorithm progresses, the limiting degree will increase, hit a maximum, and then decrease as the codebook gets depleted. We choose, of course, the set of codewords that was active when the limiting degree was at its maximum. This algorithm is optimal, in that it is guaranteed to come up with the best possible set of codewords. This is proven in the next theorem.

Theorem 5.1: *There is no set of codewords that has a higher minimum degree than that produced by the Optimal Codeword Pruning Algorithm.*

Proof: At each point during the algorithm, the set of codewords with the lowest degree is removed. Let us call this set S_W. Consider what would happen if a codeword W not in this set were to be removed instead. The degrees of W's neighbors would decrease, and the limiting degree would decrease if one of these neighbors were in S_W. But the limiting degree could never increase, because it would still be limited by the codewords in S_W. Pruning any codeword results in other codewords' degrees decreasing, and until all codewords in S_W are removed, the limiting degree can only decrease or stay constant. Thus, removing W can have no benefits, either future or present, until S_W is removed, which is the procedure followed by the algorithm. ❑

To visualize the pruning process, we can make a plot of the limiting degree versus codewords pruned as the algorithm runs. Such a plot is called a *pruning curve*. Figure 5.1 shows the pruning curves for a few values of n. A more extensive collection of pruning curves can be found in Appendix A. Examining the curves, we observe the expected shape. The limiting degree increases, hits a maximum roughly in the middle, and then drops to zero. Table 5.1 shows what this maximum degree is for some values of n.

As Theorem 5.1 states, the above algorithm is optimal, and can be used to find the fundamental performance limit for any self-shielding code. Unfortunately, pruning as described by the optimal algorithm is extremely computationally intensive, both in terms of running time and memory use. For example, if we implement the algorithm directly, without using any theoretical tricks, the program requires 2^n bits simply to keep track of which codewords have been pruned. We can divide memory use in half by only keeping track of codewords where the first bit is 0, since every codeword has the same properties as its bitwise inverse, but that still would require 256 megabytes of memory for $n = 32$. But, such an implementation would be unusably slow because the program would have to recalculate the degree of every codeword on every pass. Any practical implementation would keep track of the degree of each codeword, and that would require 8 gigabytes of memory for $n = 32$.

Even more prohibitive than memory use is running time. The algorithm has a

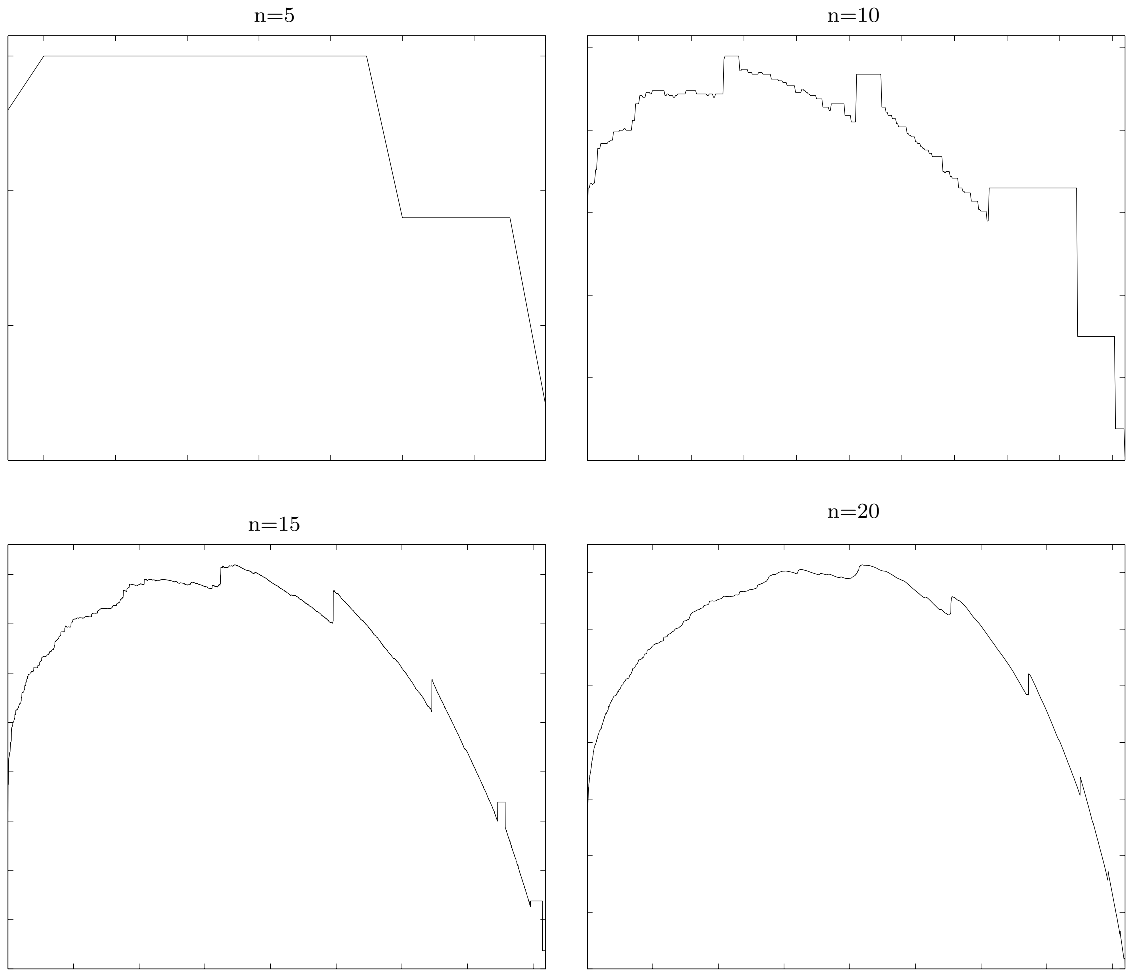


Figure 5.1 Pruning Curves.

computational complexity of $o(2^{3n})$, because there are three nested traversals of the codeword set, which goes as 2^n. With moderately optimized code running on a DEC AlphaServer 8400 5/625, the algorithm took several days to generate the pruning curve for $n = 23$. That's where the author stopped.

But from a theoretical standpoint, the computational complexity of the algorithm isn't the main concern—it's that an algorithm is required in the first place. Optimal pruning, as described, is a purely experimental procedure. It is very difficult to come up with a mathematical description for the behavior of the algorithm, and it is not amenable to the rigorous analysis that we seek. We would prefer to work with

n	max degree	$\log_2$(max degree)
1	2	1.00
2	3	1.58
3	5	2.32
4	9	3.17
5	15	3.91
6	27	4.75
7	46	5.52
8	81	6.34
9	141	7.14
10	245	7.94
11	431	8.75
12	745	9.54
13	1322	10.37
14	2308	11.17
15	4099	12.00
16	7178	12.81
17	12859	13.65
18	22622	14.47
19	40255	15.30
20	71375	16.12
21	126866	16.95
22	225668	17.78
23	400467	18.61

Table 5.1 Maximum Performance After Optimal Pruning.

formulas and equations than experimental data. So, we do what every good engineer does in such a situation. We make an approximation.

5.3 Class Pruning

Figure 5.2 shows some pruning curves, but with some additional data plotted. We can now see not just the minimum degree at each point in the algorithm, but also the class of the codewords being pruned. This appears as a descending staircase, with class 1 at the top and class n at the bottom. (Again, a more complete set of pruning curves can be found in Appendix A.) We see that for small n, the codewords are pruned in class order. That is, all codewords in class c are entirely pruned before any in class $c + 1$ are touched. For moderate n, we see a bit of aberration, but the pruning is still very close to class order. For large n, it appears that most of the time, there are two classes that are undergoing the bulk of the pruning, which smears out

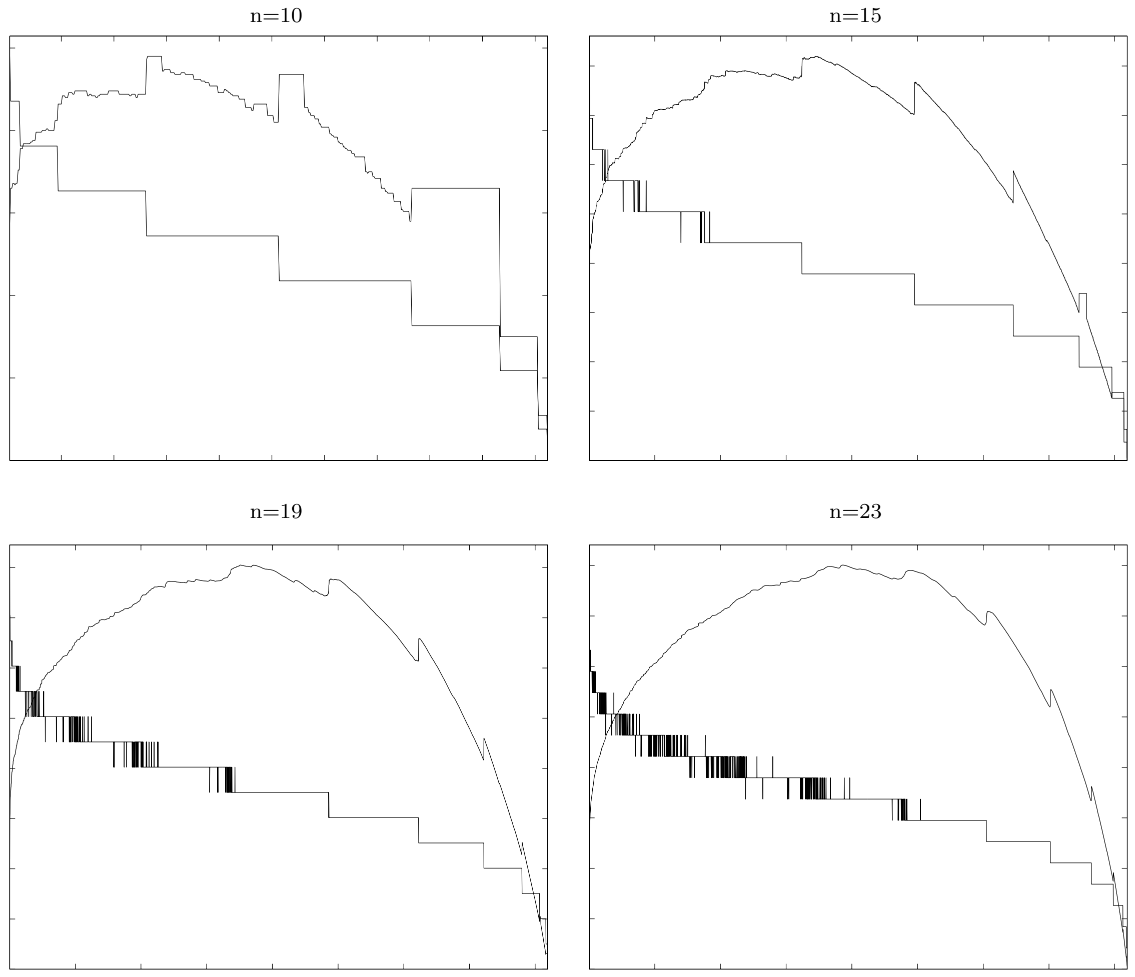


Figure 5.2 Pruning Curves With Class Designation Line.

the class designation line. However, it is clear that the concept of codeword class is still playing a strong role in determining the rough pruning order, even if it is not as strict as we might hope.*

Furthermore, we see that the pruning curves for small and moderate n exhibit very distinct spikes, one of which is typically at or near the global maximum. Comparing the location of the spikes to the class designation line reveals that they usually occur when a particular codeword class has been entirely pruned.

* One may be tempted to blame the breakdown of class order during pruning on the overlaps in the ranges of degrees for each class, as pointed out earlier in Table 4.2. Although this connection is not formally justified or even strictly correct, the author likes to believe in it anyway, at least from a qualitative point of view.

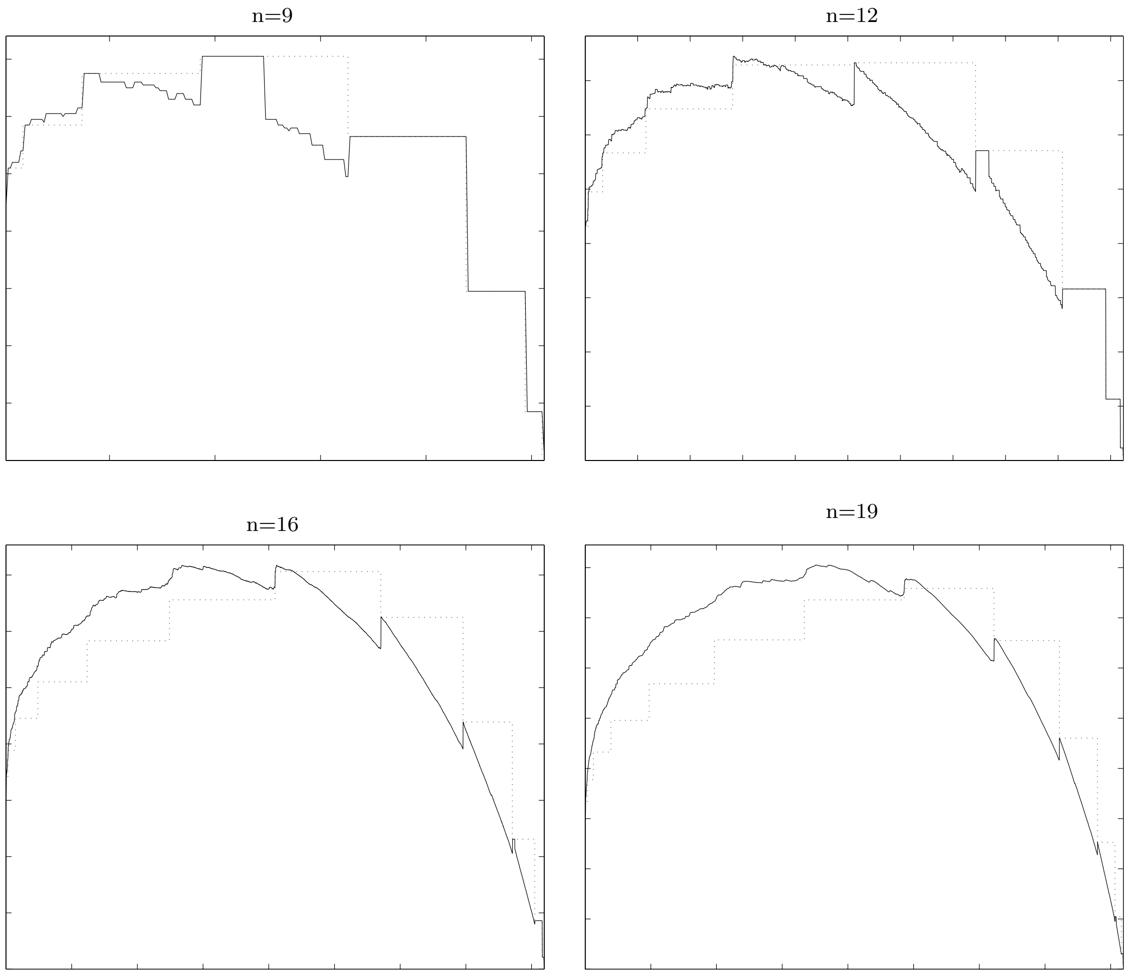


Figure 5.3 Optimal (solid) and Class (dotted) Pruning Curves.

These observations imply that we might be able to approximate the optimal pruning behavior by pruning entire classes of codewords at once. The revised pruning algorithm could be written as follows:

Algorithm 5.2: *Class Pruning Algorithm*

```
For each $c$ from 1 to $n$:
    Remove all codewords in class $c$ from the set of valid codewords
    Recalculate the degrees of the rest of the codewords
```

Again, the limiting degree will rise, hit a maximum, and fall. Figure 5.3 shows some curves generated by this algorithm superimposed on the corresponding optimal curves.

Data is given in the form: max degree (bits)

n	optimal	class pruned	error
1	2 (1.00)	2 (1.00)	
2	3 (1.58)	3 (1.58)	
3	5 (2.32)	5 (2.32)	
4	9 (3.17)	9 (3.17)	
5	15 (3.91)	15 (3.91)	
6	27 (4.75)	27 (4.75)	
7	46 (5.52)	46 (5.52)	
8	81 (6.34)	81 (6.34)	
9	141 (7.14)	141 (7.14)	
10	245 (7.94)	243 (7.92)	0.8% (0.02)
11	431 (8.75)	431 (8.75)	
12	745 (9.54)	733 (9.52)	1.6% (0.02)
13	1322 (10.37)	1314 (10.36)	0.6% (0.01)
14	2308 (11.17)	2281 (11.16)	1.2% (0.01)
15	4099 (12.00)	3997 (11.96)	2.5% (0.04)
16	7178 (12.81)	7061 (12.79)	1.6% (0.02)
17	12859 (13.65)	12135 (13.57)	5.6% (0.08)
18	22622 (14.47)	21763 (14.41)	3.8% (0.06)
19	40255 (15.30)	37932 (15.21)	5.8% (0.09)
20	71375 (16.12)	66832 (16.03)	6.4% (0.09)
21	126866 (16.95)	118228 (16.85)	6.8% (0.10)
22	225668 (17.78)	204600 (17.64)	9.3% (0.14)
23	400467 (18.61)	366689 (18.48)	8.4% (0.13)

Table 5.2 Comparison of Optimal and Class Pruning.

Numerical results are listed in Table 5.2. There is no guarantee of optimality with this algorithm. However, we see that for $n < 10$, the results match exactly with the optimal. For at least $n \leq 23$, which is all the optimal data that is available, the error is less than 10%, or 0.15 bits. Thus, the approximation is fairly good. Furthermore, because it is sub-optimal, the results are always achievable.

From an analysis point of view, class pruning is superior to optimal pruning in two ways. First, a class pruning curve consists of n data points, in contrast to the 2^n points generated by the optimal algorithm. This amount of data is, of course, exponentially easier to deal with. Secondly, with the right mathematical tools, we can directly calculate the limiting degree at each point with no need for experimentation. The n data points can be generated analytically without ever having to traverse the codeword set. The key to this formulation is the class distribution polynomial.

5.4 Derivation of the Class Distribution Polynomial

Definition: $D_{\{n_1,n_2,\ldots,n_c\}}(x)$ is called the *class distribution polynomial* for the codeword $\{n_1, n_2, \ldots, n_c\}$. The number of class k codewords in this codeword's neighbor set is equal to the coefficient of the x^{k-1} term of the polynomial. Equivalently, we can say that the number of neighbors with r independent boundaries is equal to the coefficient of the x^r term. We may also speak of distributions of general sets of codewords, such as subsets of a codeword's neighbor set, and this again refers to a polynomial as defined above.

As an example, the class distribution polynomial for the codeword 01001011, or $\{3, 4, 1\}$, is:

$$D_{\{3,4,1\}}(x) = 3x + 9x^2 + 17x^3 + 22x^4 + 18x^5 + 9x^6 + 2x^7 \qquad \textbf{(5.1)}$$

This means that the codeword $\{3, 4, 1\}$ can transition to zero class 1 codewords, three class 2 codewords, nine class 3 codewords, and so on.

Note that $D_W(1) = d_W$ for any codeword W. That is, the sum of the coefficients of the polynomial is the total degree of the codeword. Thus, in some sense, the theory in the previous chapter is simply a special case of the more general theory that will be developed here.* Note also that the highest term of $D_W(x)$ will always be $2x^{n-1}$, because there are two class n codewords, $00000\ldots$ and $11111\ldots$, and both of them can transition to every codeword.

We will now derive the method for computing $D_W(x)$.

Definition: Consider a class 1 codeword of width n. $P_n(x)$ is the class distribution of the set of neighbors of the codeword whose first and last bits are the same value as those of the codeword. Less formally, we say that $P_n(x)$ is the class distribution for a class 1 codeword *when the first and last bits do not transition.*

* As is commonly the situation, the more general theory will prove to be much harder to work with than the special case.

neighbors of 01010 described by:	$P_5(x) = 1 + 3x^2 + x^4$		$Q_5(x) = x + 2x^3$		$R_5(x) = x^2 + x^4$	
	00000	*(class 5)*	11000	*(class 4)*	11011	*(class 3)*
	00010	*(class 3)*	11010	*(class 2)*	11111	*(class 5)*
	01000	*(class 3)*	11110	*(class 4)*		
	01010	*(class 1)*	–or–			
	01110	*(class 3)*	00011	*(class 4)*		
			01011	*(class 2)*		
			01111	*(class 4)*		

Figure 5.4 Examples of $P_n(x)$, $Q_n(x)$, $R_n(x)$.

Definition: $Q_n(x)$ is the class distribution for a class 1 codeword of width n, when the first bit transitions and the last bit does not. (By symmetry, it is also the distribution when the last bit transitions and the first bit does not.) $R_n(x)$ is the class distribution when both the first and last bits transition. $Q_n(x)$ and $R_n(x)$ will only be used in this derivation, whereas $P_n(x)$ is a general concept which will be used throughout the theory.

Examples of the three distributions are shown in Figure 5.4.

Theorem 5.2:

$$Q_n(x) = xP_{n-1}(x) \tag{5.2}$$

Proof: Consider a class 1 codeword of width n. $Q_n(x)$ is the distribution of the neighbors which cause the first bit to transition and the last bit to stay. In a class 1 codeword, if the first bit transitions, the second bit must stay; otherwise, it would violate the Fundamental Rule. The second through nth bits now form a class 1 codeword with the first and last bits staying, and thus the distribution of this smaller codeword can be described with $P_{n-1}(x)$. However, in a class 1 codeword, if one bit transitions and an adjacent bit does not, an independent boundary is created between the two bits, because they are now the same value. Thus, there is an independent boundary between the first and second bits of every neighbor described by $Q_n(x)$, so we must increment the class numbers in $P_{n-1}(x)$ by one. Thus, we get $xP_{n-1}(x)$. This is illustrated in Figure 5.5. ❑

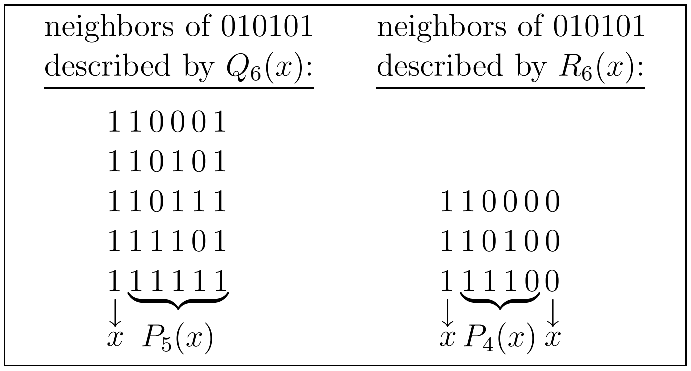

neighbors of 010101 described by $Q_6(x)$:
110001
110101
110111
111101
111111
x $P_5(x)$
neighbors of 010101 described by $R_6(x)$:
110000
110100
111100
x $P_4(x)$ x

Figure 5.5 Expressing $Q_n(x)$ and $R_n(x)$ in Terms of $P_n(x)$.

Theorem 5.3:

$$R_n(x) = x^2 P_{n-2}(x) \tag{5.3}$$

Proof: Consider a class 1 codeword of width n. $R_n(x)$ is the distribution of neighbors that cause the first bit and last bits to transition. Thus, the second and second-to-last bits must stay, and the second through second-to-last bits form a class 1 codeword that can be described by $P_{n-2}(x)$. Two independent boundaries are created between the first and second bits and between the second-to-last and last bits. Thus, the class numbers in $P_{n-2}(x)$ must be bumped up by two. This gives us $x^2 P_{n-2}(x)$. This is illustrated in 5.5. ❑

Theorem 5.4: $P_n(x)$ *is an even polynomial.*

Proof: Consider, without loss of generality, a class 1 codeword that begins with a 0 bit. If we name the first bit "bit 1", this codeword has 0's in odd-numbered bits and 1's in even numbered bits. Because $P_n(x)$ only considers neighbors who also begin with a 0 bit, we can generate all of the interesting neighbors by replacing some set of dependent boundaries with independent boundaries. (Recall that a class 1 codeword has only dependent boundaries.) However, each time we add an independent boundary, we swap the phase described above. That is, to the right of the first independent boundary, there are 0's in even-numbered positions and 1's in odd-numbered positions. To the right of the second independent boundary, we are back to the original

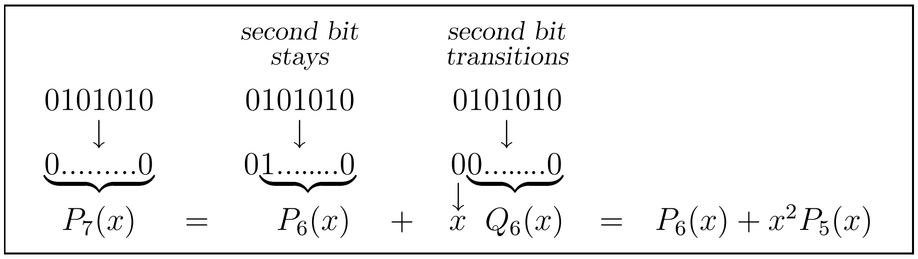


Figure 5.6 Recursion of $P_n(x)$.

phase, and so on. It is easy to see that if there is an odd number of independent boundaries, the last bit will be different than it was in the original codeword, and if there is an even number, it will be the same. However, the definition of $P_n(x)$ requires the last bit not to transition. Thus, we can only consider neighbors with an even number of independent boundaries. Since class was defined as the number of independent boundaries plus one, these are the neighbors in odd-numbered classes. But, for each term in a class distribution polynomial, the power of x is one less than the class number that the term represents. $P_n(x)$ will therefore only have terms with even powers of x. Thus, it is an even polynomial. ❑

Corollary: *$Q_n(x)$ is an odd polynomial, and $R_n(x)$ is an even polynomial.*

Proof: If $P(x)$ is even, then $xP(x)$ is odd, and $x^2P(x)$ is even. Using the identities in Theorem 5.2 and 5.3, we see that $Q(x)$ is odd and $R(x)$ is even. ❑

Theorem 5.5:

$$P_n(x) = P_{n-1}(x) + x^2 P_{n-2}(x) \tag{5.4}$$

Proof: Consider a class 1 codeword of width n. $P_n(x)$ only considers neighbors that cause the first and last bits not to transition. Given that the first and last bits are staying, the second bit can either stay or transition. If the second bit stays, then the second through nth bits form a class 1 codeword with a distribution described by $P_{n-1}(x)$. Because the first and second bits are both staying, there is still a dependent boundary between them, and the class numbers in $P_{n-1}(x)$ do not have to be adjusted. If the second bit transitions, then the second through nth bits form a class 1 codeword with a distribution described by $Q_{n-1}(x)$. However, the first and second bits are now the same, creating an independent boundary which increments the class numbers. Thus, the distribution for this possibility is $xQ_{n-1}(x)$, which by Theorem 5.2 can be written as $x^2 P_{n-2}(x)$. The overall distribution is the sum of these two possibilities, so $P_n(x) = P_{n-1}(x) + x^2 P_{n-2}(x)$. This is illustrated in Figure 5.6. ❑

Corollary:

$$Q_n(x) = Q_{n-1}(x) + x^2 Q_{n-2}(x) \tag{5.5}$$

$$R_n(x) = R_{n-1}(x) + x^2 R_{n-2}(x) \tag{5.6}$$

Proof: Multiplying both sides of (5.4) by x or x^2 and applying Theorem 5.2 or 5.3 respectively will generate the same recursion in terms of $Q_n(x)$ or $R_n(x)$. ❑

Theorem 5.6:

$$P_n(x) = \sum_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-j-1}{j} x^{2j} \tag{5.7}$$

Proof: Instead of proving this directly, we will express $P_n(x)$ in terms of another polynomial, and use the well-known properties of that polynomial to get our result. The *Fibonacci Polynomial* is defined by the recurrence equation:

$$F_n(u) = uF_{n-1}(u) + F_{n-2}(u) \tag{5.8}$$

with $F_1(u) = 1$ and $F_2(u) = u$. If we make the change of variables $u = 1/x$, we can rewrite (5.8) as

$$xF_n\left(\frac{1}{x}\right) = F_{n-1}\left(\frac{1}{x}\right) + xF_{n-2}\left(\frac{1}{x}\right). \tag{5.9}$$

Now, let us try the substitution

$$F_k\left(\frac{1}{x}\right) = x^{1-k} P_k(x)\,. \tag{5.10}$$

Plugging this into (5.9), we get

$$x^{2-n} P_n(x) = x^{2-n} P_{n-1}(x) + x^{4-n} P_{n-2}(x)\,. \tag{5.11}$$

Multiplying both sides by x^{n-2} results in an equation identical to (5.4). Thus, (5.10) is a valid identity, *provided it holds for two initial conditions as well.* (Two initial conditions are required because the difference equation is second-order.)

A 1-bit class 1 codeword is the codeword "0". It has two neighbors, 0 and 1, but only one of them leaves the first and last bits unchanged. That neighbor is, of course, itself, and the first and last bits refer to the same bit. A 2-bit class 1 codeword is "01". It has three neighbors, 00, 01, and 11, but only one of them, namely itself, leaves the first and last bits unchanged, because the codeword is only two bits to begin with. Thus, our initial conditions are:

$$P_1(x) = 1, \qquad P_2(x) = 1\,. \tag{5.12}$$

Trying these conditions with our identity in (5.10) results in

$$F_1\left(\frac{1}{x}\right) = 1, \qquad F_2\left(\frac{1}{x}\right) = \frac{1}{x}\,. \tag{5.13}$$

Changing variables back to u, we get

$$F_1(u) = 1, \qquad F_2(u) = u \tag{5.14}$$

which is identical to the initial conditions of the Fibonacci Polynomial given earlier. Thus (5.10) is a valid identity.

It has been shown [10] that the Fibonacci Polynomial can also be written as a sum of terms as follows:

$$F_n(x) = \sum_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-j-1}{j} x^{n-2j-1} \tag{5.15}$$

Simply applying (5.10) to (5.15) results in the equation in (5.7). ❑

Remark: Note that (5.7) is only valid for $n \geq 1$. In order to find $P_n(x)$ for $n \leq 0$, we must use Theorem 5.5 and iterate back to it. Interestingly, $P_n(x)$ is well-defined and consistent for for $n \leq 0$, although it loses physical meaning.

$$\begin{aligned}
P_0(x) &= 0\\
P_1(x) &= 1\\
P_2(x) &= 1\\
P_3(x) &= 1 + x^2\\
P_4(x) &= 1 + 2x^2\\
P_5(x) &= 1 + 3x^2 + x^4\\
P_6(x) &= 1 + 4x^2 + 3x^4\\
P_7(x) &= 1 + 5x^2 + 6x^4 + x^6\\
P_8(x) &= 1 + 6x^2 + 10x^4 + 4x^6\\
P_9(x) &= 1 + 7x^2 + 15x^4 + 10x^6 + x^8
\end{aligned}$$

Table 5.3 $P_n(x)$.

Remark: $P_n(x)$ is going to become a cornerstone of many of the expressions and theorems that follow in this chapter. However, a comparison of (5.7) and (5.15) reveals that $P_n(x)$ and the Fibonacci Polynomial have the exactly same coefficients, but in the opposite order. This implies that if we had defined the class distribution polynomial differently, or perhaps even defined the notion of class in terms of dependent boundaries instead of independent boundaries, we could be using $F_n(x)$ instead as the basis for class distribution theory. There is something to be said for attempting to express results in terms of classical functions whenever possible instead of introducing new functions, both because of the potential to leverage previous research and the stigma associated with renaming an old idea. However, using the Fibonacci Polynomial would have made the work awkward and messy, and was not worth it.* Thus, we stick with $P_n(x)$.

$P_n(x)$ for some values of n is listed in Table 5.3. A more extensive table of these polynomials can be found in Appendix B.

* In a similar vein, some thought was given to using the classical Fibonacci sequence F_n instead of d_n in the work in the previous chapter, since $d_n = F_{n+2}$. This proved to be extremely awkward. It was much more natural to simply define a new sequence d_n and express results in terms of it.

$$\boxed{\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} W_A & = & 011010 & W_B & = & 01101 & W_C & = & 01101001101 \\ & & \downarrow & & & \downarrow & & & \downarrow \\ A(x) & \rightarrow & ....1 & B(x) & \rightarrow & 0....... & C(x) & \rightarrow & ....10....... \end{array}}$$

Figure 5.7 Illustration of Theorem 5.7.

Theorem 5.7: *Let $A(x)$ be the distribution of some codeword W_A (not necessarily class 1) where its rightmost bit transitions. Let $B(x)$ be the distribution of some other codeword W_B where its leftmost bit does not transition. Further, let the rightmost bit of W_A and the leftmost bit of W_B have the same value. Suppose we form a new codeword W_C by concatenating the two codewords, $W_A W_B$, and let $C(x)$ be the distribution of W_C where the bits across the concatenation boundary transition and don't transition respectively, as illustrated in Figure 5.7. Then,*

$$C(x) = A(x)B(x)\,. \tag{5.16}$$

Proof: Because the rightmost bit of W_A and the leftmost bit of W_B are the same, the concatenation has an independent boundary where they were joined. According to Theorem 4.2, the set of transitions allowed on one side of the boundary cannot be restricted by the choice of a transition on the other side. Thus, each neighbor of W_A described by $A(x)$ can be paired with each neighbor of W_B described by $B(x)$. Suppose a neighbor of W_A with r_A independent boundaries is paired with a neighbor of W_B with r_b independent boundaries. The neighbor of W_C that is formed this way has $r_A + r_B$ independent boundaries, because the independent boundaries that were in W_A's neighbor and W_B's neighbor are still there, and no new independent boundary was created in the concatenation because W_A's neighbor's rightmost bit and W_B's neighbor's leftmost bit are different. We know this because $A(x)$ describes neighbors where the rightmost bit transitions and $B(x)$ describes neighbors where the leftmost bit does not transition, and both bits started out as the same value.

Thus, if there are k_A neighbors of W_A that have r_A independent boundaries, and k_B neighbors of W_B that have r_B independent boundaries, then they will generate

$$\boxed{\begin{array}{rcr@{\qquad}rcr@{\qquad}rcr} W_A &=& 011010 & W_B &=& 01101 & W_C &=& 01101001101 \\ & & \downarrow\; & & & \downarrow\quad\;\; & & & \downarrow\;\; \\ A(x) &\to& ....1 & B(x) &\to& 1....... & C(x) &\to& ....11....... \\ \\ W_A &=& 011010 & W_B &=& 01101 & W_C &=& 01101001101 \\ & & \downarrow\; & & & \downarrow\quad\;\; & & & \downarrow\;\; \\ A(x) &\to& ....0 & B(x) &\to& 0....... & C(x) &\to& ....00....... \end{array}}$$

Figure 5.8 Illustration for Second Corollary of Theorem 5.7.

$k_A\, k_B$ neighbors of W_C with $r_A + r_B$ independent boundaries. That is, the $k_A x^{r_A}$ term in $A(x)$ and the $k_B x^{r_B}$ term in $B(x)$ should combine somehow to form a $k_A\, k_B\, x^{r_A+r_B}$ term in $C(x)$. Furthermore, $C(x)$ should consist of the sum of all such terms that can be generated in this way by pairing terms of $A(x)$ and terms of $B(x)$. Polynomial multiplication is exactly this operation. Thus $C(x)$ can be formed by simply multiplying $A(x)$ and $B(x)$. ❑

Corollary: *If we instead let $A(x)$ be the distribution where W_A's rightmost bit does not transition, and $B(x)$ be the distribution where W_B's leftmost bit does transition, and swap the definition of $C(x)$ accordingly, (5.16) still holds.*

Proof: This is exactly equivalent to the situation in the Theorem, but with the bit ordering reversed. ❑

Corollary: *If we let $A(x)$ be the distribution where W_A's rightmost bit transitions and $B(x)$ be the distribution where W_B's leftmost bit transitions, or if we let $A(x)$ be the distribution where W_A's rightmost bit does not transition and $B(x)$ be the distribution where W_B's leftmost bit does not transition, and restrict the transitions across the concatenation boundary in W_C accordingly as illustrated in Figure 5.8, then we get*

$$C(x) = xA(x)B(x)\,. \tag{5.17}$$

Proof: In both cases, the two bits around the concatenation boundary in the neighbors of W_C are the same. This creates an additional independent boundary, and thus the powers of x must all be incremented by one. ❑

Definition: $M_n(x)$ is the two-by-two matrix of polynomials:

$$M_n = \begin{bmatrix} xP_n & P_n + x^2 P_{n-1} \\ P_n + (x^2 - 1)P_{n-1} & xP_n + xP_{n-1} \end{bmatrix} \tag{5.18}$$

where the "(x)" suffix has been omitted for clarity.

Theorem 5.8: *Suppose we have* $[\,A(x) \quad B(x)\,]$, *where* $A(x)$ *is now the distribution of some codeword* W *where the rightmost bit transitions, and* $B(x)$ *is the distribution where the rightmost bit does not transition. Now, let us form a new codeword* W_{new} *by adding an* n*-bit section to the end. The new distributions* $[\,A_{new}(x) \quad B_{new}(x)\,]$, *defined as above, can be calculated as*

$$[\,A_{new}(x) \quad B_{new}(x)\,] = [\,A(x) \quad B(x)\,]M_n(x) \tag{5.19}$$

where standard matrix multiplication is implied.

Proof: Because the n-bit section being added is, in isolation, a class 1 codeword, the $P_n(x)$, $Q_n(x)$, and $R_n(x)$ distributions, as defined, apply to it.

First, we will calculate $A_{new}(x)$. Neighbors described by $A_{new}(x)$ must only be those that cause the rightmost bit of W_{new} to transition. That rightmost bit is the rightmost bit of the new section. Given that the rightmost bit of the section must transition, we see by definition that $R_n(x)$ is the distribution of the section where its leftmost bit transitions, and $Q_n(x)$ is the distribution where the leftmost bit stays. We know that $A(x)$ is the distribution of W where the rightmost bit transitions, and $B(x)$ is the distribution where the rightmost bit stays. Furthermore, adding a section to a codeword by definition entails adding an independent boundary, meaning that the leftmost bit of the section and the rightmost bit of W are the same value. We are concatenating W and the section, and thus can use the results of Theorem 5.7 and its corollaries.

$$\begin{aligned}
A_{new}(x) = \quad & xA(x)R_n(x) && \textit{(both bits across junction tranistion)} \\
+\ & xB(x)Q_n(x) && \textit{(neither bit across junction tranisitions)} \\
+\ & A(x)Q_n(x) && \textit{(transition on left side of junction only)} \\
+\ & B(x)R_n(x) && \textit{(transition on right side of junction only)}
\end{aligned} \quad \textbf{(5.20)}$$

Calculation of $B_{new}(x)$ is similar. Neighbors described by $B_{new}(x)$ must only be those that cause the rightmost bit of W_{new}, and thus the section, to stay. Given that the rightmost bit of the section must stay, we see by definition that $Q_n(x)$ is the distribution of the section where its leftmost bit transitions, and $P_n(x)$ is the distribution where the leftmost bit stays.

$$\begin{aligned}
B_{new}(x) = \quad & xA(x)Q_n(x) && \textit{(both bits across junction tranistion)} \\
+\ & xB(x)P_n(x) && \textit{(neither bit across junction tranisitions)} \\
+\ & A(x)P_n(x) && \textit{(transition on left side of junction only)} \\
+\ & B(x)Q_n(x) && \textit{(transition on right side of junction only)}
\end{aligned} \quad \textbf{(5.21)}$$

We find that we can write (5.20) and (5.21) compactly in matrix form:

$$[\,A_{new} \quad B_{new}\,] = [\,A \quad B\,]\begin{bmatrix} xR_n + Q_n & xQ_n + P_n \\ xQ_n + R_n & xP_n + Q_n \end{bmatrix} \quad \textbf{(5.22)}$$

Substituting the identities in Theorems 5.2 and 5.3 for $Q_n(x)$ and $R_n(x)$, we can write the matrix on the right as

$$\begin{bmatrix} x^3P_{n-2} + xP_{n-1} & x^2P_{n-1} + P_n \\ x^2P_{n-1} + x^2P_{n-2} & xP_n + xP_{n-1} \end{bmatrix}. \quad \textbf{(5.23)}$$

We can then use the recursion formula in Theorem 5.5 to write (5.23) entirely in terms of $P_n(x)$ and $P_{n-1}(x)$.

$$\begin{bmatrix} xP_n & P_n + x^2P_{n-1} \\ P_n + (x^2-1)P_{n-1} & xP_n + xP_{n-1} \end{bmatrix} \quad \textbf{(5.24)}$$

This is by definition the $M_n(x)$ matrix, and thus (5.19) holds. ❑

Theorem 5.9:

$$D_{\{n_1,n_2,\ldots,n_c\}}(x) = \frac{1}{x+1}[1 \quad 1]\left(\prod_{i=1}^{c} M_{n_i}(x)\right)\begin{bmatrix}1\\1\end{bmatrix} \tag{5.25}$$

Proof: Consider the codeword $W = \{n_1, n_2, \ldots, n_c\}$. If we let $A_1(x)$ be the distribution of the first section of W where its rightmost bit transitions, and $B_1(x)$ be the distribution where its rightmost bit stays, and define $A_W(x)$ and $B_W(x)$ similarly for the entire codeword, Theorem 5.8 can be applied $(c-1)$ times to give us

$$[\,A_W(x) \quad B_W(x)\,] = [\,A_1(x) \quad B_1(x)\,]\left(\prod_{i=2}^{c} M_{n_1}(x)\right) \tag{5.26}$$

Although (5.26) is a valid and even usable formula for calculating the distribution of a codeword, it is not especially convenient because the first section is treated special. We would prefer to express $[\,A_1(x) \quad B_1(x)\,]$ in terms of $M_{n_1}(x)$, so the first section simply can be incorporated into the product. Because $A_1(x)$ and $B_1(x)$ describe the distribution of a class 1 codeword, we can write them in terms of $P_n(x)$, $Q_n(x)$, and $R_n(x)$. Simply from the definitions, we see that

$$[\,A_1(x) \quad B_1(x)\,] = [\,R_{n_1}(x) + Q_{n_1}(x) \quad Q_{n_1}(x) + P_{n_1}(x)\,]. \tag{5.27}$$

Recalling from (5.22) that $M_n(x)$ can be written as

$$\begin{bmatrix} xR_n + Q_n & xQ_n + P_n \\ xQ_n + R_n & xP_n + Q_n \end{bmatrix} \tag{5.28}$$

we can factor out the matrix in (5.27) to find that

$$[\,R_{n_1}(x) + Q_{n_1}(x) \quad Q_{n_1}(x) + P_{n_1}(x)\,] = \begin{bmatrix} \frac{1}{x+1} & \frac{1}{x+1} \end{bmatrix}\begin{bmatrix} xR_{n_1} + Q_{n_1} & xQ_{n_1} + P_{n_1} \\ xQ_{n_1} + R_{n_1} & xP_{n_1} + Q_{n_1} \end{bmatrix} \tag{5.29}$$

or equivalently,

$$[\,A_1(x) \quad B_1(x)\,] = \frac{1}{x+1}[1 \quad 1]M_{n_1}(x) \tag{5.30}$$

Combining this identity with (5.26) results in

$$[\,A_W(x) \quad B_W(x)\,] = \frac{1}{x+1}[1 \quad 1]\left(\prod_{i=1}^{c} M_{n_i}(x)\right) \tag{5.31}$$

Because $A_W(x)$ represents the distribution when the rightmost bit of W transitions, and $B_W(x)$ represents the distribution when the rightmost bit does not transition, the total distribution is simply the sum of these two:

$$D_{\{n_1,n_2,\ldots,n_c\}}(x) = [\,A_W(x) \quad B_W(x)\,]\begin{bmatrix}1\\1\end{bmatrix} \tag{5.32}$$

Combining (5.31) with (5.32) gives us our final answer:

$$D_{\{n_1,n_2,\ldots,n_c\}}(x) = \frac{1}{x+1}[\,1 \quad 1\,]\left(\prod_{i=1}^{c} M_{n_i}(x)\right)\begin{bmatrix}1\\1\end{bmatrix} \tag{5.33}$$

which is identical to (5.25). ❏

Remark: Recall from (4.9) that the expression for computing the total degree of a codeword is

$$d_{\{n_1,n_2,\ldots,n_c\}} = \prod_{i=1}^{c} d_{n_i} \tag{5.34}$$

Notice how similar in form this expression is to (5.33). Both involve the product over all sections of an entity that depends only on the section width. The distribution polynomial, however, depends on the order the sections are in, and this is reflected by the fact that matrix multiplication is non-commutative. It is, however, associative, which means that

$$\prod_{i=1}^{c} M_{n_i}(x) \tag{5.35}$$

can be computed by grouping the sections in any way we choose. Once we have (5.35), finding $D_{\{n_1,n_2,\ldots,n_c\}}(x)$ is a simple matter of summing the elements of the matrix and dividing by $(x+1)$.

Table 5.4 lists $M_n(x)$ for a few values of n. A more extensive table of these matrices and some of their products can be found in Appendix C.

We have now derived the method for computing the class distribution polynomial. Before we go on to wield it and prove lots of interesting results, it is worth mentioning that the use of a polynomial to represent class distribution is admittedly completely a gimmick. The author feels this way because the polynomial is *never*

$$M_1(x) = \begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}$$

$$M_2(x) = \begin{bmatrix} x & 1+x^2 \\ x^2 & 2x \end{bmatrix}$$

$$M_3(x) = \begin{bmatrix} x+x^3 & 1+2x^2 \\ 2x^2 & 2x+x^3 \end{bmatrix}$$

$$M_4(x) = \begin{bmatrix} x+2x^3 & 1+3x^2+x^4 \\ 2x^2+x^4 & 2x+3x^3 \end{bmatrix}$$

$$M_5(x) = \begin{bmatrix} x+3x^3+x^5 & 1+4x^2+3x^4 \\ 2x^2+3x^4 & 2x+5x^3+x^5 \end{bmatrix}$$

$$M_6(x) = \begin{bmatrix} x+4x^3+3x^5 & 1+5x^2+6x^4+x^6 \\ 2x^2+5x^4+x^6 & 2x+7x^3+4x^5 \end{bmatrix}$$

Table 5.4 $M_n(x)$.

actually evaluated, except possibly at $x = 1$ to represent the sum of the terms. The variable x has no physical meaning; it simply serves as a placeholder. In fact, if we chose to represent class distribution with a vector instead of a polynomial, the theory would work out exactly the same, except with vector convolution in place of polynomial multiplication, as shown in Figure 5.9. (This is in fact how the class distribution functions were implemented with Matlab.) There are three reasons for using polynomials instead of the somewhat more honest approach of using vectors. First, matrices of vectors look awkward. Secondly, (5.33) and (5.34) can be written in a similar form, as a product over sections, which emphasizes that they are related. Thirdly, we are all familiar with the operation of factoring polynomials, whereas few of us learned to deconvolve vectors in high school algebra.

A further justification has to do with the similarity of $P_n(x)$ to the Fibonacci Polynomial. If expressing the work in terms of polynomials results in a classical, well-known function popping up, then we are probably doing something right.

polynomial multiplication

$$\left(x+3x^3+x^5\right)\left(1+6x^2+10x^4+4x^6\right) \;=\; x+9x^3+29x^5+40x^7+22x^9+4x^{11}$$

vector convolution

$$<0,1,0,3,0,5> * <1,0,6,0,10,0,4> \quad = \quad <0,1,0,9,0,29,0,40,0,22,0,4>$$

Figure 5.9 Polynomial Multiplication Versus Vector Convolution.

5.5 Properties of the Class Distribution Polynomial

Some experimentation with the class distribution polynomial reveals an interesting fact. Many codewords have a particular class number below which they have no neighbors whatsoever. For example, the codeword $\{1,2,3,4,5\}$ can only transition to codewords in class 4 or higher; it has no neighbors in classes 1 through 3. Exploration of this phenomenon proves useful for the analysis of class pruning, so we will derive a method for computing the lowest class that a given codeword can transition to. Because this derivation is somewhat complicated, it has been broken up into a number of lemmas, which will be combined in Theorem 5.10.

Definition: Let oT, where T is a matrix of polynomials, denote a matrix that consists of the *lowest-ordered term* of each element of T, with the coefficient set to one.* For example,

$$o\begin{bmatrix} x^3+x & 2x^2+1 \\ x^4+2x^2 & 3x^3+2x \end{bmatrix} = \begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix} \tag{5.36}$$

Lemma 5.1: *For two matrices T and S,*

$$o(TS) = o((oT)(oS)) \tag{5.37}$$

Proof: Let us represent T and S in a general form as

$$T = \begin{bmatrix} k_{a1}x^a + k_{a2}x^{a+1} + \dots & k_{c1}x^c + k_{c2}x^{c+1} + \dots \\ k_{b1}x^b + k_{b2}x^{b+1} + \dots & k_{d1}x^d + k_{d2}x^{d+1} + \dots \end{bmatrix}$$

* Do not confuse this with the customary usage of o, which is typically used to refer to an unspecified polynomial with the argument as the *highest-ordered* term.

$$S = \begin{bmatrix} k_{e1}x^e + k_{e2}x^{e+1} + \ldots & k_{g1}x^g + k_{g2}x^{g+1} + \ldots \\ k_{f1}x^f + k_{f2}x^{f+1} + \ldots & k_{h1}x^h + k_{h2}x^{h+1} + \ldots \end{bmatrix} \tag{5.38}$$

where "..." represents higher order terms. If we calculate the top-left element of TS, we get

$$\begin{aligned} &k_{a1}k_{e1}x^{a+e} + (k_{a1}k_{e2} + k_{a2}k_{e1})x^{a+e+1} + \ldots \\ &+ k_{c1}k_{f1}x^{c+f} + (k_{c1}k_{f2} + k_{c2}k_{f1})x^{c+f+1} + \ldots \end{aligned} \tag{5.39}$$

Applying the o operation to (5.39) results in

$$o\big(x^{a+e} + x^{c+f}\big) \tag{5.40}$$

which is the same as the top-left element of $o((oT)(oS))$. The other three elements of the matrix can similarly be verified. ❑

With the previous lemma established, the distributive property of the o operator will often be used implicitly from this point on.

Lemma 5.2:

$$oM_n = \begin{cases} \begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix} & \textit{for } n > 1 \\ \begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix} & \textit{for } n = 1 \end{cases} \tag{5.41}$$

Proof: (5.7) tells us that the lowest term of $P_n(x)$ is

$$\binom{n-j-1}{j} x^{2j} \bigg|_{j=0} = \binom{n-1}{0} = 1 \tag{5.42}$$

That is, a class 1 codeword can only transition to one class 1 codeword, namely, itself. However (5.7) is only valid for $n \geq 1$. With the help of Theorem 5.5, we can calculate $P_0(x)$:

$$P_0(x) = \frac{1}{x^2}(P_2(x) - P_1(x)) = \frac{1}{x^2}(1-1) = 0 \tag{5.43}$$

Thus, the lowest term of $P_n(x)$ and $P_{n-1}(x)$ is always 1, *except* for $n = 1$, for which $P_{n-1}(x)$ equals zero. Plugging this information into the definition of $M_n(x)$

$$\begin{bmatrix} xP_n & P_n + x^2P_{n-1} \\ P_n + (x^2-1)P_{n-1} & xP_n + xP_{n-1} \end{bmatrix} \tag{5.44}$$

and taking the lowest term of each element, we see that (5.41) holds. ❑

Lemma 5.3: *If, for some matrix T, oT is equal to*

$$\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix} \tag{5.45}$$

or any mirror image thereof (horizontally flipped, vertically flipped, or both), then

$$o\left(T\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}\right) \tag{5.46}$$

is equal to oT flipped horizontally.

Proof: Let us represent oT in a general form as

$$\begin{bmatrix} x^a & x^c \\ x^b & x^d \end{bmatrix} \tag{5.47}$$

(5.46) is then equal to

$$o\left(\begin{bmatrix} x^a & x^c \\ x^b & x^d \end{bmatrix}\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}\right) = o\begin{bmatrix} x^{a+1} + x^c & x^{c+1} + x^a \\ x^{b+1} + x^d & x^{d+1} + x^b \end{bmatrix} \tag{5.48}$$

When we apply the o operation to the matrix on the right, the higher of the two terms in each element will disappear. We see that the term that is of the form x^{k+1} will always be higher or equal to the term that it is being added to, as long as

$$-1 \leq a - c \leq 1 \qquad \text{and} \qquad -1 \leq b - d \leq 1 \tag{5.49}$$

It is easy to see that in (5.45) or any mirror image, $(a-c)$ and $(b-d)$ are always ± 1. Thus, (5.49) holds, and we can neglect the x^{k+1} terms in (5.48). This leaves us with

$$\begin{bmatrix} x^c & x^a \\ x^d & x^b \end{bmatrix} \tag{5.50}$$

which is indeed the horizontal mirror image of (5.47). ❏

Lemma 5.4: *If, for some matrix T, oT is equal to*

$$\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix} \tag{5.51}$$

or any mirror image thereof (horizontally flipped, vertically flipped, or both), then

$$o\left(\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}T\right) \tag{5.52}$$

is equal to oT flipped vertically.

Proof: This proof is very similar to that of Lemma 5.3. If we represent oT as in (5.45), then (5.52) is equal to

$$o\left(\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}\begin{bmatrix} x^a & x^c \\ x^b & x^d \end{bmatrix}\right) = o\begin{bmatrix} x^{a+1}+x^b & x^{c+1}+x^d \\ x^{b+1}+x^a & x^{d+1}+x^c \end{bmatrix} \qquad \textbf{(5.53)}$$

The terms of the form x^{k+1} will disappear, as long as

$$-1 \le a-b \le 1 \qquad \text{and} \qquad -1 \le c-d \le 1 \qquad \textbf{(5.54)}$$

In (5.51) or any mirror image, $(a-b)$ and $(c-d)$ are always ± 1. Thus, (5.54) holds, and we can neglect the x^{k+1} terms in (5.53). This leaves us with

$$o\begin{bmatrix} x^b & x^d \\ x^a & x^c \end{bmatrix} \qquad \textbf{(5.55)}$$

which is indeed the vertical mirror image of (5.47). ❑

Lemma 5.5: *Let T be some matrix such that*

$$oT = \begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix}. \qquad \textbf{(5.56)}$$

Then,

$$o(T^k) = x^{k-1} oT \qquad \textbf{(5.57)}$$

Proof:

$$o\left(\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix}\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix}\right) = \begin{bmatrix} x^2 & x \\ x^3 & x^2 \end{bmatrix} = x\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix} \qquad \textbf{(5.58)}$$

If we apply (5.58) $k-1$ times, we come out with (5.57). ❑

Lemma 5.6: *Let T and S be some matrices such that*

$$oT = oS = \begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix} \qquad \textbf{(5.59)}$$

Then,

$$o\left(T\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix}S\right) = oT \qquad \textbf{(5.60)}$$

Proof:

$$o\left(\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix}\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix}\right) = o\left(\begin{bmatrix} 1 & x \\ x & x^2 \end{bmatrix}\begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix}\right) = \begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix} \qquad \textbf{(5.61)}$$

❑

We are now equipped to tackle the theorem.

Theorem 5.10: *Consider a codeword W. Let W' be W with all pairs of adjacent 1-bit sections removed. Let k be the number of boundaries between adjacent non-1-bit sections in W'. The lowest class number in the neighbor set of W is $k+1$.*

Proof: Consider a codeword $W = \{n_1, n_2, \ldots, n_c\}$. According to Lemma 5.1,

$$o\left(\prod_{i=1}^{c} M_{n_i}\right) = o\left(\prod_{i=1}^{c} oM_{n_i}\right) \tag{5.62}$$

Therefore, we need only consider oM_{n_i}, never M_{n_i} itself. From Lemma 5.2 we know that

$$oM_n = \begin{cases} \begin{bmatrix} x & 1 \\ x^2 & x \end{bmatrix} & \text{for } n > 1 \\ \begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix} & \text{for } n = 1 \end{cases} \tag{5.63}$$

and (5.62) is thus the product of some combination of the two matrices in (5.63). From Lemmas 5.3 and 5.4, we know that multiplying by an oM_1 matrix simply results in a flip. Since two flips in a row cancel each other out, we know we can remove adjacent pairs of oM_1 matrices without affecting the final result. The product now consists of strings of $oM_{n>1}$ matrices separated by single oM_1 matrices. By Lemma 5.5, we know that the product of $(k+1)$ $oM_{n>1}$ matrices is equal to $(x^k\, oM_{n>1})$. If we apply this rule to each string of $oM_{n>1}$ matrices, letting k_j represent the length of each string minus one, we get a factor of $x^{\sum_j k_j}$ times an alternating pattern of $oM_{n>1}$ matrices and oM_1 matrices. Employing Lemma 5.6, we can collapse this down to a single $oM_{n>1}$ matrix, possibly with a oM_1 matrix to the left and/or right of it. By Lemmas 5.3 and 5.4, this is simply some mirror image of $oM_{n>1}$, and thus its lowest term is 1. Therefore, the lowest term in

$$\prod_{i=1}^{c} M_{n_1}(x) \tag{5.64}$$

has the power $\sum_j k_j$ and consequently, so does the lowest term in the sum of the elements of (5.64). Dividing by $(x+1)$ does not change the power of the lowest term,

$$W = \{2, 3, 1, 1, 1, 4, 1, 1, 5, 6, 1, 7\}$$

$$W' = \{2 \underset{\uparrow}{,} 3, 1, 4 \underset{\uparrow}{,} 5 \underset{\uparrow}{,} 6, 1, 7\} \qquad \text{(all } \{1, 1\} \text{ pairs removed)}$$

Three boundaries between non-1-bit sections.

Lowest class in W's neighbor set is 4.

Figure 5.10 Example of Theorem 5.10.

and thus the lowest term of

$$D_{\{n_1,n_2,\ldots,n_c\}}(x) = \frac{1}{x+1}\begin{bmatrix} 1 & 1 \end{bmatrix}\left(\prod_{i=1}^{c} M_{n_i}(x)\right)\begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \textbf{(5.65)}$$

has this power. This means that the lowest class number in the neighbor set is equal to

$$1 + \sum_j k_j \qquad \textbf{(5.66)}$$

which is equivalent to what is stated in the theorem. An example is shown in Figure 5.10. ❑

Corollary: *A codeword that contains no 1-bit sections cannot transition to any codeword in a class lower than its own.*

Proof: If c is the codeword class, than such a codeword has $(c-1)$ boundaries between non-1-bit sections, and by the Theorem, the lowest class number in its neighbor set is $((c-1)+1)$, or c. ❑

Our next theorem concerns an interesting and useful relationship that shows up when a codeword is bitwise rotated through an independent boundary.

Theorem 5.11: *Consider a codeword $W = \{n_1, n_2, \ldots, n_c\}$. Let W_r be the codeword that results when W is rotated through r independent boundaries. That is,*

$$W_r = \{n_{r+1}, n_{r+2}, \ldots, n_c, n_1, n_2, \ldots, n_r\} \qquad \textbf{(5.67)}$$

Although W and W_r may have different distributions, the sum of the neighbors in the pair of classes, k and $k+1$, is the same for all r, where k is any even number if c is even, or any odd number if c is odd. That is, the sum of the coefficients of the x^{k-1} and x^k terms of the distribution stay constant when the codeword is rotated through any independent boundary.

Proof: By Theorem 5.4, we know that $P_n(x)$ is an even polynomial. Studying the definition of $M_n(x)$

$$\begin{bmatrix} xP_n & P_n + x^2P_{n-1} \\ P_n + (x^2-1)P_{n-1} & xP_n + xP_{n-1} \end{bmatrix} \tag{5.68}$$

we see that it has the form

$$\begin{bmatrix} odd & even \\ even & odd \end{bmatrix} \tag{5.69}$$

It is easy to verify that, for any matrices of polynomials,

$$\begin{bmatrix} odd & even \\ even & odd \end{bmatrix}\begin{bmatrix} odd & even \\ even & odd \end{bmatrix} = \begin{bmatrix} even & odd \\ odd & even \end{bmatrix} \tag{5.70}$$

and

$$\begin{bmatrix} even & odd \\ odd & even \end{bmatrix}\begin{bmatrix} odd & even \\ even & odd \end{bmatrix} = \begin{bmatrix} odd & even \\ even & odd \end{bmatrix} \tag{5.71}$$

It follows that

$$\prod_{i=1}^{c} M_{n_i} = \begin{cases} \begin{bmatrix} odd & even \\ even & odd \end{bmatrix} & \text{if } c \text{ is odd} \\ \begin{bmatrix} even & odd \\ odd & even \end{bmatrix} & \text{if } c \text{ is even} \end{cases} \tag{5.72}$$

Now, let us realize that rotation through the rth independent boundary is equivalent to splitting the codeword at this boundary, and swapping the two halves. That is,

$$D_{W_r} = \frac{1}{x+1}[1 \quad 1]\left(\prod_{i=r+1}^{c} M_{n_i}(x) \prod_{i=1}^{r} M_{n_i}(x)\right)\begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{5.73}$$

This can also be seen by simply examining (5.67). Let us say that

$$\prod_{i=1}^{r} M_{n_i} = \begin{bmatrix} A_1 & C_1 \\ B_1 & E_1 \end{bmatrix} \qquad \text{and} \qquad \prod_{i=r+1}^{c} M_{n_i} = \begin{bmatrix} A_2 & C_2 \\ B_2 & E_2 \end{bmatrix} \tag{5.74}$$

We then calculate that

$$\prod_{i=1}^{r} M_{n_i} \prod_{i=r+1}^{c} M_{n_i} = \begin{bmatrix} A_1A_2 + C_1B_2 & A_1C_2 + C_1E_2 \\ B_1A_2 + E_1B_2 & B_1C_2 + E_1E_2 \end{bmatrix} \tag{5.75}$$

$$\boxed{\begin{array}{lllllllllllllll}
D_{\{1,2,3,4,5\}} = & & 27x^3 & +72x^4 & +210x^5 & +375x^6 & +542x^7 & +636x^8 & +557x^9 & +398x^{10} & +206x^{11} & +77x^{12} & +18x^{13} & +2x^{14} \\
D_{\{2,3,4,5,1\}} = & & 27x^3 & +72x^4 & +210x^5 & +375x^6 & +542x^7 & +636x^8 & +557x^9 & +398x^{10} & +206x^{11} & +77x^{12} & +18x^{13} & +2x^{14} \\
D_{\{3,4,5,1,2\}} = & 9x^2 & +18x^3 & +111x^4 & +171x^5 & +420x^6 & +497x^7 & +646x^8 & +547x^9 & +399x^{10} & +205x^{11} & +77x^{12} & +18x^{13} & +2x^{14} \\
D_{\{4,5,1,2,3\}} = & 9x^2 & +18x^3 & +105x^4 & +177x^5 & +405x^6 & +512x^7 & +641x^8 & +552x^9 & +399x^{10} & +205x^{11} & +77x^{12} & +18x^{13} & +2x^{14} \\
D_{\{5,1,2,3,4\}} = & 9x^2 & +18x^3 & +106x^4 & +176x^5 & +412x^6 & +505x^7 & +646x^8 & +547x^9 & +398x^{10} & +206x^{11} & +77x^{12} & +18x^{13} & +2x^{14} \\
 & \multicolumn{2}{c}{\underbrace{\hphantom{xxxxx}}_{27}} & \multicolumn{2}{c}{\underbrace{\hphantom{xxxxx}}_{282}} & \multicolumn{2}{c}{\underbrace{\hphantom{xxxxx}}_{917}} & \multicolumn{2}{c}{\underbrace{\hphantom{xxxxx}}_{1193}} & \multicolumn{2}{c}{\underbrace{\hphantom{xxxxx}}_{604}} & \multicolumn{2}{c}{\underbrace{\hphantom{xxxxx}}_{95}} &
\end{array}}$$

Figure 5.11 Example of Theorem 5.11.

and

$$\prod_{i=r+1}^{c} M_{n_i} \prod_{i=1}^{r} M_{n_i} = \begin{bmatrix} A_1A_2 + B_1C_2 & C_1A_2 + E_1C_2 \\ A_1B_2 + B_1E_2 & C_1B_2 + E_1E_2 \end{bmatrix} \tag{5.76}$$

Now, let us assume that c is odd. When we sum the elements of (5.75) or (5.76), the odd part of the result will be the sum of the top-left element and the bottom-right element, as implied by (5.72). Notice that this sum is the same for both (5.75) and (5.76). That is,

$$\begin{aligned}
& \operatorname{Odd}\left(\begin{bmatrix} 1 & 1 \end{bmatrix} \prod_{i=1}^{r} M_{n_i} \prod_{i=r+1}^{c} M_{n_i} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\
= \; & \operatorname{Odd}\left(\begin{bmatrix} 1 & 1 \end{bmatrix} \prod_{i=r+1}^{c} M_{n_i} \prod_{i=1}^{r} M_{n_i} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \\
= \; & A_1A_2 + B_1C_2 + C_1B_2 + E_1E_2
\end{aligned} \tag{5.77}$$

This means that the odd terms of $(x+1)D_{W_r}$ are constant for all choices of r. If the odd terms of $D_{W_r} + x\, D_{W_r}$ are constant, then the x^k coefficient plus the x^{k-1} coefficient of D_{W_r} must be constant for all odd k. If we instead assume that c is even, we can simply replace the word "odd" with "even" in the above argument. An example is shown in Figure 5.11. ❑

Corollary: *Even if all codewords in the classes less than c are pruned away, the degree after pruning of W_r is the same for all r.*

Proof: By the Theorem, we know that the sum of the neighbors in classes c and $c+1$ is constant with respect to r, as well as classes $c+2$ and $c+3$, classes $c+4$ and

$c + 5$, and so on. This may leave one class unpaired at the end, but we know that every codeword has exactly two neighbors in class n. Thus, the sum of the neighbors in classes c through n is constant with respect to r. ❑

As a result of Theorem 5.11 and its corollary, we often will be able to disregard the rotation of a codeword when dealing with class pruning. That is, we will often be able to prove a result for one specific codeword, and have the result hold for all rotations of that codeword.

Unlike the previous two theorems, which are quite useful, the following two formulas are unfortunately rather unwieldy and difficult to apply. They are here because I calculated them, and because there might someday be a use for them, however unlikely that appears. Derivations are not shown.

Useless Formula 5.1: $M_aM_b =$

$$\begin{bmatrix} (x^2+1)P_aP_b + x^2P_{a-1}P_b + (x^2-1)P_aP_{b-1} + (x^4-x^2)P_{a-1}P_{b-1} \\ 2xP_aP_b + x^3P_{a-1}P_b + (x^3+x)P_aP_{b-1} + x^3P_{a-1}P_{b-1} \\ 2xP_aP_b + x^3P_{a-1}P_b + (x^3-x)P_aP_{b-1} + (x^3-x)P_{a-1}P_{b-1} \\ (x^2+1)P_aP_b + (2x^2-1)P_{a-1}P_b + 2x^2P_aP_{b-1} + x^4P_{a-1}P_{b-1} \end{bmatrix} \tag{5.78}$$

Useless Formula 5.2:

$$M_n^k = \begin{bmatrix} a_1u^k + a_2v^k & c_1u^k + c_2v^k \\ b_1u^k + b_2v^k & e_1u^k + e_2v^k \end{bmatrix} \tag{5.79}$$

where

$$q = \sqrt{P_n^2 + (2x^2+1)P_nP_{n-1} + \left(x^4 - \tfrac{3}{4}\right)P_{n-1}^2} \tag{5.80}$$

$$u = xPn + \tfrac{1}{2}P_{n-1} + q, \qquad v = xPn + \tfrac{1}{2}P_{n-1} - q \tag{5.81}$$

$$a_1 = \frac{1}{2} - \frac{xP_{n-1}}{4q}, \qquad a_2 = \frac{1}{2} + \frac{xP_{n-1}}{4q} \tag{5.82}$$

$$b_1 = \frac{P_n + (x^2-1)P_{n-1}}{2q}, \qquad b_2 = -\frac{P_n + (x^2-1)P_{n-1}}{2q} \tag{5.83}$$

$$c_1 = \frac{P_n + x^2P_{n-1}}{2q}, \qquad c_2 = -\frac{P_n + x^2P_{n-1}}{2q} \tag{5.84}$$

$$e_1 = \frac{1}{2} + \frac{xP_{n-1}}{4q}, \qquad e_2 = \frac{1}{2} - \frac{xP_{n-1}}{4q} \tag{5.85}$$

For k = 0, (5.79) evaluates to the identity matrix, as we would expect. For general k and most values of n, (5.79) evaluates to a horrible mess. But for $n = 1$, it's fairly tame:

$$M_1^k = \begin{bmatrix} \frac{1}{2}(x+1)^k + \frac{1}{2}(x-1)^k & \frac{1}{2}(x+1)^k - \frac{1}{2}(x-1)^k \\ \frac{1}{2}(x+1)^k - \frac{1}{2}(x-1)^k & \frac{1}{2}(x+1)^k + \frac{1}{2}(x-1)^k \end{bmatrix} \qquad \mathbf{(5.86)}$$

(5.86) is somewhat interesting, because it says that diagonally opposed elements in M_1^k are equal, and the elements are the even and odd parts of the binomial expansion. Furthermore, noticing that the codeword $\{1\}^n$ is a class n codeword, whose neighbor set is the entire set of codewords, summing the elements of (5.86) for M_1^n and dividing by $(x + 1)$ as in Theorem 5.9 provides an alternate proof of Theorem 4.7.

5.6 Limiting Codewords with Class Pruning

We now have a way of quickly computing the degree of any codeword W at any stage of class pruning. We simply generate the class distribution polynomial D_W and sum the coefficients of x^k and higher terms, where classes k and below have been pruned. Alternately, we can sum the coefficients of all terms lower than x^k, and subtract the result from d_W. Either way, we obtain the answer analytically instead of through an expensive experimentation process.

However, simply being able to calculate the degree of a codeword is not enough. If we wish to find the maximum performance of this code, we need a pruning curve. And to generate a pruning curve, we have to know the minimum degree in each class, and preferably the codeword responsible for this minimum. Otherwise, generating the curve would still require going through every codeword at every step, searching for the minimum degree. Because there are only n steps, this would lead to a computational complexity of $o(n2^n)$, which is much better than the $o(2^{3n})$ complexity of the optimal algorithm, but still unacceptable.

Fortunately, it is clear from experimentation exactly what the limiting codewords are for each class. Unfortunately, *the theory has not yet developed to the point where this can be proven.* Therefore, we must humbly submit it as a conjecture.

Conjecture 5.1: *If all codewords in classes lower than some class c have been pruned*

c	$n = 10$	$n = 11$	$n = 12$	$n = 13$
1	{10}	{11}	{12}	{13}
2	{2,8}	{2,9}	{2,10}	{2,11}
3	{2,2,6}	{2,2,7}	{2,2,8}	{2,2,9}
4	{2,2,2,4}	{2,2,2,5}	{2,2,2,6}	{2,2,2,7}
5	{2,2,2,2,2}	{2,2,2,2,3}	{2,2,2,2,4}	{2,2,2,2,5}
6	{2,1,2,1,2,2}	{2,1,2,2,2,2}	{2,2,2,2,2,2}	{2,2,2,2,2,3}
7	{2,1,1,2,1,2,1}	{2,1,2,1,2,1,2}	{2,1,2,1,2,2,2}	{2,1,2,2,2,2,2}
8	{2,1,1,1,2,1,1,1}	{2,1,1,2,1,1,2,1}	{2,1,2,1,2,1,2,1}	{2,1,2,1,2,1,2,2}
9	{2,1,1,1,1,1,1,1,1}	{2,1,1,1,1,2,1,1,1}	{2,1,1,2,1,1,2,1,1}	{2,1,1,2,1,2,1,2,1}
10	{1,1,1,1,1,1,1,1,1,1}	{2,1,1,1,1,1,1,1,1,1}	{2,1,1,1,1,2,1,1,1,1}	{2,1,1,1,2,1,1,2,1,1}
11		{1,1,1,1,1,1,1,1,1,1,1}	{2,1,1,1,1,1,1,1,1,1,1}	{2,1,1,1,1,1,2,1,1,1,1}
12			{1,1,1,1,1,1,1,1,1,1,1,1}	{2,1,1,1,1,1,1,1,1,1,1,1}
13				{1,1,1,1,1,1,1,1,1,1,1,1,1}

Table 5.5 Limting Codewords In Class Pruning.

away, the set of codewords in class c with the minimum degree include:

$$\{2\}^{c-1}\{n-2c+2\} \quad \textit{and all rotations,} \qquad \text{for } 1 \le c \le \tfrac{n}{2}$$

$$\{2,1\}^{2c-n}\{2\}^{2n-3c} \quad \textit{and all rotations,} \qquad \text{for } \tfrac{n}{2} \le c \le \tfrac{2n}{3}$$

$$\left\{\{2\}\{1\}^{\left\lfloor\frac{c}{n-c}\right\rfloor}\right\}^{c-(n-c)\left\lfloor\frac{c}{n-c}\right\rfloor}\left\{\{2\}\{1\}^{\left\lfloor\frac{c}{n-c}-1\right\rfloor}\right\}^{(n-c)\left\lfloor\frac{c}{n-c}+1\right\rfloor-c}$$

$$\textit{and all rotations,} \qquad \text{for } \tfrac{2n}{3} \le c < n \qquad \mathbf{(5.87)}$$

Because the above formulation is somewhat convoluted, the codewords will be described a little less formally. For $1 \le c \le \frac{n}{2}$, the limiting codewords are simply those with the most 2-bit sections. For $\frac{n}{2} \le c < n$, the codewords are again those with the most 2-bit sections, with the 1-bit sections distributed as uniformly as possible between the 2-bit sections, and regions with the same number of 1-bit sections between 2-bit sections grouped together. The still confused reader is advised to examine Table 5.5 for some examples. Note the careful wording in the conjecture—these codewords are not necessarily the *only* codewords with the limiting degree. But there are no codewords with a lower degree.

There are several things to notice about these codewords. For one thing, they form a subset of the codewords described by Theorem 4.5, which described the set of limiting codewords *without* pruning. That is, these codewords have the minimum degree in the class, both before and after class pruning. Another interesting point is that, by Theorem 5.10 and particularly its corollary, the degree of these codewords for $1 \leq c \leq \frac{n}{2}$ does not change after pruning. Thus, we know immediately from the corollary of Theorem 4.5 exactly what the limiting degree is:

$$3^{c-1} d_{n-2c+2} \qquad \text{for } 1 \leq c \leq \tfrac{n}{2} \tag{5.88}$$

Unfortunately, the global maximum of the class pruning curve is almost never in this region, but it is still a noteworthy fact.

Conjecture 5.1 might be considered to be "unsafe", because the conclusions that we draw from it are optimistic rather than pessimistic. That is, if the conjecture is wrong, the true performance of the code is worse than what we state. However, the power of Theorem 5.11 makes it easy to experimentally verify the conjecture for a wide range of n, and it has always been found to hold.

Furthermore, with some effort and a lot of lemmas, we can employ *lowest-term analysis* to prove the conjecture for a couple of special cases.

Definition: Let ℓT, where T is a matrix of polynomials, denote a matrix that consists of the *lowest-ordered term* of each element of T. This is similar to the o operator introduced earlier, except we do not discard the coefficient. For example,

$$\ell \begin{bmatrix} x^3 + x & 2x^2 + 1 \\ x^4 + 2x^2 & 3x^3 + 2x \end{bmatrix} = \begin{bmatrix} x & 1 \\ 2x^2 & 2x \end{bmatrix} \tag{5.89}$$

Lemma 5.7: *For two matrices T and S,*

$$\ell(TS) = \ell((\ell T)(\ell S)) \tag{5.90}$$

Proof: Let us represent T and S in a general form as

$$\begin{aligned} T &= \begin{bmatrix} k_{a1}x^a + k_{a2}x^{a+1} + \dots & k_{c1}x^c + k_{c2}x^{c+1} + \dots \\ k_{b1}x^b + k_{b2}x^{b+1} + \dots & k_{d1}x^d + k_{d2}x^{d+1} + \dots \end{bmatrix} \\ S &= \begin{bmatrix} k_{e1}x^e + k_{e2}x^{e+1} + \dots & k_{g1}x^g + k_{g2}x^{g+1} + \dots \\ k_{f1}x^f + k_{f2}x^{f+1} + \dots & k_{h1}x^h + k_{h2}x^{h+1} + \dots \end{bmatrix} \end{aligned} \tag{5.91}$$

where "…" represents higher order terms. If we calculate the top-left element of TS, we get

$$\begin{aligned} &k_{a1}k_{e1}x^{a+e} + (k_{a1}k_{e2} + k_{a2}k_{e1})x^{a+e+1} + \dots \\ +\, &k_{c1}k_{f1}x^{c+f} + (k_{c1}k_{f2} + k_{c2}k_{f1})x^{c+f+1} + \dots \end{aligned} \tag{5.92}$$

Applying the ℓ operator to (5.92) results in

$$\ell\big(k_{a1}k_{e1}x^{a+e} + k_{c1}k_{f1}x^{c+f}\big) \tag{5.93}$$

which is the same as the top-left element of $\ell((\ell T)(\ell S))$. The other three elements of the matrix can similarly be verified. ❑

With the previous lemma established, the distributive property of the ℓ operator will often be used implicitly from this point on.

Lemma 5.8:

$$\ell M_n = \begin{cases} \begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix} & \textit{for } n = 1 \\ \begin{bmatrix} x & 1 \\ x^2 & 2x \end{bmatrix} & \textit{for } n = 2 \\ \begin{bmatrix} x & 1 \\ 2x^2 & 2x \end{bmatrix} & \textit{for } n \geq 3 \end{cases} \tag{5.94}$$

Proof: The $n = 1$ and $n = 2$ cases can be computed directly. For $n \geq 3$, we again must examine the definition of M_n:

$$\begin{bmatrix} xP_n & P_n + x^2P_{n-1} \\ P_n + (x^2-1)P_{n-1} & xP_n + xP_{n-1} \end{bmatrix} \tag{5.95}$$

We know from (5.42) that the lowest term of P_n is 1 for $n \geq 1$. This is enough to compute ℓ of all elements of M_n except the bottom-left. Turning back to (5.7), we can compute the second lowest term of P_n:

$$\binom{n-j-1}{j}x^{2j}\bigg|_{j=1} = \binom{n-2}{1}x^2 = (n-2)x^2 \quad \text{for } n \geq 3 \tag{5.96}$$

The lowest term of the bottom-left element of M_n is thus:

$$\begin{aligned} &\ell\Big(\big((n-2)x^2+1\big) + \big(x^2-1\big)\big((n-3)x^2+1\big)\Big) \\ &\quad = ((n-2) - (n-3) + 1)x^2 \quad = \quad 2x^2 \end{aligned} \tag{5.97}$$

Therefore,

$$\ell M_n = \begin{bmatrix} x & 1 \\ 2x^2 & 2x \end{bmatrix} \qquad \text{for } n \geq 3 \tag{5.98}$$

as was stated in (5.94). ❑

Lemma 5.9:

$$\ell M_1^k = \begin{cases} \begin{bmatrix} k\,x & 1 \\ 1 & k\,x \end{bmatrix} & \textit{for odd } n \\ \begin{bmatrix} 1 & k\,x \\ k\,x & 1 \end{bmatrix} & \textit{for even } n \end{cases} \tag{5.99}$$

Proof: Let us define a matrix T_a such that

$$\ell T_a = \begin{bmatrix} a\,x & 1 \\ 1 & a\,x \end{bmatrix} \tag{5.100}$$

Then,

$$\ell(T_a M_1) = \ell\left(\begin{bmatrix} a\,x & 1 \\ 1 & a\,x \end{bmatrix}\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}\right) = \begin{bmatrix} 1 & (a+1)x \\ (a+1)x & 1 \end{bmatrix} \tag{5.101}$$

Furthermore,

$$\begin{aligned} \ell\big(T_a\, M_1^2\big) &= \ell\left(\begin{bmatrix} 1 & (a+1)x \\ (a+1)x & 1 \end{bmatrix}\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}\right) \\ &= \begin{bmatrix} (a+2)x & 1 \\ 1 & (a+2)x \end{bmatrix} = \ell T_{a+2} \end{aligned} \tag{5.102}$$

Recognizing that $\ell T_1 = M_1$ and following the recursion established above, we see that (5.99) holds. An alternate proof can be derived by examining (5.86). ❑

Lemma 5.10:

$$\ell M_2^k = x^{k-1} \begin{bmatrix} d_{2k-3}\,x & d_{2k-2} \\ d_{2k-2}\,x^2 & d_{2k-1}\,x \end{bmatrix} \tag{5.103}$$

Proof: Suppose we have a sequence of matrices T_k such that

$$\ell T_k = \begin{bmatrix} a_k\,x & c_k \\ b_k\,x^2 & e_k\,x \end{bmatrix} \tag{5.104}$$

Then,

$$\begin{aligned} \ell(T_k M_2) &= \ell\left(\begin{bmatrix} a_k\,x & c_k \\ b_k\,x^2 & e_k\,x \end{bmatrix}\begin{bmatrix} x & 1 \\ x^2 & 2x \end{bmatrix}\right) \\ &= x\begin{bmatrix} (a_k+c_k)x & (a_k+2c_k) \\ (b_k+e_k)x^2 & (b_k+2e_k)x \end{bmatrix} = x\,\ell T_{k+1} \end{aligned} \tag{5.105}$$

This suggests the following system of recurrence equations for the top row of ℓT_k:

$$\begin{aligned} a_{k+1} &= a_k + c_k \\ c_{k+1} &= a_k + 2c_k \end{aligned} \qquad \textbf{(5.106)}$$

We can combine these equations to yield the individual recurrence equations

$$\begin{aligned} a_k &= 3a_{k-1} - a_{k-2} \\ c_k &= 3c_{k-1} - c_{k-2} \end{aligned} \qquad \textbf{(5.107)}$$

which in fact describe the same recurrence. Noticing that coefficients in the bottom rows of the matrices in (5.105) are identical in form to those in the top rows, we conclude that b_k and e_k follow this recursion as well. Now, consider the recurrence equation followed by d_n:

$$d_n = d_{n-1} + d_{n-2} \qquad \textbf{(5.108)}$$

Creatively transposing and resubstituting (5.108) a few times leads to the identity

$$d_n = 3d_{n-2} - d_{n-4} \qquad \textbf{(5.109)}$$

Making the substitution $n = 2k$ in the above equation gives us

$$d_{2k} = 3d_{2(k-1)} - d_{2(k-2)} \qquad \textbf{(5.110)}$$

Comparing (5.110) and (5.107), we see that d_{2k} follows the same recursion as a_k, b_k, c_k, and e_k. Now we must establish two initial conditions. If we let $T_1 = M_2$ and $T_2 = M_2^2$, we find that

$$\begin{aligned} a_1 &= 1, \quad a_2 = 2 \\ b_1 &= 1, \quad b_2 = 3 \\ c_1 &= 1, \quad c_2 = 3 \\ e_1 &= 2, \quad e_2 = 5 \end{aligned} \qquad \textbf{(5.111)}$$

Comparing these numbers to d_n, we can conclude that:

$$\begin{aligned} a_k &= d_{2k-3} \\ b_k &= d_{2k-2} \\ c_k &= d_{2k-2} \\ e_k &= d_{2k-1} \end{aligned} \tag{5.112}$$

Given that $T_1 = M_2$, and $\ell(T_k M_2) = x\,\ell T_{k+1}$ as in (5.105), (5.112) leads directly to (5.103). ❑

Lemma 5.11:

$$\frac{d_{2k}}{3^k} \tag{5.113}$$

is monotonic decreasing for $k \geq 1$, *and non-increasing for* $k \geq 0$.

Proof: We can state this as

$$\frac{d_{2(k+1)}}{3^{k+1}} < \frac{d_{2k}}{3^k} \tag{5.114}$$

Expressing d_n in terms of (4.3), we can write this inequality as

$$\phi^2\left(\frac{\phi^2}{3}\right)^{k+1} - \phi^{-2}\left(\frac{1}{3\phi^2}\right)^{k+1} < \phi^2\left(\frac{\phi^2}{3}\right)^{k} - \phi^{-2}\left(\frac{1}{3\phi^2}\right)^{k} \tag{5.115}$$

Factoring, we get

$$\left(\frac{\phi^4}{3} - \phi^2\right)\left(\frac{\phi^2}{3}\right)^k < \left(\frac{1}{3\phi^4} - \frac{1}{\phi^2}\right)\left(\frac{1}{3\phi^2}\right)^k \tag{5.116}$$

Interestingly, both constant terms in parentheses above are equal to $-\frac{1}{3}$. (A clue as to why can be found in (5.110).) Thus, we get

$$\left(-\frac{1}{3}\right)\left(\frac{\phi^2}{3}\right)^k < \left(-\frac{1}{3}\right)\left(\frac{1}{3\phi^2}\right)^k \tag{5.117}$$

Cross-multiplying gives us

$$\phi^{4k} > 1 \tag{5.118}$$

which is true for $k > 0$. This proves that (5.113) is monotonic decreasing for $k \geq 1$. Now, simply observing that

$$\frac{d_0}{3^0} = \frac{d_2}{3^1} = 1 \tag{5.119}$$

proves that (5.113) is non-increasing for $k \geq 0$. A plot is shown in Figure 5.12. An alternate proof can be derived by taking a closer look at (5.109). ❑

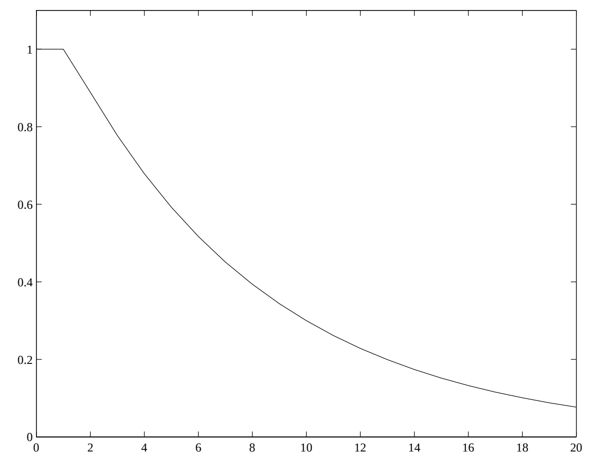


Figure 5.12 $d_{2n}/3^n$.

Definition: d_{pW} denotes the degree of the codeword W after all codewords in classes lower than the class of W have been pruned away.

Theorem 5.12: *Consider the codeword $W = \{2, 2, 2, \ldots\}$, or $\{2\}^c$. Let W' be W with any two 2-bit sections replaced with a 1-bit section and a 3-bit section. Then,*

$$d_{pW} \leq d_{pW'} \tag{5.120}$$

Proof: We know from the corollary of Theorem 5.10 that $d_{pW} = d_W$, because W is unaffected by class pruning. Thus, from Theorem 4.3, we can calculate d_{pW} simply to be 3^c. Finding $d_{pW'}$ is not so trivial, because W' is affected by class pruning. However, it is not affected very much. From Theorem 5.10, we see that the only neighbors of W' that are pruned are in classes $c-1$ and possibly $c-2$.* Furthermore, Theorem 5.11 tells us that the sum of the neighbors in these two classes is constant for any rotation of W'. More importantly, we can see from the proof of Theorem 5.11 exactly what this sum is. It is equal to the lowest term of the sum of the top-left and bottom-right elements of

$$\prod_{i=1}^{c} M_{n_i}(x) \tag{5.121}$$

* Specifically, if the 1-bit section is not placed on the end of the codeword, there will be neighbors is class $c-2$.

If we can calculate this sum, we can simply subtract it from $d_{W'}$ to find $d_{pW'}$, which is what we seek. We know from Lemma 5.7 that

$$\ell \prod_{i=1}^{c} M_{n_i}(x) = \ell \prod_{i=1}^{c} \ell M_{n_i}(x) \tag{5.122}$$

and thus we need only deal with ℓM_{n_i}, not M_{n_i} itself. W' can be described by a rotation of some codeword

$$\{2\}^j\{3\}\{2\}^k\{1\} \tag{5.123}$$

where $c = j + k + 2$. With the help of Lemmas 5.8 and 5.10, we can calculate the lowest-term matrix of this codeword. (The ℓ operator is implied for all of the following expressions.)

$$\begin{aligned}
&\left(\ell M_2^j\, \ell M_3\right)\left(\ell M_2^k\, \ell M_1\right) \\
&= \left(x^{j-1}\begin{bmatrix} d_{2j-3}\,x & d_{2j-2} \\ d_{2j-2}\,x^2 & d_{2j-1}\,x \end{bmatrix}\begin{bmatrix} x & 1 \\ 2x^2 & 2x \end{bmatrix}\right)\left(x^{k-1}\begin{bmatrix} d_{2k-3}\,x & d_{2k-2} \\ d_{2k-2}\,x^2 & d_{2k-1}\,x \end{bmatrix}\begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}\right) \\
&= x^{j+k-1}\begin{bmatrix} d_{2j}x & d_{2j} \\ d_{2j+1}x^2 & d_{2j+1}x \end{bmatrix}\begin{bmatrix} d_{2k-2} & d_{2k-1}x \\ d_{2k-1}x & d_{2k} \end{bmatrix} \\
&= x^{j+k}\begin{bmatrix} d_{2j}d_{2k} & d_{2j}d_{2k+1}\,x \\ d_{2j+1}d_{2k}\,x & d_{2j+1}d_{2k+1}\,x^2 \end{bmatrix}
\end{aligned} \tag{5.124}$$

where the identities

$$\begin{aligned} d_n &= d_{n-1} + d_{n-2} \\ d_n &= 2d_{n-2} + d_{n-3} \end{aligned} \tag{5.125}$$

were applied as needed. The sum of the top-left and bottom-right elements of this matrix is

$$d_{2j}d_{2k}\,x^{j+k} \;+\; d_{2j+1}d_{2k+1}\,x^{j+k+2} \tag{5.126}$$

and the coefficient of the lowest term is simply $d_{2j}d_{2k}$. This is then the number of neighbors this codeword has in classes $c-1$ and $c-2$. Thus, we have

$$\begin{aligned} d_{pW'} &= d_{W'} - d_{2j}d_{2k} \\ &= 10 \times 3^{j+k} - d_{2j}d_{2k} \end{aligned} \tag{5.127}$$

We can now rewrite (5.120) as

$$3^{j+k+2} \leq 10 \times 3^{j+k} - d_{2j} d_{2k} \tag{5.128}$$

Rearranging, we see this is equivalent to

$$\frac{d_{2j} d_{2k}}{3^{j+k}} \leq 1 \tag{5.129}$$

We have equality in (5.129) for $j = k = 0$. By Lemma 5.11, we see that the left hand side of (5.129) cannot increase for any positive values of j and k. Therefore, (5.120) holds. ❑

Theorem 5.13: *Consider a codeword* $W = \{a, 2, 2, 2, \ldots\} = \{a\}\{2\}^{c-1}$ *or any rotation thereof, where* $a > 2$. *Let* W' *be* W *with any two 2-bit sections replaced with a 1-bit section and a 3-bit section. Then,*

$$d_{pW} < d_{pW'} \tag{5.130}$$

Proof: As in the proof of Theorem 5.12, W is unaffected by class pruning, and the change in the degree of W' can be calculated by examining the lowest terms of the product of M_{n_i}. W' can be described as a rotation and/or reversal of some codeword

$$\{2\}^i\{a\}\{2\}^j\{3\}\{2\}^k\{1\} \tag{5.131}$$

where $c = i + j + k + 3$. With the help of Lemmas 5.8 and 5.10, we can calculate the lowest-term matrix of this codeword. Notice that $\ell\big(M_2^j\, M_3\, M_2^k\, M_1\big)$ has already been calculated in (5.124), so that saves us some work. The ℓ operator is implied for all of the following expressions.

$$\begin{aligned}
&\big(\ell M_2^i\, \ell M_a\big)\big(\ell M_2^j\, \ell M_3\, \ell M_2^k\, \ell M_1\big) \\
&= \left(x^{i-1}\begin{bmatrix} d_{2i-3}\, x & d_{2i-2} \\ d_{2i-2}\, x^2 & d_{2i-1}\, x \end{bmatrix}\begin{bmatrix} x & 1 \\ 2x^2 & 2x \end{bmatrix}\right)\left(x^{j+k}\begin{bmatrix} d_{2j} d_{2k} & d_{2j} d_{2k+1}\, x \\ d_{2j+1} d_{2k}\, x & d_{2j+1} d_{2k+1}\, x^2 \end{bmatrix}\right) \\
&= x^{i+j+k}\begin{bmatrix} d_{2i} x & d_{2i} \\ d_{2i+1} x^2 & d_{2i+1} x \end{bmatrix}\begin{bmatrix} d_{2j} d_{2k} & d_{2j} d_{2k+1}\, x \\ d_{2j+1} d_{2k}\, x & d_{2j+1} d_{2k+1}\, x^2 \end{bmatrix} \\
&= x^{i+j+k+1}\begin{bmatrix} d_{2i} d_{2j+2} d_{2k} & d_{2i} d_{2j+2} d_{2k+1}\, x \\ d_{2i+1} d_{2j+2} d_{2k}\, x & d_{2i+1} d_{2j+2} d_{2k+1}\, x^2 \end{bmatrix}
\end{aligned} \tag{5.132}$$

The coefficient of the lowest term of the sum of the top-left and bottom-right elements is simply $d_{2i}d_{2j+2}d_{2k}$. This is then the number of neighbors this codeword has in classes $c-1$ and $c-2$. Thus, we have

$$\begin{aligned} d_{pW'} &= d_{W'} - d_{2i}\, d_{2j+2}\, d_{2k} \\ &= 10d_a\, 3^{j+k} - d_{2i}\, d_{2j+2}\, d_{2k} \end{aligned} \tag{5.133}$$

We can now rewrite (5.130) as

$$d_a\, 3^{i+j+k+2} < 10d_a\, 3^{j+k} - d_{2i}d_{2j+2}d_{2k} \tag{5.134}$$

Rearranging, we see this is equivalent to

$$\frac{d_{2i}\, d_{2j+2}\, d_k}{3^{i+j+k}} < d_a \tag{5.135}$$

The inequality is true for $i = j = k = 0$ and $a = 3$. We see that the left hand side of (5.135) is non-increasing for positive i, j, and k, and the right hand side is monotonic increasing for all a that we are concerned with. Therefore, (5.130) holds. ❏

Conjecture 5.1 described the codewords with the limiting degree for each class after pruning. Theorems 5.12 and 5.13 demonstrate that changing this limiting codeword by "one step", that is, shifting a bit from one section to another, results in the degree increasing for $c < \frac{n}{2}$ and either increasing or staying constant for $c = \frac{n}{2}$. This is obviously not a full proof of the conjecture, but it's a start. The problem is that these theorems rely on a trick—recognizing that the degrees of the limiting codewords do not change at all with pruning, and that the degrees of codewords that are one step away from them change only by a lowest term that is easy to derive. Changing the limiting codeword by two steps makes things much more complicated, because more than just a lowest term is involved. Likewise, the degrees of the limiting codewords for $c > \frac{n}{2}$ are themselves affected by class pruning, so this lowest term trick simply won't work. Be aware that the *calculation* of these degrees is still very easy and straightforward. The difficulty comes when we try to prove a result for the general case.

5.7 Results

With the data obtained from the optimal pruning algorithm, it is fair to say that we have determined the fundamental limits of the performance of self-shielding codes. As in the last chapter, we have not designed a code, but simply found the best any code can do given the specifications. However, unlike in the last chapter where the code design was left completely arbitrary, the pruning process gives a specific set of codewords to use. The extra performance over the last chapter's code comes from restricting ourselves to this set.

For large n, we can use the class pruning algorithm in lieu of the optimal one. This does not give us the fundamental limit on code performance, but it appears to be fairly close. Furthermore, with faith in Conjecture 5.1 (or alternatively, a small amount of experimentation), we can calculate class pruning results quickly and analytically.

Figure 5.13 plots the performance of the pruned code, and compares it to the other codes discussed. The data shown is from the class pruning algorithm; optimal data would visually coincide with the plotted line for $n \leq 23$, so it need not be plotted separately. Table 5.6 lists the channel widths required to transit data of various widths. We see that a 32-bit bus could be implemented with 40 wires, which compares quite favorably to the other schemes. If we conceptually consider $n - b$, channel width minus data bits, to be the number of "extra wires" required to eliminate crosstalk delay, shielding uses 31 extra wires, whereas pruned coding uses merely 8.

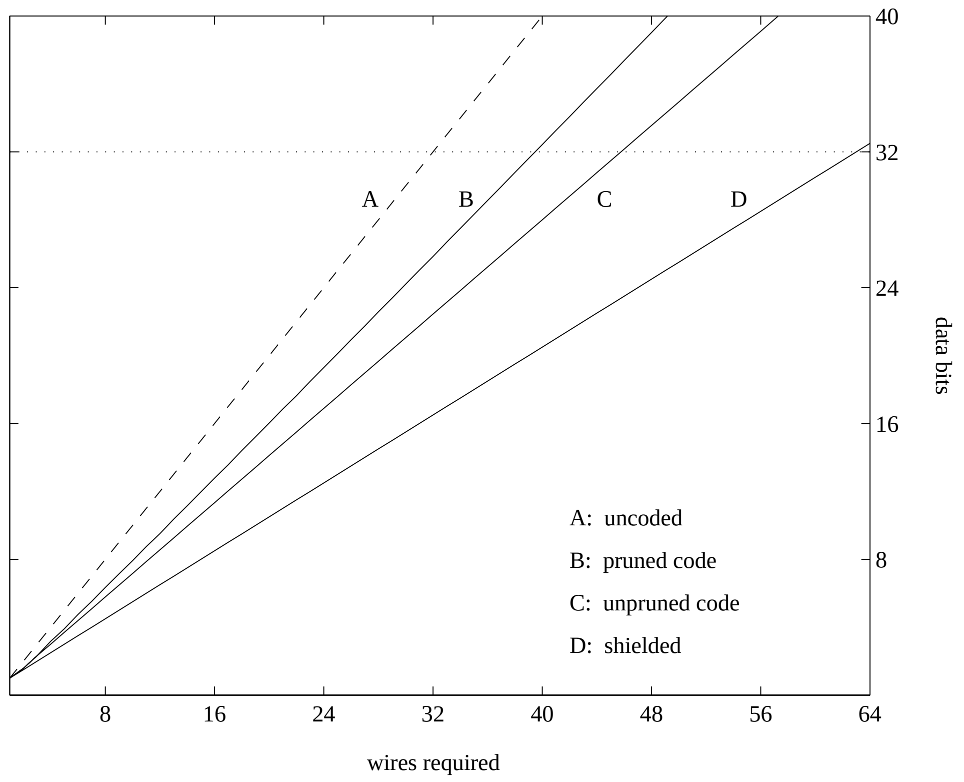


Figure 5.13 Performance of Pruned Code.

bits	wires required			bits	wires required		
	pruned	*unpruned*	*shielded*		*pruned*	*unpruned*	*shielded*
1	1	1	1	17	22	25	33
2	3	3	3	18	23	26	35
3	4	4	5	19	24	28	37
4	6	6	7	20	25	29	39
5	7	7	9	21	27	30	41
6	8	9	11	22	28	32	43
7	9	10	13	23	29	33	45
8	11	12	15	24	30	35	47
9	12	13	17	25	31	36	49
10	13	15	19	26	33	38	51
11	14	16	21	27	34	39	53
12	16	17	23	28	35	41	55
13	17	19	25	29	36	42	57
14	18	20	27	30	38	43	59
15	19	22	29	31	39	45	61
16	20	23	31	32	40	46	63

Table 5.6 Performance of Pruned Code.

Chapter 6

Memoryless Code

(Connecting the Neighbors)

6.1 Introduction

The previous chapter explored the maximum possible performance obtainable with a self-shielding code. The basic idea was that, for each codeword allowed on the channel, a codebook with 2^b entries could be constructed. The encoder and decoder would both be aware of the current codeword on the channel and the mapping between codewords and codebooks. When the encoder placed the next word on the channel, the decoder would look it up in the current codebook and translate it to a symbol, as well as change the current codebook to reflect the new word on the channel.

Although this scheme gives the maximum performance possible, it may not be easy to design or implement such a code. Code design involves mapping 2^b data words to codewords, and is a difficult task, especially when the designer is aiming for an efficient implementation. But our coding scheme as described requires designing a 2^b-entry codebook *for each codeword.* The designer thus has to juggle something on the order of $2^{n-1}2^b$ codebook entries. Of course, with the proper analysis techniques and

CAD tools, this job may be feasible, or even easily automated. But until the theory and toolset have advanced to that point, we will give the poor designer a break, and consider what performance can be obtained with a single 2^b-sized codebook.

The codebook in such a code is fixed—the mapping between symbols and codewords is not dependent on the codeword previously on the channel, or on anything else. Because the codebook does not change, this encoding is *memoryless*.

6.2 Analysis of the Memoryless Code

In the codes with memory, the reason multiple codebooks had to be used was that most codewords could not transition to every valid codeword. They could only transition to a subset of the codeword set, so the codebooks associated with them had to reflect the mapping between the symbols and that particular subset. Now, we are restricted to a single codebook. This codebook must be used when any codeword is on the channel, and the codebook must contain 2^b entries, mapping every symbol to every valid codeword. These two observations lead to the fundamental stipulation behind memoryless coding: *every codeword in the codebook must be able to transition to every other codeword in the codebook*. Our goal, in determining the maximum performance of a memoryless code, is to find the largest such codebook.

In graph theory, a *clique* in an undirected graph is defined as a subgraph where every pair of nodes is connected with an edge.* We can build a graph with all possible codewords as vertices and represent valid transitions with edges. The largest memoryless codebook is then represented by the largest clique in this graph.

In the following two theorems, we will prove that this largest clique is in fact the neighbor set of a class 1 codeword.

Theorem 6.1: *The entire neighbor set of a codeword is a clique if and only if the codeword is class 1.*

Proof: *("if" case):* A class 1 codeword has only dependent boundaries, so the pair of bits across any boundary is either 01 or 10. Consider a boundary between a 01 pair.

* A clique is sometimes referred to as a *complete subgraph*, or a *fully-connected subgraph*. Do not confuse this with a *strongly-connected subgraph*, which applies to directed graphs.

All neighbors of the codeword will have either a 00, 01, or 11 across that boundary. No neighbor can have a 10 across the boundary because that would violate the Fundamental Rule. Notice that all three possibilities, 00, 01, and 11, can transition to one another. Therefore, every neighbor can transition to every other neighbor without violating the Fundamental Rule across that boundary. Consider now a boundary between a 10 pair. All neighbors have either 00, 10, or 11 across this boundary, and again, these three pairs can all transition to each other. This argument holds for every boundary in the codeword. The Fundamental Rule cannot be violated across any boundary when any neighbor transitions to another. Thus, the neighbor set of a class 1 codeword is a clique.
("only if" case): A codeword in class $c > 1$ has, by definition, at least one independent boundary. Consider the pair of bits across an independent boundary, either 00 or 11. There are neighbors of this codeword with 00, 01, 10, and 11 bit pairs across this boundary. However, the neighbors with 01 across the boundary cannot transition to the neighbors with 10 across the boundary, because that would violate the Fundamental Rule. Thus, the neighbor set of a codeword in class $c > 1$ is not a clique. ❑

Definition: In accordance with the above theorem, we can use the term *class 1 clique* to refer to the neighbor set of a class 1 codeword.

Definition: A clique is said to be *prime* if no codeword can be added to the set with the set remaining a clique. (This does not imply that the clique is the largest possible clique; it simply means that, given the codewords already in the set, there is no room to grow.)

We can see readily that the class 1 clique is prime, because if any codeword were added to the set, the class 1 codeword would not be able to transition to it. However, we do not know that the class 1 clique is the largest clique. To prove that, we need another theorem.

case 1	case 2	case 3
↓	↓	↓
0000	0000	0000
0001	$\underline{00}01$	$0\underline{01}0$
1000	$01\underline{11}$	$0\underline{10}0$
1001	1111	0110
does not contain 1111	*does not contain* 0011	*not a clique*

Figure 6.1 Example of Lemma 6.1.

Definition: To say that a bit boundary in a set of codewords is *01-type* implies that in the set, there are codewords with 00, 01, and 11 across that boundary, but no codewords with 10 across the boundary. A *10-type* boundary similarly implies that there are codewords in the set with 00, 10, and 11, but not 01, across that boundary. Be aware that 01-type and 10-type boundaries are concepts defined for entire sets of codewords, not individual codewords.

Lemma 6.1: *In a prime clique, every boundary is either 01-type or 10-type.*

Proof: There are three ways in which a boundary in a set of codewords can be neither 01-type nor 10-type. The first is if no codeword in the set has 00 across the boundary, or if no codeword has 11 across the boundary. The second is if all codewords in the set have either 00 or 11 across the boundary. The third is if there are codewords with both 01 and 10 across the boundary. These three cases are illustrated in Figure 6.1. Consider the first case. Such a boundary cannot occur in a prime clique, because every prime clique must contain the 0000... and 1111... codewords. Next, consider a clique with a boundary of the second case. If we select one codeword with 00 across the boundary and another with 11 across the boundary, we can form a new codeword by taking the portion of the first codeword to the left of the boundary and concatenating it with the portion of the second codeword to the right of the boundary. This new codeword is not a member of the clique, but it can transition to all codewords in the clique. Thus, the clique is not prime. Now, consider a set of codewords with a boundary of the third case. The codewords with 01 across the boundary cannot transition to the codewords with 10 across the boundary. Thus, the set is not even a clique. Because none of these cases can occur in a prime clique, every boundary must be 01-type or 10-type. ❑

Theorem 6.2: *The class 1 clique is the largest clique.*

Proof: As implied by Lemma 6.1, there are 2^{n-1} different prime cliques for a given n, because each boundary can be one of the two types. We will first derive a method for calculating the size of each prime clique. For a given clique of width n, let x_n and y_n be the number of members whose rightmost bits are 0 and 1 respectively. Now we wish to extend the width by one by adding a boundary and a bit, and calculate x_{n+1} and y_{n+1}. But we must choose the boundary to be either 01-type or 10-type. Suppose we choose 01-type. This means that the last bit pair can take on the values 00, 01, or 11. Specifically, there can be x_n codewords of length $n+1$ that end in 00, x_n codewords that end in 01, and y_n codewords that end in 11. The codewords that end in 00 contribute toward x_{n+1}, and the codewords that end in 01 and 11 contribute toward y_{n+1}. We can express this as:

$$x_{n+1} = x_n, \quad y_{n+1} = x_n + y_n \qquad \textit{(01-type boundary)} \tag{6.1}$$

Now, suppose we had chosen to add a 10-type boundary instead. Following similar reasoning, we find that

$$x_{n+1} = x_n + y_n, \quad y_{n+1} = x_n \qquad \textit{(10-type boundary)} \tag{6.2}$$

Now that we've defined some recurrences, we need initial conditions. Consider a prime clique of 1-bit codewords. There is only one such clique, consisting of the codewords 0 and 1. Thus,

$$x_1 = 1, \quad y_1 = 1 \tag{6.3}$$

We now have a method for computing the size of a prime clique, given the types of its boundaries. We can't simply solve these recurrence equations though, because the proper equation to use at each step depends on the boundary type at that point. Instead, we will summarize our method as an algorithm.

Algorithm 6.1: *Size of Prime Clique*

```
$x = y = 1$
For each boundary from 1 to $n - 1$:
  If the boundary is 01-type, $y = x + y$
  If the boundary is 10-type, $x = x + y$
Clique size $= x + y$
```

We wish to construct a clique with boundary types assigned in such a way as to maximize $x + y$ at the end of the algorithm. It can be seen that the optimal decision at each step of the algorithm, in order to maximize the running total of x and y, is to choose:

$$\begin{aligned} x &= x + y \quad \textit{(or 10-type boundary)} \qquad \text{when } x < y \\ y &= x + y \quad \textit{(or 01-type boundary)} \qquad \text{when } x > y \end{aligned} \tag{6.4}$$

Suppose, at some point in the algorithm, $x < y$. In accordance with (6.4), we choose $x = x+y$. Notice that now, $x > y$. In the next step, we apply (6.4) again and choose $y = x + y$. This brings us back to the $x < y$ situation. We see therefore that the optimal decision flops back and forth between the two. This corresponds to a clique where the boundary type alternates between 01-type and 10-type. A class 1 codeword is a member of this clique; therefore, this is a class 1 clique. Because $x = y$ at the start of the algorithm, the initial choice of boundary type is arbitrary. Thus, there are two largest prime cliques, both class 1 cliques, one of each polarity. ❏

Remark: It is interesting to take a closer look at some of the other prime cliques. For example, consider a clique with only 01-type boundaries. For $n = 5$, the members of this clique are 00000, 00001, 00011, 00111, 01111, 11111. These are often referred to as *thermometer* codewords, because with some imagination, they resemble the mercury in a thermometer. In general (and less formally), a prime clique may have regions of alternating-type boundaries and same-type boundaries; we might refer to the former as “class 1 regions” and the latter as “thermometer regions”. Throughout the clique, the bits in a class 1 region resemble neighbors of a class 1 codeword, whereas the bits in a thermometer region resemble a thermometer code. This is unimportant.

6.3 Results

The theory developed in this chapter has demonstrated that the largest set of codewords that can all transition to one another is the neighbor set of a class 1 codeword. This means that the maximum performance of a memoryless code is the same as that of the unpruned code with memory. Specifically, for a given n, we can have d_n codewords in the codebook.

Note that this code, like the pruned code with memory in the last chapter, comes with a specific set of codewords to use. The set found in the previous chapter gave us additional performance beyond d_n; the set found in this chapter gives us no additional performance, but provides a code property that may considerably ease implementation. In some sense, it’s remarkable that we can place on our code the severe restriction of a single, fixed codebook, and still come out with a performance of d_n.

Figure 6.2 plots the performance of this code, and Table 6.1 lists the channel widths required to transmit data of various widths. If the reader experiences a feeling of déjà vu, it may be because these are identical to those at the end of the previous chapter, with only the captions altered.

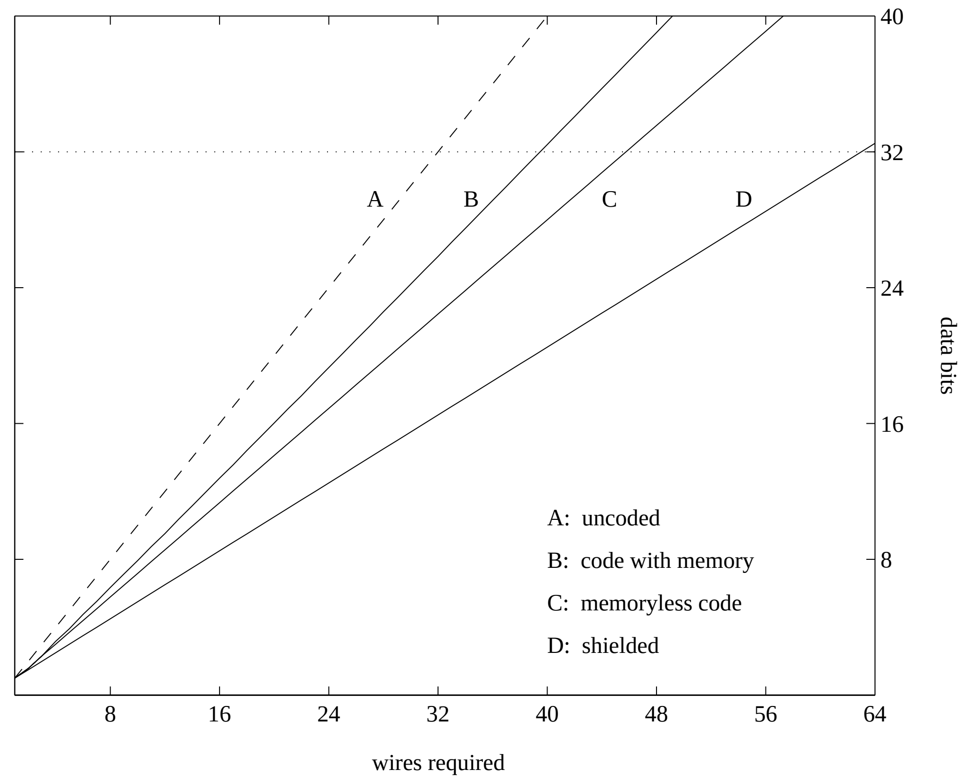


Figure 6.2 Performance of Codes.

<u>bits</u>	<u>wires required</u>			<u>bits</u>	<u>wires required</u>		
	memory	*memoryless*	*shielded*		*memory*	*memoryless*	*shielded*
1	1	1	1	17	22	25	33
2	3	3	3	18	23	26	35
3	4	4	5	19	24	28	37
4	6	6	7	20	25	29	39
5	7	7	9	21	27	30	41
6	8	9	11	22	28	32	43
7	9	10	13	23	29	33	45
8	11	12	15	24	30	35	47
9	12	13	17	25	31	36	49
10	13	15	19	26	33	38	51
11	14	16	21	27	34	39	53
12	16	17	23	28	35	41	55
13	17	19	25	29	36	42	57
14	18	20	27	30	38	43	59
15	19	22	29	31	39	45	61
16	20	23	31	32	40	46	63

Table 6.1 Performance of Codes.

Chapter 7

Implementation

7.1 Introduction

As was stated in the first chapter, the primary thrust of this work is theoretical. The intent was to determine the fundamental limits on the performance of self-shielding codes and to develop a mathematical framework for understanding them, and this goal was realized in the previous chapters.

Nevertheless, we will now venture into the realm of the practical, and take a brief look at some issues related to the logical and physical implementation of these codes. Circuit models for implementation of the encoder and decoder will be discussed. We will see how to implement a b bit bus when we can't afford to encode b bits all at once. Finally, an example design will be presented.

7.2 Encoder and Decoder Models

Although Chapter 3 presented some conceptual models for understanding the coding process, they wouldn't be especially useful in the implementation of a real encoder

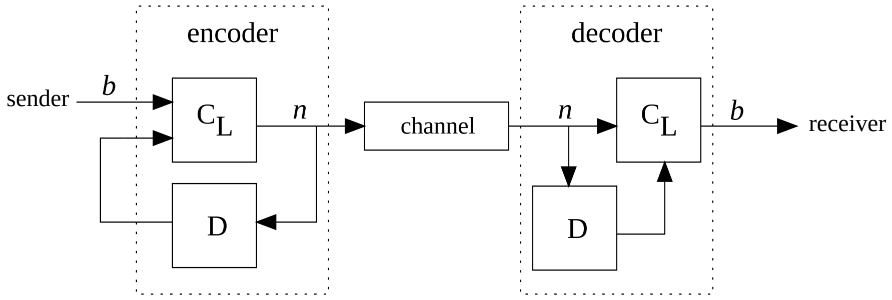


Figure 7.1 Unpipelined Circuit Model For Code With Memory.

or decoder. A circuit model that can be used to implement a code with memory is shown in Figure 7.1. The "C_L" blocks represent combinational logic, which may contain digital logic gates but no storage or timing elements. The "D" blocks represent memory elements; specifically, a bank of D-type flip-flops, whose outputs are equal to their inputs delayed by one clock cycle.

It is easy to see that the circuit model shown can implement a self-shielding code. The output of the first C_L block, which is the codeword placed on the channel, is a function of the previous codeword on the channel and the data word to be transmitted. In our conceptual model, the previous codeword on the channel selects the codebook, and the data word is mapped to a symbol, which selects the entry of the codebook to use. In the implementation, this process can be condensed into a single block of combinational logic with the proper inputs, as shown. The memory element is simply there to provide one of the necessary inputs, namely the previous codeword. The encoder shown in the figure is a case of a special class of synchronous circuits with feedback called *finite state machines*. Specifically, it is a *Mealy machine*, because the output changes combinationally with the input. Finite state machines are a well-studied class of circuits, and there are formalized methods and CAD tools to aid in their design.*

The output of the second C_L block, which is the decoded data word, is a function

* Although the quality of the methods and tools is questionable, especially in comparison to a good human designer. It is generally accepted that the "state-encoding problem" is a long way from being well-solved.

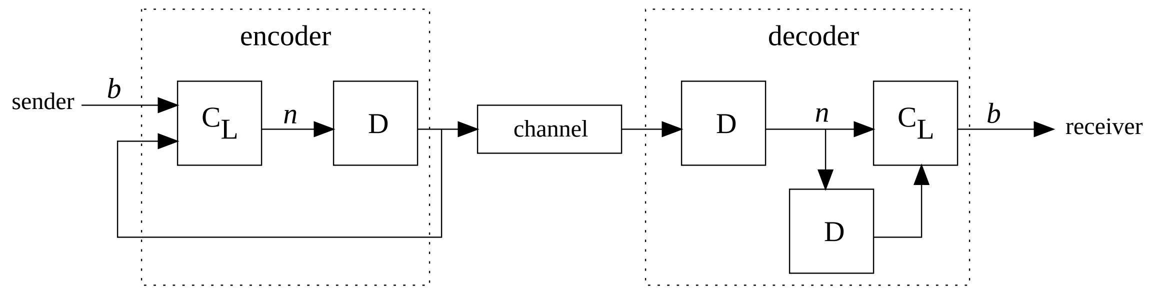


Figure 7.2 Pipelined Circuit Model For Code With Memory.

of the current codeword on the channel and the previous codeword on the channel. In the conceptual model, the previous codeword on the channel selects the codebook, and the current codeword on the channel selects the entry of the codebook that contains the desired symbol. This symbol is mapped to a data word. Again, this process is contained within a single block of combinational logic.

Although Figure 7.1 depicts a valid circuit model for the implementation of our code, it may not be a very useful one. The problem is that it is not *pipelined*. A combinational path exists from the input of the encoder to the output of the decoder, and in order for this to be a proper synchronous design, the propagation delay across this entire path must be less than one clock cycle. Given that the motivation for coding in the first place was to overcome a critical timing problem with the channel, it seems counter-productive to stack our encoder and decoder on this critical path.

Figure 7.2 shows a pipelined circuit model. Note that there is now a memory element immediately on either side of the channel. This gives the codeword almost the entire clock cycle to travel across the channel. There are now two delay elements in the forward path, so the data will arrive at the destination in three clock cycles instead of one. But in most situations, the throughput is much more important than latency, and we still have a throughput of one data word per clock cycle.

Notice that no additional memory element needed to be added to the encoder in order to pipeline it. Instead, the existing memory element was simply brought into the forward path. This changes the classification of this circuit from a Mealy machine to a *Moore machine*, because the output now only changes on clock edges.

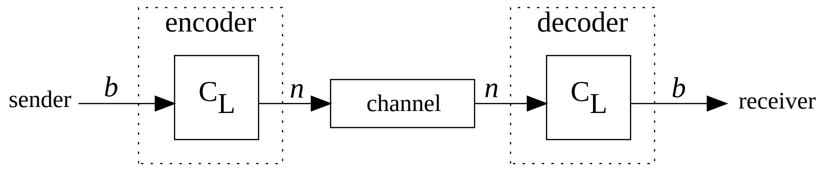


Figure 7.3 Unpipelined Circuit Model For Memoryless Code.

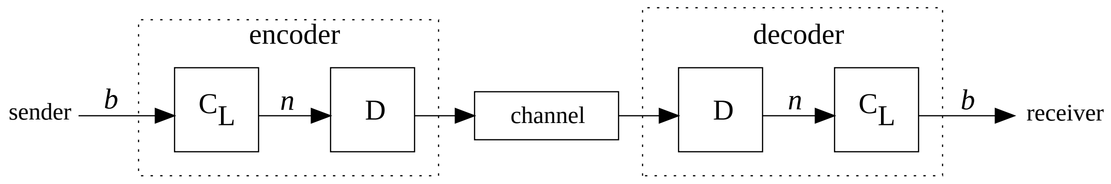


Figure 7.4 Pipelined Circuit Model For Memoryless Code.

With the decoder, no such trick is possible, and an additional memory element had to be inserted.

Memoryless codes are much easier to deal with. Figures 7.3 and 7.4 show a non-pipelined and pipelined circuit model for a memoryless encoder and decoder. The non-pipelined case is almost trivial. The first C_L block performs the encoding and the second the decoding. Because the code is memoryless, they need no inputs other than the current data word and codeword respectively. The pipelined case is similar, except memory elements were added on either side of the channel in order to shrink the critical path as much as possible. The latency and throughput are the same as in the pipelined coders with memory.

7.3 Partial Coding

The results of the previous chapters gave us the theoretical maximum performance for a code of the given width. However, like so many other theoretical maximums, we may have some trouble achieving it in practice. With a bus width b of 32 or 64 data bits, it may be infeasible or impractical to design a circuit to encode all b bits at once.

However, there is an easy solution, and that is to gently reintroduce the shielding concept. Instead of encoding b bits, the bus can be broken into sub-buses of smaller

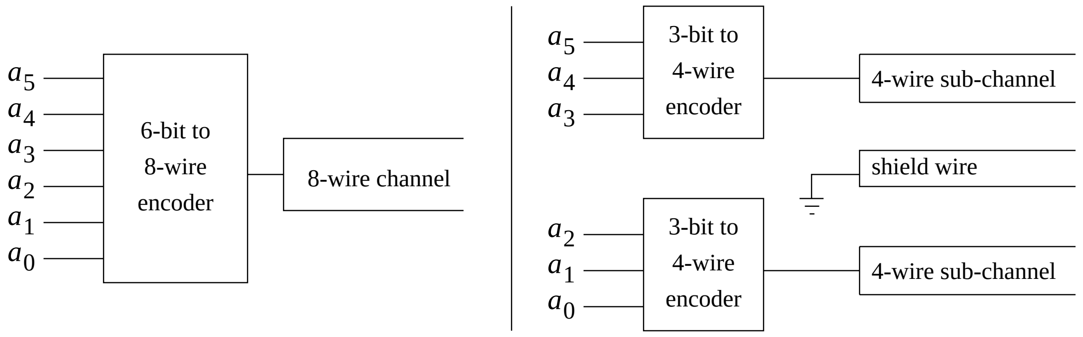


Figure 7.5 Breaking Bus and Channel into Sub-Buses and Sub-Channels.

width, which can then be encoded individually and output on sub-channels. Each sub-channel would have to be explicitly shielded from its neighbors with a dedicated ground wire. This is illustrated in Figure 7.5.

The reader may worry that these extra shield wires will degrade the performance of our code, and indeed they do somewhat. But they may be necessary for a reason unrelated to coding. In modern high-speed designs, many buses are routed with a ground wire between sections of data wires.* These wires are not used as crosstalk shields, but as current return paths, and serve to ensure that the loop created by current traveling across the bus is both small and predictable. This is important for a number of reasons, such as controlling the supply voltage droop during current spikes due to the finite resistance of the wires, and preventing inductive interference. The shield wires in our partial-coding scheme can thus serve the dual purpose of preventing crosstalk between sub-channels and creating a return path for bus current.

Figure 7.6 shows the number of wires required, including shield wires, for encoding a 32-bit bus when the individual sub-buses are no wider than a given number of bits. The most striking feature of this plot is the steep drop-off at the beginning. At $b = 3$, we have already gone from 63 wires to only 53. That is, an array of simple 3-bit encoders and decoders will immediately cut 10 wires from the channel. Another interesting feature is the shallow trail-off at the end. We see that, with either code

* Here, we use "ground" in the sense of AC-ground. That is, a low-impedance constant-voltage wire, typically one of the supply rails.

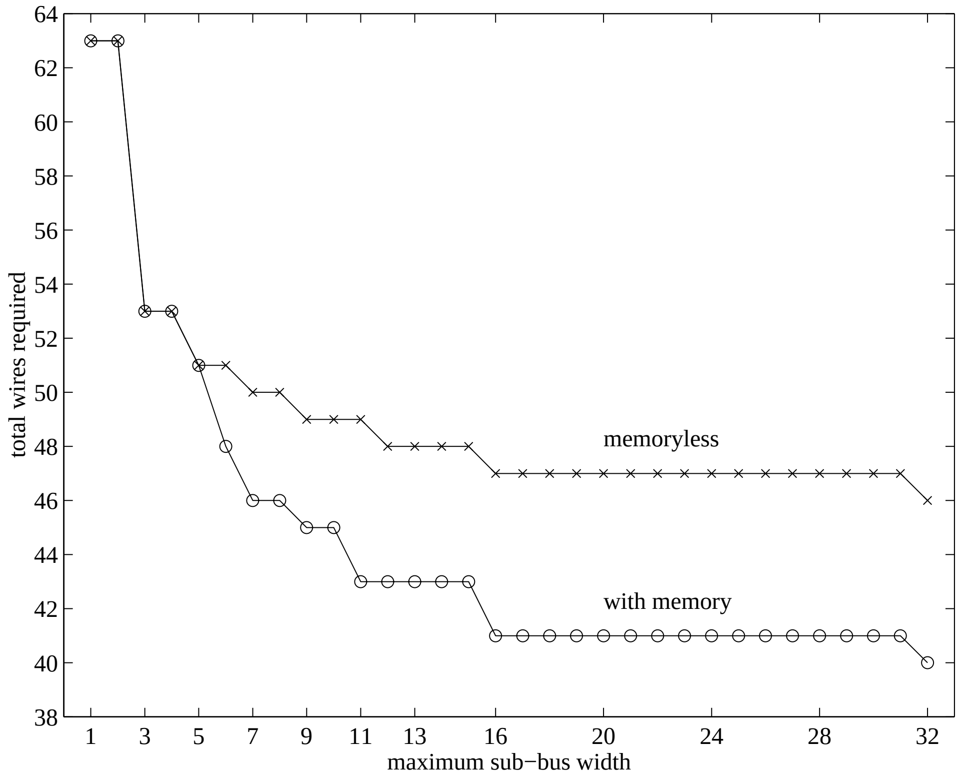


Figure 7.6 Wires Required for Partial Coding a 32-bit Bus.

type, dividing the bus in half and coding 16 bits at a time instead of 32 will cost only one extra channel wire.

Table 7.1 lists combinations of sub-bus widths that can be used to achieve the points on the convex hull of Figure 7.6. These are the combinations with the smallest maximum sub-bus width for the given performance. Studying Table 7.1 reveals a rather curious fact. Notice that no sub-buses of width 4 or 8 are listed. This is because, with both code types, a 3- or 7-bit sub-bus, followed by a shield wire and a 1-bit sub-bus, requires just as many wires as a 4- or 8-bit sub-bus respectively. This is unfortunate, because these unusable widths happen to be the ones that divide evenly into 32. Looking at the first few entries of the tables, we see that only 30 bits of the bus are effectively being encoded, with remaining two simply shielded individually. If our bus width were 30 instead of 32, the gains of partial coding over shielding would be even more impressive.

"$k_1(b_1) + k_2(b_2) + \ldots$" *means:*
k_1 *sub-buses of width* b_1, k_2 *sub-buses of width* b_2, *etc.*

with memory		memoryless	
53 wires:	$10(3) + 2(1)$	53 wires:	$10(3) + 2(1)$
51 wires:	$6(5) + 2(1)$	51 wires:	$6(5) + 2(1)$
48 wires:	$5(6) + 2(1)$	50 wires:	$4(7) + (3) + (1)$
46 wires:	$4(7) + (3) + (1)$	49 wires:	$3(9) + (5)$
45 wires:	$2(9) + 2(7)$	48 wires:	$2(12) + (7) + (1)$
43 wires:	$2(11) + (10)$	47 wires:	$2(16)$
41 wires:	$2(16)$	46 wires:	(32)
40 wires:	(32)		

Table 7.1 Sub-Bus Width Combinations and Wire Usage for $b = 32$.

7.4 Design Example

Figures 7.7 and 7.8 are gate-level schematic diagrams of circuits that implement a 3-bit to 4-wire memoryless encoder and decoder respectively. The mapping between data words and codewords is shown in Table 7.2. Notice that, indeed, the set of codewords used is the neighbor set of a class 1 codeword.

The code design featured in this example is optimal in the sense that no 3-bit memoryless encoder or decoder can be implemented with fewer two-input gates. After the rigor of the previous chapters, it may refresh the reader to learn that this code was designed through pure brute force. There are 8! or 40,320 ways of mapping the eight data words to the eight codewords in the class 1 clique. In an automated fashion, a circuit was designed for each of these mappings. Each circuit was fed into the SIS Boolean network manipulation package [11], which was instructed to simplify it as much as possible. Of the resulting circuits, 48 of them had five-gate encoders and four-gate decoders. These circuits were all of the same form as the circuit shown here, except with variables permuted or inverted.

Using the partial coding technique described in the previous section, we can encode a 32-bit bus using ten 3-bit sub-buses and two simply shielded wires. With

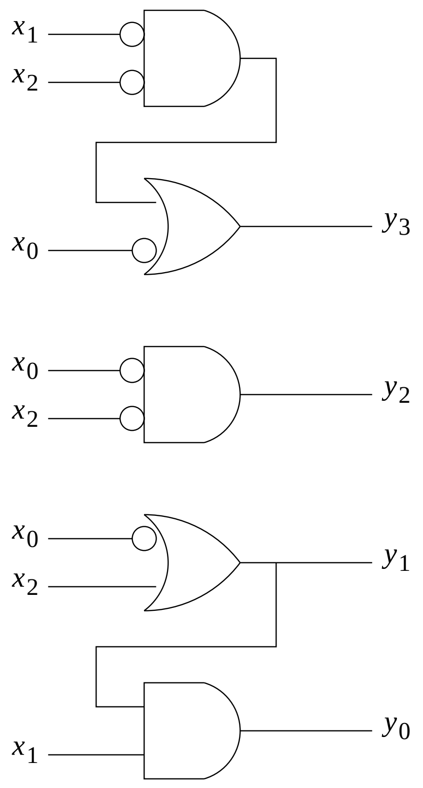


Figure 7.7 Encoder.

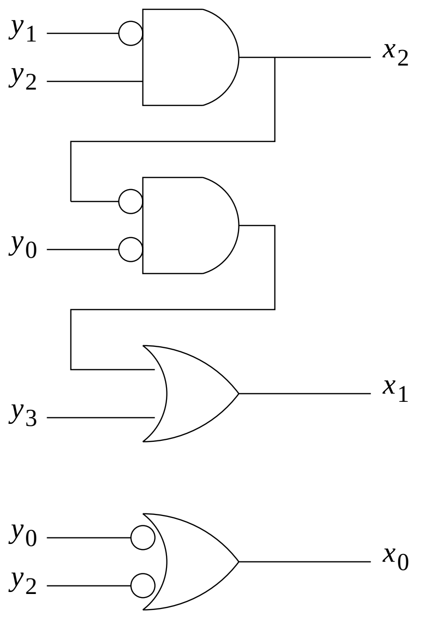


Figure 7.8 Decoder.

data word	codeword
000	0111
001	0001
010	1111
011	0000
100	0101
101	0100
110	1101
111	1100

Table 7.2 Data Word to Codeword Mapping for Design Example.

this design example, we see that the encoder would require 50 two-input gates, and the decoder 40 two-input gates. The channel would be 53 wires wide.

Unfortunately, code design through exhaustive search only works with $b = 3$. For a larger bus width, $2^b!$ is unfathomably huge, and the design of a good code would require human insight and possibly further theoretical research. Nevertheless, even this little 3-bit code is useful, given the partial coding technique. The ability to remove ten wires from a crosstalk-immune channel for the cost of a handful of gates is quite impressive, and not something that an IC designer should overlook.

Chapter 8

Conclusion

8.1 Summary

As device geometries shrink, chip sizes increase, and clock speeds get faster, interconnect delay is becoming increasingly significant. In particular, the propagation delay through long cross-chip buses is already proving to be a limiting factor in the speed of some designs, and this trend will only get worse. In such cases, it becomes necessary to reduce or prevent crosstalk delay along the bus.

This report has discussed a novel technique whereby crosstalk delay can be eliminated at a high level, with no need to worry about circuit tricks or technology-dependent issues. This technique is to encode the bus data onto a number of channel wires, which by design will never incur crosstalk delay.

Understanding the coding process and deriving the fundamental limits on performance required the development of "self-shielding code theory". The maximum performance obtainable, and the set of codewords required to achieve this performance, was derived for codes with and without memory. Some implementation issues were discussed, and an example design was presented.

8.2 Future Work

There is still work to be done, both theoretical and practical. The proof of Conjecture 5.1 will require a more sophisticated theory and understanding of the class pruning process. Perhaps, even the behavior of the optimal pruning algorithm can be described mathematically. The theory gives a set of codewords, but says nothing about how to map them to data words for the most efficient implementation. The ultimate usefulness of this technique will depend on how well the codes can be designed and implemented.

This report described the analysis of codes with and without memory. However, there is another type of code that was not discussed—codes that are *memoryless at the decoder*, but use memory at the encoder. It is possible to imagine a code that is described by a fixed, many-to-one mapping between codewords and data words. That is, each data word can be represented by a particular set of codewords. In such a case, the decoder needs no memory, because the mapping is fixed and the sets are mutually exclusive. However, we allow the encoder to use memory to determine, of the possible codewords that map to the desired data word, which one will obey the Fundamental Rule given the codeword already on the channel. The analysis of this type of code has proven so far to be extremely difficult, and the possible performance of such a code is unknown.

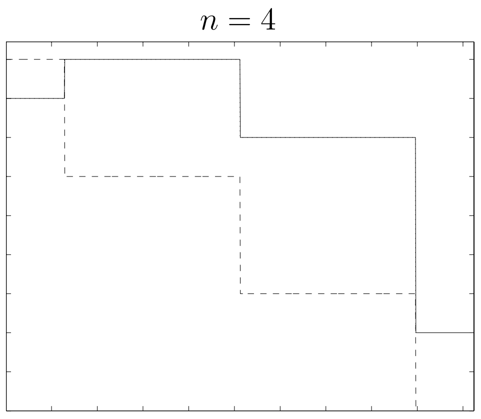
Finally, an interesting direction for future work is research into *hybrid codes*. If there is going to be an encoder and decoder around the channel, it might make sense to incorporate other desirable channel properties into the code. Codes could be designed that eliminate crosstalk delay as well as reduce average power consumption, prevent current spikes, perform error detection or correction, or accomplish some other task that is well-suited to bus encoding.

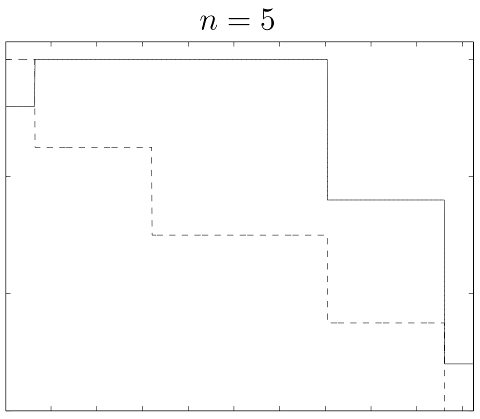
Hopefully, the theory and associated mathematical framework that was developed in this report will aid future designers who attempt to design self-shielding codes.

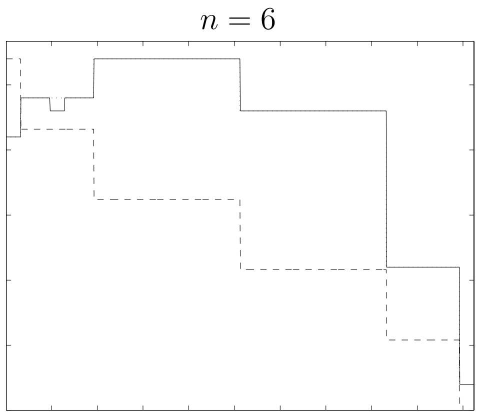
Appendix A

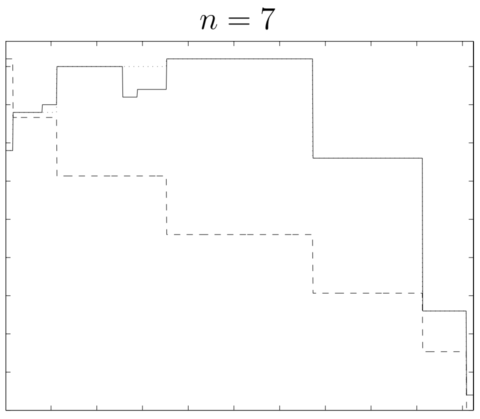
Pruning Curves

These are plots of minimum degree versus codewords pruned for the two pruning algorithms. The solid line is from the optimal algorithm, and the dotted line is from the class pruning algorithm. The "descending staircase"-type line represents the class of the codewords being pruned in the optimal algorithm, with class 1 at the top of the plot and class n at the bottom. This line is dashed for $n < 10$, and solid for $n \geq 10$.

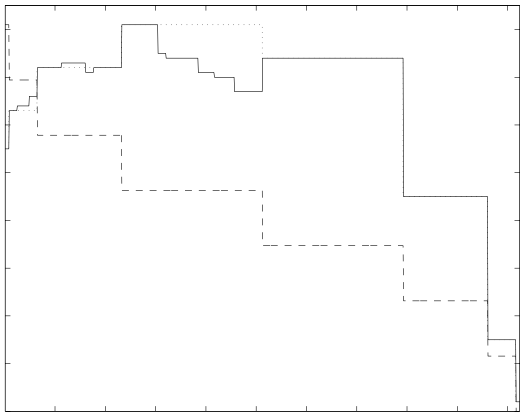




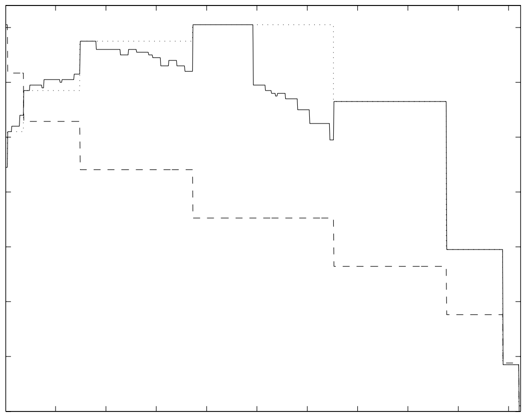




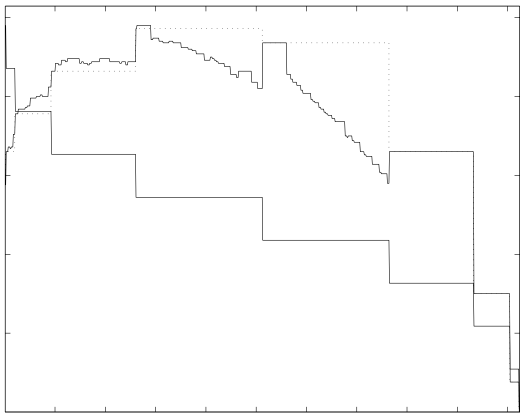
$n = 8$



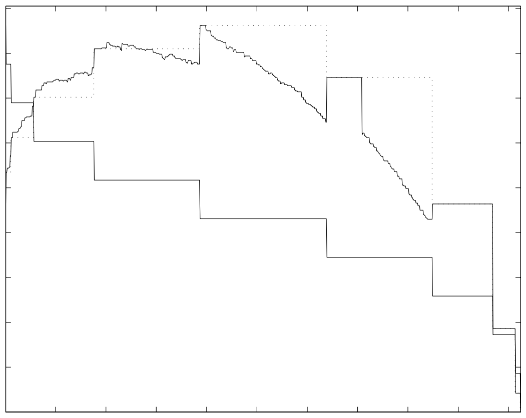
$n = 9$



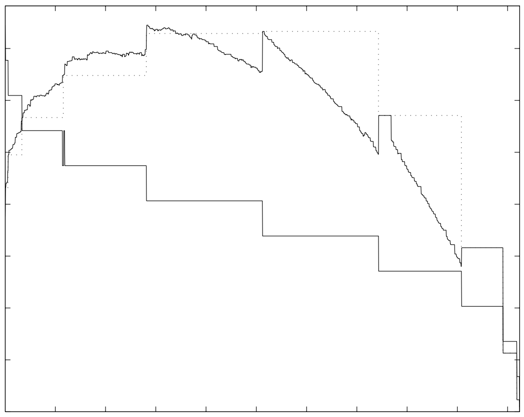
$n = 10$



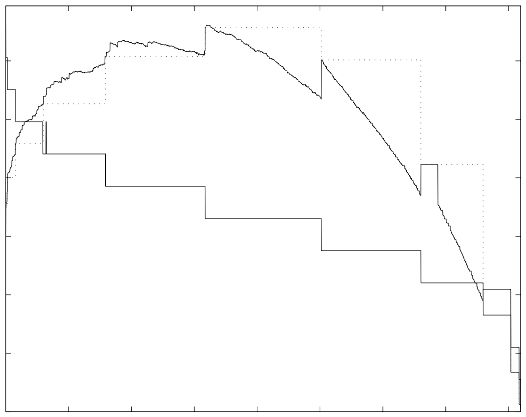
$n = 11$



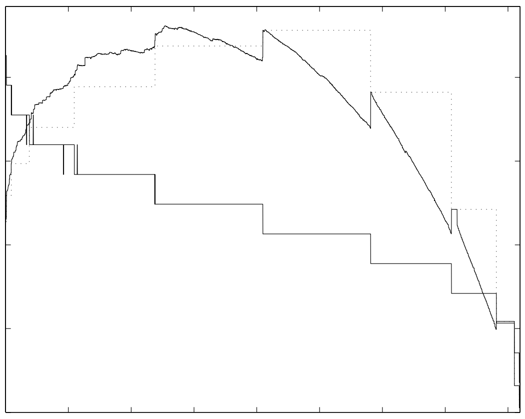
$n = 12$



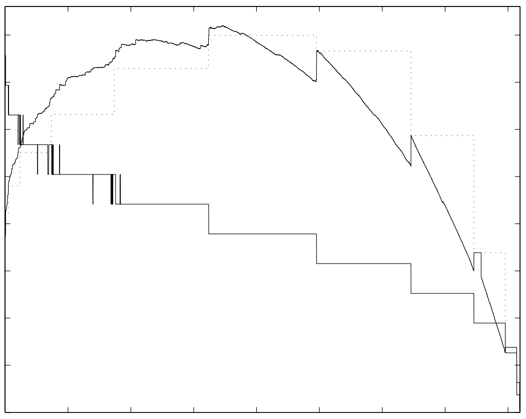
$n = 13$



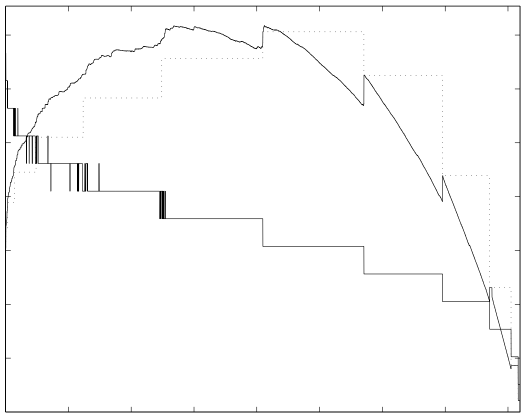
$n = 14$



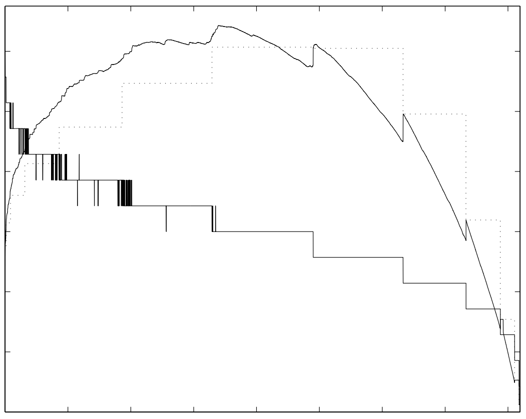
$n = 15$



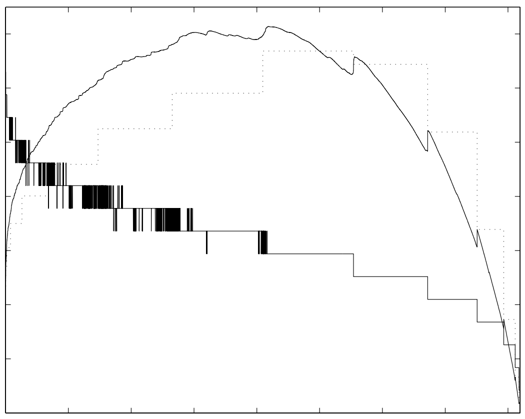
$n = 16$



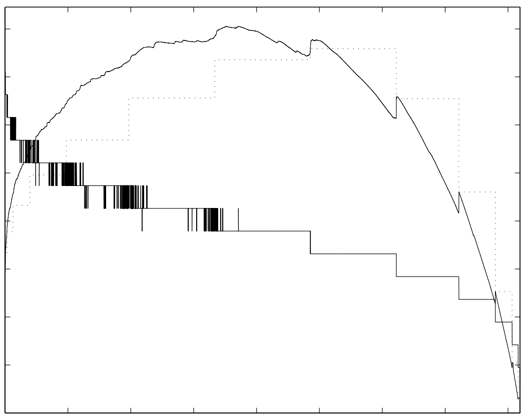
$n = 17$



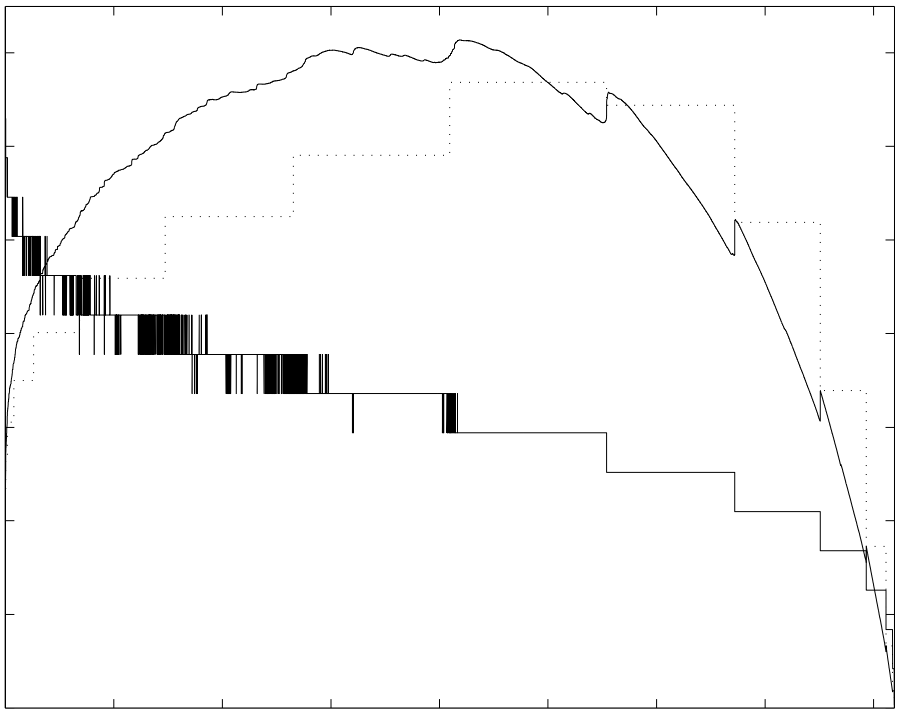
$n = 18$



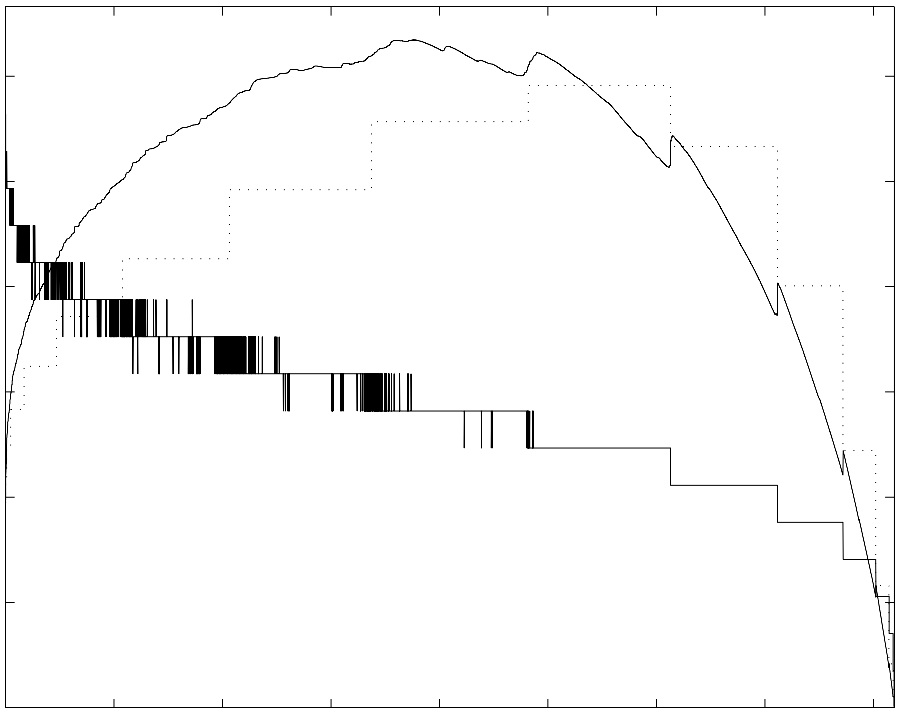
$n = 19$



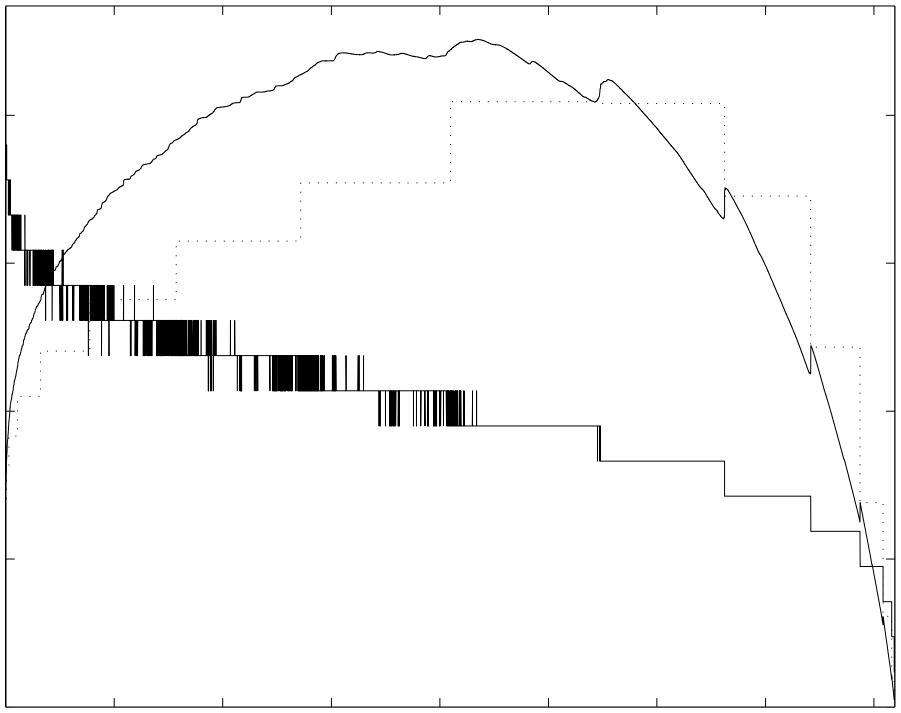
$n = 20$



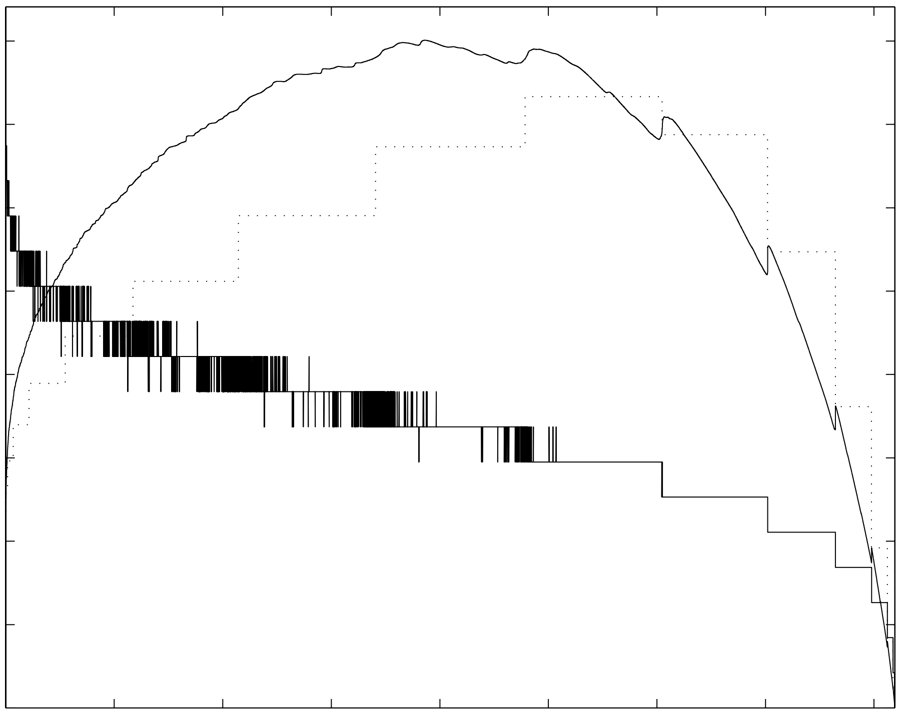
$n = 21$



$n = 22$



$n = 23$



Appendix B

Table of $P_n(x)$

$$P_0(x) = 0$$

$$P_1(x) = 1$$

$$P_2(x) = 1$$

$$P_3(x) = 1 + x^2$$

$$P_4(x) = 1 + 2x^2$$

$$P_5(x) = 1 + 3x^2 + x^4$$

$$P_6(x) = 1 + 4x^2 + 3x^4$$

$$P_7(x) = 1 + 5x^2 + 6x^4 + x^6$$

$$P_8(x) = 1 + 6x^2 + 10x^4 + 4x^6$$

$$P_9(x) = 1 + 7x^2 + 15x^4 + 10x^6 + x^8$$

$$P_{10}(x) = 1 + 8x^2 + 21x^4 + 20x^6 + 5x^8$$

$$P_{11}(x) = 1 + 9x^2 + 28x^4 + 35x^6 + 15x^8 + x^{10}$$

$$P_{12}(x) = 1 + 10x^2 + 36x^4 + 56x^6 + 35x^8 + 6x^{10}$$

$$P_{13}(x) = 1 + 11x^2 + 45x^4 + 84x^6 + 70x^8 + 21x^{10} + x^{12}$$

$$P_{14}(x) = 1 + 12x^2 + 55x^4 + 120x^6 + 126x^8 + 56x^{10} + 7x^{12}$$

$$P_{15}(x) = 1 + 13x^2 + 66x^4 + 165x^6 + 210x^8 + 126x^{10} + 28x^{12} + x^{14}$$

Appendix C

Table of $M_n(x)$ and Products

$$M_1(x) = \begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}$$

$$M_2(x) = \begin{bmatrix} x & 1+x^2 \\ x^2 & 2x \end{bmatrix}$$

$$M_3(x) = \begin{bmatrix} x+x^3 & 1+2x^2 \\ 2x^2 & 2x+x^3 \end{bmatrix}$$

$$M_4(x) = \begin{bmatrix} x+2x^3 & 1+3x^2+x^4 \\ 2x^2+x^4 & 2x+3x^3 \end{bmatrix}$$

$$M_5(x) = \begin{bmatrix} x+3x^3+x^5 & 1+4x^2+3x^4 \\ 2x^2+3x^4 & 2x+5x^3+x^5 \end{bmatrix}$$

$$M_6(x) = \begin{bmatrix} x+4x^3+3x^5 & 1+5x^2+6x^4+x^6 \\ 2x^2+5x^4+x^6 & 2x+7x^3+4x^5 \end{bmatrix}$$

$$M_7(x) = \begin{bmatrix} x+5x^3+6x^5+x^7 & 1+6x^2+10x^4+4x^6 \\ 2x^2+7x^4+4x^6 & 2x+9x^3+9x^5+x^7 \end{bmatrix}$$

$$M_8(x) = \begin{bmatrix} x+6x^3+10x^5+4x^7 & 1+7x^2+15x^4+10x^6+x^8 \\ 2x^2+9x^4+9x^6+x^8 & 2x+11x^3+16x^5+5x^7 \end{bmatrix}$$

$$M_9(x) = \begin{bmatrix} x+7x^3+15x^5+10x^7+x^9 & 1+8x^2+21x^4+20x^6+5x^8 \\ 2x^2+11x^4+16x^6+5x^8 & 2x+13x^3+25x^5+14x^7+x^9 \end{bmatrix}$$

$$M_{10}(x) = \begin{bmatrix} x+8x^3+21x^5+20x^7+5x^9 & 1+9x^2+28x^4+35x^6+15x^8+x^{10} \\ 2x^2+13x^4+25x^6+14x^8+x^{10} & 2x+15x^3+36x^5+30x^7+6x^9 \end{bmatrix}$$

$$M_1^1(x) = \begin{bmatrix} x & 1 \\ 1 & x \end{bmatrix}$$

$$M_1^2(x) = \begin{bmatrix} 1+x^2 & 2x \\ 2x & 1+x^2 \end{bmatrix}$$

$$M_1^3(x) = \begin{bmatrix} 3x+x^3 & 1+3x^2 \\ 1+3x^2 & 3x+x^3 \end{bmatrix}$$

$$M_1^4(x) = \begin{bmatrix} 1+6x^2+x^4 & 4x+4x^3 \\ 4x+4x^3 & 1+6x^2+x^4 \end{bmatrix}$$

$$M_1^5(x) = \begin{bmatrix} 1+10x^2+5x^4 & 5x+10x^3+x^5 \\ 5x+10x^3+x^5 & 1+10x^2+5x^4 \end{bmatrix}$$

$$M_2^1(x) = \begin{bmatrix} x & 1+x^2 \\ x^2 & 2x \end{bmatrix}$$

$$M_2^2(x) = \begin{bmatrix} 2x^2+x^4 & 3x+3x^3 \\ 3x^3 & 5x^2+x^4 \end{bmatrix}$$

$$M_2^3(x) = \begin{bmatrix} 5x^5+4x^5 & 8x^2+9x^4+x^6 \\ 8x^4+x^6 & 13x^3+5x^5 \end{bmatrix}$$

$$M_2^4(x) = \begin{bmatrix} 13x^4+13x^6+x^8 & 21x^3+27x^5+6x^7 \\ 21x^5+6x^7 & 34x^4+19x^6+x^8 \end{bmatrix}$$

$$M_2^5(x) = \begin{bmatrix} 34x^5+40x^7+7x^9 & 55x^4+80x^6+26x^8+x^{10} \\ 55x^6+25x^8+x^{10} & 89x^5+65x^7+8x^9 \end{bmatrix}$$

$$M_1(x)M_2(x) = \begin{bmatrix} 2x^2 & 3x+x^3 \\ x+x^3 & 1+3x^2 \end{bmatrix}$$

$$M_2(x)M_1(x) = \begin{bmatrix} 1+2x^2 & 2x+x^3 \\ 2x+x^3 & 3x^2 \end{bmatrix}$$

$$M_1(x)M_3(x) = \begin{bmatrix} 3x^2+x^4 & 3x+3x^3 \\ x+3x^3 & 1+4x^2+x^4 \end{bmatrix}$$

$$M_3(x)M_1(x) = \begin{bmatrix} 1+3x^2+x^4 & 2x+3x^3 \\ 2x+3x^3 & 4x^2+x^4 \end{bmatrix}$$

References

[1] C. E. Shannon. “A Mathematical Theory of Communication,” *The Bell System Technical Journal*, Vol. 27, pp. 379–423, 623–656, October, 1948.

[2] J. Rabaey. *Digital Integrated Circuts—A Design Perspective, Second Edition.* Prentice Hall, New Jersery, To Be Published.

[3] J. Rabaey. *Digital Integrated Circuts—A Design Perspective.* Prentice Hall, New Jersery, 1995.

[4] A. Vittala and M. Marek-Sadowska. “Crosstalk Reduction for VLSI,” *IEEE Transactions on Computer-Aided Design*, Vol. 7, pp. 392–96, March 1997.

[5] T. Gao and C. L. Liu. “Minimum Crosstalk Channel Routing,” *IEEE Transactions on Computer-Aided Design*, Vol. 15, pp. 465–74, 1996.

[6] T. Xue, E. Kuh, and D. Wang. “Post Global Routing Crosstalk Systhesis,” *IEEE Transactions on Computer-Aided Design*, Vol. 16, pp. 1418–30, 1997.

[7] J. Yim and C. Kyung. “Reducing Cross-Coupling among Interconnect Wires in Deep-Submicron Datapath Design,” *Proceedings. 1999 Design Automation Conference*, pp. 485–90, 1999.

[8] K. Hirose and H. Yusuura. “A Bus Delay Reduction Technique Considering Crosstalk,” *Proceedings. Design, Automation, and Test in Europe Conference and Exhibition 2000*, pp. 441–5, 2000.

[9] D. Li, A. Pua, P. Srivastava, and U. Ko. “A Repeater Optimization Methodology for Deep Sub-Micron, High-Performance Processors,” *Proceeding. International Conference on Computer Design, VLSI in Computers and Processors*, pp. 726–31, 1997.

[10] E. Weisstein. *CRC Concise Encyclopedia of Mathematics.* CRC Press, LLC, 1998.

[11] E.M. Sentovich, et al. “SIS: A System for Sequential Circuit Synthesis,” *Technical Report of the UC Berkeley Electronics Research Lab*, May 1992.

Colophon

About forty pages of this document were written in Microsoft Word. At that point, the author saw the light (or rather, turned away from the evil, evil darkness) and converted it all to TEX format. The remainder of the document was written in a Windows 95 port of GNU Emacs, and compiled with MiKTEX 2.0. It was typeset using Plain TEX and a "thesis" macro package that was pilfered from Brian Limketkai and mangled to suit the author's needs. Paragraphs were typeset using the included Computer Modern Roman fonts (CMR) at 12 point with 1.5 line spacing. Chapter and section headings were typeset using the Century Gothic font, converted from the Windows Truetype font collection. Figures were drawn in xfig 3.2 or TEX, and plots were generated with MATLAB 5.3.

The author plans to never again attempt to use a Microsoft product to write a technical paper.

Ah... it seems the student has become... the Master.