CS 118 Computer Networks: Data Link Error Detection
Second Data Link Lecture

In the last lecture we said that Data Link protocols offer five functions: the two required functions are *framing* and *error detection*. *Media access* is required for broadcast links, while *multiplexing* and *error recovery* are optional and can be found in some Data Links but not in others. We also studied framing techniques that were used to convert a stream of bits into a stream of frames. In this lecture we will study error detection which is accomplished by attaching redundant information like checksums to frames. Notice that error detection can only occur after framing has occurred because checksums are attached to frames — thus error detection is layered *above* framing in the sublayer model.

We will start by reviewing why we need error detection. We then study the simplest error detection schemes called parity bits and remark on a general property of error detecting codes called its Hamming Distance. Just as the Shannon Limit predicts the maximum bit rate of a medium in terms of its fundamental properties, the Hamming Distance predicts the maximum error detection and correction capabilities of an error detection scheme. Finally, we study CRCs, the most important and widely used error detection scheme used in networks today.

1 Why Error Detection

The main function of error detection is to make the probability of a receiving Data Link passing up an incorrect frame to the client layer very, very small. This is known as the *undetected error probability*; a possible goal for a system is one undetected error every 20 years. Recall that this is necessary because most transport protocols and user application programs ignore the end-to-end principle when it comes to the integrity of Data. End-to-end checksums are often not performed for performance reasons; hence it is good engineering practice to require that the probability of undetected errors on Data Links be low. Another reason, somewhat less important, is that intermediate routers may make decisions based on bogus data and misdirect packets. Perhaps in the future, if all workstations have hardware to do end-to-end checksums, the first reason will go away; however, the second reason will remain.

2 Kinds of Errors

There are two important types of errors that occur on transmission lines. Errors can occur because of intersymbol interference (because energy from a previous bit caused this bit to be wrongly interpreted) and noise (that changes a 0 level into a 1 level). One kind of noise that is always present is so-called thermal noise that can randomly corrupt bits. Other kinds of noise (due to sudden impulses) tend to generate bursts of bit errors that occur in groups. Burst errors often occur because of incorrect sychronization at the physical layers. For example, during the small

period a connector is being plugged in all the bits sent may be corrupted. The textbook gives other good examples.

Thus burst errors are errors that are highly correlated — if one bit has an error, its likely that the adjacent bits could also be corrupted. We define a burst error of size k to mean that the distance in bits from the first to the last error in the frame is at most $k-1$. The intermediate bits may or may not be corrupted.

Lets compare random and burst errors. Suppose we have a burst error of size 5 that starts in bit position 50. Then bits 50 and 54 are corrupted and bits 51 through 53 may or may not be corrupted. On the other hand we may have two random bit errors at positions 1 and 100. Thus burst errors tend to be more localized but allow for a larger number of actual bit errors; bit errors can be spread out arbitrarily but can be smaller. An ideal error detection code should be able to detect large localized burst errors and also be able to detect as many random bit errors as possible. Note that there is no contradiction in having a code that detects 8 bit burst errors (e. g.,, a clump of up to 8 consecutive bit errors) but can only detect 3 random bit errors (i.e., 3 randomly spaced bit errors).

Tannenbaum gives a nice example of frames of size 1000 bits and an overall error rate of 1 in 1000. If all errors were random, almost all frames would be corrupted. However, if errors come in bursts of size 1000 only 1 frame in a 1000 could be corrupted (assuming that the errors are really clusters of a 1000 corrupted bits, recall that the burst error definition does require only that the first and last bits be corrupted.)

3 Parity Bits

The idea of using redundancy to detect errors in a large sequence of data is a common one. Suppose you want to send a large string of numbers to a friend. If you add the sum of the numbers at the end, then if the friend makes a mistake in recording any one number, the actual sum of the numbers received will not match the received sum. We could call this a checksum, for a *sum* that helps us *check* the numbers. Notice that if the friend received two numbers in error, then all bets are off. One number may be increased by 2 and the second decreased by 2; the sum will not detect this change.

Now consider a string of binary bits. We could use the number of 1's in the string as a checksum but the checksum size now logarithmically with the number of data bits. (How many bit errors can this solution detect?) A cheaper solution is to add a single checksum bit that represents the parity (or more explicitly the "even"-arity) of the data": the extra parity bit is set to 1 if the number of 1's in the data bits is odd, and to 0 otherwise. A simple rule for calculating parity is take the Exclusive OR of all the data bits. (The EX-OR of two bits is 1 if and only if the two bits are not the same; you calculate the EX-OR of multiple bits two at a time.)

4 Hamming Distance

Recall that we want to take some arbitrary packets of data and add checksums to the data to form frames. By looking at the received checksums we hop to detect errors. A more general way to look at this is as a coding process, where an arbitrary packet is encoded to form a frame. At the receiver, the received frame is decoded to yield the packet. A simple way to code is to add a checksum to the original packet data. However, one could try other codes as well. For example, one could represent each 1 by three 1's and each 0 as three 0's. Thus 10 would be encoded as 111000. It is easy to see that this coding scheme allows you to detect any 2 bit error, because if all bits in any group of three do not match, we detect an error. However, this is a far more expensive code since we have to triple the number of bits sent. Also, if we have a checksum it is much easier to separate the data and the checksum. There are much better codes called CRCs that can detect many more bit errors at a much cheaper price (i.e., CRC-32 pays a fixed cost of 32 bits per packet.)

However, the general view allows us to abstract away from details of particular techniques like checksums. Let us the word *codewords* to denote the collection of all possible coded packets or frames. Define the Hamming distance between two codewords b and b' to be the number of corresponding bits that differ in b and b'. Thus if the two codewords differ in the 1st, 3rd, and 7th places, the Hamming Distance is 3. Since EX-OR returns 1 if and only if two bits differ, we can calculate Hamming Distance by taking the EX-OR of corresponding bits in the two code words.

What is the significance of the Hamming Distance? It represents the smallest number of errors needed for codeword b to be detected (incorrectly) as b' or vice versa. Define the Hamming Distance of a coding scheme to be the smallest Hamming Distance between any two codewords. For error detection, we want the Hamming Distance of the code to be as large as possible.

This is because if the Hamming Distance is d, it takes a d bit error for one valid codeword to be mistaken for another. An arbitrary (i.e., random) $d-1$ bit error will result in an invalid codeword that can be discarded. Thus a codeword with Hamming distance d can detect all $d-1$ bit errors; in other words to detect d bit errors, we need a code with distance of $d+1$. Parity codes have a Hamming Distance of 2 (since all codewords have an even number of 1's) and so can detect 1 bit errors.

In some sense this is analogous to the Physical Layer where we spaced the closest voltage levels to equal the maximum noise amplitude. Here we space the 2 closest code words equal to the maximum number of *random* bit errors we wish to tolerate. (Hamming distance only gives insight into random error detection properties, not about burst error detection.)

Notice that all error detection does is to detect that *some* bit has been corrupted in the frame, upon which the receiver can drop the frame. Suppose, however, we could tell that the 1st and 31st bit had been corrupted. Then the receiver could flip the 1st and 31st bits to recover the original data. This is called *error correction.* The difference between error correction and detection is that in the former the receiver knows the specific bits that are in error so it can correct them.

If the Hamming distance of a code is $2d+1$, then each codeword C can be allocated a set of codewords that are at distance d from the C. If any codeword allocated to C is received by the receiver, the receiver can assume that C was sent. Note that a space of $2d+1$ between codewords C and C' allows all codewords at a distance of d from C to be allocated to C, and all codewords

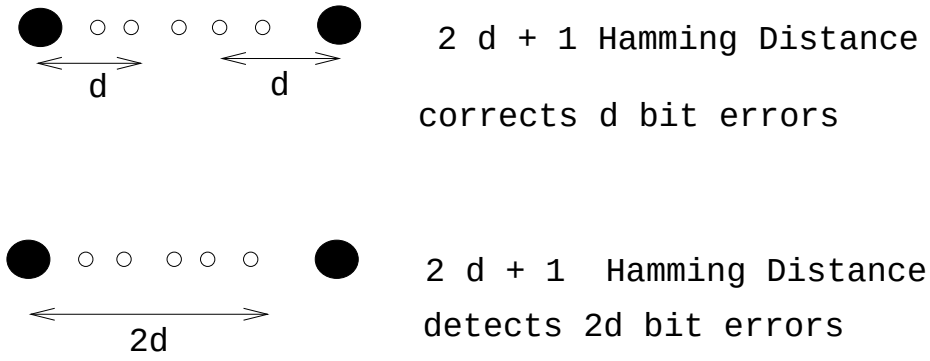

Figure 1: Hamming Distance and Error Correction

at a distance of d from C' to be allocated to C' (see Figure 1). Thus a $2d+1$ distance code can correct up to d bit errors.

Notice that one has to choose between correction or detecting a larger number of bit errors. For example, Figure refham shows that a $2d+1$ distance can also be used to *detect* $2d$ errors; alternately, the code can be used to *correct* d errors. However, the code cannot guarantee both at the same time; the designer must choose which of the two she wants to do.

Thus the distance 3 code we described earlier can even correct 1 bit errors. There is a much cheaper code, called a Hamming Code, that also detects 1 bit errors using approximately $\log P$ extra bits, where P is the number of data bits. The essential idea (which is well-explained in Section 4.2.1 in the textbook) is to use approximately $\log P$ parity bits. However, each data bit is covered by a different set of parity bits. By seeing which parity bits fail, we can pinpoint the bit that was corrupted. Roughly, if the position of a bit is N, we write down N in binary. Bit b then contributes to each parity bit for which there is a 1 in the binary expansion of N. Thus since each position has a unique binary expansion, each bit contributes to a unique set of parity bits. Thus if a bit is in error, a unique subset of the parity bits will be in error.

5 CRCs

Most network applications do not use error correcting codes. This is because they are expensive, and they can only correct a few bits of errors, and hence are not much help with (more common) burst errors. Rather than increase the number of bits transmitted with each data packet, network applications prefer to detect errors at the receiver and have the sender retransmit the correct bits. Many of today's fiber optic networks have very small error rates, and most packets are lost due to lack of buffering in routers (congestion loss); error correcting codes that correct a small number of bit errors do not help with the latter kind of error. (However, many magnetic disks use error correcting codes because they cannot depend on retransmission for error correction.)

So for the rest of this lecture, we deal with error detection. Hamming distance calculations say that we want a $d+1$ distance code to detect d bit errors. While this is correct, it is not the whole story. First, note that this says nothing about the ability of the code to catch burst errors, which

is very important. Second, the Hamming bound talks about the worst-case error detection. While it may be possible for some codeword to be corrupted into another valid codeword, it may be very unlikely to happen in practice if every bit has an equal chance of being corrupted.

Thus we would like codes or checksums that have good burst error detection properties and good random bit error detection properties. In the latter case, we want some worst-case figure that says the code will detect all (say) 4 bit errors. However, we also want to be able say that the code will detect most larger random bit errors with high probability. This is what is provided by so-called CRCs (for Cyclic Redundancy Checks), a sophisticated form of checksum that is used almost universally in real networks.

There is an interesting analogy between checksums (that have a high probability of detecting most random bit errors) and hash functions. A hash function is a function f that can be applied to a set of strings such that if we apply f to two random strings a and b, $f(a) \neq f(b)$ with high probability. Hash functions are used very commonly in computer science in order to lookup the state corresponding to arbitrary strings. If a random subset of strings hash to unique locations, hash lookup is very cheap.

Now consider the checksum to be a hash function for the data. We start with a valid codeword C and we use $f(C)$ as the checksum for C. If errors corrupt C to a random other string C', we want (with very high probability) that $f(C') \neq f(C)$, so we can detect the error. But this is what a good hash function should guarantee.

It is well known that summing up the digits is not a great hash function. There are better hash functions based on remainders. Before I actually explain CRCs to you, I will explain a hash function based on remainders using ordinary arithmetic. Once you see that, you will see that CRCs use the same idea, except with a funny arithmetic, Modulo 2 arithmetic.

5.1 Ordinary Division Checksum

Here is the ordinary division Checksum algorithm I invented some time back. Its not clear how useful it is in practice but it is useful for teaching purposes.

- Consider message M and checksum generator G to be binary integers.
- Let r be number of bits in G. We find the remainder t of $Y = 2^r M$ when divided by G. Why not just M divided by G. This allows us to easily separate checksum from message at receiver by looking at last r bits.
- Thus $2^r M = k.G + t$. Thus $2^r M + G - t = (k+1)G$. Thus we add a checksum $c = G - t$ to the shifted message and the result should divide G.

For example, if $X = 50$ and $G = 7$, $r = 3$, and $Y = 400$. Then $t = 1$ and $c = 6$ and the checksummed message becomes the binary string with value 406. Note that 406 is divisible by 7.

The ordinary division checksum has some reasonable properties that mimic those of CRC as long as the checksum is a prime number. However integer division is hard to implement, especially when dividing by a prime. It is easier to implement division using a funny arithmetic where carries

do not count. We call this Modulo 2 arithmetic and we study that next. CRCs are essentially the remainder when doing Modulo 2 arithmetic. To understand CRCs, you have to understand Mod 2 Arithmetic first.

5.2 Mod 2 Arithmetic and division

Mod 2 arithmetic has the following properties:

- There are no carries because both addition and subtraction is just EXCLUSIVE-OR. Repeated addition does not result in multiplication. e.g. $1100 + 1100 = 0$; $1100 + 1100 + 1100 = 1100$. Finally $1100 - 1000 = 1100 + 1000 = 0100$
- Multiplication is normal except for no carries: e.g. $1001 * 11 = 10010 + 1001 = 11011$. Multiplication by a power of 2 corresponds to a shift (as usual) but addition is EX-OR. Thus we use Shift and Ex-or instead of Shift and Add as in normal arithmetic.
- Division uses a similar algorithm to ordinary division.

The main idea beyond ordinary division is a simple iterative step. Suppose we know the remainder of X divided by G is r and we want to find the remainder of Xb, where b is a single bit. This is the same as finding the remainder of $2r + b$ when divided by G. Thus we can have a simple iterative algorithm to find the remainder of a a number X which can be represented as a binary string $b_1b_2, ...b_n$ by finding the remainders of b_1, b_1b_2, $b_1b_2b_3$ in turn, until we get the remainder of X. This is accomplished by storing the current remainder of the bits looked at so far, shifting in the next bit, and then subtracting G from the result to leave a number less than G. We then iterate this process till all bits have been shifted in.

Division in Mod 2 arithmetic is very similar except for two differences. First, there is no subtraction any more, only EX-OR. Second, in order to find a remainder, we want to reduce the current remainder after shifting in the new bit. The only way to reduce the value of a sequence of bits in this strange arithmetic is to remove the most significant bit or MSB (if it is 1). Assuming the CRC divisor always has a 1 in its MSB, this can be done by EX-ORing the CRC divisor with the current contents in the case that the MSB of the current contents is 1. (Ordinary division uses a similar idea; it subtracts the divisor from the current contents if the current contents is greater in magnitude than the current contents; only the definition of greater than is different in this funny arithmetic.)

5.3 CRCs – the idea

Thus in doing CRCs, we use essentially the same idea as in ordinary division checksums except that we use Mod 2 arithmetic.

- Again let r be number of bits in G. We find the remainder c of $2^{r-1}M$ when divided by G. We only shift message $r - 1$ bits this time because we are sure that the remainder will be only $r - 1$ bits in this case (unlike the ordinary division checksum)

- Thus $2^{r-1}M = k.G + t$. Thus $2^{r-1}M - t = k.G$. Thus $2^{r-1}M + t = k.G$ because addition is the same as subtraction.

For example, let $M = 11$ and $G = 101$. then $2^{r-1}M = 1100$.

```
        11
      -------------
101    1100
       101
      ----
        110
        101
        ---
         11
```

Notice that in the first step we try to find the remainder of 110 when divided by 101. Since the MSB of 110 is 1, we EX-OR 110 and 111 to get 11. Then we shift in the next bit which is 0. Calculating the remainder of 1100 then amounts to finding the remainder of 110. Once again, since the MSB is 1, we EX-OR 110 and 101 to get 11. Since 11 (or more precisely, 011) is less than 101 (since its MSB is 0), and we have no more bits to shift in, we stop at this point.

On the other hand, let $M = 11$ and $G = 111$. then $2^{r-1}M = 1100$.

```
        10
      -------------
111    1100
       111
      ----
        010
```

Notice that that after shifting in the fourth bit, we get 010 which is already "smaller" than 111, so we don't do another EX-OR but take what's left as a remainder. Another example is described in Figure 2 where $M = 110$ and $G = 111$. Try to do this yourself before comparing to the answer in Figure 2. Notice the dashed lines corresponding to where the message bits are brought down after each left shift.

The Tannenbaum book and the Peterson books have nicer longer examples.

Finally, the test for a good message remains the same. After adding the CRC to the message, the result should be divisible by the CRC divisor. Any messages that fail this test are discarded. An error corresponds to EX-ORing the message with bit string that has 1's in positions corresponding to corrupted bits.

5.4 CRC Properties and the Polynomial View

So what good does the CRC do? We have just explained how to compute CRCs by doing Mod 2 remainder arithmetic on bit strings. While this is a useful view for *computation*, another view

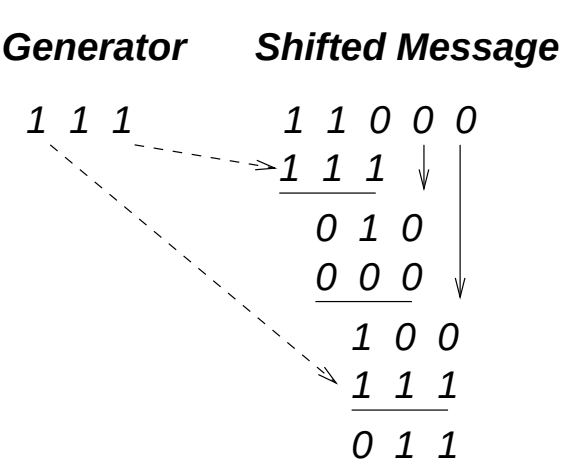

Figure 2: Another example of CRC. Make sure you try this yourself before checking the answer. You can always check answers by multiplying.

based on polynomials is better for *analysis*.

Consider two bit strings 101 and 011. We can represent them as two polynomials $x^2 + 1$ and $x + 1$. Notice that if the i-th bit position is a 1, then we have an x^i term in the polynomial. Now consider adding the two polynomials. If we add then using ordinary arithmetic, we get $x^2 + x + 2$. One nice thing about polynomial addition, is: *there are no carries*, which is similar to what we had. However using ordinary arithmetic leads to a term like 2 which cannot be represented by a single bit. But that's easily fixed: when you add, use Mod 2 arithmetic (EX-OR). So if you added the two polynomials using Mod 2 arithmetic you would get $x^2 + x$

Thus we can think of CRC computation as dividing a message polynomial (shifted by multiplying by x^{r-1}) by a CRC divisor polynomial (the generator) and adding in the remainder. While this view is equivalent to our older view, this view is easier to analyze.

CRCs are often written in terms of polynomials. For example, a common CRC is CRC-16: $x^{16} + x^{15} + x^2 + 1$. This corresponds to a CRC generator string of 11000000000000101. We can now analyze its error properties as follows:

First, notice that in the arithmetic view, having an error in the i-th bit corresponds to adding 1 mod 2 to the i-th bit (because this will flip the bit). Thus if we transmit 101 and have a bit error in the middle bit we get 111. But this is the same as adding (mod 2) 010 to the message 101. Thus the effect of errors can be represented by mod 2 addition of the message with another number with 1's in the bit positions where errors occurred. In the polynomial view, an error results in adding in a polynomial. Thus 101 was the polynomial $x^2 + 1$. After the middle bit is flipped we get $x^2 + X + 1$. But that is the same as adding x (mod 2) to the original polynomial. In general, an error in positions i, $j \ldots k$ is the same as adding in the polynomial $x^i + x^j + \ldots x^k$.

We use normal polynomial division intuition to see what happens. Recall that if an error is to be detected then the result of adding the error polynomial *should not be divisible* by the generator. Thus we now want to see why CRCs do well for both random and burst error models by showing that the error polynomials corresponding to these types of errors are not multiples of the CRC polynomial, for well chosen values of the CRC generator polynomial $G(x)$.

Single bit errors result in the addition of an x^i term. x^i will not divide $G(x)$ if the CRC polynomial $G(x)$ has at least two terms. To see this just think of your normal polynomial intuition: can you multiply a two term polynomial by something and get a one term polynomial? No way. The result will have at least two terms. In particular, it will have one distinct term corresponding to the product of the two greatest powers in the two factors, and one distinct term corresponding to the product of the two least powers.

Similarly, two bit errors correspond to adding $x^i + x^j$ which is the same as $x^i(1 + x^{j-i})$. This will not divide $G(x)$ if $G(x)$ does not divide $x^k + 1$ for sufficiently large k that is larger than the message length in bits. Most CRC polynomials like CRC-16 and CRC-32 are *chosen* to satisfy this property. This an important reason to choose special polynomials.

Finally, odd bit error polynomials are never divisible by $x + 1$. Suppose E is a polynomial caused by an odd number of bit errors. Then E has an odd number of terms. Thus if we substitute $x = 1$ in E, we will get 1. Now suppose E was divisible by $x + 1$. Then $E = Y(x + 1)$ for some polynomial Y. Then if we substitute $x = 1$ we would get $E = Y(1 + 1) = Y(0) = 0$. But we have just seen that we cannot get 0 when we substitute $x = 1$ in E. This contradiction can only be resolved if $x + 1$ does not divide E.

Using this observation, we can make the CRC catch all odd bit errors if $x + 1$ is a factor of G. If $G = Y(x + 1)$ for some Y, then clearly G cannot divide any polynomial that does not divide $x + 1$. Thus the CRC will catch all odd bit errors if G is a multiple of $x + 1$. (This is not as big a deal as it sounds because a single parity bit can detect all odd bit errors!).

Finally, burst errors of length k that start in position i of the message correspond to adding in the polynomial $x^i(x^{k-1} + ..1)$. If $k \leq$ degree of polynomial, we can catch all k bit burst errors. CRCs can detect a large amount of random errors too, at least most of the time (not guaranteed). You can look at the textbook if you want to see a more detailed analysis (but this is not required for the exam or HW). This is crucial in practice because it makes the undetected error rate using CRCs quite low and helps guarantee quasi-reliability.

Thus, to summarize: a good CRC should have at least two terms, should not divide $x^k + 1$ for sufficiently large k, and should have $x + 1$ as a factor. With these properties and a large length, it can do a good job on random and burst errors. CRC-16 and CRC-323 have these properties. Note that CRC-16 has $x + 1$ has a multiple and yet does not divide $x + 1$. This is not a contradiction! This is because the G for CRC-16 can be written as $Y(x + 1)$ and the Y does not divide $x + 1$.

5.5 Implementing CRCs

CRCs have to be implemented at a range of speeds from 1 Gbit/sec to slower rates. Higher speed implementations are typically done in hardware. First lets understand the simplest software implementation done one bit at a time.

Software Implementation: Basically our remainder algorithm consists of the following two steps that are iterated until all the message bits are consumed:

The current remainder is held in a register (that can hold r bits, where r is the number of bits in the divisor) which is initialized with the first r bits of the message.

- If the MSB of the current remainder is 1, then EXOR the current remainder with the divisor bits; if the MSB is 0, do nothing.
- Shift the current remainder register 1 bit to the left and shift in the next message bit.

Here is a naive hardware implementation corresponding to the software implementation.

Naive Hardware Implementation: CRCs have to be implemented at a range of speeds from 1 Gbit/sec to slower rates. Higher speed implementations are typically done in hardware. The simplest hardware implementation would mimic the above description and use a shift register that shifts in bits one at time. Each iteration requires three basic steps: checking the MSB, computing the Ex-Or, and then shifting. This is shown in Figure 3

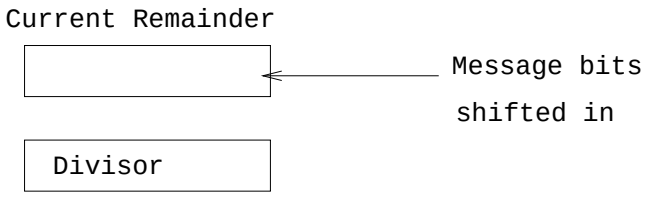

Figure 3: Naive hardware implementation requires 3 clock cycles per bit

A naive hardware implementation shown in the figure would require 3 clock cycles to shift in a bit; doing a comparison for the MSB in 1 cycle and the actual EXOR in another cycle, and the shift in the 3rd cycle. However, a cleverer implementation can be used to shift in one bit every clock cycle by *combining* the Test for MSB, the Ex-OR and the shift into a single operation.

Implementation using LFSRs: In Figure 4 the remainder R is stored as 5 separate 1-bit registers $R4$ through $R1$ instead of a single 5 bit register, assuming a 6 bit generator string. The idea makes use of the observation that the Ex-OR needs to be done only if the MSB is 1; thus in the process of shifting left the MSB, we can feed back the MSB to the appropriate bits of the remainder register. The remaining bits are EXORed during their shift to the left.

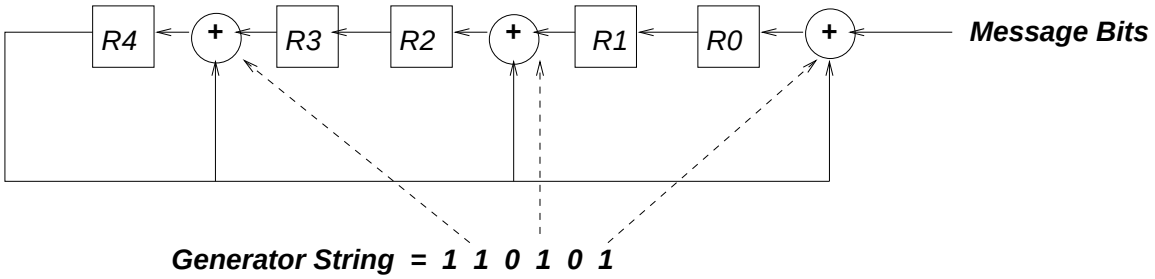

Figure 4: Linear Feedback Shift Register (LFSR) implementation of a CRC remainder calculation requires 1 clock cycle per bit. The Ex-ORs are combined with a shift by placing Ex-OR gates (the circles) to the right of some registers. Specifically, an Ex-OR gate is placed to the right of register i if bit i in the generator string (see dashed lines) is set. The only exception is Register $R5$ which does not need to be stored because it corresponds to the MSB which is always shifted out.

Notice that in Figure 4, an Ex-OR gate is placed to the right of register i if bit i in the

generator string (see dashed lines) is set. The reason for this rule is as follows. Compared to the simple iterative algorithm, the hardware of Figure 4 effectively combines the left shift of iteration J together with the MSB check and Ex-OR of iteration $J+1$. Thus the bit which will be in position i in Iteration $J+1$ is in position $i-1$ in Iteration J.

If this is grasped (and this requires mentally shifting one's mental pictures of iterations), the test for the MSB (i.e., bit 5) in Iteration $J+1$ amounts to checking MSB - 1 (i.e., bit 4 in $R4$) in Iteration J. If Bit 4 is 1, then an Ex-OR must be performed with the generator. For example, the generator string has a 1 in Bit 2, so $R2$ must be Ex-ORed with a 1 in Iteration $J+1$. But Bit 2 in iteration $J+1$ corresponds to Bit 1 in Iteration J. Thus the Ex-OR corresponding to $R2$ in Iteration $J+1$ can be achieved by placing an Ex-OR gate to the right of $R2$: the bit that will be placed in $R2$ is Ex-ORed during its transit from $R1$.

Notice that the check for MSB has been finessed in Figure 4 by using the output of $R4$ as an input to all the Ex-OR gates. The effect of this is that if the MSB of Iteration $J+1$ is 1 (recall this is in $R4$ during Iteration J), then all the Ex-ORs are performed. If not, and the MSB is 0, no Ex-ORs are done as desired; this is the same as Ex-ORing with zero in Figure 2.

The implementation of Figure 4 is called a linear feedback shift register (LFSR) for obvious reasons. This is a classical hardware building block which is also useful for the generation of random numbers for say Quality of Service. For example, random numbers using say the Tausworth implementation can be generated using 3 LFSRs and an Ex-OR.

Faster Implementations: The bottleneck in the implementation of Figure 4 is the shifting which is done 1 bit at a time. Even at one bit every clock cycle, this is very slow for fast links. Most logic on packets occurs after the bit stream arriving from the link has been deserialized[1] into wider words of say size W. Thus the packet processing logic is able to operate on W bits in a single clock cycle, which allows the hardware clock to run W times slower than the interarrival time between bits.

Faster CRC algorithms that compute remainders multiple bits at a time are possible and will sometimes be explored in the Homework of this course. These ideas can be applied to both software and hardware algorithms.

Thus to gain speed, CRC implementations have to shift W bits at a time, for $W > 1$. Suppose the current remainder is r and we shift in W more message bits whose value as a number is say n. Then in essence the implementation needs to find the remainder of $(2^W \cdot r + n)$ in one clock cycle.

If the number of bits in the current remainder register is small, the remainder of $2^W \cdot r$ can be precomputed for all r by table lookup. This is the basis of a number of software CRC implementations that shift in say 8 bits at a time. In hardware, it is faster and more space-efficient to use a matrix of XOR gates to do the same computation. The details of the parallel implementation can be found in a paper by Sarwate in Communications of the ACM.

[1] This is done in what is often called a SERDES chip for Serializer-Deserializer chip

6 Ways of Thinking

The following ways of thinking about things were used in this lecture and may be useful for you in other contexts:

- Arguing by Analogy: We used the analogy between hash functions and checksums to motivate trying to take remainders. We used the analogy between ordinary division checksums and CRCs to understand CRCs. Analogies are not just useful for understanding but also for design. For example, to compute CRCs multiple bits at a time, we can try to find remainders in ordinary arithmetic multiple digits at a time, and try to apply similar ideas to CRCs.
- Having Multiple Views: We used the bit string view to describe CRC computation and the polynomial view for analysis. We saw that adding a checksum could be looked on as the general process of coding. Multiple views of the same concept increase the number of ways you can attack a problem involving the concept. A familiar example is normal geometry and coordinate geometry.
- General and abstract approaches help: think of error detection in terms of coding (and not limiting ourselves to checksums) allows us to use simple yet powerful ideas like Hamming Distance. When we think in terms of checksums (as opposed to whole codewords) its much harder to motivate the idea of Hamming distance.