Applying Hamming Code to blocks of data

Chuck Benz

August 21, 2001

Hamming code affords a straightforward way to protect a block of data against single bit errors. It allows any single bit error to be detected and corrected. This discussion will explain Hamming code basics and discuss modifications and applications. The intended audience is digital logic designers/engineers – since the author is an engineer, mathematicians and computer scientists may cringe at some of the phrasing and terminology.

Hamming Code Basics

It's easiest for me to explain Hamming code by starting at the end – if there is a bit error in a block of data, check bits that accompany the data allow us to calculate the address of the bit error. That is because each check bit represents parity across ½ of the data – in a way that maps to a "bit address".

Let's consider a 63 bit block of data, numbering the bits like so (ignore for the moment that I've left out what would have been bit 0):

	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

The bit numbering gives us an address that easily points to a data bit. If this block of data is passed across an 8 bit bus, we can use address bits [2:0] to specify a bit on the bus, and address bits [5:3] to specify a cycle of data on the bus.

Consider calculating check bits that represent parity across all addresses that have a given bit set in their number (awkwardly phrased, but I think you'll understand the example):

```
Check[0] = 1 ^ 3 ^ 5 ^ 7 ^ 9 ^ 11 ^13 ^15 ^17 ^ 19 ^ 21 … ^ 61 ^ 63 ;
Check[1] = 2 ^ 3 ^ 6 ^ 7 ^ 10 ^ 11 ^ 14 ^ 15 … ^ 62 ^ 63 ;
Check[2] = 4 ^ 5 ^ 6 ^ 7 ^ 12 ^ 13 ^ 14 ^ 15 … ^ 62 ^ 63 ;
Check[3] = 8 ^ 9 ^ 10 ^ 11 ^ 12 ^ 13 ^14 ^ 15 … ^ 62 ^ 63 ;
Check[4] = 16 ^ 17 ^18 ^ 19 ^ 20 …  ^ 62 ^ 63 ;
Check[5] = 32 ^ 33 ^ 34 ^ 35 ^ 36 …  ^ 62 ^ 63 ;
```

Another view would be like so

```
reg [63:1] data ;
reg [5:0] address, check ;
int addrbit ;
for (addrbit = 0; addrbit < 6 ; addrbit ++) {
  for (address = 1 ; address < 64 ; address++) {
    if (address[addrbit])
      check[addrbit] = check[addrbit] ^ data[address] ;
  }
}
```

The idea is that the source of the data calculates the check bits, and sends them with the data. The destination repeats the calculation, and compares the received check bits with the calculation – if they are the same, all is assumed well, but if not, then by exclusive-or'ing them, a syndrome is generated which points at the errored bit. Example: if bit 49 was changed, then the destination would get different check results for check bits 5, 4, and 0 – those are the bits that include data bit 49 in their parity. So the syndrome would be 110001 – binary 49.

But how to send the check bits ? By placing them in appropriate spots in the data block, they are protected, too. Put check[0] in position 1, check[1] in position 2, check[2] in position 4, check[3] in position 8, [4] in 16, [5] in 32. If a bit error hits a check bit in one of those positions, the syndrome points directly at that position. Simple.

So, now look at the data as so:

	C0	C1	3	C2	5	6	7
C3	9	10	11	12	13	14	15
C4	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
C5	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

With 63 bits, we have 6 check bits, and 57 bits for our real data. For each additional check bit, we get another 2^n-1 bits for real data (n being the current number of check bits). A 7th bit of data gives us 63 more data bits.

Applying Hamming code to a real block of data

It's not necessarily helpful to use Hamming code in the way it appears above – data would have to be shifted around, and check bits have to be calculated and inserted afterwards.

One alternative would be to move through the data backwards – numbering the bits in reverse order:

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
C5	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
C4	17	18	19	20	21	22	23
C3	9	10	11	12	13	14	15
	C0	C1	3	C2	5	6	7

In this way, all data that contributes to C5 is passed on the bus before C5 is inserted, and the same is true for C4, C3, C2, C1, and C0.

Another approach is to skip some data, and make the check bit positions virtual. Data that is skipped is assumed to be 0.

24	25	26	27	28	29	30	31
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63
C0	C1	C2	C3	C4	C5		

In this example, 32 bits are protected by adding 6 check bits. Or 24 bits could be protected by 5 check bits – with just 40-63, C5 is redundant as it will indicate the range 32-63 for any single bit error.

Improving protection – double bit error detection

If two errors occur, Hamming code still produces an error syndrome, but it points to an uninvolved bit. Bit errors in positions 41 (101001) and 56 (111000) together would appear to point to position 17 (010001).

This is because the Hamming distance between 2 code points (the data and check bits combined are a "code point") is enough that single bit errors can be corrected, but double bit errors appear to be a single bit error on a different code point.

Consider 2 code points, A and B (think of A as binary 0, B as binary 1). If we use 1 bit to represent them, then a single bit error changes the value from one code point to the other. If we use two bits (say, 00 and 11), then a single bit error would be detected, but it wouldn't be clear what the original value was. With three bits for the codes (000/111), a single bit error still leaves enough information to reveal the original code (so it can be corrected) – but a double bit error looks like a single bit error on the other code.

With 4 bits for the codes (0000/1111), a single bit error can be corrected, and a double bit error can be detected (it doesn't look like a single bit error).

The term Hamming distance can be understood to refer to the number of bits that have to change to move from one code point to another. A Hamming distance of 1 would offer no protection, 2 allows single bit detection, 3 offers single bit correction, 4 offers single bit correction and double bit detection, 5 allows single and double bit correction, and so forth.

Great – how can we extend Hamming code to offer double bit error detection ? I'll describe one way, which is more brute force, rather than mathematically developed.

Adding one more check bit that represents parity across all data and check bits will separate single bit errors from double bit errors – if this check bit is calculated to match the received value, then we either have two bit errors, or none (or some other even number of errors, but we're not considering those cases). If the syndrome is 0, then data has no errors; a non-zero result indicates a double bit error. What about a single bit error in this added check bit ? We can recognize that case because the syndrome is 0. And a double bit error with one errored bit being this added check bit will still appear as other double error cases – a non-zero syndrome, but the appearance of a match in this new check bit.

References, acknowledgements:
My understanding of Hamming code came from a discussion with Jeff Parker. If/when I find an original reference, I'll add it to this document. The idea of extending coverage to double bit detection was something I arrived at myself, but I'm sure that it's been done before, and I wouldn't be surprised to find out that it's in the original reference. (Initially, I'd worked out a clumsier extension to double bit detection involving just a parity of the check bits, but the simpler way above occurred to me as I wrote this out).

Addendum, OCT-2002: One reference recommended to me is "Error Control Systems" by Steven Wicker (1995). Also, I have seen, but never read, "Error Control Coding" by Shu Lin and Daniel Costello (1983). Readers have suggested that I mention some other types of error correction codes: BCH codes can correct double bit errors (adjacent only?), and Turbo Codes are a recent development of interest (in the 90's, so they are not covered in those texts). I'm not yet familiar with these codes.