The Mathematics of Chipkill ECC.txt -- 9k

The Mathematics of Chipkill ECC

OVERVIEW

Chipkill ECC is an advanced form of computer memory error checking and correction that corrects not only single bit errors but many multi-bit errors up to 4 bits long. Originally, the chipkill circuitry was located on the memory module itself, which prevented the use of standard memory modules. Later, the chipkill circuitry was placed either on the northbridge chipset of the computer's motherboard or within the CPU. This allowed the use of standard SDRAM (for example DDR2 or DDR3) memory modules.

Chipkill operates on 4-bit nibbles (1/2 bytes), which, in the parlance of chipkill, are called "symbols". If just one symbol, i.e. nibble, is erroneous, chipkill can fully correct it -- all four bits if necessary. But if more than one symbol is erroneous, chipkill can only detect it. As a consequence of its operating on nibbles, chipkill is most effective if used with memory modules built with x4 (4-bit width) chips because each symbol corresponds to a chip. In that case, multi-bit errors will be isolated to a single chip or symbol because of the physical distance between the chips. Contrary to many posts on the internet, chipkill will also work just fine with memory modules built with x8 (8-bit width) chips. It's less effective though with x8 chips because a multi-bit error could then straddle a nibble boundary, thus causing a double symbol error that would only be detected.

Also, contrary to several posts on the internet, chipkill has nothing whatsoever to do with registered memory. It makes no difference whether the memory modules used with chipkill are registered or unbuffered. Unfortunately for desktop motherboards though, x4 memory modules only seem to be available in the registered version.

MATHEMATICS

I recently surfed the Internet in at attempt to find out more about the mathematics of how Chipkill ECC works. I didn't discover much, but I found out that it involves Galois Field mathematics! Omigosh! That's really deep, obscure, arcane, and recondite!! Nevermind complicated!! I could never understand that! But actually it turns out to be really really simple. All you need to know is the Galois multiplication table from 0 to 15! Here it is:

Multiplicand

Multiplier	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
2	0	2	4	6	8	a	c	e	3	1	7	5	b	9	f	d
3	0	3	6	5	c	f	a	9	b	8	d	e	7	4	1	2
4	0	4	8	c	3	7	b	f	6	2	e	a	5	1	d	9
5	0	5	a	f	7	2	d	8	e	b	4	1	9	c	3	6
6	0	6	c	a	b	d	7	1	5	3	9	f	e	8	2	4

```
7       0 7 e 9 f 8 1 6 d a 3 4 2 5 c b
8       0 8 3 b 6 e 5 d c 4 f 7 a 2 9 1
9       0 9 1 8 2 b 3 a 4 d 5 c 6 f 7 e
a       0 a 7 d e 4 9 3 f 5 8 2 1 b 6 c
b       0 b 5 e a 1 f 4 7 c 2 9 d 6 8 3
c       0 c b 7 5 9 e 2 a 6 1 d f 3 4 8
d       0 d 9 4 1 c 8 5 2 f b 6 3 e a 7
e       0 e f 1 d 3 2 c 9 7 6 8 4 a b 5
f      0 f d 2 9 6 4 b 1 e c 3 8 7 5 a
```

That's it! We'll do a little division later, but that's just multiplication by the inverse -- you find the number to multiply the divisor by so that the product is 1 and multiply by that number. For example, dividing by 9 is the same as multiplying by 2, as you can verify from the table. When memory is organized for Chipkill, numbers are read in 128 bits at a time, along with 16 check bits, making a total of 144 bits for each number. Chipkill views each number as consisting of 32 4-bit "nibbles", which I'll call N0 through N31. Chipkill also divides the 16 check bits into 4-bit nibbles, which I'll label C0 through C3. When the computer stores a number in memory, it calculates values for the check nibbles and stores them along with the number. The check nibble calculation is long, but not complicated:

C0 = N0 + 2*N1 + 3*N2 + 4*N3 + 5*N4 + 6*N5 + 7*N6 + 8*N7 + 9*N8 + a*N9 + b*N10 + c*N11 + d*N12 + e*N13 + f*N14 + N15 + 2*N16 + 3*N17 + 4*N18 + 5*N19 + 6*N20 + 7*N21 + 8*N22 + 9*N23 + a*N24 + b*N25 + c*N26 + d*N27 + e*N28 + f*N29 + N31

C1 = N0 + N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8 + N9 + N10 + N11 + N12 + N13 + N14 + N30 + N31

C2 = N15 + N16 + N17 + N18 + N19 + N20 + N21 + N22 + N23 + N24 + N25 + N26 + N27 + N28 + N29 + N30 + N31

C3 = N0 + 9*N1 + e*N2 + d*N3 + b*N4 + 7*N5 + 6*N6 + f*N7 + 2*N8 + c*N9 + 5*N10 + a*N11 + 4*N12 + 3*N13 + 8*N14 + N15 + 9*N16 + e*N17 + d*N18 + b*N19 + 7*N20 + 6*N21 + f*N22 + 2*N23 + c*N24 + 5*N25 + a*N26 + 4*N27 + 3*N28 + 8*N29 + N30

The '*' symbols in the above equations stand for Galois Field multiplication (use the table!), and the '+' signs mean XOR (exclusive or). When the computer reads a number from memory, it runs through the same calculation again on the number it has read, and produces another set of check nibbles that I'll call C0' through C3' ("C0 prime" through "C3 prime"). Then it compares the two sets of check nibbles by combining them into a set of nibbles called a "syndrome" in the terminology of Chipkill. The syndrome nibbles are labeled S0 through S3:

S0 = C0 + C0'
S1 = C1 + C1'
S2 = C2 + C2'
S3 = C3 + C3'

Again, the '+' signs mean XOR. Now the computer looks at the syndrome nibbles to see whether there's an error.

First, notice that if there's no error, each check nibble will be

the same as its primed counterpart, causing the XOR of each check nibble with its primed counterpart to be zero. Consequently, S0 through S3 will all be zero.

Let's see what happens if just one of the nibbles, say N7, is read in error. That would cause S0, S1, and S3 to be nonzero, since N7 appears in each of those. How can we determine that it's N7? We notice that the only thing that causes a syndrome nibble to be nonzero is the check nibble that's in error. In this example, we know that the error we're looking for is among the first 15 nibbles, N0 through N14, since S1 is nonzero and S2 is zero. Knowing that, we divide S0 by S1. Looking at the equations for C0 and C1 above, we see that the result will be 8 for our example. Since the nibbles are numbered from zero, we subtract 1, which designates N7 to be the erroneous nibble.

How should we fix it? The current value (the value we read from memory) of N7 is that of the erroneous nibble, and S1 is the XOR of the erroneous N7 with the original correct value of N7. So if we XOR the erroneous value we have with S1, we will recover the original correct value.

An error in the second 15 nibbles (N15 through N29) can be found and fixed in the same way.

Locating an error in nibble 30 or 31 is similar. Say S0 is zero and S1 and S2 are nonzero and equal. Then we know immediately that N30 is erroneous because it appears in S1 and S2 but not in S0. The correct value is recovered by XORing the value we have for N30 with either S1 or S2. Note that S3 will also have the same value as S1 and S2.

If just one of the syndrome nibbles is nonzero, we know that the corresponding check nibble is in error. It can be corrected by recalculating it from the data nibbles, N0 through N31.

Now, let's consider double error detection. In every case, if there is an error just in a single nibble, the number of nonzero syndrome nibbles is odd, so if two or four of the syndrome nibbles are nonzero, its a double error or more. It's also at least a double error if the following two conditions apply:

a) S1 or S2 is zero and the rest of the syndrome nibbles are nonzero.
b) Letting X represent whichever of S1 or S2 is nonzero, the position given by S0/X does not equal the position given by X/S3. Why? Let's pick on N7 again and let it be erroneous. Then, if it's a single error, S0/X = 8*N7/N7 which equals 8, and X/S3 should also equal 8 because X/S3 = N7/(f*N7) = N7*(8/N7) = 8, since 8 is the Galois multiplicative inverse of 'f' (from the table). If the positions are not the same, then it's not a single error, so it must be a double error or more.

Finally, it's at least a double error if S1 and S2 are nonzero and the nonzero syndrome nibbles are not all equal (otherwise it would be a single nibble error in N30 or N31).

Comments? Corrections? Questions?