2011 20th IEEE Symposium on Computer Arithmetic

Self Checking in Current Floating-Point Units

Daniel Lipetz, Eric Schwarz
IBM Corp
Systems and Technology Group
Poughkeepsie, NY, USA
{dlipetz, eschwarz}@us.ibm.com

***Abstract*—High performance microprocessors are protected against transient and early end of life failures using a variety of error detection and fault isolation technologies. Execution units can be protected with duplication, parity prediction, or residue checking. Residue checking has an advantage due to its small size. A modulus is selected based on the radix of the numbers being checked. In a decimal floating-point unit there are two types of numbers in different bases. There are base 10 decimal numbers and base 2 integers being used. A residue checking system that makes it easy to check both base 2 and 10 numbers is discussed. Current state of the art designs that are currently in use are described as well as a novel hybrid moduli 9 and 3 residue system. The checking systems for the decimal and binary floating-point units of some recent IBM microprocessors including the Power6, Power7, z10, and z196 microprocessors are detailed.**

Keywords-residue checking, error detection, fault isolation, microprocessor design, floating-point unit, self checking

I. INTRODUCTION

IBM mainframe systems are used for critical applications that require fault detection and continuous availability. Other microprocessor designs in the server market are trying to enter this reliability market. The design protects against transient failures from beta particles flipping bits in circuits due to exposure to radiation that can even be caused by the decay of materials in solder points. There is also logic designed to detect early end of life failures such as from electromigration. This is becoming more of problem today as oxide thicknesses approach the width of a few molecules.

Mainframes even have the ability to concurrently spare a processor if one detects a fault [1]. If an error is detected, the processor state is restored to the last checkpointed state which on zSeries is protected with ECC. The operation is then retried, but if the error is detected again, the processor state is swapped out to another processor. On a processor with this recovery capability, the important item is detecting that an error took place prior to checkpointing the machine state.

In the late 1990s on the G4 and G5 zSeries microprocessors the execution units were duplicated to detect errors[1]. This saved design time and resulted in great error coverage. As the integration of larger caches and additional cores per chip occurred, area is again becoming a premium. Covering all one bit errors is still critical but area is also highly valued. So, parity prediction and residue checking schemes are back in use.

The most area efficient technique for checking an execution unit today is residue checking. In practice parity prediction of an addition operation takes on the order of 1/3 the size of the adder. Prior estimates have varied and some authors estimate the area of a parity prediction circuit at close to the area of duplication when all elements are isolated[2,3,4]. Parity prediction is an excellent technique that needs to be used when the data needs to be immediately transmitted with its parity. This can happen when the receiving unit may be cache memory or register file which is protected with parity. If there is some flexibility in how the storage medium is protected then residue codes are an excellent substitute for parity. In the design of recent IBM microprocessors such as the Power6[5], z10[6], and z196 microprocessors, the L1 caches are protected with parity and there was flexibility in the choice of error codes for the register files.

We will first go through the ideas behind residue checking, then discuss the requirements for the decimal units, and then discuss several methods for handling both base 10 and base 2 numbers with good coverage. We will detail the changes between the recent IBM pSeries and zSeries designs.

II. RESIDUE CHECKING

Concurrent detection of errors in an operation can be accomplished by independently calculating a property of the result, such as its parity or its residue, based on the inputs to operation. Residue is the remainder that occurs when dividing a number by a modulus. For instance: $R_m(N) \equiv N \bmod m$ where the number, $N = a_n r^n + a_{n-1} r^{n-1} + a_0 r^0$ and r is the radix of the number system, m is the modulus, and $R_m(N)$ is the residue of the number N mod m. $R_m(N)$ can be obtained by dividing N by a positive integer m and retaining the remainder or "residue." In other words: $N = qm + R$, $0 \le R < m$ where R is the residue of N. All integers 0, 1, …, (m-1) all have unique residues modulo m.

There are moduli that are better choices than others for a number represented in a particular radix. A diminished radix (r -1) system makes it particularly easy to combine residues of each digit.

$$R_m(N) = R_m(a_n r^n + a_{n-1} r^{n-1} + \ldots + a_0 r^0)$$

$$R_m(N) = R_m(\, R_m(a_n) * R_m(r^n) + R_m(a_{n-1}) * R_m(r^{n-1}) + \ldots + R_m(a_0) * R_m(r^0))$$

Residues are commutative, associative and distributive for the operations of addition and multiplication. For a diminished radix system, the residue of the radix equals one

1063-6889/11 $26.00 © 2011 IEEE
DOI 10.1109/ARITH.2011.18

as well as any power of the radix, which simplifies many of the terms.

$R_m(N) = R_m(R_m(a_n) + R_m(a_{n-1}) + \ldots + R_m(a_0))$

The residue of "N" is simply a sum of the digits in the number. Typically for base 2 systems, a power of 2 is chosen for the radix such as 4, 8, or 16 and the corresponding diminished radix moduli are 3, 7, and 15. For a base 10 checking system, a simple modulus to use is 9

Binary floating-point units have coefficients in a base 2 number system and will typically use residue 3 or 15 for detecting errors. Residue 3 is thought to be smaller but provides less coverage than residue 15. For decimal floating-point numbers most of the coefficients are in base 10 and can use either residue 3 or 9 for checking. Though fixed point integers are also present in the decimal floating-point unit for conversions, and they should be checked as a base 2 number using residue 3 or 15. For base 10, residue 9 is very effective but residue 3 is very nice since it can be used for both BCD numbers as well as fixed point integers.

III. RESIDUE 15

Prior to the execution units being duplicated in zSeries microprocessors such as in the G4 and G5 systems, it was typical to design modulo-15 residue generators and also use parity prediction. This was prior to CMOS technology and instead high-speed power-consuming bipolar transistor circuits were common and they were more prone to transient failures. The residue 15 generators were used in binary/hexadecimal floating-point units as well as integer units to check arithmetic circuits. A modulo 15 system is a diminished radix system and all powers of 16 have a residue of 1. The modulo-15 of a binary number is simply the sum of its hexadecimal digits.

$X = \sum x_j 16^j$ where x_i is a digit from 0 to 15.

$R_{15}(X) = R_{15}(\sum R_{15}(R_{15}(x_j) * R_{15}(16^j)))$

$R_{15}(X) = R_{15}(\sum (R_{15}(x_j)))$

$R_{15}(X) = R_{15}(x_0 + x_1 + \ldots + x_n)$

For a modulo-15 generator it is typical to implement it by using a counter tree such as 3:2 counter trees similar to a multiplier's partial product array implementation. This is costly in terms of area and delay.

The residue 15 system provides excellent coverage and is used in the current z196 microprocessor. The z196 binary floating-point unit design uses 4:2 carry save adders (CSA). For a 64-bit operand there are 16 hexadecimal digits and they need to be summed to one digit.

IV. RESIDUE 3

With the advent of CMOS, modulo-3 checking systems have become popular due to their speed and size and the belief that CMOS might have fewer transients fails. Modulo-3 checking systems reduce a base 4 system as shown by the following:

$X = \sum x_i 4^i$ where x_i is a digit from 0 to 3.

$R_3(X) = R_3(\sum R_3(R_3(x_i)\ R_3(4^i)))$

$R_3(X) = R_3(\sum (R_3(x_i)))$

$R_3(X) = R_3(x_0 + x_1 + \ldots + x_n)$

A modulo-3 system can also check a base 10 system since every power of 10 has a remainder of 1 when divided by 3.

$X = \sum (x_{2i} *4 + x_{2i+1})10^i$

Where x_{2i} represents the 2 high order bits of a BCD digit and x_{2i+1} represents the 2 low order bits.

$R_3(X) = R_3(\sum R_3((R(x_{2i})\ R_3(4) + R_3(x_{2i+1}))\ R_3(10^i)))$

$R_3(X) = R_3(\sum R_3(R_3(x_{2i}) + R_3(x_{2i+1})))$

$R_3(X) = R_3(x_0 + x_1 + \ldots + x_n)$

So, just like the binary system, groups of 2 bits can be summed together in the form of a counter tree.

There is a faster implementation of modulo-3 that has been used. Every 2 bit group can be recoded into 3 output signals: res0, res1, or res2:

res0 = x0' x1' + x0 x1

res1 = x0' x1

res2 = x0 x1'

Note bit 2 of the 3 bit vector has a weight of 2, bit 1 a weight of 1 and bit 0 a weight of 0. For example, "100" is equal to 0, "010" is equal to 1, and "001" is equal to 2. The one-hot expansion is a very useful encoding[7,8]. If the whole tree can not be completed in one cycle, only two out the three bits need to be latched.

The addition of residues could be implemented by a 2x3 And-Or gate. Since res0, res1, and res2 are orthogonal, this could be implemented with a pass-gate multiplexer. These gates are extremely small and fast, and thus, modulo-3 systems will be smaller than a discrete gate 3:2 counter based modulo-15 system. The pass-gate multiplexers perform the addition of the residues via rotation. A residue of "001" + "010" (2 +1) would rotate the "010" one bit left to "100" or a value of 0 [7,8]. These pass-gate multiplexers need two different implementations to compensate the selects being either polarity. Gajski [9] describes a similar system using bipolar junction transistors and a similar tree design. The idea is extended to our technology.

V. RESIDUE 15 BY 3 AND 5

Another method of implementing residue 15 is to use both moduli 3 and 5. A one hot encoding system can be used for both residue 3 and 5 providing a very fast and small implementation with pass gate multiplexors[7,8]. When two relatively prime moduli are used, the effective modulus is their product, thus, providing the equivalent of residue 15 checking. Two pass gate multiplexor implementations may be smaller than a counter tree for residue 15.

VI. RESIDUE 9-3

Decimal floating-point units receive both BCD coefficients and integers as input to their arithmetic operations. Modulus 3 checking circuits can easily be applied to both of these number systems. If greater error coverage is needed then a system of implementing both moduli-3 and 9 is useful. Rather than implement these two systems separately there is a novel way of combining them. The system utilizes modulo-3 gates but is connected in a way to provide either a modulo-9 or modulo-3 residue. First, the modulo-9 residue generator is considered for decimal formats and then modulo-3 is an easy extension of this logic.

The input operand is separated into hex digits, which for decimal numbers is expanded to the BCD format as shown by the following:

$X = \sum x_j \, 10^j$ where x_i is a digit from 0 to 9.

$R_9(X) = R_9(\sum R_9(R_9 (x_j) \; R_9 (10^j)))$

$R_9(X) = R_9(\sum (R_9 (x_j)))$

$R_9(X) = R_9(x_0 + x_1 + \ldots + x_n)$

Next we break each of the operand digits into two base 3 digits and then take their residue modulo-9.

$x_i = x_{i,1} \, 3^1 + x_{i,0} \, 3^0$

$x_i = x_{i,1} \, 3 + x_{i,0}$ with $x_{i,1}$ and $x_{i,0} \in \{0,1,2\}$

$R_9(X) = R_9(\sum (x_{i,1} \, 3 + x_{i,0}))$

$R_9(X) = R_9(R_9(\sum x_{i,1}) \, 3 + R_9(\sum x_{i,0}))$

The first step in creating this base 3 modulo 9 residue is to decode each 4-bit input into two base 3 numbers, 'ah' being the high digit and 'al' being the low digit.

$x_i = x_{i,1} \, 3^1 + x_{i,0} \, 3^0$

$x_i = x_{i,1} \, 3 + x_{i,0}$

$a_i = ah_i \, 3 + al_i$

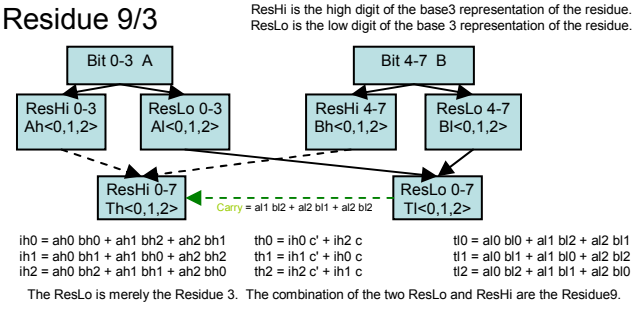
$a = (ah_{(0,1,2)} \, 3 + al_{(0,1,2)})$

The high residue is only used for base-10 input and the non-BCD (e.g. "1111") are don't care values. If any of the input is not BCD only residue modulo 3 is used.

Once each digit is recoded in base 3 format, a tree of high and low digits is formed. For 4 digits (16-bits), first take two 4-bit digits. Calculate their residue: a high digit, a low digit, and a carry from the low bits. The residue is generated by adding the low digits of the two numbers and adding the high digits of the two numbers. In the high digits, the carry out is ignored. This carry represents a '9' added, which in residue 9 is a multiple of the residue. There is also a carry generated from the low digits (this can be pre-computed). This carry will then later be added into the high base 3 digit to complete the residue. It is saved for the last step so the merging of the residues can be done in parallel and not need to wait for the carry to be generated.

After the first merge, there are two base 3 digits representing the residue 9 and a carry (also a base 3 number). For the next and all subsequent merges, the two residue merging (high and low) are computed independently. The carry is still generated and rolled into the carry by choosing to increment or not. Our implementation is a one-hot configuration which allows us to rotate the bits for easy addition and incrementing.

The final merge is the same as the previous, but the high digit has one extra step, in that the carry is finally added into the high residue bits. An example is shown in Figure 1.



The ResLo is merely the Residue 3. The combination of the two ResLo and ResHi are the Residue9.

Figure 1. Residue 9/3 Tree For 1 Byte

VII. APPLICATIONS

Parity and residues are used for data integrity checking of transmitted or stored data. They are also used for verification of the integrity of arithmetic circuits.

Parity can detect if one (or any odd amount) of bits have been switched from a 0 to a 1 or visa versa. Parity is a little different from residue in that it is typically implemented 1 bit per each byte of the data. Parity can detect all 1 bit flips within a byte. Residue checking can fully detect a 1 bit flip in the overall result.

Errors that cause multiple bit flips in the result are rare but they can occur if a circuit in the carry chain fails, or in general, if a circuit driving many bits fails. To protect against these types of fails, higher modulus residue checking is used. For totally random fail patterns (multiple bits), residue 3 checking provides 2/3 or 66% detection coverage. Residue 9 can detect 8/9 = 88%. Residue 15 can detect 14/15 = 93%.

For the addition operation, the residues of the addends added together will be equal to the residue of the sum. For subtraction, the difference of the residues of the minuend and subtrahend will be equal to the residue of the difference. For subtraction, if the residue of the minuend minus the residue of the subtrahend is negative, the modulus, 9 for residue mod 9, should be added for a positive residue. For the multiplication operation, the product of the residues of the multiplier and the multiplicand multiplied will be equal to the residue of the product. For division, multiply the residues of the divisor and quotient, and add the residue of the remainder and this should equal the dividend's residue.

Conversion is an operation that presents some difficulty in checking. Comparing a binary floating-point significand to an integer is easy since both have a base 2 representation. But converting from decimal to integer is a little different. Modulo 9 is nice for checking decimal operations but for conversions to and from integers modulo 3 is used. Modulo 3 works well for both decimal and binary systems and provides a common residue that can be used to check conversion between them. Thus, the hybrid modulo 9-3 system is excellent for use in decimal units.

Residues can also be used to sanity check certain mathematical operations in decimal arithmetic. The residue modulo 9 of a square must be 0, 1, 4, or 7[10].

For floating point operations this can get tricky when alignment is needed. When truncating an intermediate result, one must keep track of the truncated digits. Rounding also must be considered. Checking must also be aware of NANs, infinities or other special numbers. The residue of a NAN obviously has no mathematical significance, though it can be used to protect the payload or coefficient during transmission or storage. In general, the residue checking is applied to the significand calculation of multiply-add dataflow. For the exponent calculation there should also be some form of checking. If error coverage is a priority, duplication is the best method. An error in the exponent has a dramatic effect on the result. If less checking coverage is required for

multiple bit errors then residue 15 is an excellent choice. The exponent, significand, and the interface, as well as the controls are usually checked separately.

One important characteristic of residue checking is that it is independent of the design of the adder or multiplier or arithmetic block. It only depends on the inputs and outputs.

VIII. History

When choosing a value for modulus m, there are several considerations. Included are the cost of the special checking hardware, both in power consumption and area. Other factors include storage capacity, the increase in time to perform the checking and the complexity added to the control structure.

There is some data on some of the first computers of using error checking. The Raydac computer designed in the 1950s used a mod 31 check on addition. The Univac III used a modulo 3 check[4].

The more recent CMOS line of zSeries mainframes such as the 1997 G4, 1998 G5, 1999 G6 processors have implemented duplication. The z900 processor was released in 2000 and is the first 64-bit zSeries architecture processor. Like the processors before it, the execution units use full duplication with the two copies simultaneously performing the same operations. There is the unlikely event of having the same transient fails in both copies as well as some exposure in the comparison hardware. This provides almost 100% coverage but uses a large amount of area and power. Additionally, the outputs must be compared before using.

The z990, released in 2003, was the first zSeries processor with a fused multiply-add dataflow. This used a skewed duplication checking technique. The core execution units are duplicated and the second copy is running one cycle behind the first copy. This allows almost 100% coverage and allows use of the output of the unit prior to checking. The z9-109, released in 2005, also used skewed duplication technique.

The z10 mainframe released in 2008 utilizes residue 15 in the binary floating-point unit and residue 3 in the decimal floating point unit. For the exponent dataflow, duplication is used.

The z196 released in 2010 uses a more comprehensive residue checking technique. It uses residue 15 in the binary floating-point unit and the fixed-point unit. It also uses a combined residue 3 and 9 checking system for the decimal floating-point unit.

POWER6, released in 2007, uses residue 3 for both the binary and decimal floating-point units. This was the first time the floating-point units had internal dataflow checking on pSeries servers.

POWER7, released in 2010, utilizes residue 15 in its vector scalar unit (VSU). It takes up about 18% of the total FPU area.

IX. Comparison

A comparison between several of the encoding techniques is shown in Table 1. The 45 nm technology used in the z196 processor is used for these comparisons. The residue 3 (Res 3 t-gate) and residue 9-3 use a normal 18 tracks per bit design. They use a pass gate implementation that makes them very small and fast. The residue 15 circuit is also very small since it is able to utilize an ultra-compact 9 tracks per bit topology. 4:2 counters are also able to utilize a pass gate design. Traditional 3:2 counters are much bigger. The Res 15 18 track implementation demonstrates this. The Res 3 CMOS implementation uses standard CMOS gates. It is the largest method and shows the disadvantage of standard gates versus transmission gates. The Res 3 Serial implementation is a serial generating method which loops on itself. This is the smallest design but has the obvious drawback of being the slowest.

TABLE I. Area comparisons

Method	input	Output	Tracks per bit	Area
Res 3, t-gate	64 bit	2 bit	18	197x8.5um
Res 3, cmos	64 bit	2 bit	18	197x14.5um
Res 3, Serial	64 bit	2 bit	18	197x3.75um
Res 9-3	64 bit	2 – 2 bit	18	191.5x14um
Res15	64 bit	2 – 4 bit	18	158.5x10.5um
Res 15	64 bit	2 – 4 bit	9	158.5x5.9um

X. Conclusion

Many residue implementations are discussed. Each has its own cost and advantage and a design decision must be made to weigh this. The introduction of transmission gates into circuit topologies can greatly increase the speed and the area consumed by the residue checking. This has allowed the increase of the checking modulus which has in turn increased the error checking capability for the same amount of area in a smaller cycle time.

References

[1] T. Slegel, et. al., "IBM S/390 G5 Microprocessor," IEEE Micro, vol. 19, no. 2, pp. 12-23, March 1999.

[2] Langdon, G. G. Tang, C. K., "Concurrent Error Detection for Group Look-ahead Binary Adders" IBM Journal of Research and Development Volume: 14 , Issue: 5. 1970 , Page(s): 563 – 573

[3] S. Mitra, E.J. McCluskey, "Which concurrent error detection scheme to choose?," Proceedings of Int. Test Conference, pp. 985-994, 2000.

[4] F. Sellers, M. Hsiao, and L. Bearnson, Error Detecting Logic for Digital Computers, McGraw-Hill, New York, 1968.

[5] L. Eisen et. al. "The IBM Power6 accelerators: VMX and DFU," IBM Journal of R & D, vol. 51, no. 6, pp. 663-684, Nov. 2007.

[6] E. Schwarz, J. Kapernick, M. Cowlishaw, "Decimal Floating-point support on the Z10 processor, " IBM Journal of R & D, vol. 53, no. 1, pp. 4:1-4:10, Jan. 2009.

[7] W.A. Chren, "One-hot residue coding for high-speed non-uniform pseudo-random test pattern generation," proceedings of ISCAS 1995, vol. 1,, p. 401-404, 1995.

[8] W.A. Chren, "One-hot residue coding for low delay-power product CMOS design," IEEE Trans. On Circuits and Systems II, vol. 45, no. 3, pp. 303-313, 1998.

[9] Gajski, " Modular modulo 3 module" US 4190893, 1977.

[10] http://mathworld.wolfram.com/SquareNumber.html.