Efficient Fault-Tolerant Addition by Operand Width Consideration

Bernhard Fechner, FernUniversität in Hagen, Dept. of Mathematics and Computer Science, 58084 Hagen, Germany
Jörg Keller, FernUniversität in Hagen, Dept. of Mathematics and Computer Science, 58084 Hagen, Germany

Abstract

Addition is a central operation in microcontrollers and hence faults should be detected for safety reasons. We extend the principle of recomputing with shifted operands (RESO) by doing the re-computation concurrently to the computation in the case of small operands. Thus, we generate a solution cheaper than two adders and faster than simple repetition. To extend RESO, we consider the actual bit-widths of the operands. We validate our method with data from static code analysis of two application kernels.

1. Introduction

Detection of hardware faults, permanent or transient, is achieved by exploiting redundancy. Addition can be made fault-tolerant by structural redundancy, e.g. by using two adders and comparing the results, or by temporal redundancy, i.e. by having the same adder perform the addition twice, in different manners. We present a compromise that needs fewer resources than two adders and needs less time than two cycles per addition.

Addition was chosen as an example because it presents several challenges. First, addition is central to all types of computation in all kinds of processors, from simple microcontrollers to multi-billion transistor high-performance superscalar microprocessors. Second, an adder circuit itself is not very costly concerning its area. Thus, measures that avoid another adder must be very efficient. Third, while in large processors a second adder may be the solution at hand because area is available in plenty, another adder makes a difference in small microcontrollers and in terms of area and of energy consumption. But microcontrollers need fault-tolerance as well, sometimes more than microprocessors, because microcontrollers are often used to control devices interacting with the real world.

A general overview over redundant number systems as well as error detection and correction is given in [5]. Error detection is currently achieved by duplicating or triplicating hardware [7], time redundancy [6] and recomputation by using shifted operands (RESO) [4, 8]. Further techniques based on RESO include the recomputation with swapped operands [1] or the recomputing with duplication with comparison [2].

Our method extends RESO by considering that not all additions need the full width of the adder. If we have an n-bit adder, and an addition only has $n/2$-bit operands, then the two versions of the same addition might be executed on the adder concurrently.

The remainder of this paper is organized as follows. Section 2 presents our method, Section 3 presents an analysis with two application kernels, and Section 4 presents a conclusion and an outlook to future work.

2. Operand Width Consideration

An n-bit adder of any type can easily be used to perform multiple additions of smaller widths concurrently by suppressing carry signals that cross the border of the different additions. A prominent example where this method is applied are multimedia extensions in microprocessor instruction sets. In our case however, the sizes of the operands are not known in advance. To compute the number of leading and trailing zeroes in an operand requires a parallel prefix computation [3] and thus is as complex ($O(n)$ gates, $O(\log_2 n)$ depth) as an addition itself.

To provide fault-tolerance, we can restrict to a simpler question: can two identical additions be computed concurrently or not? The answer is yes if both operands of an addition only require up to $n/2$ bits. This is the case if the $n/2$ leading bits of each operand are zero. If this test is positive, then the upper $n/2$ bits of the adder can be used to perform the same addition as the lower $n/2$ bits do. Hence we need $n/2$-bit multiplexers connected to the upper $n/2$ input bits and the lower $n/2$ input bits of each operand, and a 1-bit multiplexer on the carry from position $n/2-1$ to $n/2$. This multiplexer is fed with the original carry, and with the carry in. The latter is needed if two additions are performed concurrently. The principle is illustrated in **Figure 1**, where a_l, b_l and a_h, b_h denote the lowermost and uppermost $n/2$ bits of n-bit operands a and b respectively and c_{in} denotes the carry in.

We exemplary extend the RESO principle for 16-bit operands and perform the addition twice but concurrently, where the second addition is done with operands shifted to the uppermost bits. For 32-bit operands, we repeat the addition after it is finished. For the repetition, we might swap the operands if both adder inputs are connected to different read ports of the register file. Note, that we implicitly assume 32-bit addresses, if not otherwise stated.

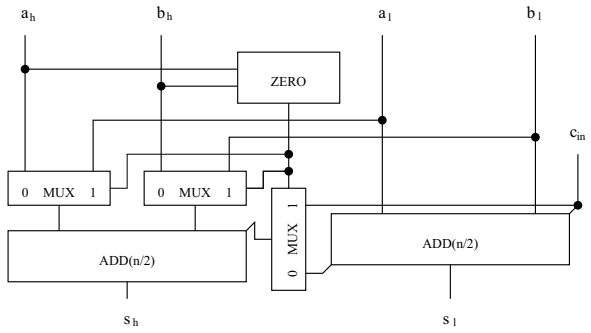

Figure 1: Redundant addition hardware.

3. Analysis

The cost of our method is two multiplexers of $n/2$ bits each and a 1-bit multiplexer, having the total complexity of $3n + 4$ gates, counted conservatively, as multiplexers can be made almost as cheap as a single gate at transistor level. Furthermore, we need circuits to test whether the uppermost $n/2$ bits of both operands are zero. This can be achieved by a simple OR-tree over $n = 2 \cdot n/2$ bits. If we assume that 4-input OR gates are available at twice the cost of 2-input OR gates and only slightly larger delay, then for $n = 32$ we only have cost 21 and delay 3. Thus our method is cheaper than another adder which would have cost $5n$ even for the cheap ripple-carry adder.

We admit that our solution might slightly increase the cycle time if the adder is on the critical path because of the zero-counter and the multiplexers. However, the scheme can be pipelined and the increase is only 4 gate delays, 2 for the zero-counter, 2 for the multiplexer. The comparison of both results is needed in any redundancy scheme and can be postponed into the next cycle.

Our method needs two cycles for large operands, but only one for additions of $n/2$-bit operands.

In order to experimentally assess the advantage of our method, we deduct the operand widths by using common codes. As a first example, we choose Livermore Loop 1, as given below.

```
for(k = 1; k <= 990; ++k){
  space1_1.x[k-1] = \
    spaces_1.q + space1_1.y[k-1] \
      * (spaces_1.r * space1_1.z[k+9] + \
         spaces_1.t * space1_1.z[k+10]);
}
```

There are 19 additions (and subtractions) in each loop iteration. These are the comparison by subtraction, the increment of `k`, four additions to compute array indices (`k-1,k-1,k+9,k+10`), seven additions to compute the address of a struct element (e.g. `space1_1.x`), four additions to add the array index to the base address, and two additions to evaluate the expression.

Of these 19 additions, only the first 6 use 16-bit operands. All address calculations are 32 bit, and the data in the arrays is unknown at compile time and thus must be assumed to be 32 bit.

Thus, 6 out of 19 additions (31.6%) are 16 bit additions and can be done in one cycle, so that the average addition time is 1.68 cycles instead of two cycles. One may assume Livermore Loop 1 to be not the typical code to be executed on a microcontroller, but for clarity we did not want to take a code too advantageous for our method. Control codes will often have less accesses in arrays and structs, so that a greater fraction of additions will be 16 bit.

As a second example, we choose multiplication of dense, square matrices of single precision floating point numbers. We assume that our processor emulates floating point multiplication in software, and contains a multiplier for 16-bit unsigned integers only. We further assume that the matrix dimensions are 256 at most. This assumption is based on the fact that dense matrices are used.

```
unsigned int i, j, k;
float a[n][n], b[n][n], c[n][n];

for(i = 0; i < n; i++)
  for(j = 0; j < n; j++){
    c[i][j] = 0;
    for(k = 0; k < n; k++)
      c[i][j] += a[i][k] * b[k][j];
  }
```

Access to a matrix element such as `c[i][j]` is typically implemented as `*(cbase+(i*m+j))`, where `cbase` is the base address of the matrix `c`. As i, m and j are less than 256, the addition $i \cdot m + j$ is a 16-bit addition. As `cbase` is a 32-bit address, the second addition is a 32-bit addition. Access to elements of other matrices also comprises one 16-bit addition and one 32-bit addition each.

Each loop iteration comprises two 16-bit additions, one for the subtraction to implement the less-than comparisons such as $i < n$, and one to increment the loop variable. After the last iteration, there is a final comparison that terminates the loop.

Each floating-point multiplication is realized in software as follows. For clarity of presentation, we only present the case where both factors are normalized and where the resulting product is normalized, too. For details on floating-point representations and their arithmetic, cf. a textbook on arithmetic such as [5]. A normalized, single-precision IEEE 754 floating-point representation is stored as a sign bit s, 23 fractional bits m (plus the implicit 1 for the integral part) of the mantissa and 8 bits for the characteristic c. The number represented is

$$(-1)^s \cdot 1.m \cdot 2^{c-b} ,$$

where $b = 127$ is the bias. The product of two such representations (s_1, m_1, c_1) and (s_2, m_2, c_2) is (s', m', c') with $s' = s_1 \oplus s_2$, $c' = c_1 + c_2 - b$ and $1.m' = 1.m_1 \cdot 1.m_2$.

Table 1: Frequencies of 16-bit and 32-bit additions in matrix multiplication

Code line	16-bit additions	32-bit additions
i loop	$2n+1$	
j loop	$(2n+1)n$	
setting c_{ij}	n^2	n^2
k loop	$(2n+1)n^2$	
access matrix el.	$3n^3$	$3n^3$
float mult.	$2n^3$	$3n^3$
float add.	n^3	n^3

Note that if $1.m_1 \cdot 1.m_2 \geq 2$, then we have to shift the result to the right and increase the resulting characteristic c'. To multiply the two mantissas, we split both of them into 12-bit high and low halves, and perform four multiplications and three additions.

Thus, the computation of the resulting mantissa requires three 32-bit additions while the computation of the result characteristic comprises three 16-bit additions if normalization with shift and increment is needed, and two 16-bit additions if not. For simplicity (and to our disadvantage), we assume that all floating-point values used have mantissa values less than $\sqrt{2}$ so that their product is less than 2. Thus no result normalization with mantissa shift and characteristic increment is needed, and two 16-bit additions suffice.

A floating-point addition comprises one subtraction of the characteristics to align the mantissas, and one mantissa addition, i.e. one 16-bit and one 32-bit addition.

In total, we have $8n^3 + 4n^2 + 3n + 1$ 16-bit additions and $7n^3 + n^2$ 32-bit additions, as **Table 1** indicates. Thus, about $8/15 \approx 53.3\%$ of all additions are 16-bit additions, reducing the average addition time to 1.467 cycles.

If we would use pointer arithmetic to avoid multiplications in addressing matrix elements, an appropriate alignment of matrix rows to addresses that are multiples of 2^{10} would even improve the situation to a single 16-bit addition, without 32-bit additions. To achieve this, one would use a 10-bit unsigned integer `offset` and address matrix elements by `*(cbasei|offset++)`, where `cbasei` is the base address of row i of matrix c.

4. Conclusions

We have presented a simple method that extends RESO to provide fault-tolerant additions with smaller cost than methods based on structural redundancy, and less time penalty than approaches based on temporal redundancy, thus providing a suitable compromise. Our method is valuable for small devices where die area is scarce and time is valuable. We have assessed the speed of our method to be 1.47 to 1.68 cycles on average for two example codes.

Future work will comprise evaluation with more and diverse codes, extension to other, more elaborate types of operations such as multiplication, and extension to use this method not only for redundancy but for acceleration of codes as well.

References

[1] B.W. Johnson. Fault-tolerant microprocessor-based systems. *IEEE Micro*, 4:6–21, December 1984.

[2] B.W. Johnson, J.H. Aylor, and H.H. Hana. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. *IEEE Journal of Solid-State Circuits*, 23:208–215, February 1988.

[3] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[4] J. Li and E. E. Swartzlander Jr. Concurrent error detection in ALUs by recomputing with rotated operands. In *Proc. IEEE Int.l Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 109–116, 1992.

[5] B. Parhami. *Computer arithmetic and hardware designs*. Oxford University Press, 2000.

[6] J.H. Patel and L.Y. Fung. Concurrent error detection in ALUs by recomputing with shifted operands. *IEEE Trans. Comp.*, 27:1093–1098, December 1978.

[7] D.P. Siewiorek and R.S. Swarz. *Reliable Computer Systems Design and Evaluation*. A.K. Peters, 3rd edition, 1998.

[8] Whitney J. Townsend, Jacob A. Abraham, and Earl E. Swartzlander, Jr. Quadruple time redundancy adders. In *Proc. 18th IEEE Int.l Symp. Defect and Fault Tolerance in VLSI Systems*, pages 250–256, 2003.