Safety Computations in Integrated Circuits

Jean-Louis DUFOUR
MATRA TRANSPORT International - RAMS department
48-56 rue Barbès, B.P. 531, 92542 Montrouge Cedex, FRANCE

Abstract

In order to ensure the safety of software-based railway control systems, MATRA TRANSPORT has developped at the beginning of the eighties an "informational redundancy" technique associating arithmetic coding and signature checking, with the adequate environment interfaces (generally fail-safe devices). Compared to traditional redundancy, the "coded processor" has the advantage of a rigorous mathematical safety demonstration, independant of the reliability of the underlying hardware, but there is an important cost to pay in terms of execution speed.

One of the (strongly) desired evolutions of our systems is to have a unique centralized wayside equipment, the immediate corollary beeing the decentralization of inputs/outputs. In order to reach this goal, a new generation has been designed, replacing the software code calculations and the discrete numeric components used in coded input acquisition / coded ouput command by ASICs. Our experience shows that it is possible to perform safe computations in an ASIC, and even that in some cases ASICs are more adapted to the safety constraints than software computations.

Principles of coded arithmetic

The Vital Coded Processor is a transposition to software computations of the "casting-out nines" verification method used by pupils. The main difference is that the check is not the modulo 9 of the value, but the modulo A, with A big enough to satisfy the safety requirement (typically 10^{-9} dangerous events per hour, hence A is a 32 or 48 bits value).

The software representation of a variable "Z" is no more a single memory word, but a record with
- a functional field "Z.F", which contains the "natural" value of the Z variable,
- a check field "Z.C", which contains the redundancy check bits of the Z variable .

To add two coded variables (X and Y, result in Z), we must :
- add the functional fields : Z.F := X.F + Y.F,
- add modulo A the check fields : Z.C := (X.C + Y.C) [A].

For a safety reason, the check is the modulo A not simply of the functional value x, but of the product r.x, with r well chosen (the same phenomenon occurs for the CRCs : there, $r = X^k$, where k is the size of the redundancy; see Peterson [1]). This is not essential for the understanding of the principles.

A random error occuring on the functional or the check field has 1/A probability of being undetected. The major hypothesis concerning the safety level evaluation of the Coded Processor is that **"hardware failures can be considered as random errors with regard to the code"**, or maybe formulated more simply, **"hardware failures don't know how to build multiples of A"**. This hypothesis can be intuitively based on considerations about arithmetic distance between code words.

Specificities of the Coded Processor

This technique is well known since the late sixties, where it was intended for error detection in arithmetic units of processors (see Rao [2]). But for our purpose, it turns out to be inadequate, and therefore to meet the assigned goal, it is slightly modified.

In fact, when performing z := x+y, (z,x and y are coded variables, + is overloaded) if x is taken instead of y (that is, if z := x+x is performed, this will be called an "addressing error"), this is not detected by the code. To address this problem, a **signature** specific to each variable (noted "$\mathbf{B_x}$" for the variable x) is introduced in the check field, which has now the form : $r.x + B_x$. After the evaluation of z := x+y, the B_z must have the value : $B_x + B_y$, and if an addressing error occurs, it is detected. A signature B_x is specific to a precise occurrence of the variable x, and if all occurrences of all variables have different signatures, an addressing error has a 0 probability to be undetected. Now in a big program with a "little" A (or take simply a program of 10 instructions with A = 9), you can't avoid "collisions" of B_xs. But even in this case, **if there is a uniform repartition of the B_xs**, it can be easily shown that **the probability of undetection is 1/A.**

Unfortunately, there is a program construction which deliberately violates this requirement of regular repartition of signatures : the loop. Indeed, each occurrence of variable must have a predictable signature, and this conflicts with

0-8186-7304-4/96 $05.00 © 1996 IEEE

the unpredictability of the number of iterations in a loop. Hence, at the end of the body of a loop, if a new iteration is to be performed, the coded fields of the variables used are "compensated" (by addition of a specific compensation for each variables) in order to retrieve the signatures at the entry in the body of the loop. This create deliberate collisions, because each iteration uses the same signatures, and storage errors (use of an "old" value of a variable) are not detected. To address this problem, a **date** (time stamp, or "dynamic" signature [the B_x is then called "static" signature], noted "**D**") incremented at each new iteration is introduced in the coded field, which has now its definitive form : $r.x + B_x + D$. Even if the program has no loop, it is itself cyclic (there is a toplevel infinite loop) and this protection is mandatory.

Moreover, signatures are a protection against "operator errors", for instance performing a multiplication when you must perform an addition (the condition is of course that the functions giving the signature of the result from the signatures of the operands are different between any of the two operators).

In fact, any program can be represented by a tree of operators whose leaves are the variables. Hence the effects of hardware errors will be either
- operator errors (ex : to perform a + instead of a *, to forgive an instruction, to exchange two instructions),
- operand errors, the operator being the correct one (memorizations are classified in this category, see Forin [3]),
- operation errors, when operation and operands are correctly chosen (ex : random error on the operands or on the result).

The Coded Processor detects these 3 kinds of errors, hence detects all hardware failures.

Comparison with redundancy

Sometimes people have the following first reaction : **"why this complex encoding of x (r.x modulo A), why don't you simply take x itself as its own check, that is why don't you simply manipulate (x,x) ?"**

What they suggest is essentially traditional redundancy (on data), which can take two forms :
- (parallel) encoding X into (X,X), the second X not being stored on the same data bank as the first,
- (sequential) not encoding X, but running the program twice on different data banks.

Let's suppose the software delivers the output (1,1). Can you answer the following questions :
- what is the probability to have an undetected error ?
- what about common modes (bus, power supply, ...) ?

The first question can be partially answered by reliability arguments (the error considered is a double one, one on each data bank), but then the second question remains unanswered.

The safety of the coded processor is absolutely not based on reliability, but only on the power of an error detection code (principle of cyclic codes used in transmissions).

Comparison with test

An alternative to redundancy is periodic test, a possible implementation is the following : once your software has given its output, before applying it, a memory test is performed. In this case, error detection is less dependent of the (permanent faults) l, but depends mainly on the "coverage ratio" of the test and of the ratio of transient errors.

The comparison between the coded processor and periodic test is more pertinent, with the following difference : a test checks all what you want, excepted the applicative software; a coded computation checks exactly and only the applicative software. Hence, coded computations can be viewed as "built-in tests", "in" signifying "in the applicative computation", and in this respect the "coverage ratio" of coded computations is 100%.

Interfaces with the environment

The acquisition of the inputs is done via a fail-safe analog circuitry which generates a precise sequence of bits only if the entry is permissive (i.e. powered on; usually this sequence is the exclusive-or of two primary sequences). Bit by bit this sequence is communicated by numerical circuits (bus drivers) to the CPU, which transforms it into a dated code (datation and deserialisation). Bus drivers and discrete numerical circuits have no idea of the code (or sequence) which is used, hence the only danger is the storage (and later reuse) of the permissive sequence, and the datation must be done before the deserialisation in the CPU. In practice, sometimes datation is the first operation after the analogical acquisition block, sometimes it is done in the CPU before the deserialisation, the criterium is that **"before dating, it must be impossible (or compatible with the safety requirements) to memorize a sequence"**.

The application of the outputs is very similar, because they are not safely driven, but safely checked when commanded at restrictive state.

If the check of an output shows a problem (or if a problem is detected in the software computation), the power supply of the outputs is switched off, and by design this must correspond to a safe state of the system. This is done by the Vital Controller, which takes as input a dated code which is the synthesis (in fact, the sum) of the software check codes and the output check codes, and transforms it into an (undated) sequence, which will be compared by a fail-safe analogical circuit with a reference

sequence in PROM. The criterium is that "**after the dedatation, it must be impossible (or compatible with the safety requirements) to memorize the sequence**".

The new generation

The main goal of the new generation is to have a unique centralized wayside equipment, the immediate corollaries being the decentralization of inputs/outputs and a significant increase of performance of the coded computations (see Lardennois [4]). Among the reasons for wishing this are economy and simplicity (suppression of the train transfer management between adjacent wayside equipments).

The software computations on the check fields induce an important loss of performance. A coprocessor has been designed to work in parallel with the CPU : the CPU works on the functional fields and the coprocessor works on the coded fields, and if you choose as CPU a 68020 running at 20 MHz (available in double source and industrial temperature range), it won't be slowed down and will give you the performance of a PowerPC (running software coded computations). From the safety point of view, the coprocessor works internally on undated data, this possibility was very restricted in the software implementation. We will see why an ASIC is better than a CPU in this respect.

In the current generation, the acquisition of one input requires 48 bits, and even if it is a sequence, it also requires a parallel star bus (because each acquisition card handles 4 inputs). The new generation requires only a serial ring bus and one bit per entry plus 48 check bits. This is possible because a software computation on sequences has been "delocalised" on the acquisition board in an ASIC. From the safety point of view, the problem is that this ASIC manipulates the (undated) acquisition sequences. We will see how it is possible.

The same system is used to check the vital outputs at restrictive state.

The new Vital Controller is able to tolerate errors from time to time (under strictly controlled circumstances), and in order to do this it operates on the undated global check just before the creation of the final control sequence (which drives the fail-safe power supply). From the safety point of view, this would have been difficult in the software implementation, and again the ASIC solution has shown significant advantages from a safety demonstration point of view.

The development of an ASIC is a rather heavy process, but the advantages are significant, among which :
- the expression power of these components suppresses many conventional development constraints,
- possibility of incorporating efficient built-in tests and error diagnostics,
- excellent safety control of code computations.

Acquisition of inputs

An undated 48 bits code is elaborated from the acquisition sequence and is stored in a serial register, waiting for the right moment to be incorporated in a message on the ring serial bus. The danger is for example the transformation of this serial register into a cyclic register, and the systematic reuse of the code memorised, no more taking into account the acquisition sequence.

The technique used is called "destructive generation" or "destructive reading", this ensures that when the right code is inserted on the serial bus, the content of the serial register is altered, and so the only way to generate again the right code is to start a new acquisition sequence.

The coprocessor

The coprocessor handles internally undated codes, for example when it performs an addition, it
- dedates the arguments,
- adds the dedated codes,
- dates the result.

The danger is to memorize an undated code (either by a freeze of the register which contains it, or by a copy in another register), and (if we are for example in a short loop with only one instruction in the body) to reuse it at the next iteration, instead of the code normally computed.

The technique used is the following :
- there are several internal register banks, all the registers which contain undated codes are on the same bank, all nominal links from the "undated bank" to the other ones either date the code or break it. To create a (non-nominal) link between two banks, a failure must create 32 ordered short-circuits : we consider this as impossible.
- after each loop iteration, all the registers on the "undated bank" are forced to specific values and used in the computation of a code, which is checked.

To ensure that if the undated output "O" of the undated region of the circuit fails to be dated, all further results will be declared badly coded, we use a "destructive datation" : **if dating process fails, the date register is destroyed.**

The vital controller

For availability, the Vital Controller must be able to tolerate from time to time a bad code, but for safety the period of time between two consecutive corrections must be higher than a certain value. In order to do this the con-

troller must be able to reconstitute the code from the preceding one, by adding the difference of dates. The danger is to make this special mode the nominal mode.

Safety is garanteed by the destructive generation of the date's increment : **once it has been generated, it is safely destroyed and can no more be produced for a safely determined period of time.**

Conclusion

Our claim is that it is possible to perform safe code computations in an ASIC, and even that in some cases ASICs are more adapted to the safety constraints than software computations.

Bibliography

[1] W. Wesley PETERSON and E. J. WELDON, *Error-correcting codes (2nd Edition)*, The MIT Press, 1972

[2] T.R.N. RAO, *Error coding for arithmetic processors*, Academic Press, 1974

[3] Ph. FORIN, *Vital coded microprocessor principles and applications for various transit systems*, Proc. of the CCCT-89 (Paris, Sept. 89)

[4] R. LARDENNOIS, *Safety : single coded processor architecture combined with ASIC provide a cost efficient and flexible solution to safety issues*, IFAC symposium on Transportation Systems (Tianjin, PRC, August 94), Pergamon, 1994