On units of consistency.

(This is a replacement of EWD490 "Detection of malfetching of instructions", which never got completed.)

I am getting more and more worried about reliability. My natural pessimism got new food when I was told that the chips of pocket calculators are, in general, totally unchecked: the manufacturers don't know how to do it, and the rule is, therefore, that, if the customer complains, he gets a new one. Isn't that a simple solution.....

With regard to the reliability issue with large computers, we can try similar, but on closer inspection quite different things:

a) reduce the probability of an undetected machine malfunctioning

b) increase the probability that a correct result will be produced

c) decrease the propability that a wrong result will be produced as if it were a correct one.

Techniques primarily directed towards target (a) are parity checks and redundant execution of arithmetic operations. They are very popular, and their popularity is well-deserved. They are neutral --i.e. generally applicable-- built-in checks of great diagnostic value as far as hardware maintenance is concerned. They only serve the next two targets under the additional assumption that detected errors have a much greater probability of occurrence than undetected ones, an assumption which, alas, is often unjustified.

Techniques primarily directed towards target (b) combine --an effort at-- error detection with --an effort at-- error correction, under the motto "The show must go on." It may be a very local error correction, it may be a much more global recovery procedure. I have limited experience with such techniques, and, besides that, I have my doubts, because to the possibility of undetected errors the possibility of erroneously corrected errors has been added.

Target (c) is the most modest one: we sincerely hope, that the machine performs so well that the correct result will be produced and would be happy if each harmful machine malfunctioning would lead to abortion before the wrong result has been produced. The latter goal was the one discussed in

EWD482. We now return to it in a more restricted environment, viz. the problem of the fetching of instructions.

In order to simplify my discussion I shall assume that programs are represented with one instruction per word, and that each word is stored with a parity bit. (The following is also valid under the assumption of a more redundant word-wise redundancy.) The question is: how certain are we, that the correct instructions are executed, i.e. that the correct bit-patterns appear in the instruction register? There seem to be roughly three things that can go wrong:
a) while reading the instruction a multiple-bit error has occurred that has not been caught by the parity check
b) the instruction that reached the instruction register is the result of the perfect reading of the wrong word
c) the correct word has been read correctly, but the bit pattern has been corrupted somewhere on its way from the last checking station to the instruction register.

Error (b) can be caused in two different ways:
b1) the program is correctly stored, but the wrong word has been read because something happened in the instruction counter or on the way from instruction counter to memory selection register
b2) one of the instructions of the stored program has erroneously been overwritten (with a perfect parity of course!).

At least one system has been designed in which the instructions of a program are stored with a sum-check included, and, regularly, the machine interrupts its normal activities and inserts a sum-checking interlude in order to increase the confidence that all programs are still correctly stored. This approach, however, has a considerable number of unattractive consequences.
1) because the sum-checking interludes have to be inserted at the expense of the machine's productivity, we have to settle the conflict, how frequently these interludes should be inserted
2) in a paging machine, we have to perform the sum-check before a program page (of which I assume that it is not dumped) is overwritten by another page
3) it only catches error (b2), and if errors (a), (b1) and (c) can occur as well --and why shouldn't they?-- it is not very effective in view of our

target (c). (In all fairness I should mention that the system in which this technique was applied had been designed for message switching, and in that area the usual attitude is that anything may go wrong, provided that the system comes in the air again. Whether this is a realistic and defensible attitude, is a question beyond the scope of this note.)

The only way to catch errors (a), (b) and (c), all at the same time, is to perform the sum-check --or the embellishment of a cyclic redundancy check as is done with words from a tape-- on the successive contents of the instruction register, more precisely: each time after the instruction has been executed. (For the sake of simplicity I assume that during instruction execution the contents of the instruction register remains unchanged, knowing full well that there are machines for which this assumption does not hold: in that case the following can probably be made applicable by extracting for checking purposes the contents of the instruction register just before it is going to be modified. In the sequel we shall ignore the complication.)

The considerations in the previous paragraph tie the checking to the sequencing of the control; to do justice to this we introduce the notion of "a chunk":

> a chunk occupies a number of consecutively stored instructions, of which the first is always the only so-called "redundancy dummy" in the chunk, and the last one is the only jump-intruction in the chunk; each execution of the redundancy dummy of a chunk leads some time later to the execution of the terminal jump-instruction of that chunk --because no jump-instructions occur in the interior of a chunk; each execution of a terminal jump-instruction of a chunk has been preceded by a corresponding execution of the redundancy dummy at the beginning of that chunk, because no jump-instruction is expected to transfer control to anything else but a redundancy dummy of a chunk. Note. When for a conditional jump-instruction the condition is not satisfied and the jump-instruction acts like a skip, it is regarded "to transfer control" to the next instruction in store. (End of note.)

Behind the instruction register an extra register of the length of a word is introduced, the so-called Instruction Redundancy Register "IRR". Each instruction, as it has arrived and has been executed in the instruction register is added --without carry propagation, I presume, i.e. bit-wise modulo 2-- to the IRR, after IRR has been "rotated" over one place to the left; I have put "rotated" between quotes: a pure rotation means that the digit pushed out at the left is added at the least significant position at the right-hand side of IRR, but in a general "rotation" that addition may take place at a number of other bit-positions as well. The processing of a jump-instruction --which is always a terminal jump-instruction-- has two additional effects:

1) after the jump-instruction has been processed (its addition to the "rotated" IRR included) the contents of IRR is required to be all zeros; with only target (c) in mind, the machine could stop immediately.

2) the next word selected under control of the instruction counter and brought to the instruction register is by definition a redundancy dummy and, independent of its bit-pattern, its execution is further semantically equivalent to a skip.

The additional storage space occupied by each chunk is the redundancy dummy at its begin; because the interpretation of the redundancy dummy is independent of its bit pattern, the redundancy dummy can always be determined such that, when the chunk was entered with IRR = 0 , it will be left with IRR = 0. This is the reason why IRR has been chosen with the length of one word = the length of one instruction: it may not have more bits than the redundancy dummy, which was supposed to occupy one word.

Remark. Not rotating IRR would make the check insensitive to the interchange of any two instructions; pure rotation would make it insensitive to the interchange of two instructions for which the addresses of the locations occupied by them have a difference equal to the word-length. "Rotation" can rule that out. (End of remark.)

* * *

Since the above was written I have been reading in "Algebraic Coding Theory" by Elwyn R.Berlekamp (McGraw-Hill Book Company, New York, 1968), in particular chapter 4 "The Structure of Finite Fields"; for our purpose the most important mathematical result is the following. As operation on IRR we consider the following "rotating" step:

> the contents of IRR are shifted over one place to the left and the bit shifted out is added (modulo 2) at a well-chosen number of bit positions in IRR (among which the right-most one).

For any wordlength n the positions where the bit shifted out is added can be chosen in such a way that the step induces a cyclic order of the 2^n-1 IRR-contents $\neq 0$, in the sense that the step transforms each value into the next. Starting with a value $\neq 0$, we have to repeat the step 2^n-1 times, before the starting value reappears. The value 0 is obviously transformed by the step into itself.

The above made me understand why I was so much attracted by the "rotating" step as described above: it gives a great protection if there is a relatively large probability that while reading and executing a chunk two instructions will be corrupted in the <u>same</u> bit positions --with a parallel data path transporting the instructions from memory to instruction register, this is indeed a form of malfunctioning that should be considered-- . If we don't shift at all, two such equal corrputions will always cancel; in the case of a pure rotation, such a corruption has returned to its original position in IRR after every n steps, and, therefore, two identical misreadings will cancel if they occur in the same chunk n (or $2n$ or $3n$, etc.) words apart. With the proper choice of a "rotating" step we can ensure that this distance should be $2^n - 1$.

With $n = 30$, we have the probability $2^{-30} = 10^{-9}$ that the erroneous reading of a chunk will not be caught. If, for instance, half of the instructions of one chunk have been executed and the instruction counter contents is disturbed, and program execution is continued somewhere in another chunk, the probability that this error would escape the zero-check at the end of the second chunk is 10^{-9}.

<u>Note</u>. Under the (unrealistic) assumption that "skipping" an instruction --i.e. increasing the instruction counter by 2 instead of by 1-- is the only malfunctioning to guard against, our "rotating" step has a curious consequence. For each value of IRR there exists exactly <u>one</u> next instruction whose execution would not change the value of IRR. If that were the next instruction, skipping it would not be detected! And, if all our 10^9 instructions are equally probable, there would be a probability of 10^{-9} that skipping an instruction would remain undetected. There is, however, a remedy against that. Let at a given place Q be the instruction whose skipping would not be detected. If Q is the skip-instruction, we don't need to continue our text with it, but we may (skipping the skip-instruction

is a harmless error that we don't need to detect); if, however, Q is not the skip-instruction and it so happens that it is the next instruction with which we would like to continue our code, we can insert a skip-instruction. The execution of that skip-instruction would then change the contents of IRR and at the next place it is safe to place the instruction Q . The moral of this story is that either we accept the probability of 10^{-9} that the skipping of an instruction will escape our check, or once every 10^9 instructions our compiler would have to insert a skip-instruction! As for other reasons we have to be content with the probability of 10^{-9} for an undetected error, I would not burden a compiler with that obligation of inserting a skip; I only mention the result, because it is a remarkable one: I have never seen this form of trade off between reliability and storage space before. (End of note.)

* * *

The idea of this form of added protection to the reading of instructions emerged a year ago (December 1974, Tripreport EWD466:"on the way back I thought [...] on redundant object code representations. (To think again thoughts with a possibly direct bearing on machine design is great fun!) Bu the time I crossed the German/Dutch border I had arrived to a few firm conclusions. [...] I am not sure when I shall find the time to work this out and write a readable report about it."). Now it is eleven months later; the fact that the above was not written down earlier was, however, not the result of a lack of time, but was caused by the fact that I got stuck. As the design stands, there is one sequencing error that is not caught. At the end of the execution of a chunk it is checked whether IRR equals zero. (If the IRR-contents differ from zero but the check fails to discover it, the non-zero contents of IRR are "rotated" before the next redundancy dummy is added to it, so we have at the completion of the next chunk a second chance of detecting that IRR differs from all zeroes.) If, however, program execution now proceeds with the wrong chunk, it will not be detected! I tried all sorts of embellishments, but none of them was convincing. This report is written now, because I think that now I understand, why I failed.

The word is the minimum unit of consistency, and is, as such, protected by the parity check. Even if this check is effective --many parity bits per word, say-- , so effective in fact that the risk of undetected corruption internal to a word can be tolerated, error (b) of page 1: the perfect reading of the wronq word remains an unchecked risk.

For that reason a next larger unit of consistency is introduced, viz. the chunk. The linear execution of all the individual instructions of a chunk and the check upon IRR = 0 is the perfect analogy of the parity bit! (Compare it with the parity bit for a serial store, delivering the bits of a word one after the other.) I now tend to consider it as the main function of the chunk with its redundancy to guard against error (b); the fact that it also picks up most of the "left-overs" of the parity check is a nice accidental feature --on account of which we may tell ourselves that a single parity bit per word will suffice--, but it is not the essential one: it does as a matter of fact, blur the picture. (With many parity bits per instruction, we could have sent them all to the instruction register and have performed the parity check from there, after instruction execution.)

But now, IRR and the redundancy dummy per chunk appear as a completely independent check, that is active each time a whole chunk is executed. But just as this mechanism is independent of the parity bits --i.e. the internal protection of the next smaller unit of redundancy-- in the same way a mechanism checking that we don't have the perfect execution of the wronq chunks should be designed independent of the internal protection of the chunks themselves. There are a number of reasons in favour of this separation.

1) with a probability of 10^{-9} of a left-over --i.e. an undetected error of the internal chunk consistency-- I think that we can afford to forget about the left-overs

2) IRR and the redundancy dummy are neutral in the sense that they can be built into any machine that has an instruction counter and a jumpinstruction; I have considered additional techniques for checking the admissibility of the dynamic successor relation between chunks, but they turned out to assume some sequency discipline, at least more disciplined than wild goto's. Perhaps I find a solution without such assumptions; in any way it makes clear that checking the sequencing from chunk to chunk is a separate issue.

10th November 1975
Plataanstraat 5
NUENEN - 4565, the Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow