

Computer System Support for Scientific and Engineering Computation

Lecture 25 - July 26, 1988 (notes revised June 14, 1990)

Copyright ©1988 by W. Kahan and David Goldberg.
All rights reserved.

1 Retrospective Diagnostics

The IEEE standard categorically defines the numerical results of the basic operations of addition, subtraction, multiplication, division and square root, even for exceptional arguments like NaN and $\pm\infty$. For transcendental functions, it is up to implementors to define reasonable values, but this is not always easy. In lecture 24, we discussed 0^0 , and gave reasons why it might be reasonable to define $0^0 = 1$ as well as $\text{NaN}^0 = 1$. A more controversial example is $(-3.0)^\infty$. Since large floating point numbers which are integers are always even (large means bigger than β^{p+1} , where p is the precision), some would argue that $(-3.0)^\infty = +\infty$, whereas others would find this unconvincing. A good way to deal with these uncertain cases is to use *retrospective diagnostics*. In this way, the implementor can provide what he feels is the most reasonable value (such as $(-3.0)^\infty = +\infty$), but make a record in the log of retrospective diagnostics.

One problem with keeping a log of exceptions is that the log can get very large. Since users rarely have the patience to study long logs, failure to control the length of the log will seriously decrease its value. Retrospective diagnostics keep the log size small in two ways. First, a log entry is only made when an exception flag is changed from false to true. Thus if an exception is raised in a loop that is executed 1000 times, but the exception flag is not cleared within that loop, then a log entry will be made only the first time thru the loop (and not even then, if the flag was true upon entry to the loop).

But suppose the exception flag is cleared with the loop? Each exception has an exception type and a location (where the location may be only approximate), and these two values form a key for the exception record. When a second exception occurs with the same key, it overwrites the previous record. Hashing is one simple way to implement this scheme. If an exception occurs in a loop that is executed 1000 times, and the exception flag is cleared within the loop, the first entry in the log will be overwritten 999 times, but the log will always contain exactly one record for the exception. There will probably be more exception types for log entries than just the five IEEE exceptions. For example, if the implementor decides to set $(-3.0)^\infty = +\infty$ and log it, it is not clear which of the five IEEE exceptions applies: a new one would probably be more appropriate. The subroutine that evaluates x^y would call a library routine to make an entry in the log, in the case of controversial values of x^y .

Using this scheme, the size of the log is not much bigger than $\#(\text{flag types})(1 + \# \text{flag lowering sites})$, that is, the number of exception types multiplied by one plus the number of flag lowering sites. The "1" comes from the fact that all flags are implicitly lowered before a program starts execution.¹

We earlier discussed why precise interrupts are hard to implement. Retrospective diagnostics do not require precise interrupts. The location used as part of the key for a log entry does not have to be the exact PC where the exception occurred. It might be the last checkpoint since the exception occurred, where checkpointing is done in hardware or by the compiler. Or it might only be the name of the subroutine that caused the exception. The best system for the point of view of the programmer is to have a clear association between locations and source code statements.

Rudimentary retrospective diagnostics were implemented on the IBM 7094-II, and are described in the SHARE Secretary Distribution, SSD 159, C4537, pp 1-54.

2 Other Aspects of Exception Handling

Imagine giving data to a statistics program at a time when some of the data is not yet in hand and so is entered as NaN. In some cases, the program may never reference the NaNs, and so the program will complete uneventfully. However, if it does reference the NaN, the user specified trap handler can interactively query the user for the necessary data. Thus the user only needs to produce that data if needed, by which time it may have become available.

A simple example of a program that benefits from being able to manipulate the exception flags occurs in evaluating x^j , for j an integer. If $j < 0$, then $1/x^{-j}$ will be more accurate than $(1/x)^{-j}$. However, if this underflows, then x^j may overflow. So simply computing $1/x^{-j}$ could give a misleading and spurious exception. Thus a careful program would save the under/overflow flags, turn off under/overflow trapping, compute $1/x^{-j}$, and then check the under/overflow flags. If the flags are not set, everything is fine, and the program restores the flags and trap handlers. However, if the under or overflow flag is set, then the computation must be redone as $(1/x)^{-j}$ (after resetting the flags and trap handlers).

A feature that can be useful to users is an annunciator, which can be thought of as the modern day analogue of the flashing light on a panel, which blinks whenever the overflow bit is set. It could be implemented by blinking part of the display. Here's a hypothetical situation where the annunciator would be useful. John Doe buys a stock investment program for his home computer, which predicts the price of stocks using time series. One day, the time series routine overflows, resulting in the advice to sell IBM. If John sells IBM and then it goes up, he will be justifiably angry. However, if the screen flashed when it gave the advice to sell, John could call up the help line listed in his manual, and technical support personnel could examine his log of retrospective diagnostics to find out what went wrong.

2.1 Response to Exceptions

The IEEE standard gives two possible responses to an exception: trap or continue. However, a software environment might give the programmer more choices, such as

¹ The reason that the bound $\#(\text{flag types})(1 + \# \text{flag lowering sites})$ doesn't hold exactly is this. Suppose the top of a loop has a flag lowering site, but the flag can be raised at 5 different places in the loop. Then there could be as many as 5 different log entries, even though there is only one flag lowering site.

ABORT There are different degrees of abort. For example, the subroutine might abort, returning immediately to its caller. Or the entire process might abort.

PREMT Pre-emption. This is the programmers view of trap handlers. In BASIC, this is invoked using the `on error` statement.

DEFLT Do some default action. The IEEE standard gives defaults for all operations.

PAUSE This only makes sense in interactive systems. It might throw the user into a debugger, for example.

COUNT Under/overflow counting, as described in an earlier lecture.

PRESBS Presubstitution, as described in an earlier lecture.

2.2 Scope

There are two main strategies for the scope of exception handling: dynamic and lexical. Dynamic scope is what the Apple SANE package uses with `ProcEntry` and `ProcExit`. The state of flags is saved when the `ProcEntry` statement is executed. The user has to arrange his code so that `ProcExit` will be called to restore the flags.

APL uses the lexical approach, where saving and restoring occurs within the scope of a block. Unlike the dynamic approach, no matter how the flow of execution leaves the block, the state of flags is restored automatically: the programmer doesn't have to find each exit path and put a `ProcExit` there. However, the lexical approach requires compiler support.

3 The Value of $(1.0)^\infty$

We have discussed a number of (potentially) exceptional values of the power function x^y . One approach to deciding these questions is to examine $\lim_{z \rightarrow 0} f(z)^{g(z)}$ where $f(z)$ is an analytic function satisfying $\lim_{z \rightarrow 0} f(z) = x$ and $g(z)$ satisfies $\lim_{z \rightarrow 0} g(z) = y$. Let's try this technique on 0^0 . Then $f(z) = a_k z^k + a_{k+1} z^{k+1} + \dots$ and $g(z) = b_l z^l + b_{l+1} z^{l+1} + \dots$, with $k > 0$ and $l > 0$. Thus $\lim_{z \rightarrow 0} f(z)^{g(z)} = \lim_{z \rightarrow 0} z^{kz^l} = \lim_{z \rightarrow 0} e^{kz^l \log z} = 1$, since $\lim_{z \rightarrow 0} z^l \log z = 0$. Thus once again we get that $0^0 = 1$.

Trying this technique on $(1.0)^\infty$, we get $\lim_{z \rightarrow 0} f(z)^{g(z)} = \lim_{z \rightarrow 0} e^{g(z) \log f(z)}$. If $f(z) = z + 1$, and $g(z) = k/z$, then the limit is e^k . In other words, the limit depends on the exact function $f(z)$ and $g(z)$ unlike the 0^0 case where the limit is always 0 independent of f and g . Thus we can argue that $(1.0)^\infty$ should be NaN and signal INVALID OPERATION.

12
FLOATING-POINT ARITHMETIC

EXCEPTIONS

AND

RETROSPECTIVE RUN-TIME
DIAGNOSTICS.

Prof. W. Kahan
Univ. of Calif. @ Berkeley

26 July 88

EXCEPTION HANDLING

1. An EXCEPTION is not an ERROR unless handled badly.
2. What makes it EXCEPTIONAL is that NO UNIFORM POLICY ADOPTED IN ADVANCE would be universally acceptable; someone would take exception to that policy's application to his exceptions, and with good reason.
3. DIVERSE APPROACHES are needed; none are acceptable in the long run unless compatible with ...
CONCURRENT ARITHMETIC OPERATIONS,
PIPELINES, VECTORIZED & PARALLEL MACHINES,
IMPRECISE INTERRUPTS.
4. EXCEPTION HANDLING may have to be implemented in ways INDEPENDENT OF PROGRAMMING LANGUAGES because COMPILER-WRITERS WON'T HELP.
(cf. APPLE's S.A.N.E., all in run-time library.)
5. ... BUT PROGRAMMING LANGUAGES HAVE THEIR OWN BENIGHTED IDEAS ABOUT EXCEPTION HANDLING, and we must respect them.
6. LANGUAGE DESIGNERS & IMPLEMENTERS could help exception-handling in 2 areas:
 - (i) SCOPE of MODES ...
 - (ii) RUN-TIME EFFICIENCY.

```

ARCCOS(X);
  ATOMIC;
  SAVE  DIV2  FLAG;

```

$$A := 2 \arctan \sqrt{\frac{1-X}{1+X}} ;$$

```

  RESTORE  DIV2  FLAG;

```

```

RETURN  A

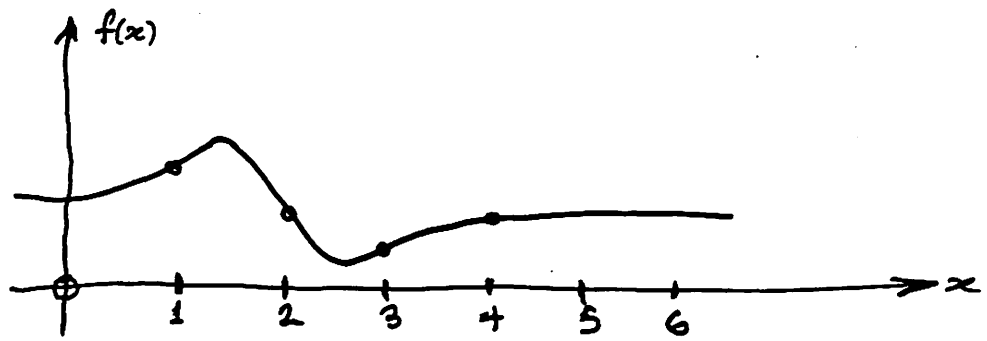
```

SIGNALS INVALID if $|x| > 1$.

NO SIGNAL if $x = -1$.

EXACT WHEN APPROPRIATE

NO TESTS & BRANCHES!



$$cf(x) := 4 - \frac{3}{x - 2 - 1}$$

4 ÷
4 ... / 0

(nonzero)/0 = ±∞
(finite)/∞ = ±0

$$\frac{x - 7 + 10}{x - 2 - \frac{2}{x - 3}}$$

$$rf(x) := \frac{622 - x(751 - x(324 - x(59 - 4x)))}{112 - x(151 - x(72 - x(14 - x)))}$$

7 x
1 ÷
Overflow
100% roundoff

$$f(x) := a_0 + \frac{b_0}{x + a_1 + \frac{b_1}{x + a_2 + \frac{b_2}{x + \dots + \frac{b_N}{x + a_N}}}}$$

$$f := a_N ;$$

for j = N-1 to 0 step -1 do

$$f := a_j + b_j / (x + f) ;$$

... f(x) = f now.

SIMPLICITY!
∞

NOT IBM '370
NOT VAX after DEC

IEEE 754, 854 (sun, pc, ...)
VAX after D. Barnett, Berkeley
CDC ? CRAY ?

PRESUBSTITUTION

```

 $\lambda := \dots$  ;
FOR 0/0 PRESUBSTITUTE  $\lambda$  ;
FOR J = 1 TO N DO ... in parallel
     $y_J := \sin(\lambda x_J) / \sinh(x_J)$  ;
REPEAL PRESUBSTITUTION

```

```

FOR 0/0 OR  $\infty/\infty$  PRESUBSTITUTE  $\infty$  ;
 $f' := 0$  ;  $f := a_N$  ;
FOR J = N-1 TO 0 STEP -1 DO
    {  $d := x + f$  ;  $d' := 1 + f'$  ;
       $q := b_J / d$  ;
       $f' := -(d'/d)q$  ;  $f := a_J + q$  ;
      FOR  $\infty/\infty$  PRESUBSTITUTE  $b_{J-1} \cdot d' / b_J$  } .
... Now  $f =$  continued fraction  $f(x)$ ,
...  $f' = df(x)/dx$ .

```

Without Presubstitution:

```

 $f' := 0$  ;  $f := a_N$ 
FOR J = N-1 TO 0 STEP -1 DO
    {  $d := x + f$  ;  $d' := 1 + f' + \epsilon$  ;
       $q := b_J / d$  ;
       $f' :=$  IF  $|d'| = \infty$  THEN p
              ELSE  $-(q/d)d'$  ;
       $f := a_J + q$  ;  $p := b_{J-1} \cdot d' / b_J$  } .
... Now  $f =$  continued fraction  $f(x)$ ,
...  $f' = df(x)/dx$ .

```


TO SOLVE $f(x)=0$ for x :

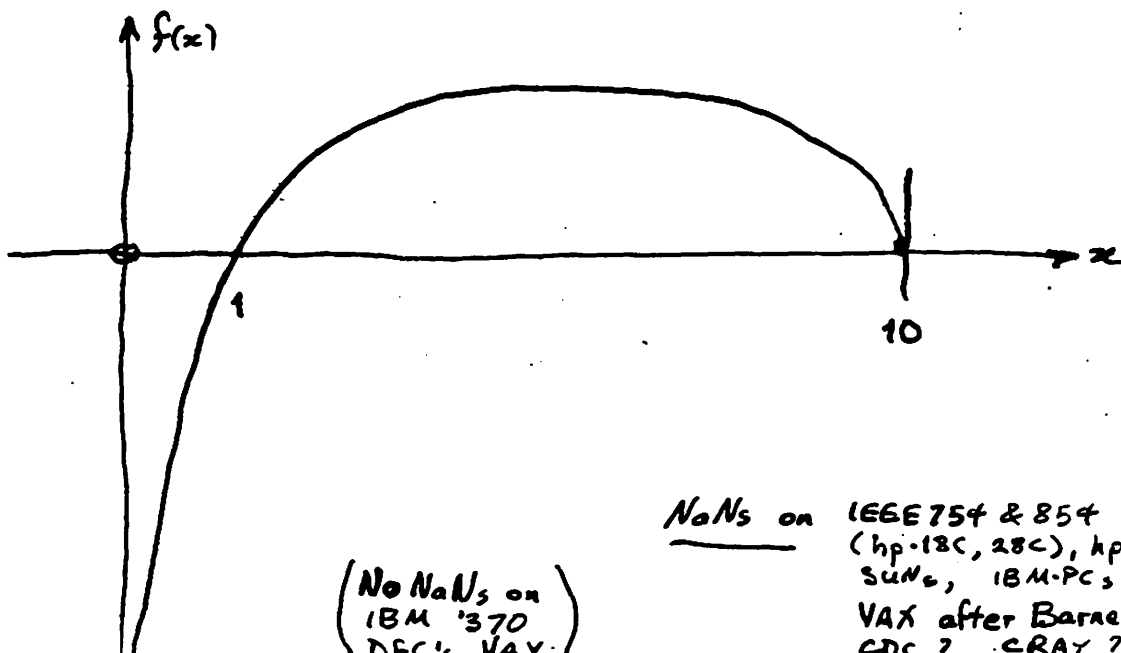
MAIN

```
...  
EXTERNAL REAL f (REAL) ;  
LIBRARY PROCEDURE SOLVER (REALFUNCTION, REAL);  
...  
x := ... GUESS ...  
CALL SOLVER (f, x) ;  
... USE x ...
```

SOLVER (REAL lhs (REAL), REAL arg) :
... start with 1st guess at arg ;
v := lhs (arg) ;
... change arg and go back
or quit.

lhs = f
arg = x

REAL FUNCTION f (REAL x) :
RETURN f := $\ln(x) \cdot \sqrt{10-x}$.



NaNs

for $k=1$ to N do ... in parallel or overlapped
if $y_k/x_k > 3$ and $|x_k| > |y_k|^3$
then $z_k := \sqrt{z_k}$;

Without "if":

for $k=1$ to N do ... in parallel or overlapped
{ $b_k := (y_k/x_k > 3) \text{ and } (|x_k| > |y_k|^3)$;
 $r_k := \sqrt{z_k}$;

SELECT $z_k := \text{if } b_k \text{ then } r_k \text{ else } z_k$ } .

WITHOUT NaNs nor OOs:

fodom := FALSE

for $k=1$ to N do ... in parallel or overlapped
{ $b_k := (x_k = 0)$; $bz_k := (z_k < 0)$;
 $x1_k := \text{if } b_k \text{ then } 1 \text{ else } x_k$;
 $y1_k := \text{if } b_k \text{ then } 1 \text{ else } y_k$;
 $z1_k := \text{if } bz_k \text{ then } 1 \text{ else } z_k$;
 $b_k := (y1_k/x1_k > 3) \text{ and } (|x1_k| > |y1_k|^3)$;
 $r_k := \sqrt{z1_k}$;
 $z_k := \text{if } b_k \text{ then } r_k \text{ else } z_k$;
fodom := fodom or (bz_k and b_k) } .

UP-DOWN COUNTING OF OVER/UNDERFLOW.

$$Q := \prod_i (A_i + B_i) / \prod_j (C_j + D_j)$$

COUNTOVER_UNDERFLOWS_IN (K) ; up-down counter.

→ K := 0 ;

Q := 1.0 ;

for j = ... do Q := Q * (C_j + D_j) ;

→ K := -K ; Q := 1/Q ;

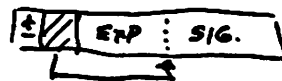
for i = ... do Q := Q * (A_i + B_i) ;

... Now $Q = 2^{-K \cdot L} \cdot \prod_i (A_i + B_i) / \prod_j (C_j + D_j)$

→ if K ≠ 0 then

On IBM 7094 @ Toronto (1962)
Burroughs B5500 @ Stanford (1966)
IBM 360 @ Waterloo (1967)
VAX @ Berkeley (Barnett, 1987)

cf: Clenshaw-Olver "Level-index Arithmetic" } DYNAMIC
Matsui & Iri } RANGE-
HAMADA } PRECISION
TRADE-OFF

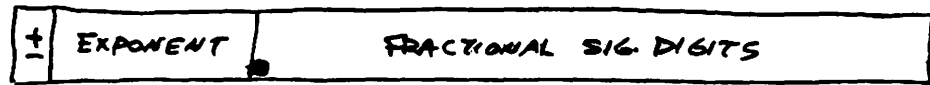


THESE ALL LOSE PRECISION !

Goldberg's Variation (Comm. A.S.M., 1960)

for BINARY FLOATING-POINT:

(DEC PDP-11, VAX)
(IEEE 754)



$$\pm 2^{\text{EXPONENT} - \text{BIAS}} \times (1.\text{FRACTIONAL SIG. DIGITS})$$

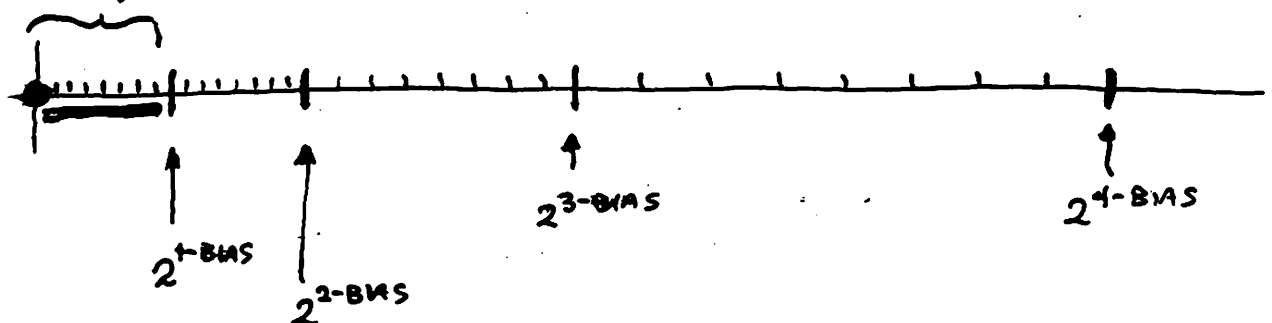
↑ "HIDDEN BIT"

GRADUAL UNDERFLOW:

(NOT DEC ...)

When EXPONENT = Minimum exponent ... 0 usually
then $\pm 2^{\text{Minimum exponent} - \text{BIAS} + 1} \times (\text{FRACTIONAL SIG. DIGITS})$

SUBNORMAL (De-normalized) NUMBERS (IEEE 754)



$$|\text{Error due to Gradual Underflow}| \leq |\text{Roundoff elsewhere}|.$$

$$|\text{Error due to Flush-to-zero}| \gg |\text{Roundoff elsewhere}|.$$

cf. J. Demmel (1984) SIAM J. Sci. Stat. Comp.


HAZARD with Gradual Underflow:

MUST TEST UNDERFLOW FLAG,

not test for zero,

to detect underflow's loss of accuracy:

e.g. $0.0000314_{10} \cdot 10^{-99}$ instead of $3.1415926_{10} \cdot 10^{-104}$.

<u>EXCEPTION</u>		<u>EXCEPTIONAL VALUE</u>
ALLXS	all of them (array)	
* OVFL0	exponent OvERFlow	$\pm \infty$ or $\pm \text{HUGE}$
* DIVBZ	(nonzero)/0.0, <u>LN(0.0)</u> , ...	$\pm \infty$ or $\pm \text{HUGE}$
* UNFLO	exponent UNderFlow	Gradual? or 0
* INXCT	Inexact !	rounded result
INTXR	Integer exception or error	?
* INVLD	<u>INVALID operations</u>	NaN
ZOVRR	0.0/0.0	NaN
IOVRI	∞ / ∞	NaN
INDIV	one of 	NaN
ZTMSJ	0.0 * ∞	NaN
IMINI	$\infty - \infty$	NaN
FODOM	... $\sqrt{-3}$	NaN
UNDTA	Uninitialized Data	NaN
DTSTR	Outside Data Structure	NaN
NLPTR	NULL Pointer	NaN

EACH KIND OF EXCEPTION HAS ITS FLAG,
WHICH A PROGRAM CAN TEST, SAVE, RESTORE, ...

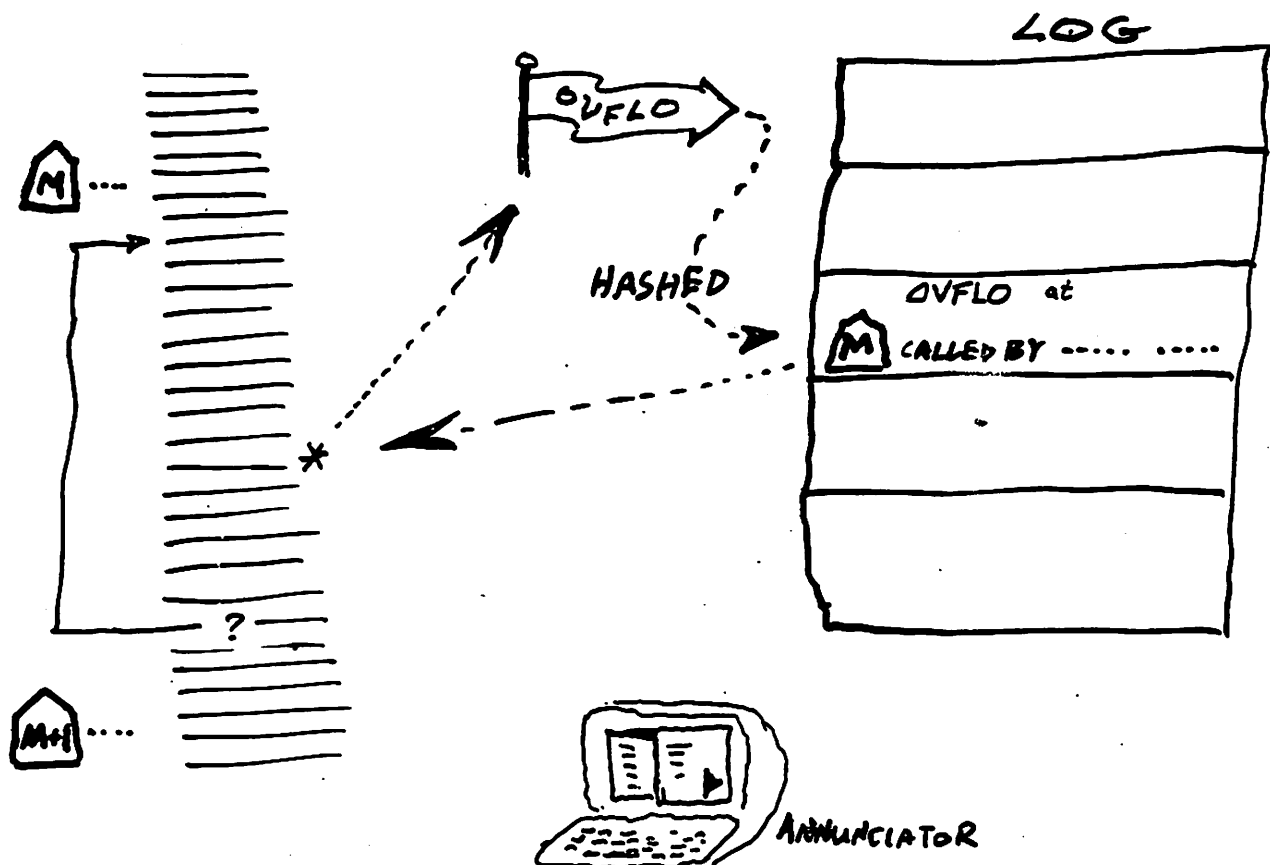
* IEEE 754, 854

SIDE-EFFECTS : FLAGS get raised !

Provably UNAVOIDABLE despite FUNCTIONAL PROGRAMMING,
DATA-FLOW, OCCAM, ...

RETROSPECTIVE DIAGNOSTICS

to identify SITES of LEADING UNREQUIRED EXCEPTIONS:



Raising FLAG disables subsequent signals (traps, logging,...) ;
lowering FLAG enables 1st subsequent signal.

- °° $\#(\text{LOG entries}) \leq (\# \text{FLAG-TYPES}) \times (1 + \#(\text{FLAG-lowering sites}))$.
- °° LOG cannot grow unmanageably.

On IBM 7094-II @ TORONTO

cf. SHARE SSD #159 (Dec. 1966) Item C-4537 .

" IF KICKED (OFF) ... " statement put into our Fortran.

Implementation Notes

- Mostly implementable in the RUN-TIME LIBRARY, independent of COMPILER and OPERATING SYS.

Hardware: needs Presubstitutable registers in place of wired-in ∞ , NaN, trap,...

& either QUEUE for signals & flags,
or QUICKLY-TESTED STATUS-CHANGE bit

Compiler: must generate MILESTONES
or QUICK-TESTS for status-change
must respect SCOPING RULES for modes

Operating System: Must allow for ANNUNCIATORS
in I/O drivers

- Must provide for LOG to be written ONTO or AFTER standard error file
- Must allow interactive interrogation of LOG

Everything else is part of RUN-TIME LIBRARY.

Modes of response to exceptions:

ABORT	...subroutine, module, task, ...
PREMT	Preemption { "ON ERROR GO ..." } OVERFLOW ∈ { ARITHMETIC ERRORS }
DEFLT	IEEE 754 Defaults
PAUSE ... & RESUME ...	INTERACTIVE DEBUGGING
COUNT	OVER/UNDERFLOWS UP/DOWN
PRSUBS	... PRESUBSTITUTION

SCOPE of MODES

- "Flat" or "Dynamic" scope: (cf. APPLE's SANE)
 - uses run-time library.
 - programmer must save & restore modes.
- "Lexical" or Language-Assisted Scope
 - cf. APL's
LOCALIZATION OF SYSTEM VARIABLES
- Recognition of Bottom-level (Leaf-) procedures
Simulation of Atomic Operations