Computer System Support for Scientific and Engineering Computation

Lecture 23 - July 19, 1988 (notes revised June 14, 1990)

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

1 Floating-Point Exceptions (Continued)

1.1 Cray

The Cray has 64 bit words, and its single precision floating point format has 1 sign bit, 15 exponent bits and 48 bits of significand. The exponent bias is 2^{14} . Cray manuals use octal notation, so that they write the bias as 40000_8 . Whenever the biased exponent is less than 20000_8 , the result is flushed to 0, and whenever the biased exponent is greater than 60000_8 , the number becomes an "indefinite". So there are effectively only 14 exponent bits.

By default, a program halts execution when an indefinite (a number of the form $6xxxx_8$) is created. The indefinite has encoded in it the operation that caused the exception. It is possible to change the default so that computation continues, with the indefinite being propagated like NaN's. However, unlike IEEE arithmetic, which has three kinds of indefinites $(-\infty, +\infty, \text{ and NaN})$ with a careful set of rules for how they propagate thru arithmetic operations, the Cray's indefinites do not combine in a useful way.

Cray expects exceptions to be handled by not getting any! For this purpose, Cray lets you provide a bit vector that specifies elements of a vector on which no operation should be performed. The intended use of this facility is to inhibit division by zero, so that exceptions can be avoided.

1.2 Apple

Apple has a very complete set of primitive functions, language extensions and library routines called SANE that work across their entire product line (except UNIX on the Mac II). SANE has the three precisions single, double, extended, and always computes intermediate results in extended precision (assignments implement rounding to narrower precision). Thus the primary reason for providing three types is to control the amount of storage. SANE includes routines for setting and saving rounding modes, rounding precision, exception flags, and trap enables. Users can install their own trap handler(s). When a trap handler is called, it is given the operation and the addresses of the operands and result, so that it can modify them. Up to this point, SANE has the same flavor (although more carefully done) as exception handling on the VAX and IBM/370. The new element that SANE provides is a pair of routines called ProcEntry and ProcExit. ProcEntry saves the current floating point environment, and resets the environment to the IEEE default. ProcExit restores the environment, adding to it any flags that were set at the time of the ProcExit call (and causing corresponding halts if they are enabled). An example of its use is

```
myprog()
ProcEntry(myCallerEnvironment)
...
y = 2*arctan(sqrt((x+1)/(x-1)))
SetException(DivByZero, false)
ProcExit(myCallerEnvironment)
```

If x is 1, then there will be a divide by zero exception, but y will get the correct value of π , so the divide-by-zero flag is cleared. Any other exception that occurs (for example an invalid exception if x is -3) will be passed on to the calling program. Although ProcEntry and ProcExit are very useful, they could become inefficient if in a long chain of procedure calls, each procedure invoked ProcEntry and ProcExit.

On machines with a Motorola 68881, exception handling is the same, but programmers must use the 68881's trap handling mechanism rather than the software trap mechanism. By default, transcendental functions are computed in software. Users desiring more speed can ask for the on-chip transcendentals, which are faster but less accurate.

SANE is designed to require the minimum number of changes to the host language, but some are unavoidable. For example, standard Pascal has only one floating point type, which isn't sufficient to exploit SANE's three precisions. Compilers must also be altered to cope with IEEE comparison involving NaNs and with I/O of NaNs and infinities.

In summary, SANE provides a portable floating point environment with little overhead, but it appears that its exception handling and trapping features are not heavily used by customers, perhaps because most are disinclined to change their habits developed elsewhere.

1.3 APL

A slightly different strategy from the ProcEntry approach is used in STSC APL for the IBM PC, to handle a similar problem. APL has a global variable called CT which controls the tolerance for comparisons. When comparing two numbers, the comparison will be true if it holds within this tolerance. This has strange consequences. For example, you can have x=y and y=z but not x=z!

In STSC APL, you can declare CT to be local to a procedure, and it will have a "copy-onwrite" behavior. It initially has the same value as the global CT, but if it is modified in the subroutine, it will affect the compare tolerance only in the procedure, and not change the global value of CT. Thus if you write to the local CT, you will get a local copy. This is a nicer model for the programmer than ProcEntry, but it does require changing the compiler. There is a fine point connected to associating complex operations with an assignment. Suppose you have the code fragment

C = A*B; RndMode = Up D = A*B;

Suppose the assignment to RndMode has the side effect of changing the rounding mode. An over-optimizing compiler might see the common subexpression A*B and avoid the second multiply, not realizing that it may have a different value because the rounding mode has changed.

2 Precise Interrupts

When discussing exceptions, it is useful to have the concept of a *precise interrupt*. Consider the instruction sequence

100	Ε	=	Ε	+	1
200	A	8	B] .	С
300	В	a	D	+	F

Suppose that the division operation causes an overflow exception. We would like the trap handler to see the state of the machine as if all the instructions preceeding the division had completed, and none of the instructions following the division had initiated. Thus if the trap handler accesses E, it should have its value incremented by statement 100, and if it accesses B, it should not have had its value destroyed by statement 300.

There are two main obstacles to precise interrupts. The first is the compiler. An optimizer might have decided to move statement 100 after statement 200. The second is the hardware. The hardware might perform statements 200 and 300 in parallel, and by the time the division operation overflows, statement 300 could have completed and overwritten B. To some extent this is similar to what happens with virtual memory systems, where an instruction can fault because the data it is accessing has been paged out. However, in this case the fault handler is specified by the system and is going to do a very predictable thing, namely bring the data into memory. User-specified trap handlers for floating point show much more diversity in their behavior.

The DEC VAX family has a further distinction between *traps* and *faults*. A fault occurs (logically) before the offending instruction executes, and so returning from the exception handler will execute the instruction that caused the fault. A trap occurs after the offending instruction executes, so returning from the exception handler will begin with the instruction after the offending instruction.

Machines that have overlapped instruction execution (due to pipelining or multiple floating point units) often simulate precise interrupts with extra hardware. One approach is to take checkpoints, and when an exception occurs roll back to the last checkpoint, proceeding from the checkpoint one instruction at a time. Another approach is to have shadow registers to save the state of registers that might be needed later by the trap handler. The approach taken by the HP spectrum is to ensure that the hardware never overwrites the arguments to a floating point operation until it is sure that operations will not trap. In each of these schemes, extra hardware is required to guarantee precise interrupts.



Standard Apple Numeric Environment

14 July 1988 Chyton Lewis

SANE Data Types

É Single

 Precision 7-8
 Range 10
 $^{-45}$ to 10^{-38}

 1
 8
 23
 = 32 bits

É Double

 Precision 15 - 16 Range 10^{-324} to 10^{-308}

 1
 11
 52
 = 64 bits

🗳 Extended

Precision 19 - 20 Range 10⁻⁴⁹⁵¹ to 10⁴⁹³²

		· · · · · · · · · · · · · · · · · · ·	
1	15	64	= 80/96 bits
	6888	31 format stores 16-l	oits of junk bere



* Range and Precision are decimal approximations

SANE Operations



random numbers

<u>Apple Numerics Group</u> <u>SANE in High Level Languages</u>

É SANE languages support the abstract environment:



SANE languages offer:

 \rightarrow Minimum impact on language standards

SANE features not expressible in the minimallymodified language are placed in a library

 \Rightarrow Old sources run (more accurately)

Expressions evaluated in extended precision

Data types, operations, and environment under full user control

 \Rightarrow I/O supports NaNs and Infinities

Source constants held in extended precision



SANE Availability

• SANE is available on all Apple machines



• SANE is available in software and in bardware

Software on all machines without FPU Hybrid on machines with FPU

Direct 68881 on machines with FPU

• SANE is available in high-level languages



SANE Library Services



NextExtended

Ę).	Envir	onmental	l access	routines
-----	-------	----------	----------	----------

l										***	***	***					***		***	***												***																					\sim
			27	5	299		- 2	•	8	~		~	-		8						,,	20		Ψ.	5		***	99								C.		1		88		**											
		2	24		U,	ν.	"	۰C	11		4		43										2		-	24	1.		У			**					Е.	2		٨.	1.		S	4	1	12							
*							82										8				<u></u>				***														-						8								
		6	e	1	37	77	11	7	01	21	11	Y.	11	18						88	P	11	10	1	1	1	¥.	1							*	6	\mathcal{C}	U,		18	e			Y.	О	Ľ							
				Ú.																					**		ŵ			*																							
		~		÷.,	<u>)</u>		۲			2						8			-	-	2	22										20	-	2						***		~		÷	-	0	87	-	÷_				
		ЭX	-U		X	è	e	D,	20	U	11					 \mathbf{D}	r	2	L	16	7.	u							1	ъX	Ы	9	1	0	\mathcal{L}	11	9					SY.	4	9	7.	a	u	Y	e	С	U)7	
8										8									*		2												2												4								
		L	es	19	Ľ		C	e	D	U	0	11			8		C	28	U	L	1	а	u			***		×		6	e	L	1	G	n	11	U	4				8	e	Ľ.		0	U	L	Υŧ	20	L	01	r



Relation

Logb

14 July 1988

ClassExtended

Apple Numerics Group

SANE Exceptions

What happens when an exception occurs

Flags are raised



Corresponding halts are checked

Invalid Overflow Underflow Division-by-zero Inexact

É A reasonable value is returned

 \odot

Closest machine representation NaN (Not a Number) Infinity

É Examples

$\sqrt{-3}$	-	NaN	(invalid)	
8/0	-	Inf	(division-by-ze	ero)
0/0	-	NaN	(invalid)	
2*maxvalue	-	Inf	(overflow, ine:	xact)
minvalue/2	-	0	(underflow, in	exact)
1/3	-	· <i>333333</i>	3333333333333333	(inexact)



Control after Exceptions

* Software SANE (software-based)

User sets and queries current balt vector

Information is placed on the stack Operation Arguments (with result already delivered) Exceptions from this operation

* Hardware SANE (68881-based)

User sets and queries current balt vectors

The 68881 trapping model is followed (programs run in supervisor mode)