INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

H4.SMR. 403/23

FIFTH COLLEGE ON MICROPROCESSORS: TECHNOLOGY AND APPLICATIONS IN PHYSICS

2 - 27 October 1989

Floating Point Numbers

P. BARTHOLDI
Observatory of Geneva, Sauverny, Switzerland

Floating Point Numbers

Paul Bartholdi
Observatory of Geneva
CH-1290 Sauverny
Switzerland

Preliminary notes - Trieste - october 1989

Fifth College on Microprocessors

Contents

1 Introduction, what is the problem

In mathematics, we have the following sets of numbers :

Natural	$\mathcal{N}$	7 , 13
Rational	$\mathcal{Q}$	2/3 , 3/7
Real	$\mathcal{R}$	1.4 , π , $\sqrt{2}$
Complex	$\mathcal{C}$	$7 - i\sqrt{3}$

Remarks :

- All sets are unbounded, very regular and easily defined (cf. Peano)
- Arithmetic operations behave in a uniform way
- We may add $\pm\infty$ and/or exclude 0 as divisor

In computers, we have :

Integers	*integer i, j, a*	$-127 \ldots 128$
Packed decimal	*packed_decimal salary*	17846393275
Floating point	*real*4 weight, surface*	3.2433 , 0.0014
Extended/multiple precision	*real*16 speed, mass*	3.14159265358979323846

- All sets are of limited range and coverage.
- Arithmetic operations behave non uniformly, rules are not obeyed (irreversible etc).
- We may add ± 0 , $\pm\infty$, $\mathcal{NAN}$ etc.

The problem comes from the fact that the number of bits (or bytes) available to represent numbers is strictly limited, usually fixed and not dynamically extendable.

Remember : with n bits, you can represent 2^n numbers.

In the real world, we have :

- data and results rarely exceeding 1% precision
- (very) large dynamic of numbers, due to units used
  (ex. the mass of the Earth is 59850 00000 00000 00000 00000 Kg)
- complex arithmetic computations (ex. transcendental functions)
- some *ill-posed* problems, that we would like nevertheless to solve

We also want situation independent programmes.

n	2^n	$0 \ldots m$	$-k \ldots l$
8	256	0 ... 255	-128 ... 127
10	1024	0 ... 1023	-512 ... 511
16	65536	0 ... 65535	-32768 ... 32767
24	167772216	0 ... 1677216	-8388608 ... 8388607
32	4294967296	0 ... 4294967296	-2147483647 ... 2147483648

Figure 1: Range of integers with various word length

2 Solutions ?

1. Carefully scaled integer arithmetic
   - can be very fast and precise
   - may need well planned programming and good problem understanding by programmer and user !
2. General purpose floating point arithmetic
   - usually (much) slower, use more memory
   - can be used almost blindly
   - good library are available
   - rounding and truncating effects are more difficult to ascertain

3 Scaled integer arithmetic

1. Adapt units in such a way that all numbers are close to 1
2. Get an upper limit for all numbers operated on : $|m| < 2^N$ (if possible, choose $N = 0, 1$ *or* $2 \ldots$)
3. Scale your numbers, (multiply them by 2^k) in such a way that $2^{k+N} - 1$ is the largest representable integer in your computer
   NB: you may have different k in different part of your programme, k may also be adjusted during the computation.
4. Use normal operations for addition and subtraction
   If you get an overflow, decrease k by one, and adjust all necessary numbers.
5. Use double (extended) precision result multiplication in *registers*, followed immediately by division, or scaled by 2^{-k} (right shift k bits). Keep low order bits. If you get an overflow, decrease k and adjust all necessary numbers.

Remarks :

- **Add and Sub are usually fast**
- **Mul is much slower and may produce double precision**
  - What do we keep ?
  - What do we through away ?
- **Div is still slower, and should start with double precision dividend**
  Whenever possible, replace :

Mul	by	left shifts
Div		right shifts

  and avoïd calculating unwanted partial results (lower of higher order bytes)
- Low precision functions are best dealt with tables (in ROM ?), with as many entries as parameter values.
  Ex. trigonometric functions
  1. scale angles in such a way that 360° is represented by 256
  2. have 256 *sin* and *cos* tables
     1 memory fetch per function evaluation, or
  3. have 64 (0 – 90°) or even 32 (0 – 45°) entries, and use high order bits for sign and table.

3.1 Note on modulo arithmetic

- Usually, integer arithmetic operations work modulo 2^{size}
  you do not need extra operators for them
- Modulo is natural for many variables (ex angles)
- If high order bits are constant throughout calculation, they can be ignored during them, and reset at the end. ex mean, deviation etc.

3.2 Pro and Con

Good :
- Very fast, even on microprocessors
- You get what you want
- truncation effects are visible and manageable
- it use all available bits for precision

Bad :
- Programme must be adapted to your problem, it may be difficult to use again
- large dynamic is intractable
- large memory may be necessary for tables (algorithms need less memory only for high precision)

Best for :
- Real-time control
- signal processing
- FFT
- filtering
- graphics

4 Floating point arithmetic

4.1 Floating point software libraries

Many very good software libraries are now available, some at very low cost, for almost all numerical problems. They use the best known strategies for fast and accurate computation. They also represent many tens or hundred man years of effort, an effort that should not be replicated unnecessarely.

Among the best :

- look at the book by Cody for the elementary functions (trigonometric, exp etc)
- complete general purpose libraries (very good, expensive) :

  NAG *Numerical Algorithms Group Ltd*
  NAG Central Office
  Mayfield House
  256 Banbury Road
  Oxford OX2 7DE , UK

  IMSL *International Mathematical and Statistical Library*
  NBC Building, 7500 Bellaire Boulevard
  Houston, Texas 77036, USA

- Numerical recipes

  A book (25£+ shipping) about good numerical methods
  - when and why to choose one or an other
  - example (short qith results)
  - code for all routines in fortran 77 and pascal

  A second book with examples – code in fortran 77 or in pascal (only on choice)

  A floppy or tape with all codes (routines & examples)

  - It is very practical and usefull, but not good, nor sufficient to learn about numerical analysis.
  - It should be considered as a compagnion book

- specific libraries

  B-SPLINE spline interpolation
  EISPACK eigenvalue/vector, matrix inversion,decomposition etc.
  ELEFUNT elementary functions
  ELLPACK partial differential elliptical equations
  LINPACK linear equations
  LLSQ linear least square problems
  MINPACK function minimization
  QUADPACK integration

  They are all available at the cost of support through IMSL (see above)

- ACM Collected Algorithms, all kinds of algorithms, good and bad, also available through IMSL

- digital signal processing

IEEE complete set of algorithms, from basic fft routines to complex filter design, available through :

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47 Street
New York, NY 10017, USA

or through John Wiley and Sons, Inc.

4.2 Floating point representation

A floating point number ($\mathcal{FP}$) is very similar to the so called "scientific notation" : ex $6.359 \cdot 10^{-19}$
We can always represent them in the form :

$$s \cdot f \cdot \beta^{e-b}$$

where :

s	sign of the fraction
f	fraction or mantissa
e	exponent or characteristic
β	implied base and decimal position
se	sign of exponent, or
b	implied bias of exponent

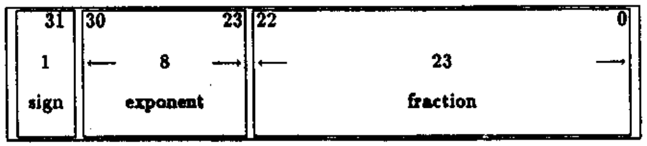


Figure 2: Floating point number inside a 32 bit word

- for more dynamic : wider exponent
- for more precision : wider fraction (double word)

The fraction is usually normalised :

$$\frac{1}{\beta} \leq f < 1$$

or

$$1 \leq f < \beta$$

Typically, $\beta = 2, 10$ or 16

In some cases, when $\beta = 2$ the most significant bit of the fraction (the one to the left) which is always 1 anyway, is implicit and not represented in the computer. This increase the precision without an extra bit.

4.3 Range and precision

The range is defined as the set of all numbers that can be represented, from the smallest to the largest, irrespectively of the precision.
The smallest and largest absolute representable numbers are :

$$Min = \beta^{e_{min}} \cdot f_{min}$$
$$Max = \beta^{e_{max}} \cdot f_{max}$$

Typically : $f_{min} = 1/\beta$ and $f_{max} \sim 1$

For example, if $\beta = 2$, $e_{min} = -64$ $e_{max} = 63$

$$Min \simeq 4 \cdot 10^{-78}, \quad Max \simeq 7 \cdot 10^{77}$$

Note that Min and Max may differ when positive or negative.

The relative precision is given by the smallest non-zero difference between two fractions. Typically, if f is made of 24 bits (including the possible implicit most significant bit), then the precision $\epsilon \simeq 0.6 \cdot 10^{-7}$

The fraction has a fixed number of bits $\longrightarrow$ the relative precision is constant, whatever the exponent.
Is it always meaningful ? NO

When you subtract 2 similar numbers, significant bits are lost :

operand	relative error
$0.3141592 \cdot 10^1$	10^{-7}
$-0.3141000 \cdot 10^1$	10^{-7}
$0.5920000 \cdot 10^{-3}$	10^{-3}

The zeros at the right of the result are not significant !

5 Floating point operations

5.1 Normalize a number

Basic algorithm :

```
While MSB of fraction = 0
do      shift_left fraction one position
        subtract one to exponent
end
```

Example in binary ($\beta = 2$)

e		f
7		0.001011...
	-1	
6		0.01011...
	-1	
5		0.1011...

NB : The MSB (Most significant bit) may be implicit, see above.

Example in decimal ($\beta = 10$)

	0.0041	10^5	
* 10			÷ 10
	0.0410	10^4	
* 10			÷ 10
	0.4100	10^3	
for normalized unity			
* 10			÷ 10
	4.1000	10^2	

NB: The value is unchanged, only the representation is different.

5.2 Multiplication $F_R = F_1 * F_2$

`Multiply fractions`	$f_R = f_1 * f_2$
`Add exponents`	$e_R = e_1 + e_2$
`Renormalize (if necessary)`	
`Add signs modulo 2`	$s_R = s_1 \oplus s_2$

The division is similar

Example of multiplication in decimal :

$(-1)^0 \cdot 0.4 \cdot 10^3$	400
$* (-1)^1 \cdot 0.2 \cdot 10^{-1}$	-0.02
$(-1)^{0+1} \cdot 0.08 \cdot 10^{3-1}$	
renormalize	
$(-1)^1 \cdot 0.8 \cdot 10^{2-1}$	-8
$(-1)^1 \cdot 0.8 \cdot 10^1$	-8

5.3 Addition - Subtraction

The operation on the fractions can be done only if the exponents are equal.

Basic algorithm :

```
If exponents not equal do
     for number with smallest exponent
          do shift right fraction 1 position
          add 1 to exponent
          until exponents are equal
Add / subtract fractions (with signs)
Renormalize if necessary
```

Example of an addition in decimal :

$0.4 \cdot 10^3$	400
$+ \quad 0.2 \cdot 10^{-1}$	0.02
align exponents	
$0.40000 \cdot 10^3$	same exponents
$+ \quad 0.00002 \cdot 10^3$	
$0.40002 \cdot 10^3$	400.02 no normalization necessary

5.4 $\mathcal{R}$ versus floating point representations

The range and precision of floating point numbers are generally sufficient for all purposes, yet can produce unexpected results :

Example : $f(x) = (x-1)(x-2)\ldots(x-10)$

Mathematically, if $x = 1, 2 \ldots 10$, $f(x) = 0$

We can also represent f as a polynomial : $f(x) = a_0 + a_1 x + \ldots a_{10} x^{10}$

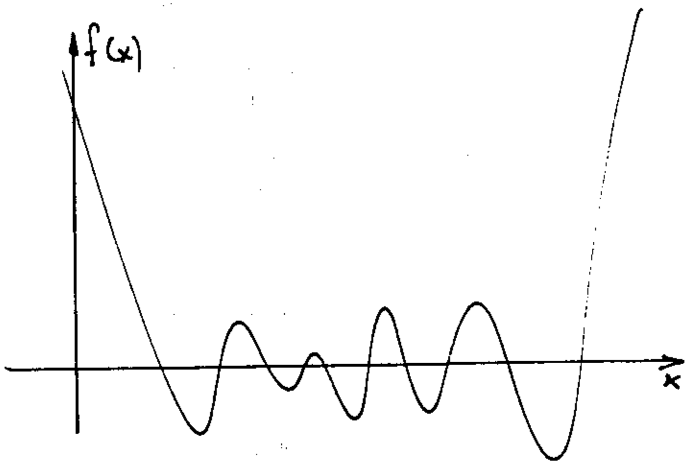


Figure 3: $f(x)$ for $-1\ leq\ x \leq 11$

What happens around $x = 1$ for example ?

$$x = x^2 = x^3 = \ \dots \ = x^{10} = 1$$

and so

$$f = a_0 + a_1 + \ \dots \ a_{10}$$

in a first approximation : $a_0 = (-1)(-2)\dots(-10) = 10 \simeq 3 \cdot 10^6$

and the slope : $\simeq -3 \cdot 10^6$

- We are adding/subtracting very large numbers whose (not roounded) sum should be zero
- for each small step δx, f changes by $3 \cdot 10^6 \cdot \delta x$
- it follows that f is numerically never equal to 0, not even small !

5.4.1 Arithmetic / mathematic errors in general

Whenever we do complex numerical computations, we are facing the following dilemma : the round errors tend to increase when we try to increase the mathematical precision (number of integration steps, number of terms in developments etc). In general, the optimum is obtained when the round off error equal the mathematical (truncation) error.

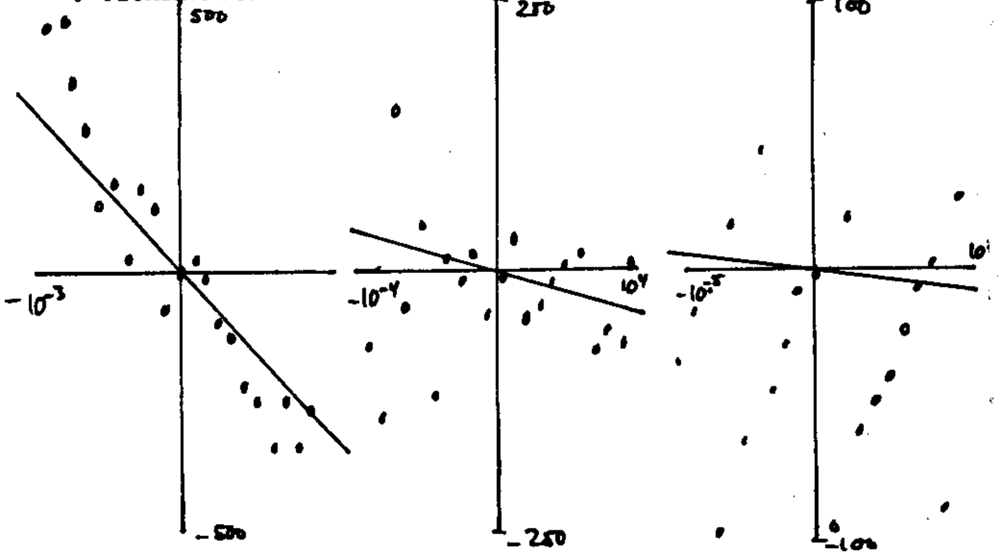


Figure 4: $f(x)$, for x very close to 1

5.5 Some practical rules

- Never compare numbers to strictly, especially near zero
- Avoid subtracting similar numbers
- Avoid very small or large exponents (change units)
- Avoid taking powers,
  - use Hoerner's schema :
    $f = a_0 + x(a_1 + x(a_2 + \dots) \dots))$
  - use Householder or Givens transforms for matrices
  - use double precision sums etc.

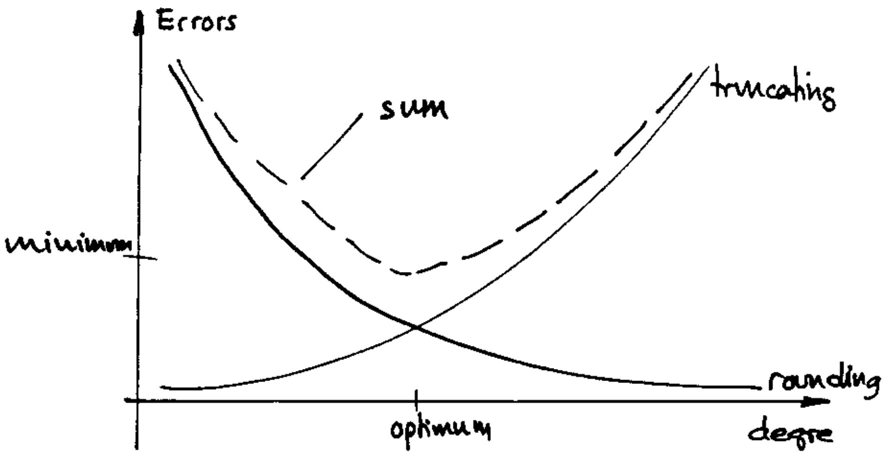


Figure 5: rounding / truncating error behaviour

5.6 Pro and con

Good :
- dont care about dynamic or scaling
- good extended libraries, lots of experience accumulated
- almost problem independent
- code can be used again and again
- fast (?) hardware floating point chips available

Bad :
- may be to slow
- use unnecessary memory
- no standard floating point number representation
- error, truncation, rounding more difficult to analyse

Best for :
- general purpose software
- complex calculations

6 The floating point axis

Floating point numbers can be represented (or not represented) in 7 region of $\mathcal{R}$:

negative overflow	representable negative numbers ⟵ ⟶	negative underflow	zero	positive underflow	representable positive numbers ⟵ ⟶	positive overflow
$-\infty$	Max^- Min^-		0		Min^+ Max^+	$+\infty$

- The spacing between Min^- , 0 , Min^+ depends on the exponents only
- The spacing between representable numbers depends on the fractions

6.1 Truncation and rounding

Whenever we map $\mathcal{R} \longrightarrow \mathcal{FP}$, or when we renormalize a $\mathcal{FP}$ after some operation, we may lose extra significant bits.

There are then many possibilities for truncation, some of the solutions are :

- downward direct rounding $\lfloor a \rfloor$
- upward direct rounding $\lceil a \rceil$
- truncation towards zero
- rounding away from zero
- rounding to closest $\mathcal{FP}$, best, but needs three extra bits

The last three solutions are symmetric around zero, but only the last is also regular around zero.

- We may get a good understanding of the rounding effects of a whole programme, if we can run it twice with different rounding procedure.
- With integer arithmetic, rounding should be in agreement with remainder :

" remainder must have same sign as divisor " $\longrightarrow$ $\lfloor\ \rfloor$ if divisor > 0
$\lceil\ \rceil$ if divisor < 0

7 IEEE floating point standard

Currently, more than 20 different floating point representations are in use ! None is the best we know off !

As a result, complex numerical software are not portable, while more precision could be gained without extra bits.

The IEEE 754 standard defines :

- 3 (4) data formats
- arithmetic operations
- rounding
- exception handling
- special numbers (denormalized numbers and $\mathcal{NAN}$)

7.1 Data formats

	single	double	[quad]	extended min	extended max
word length	32	64	128	44	80
sign	1	1	1	1	1
exponent	8	11	15	11	15
fraction	(1)+23	+52	+112	+31	+63
bias	127	1023	16383	1023	16383
Max	$1.7 \cdot 10^{38}$	$9 \cdot 10^{307}$	$1.2 \cdot 10^{4932}$	$9 \cdot 10^{307}$	
Min	$1.2 \cdot 10^{-38}$	$2.2 \cdot 10^{-308}$	$1.6 \cdot 10^{-4932}$		
precision	10^{-7}	10^{-15}	10^{-33}	10^{-10}	10^{-20}

7.2 Special numbers

sign	biased exponent	fraction	meaning
0	0	0	+0
1	0	0	−0
0/1	0	$\neq 0$	denormalized
0	255	0	$+\infty$
1	255	0	$-\infty$
0/1	255	$\neq 0$	$\mathcal{NAN}$

- A denormalized $\mathcal{FP}$ is for results between 0 and $\pm Min^{\pm}$.
  - It has less significant bits
  - It can be added or subtracted to normalized numbers, but cannot be multiplied or divided
  - It implements **gradual underflow**
- $\mathcal{NAN}$ is the result of an invalid operation ($\sqrt{(-3)}$, $0 * \infty$, $\infty - \infty$) etc.
- ± 0 , $\pm\infty$ are valid numbers, but only some operations are possible with them.
- Extended format should be used, whenever possible, for temporary results, to reduce over/underflow and round off errors in long chains of operations.
- All operations keep internally 3 guard bits (extended fraction) for rounding.

7.3 IEEE standard floating point operations

1. Basic operations : $+ \ - \ * \ /$
2. $\sqrt{FP}$, *remainder*
3. Conversion $\mathcal{FP} \leftrightarrow$ integers (with round/floor)
   Binary $\leftrightarrow$ packed_decimal (integer/$\mathcal{FP}$)
4. Conversions between **single** , **double** , **extended** and **quad** precision
5. Compare and set condition code : $> \ < \ = \ \neq$ *"not comparable"*
6. Rounding

necessary	- unbiased to nearest $\mathcal{FP}$
	- towards zero
optional	- towards $-\infty$
	- towards $+\infty$

7.4 Exception handling

5 exceptions :

- Invalid operation, $\mathcal{NAN}$
- Overflow
- division by zero
- underflow
- inaccurate result

They must :

- set a flag
- execute specified procedure

Trap procedure should :

- indicate what and where it was wrong
- deliver acceptable result if continuation is wanted

7.5 What are $\mathcal{NAN}$?

- you can put any ($\neq 0$) information in the fraction for later analysis
- $\mathcal{NAN}$ propagate through any $\mathcal{FP}$ operation :

$$\mathcal{FP} \begin{matrix} + \\ - \\ * \\ / \end{matrix} \; \mathcal{NAN} \longrightarrow \mathcal{NAN}$$

7.6 Operations with ± 0 , $\pm\infty$

Overflow and division by zero produce $\pm\infty$

$\mathcal{FP}$ / $\pm\infty \longrightarrow \pm 0$

$\mathcal{FP}$ / $\pm 0 \longrightarrow \pm\infty$

± 0 , $\pm\infty$ can be used for comparison

But all these operations activate the "**invalid operation flag**".

8 Floating point processors

Currently, 4 different types of floating point hardware processor are available. They differ considerably by their speed, operations available, and the way they can be attached to given microprocessors.

1. AMD 9511-9512 for 8 bit μProcessors
2. Intel 8087-80287 , NS 32081 , M 68881 for 16/32 bit μProcessors
3. Am, Wytek etc. high speed units for 32 bit μProcessors
4. Sky-Map , HP-FFT , FPSxxx attached signal/vector processors for μProcessors

They are :

- 10-1000 times faster than software equivalent
- very secure (exceptions handled correctly)
- easy to connect to μProcessors
- may not be supported by standard software (specially old versions)

8.1 Floating point processors main characteristics

	AMD 9511	AMD 9512	8087	NS 32081	M 68881	WD
used as	periph	periph	co-proc	co-proc	co-proc	??
Structure	μprog	μprog	μprog	μprog	μprog	??
	stack	stack	stack	8 reg	8 reg	??
integer	yes	no	yes	yes	yes	??
$\mathcal{FP}$ bits	32	32,64	32,64,(80)	32,64,(80)	32,64,80	??
IEEE 754	no	no	~ yes	~ yes	~ yes	??
timing				32/64bits		
FADD	28-128	276-1235	10	7.4/7.4	?	?
FMUL	57	768-863	24	4.8/6.8	?	
FDIV	57	2043-2319	38	8.9/11.8	?	?
FEXP	1955	-	?	?	?	-
FSIN	1882	-	?	?	?	-
SQRT	?	-	38	?	?	-

8.2 Stack operation (zero address processor)

Like pocket calculators, many floating point processors use a stack instead of registers for operands.

All operations are done on the operand(s) at the top of the stack. New operands can be pushed onto the stack, results can be popped out of the stack. The order of operands into the stack can be changed with other instructions. Instructions (operators) do not specify address or registers. This suppress the problem of memory/register allocation for intermediate results.

Basic algorithm :

```
- push operand(s) on stack
- do operation(s) on them
- pop result from stack
```

8.2.1 Rules to go from usual algebraic to postfix (stack) notation

1. Try to keep intermediate results on stack
2. Start from inner part of "usual" formula (brackets)
3. Do not push operands in advance
4. Do not be afraid of stack length. With the exception of recursive functions, depth of 3 or 4 is sufficient in most cases
5. Do not store intermediate results in memory

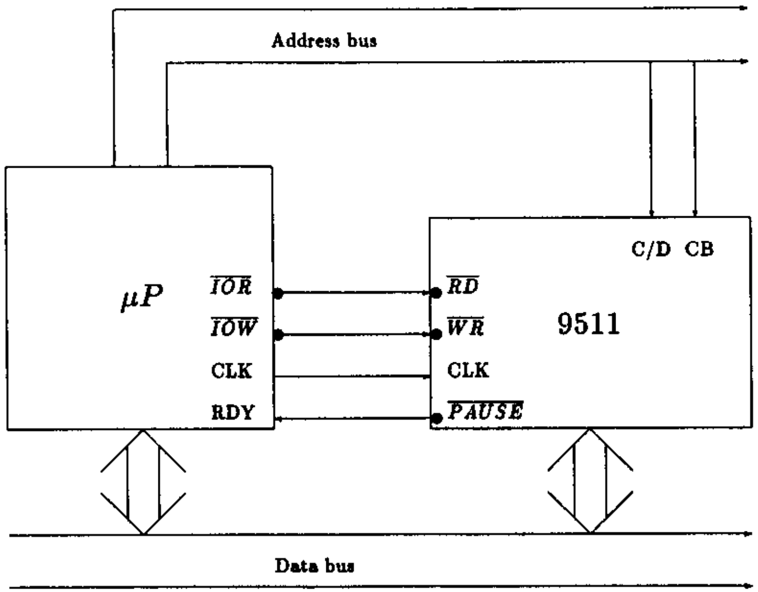
Example of transformation from usual to stack :
Usual mathematical notation :

$$\sqrt{5 + \sin \pi/3}$$

Stack operations : The lowest line represent the operations. An *uparrow* indicates a *push*. The upper lines show the content of the stack, with the *top-of-stack* at the bottom. The final result is always there.

		5				
	5	3	5	5		
5	3	π	$\pi/3$	$\sin \pi/3$	$5 + \sin \pi/3$	$\sqrt{5 + \sin \pi/3}$
5 ↑	3 ↑	π ↑	/	sin	+	$\sqrt{}$

8.3 Attachement of AMD 9511 as a peripheral to a 8bits μP



Note :

- C/D distinguish between data and opcodes & select
- Data bus is used both for data (a byte at a time) and for opcodes

8.3.1 Operations

- Load data, starting with least significant byte of first operand
- send opcode
- read result when ready

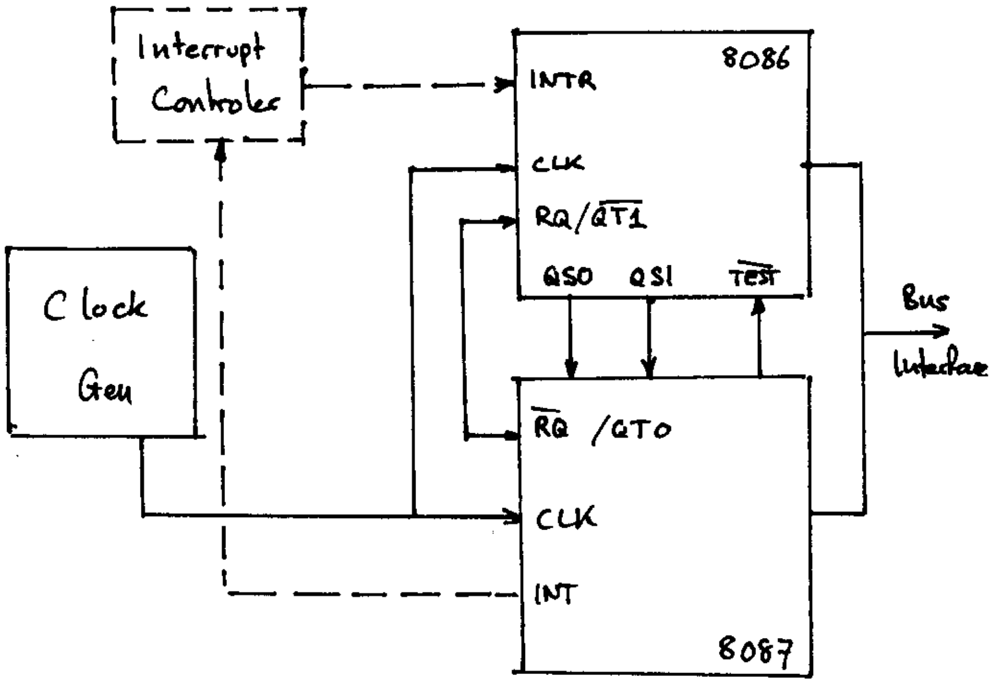
Have subroutines with addresses of operands as parameters

8.3.2 What to do while 9511 is *crunching*

- read status and check busy bit (slows 9511)
- suspend μP with $\overline{PAUSE}$
- do something else until interrupt
- execute **WAI** on 6809

8.4 8086-8087 interconnection

- work in parallel
  - instructions may overlap
  - 8087 let 8086 fetch operands from memory when necessary
  - 8087 process only instructions with ESC code



8.5 Very fast Floating point processor : Am 29325

- Part of Am29300 family
- Single VLSI, 144 pin-grid array
- 32 bits + − * *int* ↔ *fp* in single clock cycle
- internal double precision sum of products
- Newton-Raphson for $1/x$, y/x
- Full IEEE format (+ DEC vax format also)
- 6 flags for status
- 3 * 32 bits-bus flow through + registers bus

1.5C USING ALGEBRAIC REARRANGEMENT TO AVOID LOSS OF SIGNIFICANCE

If $b_1^2 \gg |c_1|$,† then $b_1^2 - c_1$ will not differ much from b_1^2, hence

$$r(-) = b_1 - \sqrt{b_1^2 - c_1} \text{ will suffer loss of significance if } b_1 > 0$$

and

$$r(+) = b_1 + \sqrt{b_1^2 - c_1} \text{ will suffer loss of significance if } b_1 < 0$$

(This is what we saw in Section 1.2B.) So the real roots of the quadratic

$$ax^2 + bx + c = 0 \qquad (a \neq 0)$$

can be calculated without loss of significance using the formula

$$\text{root}_1 = (\pm)(|b_1| + \sqrt{b_1^2 - c_1}) \qquad \text{and} \qquad \text{root}_2 = \frac{c_1}{\text{root}_1} \tag{6a}$$

where

$$b_1 = -\frac{b}{2a}, \quad c_1 = \frac{c}{a}, \quad \text{and} \quad (\pm) \text{ is } (+) \text{ unless } b_1 < 0 \tag{6b}$$

```
      SUBROUTINE QROOTS(A, B, C, ROOT1, ROOT2, COMPLX, IW, PRINT)
      LOGICAL PRINT, COMPLX
C - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - C
C THIS SUBROUTINE FINDS THE TWO ROOTS OF THE QUADRATIC           C
C            AX**2 + BX + C                                      C
C IF PRINT = TRUE, IT PRINTS THEM ON OUTPUT DEVICE IW.           C
C REAL ROOTS (COMPLX=FALSE) ARE RETURNED AS ROOT1 AND ROOT2,     C
C AND COMPLEX ROOTS (COMPLX=TRUE) AS ROOT1 +OR- I*ROOT2.         C
C - - - - - - - - - - - - - - - - VERSION 1  5/1/81  - - - - - - C
      B1 = -0.5*B/A
      C1 = C/A
      DISCR = B1*B1 - C1
      IF (DISCR .LT. 0.) GOTO 10
C
C     REAL ROOTS:  ROOT1 AND ROOT2
         COMPLX = .FALSE.
         ROOT1 = ABS(B1) + SQRT(DISCR)
         IF (B1 .LT. 0.) ROOT1 = -ROOT1
         ROOT2 = 0.0
         IF (ROOT1 .NE. 0.) ROOT2 = C1/ROOT1
C
         IF (PRINT) WRITE(IW,1) ROOT1, ROOT2
    1    FORMAT(' REAL ROOTS:  ',E14.7,'  AND  ',E14.7)
         RETURN
C
C     COMPLEX CONJUGATE ROOTS:  ROOT1  +OR-  I*ROOT2
   10    COMPLX = .TRUE.
         ROOT1 = B1
         ROOT2 = SQRT(-DISCR)
C
         IF (PRINT) WRITE(IW,2) ROOT1, ROOT2
    2    FORMAT(' COMPLEX ROOTS:',E15.7,'  +OR-  I*(',E14.7,')')
         RETURN
C
      END
```

1 High Level Languages

- Allen, J.R.
  Anatomy of Lisp
  Mc Graw Hill 1978
- Brodie, L.
  Starting Forth
  Prentice-Hall 1981
- Brodie, L.
  Thinking Forth
  Prentice-Hall 1984
- Downes, V.A. and S.J. Goldsack
  Programming embedded systems with Ada
  Prentice-Hall 1982
- Friedman, D. and M. Felleisen
  The little LISPer
  Science Research Associates, Inc. 1986
- Ghezzi, C. and M. Jazayeri
  Programming language concepts
  John Wiley and sons 1982
- Griswold, R.E. and M.T. Griswold
  A SNOBOL 4 primer
  Prentice-Hall 1973
- Henderson, P.
  Functional programming, application and implementation
  Prentice-Hall 1980
- Kernighan, B.W. and D.M. Ritchie
  The C programming language
  Prentice-Hall 1978
- Kerridge, J.
  occam programming : a practical approach
  Blackwell Scientific Publications 1987
- Metcalf, M.
  Effective Fortran 77
  Clarendon Press 1985
- Winston, P.H. and B.K.P. Horn
  LISP
  Addison-Wesley 1981
- Wirth, N.
  Programming in Modula-2
  Springer Verlag 1983

2 Operating Systems

- Brinch-Hansen, P.
  Operating system principles
  Prentice-Hall 1973
- Brinch-Hansen, P.
  The architecture of concurrent programs
  Prentice-Hall 1977
- Christian, K.
  The Unix Operating system
  John Wiley and sons, 1983
- Coffman, E.G. and P.J. Denning
  Operating system theory
  Prentice-Hall 1973
- Comer, D.
  Operating system design, the XINU approach
  Prentice-Hall 1984
- Comer, D.
  Operating system design, Internetworking with XINU
  Prentice-Hall 1988
- Gilton, H. and R. Morgan
  Introducing the Unix system
  Mc Graw Hill 1983
- Hoare, C.A.R.
  Communicating Sequential processes
  Prentice-Hall 1985
- Kernighan, E.G. et al.
  Unix time-sharing system
  The Bell system technical journal, **57** *1897-1991 (1978)*
- Kernighan, E.G. and J.R. Mashey
  The Unix programming environment
  Soft. Pract. and Exp. **9** *1-5 (1979)*
- Pechura, M.A.
  Comparing two microcomputer operating systems : CP/M and HDOS
  CACM **26**, *188-195 (1983*
- Tanembaum, A. S.
  Operating Systems: Design and Implementation
  Prentice-Hall 1987

3 Structured programming

- Dahl, O. Dijkstra, E.W. and C.A.R. Hoare
  Structured programming
  Academic Press 1972
- Dijkstra, E.W.
  GOTO statements considered harmful
  CACM 11 *147-148 (1968)*
- Dijkstra, E.W.
  A discipline of programming
  Prentice-Hall 1976
- Hughes, Ch.E. et al.
  Advanced course in programming using Fortran
  John Wiley and sons 1978
- Kruse, R.L.
  Data structures and program design
  Prentice-Hall 1984
- Kernighan, B.W. and P.J. Plauger
  Software tools
  Addison-Wesley 1976
- Kernighan, B.W. and P.J. Plauger
  The elements of programming style
  Addison-Wesley 1978
- Meek, B. and P. Heath
  Guide to good programming practice
  Ellis Horwood Ltd 1980
- Wirth, N.
  Program development by stepwise refinement
  CACM 14, *221-227 (1971)*
- Wirth, N.
  Systematic programming
  Prentice-Hall 1973
- Wirth, N.
  Algorithms + Data-Structure = Programs
  Prentice-Hall 1976
- Yourdon, E.
  Techniques of program structure and design
  Prentice-Hall 1975

4 Modular decomposition, abstraction and information hiding

- Parnas, D.L.
  On the criteria to be used in decomposing systems into modules
  CACM 15, *1053-1058 (1972)*
- Parnas, D.L. et al.
  The modular structure of complex systems
  IEEE Trans. on Soft. Eng. SE11,3 *259-266 (1985)*

5 Jump tables and other algorithms

- Dewar, R.B.K.
  Indirect threaded code
  CACM 18 *330-331 (1975)*
- Knuth, D.E.
  The art of computer programming
  vol 1 : Fundamental algorithms
  vol 2 : Seminumerical algorithms
  vol 3 : Sorting and searching
  Addison-Wesley
- Loeliger, R.G.
  Threaded interpretive languages
  Byte Books 1981
- Sedgewick, R.
  Algorithms
  Addison-Wesley 1983

6 Floating Point operations

- Cody, W.J.Jr and W. Waite
  Software manual for the elementary functions
  Prentice-Hall 1980
- Cody, W.J.Jr
  Analysis of the proposals for the floating point standard
  IEEE Computer march *(1981)*
- Tanenbaum, A.S.
  Structured computer organisation
  Prentice-Hall 1976
- Titus, j.
  Design precaution ensure the benefits of using floating-point coprocessors
  EDN june *57-64 (1986)*
- Waser, S. and M.J. Flynn
  Introduction to arithmetic for digital systems designers
  Holt, Rinehart and Winston 1982

7 Numerical methods

- -
  Numerical recipes : Methods for numerical computation
  (with sources in Fortran 77 and turbo pascal)
  Cambridge University Press 1986
- -
  Numerical recipe examples
  (with sources in Fortran 77 and turbo pascal)
  Cambridge University Press 1986
- Abramowitz, M. and I.A. Stegun
  Handbook of mathematical functions, with formulas, graphs and mathematical tables
  Dover 1965
- Atkinson, L.V. and P.J. Harley
  An introduction to numerical methods with Pascal
  Addison-Wesley 1983
- de Boor, C.
  A practical guide to splines
  Springer 1978
- Bracewell, R,
  The Fourier transform and its applications
  Mc Graw-Hill 1965
- Chan, T.F. and J.G. Lewis
  Computing standard deviations : accuracy
  CACM **22** *526-531, (1979)*
- Cloutier, M.J. and M.J. Friedman
  Precision averageing for real-time analysis
  CACM **26** *525-529, (1983)*
- Garbow, B.S. et al.
  Matrix eigensystem routines, EISPACK guide extensions
  Springer 1977
- Hamming, R.W.
  Digital filters
  Prentice-Hall 1977
- Lawson, C.L. and R.J. Hanson
  Solving least squares problems
  Prentice-Hall 1974
- Maron, M.J.
  Numerical analysis
  Macmillan 1982

- Piessens, R. et al.
  QUADPACK, a subroutine package for automatic integration
  Springer 1983
- Shampine, L.F. and M.K. Gordon
  Computer solution of ordinary differential equations, the initial value problem
  Freeman 1975
- Smith, B.T. et al.
  Matrix eigensystem routines, EISPACK guide
  Springer 1970
- West, D.H.D.
  Updating mean and variance estimates : an improved method
  CACM **22** *532-535, (1979)*

8 Unclassified

- Bentley, J.L.
  Writing efficient programs
  Prentice-Hall 1982
- Dreyfus, H.L. and S.F. Dreyfus
  Mind over Machine : The Power of Human Intuition and Expertise in the Era of the Computer
  Basil Blackwell Ltd. 1986
- Weizenbaum, J.
  Computer power and human reason, from judgment to calculation
  Freeman 1975