Floating-point computation

- Real values and floating point values
- Floating-point representation
- IEEE 754 standard
  - representation
  - rounding
  - special values

Real values

- Not all values can be represented exactly in a computer with limited storage capacity
  - rational numbers: 1/3 ≈ 0,33333 . . .
  - irrational numbers: π ≈ 3,1415 . . .
- If we know the needed accuracy, fixed-point representation can be used
  - always use a fixed number of decimals
- *Example*: two significant decimals
  - 125,01 is scaled by 100
  - 12501 can be exactly stored in binary representation
- When stored in a computer, real values have to be rounded to some accuracy

Real value representation

- In scientific notion, values are represened by a mantissa, base and exponent
  - 6.02 x 10^6 = 6020000,   3.48 x 10^{-3} = 0,00348
- When stored in a computer, we use a fixed number of positions for the mantissa and exponent
  - the base is implicit and does not have to be stored
- Difference between two successive values is not uniform over the range of values we can represent
- *Example*: 3 digit mantissa, exponent between -6 and +6
  - two consecutive small numbers: 1.72 x 10^{-6} and 1.73 x 10^{-6} the difference is 0.00000001 = 1.0 x 10^{-8}
  - two consecutive large numbers: 6.33 x 10^6 and 6.34 x 10^6 the difference is 10000 = 1.0 x 10^4

Normalization

- There are multiple representations of a number in scientific notion
  - 2.00 x 10^4 = 2000 ← *Normalized form*
  - 0.20 x 10^5 = 2000
  - 0.02 x 10^6 = 2000
- In a normalized number the mantissa is shifted (and the exponent justified) so that there is exactly one nonzero digit to the left of the decimal point

Precision

- Assume we store normalized numbers with 7 digits of precision (float)
  - X = 1.25 x 10^8 = 125 000 000,0
  - Y = 7.50 x 10^{-3} = 0,0075
  - X+Y = 1.250000000075 x 10^8

```
 125000000.0000
+        0.0075
---------------
 125000000.0075
```

- The result can not be stored with the available presicion
  - will be truncated to 1.25 x 10^8
- If we repeat this in a loop, the result may be far off from the expected

```
float X;
float Y[100000];
  . . .
for (i=0;i<100000;i++) {
    X += Y[i]
}
```

Associativity

- For the same reason, the order of caclulation may affect the result
  - the small values in the array Y sum up to a value that is significant when added to the large value in X
- Matemathically, associative transformations are allowed
  - not computationally when using floating-point values
- Fortran is very strict about the order of evaluation of expressions
  - C is not so strict

```
float X;
float sum=1.0;
float Y[100000];
  . . .
for (i=0;i<100000;i++) {
    sum += Y[i]
}
X += sum;
```

Guard digits

- To improve the precision of floating-point computations guard digits are used
  - extra bits of precision used while performing computations
  - no need for additional significant bits for stored values
- Assume we use five digits for representing floating-point numbers
  - 10.001 - 9.9993 = 0.0017
- If we use only five digits when aligning the decimal points in the computation, we get truncation
  - if we use 6 digits of accuracy when aligning operands and round the result before normalization, we get the correct result

```
  10.001
- 9.9993
  ------
   0.002
```

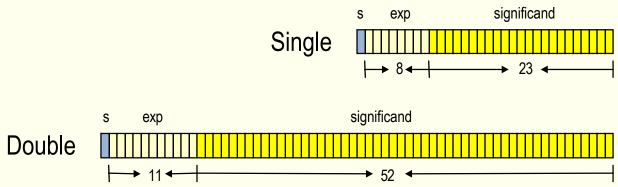
IEEE 754 floating-point standard

- IEEE 754-1985 Standard for Binary Floating-Point Arithmetic
- Describes the
  - storage formats
  - exact specificiation of the results of operations on floating-point values
  - special values
  - runtime behaviour on illegal operations (exceptions)
- Does not specify how floating-point operations should be implemented
  - computer vendors are free to develop efficient solutions, as long as they behave as specified in the standard

IEEE 754 formats

- Floating-point numbers are 32-bit, 64-bit or 80-bit
  - Fortran REAL*4 is also refered to as REAL
  - Fortran REAL*8 is also refered to as DOUBLE

IEEE 754	FORTRAN	C	Bits	Exponent bits	Significand bits
Single	REAL*4	float	32	8	24
Double	REAL*8	double	64	11	53
Double Extended	REAL*10	long double	≥80	≥15	≥64



Range and accuracy

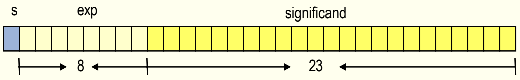
- The minimum normalized number is the smallest number that can be represented at full precision

IEEE 754	Minimum normalized nr	Largest finite nr	Base-10 accuracy
Single	1.2 E -38	3.4 E +38	6–9 digits
Double	2.2 E -308	1.8 E +308	15–17 digits
Double Extended	3.4 E -4932	1.2 E +4932	18–21 digits

- Smaller values are represented as subnormal numbers, with loss of precision
  - smallest 32-bit subnormal number is 2.0 E -45
  - accuracy 1–2 base-10 digits

IEEE format

- The high-order bit (bit 31) is the sign of the number
  - does not use 2's complement
- The base-2 exponent is stored in bits 23-30
  - biased by adding 127
  - can represent exponent values from -126 to +127
  - for 64-bit values the bias is 1023
- The mantissa is converted to base-2 and normalized
  - one non-zero digit to the left of the binary point
- All normalized binary numbers have a 1 as the first bit
  - do not have to store the leading 1
- The mantissa stored in this format is called the *significand*



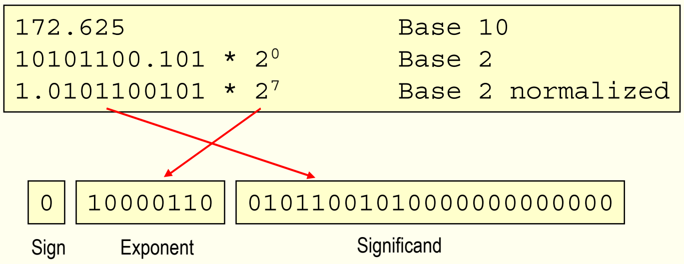
Converting from base-10 to IEEE format

- Example of converting 172.625 from base-10 to IEEE format
- First convert 172.625 to base-2
  - 172 = 128 + 32 + 8 + 4 = $2^7 + 2^5 + 2^3 + 2^2$
  - 0.625 = 0.5 + 0.125 = $2^{-1} + 2^{-3}$
- Normalize the base-2 number
  - shift the binary point 7 steps to the right
  - adjust the exponent by adding 7

172.625	Base 10
$10101100.101 * 2^0$	Base 2
$1.0101100101 * 2^7$	Base 2 normalized

Converting to IEEE format (cont.)

- Add bias 127 to the exponent
  - $7 + 127 = 134 = 128 + 4 + 2 = 2^7 + 2^2 + 2^1$
- Drop the leading 1-bit from the significand
  - extend to 23 bits
- Set sign bit to 0
  - positive number

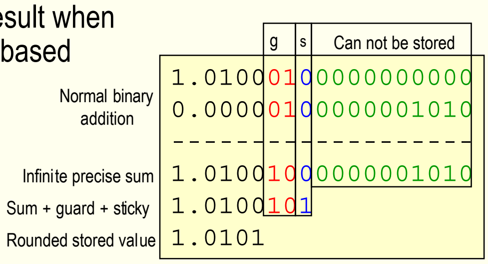


172.625	Base 10
$10101100.101 * 2^0$	Base 2
$1.0101100101 * 2^7$	Base 2 normalized

Sign	Exponent	Significand
0	10000110	01011001010000000000000

Guard digits

- The IEEE 754 standard requires the use of 2 guard digits and one sticky bit in floating-point computations
  - used for rounding the result
- The guard digits act as two extra bits of precision
  - as if the significand were 25 bits instead of 23
- The sticky bit is set to 1 if any of the bits beyond the guard bits would become nonzero, in either operand
  - used for rounding the result when we can not decide only based on the guard digits
- *Example*:
  - 5 bits of precision



		g	s	Can not be stored
Normal binary addition	1.0100	01	0	0000000000
	0.0000	01	0	0000001010
	-----	--	-	----------
Infinite precise sum	1.0100	10	0	0000001010
Sum + guard + sticky	1.0100	10	1	
Rounded stored value	1.0101			

Rounding

- Decide wether to round the last storable bit up or down
- If both guard digits are zero, the result is exactly the extended sum
- If the guard digits are 01, the result is rounded down
  - error is one guard digit unit
- If both guard digits are one, the result is rounded up
- When guard digits are 10 we have the largest error
  - look at the sticky bit do decide which way to round the result

Extended sum	Stored value
`1.0100 00x`	`1.0100`

Extended sum	Stored value
`1.0100 01x`	`1.0100`

Extended sum	Stored value
`1.0100 11x`	`1.0101`

Extended sum	Stored value
`1.0100 101`	`1.0101`

Special values

- The standard also defines a number of special values

Special value	**Exponent**	**Significand**
+ or – 0	`00000000`	0
Denormalized number	`00000000`	nonzero
NaN (Not a Number)	`11111111`	nonzero
+ or – Infinity	`11111111`	0

- Denormalized numbers are used to repesent values smaller than the minimum normalized number
  - exponent is zero
  - significand bits are shifted right (incuding the implicit leading 1-bit)
  - gradual underflow – last nonzero bit is shifted out
- Values that are increased beyond the maxmimum value get the special value Infinity
  - overflow

Special values (cont.)

- NaN indicates a number that is not mathematically defined
  - divide zero by zero
  - divide Infinity by Infinity
  - square root of -1
  - any operation on NaN produces NaN as result
- The standard defines a way of detecting results that are not mathematically defined
  - cause a trap to a subroutine when results that can not be represented are produced
  - overflow to infinity, underflow to zero, division by zero, etc.
  - cause a jump to a subroutine that handles the exception
  - can cause significant overhead on the computation

Compiler options

- Some compilers may violate some of the rules in the standard to produce faster code
  - assumes arguments to square root function is non-negative
  - assumes no results of operations will be NaN
- May produce incorrect numerical results
- *Example*:
  - gcc -ffast-math