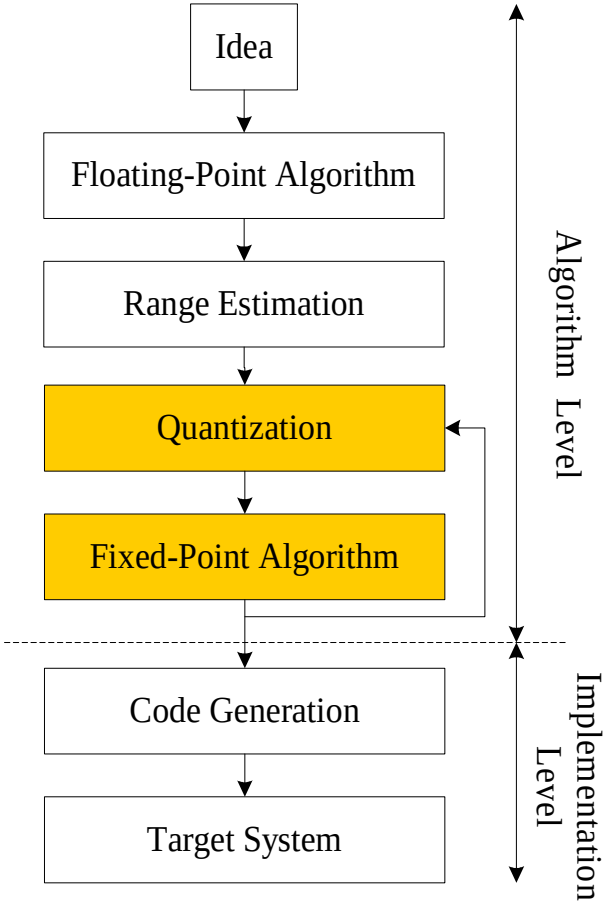
Fixed-point design

Overview

- **Introduction**
- Numeric representation
- Simulation methods for floating to fixed point conversion
- Analytical methods

Fixed-Point Design

- Digital signal processing algorithms
  - Often developed in floating point
  - Later mapped into fixed point for digital hardware realization
- Fixed-point digital hardware
  - Lower area
  - Lower power
  - Lower per unit production cost

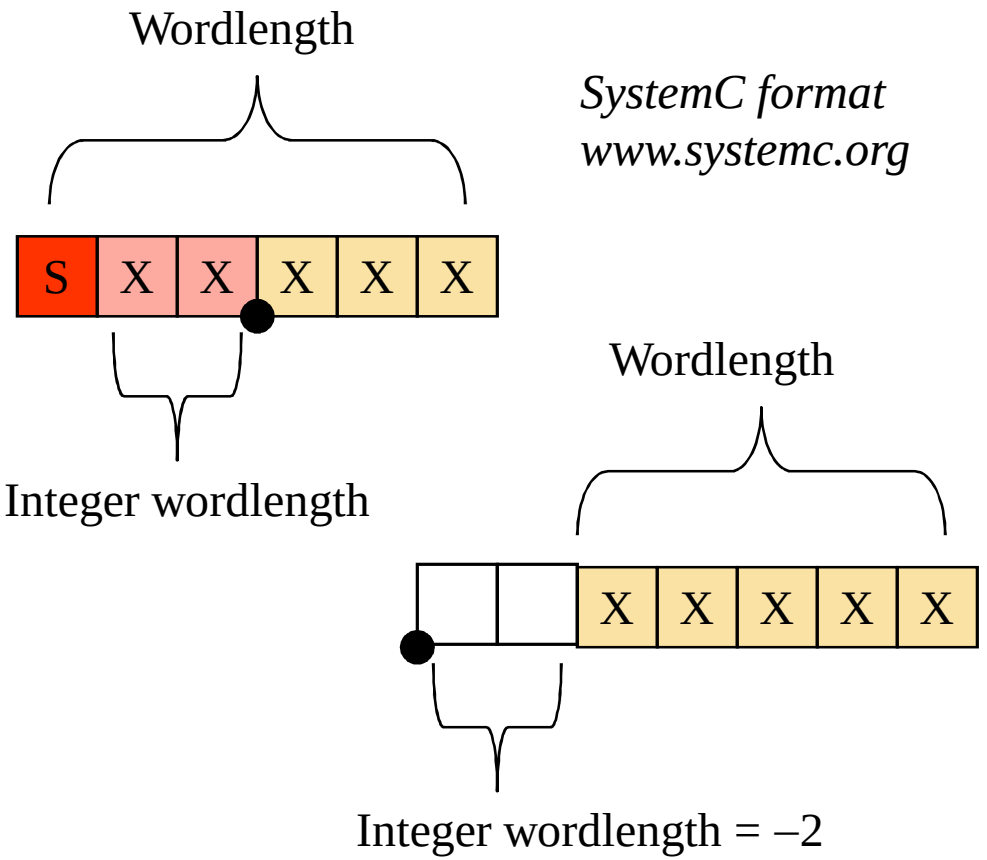


Fixed-Point Design

- Float-to-fixed point conversion required to target
  - ASIC and fixed-point digital signal processor core
  - FPGA and fixed-point microprocessor core
- All variables have to be annotated manually
  - Avoid overflow
  - Minimize quantization effects
  - Find *optimum wordlength*
- Manual process supported by simulation
  - Time-consuming
  - Error prone

Fixed-Point Representation

- Fixed point type
  - Wordlength
  - Integer wordlength
- Quantization modes
  - Round
  - Truncation
- Overflow modes
  - Saturation
  - Saturation to zero
  - Wrap-around



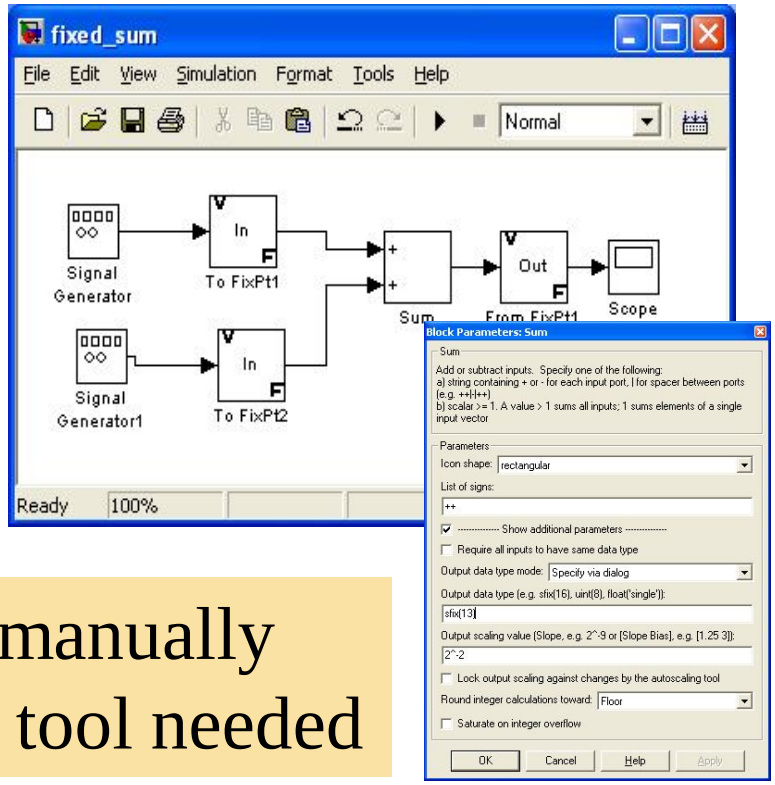
Tools for Fixed-Point Simulation

- gFix (Seoul National University)
  - Using C++, operator overloading
- Simulink (Mathworks)
  - Fixed-point block set 4.0
- SPW (Cadence)
  - Hardware design system
- CoCentric (Synopsys)
  - Fixed-point designer

```
float a;
float b;
float c;
c = a + b;
```

→

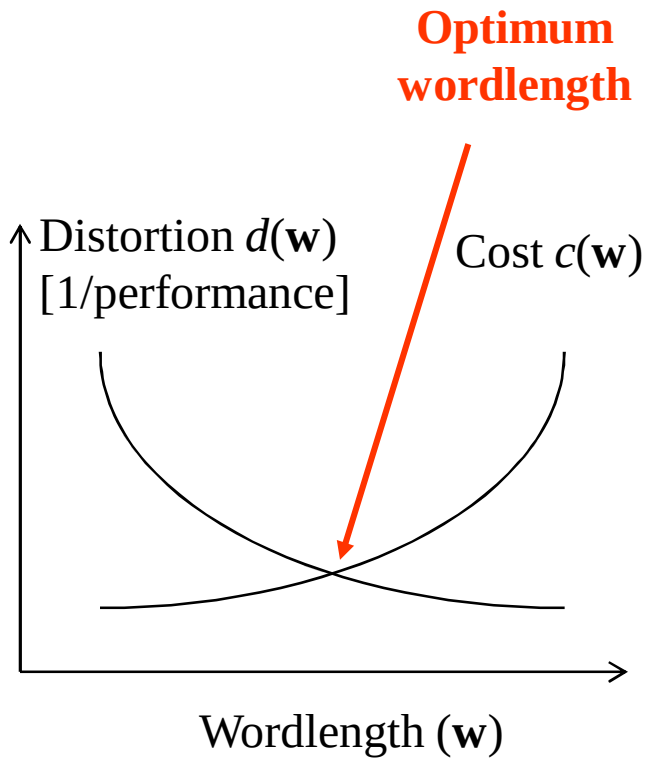
```
gFix a(12,1);
gFix b(12,1);
gFix c(13,2);
c = a + b;
```



Wordlengths determined manually
Wordlength optimization tool needed

Optimum Wordlength

- Longer wordlength
  - May improve application performance
  - Increases hardware cost
- Shorter wordlength
  - May increase quantization errors and overflows
  - Reduces hardware cost
- Optimum wordlength
  - Maximize application performance or minimize quantization error
  - Minimize hardware cost



Wordlength Optimization Approach

- Analytical approach
  - Quantization error model
  - For feedback systems, instability and limit cycles can occur
  - Difficult to develop analytical quantization error model of adaptive or non-linear systems
- Simulation-based approach
  - Wordlengths chosen while observing error criteria
  - Repeated until wordlengths converge
  - Long simulation time

Overview

- Introduction
- **Numeric representation**
- Simulation methods for floating to fixed point conversion
- Analytical methods

Number representation

Matlab examples

- Numeric circle
- fi Basics
- fi Binary Point Scaling

Fi type

- **Integer arithmetic with a fixed number of fractional digits**

```
>> a=fi(pi, true, 8, 5);
>> bin(a)
0   1   1 .  0   0   1   0    1
s   2   1 . 1/2 1/4 1/8 1/16 1/32

>> double(a)
3.15625
```

Fi object

```
>> a = fi(pi)

a =

                3.1416015625

          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
              WordLength: 16
          FractionLength: 13

              RoundMode: nearest
            OverflowMode: saturate
             ProductMode: FullPrecision
    MaxProductWordLength: 128
                SumMode: FullPrecision
        MaxSumWordLength: 128
           CastBeforeSum: true
```

data

numerictype

fimath

Fi Object

- Notation
- Multiplication
- Multiplication with KeepMSB Mode
- Addition
- Addition with KeepLsb Mode
- Numerictype
- fimath

Overview

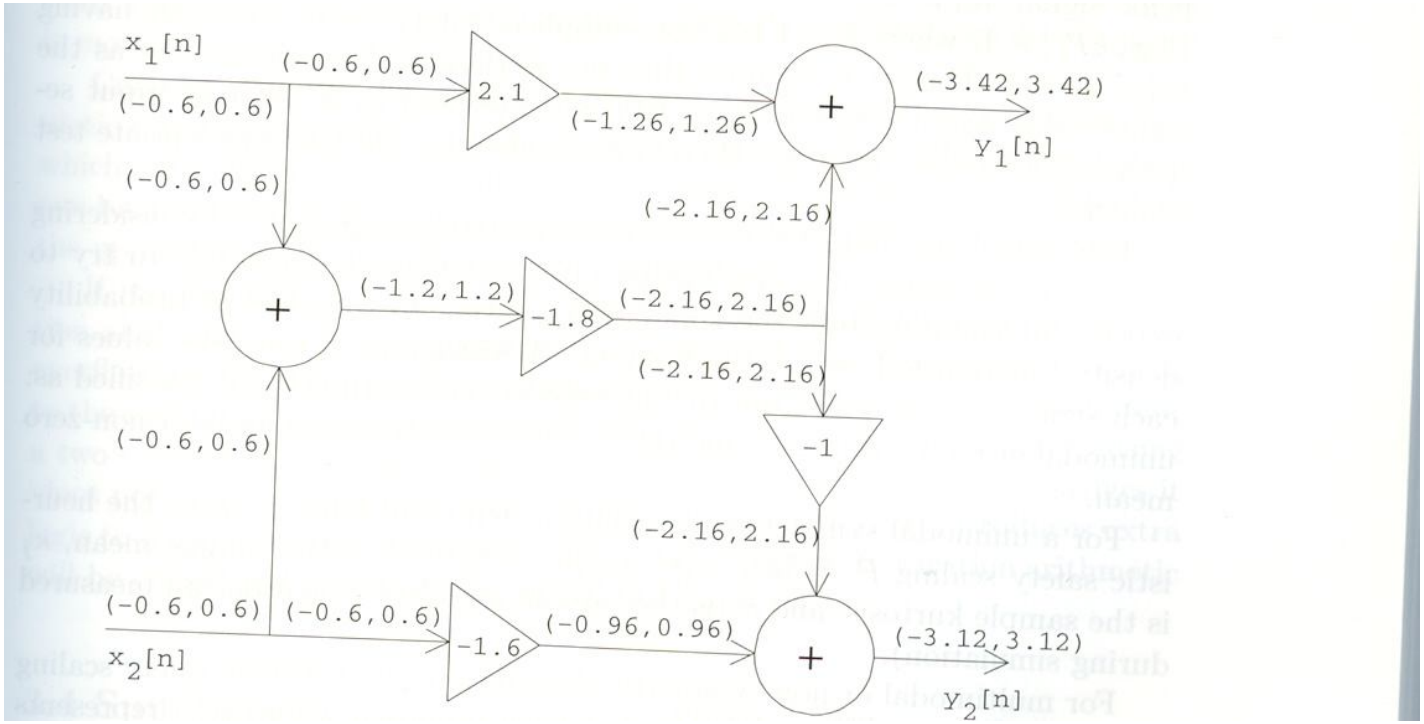
- Introduction
- Numeric representation
- **Simulation methods for floating to fixed point conversion**
- Analytical methods

Data-range propagation

y1=2.1x1-1.8(x1+x2)=0.3x1-1.8x2

Input range: (-0.6 0.6)

Output range: (-1.26, 1.26)



Data-range propagation

Disadvantages

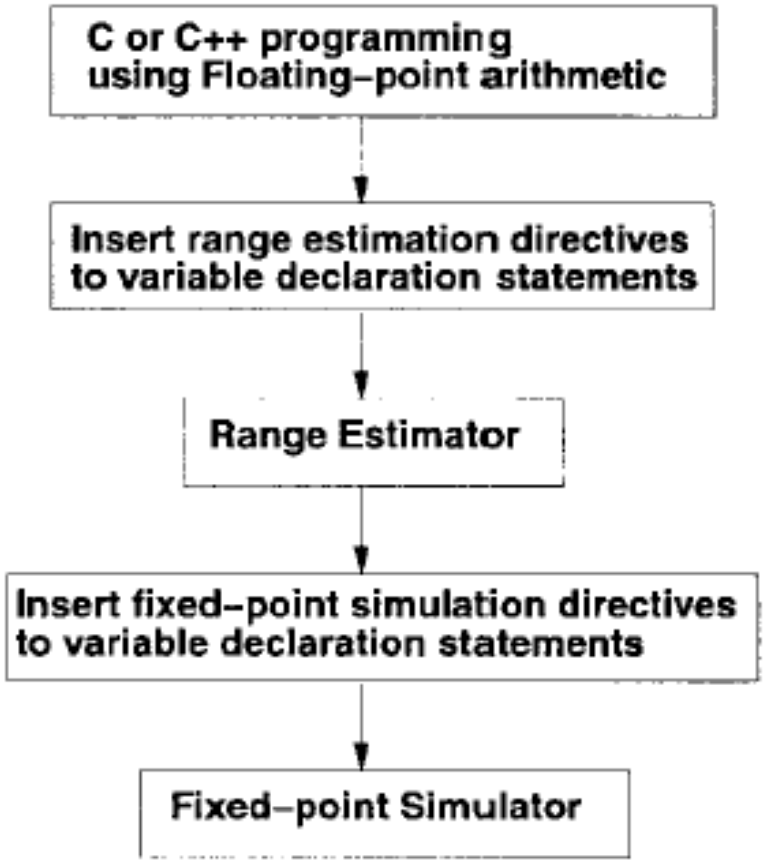
- Provide larger bounds on signal values than necessary

Solution

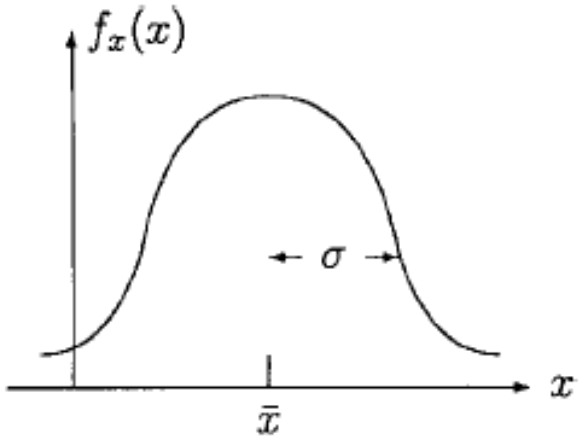
- Simulation-based range estimation

Development of fixed point programs

- Toolbox gFix

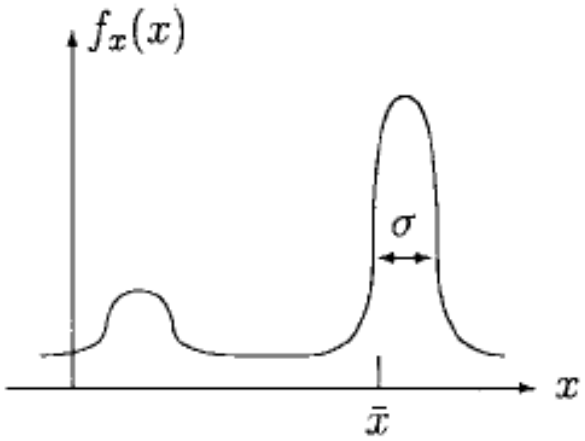


Statistical characteristics of input signals

- 

(a)

$$R = |\mu| + n \times \sigma_y \qquad n \propto k.$$



$$R = \hat{R}_{99.9\%} + g$$

Implementation – range estimation

```
class fSig
{
private:
  ......
  double        Data;
  double        Sum;
  double        Sum2;
  double        Sum3;
  double        Sum4;
  double        AMax;
  long          SumC;
  ......

public:

  ......
  fSig& operator = (const fSig&);
  fSig& operator = (double);

  friend double operator + (const fSig&, const fSig&);
  friend double operator + (const fSig&, double);
  friend double operator - (const fSig&, const fSig&);
  friend double operator - (const fSig&, double);
  friend double operator * (const fSig&, const fSig&);
  friend double operator * (const fSig&, double);
  friend double operator / (const fSig&, const fSig&);
  friend double operator / (const fSig&, double);

  friend short operator == (const fSig&, const fSig&);
  friend short operator == (const fSig&, double);
  friend short operator != (const fSig&, const fSig&);
  friend short operator != (const fSig&, double);
  friend short operator > (const fSig&, const fSig&);
  friend short operator > (const fSig&, double);
  friend short operator < (const fSig&, const fSig&);
  friend short operator < (const fSig&, double);
  ......

};
```

Result of the range estimator

```
Statistics:
  VarName     Mean  StdDev Skewness Kurtosis  R99.9%   R100% Update
iir1/Ydly -0.1133 +1.3076  +0.0220  -0.0258 +4.2638 +4.4214   3001
iir1/Yout -0.1134 +1.3078  +0.0220  -0.0268 +4.2638 +4.4214   3000

Integer wordlengths:
      VarName          Range      IWL
    iir1/Ydly      +5.309891       +3
    iir1/Yout      +5.309515       +3

Save the statistical results? [Y/n]
Filename? iir1.sta
```

Fixed point simulation

```
class gFix
{
   Integer m;           // mantissa
   short   iwl;         // integer word-length
   short   wl;          // total word-length
   char    represent;   // 't' or, 'u'
   char    saturation;  // 's' or, 'o'
   char    round;       // 'r' or, 't'

public:

   // constructors
   gFix();
   gFix(short wlen, short iwlen, char *fmt);
   gFix(double d, short wlen, short iwlen, char *fmt);
   ......

   // assignment operators
   gFix& operator = (gFix& x);
   gFix& operator = (double d);

   // basic operators
   friend gFix operator + (gFix& x, gFix& y);
   friend gFix operator - (gFix& x, gFix& y);
   friend gFix operator * (gFix& x, gFix& y);
   friend gFix operator / (gFix& x, gFix& y);
   friend gFix operator << (gFix& x, short b);
   friend gFix operator >> (gFix& x, short b);
   ......

   // assignment based operators
   gFix& operator += (gFix& x);
   gFix& operator -= (gFix& x);
   gFix& operator *= (gFix&);
   ......

   // relational operators
   friend short operator == (gFix& x, gFix& y);
   friend short operator != (gFix& x, gFix& y);
   friend short operator >= (gFix& x, gFix& y);
   ......

   // miscellaneous operators
   friend istream& operator >> (istream& s, gFix& x);
   friend ostream& operator << (ostream& s, gFix& x);
   ......

};
```

Operator overloading

-

```
gFix operator * (const gFix& x, const gFix& y)
// assume that
//    result.wl = x.wl + y.wl - 1
//    result.iwl = x.iwl + y.iwl
{
  short          iwlen, wlen;
  Integer        I;

  wlen = x.wl + y.wl - 1;
  if( wlen > MAXWL ) wlen = MAXWL;
  iwlen = x.iwl + y.iwl;
  I = x.M * y.M;

  return gFix(I,wlen,iwlen);
}
```

Fixed-precision algorithm

-

```
void
iir1(short argc, char *argv[])
{
  gFix  Xin(12,0);
  gFix  Yout(16,3);
  gFix  Ydly(16,3);
  gFix  Coeff(10,0);

  Coeff = 0.9;
  Ydly = 0.;
  for( i = 0; i < 1000; i++ ) {
    infile >> Xin ;
    Yout = Coeff * Ydly + Xin ;
    Ydly = Yout ;
    outfile << Yout << '\n';
  }
}
```

Reducing the number of overflows in Matlab

1. Implement textbook algorithm in M.

2. Verify with builtin floating-point in M.

3. Convert to fixed-point in M and run with default settings.

4. Override the fi object with 'double' data type to log min and max values.

5. Use logged min and max values to set the fixed-point scaling.

6. Validate the fixed-point solution.

7. Convert M to C using Embedded MATLAB or Simulink to FPGA using Altera and Xilinx tools.

Matlab functions

- logreport
- fi_best_numeric_type_from_logs

Overview

- Introduction
- Numeric representation
- Simulation methods for floating to fixed point conversion
- **Analytical methods**

Filter Implementation

- **Finite word-length effects (fixed point implementation)**
  - **Coefficient quantization**
  - **Overflow & quantization in arithmetic operations**
    - scaling to prevent overflow
    - **quantization noise statistical modeling**
    - **limit cycle oscillations**

Coefficient Quantization

The coefficient quantization problem :

- **Filter design in Matlab (e.g.) provides filter coefficients to 15 decimal digits (such that filter meets specifications)**
- **For implementation, need to quantize coefficients to the word length used for the implementation.**
- **As a result, implemented filter may fail to meet specifications… ??**

- **PS: In present-day signal processors, this has become less of a problem (e.g. with 16 bits (=4 decimal digits) or 24 bits (=7 decimal digits) precision). In hardware design, with tight speed requirements, this is still a relevant problem.**

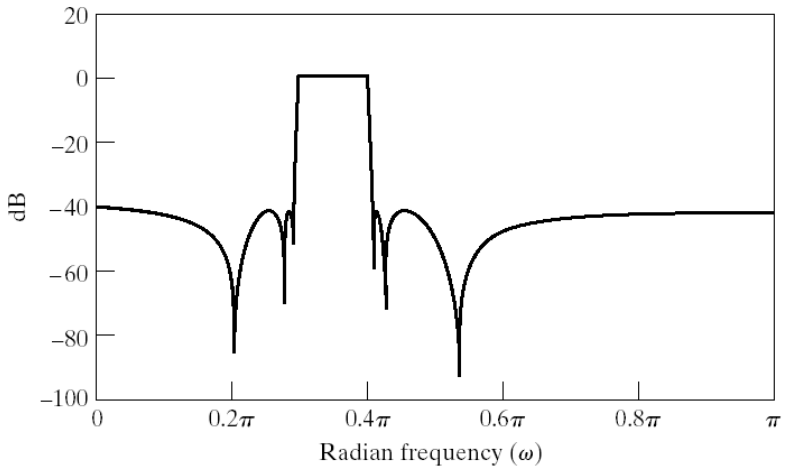
Coefficient Quantization

Coefficient quantization effect on <u>pole locations</u> :

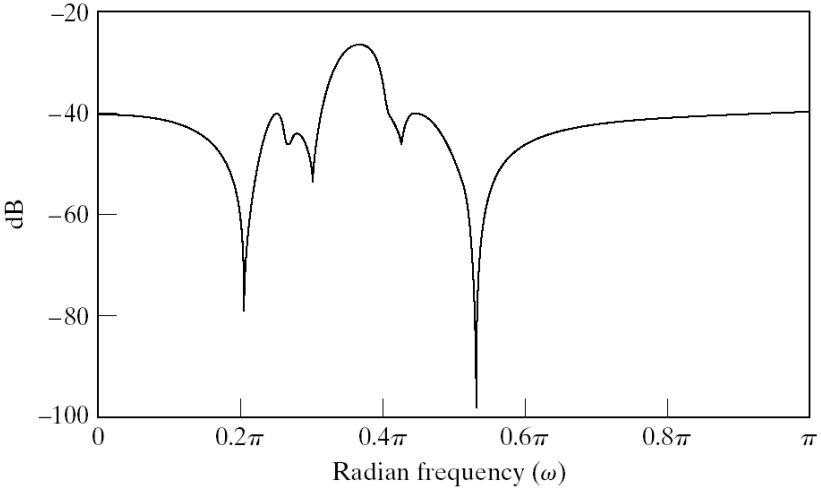
-> tightly spaced poles (e.g. for narrow band filters) imply high sensitivity of pole locations to coefficient quantization

-> hence preference for low-order systems (parallel/cascade)

Example: Implementation of a band-pass IIR 12-order filter



Cascade structure with 16-bit coeff.



Direct form with 16-bit coeff.

Coefficient Quantization

Coefficient quantization effect on <u>pole locations</u> :

- **example : 2nd-order system (e.g. for cascade realization)**

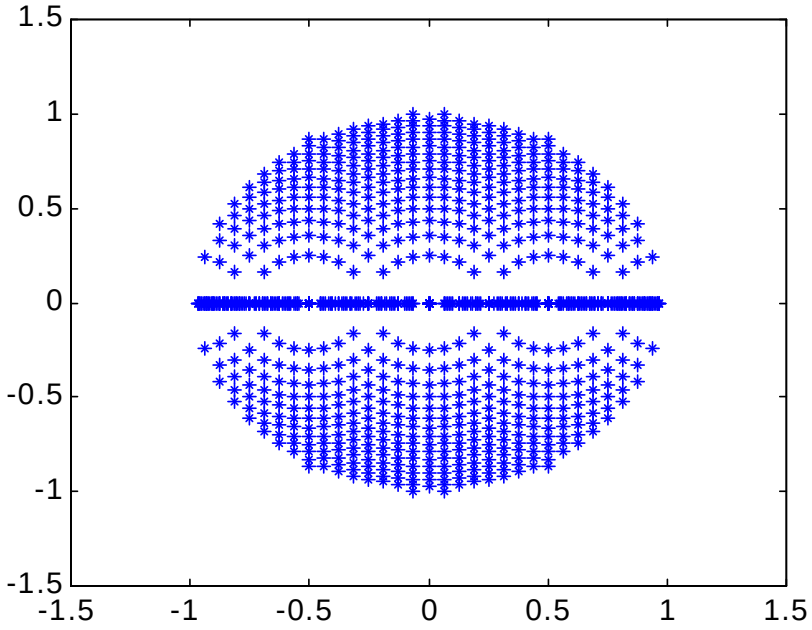
$$H_i(z) = \frac{1 + \alpha_i . z^{-1} + \beta_i . z^{-2}}{1 + \gamma_i . z^{-1} + \delta_i . z^{-2}}$$

Coefficient Quantization

- **example (continued) :**

  with 5 bits per coefficient, all possible pole positions are...

```
for γ_i = −2 : 0.1250 : 2
    for δ_i = −1 : 0.0625 : 1
        plot(roots )
    end
end
```



Low density of permissible pole locations at z=1, z=-1, hence problem for narrow-band LP and HP filters

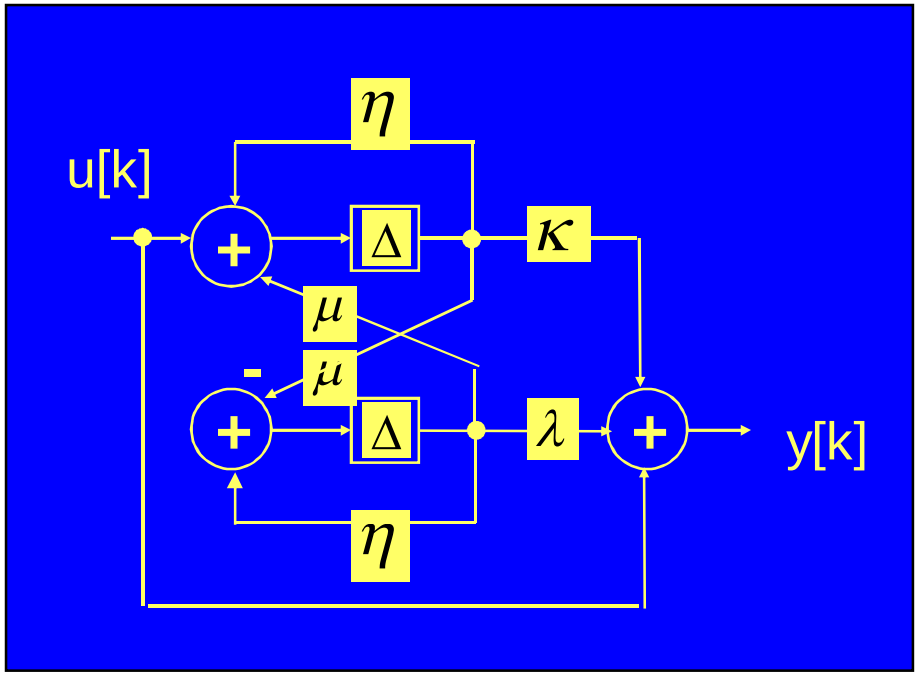
Coefficient Quantization

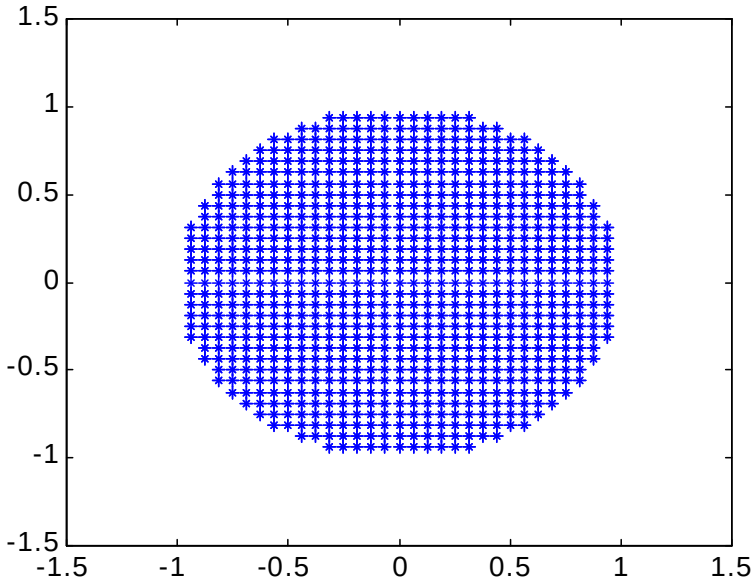
- **example (continued) :**

  possible remedy: `coupled realization'

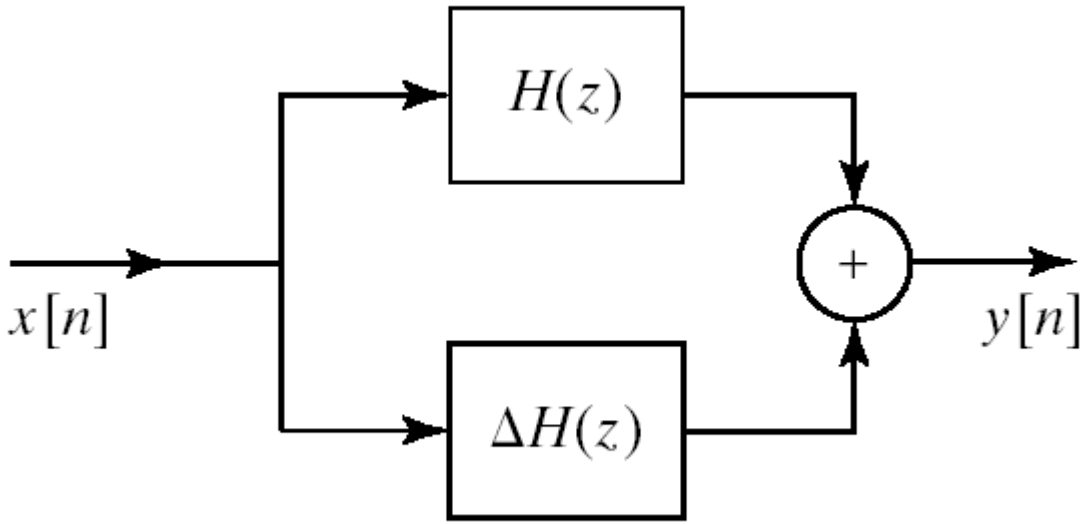
  poles are $\eta \pm j.\mu$ **where** η, μ **are realized/quantized**

  hence permissible pole locations are (5 bits)





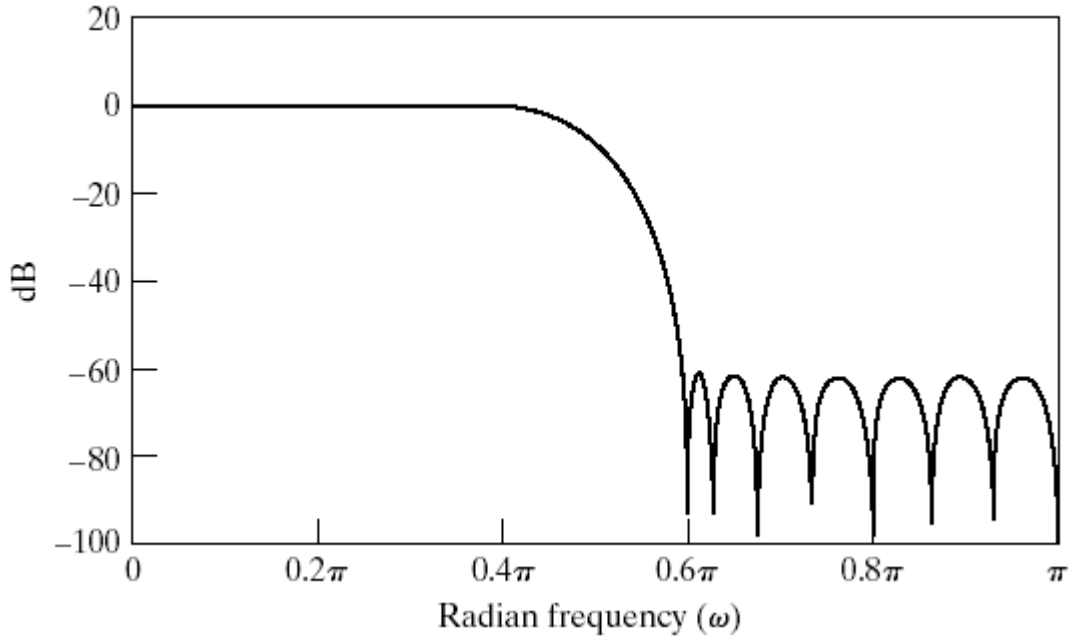
Quantization of an FIR filter

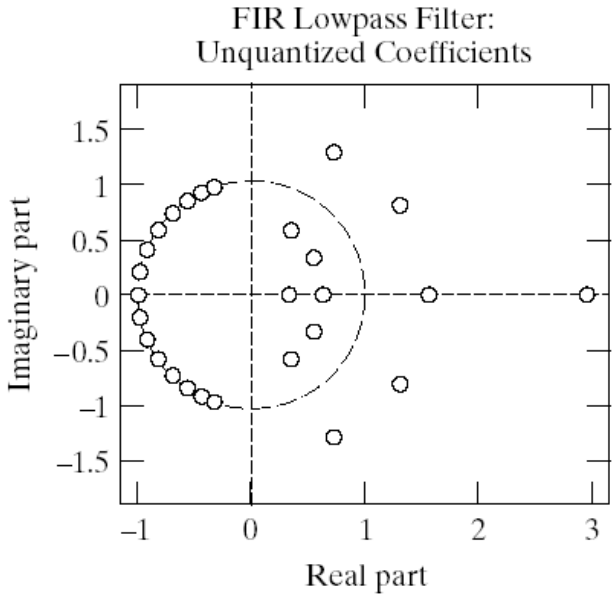


- Transfer function ΔH(z)
- The effect of coefficient quantization to linear phase

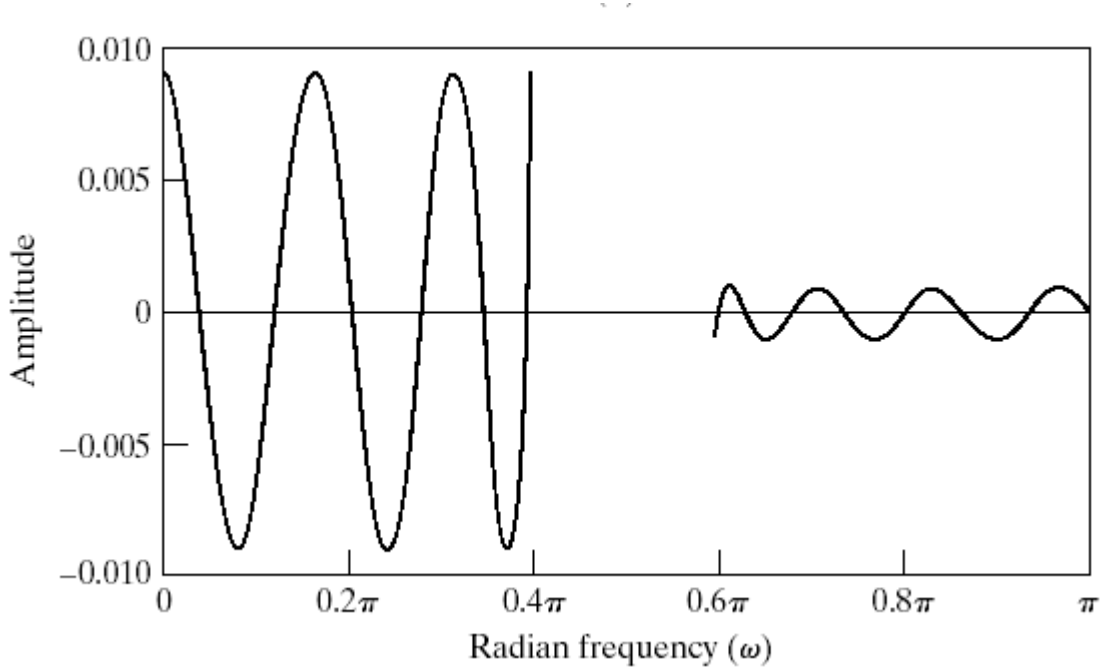
FIR filter example

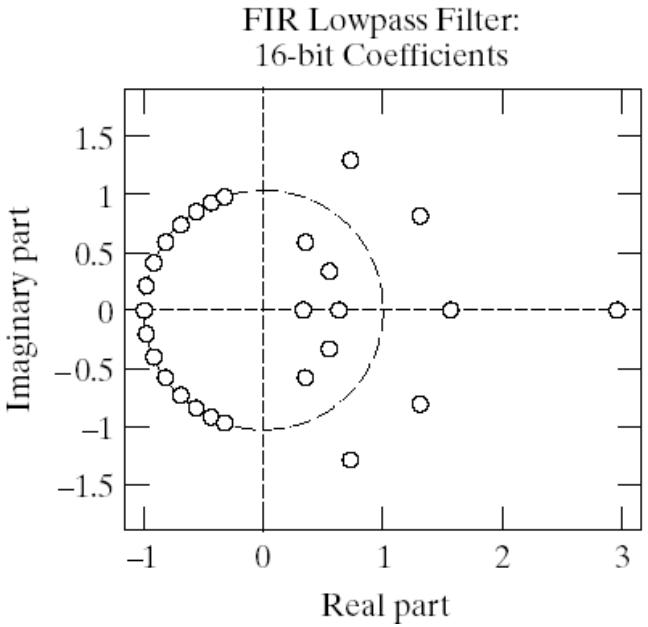
- Passband attenuation 0.01, Radial frequency (0,0.4π)
- Stopband attenuation 0.001, Radial frequency (0.4π, π)



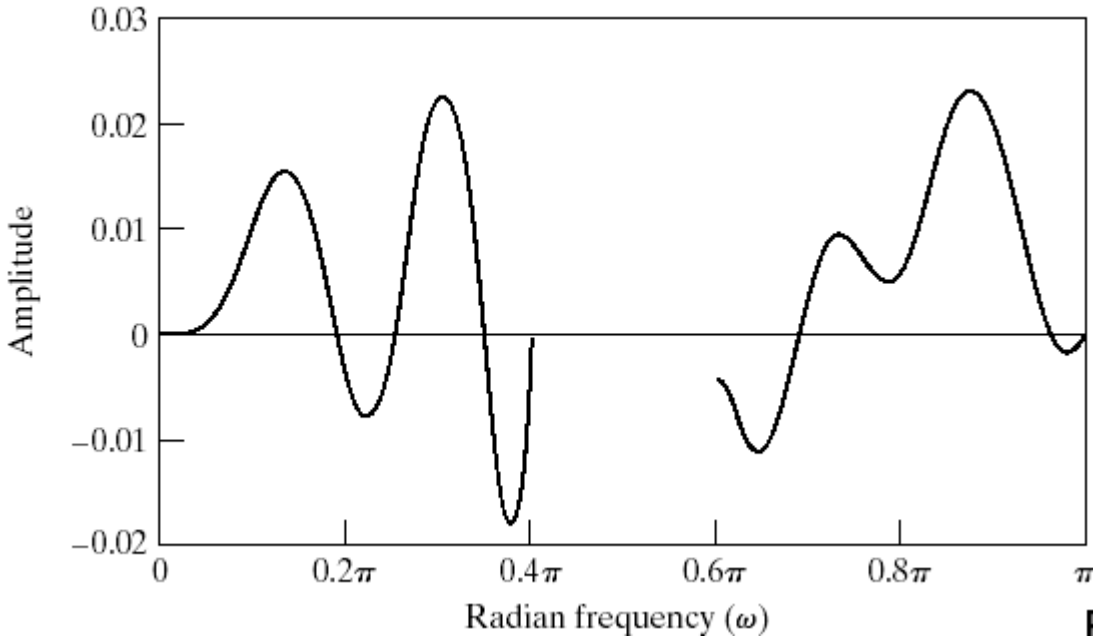


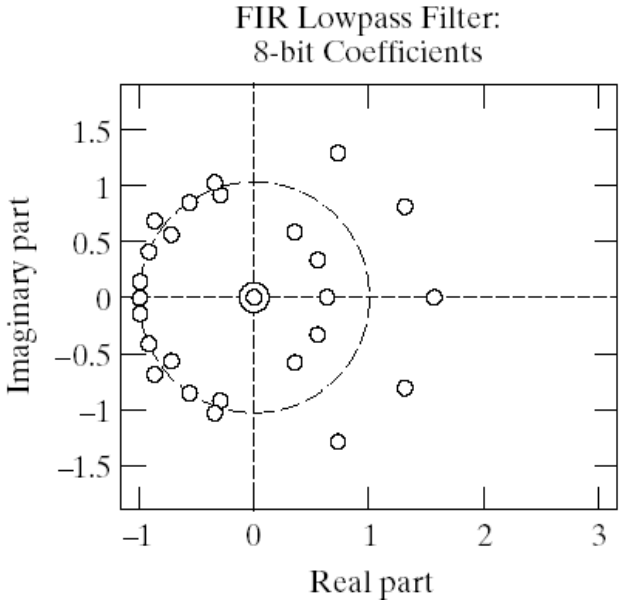
FIR filter example – 16bits





FIR filter example - 8bits





Arithmetic Operations

Finite word-length effects in arithmetic operations:

- **In linear filters, have to consider additions & multiplications**
- Addition:

  if, two B-bit numbers are added, the result has (B+1) bits.
- Multiplication:

  if a B1-bit number is multiplied by a B2-bit number, the result has (B1+B2-1) bits.

  For instance, two B-bit numbers yield a (2B-1)-bit product
- **Typically (especially so in an IIR (feedback) filter), the result of an addition/multiplication has to be represented again as a B'-bit number (e.g. B'=B). Hence have to get rid of either most significant bits or least significant bits…**

Arithmetic Operations

- **Option-1: Most significant bits**

  If the result is known to be upper bounded so that the most significant bit(s) is(are) always redundant, it(they) can be dropped, without loss of accuracy.

  This implies we have to monitor potential *overflow*, and introduce *scaling strategy* to avoid overflow.

- **Option-2 : Least significant bits**

  Rounding/truncation/… to B' bits introduces *quantization noise*.

  The effect of quantization noise is usually analyzed in a statistical manner.

  Quantization, however, is a deterministic non-linear effect, which may give rise to *limit cycle oscillations*.

Scaling

<u>The scaling problem</u>:

- **Finite word-length implementation implies maximum representable number. Whenever a signal (output or internal) exceeds this value,** *overflow* **occurs.**
- **Digital overflow may lead (e.g. in 2's-complement arithmetic) to polarity reversal (instead of saturation such as in analog circuits), hence may be very harmful.**
- **Avoid overflow through proper signal** *scaling*
- **Scaled transfer function may be c*H(z) instead of H(z) (hence need proper tracing of scaling factors)**

Scaling

Time domain scaling:

- **Assume input signal is bounded in magnitude**

$$|u[k]| \le u_{\max}$$

(i.e. u-max is the largest number that can be represented in the `words' reserved for the input signal')

- **Then output signal is bounded by**

$$|y[k]| = \left|\sum_{i=0}^{\infty} h[i].u[k-i]\right| \le \sum_{i=0}^{\infty} |h[i]|.|u[k-i]| \le u_{\max}.\sum_{i=0}^{\infty} |h[i]| = u_{\max}.\|h\|_1$$

- **To satisfy** $|y[k]| \le y_{\max}$

(i.e. y-max is the largest number that can be represented in the `words' reserved for the output signal')

we have to scale H(z) to c.H(z), with

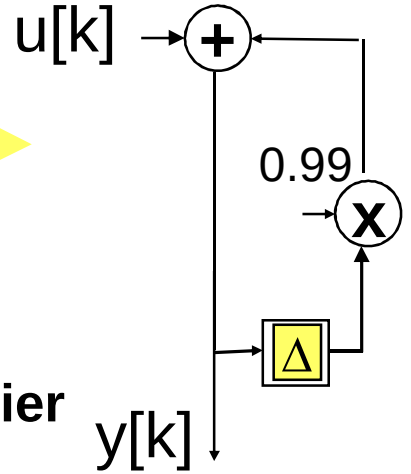
$$c = \frac{y_{\max}}{u_{\max}.\|h\|_1}$$

Scaling

- **Example:**

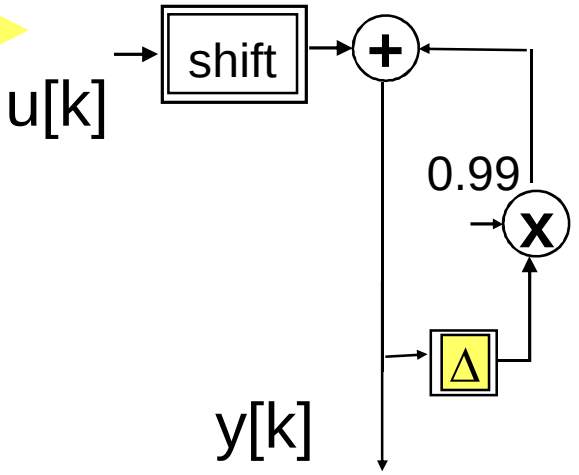
$$H(z) = \frac{1}{1 - 0.99.z^{-1}}$$

$$\|h\|_1 = \ldots = \frac{1}{1 - 0.99} = 100$$



- **assume u[k] comes from 12-bit A/D-converter**
- **assume we use 16-bit arithmetic for y[k] & multiplier**

$$c = \frac{2^{16}}{2^{12}.\|h\|_1} = 0.16 > \frac{1}{2^3}$$



- **hence inputs u[k] have to be shifted by 3 bits to the right before entering the filter (=loss of accuracy!)**

Scaling

<u>L2-scaling</u>: (`scaling in L2 sense')

- **Time-domain scaling is simple & guarantees that overflow will never occur, but often over-conservative (=too small c)**

$$c = \frac{y_{\max}}{u_{\max} \cdot \|h\|_1}$$

- **If an `energy upper bound' for the input signal is known**

$$E_{\max}^{U} = \sum_{k=0}^{\infty} |u[k]|^2$$

then L2-scaling uses

$$c = \frac{y_{\max}}{\sqrt{E_{\max}^{U}} \cdot \|h\|_2}$$

where

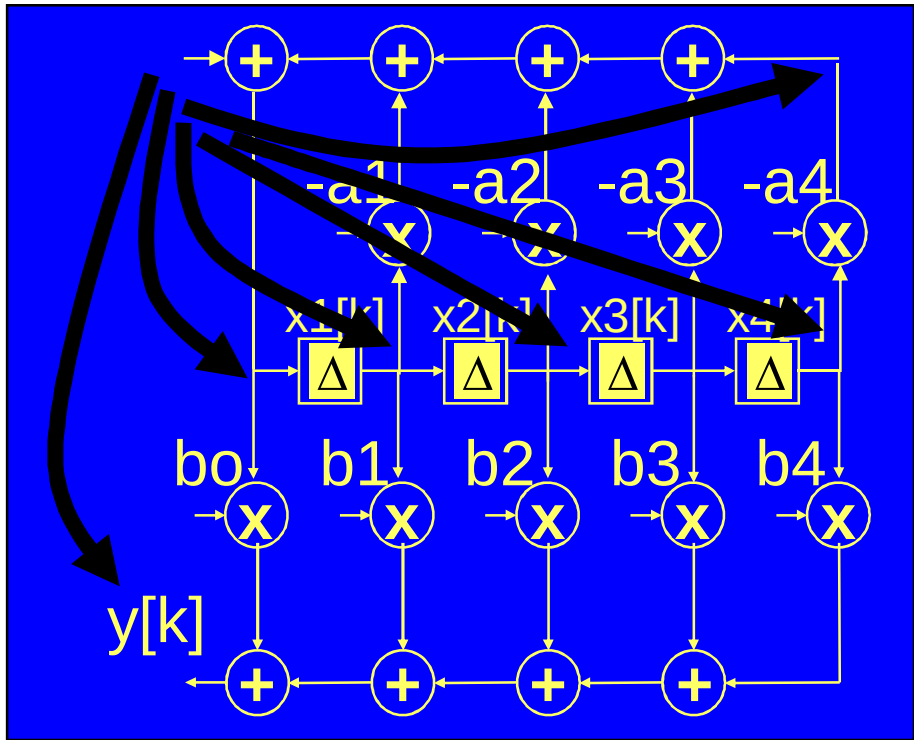
$$\|h\|_2 = \sqrt{\sum_{i=0}^{\infty} |h[i]|^2}$$

…is an L2-norm (this leads to larger c)

Scaling

- **So far considered scaling of H(z), i.e. transfer function from u[k] to y[k]. In fact we also need to consider** *overflow and scaling of each internal signal*, **i.e. scaling of transfer function from u[k] to each and every internal signal !**
- **This requires quite some thinking….**

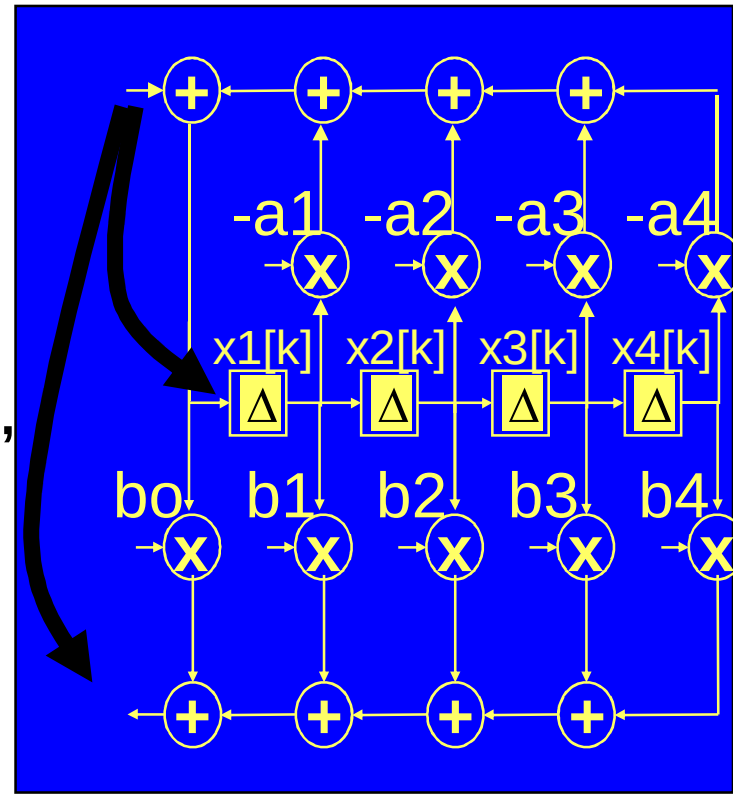
  (but doable)



Scaling

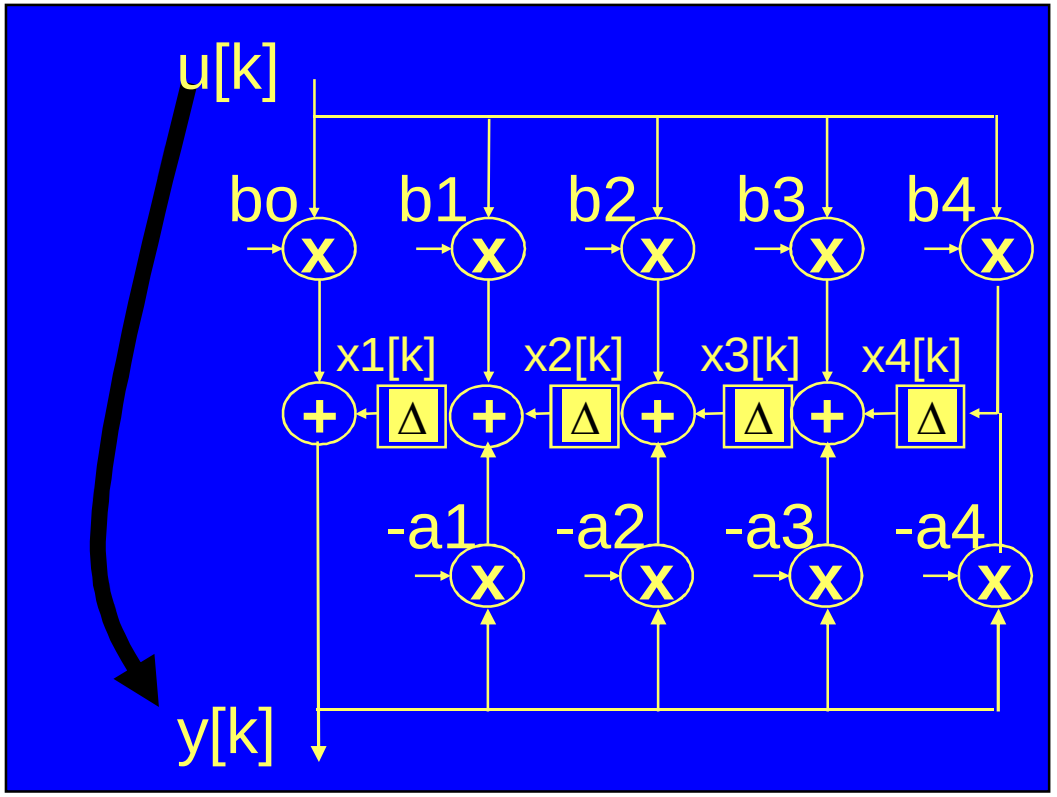
- *Something that may help:* **If 2's-complement arithmetic is used, and if the sum of K numbers (K>2) is guaranteed not to overflow, then overflows in partial sums cancel out and do not affect the final result (similar to `modulo arithmetic').**
- *Example:*

  if x1+x2+x3+x4 is guaranteed not to overflow, then if in (((x1+x2)+x3)+x4) the sum (x1+x2) overflows, this overflow can be ignored, without affecting the final result.
- **As a result (1), in a direct form realization, eventually only 2 signals have to be considered in view of scaling :**



Scaling

- **As a result (2), in a transposed direct form realization, eventually only 1 signal has to be considered in view of scaling…….:**



hence preference for transposed direct form over direct form.

Quantization Noise

The quantization noise problem :

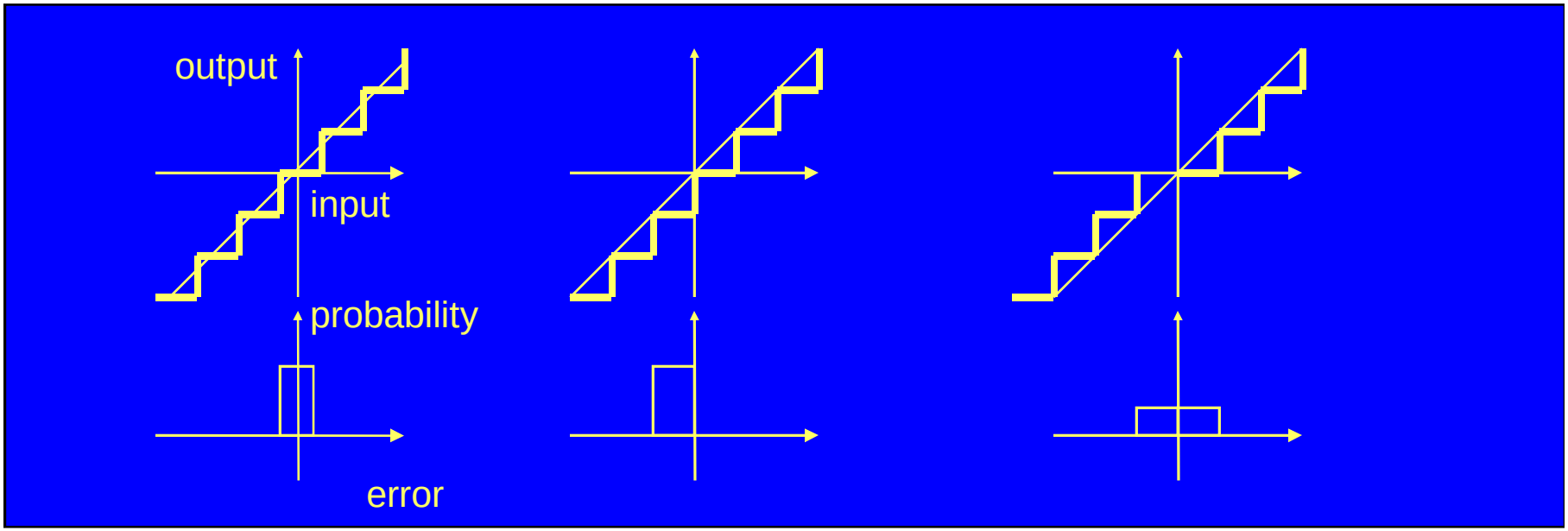
- **If two B-bit numbers are added (or multiplied), the result is a B+1 (or 2B-1) bit number. Rounding/truncation/… to (again) B bits, to get rid of the least significant bit(s) introduces** *quantization noise***.**
- **The effect of quantization noise is usually analyzed in a statistical manner.**
- **Quantization, however, is a deterministic non-linear effect, which may give rise to** *limit cycle oscillations***.**

- PS**: Will focus on multiplications only. Assume additions are implemented with sufficient number of output bits, or are properly scaled, or…**

Quantization Noise

Quantization mechanisms:

Rounding **Truncation** **Magnitude Truncation**



mean=0 **mean=(-0.5)LSB (biased!)** **mean=0**

variance=(1/12)LSB^2 **variance=(1/12)LSB^2** **variance=(1/6)LSB^2**

Quantization Noise

Statistical analysis based on the following *assumptions* **:**

- each quantization error is random, with uniform probability distribution function (see previous slide)

- quantization errors at the output of a given multiplier are uncorrelated/independent (=white noise assumption)

- quantization errors at the outputs of different multipliers are uncorrelated/independent (=independent sources assumption)

One noise source is inserted for each *multiplier***.**

Since the filter is *linear filter* **the output noise generated by each noise source is added to the output signal.**

Quantization Noise

The effect on the output signal of noise generated at a particular point in the filter is computed as follows:

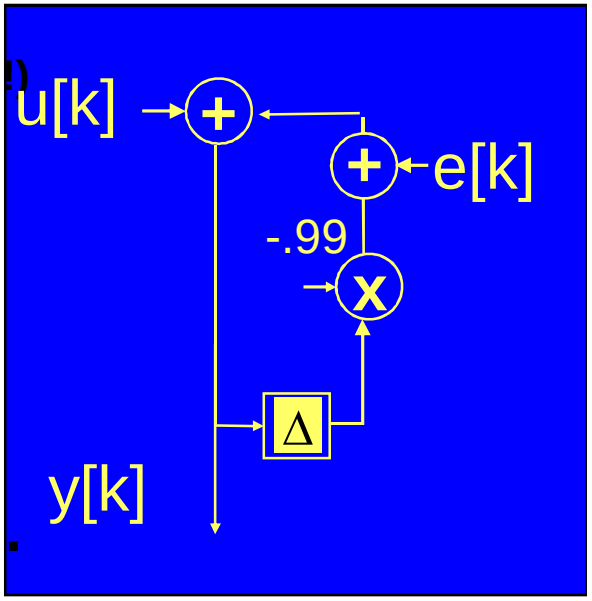
- **noise is e[k]. noise mean & variance are** μ_e, σ_e^2
- **transfer function from from e[k] to filter output is G(z),g[k] (‘noise transfer function’)**
- **Noise mean at the output is**

$$\mu_e.(\mathrm{DC}-\mathrm{gain}) = \mu_e.G(z)\big|_{z=1}$$

- **Noise variance at the output is (remember L2-norm!)**

$$\sigma_e^2.(\text{`noise - gain'}) = \sigma_e^2.\left(\frac{1}{2\pi}\int_{-\pi}^{\pi}\left|G(e^{j\omega})\right|^2 d\omega\right)$$

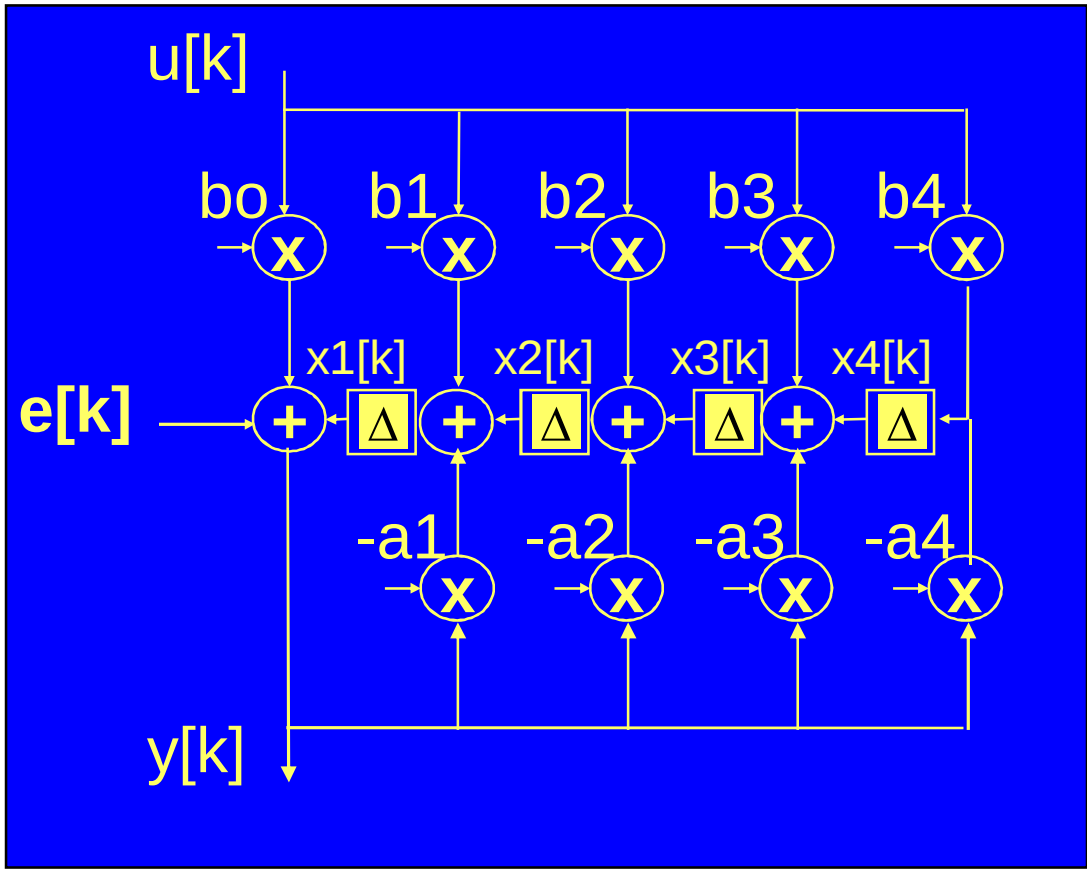
$$= \sigma_e^2.\sum_{k=0}^{\infty}\left|g[k]\right|^2 = \sigma_e^2.\|g\|_2^2$$



Repeat procedure for each noise source…

Quantization Noise

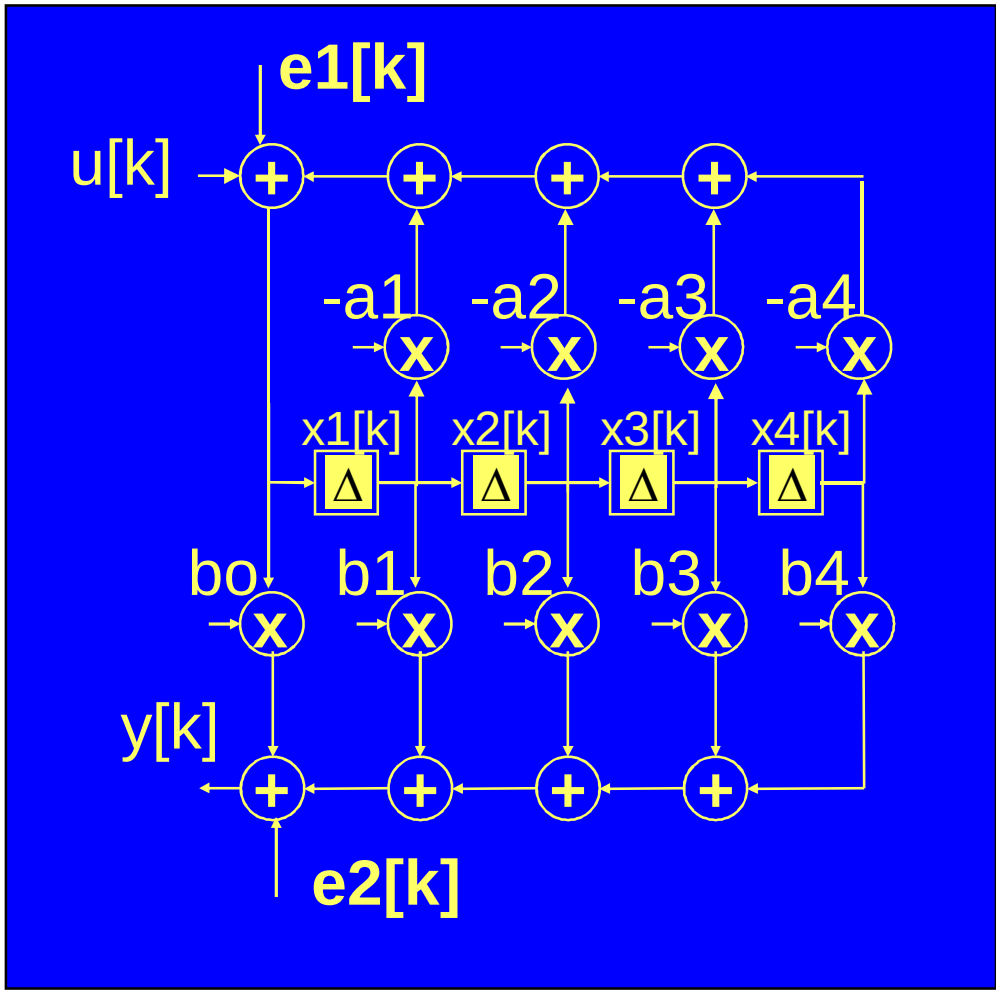
In a transposed direct realization all `noise transfer functions' are equal (up to delay), hence all noise sources can be lumped into one equivalent source



etc...

Quantization Noise

In a direct realization all noise sources can be lumped into two equivalent sources



etc...

Quantization Noise

PS: Quantization noise of *A/D-converters* can be modeled/analyzed in a similar fashion.

Noise transfer function is filter transfer function H(z).

Limit Cycles

Statistical analysis is simple/convenient, but quantization is truly a *non-linear* **effect, and should be analyzed as a** *deterministic* **process.**

Though very difficult, such analysis may reveal odd behavior:

Example: **y[k] = -0.625.y[k-1]+u[k]**

4-bit rounding arithmetic

input u[k]=0, y[0]=3/8

output y[k] = 3/8, -1/4, 1/8, -1/8, 1/8, -1/8, 1/8, -1/8, 1/8,..

Oscillations in the absence of input (u[k]=0) are called

`*zero-input limit cycle oscillations'.*

Limit Cycles

Example: **y[k] = -0.625.y[k-1]+u[k]**

4-bit truncation (instead of rounding)

input u[k]=0, y[0]=3/8

output y[k] = 3/8, -1/4, 1/8, 0, 0, 0,.. (no limit cycle!)

Example: **y[k] = 0.625.y[k-1]+u[k]**

4-bit rounding

input u[k]=0, y[0]=3/8

output y[k] = 3/8, 1/4, 1/8, 1/8, 1/8, 1/8,..

Example: **y[k] = 0.625.y[k-1]+u[k]**

4-bit truncation

input u[k]=0, y[0]=-3/8

output y[k] = -3/8, -1/4, -1/8, -1/8, -1/8, -1/8,..

Conclusion: weird, weird, weird,… !

Limit Cycles

Limit cycle oscillations are clearly *unwanted* **(e.g. may be audible in speech/audio applications)**

Limit cycle oscillations can only appear if the filter has feedback. Hence *FIR filters* **cannot have limit cycle oscillations.**

Mathematical analysis is *very difficult***.**

Truncation often helps to avoid limit cycles (e.g. *magnitude truncation***, where absolute value of quantizer output is never larger than absolute value of quantizer input (`***passive quantizer***').**

Some filter structures can be made limit cycle free, e.g. coupled realization, *orthogonal filters* **(see below).**