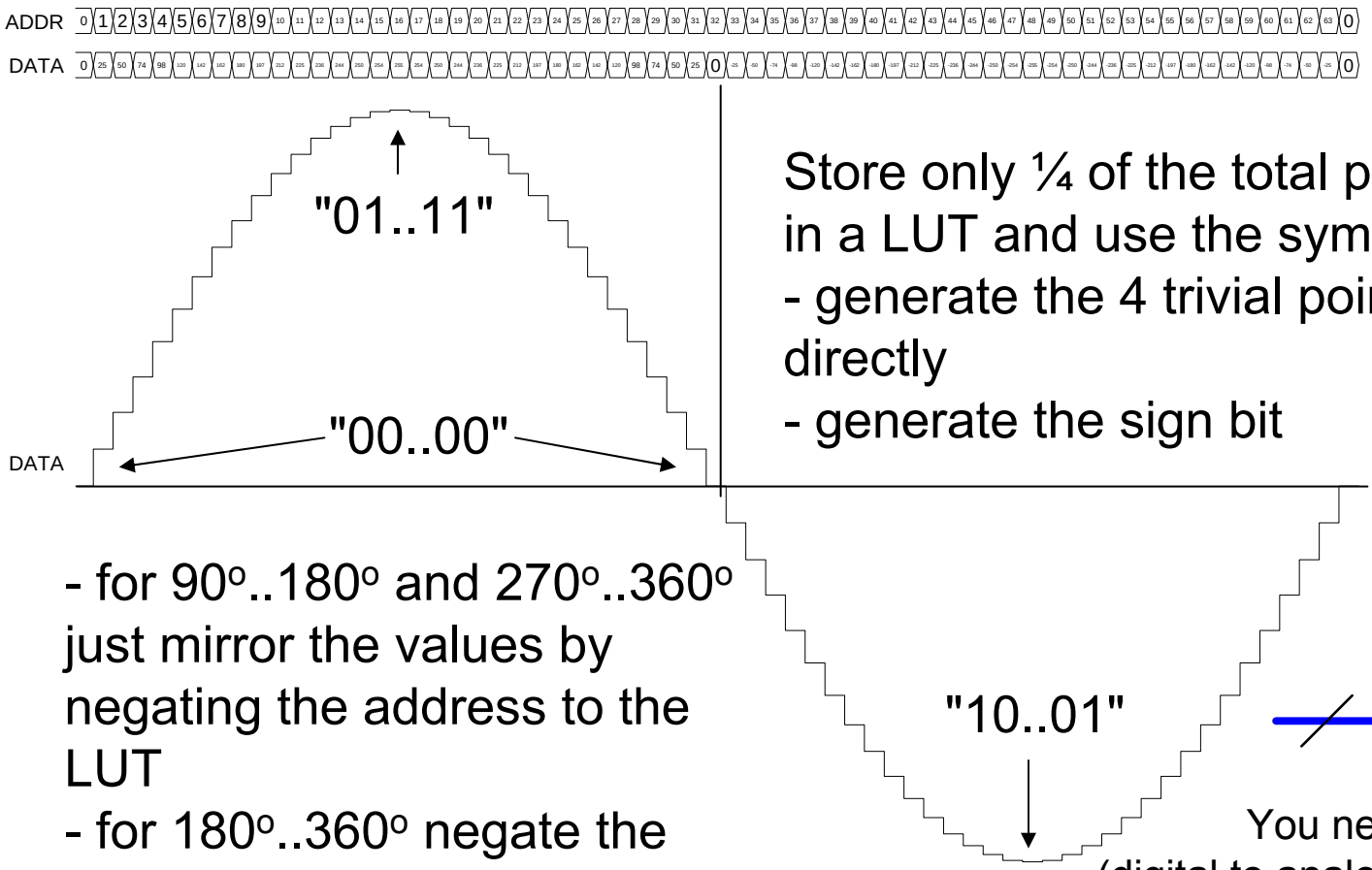
Digital Signal Processing

Digital Signal Processing

- Direct Digital Synthesis (DDS) – sine wave generation, phase accumulator
- CORDIC algorithm
- Digital Filters (linear time-invariant)
  - Finite Impulse Response (FIR), distributed arithmetic
    - Examples – moving integrator, tail cancellation
  - Infinite Impulse Response (IIR)
    - Examples – lossy integrator, rounding problems

DDS - Generating a sine wave(1)



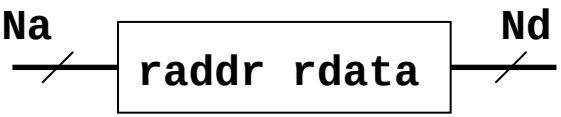
Store only ¼ of the total period in a LUT and use the symmetry:
- generate the 4 trivial points directly
- generate the sign bit

- for 90^{o}..180^{o} and 270^{o}..360^{o} just mirror the values by negating the address to the LUT
- for 180^{o}..360^{o} negate the output of the LUT

You need also a DAC (digital to analogue converter)

DDS - Generating a sine wave(2)

```
constant sub0         : std_logic_vector(Na-3 downto 0) := (others => '0');
signal suba, suba2lut : std_logic_vector(Na-3 downto 0);
signal rdata_lut      : std_logic_vector(Nd-2 downto 0);
signal sel            : std_logic_vector(2 downto 0);
...
lut0to90: sin_lut90 -- table for 0 to 90 deg generated by a C++ program
generic map(Na => Na-2, Nd => Nd-1)
port map(raddr => suba2lut, rdata => rdata_lut);

 suba <= raddr(Na-3 downto 0);
 suba2lut <= suba when raddr(Na-2) = '0' else (not suba) + 1; -- suba or -suba
 sel <= '1' & raddr(Na-1 downto Na-2) when suba = sub0 else
        '0' & raddr(Na-1 downto Na-2);

 process(sel, rdata_lut)
 begin
   rdata <= (others => '0'); -- used in "100", "110", partly in "111"
   case sel is
   when "101" => rdata <= (others => '1');  -- 90 deg
                 rdata(Nd-1) <= '0';        -- +(2**Nd-1)
   when "111" => rdata(Nd-1) <= '1';        -- 270 deg
                 rdata(0)    <= '1';        -- -(2**Nd-1)
   when "000"|"001" => rdata <= '0' & rdata_lut;              --  LUT
   when "010"|"011" => rdata <= '1' & ((not rdata_lut) + 1); -- -LUT
   when others => NULL;
   end case;
 end process;
end;
```



Two address bits and one data bit less

2^{Na-2}-suba

sign bit

Size of the design for **Nd=8+1**:

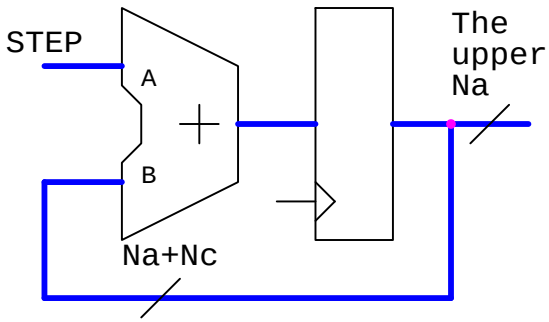
Na	LUT4
6	34
7	48
8	55

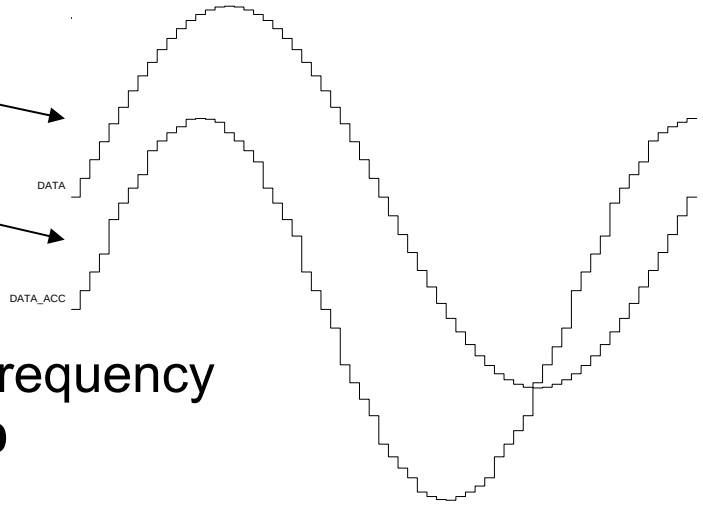
DDS - Generating a sine wave(3)

```
signal addr, addr_acc : std_logic_vector(Na-1 downto 0) := (others => '0');
signal acc, step      : std_logic_vector(Na+Nc-1 downto 0) := (others => '0');
signal data_acc, data : std_logic_vector(Nd-1 downto 0);
...
  step <= conv_std_logic_vector(5, step'length); -- 5/4
  process(clk)
  begin
    if clk'event and clk='1' then
       addr <= addr + 1;
       acc  <= acc  + step;
     end if;
  end process;
  addr_acc <= acc(acc'high downto acc'high-addr_acc'length+1);
dut: sin_lut
generic map(Na => Na, Nd => Nd)
port map(raddr => addr, rdata => data);
dut1: sin_lut
generic map(Na => Na, Nd => Nd)
port map(raddr => addr_acc, rdata => data_acc);
```

Na=6
Nd=9
Nc=2

the upper bits in the **phase accumulator**





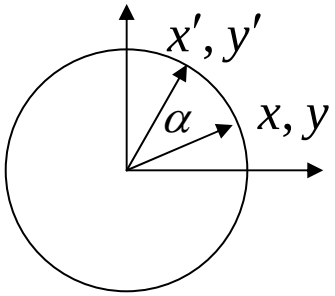
Modify the frequency by the **`step`** parameter

CORDIC

- **CO**ordinate **R**otation **DI**gital **C**omputer
- Developed to calculate trigonometric functions
- Widely used by the militaries, then in the pocket calculators, math-coprocessors (8087), communications
- The original method can directly calculate *sin*, *cos*, *arctan*, $\sqrt{x^2 + y^2}$
- Extended to hyperbolic functions

CORDIC – how it works(1)

- The idea is simple, rotate a vector by a series of predefined angles, at each step only the rotation direction must be selected



$$x' = x \cdot \cos\alpha - y \cdot \sin\alpha$$
$$y' = x \cdot \sin\alpha + y \cdot \cos\alpha$$

- All rotation angles α_n have $\tan\alpha_n = 2^{-n}, n = 0..$
- The rotation matrix divided by $\cos\alpha$ is simple: $\begin{pmatrix} 1 & -\tan\alpha \\ \tan\alpha & 1 \end{pmatrix}$

- After all rotation steps are done, the radius is scaled by a fixed factor, depending only on the number of the steps
- At each step ***x, y*** and the new **angle** are calculated using only **shift**, **add**/**subtract**, the direction of the rotation is selected properly (**compare**)

CORDIC – how it works(2)

$$\tan \alpha_n = 2^{-n}$$

n	α_n	$\sum_{k=0}^{n} \alpha_k$	$\prod_{k=0}^{n} \frac{1}{\cos \alpha_k}$
0	45.00	45.00	1.414
1	26.56	71.56	1.581
2	14.04	85.60	1.630
3	7.12	92.73	1.642
4	3.58	96.30	1.646
5	1.79	98.09	1.646
6	0.90	98.99	1.647
7	0.45	99.44	1.647
8	0.22	99.66	1.647
9	0.11	99.77	1.647

the possible range

the scaling of the radius

CORDIC in VHDL (1)

Example: calculate *sin* and *cos*. The vector lies originally on the **x** axis and starts with smaller length to compensate for the scaling. The angle has 3 bits more, the coordinates have 2 bits more.

```
entity cordic is
generic (steps : Integer := 9);
port (clk     : in  std_logic;
      clr     : in  std_logic;
      validi  : in  std_logic;
      w       : in  std_logic_vector(6 downto 0);  -- 0 to 90 deg
      valido  : out std_logic;
      sin     : out std_logic_vector(7 downto 0);  -- 0 to 255
      cos     : out std_logic_vector(7 downto 0)); -- 0 to 255
end cordic;
architecture a of cordic is
type angle_arr is array (0 to 9) of Integer range 0 to 511;
constant angles : angle_arr := (360, 213, 112, 57, 29, 14, 7, 4, 2, 1);
constant x_ini  : Integer := 620;
subtype xy_type is Integer range -1024 to 1023;
subtype w_type  is Integer range -1024 to 1023;
type xy_arr is array(0 to steps) of xy_type;
type w_arr  is array(0 to steps) of w_type;
signal angle    : w_arr;
signal x, y     : xy_arr;
signal sin_i, cos_i : Integer range 0 to 255;
signal valid    : std_logic_vector(0 to steps);
```

the rotation angles*8

the initial length, 620 instead of 1023

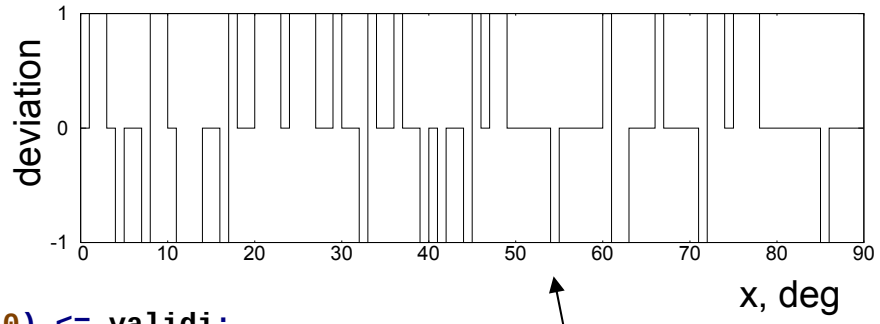
the result in 8 bit

CORDIC in VHDL (2)

```
process(clk)
begin
  if rising_edge(clk) then
    if clr='1' then
      for i in 0 to steps loop
        x(i) <= 0; y(i) <= 0; angle(i) <= 0;
      end loop;
      x(0) <= x_ini;
      valid <= (others => '0'); valido <= '0';
    else
      angle(0) <= conv_integer('0' & w)*8; valid(0) <= validi;
      for i in 1 to steps loop
        valid(i) <= valid(i-1);
        if angle(i-1) > 0 then
          x(i) <= x(i-1) - y(i-1)/2**(i-1);
          y(i) <= y(i-1) + x(i-1)/2**(i-1);
          angle(i) <= angle(i-1) - angles(i-1);
        else
          x(i) <= x(i-1) + y(i-1)/2**(i-1);
          y(i) <= y(i-1) - x(i-1)/2**(i-1);
          angle(i) <= angle(i-1) + angles(i-1);
        end if;
      end loop;
      valido <= valid(steps);
      if x(steps) < 0 then cos_i <= 0; else cos_i <= x(steps)/4; end if;
      if y(steps) < 0 then sin_i <= 0; else sin_i <= y(steps)/4; end if;
    end if;
  end if;
end process;
cos <= conv_std_logic_vector(cos_i, cos'length);
sin <= conv_std_logic_vector(sin_i, sin'length);
```

rotate left

rotate right



The deviation from the sine function, Note that the sin(x) is an 8 bit unsigned integer

copy the result to the output registers

remove the extra bits

Digital filters – LTI

- **L**inearity

If $y_1[n]$ and $y_2[n]$ are the responses of the filter to $x_1[n]$ and $x_2[n]$, then the response to $a_1 \cdot x_1[n] + a_2 \cdot x_2[n]$ is $a_1 \cdot y_1[n] + a_2 \cdot y_2[n]$

- **T**ime **I**nvariance

If $y[n]$ is the response of the filter to $x[n]$, then the response to $x[n-k]$ is $y[n-k]$

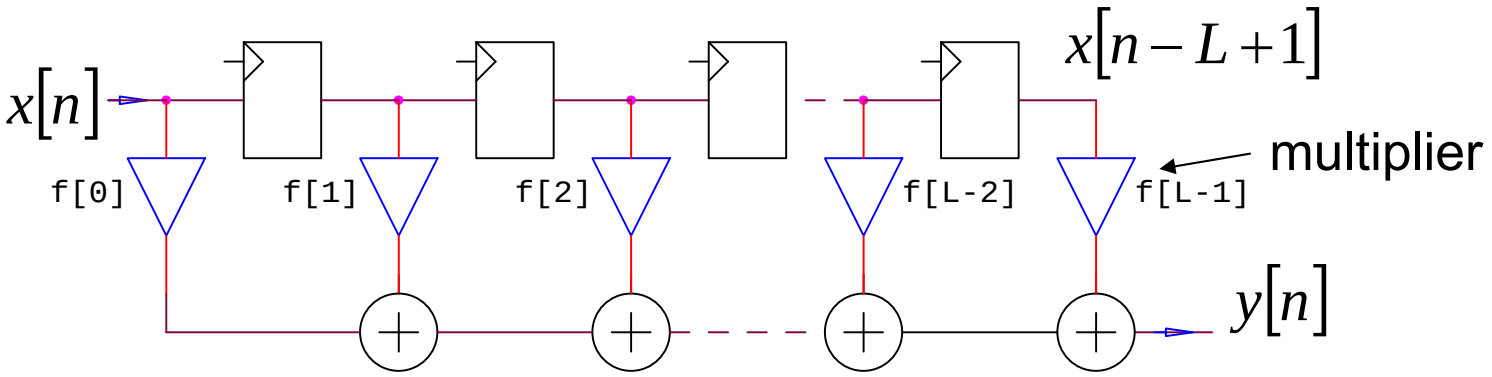
Digital filters – FIR ↔ IIR

- The reaction of a linear time-invariant (LTI) filter to an input signal $x[n]$ is the convolution

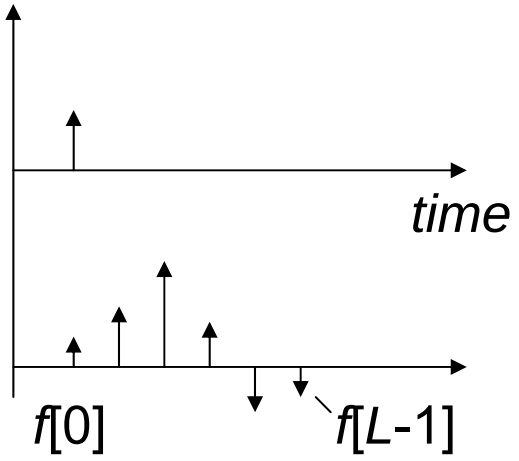
$$y[n] = x[n] * f[n] = \sum_k f[k] \cdot x[n-k]$$

- In general the filters are classified as being
  - *finite impulse response* (FIR), where the sum is over a finite number of samples
  - *infinite impulse response* (IIR)

FIR digital filters(1)

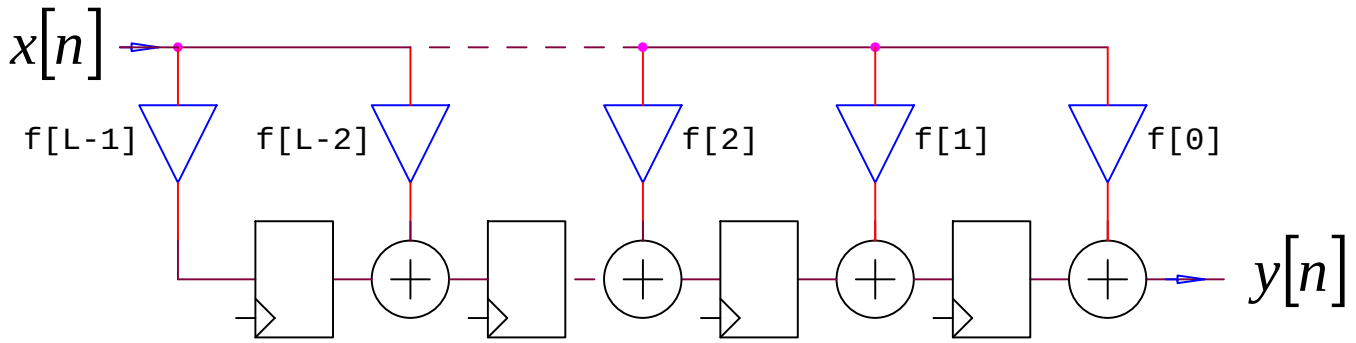


$$y[n] = \sum_{k=0}^{L-1} f[k] \cdot x[n-k]$$



- The time and the signal are discrete
- The length *L* is finite
- The filter coefficients *f*[*k*] are constant (but might be programmable)
- *f*[*k*] represent the **finite** reaction of the filter on a "delta" input impulse
- The straightforward implementation requires L multipliers, adders and registers (pipeline)

FIR digital filters(2)



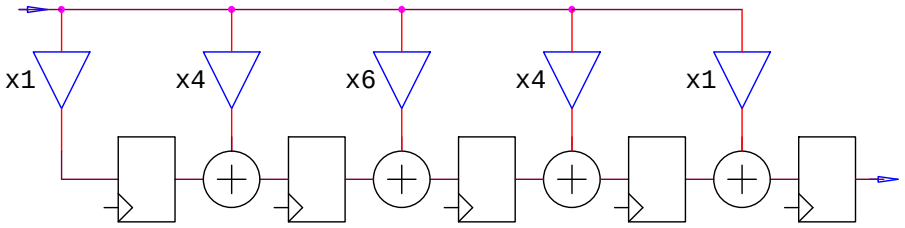
- The transposed FIR filter is mathematically the same, but the adders are automatically pipelined
- Eventual symmetry in the filter coefficients can be used to minimize the number of the multipliers
- In case of constant coefficients the multiplier can share common parts:

  x 9 = x 8 + x1 (1 adder)

  x 11 = x 9 + x 2 (1 adder more)
- Note that all negative coefficients can be converted to positive by replacing the adder ⊕ with subtractor ⊖

FIR digital filters(3)

5 tap filter with binomial coefficients




```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
entity fir5tap is
generic(N : Positive := 8);
port(
  clk  : in  std_logic;
  din  : in  std_logic_vector(N-1 downto 0);
  dout : out std_logic_vector(N-1 downto 0) );
end fir5tap;
…
type taps_type is array(0 to 4) of std_logic_vector(N+3 downto 0);
signal taps : taps_type;
begin
 process(clk)
 begin
  if rising_edge(clk) then
   taps(0) <= "0000" & din;                                          -- y0
   taps(1) <= taps(0) + ("00" & din & "00");                         -- + 4*y1
   taps(2) <= taps(1) + ( ("00" & din & "00") + ("000" & din & '0') ); -- + 4*y2+2*y2
   taps(3) <= taps(2) + ("00" & din & "00");                         -- + 4*y3
   taps(4) <= taps(3) + ("0000" & din);                              -- + y4
  end if;
 end process;
 dout <= taps(4)(N+3 downto 4);
end;
```

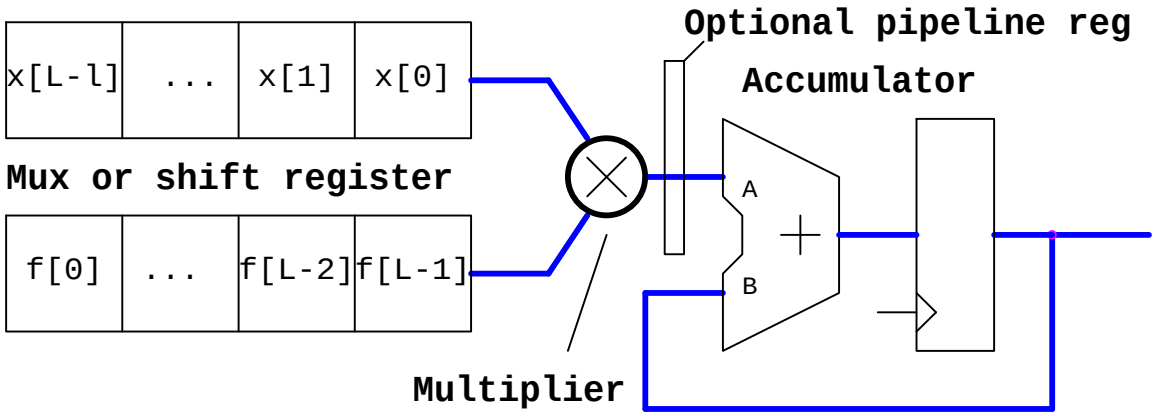
4 bits more

output shifted right

FIR digital filters(4)

- The realisation of FIR filters in FPGAs depends on the availability of hardcoded multipliers (the so called DSP blocks) and on other requirements
- When many clock cycles are available for the calculation of one sample, two strategies are applicable to reduce the resource usage:
  - Only 1 multiplier + accumulator to calculate the sum of the products (MAC) sequentially in a loop
  - Use distributed arithmetic (DA) which requires a LUT + accumulator, again calculate in a loop

FIR digital filters - MAC



Use the hardcoded multiplier (if available) or instantiate a technology specific multiplier core!

For every sample:

- clear the accumulator (not shown here)
- repeat **L** times the loop
- store the result

The size of the accumulator should be selected so that in worst case no overflow occurs

The execution time is proportional to the number of the coefficients

FIR digital filters – DA(1)

- The distributed arithmetic method is better than the MAC when the number of the bits in the input is less than the number of the coefficients
- The idea is to calculate offline the sum of products of any L x 1 bit numbers with all coefficients:

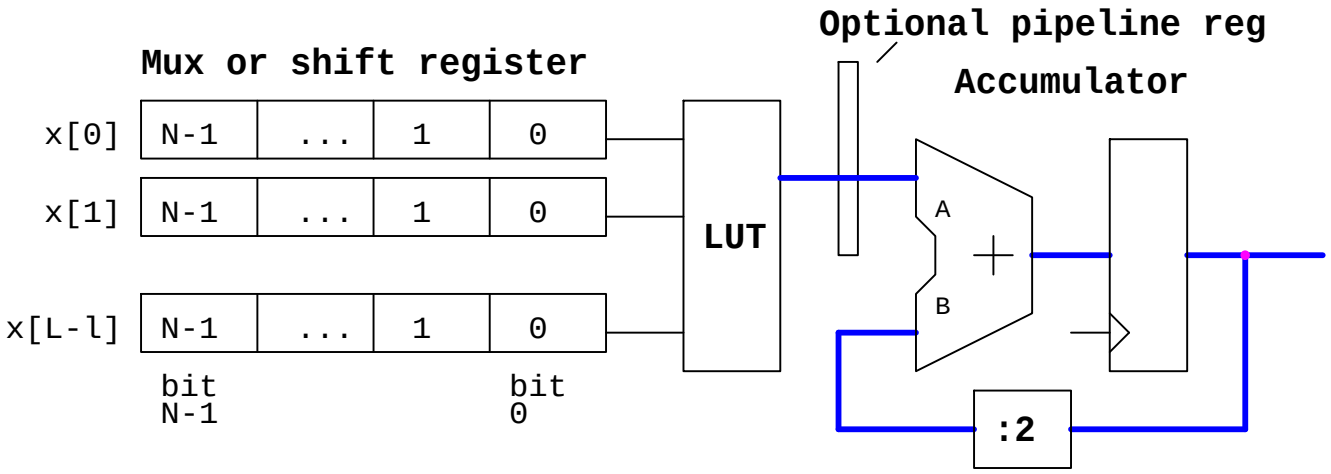
$$LUT(b_0, b_1, ..., b_{L-1}) = \sum_{k=0}^{L-1} b_k \cdot f[k], \text{ where } b_k = 0, 1$$

and to store the table in a ROM

FIR digital filters – DA(2)

$$\sum_{k=0}^{L-1} x[k]\cdot f[L-k-1] = \sum_{b=0}^{N-1}\sum_{k=0}^{L-1} 2^b \cdot x_b[k]\cdot f[L-k-1] =$$

$$= \sum_{b=0}^{N-1} 2^b \cdot LUT\left(x_b[L-1],\ldots, x_b[1], x_b[0]\right), \text{ where } x_b[k] \text{ is bit } b \text{ of } x[k]$$



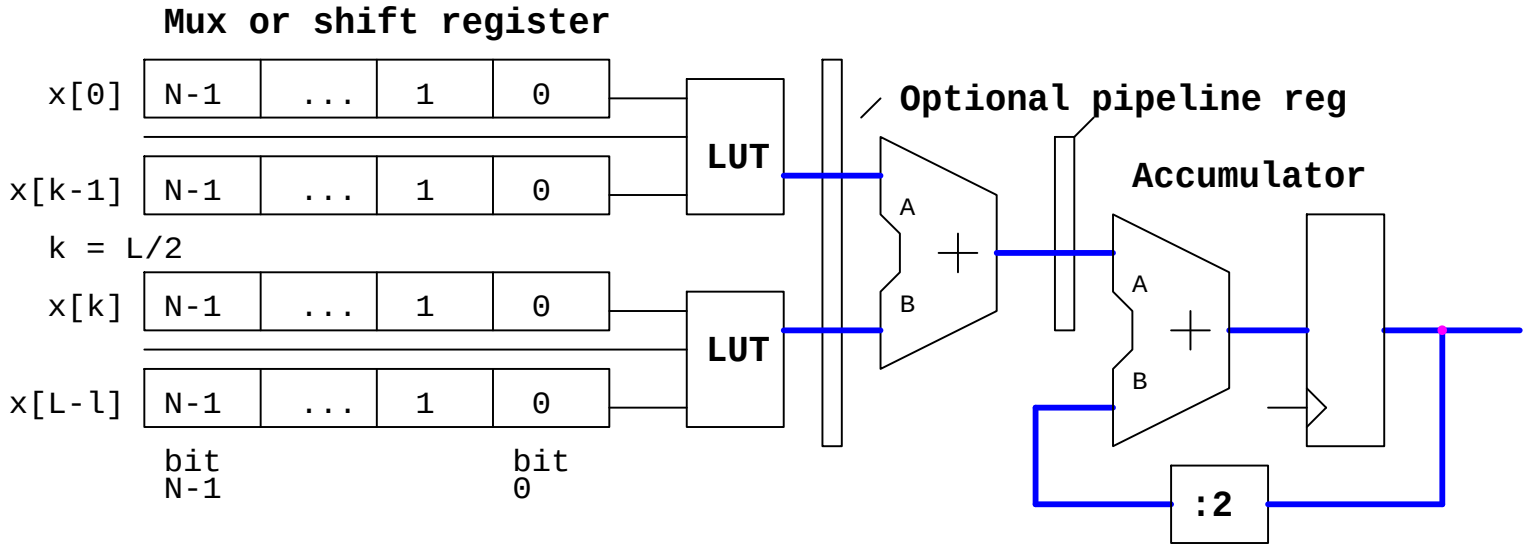
It is more convenient to start with the LSB and then shift right the accumulator output after each bit

FIR digital filters – DA(3)

- Note that the filter coefficients are still programmable if the LUT is programmable (like a RAM block), but the software doing this should recalculate the LUT
- The DA implementation is faster than the MAC in case of L > N (more coefficients than bits in the input signal)
- Some variants to reduce the size of the LUTs or the number of cycles are shown on the next slides

FIR digital filters – DA(4)

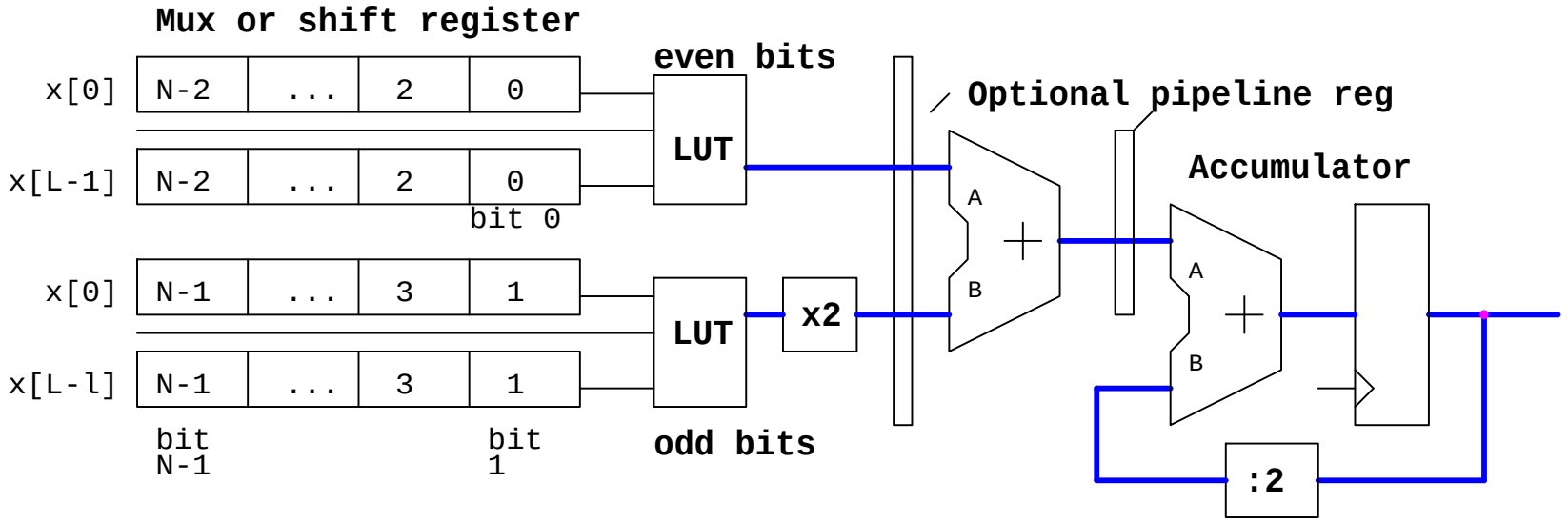
- If the LUT becomes too large, it can be split in many parts, then the output of all LUTs should be added together



Note that the LUTs have the same content, which can be used if the technology has RAM blocks with multiple read ports!

FIR digital filters – DA(5)

- To reduce the number of cycles, the odd and even bits can be processed parallel in time and the results added properly:



Note that the LUTs have the same content, which can be used if the technology has RAM blocks with multiple read ports!

Moving integrator(1)

… is a FIR filter with all coefficient equal to 1

$$y[n] = \sum_{k=0}^{L-1} f[k] \cdot x[n-k], \text{ where all } f[k] = 1$$

The standard FIR architecture would require L adders. A better solution is to use an accumulator and a pipeline

```
...
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
entity mov_int is
generic(Nd : Integer := 8;  -- data width
        Na : Integer := 2); -- window size is 2**Na samples
port (clk : in  std_logic;
      clr : in  std_logic;
      x   : in  std_logic_vector(   Nd-1 downto 0);   -- input data
      y   : out std_logic_vector(Na+Nd-1 downto 0));  -- output
end mov_int;
architecture a of mov_int is
signal diff  : std_logic_vector(Nd      downto 0);
signal diffe : std_logic_vector(Na+Nd-1 downto 0);
signal acc   : std_logic_vector(Na+Nd-1 downto 0);
type pipe_arr is array(0 to 2**Na-1) of std_logic_vector(Nd-1 downto 0);
signal pipe : pipe_arr;
```

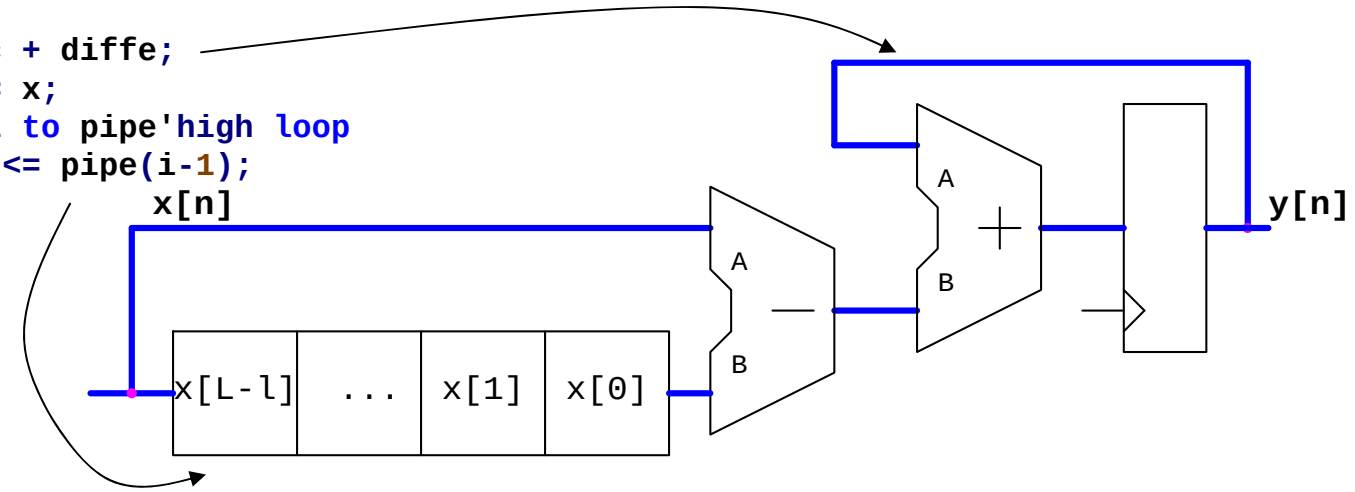
Moving integrator(2)

```
diff <= ('0' & x) - ('0' & pipe(pipe'high));
process(diff)
begin
    diffe <= (others => diff(Nd)); -- fill with the sign bit
    diffe(Nd-1 downto 0) <= diff(Nd-1 downto 0);
end process;
process(clk) -- the accumulator
begin
  if rising_edge(clk) then
    if clr='1' then
      acc <= (others => '0');
      for i in pipe'range loop
        pipe(i) <= (others => '0');
      end loop;
    else
      acc <= acc + diffe;
      pipe(0) <= x;
      for i in 1 to pipe'high loop
        pipe(i) <= pipe(i-1);
      end loop;
    end if;
  end if;
end process;
y <= acc;
```

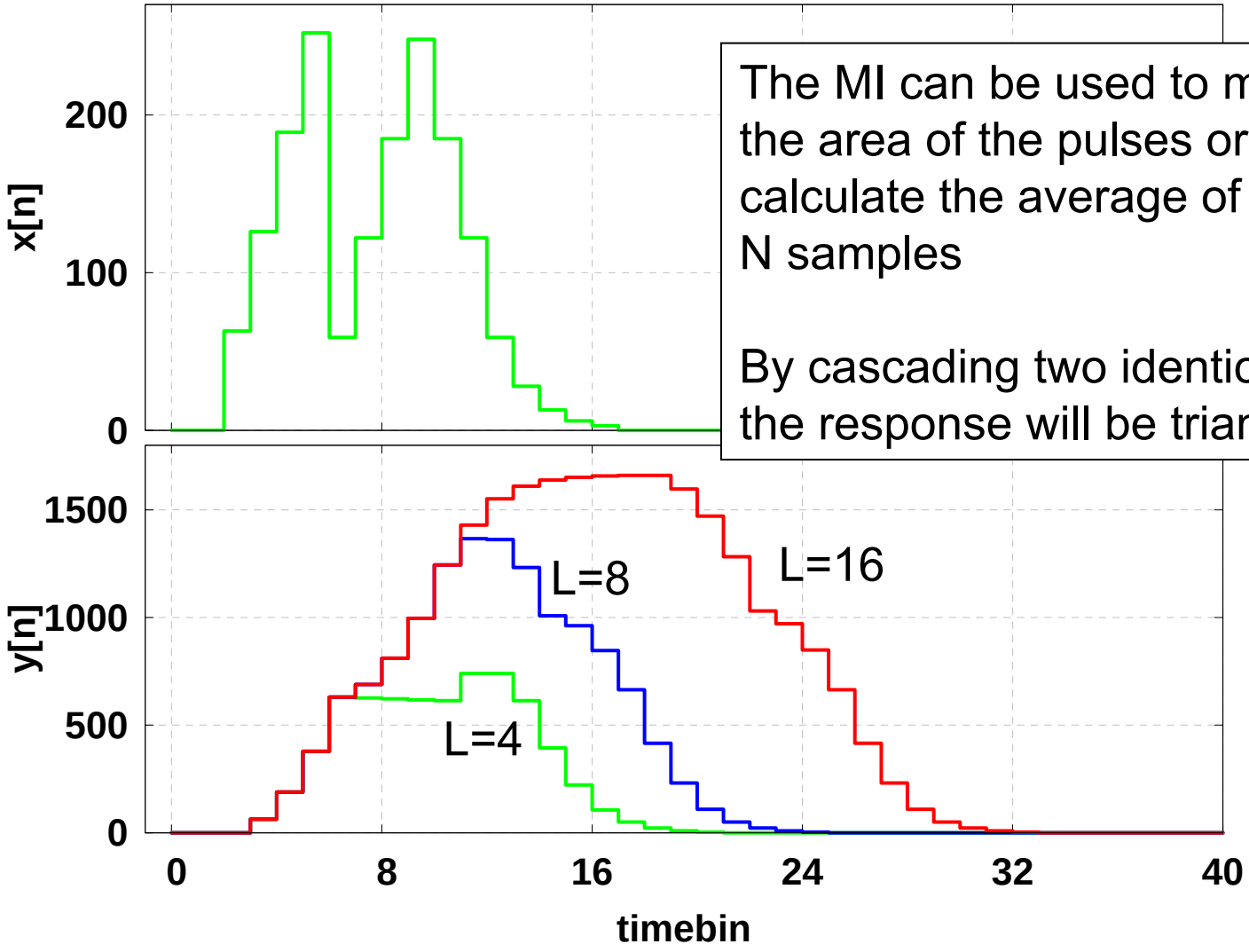
one bit more

clear

Deeper pipelines can be realised as FIFO memory



Moving integrator – simulation

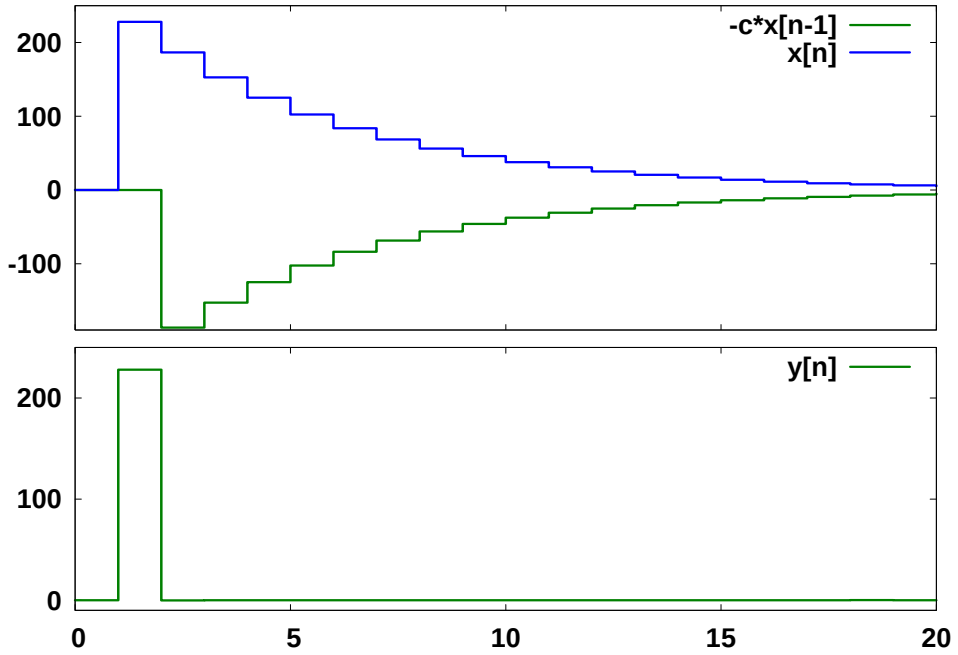


The MI can be used to measure the area of the pulses or to calculate the average of the last N samples

By cascading two identical MIs the response will be triangular

Tail cancellation filter (TC)

- The response function of a detector is typically a convolution of the signal we want to measure with something undesired
- The idea is to subtract the undesired signal from the input
- In the example only the step of the signal contains useful information



- The long tail comes from the slow ions drifting in the chamber

Tail cancellation (code)

```
port (clk : in  std_logic;
      clr : in  std_logic;
      x   : in  std_logic_vector(Nd-1 downto 0);   -- input data
      c   : in  std_logic_vector(Nc-1 downto 0);   -- coeff = c/2**Nc
      y   : out std_logic_vector(Nd-1 downto 0));  -- output
…
signal diffe  : std_logic_vector(Nd      downto 0);
signal prod   : std_logic_vector(Nd+Nc-1 downto 0);
signal xold_c : std_logic_vector(Nd   -1 downto 0);
begin
  diffe <= ('0' & x) - ('0' & xold_c);
  y <= diffe(y'range) when diffe(Nd)='0' else (others => '0');
  prod <= x * c;
  process(clk)
  begin
    if rising_edge(clk) then
      if clr = '1' then
        xold_c <= (others => '0');
      else
        xold_c <= prod(Nd+Nc-1 downto Nc);
      end if;
    end if;
  end process;
end;
```

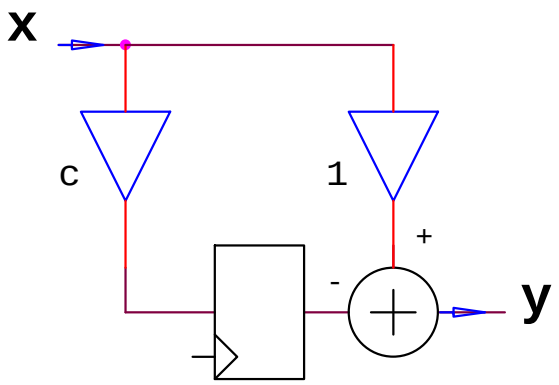
x, **c** and **y** interpreted as unsigned here

← one bit more

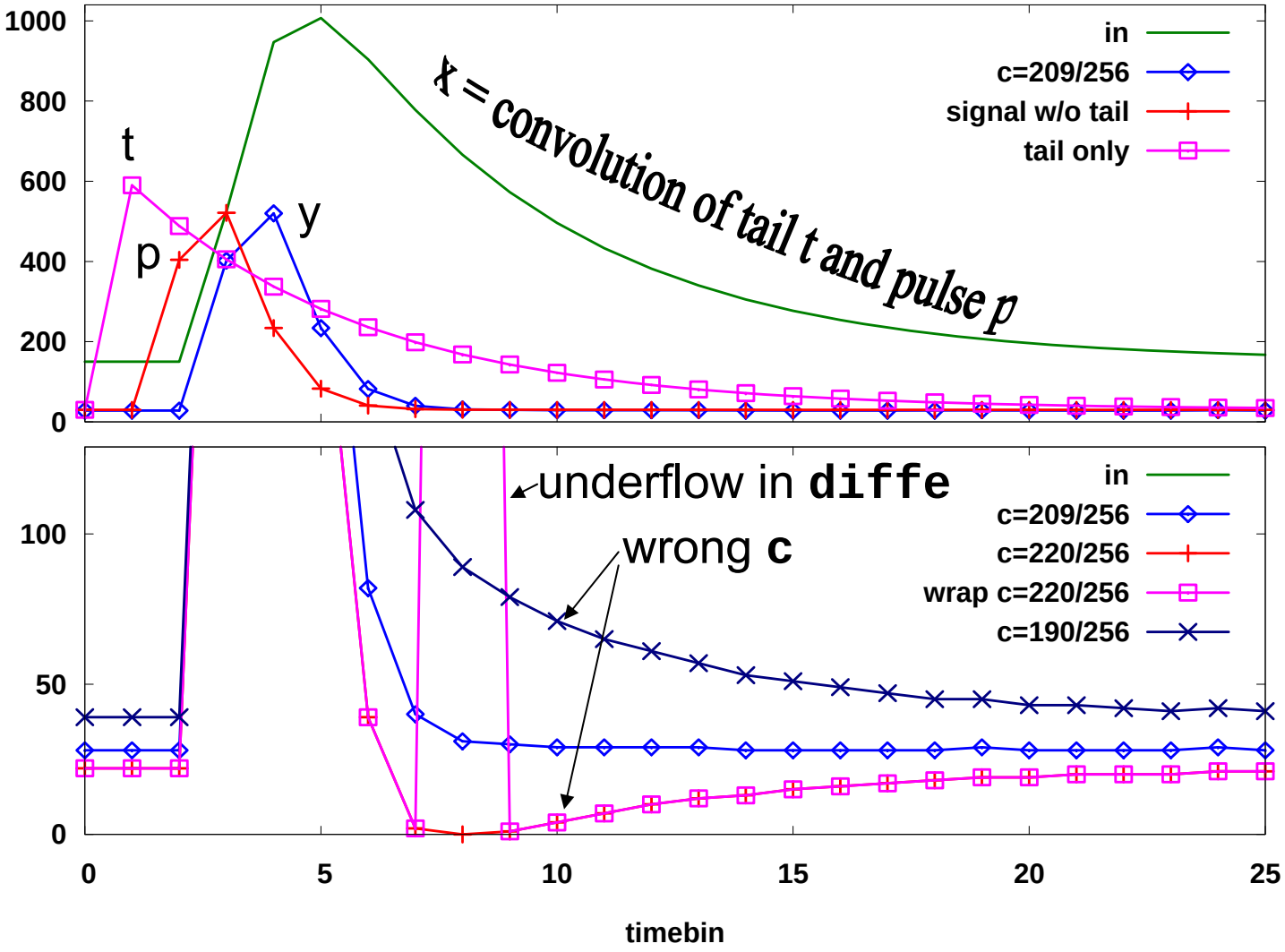
If **x < xold_c** ???

clip to 0 instead of wrap to some high value

take the MSBits



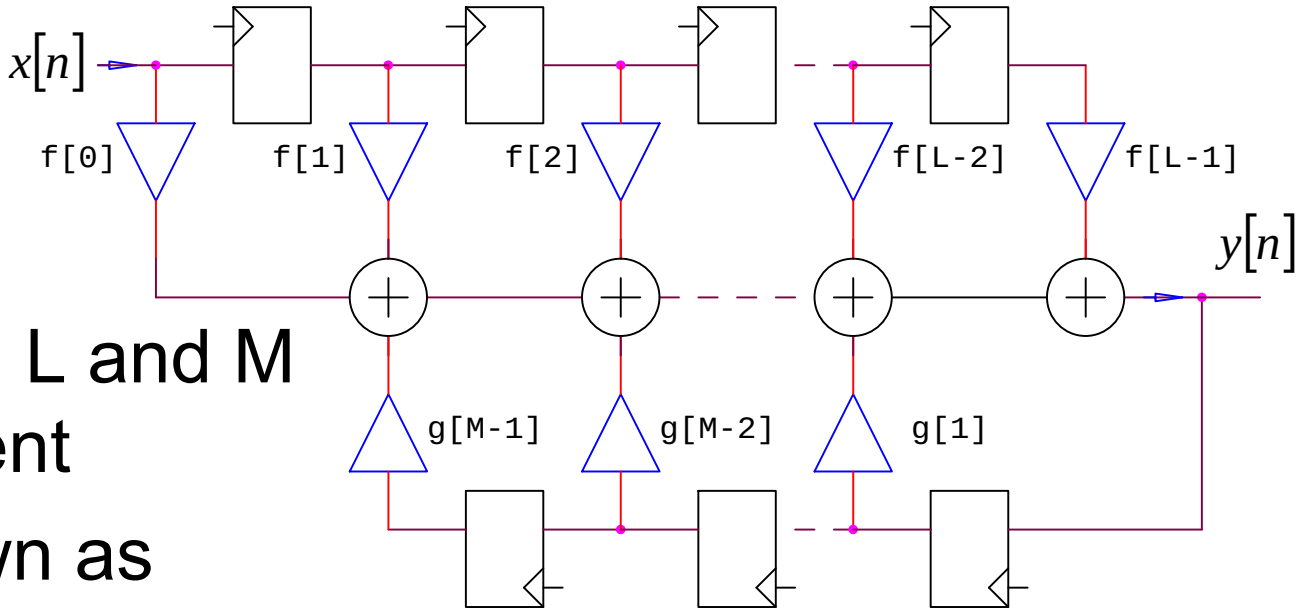
Tail cancellation – simulation



Nd=10
Nc= 8

IIR digital filters

- Infinite impulse response, achieved by a feedback



- In general L and M are different
- Also known as *recursive filter*

$$y[n] = \sum_{k=0}^{L-1} f[k] \cdot x[n-k] + \sum_{k=1}^{M-1} g[k] \cdot y[n-k]$$

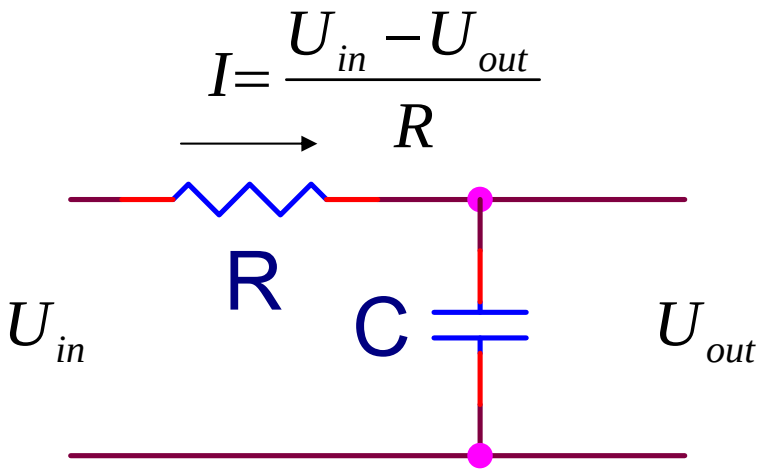
IIR – lossy integrator (LI)

- Simple example: in many cases the result of a single measurement is too noisy and an average over some time is preferred
- The lossy integrator (or relaxation filter) takes a weighted sum of the old output and the new input value:

$$y[n+1] = \frac{m-1}{m} y[n] + \frac{1}{m} x[n]$$

- In case of $m=2^K$ it can be easily realised without expensive multiplier or divider

Lossy Integrator = low-pass



$$dU_{out}(t) = I \cdot dt / C =$$

$$= \frac{1}{\underbrace{R \cdot C}_{\tau}} (U_{in} - U_{out}) \cdot dt$$

In the case of discrete time we get:

$$U_{out}[n+1] - U_{out}[n] = \frac{1}{\tau}(U_{in}[n] - U_{out}[n])$$

$$\Downarrow$$

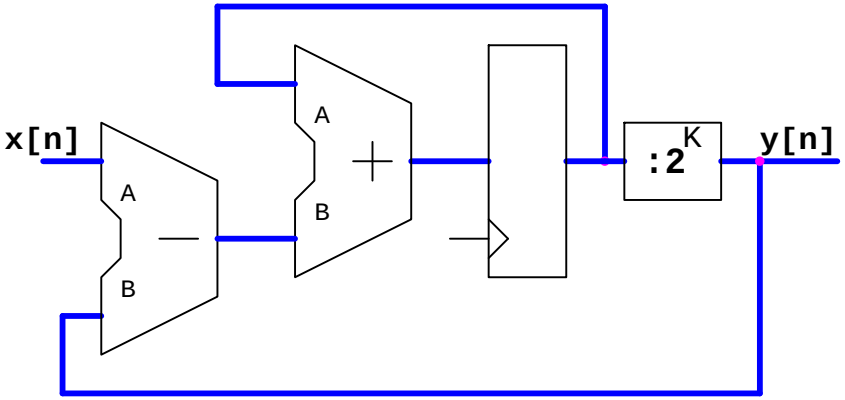
$$U_{out}[n+1] = \frac{\tau - 1}{\tau} U_{out}[n] + \frac{1}{\tau} U_{in}[n]$$

LI block diagram

- With some rearrangement we get:

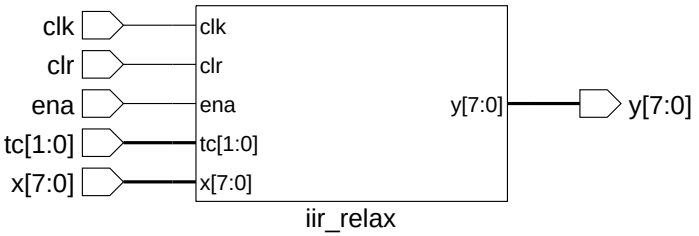
$$2^k y[n+1] = 2^k y[n] - y[n] + x[n] \Rightarrow 2^k \left(y[n+1] - y[n]\right) = x[n] - y[n]$$

- This is simple for implementation using only adders



- By varying **k** one can adjust the response time of the filter

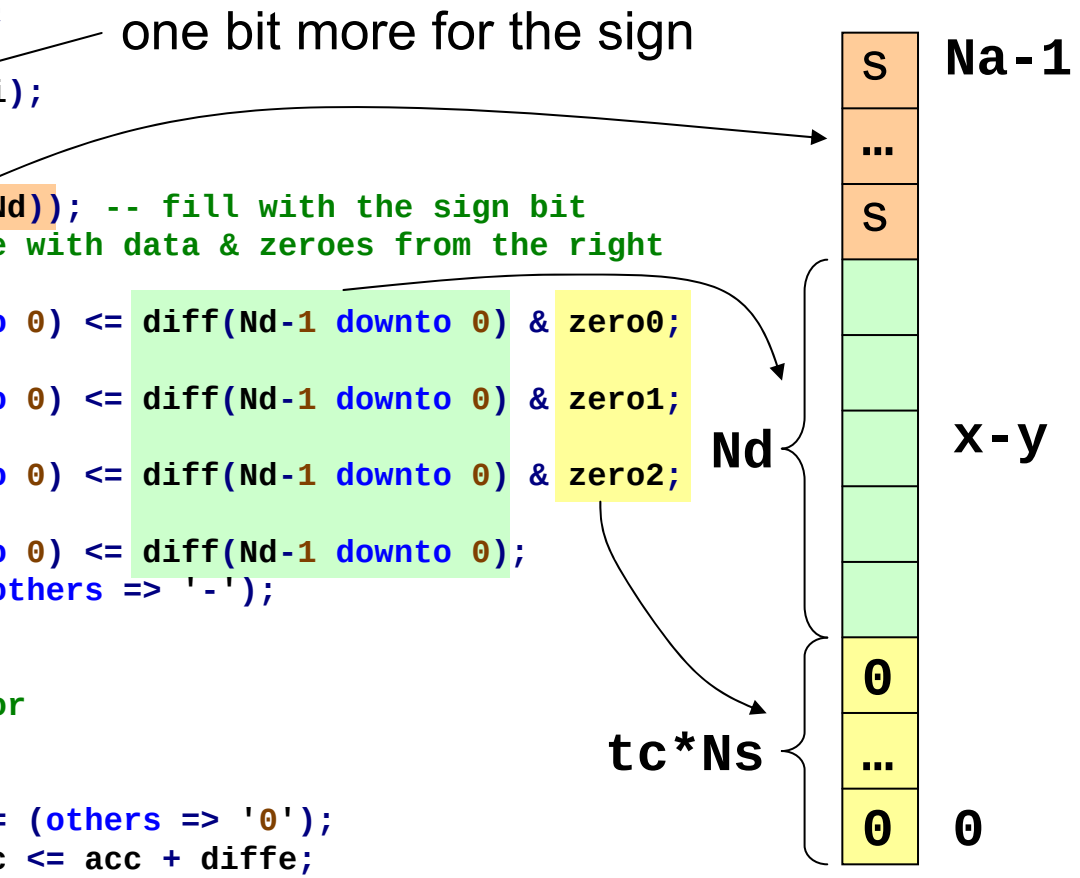
LI – **vhdl** code (1)



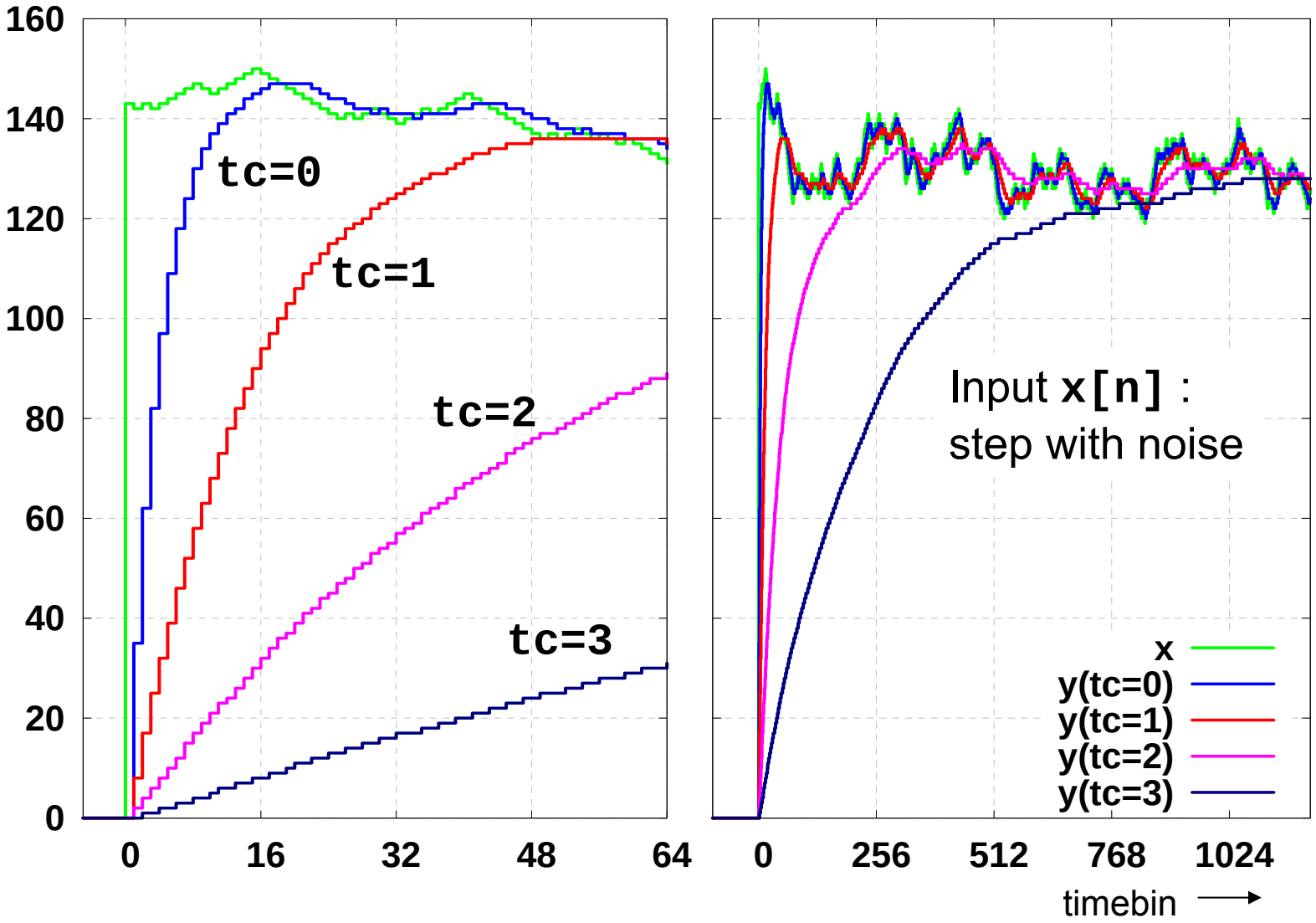

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;

entity iir_relax is
generic(Nd : Integer := 8;  -- data width
        Ns : Integer := 2;  -- shift step/tc, k=Na-Nd-Ns*tc
        Na : Integer :=18); -- acc width
port (                      -- note: Nd+3*Ns<Na!
    clk : in  std_logic;
    clr : in  std_logic;
    ena : in  std_logic;
    x   : in  std_logic_vector(Nd-1 downto 0);   -- input data
    tc  : in  std_logic_vector(   1 downto 0);   -- time constant
    y   : out std_logic_vector(Nd-1 downto 0));  -- output
end iir_relax;

architecture a of iir_relax is

signal acc     : std_logic_vector(Na-1 downto 0);
signal diff    : std_logic_vector(Nd   downto 0);
signal diffe   : std_logic_vector(Na-1 downto 0);
signal y_i     : std_logic_vector(Nd-1 downto 0);  -- output
constant zero0 : std_logic_vector(3*Ns-1 downto 0) := (others => '0');
constant zero1 : std_logic_vector(2*Ns-1 downto 0) := (others => '0');
constant zero2 : std_logic_vector(  Ns-1 downto 0) := (others => '0');
```

LI – **vhdl** code (2)

```
y_i <= acc(Na-1 downto Na-Nd);
y   <= y_i;
diff <= ('0' & x) - ('0' & y_i);
process(diff, tc)
begin
    diffe <= (others => diff(Nd)); -- fill with the sign bit
    case tc is    -- overwrite with data & zeroes from the right
    when "00" =>
        diffe(Nd+3*Ns-1 downto 0) <= diff(Nd-1 downto 0) & zero0;
    when "01" =>
        diffe(Nd+2*Ns-1 downto 0) <= diff(Nd-1 downto 0) & zero1;
    when "10" =>
        diffe(Nd+1*Ns-1 downto 0) <= diff(Nd-1 downto 0) & zero2;
    when "11" =>
        diffe(Nd     -1 downto 0) <= diff(Nd-1 downto 0);
    when others => diffe <= (others => '-');
    end case;
end process;
process(clk) -- the accumulator
begin
    if rising_edge(clk) then
        if clr='1' then acc <= (others => '0');
        elsif ena='1' then acc <= acc + diffe;
        end if;
    end if;
end process;
```



LI – simulation



Smoothing the DDS signal

See "DDS - Generating a sine wave(3)"
(use a phase accumulator to modify the frequency)

direct sin-wave from **`sin_lut`**

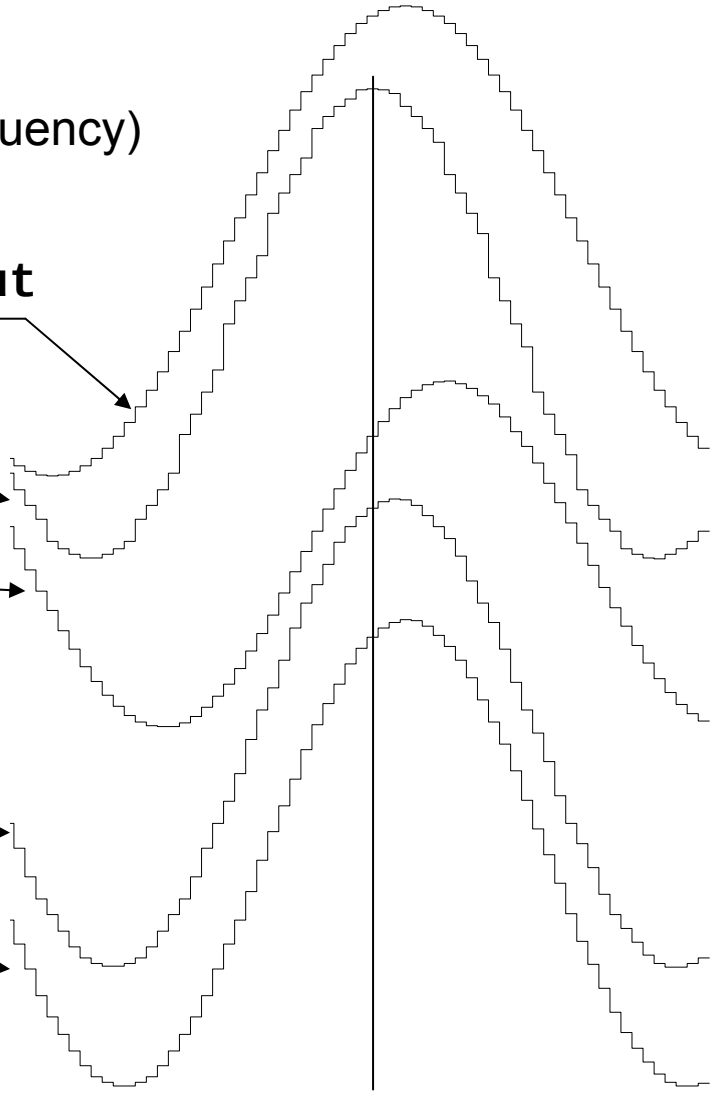
change the frequency by factor 5/4

filtered by
`LI-IIR`

filtered by 3 tap **`FIR`**
$(y_0+2*y_1+y_2)/4$

filtered by 5 tap **`FIR`**
$(y_0+4*y_1+6*y_2+4*y_3+y_4)/4$

! phase shift and amplitude attenuation!

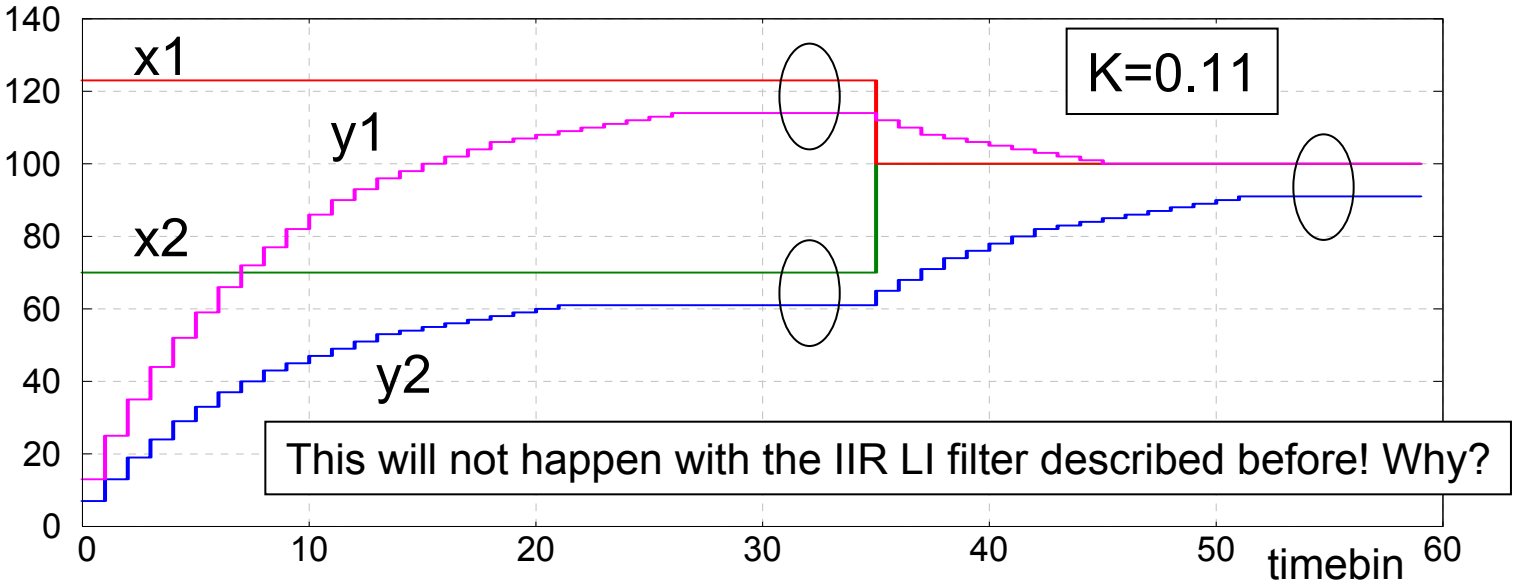


IIR – rounding problems(1)

Let's take again an IIR relaxation filter:

$$y[n+1] = k \cdot x[n] + (1-k) \cdot y[n]$$

If the input is constant, we expect that the output reaches the same value (after some time) – but this is true only if we use floating point arithmetic or enough additional bits for the calculation!



IIR – rounding problems(2)

Lets investigate when the next y is equal to the previous in

$$y[n+1] = k \cdot x[n] + (1-k) \cdot y[n]$$

For k = 0.11, $y[n+1] = y[n]$ for all y = 91..100. This means all 10 possible values are "stable"! If our input had some noise in the history and the present input is 100, the output will be between 91 and 100! This is not at all good!
If we multiply both sides by 2^M to have more precision in the calculations, then the "stable" range will be smaller:

M	range
0	91..100
1	96..100
2	98..100
3	99..100
4	100..100

Note that rounding errors exist in both FIR and IIR filters. Such strange behaviour is possible **only in case of IIR filters**.