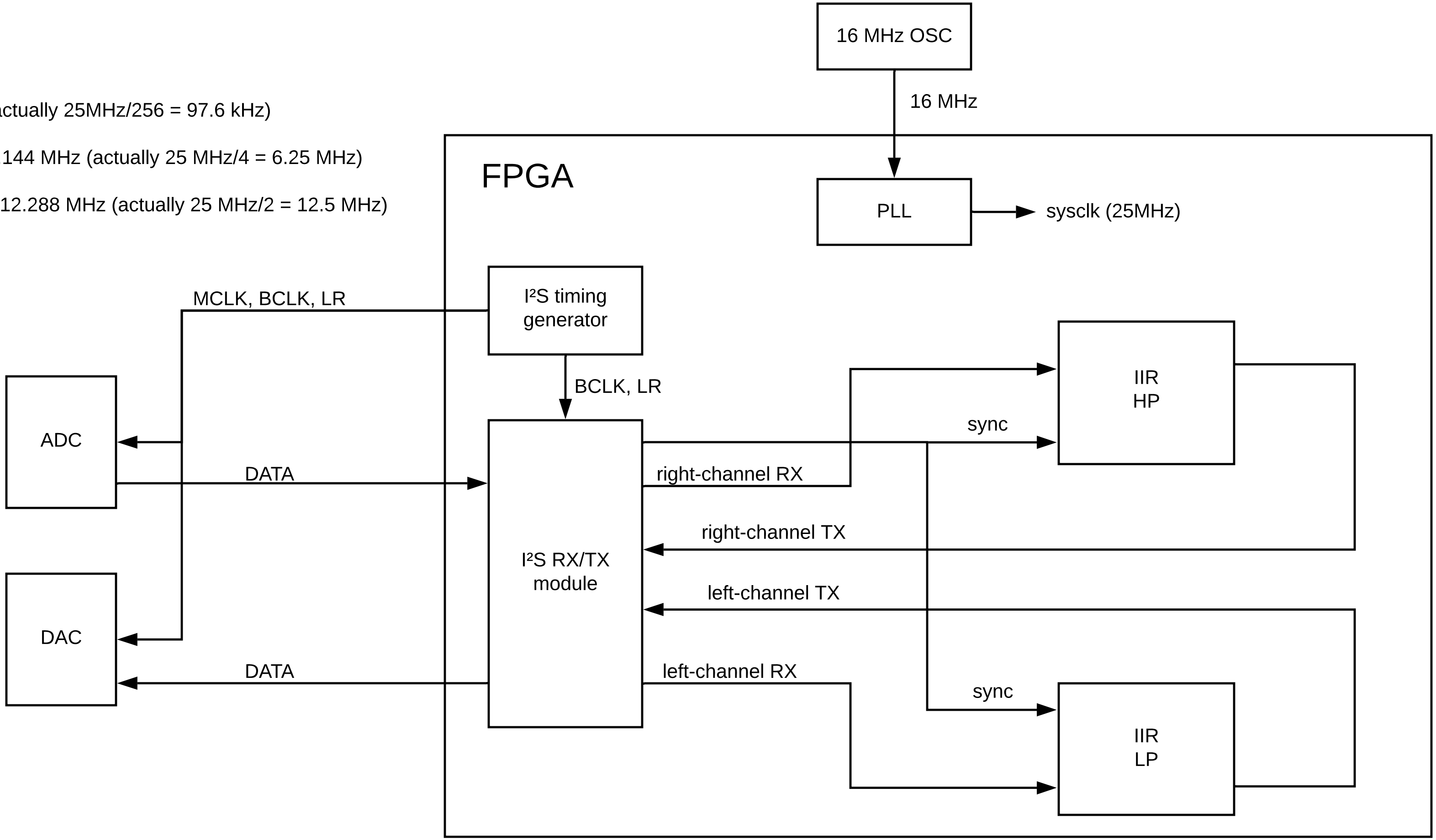
LR (fs) = 96 kHz (actually 25MHz/256 = 97.6 kHz)

BCLK = 64x fs = 6.144 MHz (actually 25 MHz/4 = 6.25 MHz)

MCLK = 128x fs = 12.288 MHz (actually 25 MHz/2 = 12.5 MHz)



Floating-point and fixed-point

> Audio-Samples are represented in 32-bit integers coming from I²S module
> Filter coefficients for IIR / FIR filters typically represented in fractional numbers

Example for DSPs / Microcontrollers (with floating-point support)

Sample: 18745
Coefficient: 0.007938475

```
int result = (int) ((float) sample * 0.007938475f)
//result would be = 149
```

FPGAs

> No floating-point support by default. However, a dedicated floating-point unit can be implemented (but not efficient in terms of FPGA usage)
> By default support for signed and unsigned integers for multiplications, summings and substractions
> Bit-shifting very easy to implement

Sample: 18745
Coefficient: 0.007938475

Step 1 (preparation)

What fixed-point format is needed?
> IIR parameters can vary between typically -2.0 and + 2.0
> Using Q2.30 format (for 32-bit resolution)
>can represent numbers between -2.0 and +1.9999999...
> Multiply all coefficients with 2^30 -> e.g. 0.007938475*(2^30) = 8523873

Step 2 (FPGA implementation)

Multiply incoming sample and fixed-point coefficient
> 18745 * 8523873 = 159779999385
> Caution: a 32bit by 32bit multiplication will result in a 64 bit output vector on a FPGA

> Apply shift_right by 30 bits on the output result. <u>This is effectively a division by 2^30</u>.
159779999385 >> 30 = 149

> Reduce output result from 64-bit again to 32-bit

Always pipeline complex operations on a FPGA

A 32 * 32 bit multiplier as needed by our IIR filter is extensively consuming logic on a FPGA.

On our TinyFPGA-BX board (iCE40LP8K), our complete design with two IIR filters (i.e. two 32*32 bit multipliers) is already using 5160 LUTs (which is already 67% of what is available).
> LUT/PLB usage mainly driven by the multipliers.

Don’t do this

```
-- add gain factors to numerator of biquad (feed forward path)
pgZFF_X0_quad <= std_logic_vector( signed(Coef_b0) * signed(ZFF_X0)) when mul_coefs = '1';
pgZFF_X1_quad <= std_logic_vector( signed(Coef_b1) * signed(ZFF_X1)) when mul_coefs = '1';
pgZFF_X2_quad <= std_logic_vector( signed(Coef_b2) * signed(ZFF_X2)) when mul_coefs = '1';

-- add gain factors to denominator of biquad (feed back path)
pgZFF_Y1_quad <= std_logic_vector( signed(Coef_a1) * signed(ZFF_Y1)) when mul_coefs = '1';
pgZFF_Y2_quad <= std_logic_vector( signed(Coef_a2) * signed(ZFF_Y2)) when mul_coefs = '1';
```

This will synthesize 5 multipliers parallel into your FPGA - using a lot of logic.
Advantage -> all calculation possible within 1 clock-cycle

Do this

Separate your multiplier in an own process:

```
-- multiplier
process(mult_in_a, mult_in_b)
begin
mult_out <= mult_in_a * mult_in_b;
end process;
```

Always run only one multiplication per clock cycle - controlled by state-machine

```
elsif (state = 1) then
    --save result of (samplein*a0) to temp and apply right-shift of 30
    --and load multiplier with in_z1 and a1
    temp <= resize(shift_right(mult_out,30),40);
    mult_in_a <= in_z1;
    mult_in_b <= to_signed(a1,32);
    state <= 2;

 elsif (state = 2) then
    --save and sum up result of (in_z1*a1) to temp and apply right-shift of 30
    --and load multiplier with in_z2 and a2
    temp <= temp + resize(shift_right(mult_out,30),40);
    mult_in_a <= in_z2;
    mult_in_b <= to_signed(a2,32);
    state <= 3;
```

Better usage of your single-multiplier and therefore space saving.
Drawback: 8 clock-cycles needed for one IIR calculation - but absolutely uncritical