

Application note: IIR filter

Goal

This application note assumes that the user has selected an infinite impulse response (IIR) filter and designed it using a tool such as Matlab, and now needs to efficiently implement it in hardware.

Alternate IIR implementations can vary the form (e.g. direct, standard, ladder,...), math (fixed point, floating point), and quantization (number of bits). These alternate implementations will have different throughput, resource usage and quantization sensitivity. Ideally a designer should simulate a few different architectures.

This application note looks at how Mobius can be used to investigate throughput, resources and quantization of a direct form IIR filter implemented with fixed point and floating point math with varying quantization.

IIR algorithm

Many common signal processing filters such as Elliptic, Butterworth, Chebychev, Bessel filters are implemented as IIR filters¹. In addition, the discrete time proportional-integral-derivative (PID) controller and lead-lag controller can be expressed² as a 2nd order IIR filter. Numerically an IIR consists of an impulse transfer function $H(z)$ with q poles and p zeros.

$$H(z) = \frac{\sum_{i=0}^P b_i z^{-i}}{1 - \sum_{k=1}^Q a_k z^{-k}}$$

This can also be expressed as a difference equation

$$y(n) = \sum_{i=0}^P b_i x(n-i) + \sum_{k=1}^Q a_k y(n-k)$$

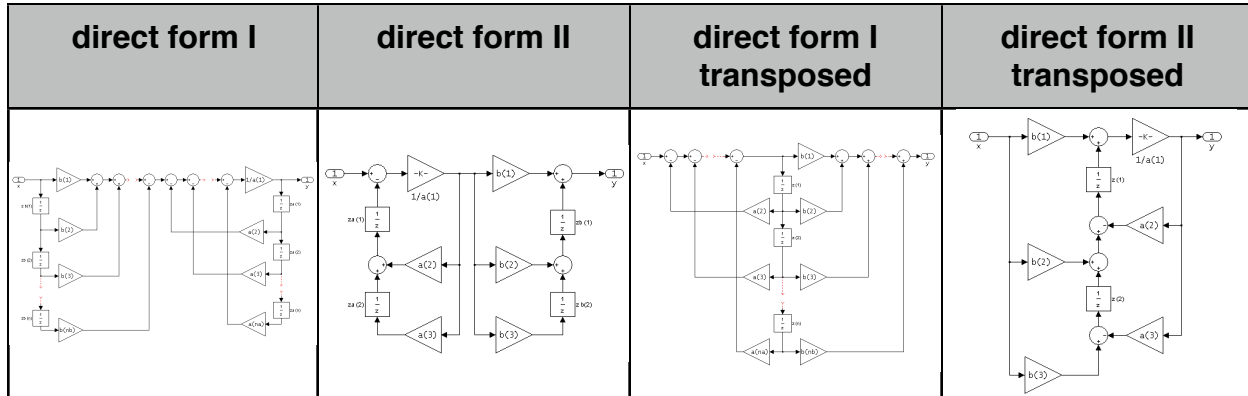
There are several possible implementations³ including direct form I and II, cascade, parallel and lattice canonical forms. The direct form I is often used by fixed point IIR filters since a larger single adder can be used to prevent saturation. Similarly direct form II is often used by floating point IIR filters, since this uses fewer states and the adders

¹ Matlab Filter toolbox, online at http://www.mathworks.com/access/helpdesk_r13/help/pdf_doc/filterdesign/filterdesign.pdf

² Franklin, *Digital Control of Dynamic Systems*, isbn 0-201-11938-2, 1992, pg. 222+

³ Matlab Signal processing toolbox, online at <http://www.mathworks.com/access/helpdesk/help/toolbox/signal/>

are not as sensitive to saturation. However the cascade canonical form has the lowest⁴ quantization sensitivity, but at the cost of additional resources. For example if $p=q$ and p is even, then the direct form I and II implementations only require $2p+1$ constant multipliers while the cascade requires $5p/2$ constant multipliers.



Because of the many tradeoffs, it is clear that numerical experiments may be needed to determine a suitable implementation.

Mobius implementation

For this application note, we select a 3rd order IIR filter where $b_0=0$, $b_1=0$, $b_2=1$, $b_3=0.5$, $a_1=-1$, $a_2=0.01$, and $a_3=0.12$. In Mobius the IIR algorithm can be codified as a difference equation (from the direct form II) as the following:

```

procedure test();

type t = sfixed(6,8); (* type t = float(8,24); *)

var c1,c2:chan of t;
var y;t;

procedure iir(in cu:chan of t; out cy:chan of t);
const b3: t = 0.5;
const a2: t = 0.01;
const a3: t = 0.12;
var u: t;
var x1,x2,x3:t;
seq
  x1:=0.0; x2:=0.0; x3:=0.0;
  while true do
    seq
      cu ? u; (* read input *)
      x1,x2,x3 := x2,x3,u-(t(a3*x1)+t(a2*x2)-x3); (* update states *)
      cy ! t(b3*x1) + x2 (* write output *)
    end
  end;

par (* testbench *)
  iir(c1,c2);
  while true do seq c1 ! 1.0; c2 ? y; write(" y="..y) end
end;

```

⁴ Oppenheim, Discrete Time Signal Processing, isbn0-13-216292-X, 1989 pg 334+

In the Mobius source, note that the IIR filter is contained in the procedure `iir()` which communicates using channels. A channel uses message passing, and blocks until both the sender and receiver are ready for communication.

Inside `iir()` a while true loop continuously reads new reference value `u`, updates the states `x1,x2,x3`, calculates and writes the output. Note how all the states are simultaneously updated. The testbench simply sends a constant reference value to the filter input and reads the filter output and writes that resulting step value to `stdout`. By separating the IIR filter from the testbench, this allows the generated `iir()` module to be synthesized. The testbench is not synthesizable since it uses print statements to `stdout`.

Parallel or sequential architecture?

As implemented above, the `iir()` procedure computes all expressions in a maximally parallel manner and does not utilize any resource sharing. The user can also create alternate architectures that (a) use pipelined operators and statements for higher speed; (b) use resource sharing for smaller resources and slower performance. The current implementation is a compromise between these two extremes.

Fixed point or floating point?

Both fixed point and floating point math are fully integrated into the Mobius language, making things easy for the designer to mix and match various sized fixed point and floating point operators. This is exploited in the above code by defining a user defined type "t" with a particular parameterized fixed point or floating point size. By changing this single definition, the compiler will automatically use the selected quantization of operands and math operators in the entire application.

For instance the signed fixed point type definition `sfixed(6,8)` uses 14 bits with 6 bits for the whole number and 8 bits for the fraction, while the floating point type definition `float(6,8)` uses 15 bits with 1 sign bit, 6 exponent bits and 8 mantissa bits.

	sfixed(6,8)	float(6,8)
structure	($w_5, \dots, w_0, \bullet, f_7, \dots, f_0$) in 2's complement representation	($s, e_5, \dots, e_0, \bullet, m_7, \dots, m_0$)
value	$w_5 2^5 + w_4 2^4 + w_3 2^3 + w_2 2^2 + w_1 2^1 + w_0 2^0 + f_7 2^{-1} + f_6 2^{-2} + f_5 2^{-3} + f_4 2^{-4} + f_3 2^{-5} + f_2 2^{-6} + f_1 2^{-7} + f_0 2^{-8}$	$(-1)^s \cdot (2^{\text{bias}+e}) \cdot (m_7 2^0 + m_6 2^{-1} + m_5 2^{-2} + m_4 2^{-3} + m_3 2^{-4} + m_2 2^{-5} + m_1 2^{-6} + m_0 2^{-7})$ where $\text{bias} = 2^{\text{ebits}} - 1$
example	14'b00000011000000 = 0.75	15'b001111011000000 = 0.75

The implementation of the fixed point and floating point operators can be inspected since they are delivered as Mobius source in the library directory. The fixed point add, sub and mul operators are 0-cycle combinatorial functions. The floating point add and sub operators take 4 cycles, the mul operators takes 2 cycles, and the div operator is iterative dependent on the size of the operands.

Note that expressions calculated using fixed point and floating point of varying quantization sizes will obviously have different results. The stability and frequency response of the quantized filter can be determined using the Matlab filter toolbox.

Inspecting the generated Verilog or VHDL shows how the IIR coefficients are quantized. For example the coefficient a2 was supposed to be 0.01, but with a sfix(6,8) quantization we see that it is stored as a signed fixed point with bits 14'b00000000000010 corresponding to a real value of 0.0078175, which is a relative error of $(0.01-0.0078175)/0.01 = 22\%$. If the poles are close to the unit circle, then clearly more fractional bits may be needed to avoid poles moving outside the unit circle. In this particular case stability was not effected by quantization.

	sfix(6,8)	sfix(8,24)	float(6,8)	float(8,24)
b3=0.5	14'b00000010000000 = 0.5	32'b00000000010000000000000000000000 = 0.5	15'b001111010000000 = 0.5	33'b00111110100000000000000000000000 = 0.5
a2=0.01	14'b00000000000010 = 0.0078125	32'b00000000000000101000111101011100 = 0.00999999046325683594	15'b001100010100011 = 0.00994873046875	33'b001111000101000111101011100001010 = 0.00999999977648
a3=0.12	14'b00000000011110 = 0.1171875	32'b000000000000111101011100001010001 = 0.119999945163726807	15'b001101111110101 = 0.11962890625	33'b001111011111101011100001010001111 = 0.119999997318

Synthesis

The Mobius implementation of the IIR was synthesized for a variety of math types and quantizations shown below. The Xilinx ISE v7.1sp4 was utilized targeting a Xilinx Virtex2Pro FPGA.

The target FPGA is a Virtex2Pro (xc2vp2-6ff672) which has 1408 slices, 2816 FFs, 2816 LUTs, 204 IOBs and 12 mult18x18s. The Xilinx ISE synthesizer will map the VHDL/Verilog multipliers onto hardware multipliers if they are available. For targets without hardware multipliers, the synthesizer will generate appropriate (but slower) combinatorial logic.

Xilinx ISE v7.1sp4 xc2vp2-6ff672	sfix(6,8)	sfix(8,24)	float(6,8)	float(8,24)
resources	64 slices 63 FFs 105 LUTs 38 IOBs 3 mult18x18s	285 slices 135 FFs 513 LUTs 74 IOBs 12 mult18x18s	826 slices 625 FFs 1361 LUTs 40 IOBs 3 mult18x18s	2770 slices 1345 FFs 4958 LUTs 76 IOBs 12 mult18x18s
clock	89 MHz	56 MHz	103 MHz	61 MHz
cycles	3	3	27	27

The number of cycles needed for the IIR filter to read the input, update the states and write the output is shown as "cycles". The resources are the results obtained from synthesis (without running place & route).

The use of many parallel combinatorial operators results in a slow clock in the designs with larger datawidths, so a redesign using more states (or pipelined) could be considerably faster.

The smaller three designs fit in the target, but the largest using single precision float(8,24) requires too many slices, indicating that resource sharing is needed for larger floating point implementations.

The resources and throughput of the generated IIR filters is competitive with hand-designs. Using Mobius dramatically simplifies the generation of hardware, since the user does not need a third party synthesizable math library or math expertise.

Summary

Mobius has been used to generate various fixed point and floating point implementations of an IIR filter, allowing throughput, resources and quantization effects to be quantitatively investigated.

Implementations with higher performance or lower resources can be created using pipelined statements or shared resources.