

Rapid Development of Control Systems with Floating Point

Generate high-quality synthesizable HDL with the Mobius compiler.

```

procedure ir(m: real; a: real; b: real; c: real; d: real; e: real; f: real; g: real; h: real; i: real; j: real; k: real; l: real; m: real; n: real; o: real; p: real; q: real; r: real; s: real; t: real; u: real; v: real; w: real; x: real; y: real; z: real; aa: real; ab: real; ac: real; ad: real; ae: real; af: real; ag: real; ah: real; ai: real; aj: real; ak: real; al: real; am: real; an: real; ao: real; ap: real; aq: real; ar: real; as: real; at: real; au: real; av: real; aw: real; ax: real; ay: real; az: real; ba: real; bb: real; bc: real; bd: real; be: real; bf: real; bg: real; bh: real; bi: real; bj: real; bk: real; bl: real; bm: real; bn: real; bo: real; bp: real; bq: real; br: real; bs: real; bt: real; bu: real; bv: real; bw: real; bx: real; by: real; bz: real; ca: real; cb: real; cc: real; cd: real; ce: real; cf: real; cg: real; ch: real; ci: real; cj: real; ck: real; cl: real; cm: real; cn: real; co: real; cp: real; cq: real; cr: real; cs: real; ct: real; cu: real; cv: real; cw: real; cx: real; cy: real; cz: real; da: real; db: real; dc: real; dd: real; de: real; df: real; dg: real; dh: real; di: real; dj: real; dk: real; dl: real; dm: real; dn: real; do: real; dp: real; dq: real; dr: real; ds: real; dt: real; du: real; dv: real; dw: real; dx: real; dy: real; dz: real; ea: real; eb: real; ec: real; ed: real; ee: real; ef: real; eg: real; eh: real; ei: real; ej: real; ek: real; el: real; em: real; en: real; eo: real; ep: real; eq: real; er: real; es: real; et: real; eu: real; ev: real; ew: real; ex: real; ey: real; ez: real; fa: real; fb: real; fc: real; fd: real; fe: real; ff: real; fg: real; fh: real; fi: real; fj: real; fk: real; fl: real; fm: real; fn: real; fo: real; fp: real; fq: real; fr: real; fs: real; ft: real; fu: real; fv: real; fw: real; fx: real; fy: real; fz: real; ga: real; gb: real; gc: real; gd: real; ge: real; gf: real; gg: real; gh: real; gi: real; gj: real; gk: real; gl: real; gm: real; gn: real; go: real; gp: real; gq: real; gr: real; gs: real; gt: real; gu: real; gv: real; gw: real; gx: real; gy: real; gz: real; ha: real; hb: real; hc: real; hd: real; he: real; hf: real; hg: real; hh: real; hi: real; hj: real; hk: real; hl: real; hm: real; hn: real; ho: real; hp: real; hq: real; hr: real; hs: real; ht: real; hu: real; hv: real; hw: real; hx: real; hy: real; hz: real; ia: real; ib: real; ic: real; id: real; ie: real; if: real; ig: real; ih: real; ii: real; ij: real; ik: real; il: real; im: real; in: real; io: real; ip: real; iq: real; ir: real; is: real; it: real; iu: real; iv: real; iw: real; ix: real; iy: real; iz: real; ja: real; jb: real; jc: real; jd: real; je: real; jf: real; jg: real; jh: real; ji: real; jj: real; jk: real; jl: real; jm: real; jn: real; jo: real; jp: real; jq: real; jr: real; js: real; jt: real; ju: real; jv: real; jw: real; jx: real; jy: real; jz: real; ka: real; kb: real; kc: real; kd: real; ke: real; kf: real; kg: real; kh: real; ki: real; kj: real; kk: real; kl: real; km: real; kn: real; ko: real; kp: real; kq: real; kr: real; ks: real; kt: real; ku: real; kv: real; kw: real; kx: real; ky: real; kz: real; la: real; lb: real; lc: real; ld: real; le: real; lf: real; lg: real; lh: real; li: real; lj: real; lk: real; ll: real; lm: real; ln: real; lo: real; lp: real; lq: real; lr: real; ls: real; lt: real; lu: real; lv: real; lw: real; lx: real; ly: real; lz: real; ma: real; mb: real; mc: real; md: real; me: real; mf: real; mg: real; mh: real; mi: real; mj: real; mk: real; ml: real; mm: real; mn: real; mo: real; mp: real; mq: real; mr: real; ms: real; mt: real; mu: real; mv: real; mw: real; mx: real; my: real; mz: real; na: real; nb: real; nc: real; nd: real; ne: real; nf: real; ng: real; nh: real; ni: real; nj: real; nk: real; nl: real; nm: real; nn: real; no: real; np: real; nq: real; nr: real; ns: real; nt: real; nu: real; nv: real; nw: real; nx: real; ny: real; nz: real; oa: real; ob: real; oc: real; od: real; oe: real; of: real; og: real; oh: real; oi: real; oj: real; ok: real; ol: real; om: real; on: real; oo: real; op: real; oq: real; or: real; os: real; ot: real; ou: real; ov: real; ow: real; ox: real; oy: real; oz: real; pa: real; pb: real; pc: real; pd: real; pe: real; pf: real; pg: real; ph: real; pi: real; pj: real; pk: real; pl: real; pm: real; pn: real; po: real; pp: real; pq: real; pr: real; ps: real; pt: real; pu: real; pv: real; pw: real; px: real; py: real; pz: real; qa: real; qb: real; qc: real; qd: real; qe: real; qf: real; qg: real; qh: real; qi: real; qj: real; qk: real; ql: real; qm: real; qn: real; qo: real; qp: real; qq: real; qr: real; qs: real; qt: real; qu: real; qv: real; qw: real; qx: real; qy: real; qz: real; ra: real; rb: real; rc: real; rd: real; re: real; rf: real; rg: real; rh: real; ri: real; rj: real; rk: real; rl: real; rm: real; rn: real; ro: real; rp: real; rq: real; rr: real; rs: real; rt: real; ru: real; rv: real; rw: real; rx: real; ry: real; rz: real; sa: real; sb: real; sc: real; sd: real; se: real; sf: real; sg: real; sh: real; si: real; sj: real; sk: real; sl: real; sm: real; sn: real; so: real; sp: real; sq: real; sr: real; ss: real; st: real; su: real; sv: real; sw: real; sx: real; sy: real; sz: real; ta: real; tb: real; tc: real; td: real; te: real; tf: real; tg: real; th: real; ti: real; tj: real; tk: real; tl: real; tm: real; tn: real; to: real; tp: real; tq: real; tr: real; ts: real; tt: real; tu: real; tv: real; tw: real; tx: real; ty: real; tz: real; ua: real; ub: real; uc: real; ud: real; ue: real; uf: real; ug: real; uh: real; ui: real; uj: real; uk: real; ul: real; um: real; un: real; uo: real; up: real; uq: real; ur: real; us: real; ut: real; uu: real; uv: real; uw: real; ux: real; uy: real; uz: real; va: real; vb: real; vc: real; vd: real; ve: real; vf: real; vg: real; vh: real; vi: real; vj: real; vk: real; vl: real; vm: real; vn: real; vo: real; vp: real; vq: real; vr: real; vs: real; vt: real; vu: real; vv: real; vw: real; vx: real; vy: real; vz: real; wa: real; wb: real; wc: real; wd: real; we: real; wf: real; wg: real; wh: real; wi: real; wj: real; wk: real; wl: real; wm: real; wn: real; wo: real; wp: real; wq: real; wr: real; ws: real; wt: real; wu: real; wv: real; ww: real; wx: real; wy: real; wz: real; xa: real; xb: real; xc: real; xd: real; xe: real; xf: real; xg: real; xh: real; xi: real; xj: real; xk: real; xl: real; xm: real; xn: real; xo: real; xp: real; xq: real; xr: real; xs: real; xt: real; xu: real; xv: real; xw: real; xx: real; xy: real; xz: real; ya: real; yb: real; yc: real; yd: real; ye: real; yf: real; yg: real; yh: real; yi: real; yj: real; yk: real; yl: real; ym: real; yn: real; yo: real; yp: real; yq: real; yr: real; ys: real; yt: real; yu: real; yv: real; yw: real; yx: real; yy: real; yz: real; za: real; zb: real; zc: real; zd: real; ze: real; zf: real; zg: real; zh: real; zi: real; zj: real; zk: real; zl: real; zm: real; zn: real; zo: real; zp: real; zq: real; zr: real; zs: real; zt: real; zu: real; zv: real; zw: real; zx: real; zy: real; zz: real;
const a2: t = 0.01;
const a3: t = 0.12;
const b3: t = 0.5;
var u, 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)+(a2*x2)+x3); (* calc 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 (* get step response of filter *)
end;

```

by Per Ljung
 President
 Codetronix LLC
ljung@codetronix.com

With the advent of ever-larger FPGAs, both software algorithm developers and hardware designers need to be more productive. Without an increase in abstraction level, today's complex software systems cannot be realized, both because of a lack of resources (time, money, developers) and an inability to manage the complexity. Electronic system level (ESL) design offers a solution where an abstraction level higher than HDL will result in higher productivity, higher performance, and higher quality.

In this article, I'll show you how to use the high-level multi-threaded language Mobius to quickly and efficiently implement embedded control systems on FPGAs. In particular, I'll demonstrate the implementation of a floating-point embedded control system for a Xilinx® Virtex™-II FPGA in a few lines of Mobius source code.

Mobius lets software engineers create efficient and robust hardware/software systems, while hardware engineers become much more productive.

Mobius

Mobius is a tiny high-level multi-threaded language and compiler designed for the rapid development of hardware/software embedded systems. A higher abstraction level and ease of use lets you achieve greater design productivity compared to traditional approaches. At the same time you are not compromising on quality of results, since benchmarks show that Mobius-generated circuits match the best hand designs in terms of throughput and resources for both compact and single-cycle pipelined implementations.

Developing embedded systems with Mobius is much more like software development using a high-level language than hardware development using assembly-like HDL. Mobius has a fast transaction-level simulator so that the code/test/debugging cycle is much faster than traditional HDL iterations. The Mobius compiler translates all test benches into HDL, so for a quick verification simply compare the Mobius simulation with the HDL simulation using the same test vectors. The compiler-generated HDL is assembled from a set of Lego-like primitives connected using handshaking channels. As a result of handshaking, the circuit is robust and correct by construction.

Mobius lets software engineers create efficient and robust hardware/software systems, while hardware engineers become much more productive. With less than 200 lines of Mobius source, users have implemented FIR filters, FFT transforms, JPEG encoding, and DES and AES encryption, as well as hardware/software systems with PowerPC™ and MicroBlaze™ processors.

Parallelism is the key to obtaining high performance on FPGAs. Mobius enables both compact sequential circuits, latency-intolerant inelastic pipelined circuits, and parallel latency-tolerant elastic pipelined circuits. Using the keywords “seq” and “par,” the Mobius language allows basic blocks to run in sequence or in parallel, respectively. Parallel threads communicate with message passing channels, where the keywords “?”

and “!” are used to read and write over a channel that waits until both the reader and writer are ready before proceeding. Message passing obviates the need for low-level locks, mutexes, and semaphores. The Mobius compiler also identifies common parallel programming errors such as illegal parallel read/write or write/write statements.

The Mobius compiler generates synthesizable HDL using Xilinx-recommended structures, letting the Xilinx synthesizer efficiently infer circuits. For instance, variables are mapped to flip-flops and arrays are mapped to block RAM. Because Mobius is a front end to the Xilinx design flow, Mobius supports all current Xilinx targets, including the PowerPC and DSP units found on the Virtex-II Pro, Virtex-4, and Virtex-5 device families. The generated HDL is readable and graphically documented, showing hierarchical control and dataflow relationships. Application notes show how to create hardware/software systems communicating over the on-chip peripheral and Fast Simplex Link buses.

Handshaking allows you to ignore low-level timing, as handshaking determines the control and dataflow. However, it is very easy to understand the timing model of Mobius-generated HDL. Every Mobius signal has a request, acknowledge, and data component where the req/ack bits are used for handshaking. For a scalar variable instantiated as a flip-flop, a read takes zero cycles and a scalar write takes one cycle. An assignment statement takes as many cycles as the sum of its right-hand-side (RHS) and left-hand-side (LHS) expressions. An assignment with scalar RHS and LHS expressions therefore takes one cycle to execute. Channel communications take an unknown number of cycles (it waits until both reader and write are ready). A sequential block of statements takes as many cycles as the sum of its children, and a parallel block of statements takes the maximum number of cycles of its children.

Mobius has native support for parameterized signed and unsigned integers, fixed

point and floating point. The low-latency math libraries are supplied as Mobius source code. The fixed point add, sub, and mul operators are zero-cycle combinatorial functions. The floating-point add and sub operators take four cycles, the mul operators take two cycles, and the div operator is iterative-dependent on the size of the operands.

Infinite Impulse Response Filter

As an example of an embedded control system, let's look at how you can use Mobius to quickly implement an infinite impulse response (IIR) filter. I will investigate several different architectures for throughput, resources, and quantization using parameterized fixed- and floating-point math.

The IIR is commonly found in both control systems and signal processing. A discrete time proportional-integral-derivative (PID) controller and lead-lag controller can be expressed as an IIR filter. In addition, many common signal processing filters such as Elliptic, Butterworth, Chebychev, and Bessel are implemented as IIR filters. Numerically, an IIR comprises 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:

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

The IIR has several possible implementations. The direct form I is often used by fixed-point IIR filters because a larger single adder can prevent saturation. The direct form II is often used by floating-point IIR filters because this uses fewer states and the adders are not as sensitive to saturation. The cascade canonical form has the lowest 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

```

procedure testbench();

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 a2: t = 0.01;
const a3: t = 0.12;
const b3: t = 0.5;
var u,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); (* calc 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 (* get step response of filter *)
end;

```

Figure 1 – Mobius source code and test bench for IIR filter

implementations only require $2p + 1$ constant multipliers, while the cascade requires $5p/2$ constant multipliers.

Because of the many trade-offs, it is clear that numerical experiments are necessary to determine a suitable implementation.

IIR Implementation in Mobius

Let's select a third-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 $a_3 = 0.12$. The difference equation can be easily written as the multi-threaded Mobius program shown in Figure 1.

The Mobius source defines the IIR filter as a procedure and a test bench to exercise it. Inside `iir()`, a perpetual loop continuously reads a new reference value `u`, updates the states `x1`, `x2`, `x3`, and then calculates and writes the output. The IIR uses four (fixed- or floating-point) adders and three (fixed- or floating-point) multipliers. Note how the

values of all states are simultaneously updated. The test bench simply sends a constant reference value to the filter input, reads the filter output, and writes that resulting step value to `stdout`. By separating the IIR filter from the test bench, just the IIR filter can be synthesized.

Fixed Point or Floating Point?

Floating-point math allows for large dynamic range but typically requires considerable resources. Using fixed point instead can result in a much smaller and faster design, but with trade-offs in stability, precision, and range.

Both fixed- and floating-point math are fully integrated into the Mobius language, making things easy for you to mix and match various-sized fixed- and floating-point operators. In the Mobius source, the type "t" defines a parameter-

```

y=0.000000
y=1.000000
y=2.500000
y=4.000000
y=5.492188
y=6.855469
y=8.031250
y=9.023438
y=9.832031
y=10.472656
y=10.964844
y=11.335938
y=11.609375
y=11.800781
y=11.933594
y=12.019531
y=12.078125
y=12.109375
y=12.121094
y=12.121094
y=12.117188
y=12.109375
y=12.097656
y=12.085938
y=12.074219
y=12.062500
y=12.054688
y=12.046875
y=12.042969
y=12.035156
y=12.031250
y=12.027344
y=12.027344
y=12.027344
y=12.027344
y=12.027344

```

Figure 2 – HDL simulation of step response running test bench

ized fixed- or floating-point size. By changing this single definition, the compiler will automatically use the selected parameterization of operands and math operators in the entire application. For instance, the signed fixed-point type definition `sfixed(6,8)` uses 14 bits, where 6 bits describe the whole number and 8 bits the fraction. The floating-point type definition `float(6,8)` uses 15 bits, with 1 sign bit, 6 exponent bits, and 8 mantissa bits.

Parallel or Sequential Architecture?

Using parallel execution of multiple threads, an FPGA can achieve tremendous speedups over a sequential implementation. Using a pipelined architecture requires more resources, while a sequential implementation can share resources using time-multiplexing.

The `iir()` procedure computes all expressions in a maximally parallel manner and does not utilize any resource sharing. You can also create alternate architectures that use pipelined operators and statements for higher speed, or use resource sharing for smaller resources and slower performance.

Mobius-Generated VHDL/Verilog

Using the Mobius compiler to generate HDL from the Mobius source (Figure 1), ModelSim (from Mentor Graphics) can simulate the step response (Figure 2).

I synthesized several Mobius implementations of the IIR filter for a variety of math types and parameterizations (Table 1) using Xilinx ISE™ software version 7.1i (service pack 4) targeting a Virtex-II Pro XC2VP7 FPGA. The Xilinx ISE synthesizer efficiently maps the HDL behavioral structures onto combinatorial logic and uses dedicated hardware (for example, multipliers) where appropriate. A constraint file was not used. As you can see, the fixed-point implementations are considerably smaller and faster than the floating-point implementations.

An alternate architecture using pipelining reduces the cycle time to 1 cycle for the fixed-point and 18 cycles for the floating-point

implementations, with similar resources and clock speeds.

You can also use an alternate architecture with time-multiplexed resource sharing to make the design smaller (but slower). Sharing multipliers and adders in the larger floating-point IIR design results in a design needing only 1,000 slices and 4 multipliers at 60 MHz, but the IIR now has a latency of 48 cycles. This is a resource reduction of 3x and a throughput reduction of 2x.

Conclusion

In this article, I've shown how Mobius users can achieve greater design productivity, as exemplified by the rapid development of several fixed- and floating-point IIR filters. I've also shown how you can quickly design, simulate, and synthesize several architectures using compact, pipelined, and time-multiplexed resource sharing to quantitatively investigate resources, throughput, and quantization effects.

The Mobius math libraries for fixed- and floating-point math enable you to quickly implement complex control and signal-processing algorithms. The Mobius source for the basic IIR filter is about 25 lines of code, whereas the generated HDL is about 3,000-8,000 lines, depending on whether fixed point or floating point is generated. The increase in productivity using Mobius instead of hand-coded HDL is significant.

Using Mobius allows you to rapidly develop high-quality solutions. For more information about using Mobius in your next design, visit www.codetronix.com.

	sfixed(6,8)	sfixed(8,24)	float(6,8)	float(8,24)
Resources	66 slices 66 FFs 109 LUTs 38 IOBs 3 mult18 x 18s	287 slices 135 FFs 517 LUTs 74 IOBs 12 mult18 x 18s	798 slices 625 FFs 1,329 LUTs 40 IOBs 3 mult18 x 18s	2,770 slices 1,345 FFs 4,958 LUTs 76 IOBs 12 mult18 x 18s
Clock	89 MHz	56 MHz	103 MHz	61 MHz
Cycles	2	2	26	26

Table 1 – Synthesis results of `iir()`

Supporting Your Future
HUNT ENGINEERING
USB connected Programmable FPGA systems

V-II Pro PowerPC

- Virtex-II Pro XC2VP7
- 256 Mbytes DDR Memory
- Configurable digital I/Os
- PowerPC boot FLASH
- USB 2 or Standalone

Software Defined Radio

- Virtex-II FPGA 1M gates
- 2 ch 125Msps A/D and D/A
- TI C6203 DSP
- 32Mbytes SDRAM
- Configurable Digital I/O
- USB 2 or Standalone

Imaging with Virtex-4FX

- Virtex-4 FPGA FX12
- 128Mbytes DDR Memory
- CameraLink connection
- VHDL and PowerPC Imaging Libs
- USB 2 or Standalone

Programmable hardware with cables, device drivers, loading tools, examples and Power Supply. Systems can be used connected to a PC using USB, or can function standalone (without USB) using the initialisation PROMs.

sales@hunteng.co.uk
+44 (0)1278 760188

www.hunt-rtg.com