Programming a Xilinx FPGA in "C"

Hardware designers are realizing they will need to use higher levels of abstraction to increase their productivity.

by Doug Johnson, Business Development Manager, Frontier Design, doug_johnson@frontierd.com; Marc Defossez, Field Applications Engineer, Xilinx, Inc. - BeNeLux, Belgium, marc.defossez@xilinx.com

Today, many complex communications and digital signal processing (DSP) systems are described using ANSI C or C++ with floating-point mathematics. ANSI C or C++ is the language most commonly used by system engineers because it is powerful and popular, and a variety of environments are available for code development, compilation, and debugging. In addition, the simulation speed using C or C++ can be substantially faster than an equivalent design environment in Verilog or VHDL.

Typically, DSP functions modeled in C are algorithms that perform filtering, modulation, demodulation, compression, coding, and other operations on digital signals. However, most hardware designers are using a design methodology based on VHDL or Verilog HDL. System designers typically deliver a C language specification which has been simulated extensively with internal system-level simulators or commercial products like SPW (Cadence), COSSAP (Synopsys), or HP-ADS (HP EESOF). The hardware designer must then rewrite the specification at least once by hand in VHDL or Verilog and re-simulate the behavior. This manual re-write is difficult, time-consuming, and error prone. In addition, the floating-point C specification must be transformed into a fixed-point implementation in hardware—a daunting task requiring extensive collaboration between the system and hardware engineers.

Frontier Design's C to HDL Solution

The EDA software tools from Frontier Design bridge the gap between the system engineers, working on a design at a high level of abstraction, and the hardware engineer, tasked with the implementation of the design in hardware such as an FPGA or an ASIC. Frontier Design has embodied its 15 years of experience (in transforming DSP algorithms into working silicon) into a methodology that is called "Algorithm to Register Transfer," or A|RT.

Two Frontier products incorporate the A|RT methodology: A|RT Library and A|RT Builder. A|RT Library extends ANSI C and C++ with fixed-point data-types that allow bit-accurate modeling of arithmetic operations as well as modeling the overflow and quantization effects associated with finite precision operations. A|RT Builder supports automatic conversion from ANSI C to VHDL or Verilog. The tools thus enable architectural design in C. As a practical test of the tools, Xilinx - BeNeLux employed the A|RT methodology on three different test cases, one of which is covered here.

Design Methodology Using A|RT

Xilinx - BeNeLux tested a "Coordinate Rotation on a Digital Computer," or CORDIC(1,2) algorithm supplied by Frontier Design using both A|RT Library and A|RT Builder. The algorithm is an iterative computing technique that is capable of evaluating mathematical functions like multiplication, square roots, and logarithms. The CORDIC algorithm is used extensively because its implementation in hardware utilizes shifts and adds only. A very important aspect of the test was to evaluate the tools to determine their applicability in implementing an algorithm based on C in a Xilinx FPGA, using HDL generated by the A|RT Builder product.

The design flow used for the evaluation is shown in Figure 1. A|RT Builder is the only C-to-HDL

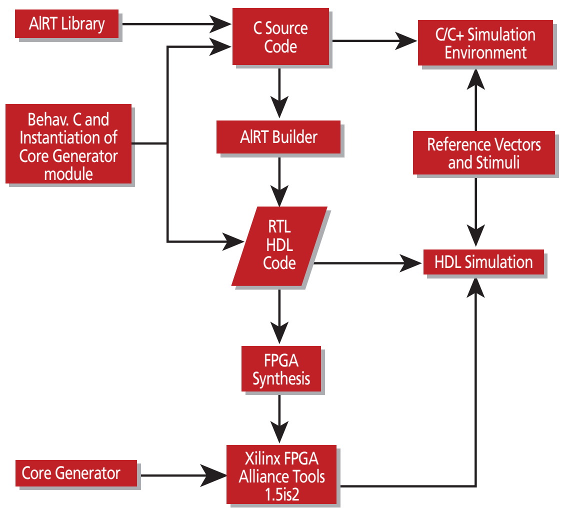
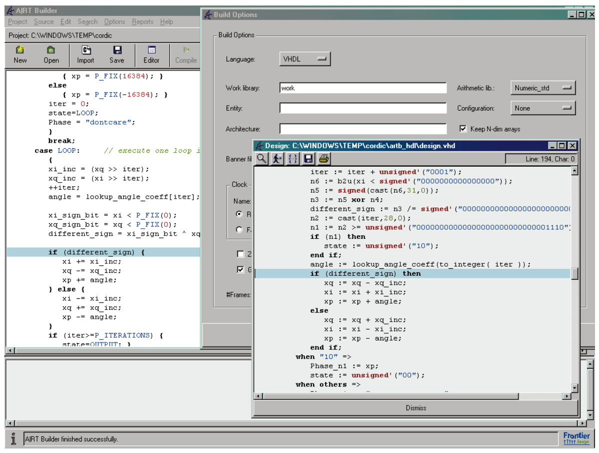

A|RT Library
C Source Code
C/C+ Simulation Environment
Behav. C and Instantiation of Core Generator module
A|RT Builder
Reference Vectors and Stimuli
RTL HDL Code
HDL Simulation
FPGA Synthesis
Core Generator
Xilinx FPGA Alliance Tools 1.5is2

Figure 1 - A|RT Design Flow

conversion tool that fully supports fixed-point algorithms. The tool employs a "What You Write Is What You Get" (WYWIWYG) paradigm. What WYWIWYG really means is that if you code in behavioral C, the output VHDL or Verilog will be behavioral. If you code in RTL C, you get RTL VHDL or Verilog. It is a supported subset of C that is utilized by A|RT Builder to generate the HDL code. Because some C constructs have no meaningful realization in hardware, there are ANSI C constructs that A|RT Builder does not support and for some constructs there are restrictions on the way they are supported. A|RT Builder automatically generates both C and HDL test benches so that the HDL and C simulations can be compared.

The A|RT software is available for both Unix and PC environments; Windows NT for the PC, HP-UX for Hewlett-Packard, and Solaris 2.6 for Sun platforms. The user interface is simple to use, and to ease the code development process, a cross-highlighting capability has been added to A|RT Builder. This feature allows you to highlight the generated HDL code by selecting the associated C code fragment.

Figure 2 shows the cross-highlighting built into the A|RT Builder user interface.


A|RT Builder
Project Source Edit Search Options Reports Help
New Open Import Save Editor Compile
Build Options
Dismiss
Frontier Design

Project: C:\WINDOWS\TEMP\cordic

```
            { xp = P_FIX(16384); }
         else
            { xp = P_FIX(-16384); }
         iter = 0;
         state=LOOP;
         Phase = "dontcare";
         }
         break;
      case LOOP:      // execute one loop i
         {
         xi_inc = (xq >> iter);
         xq_inc = (xi >> iter);
         ++iter;
         angle = lookup_angle_coeff[iter];

         xi_sign_bit = xi < P_FIX(0);
         xq_sign_bit = xq < P_FIX(0);
         different_sign = xi_sign_bit ^ xq

         if (different_sign) {
            xi += xi_inc;
            xq -= xq_inc;
            xp += angle;
         } else {
            xi -= xi_inc;
            xq += xq_inc;
            xp -= angle;
         }
         if (iter>=P_ITERATIONS) {
            state=OUTPUT; }
```

Build Options

Language: VHDL

Work library: work | Arithmetic lib.: Numeric_std

Entity: | Configuration: None

Architecture: | ☑ Keep N-dim arrays

Design: C:\WINDOWS\TEMP\cordic\artb_hdl\design.vhd — Line: 194, Char: 0

```
            iter := iter + unsigned'("0001");
            n6 := b2u(xi < signed'("0000000000000000"));
            n5 := signed(cast(n6,31,0));
            n3 := n5 xor n4;
            different_sign := n3 /= signed'("000000000000000000000000
            n2 := cast(iter,28,0);
            n1 := n2 >= unsigned'("0000000000000000000000000001110");
            if (n1) then
               state := unsigned'("10");
            end if;
            angle := lookup_angle_coeff(to_integer( iter ));
            if (different_sign) then
               xq := xq - xq_inc;
               xi := xi + xi_inc;
               xp := xp + angle;
            else
               xq := xq + xq_inc;
               xi := xi - xi_inc;
               xp := xp - angle;
            end if;
         when "10" =>
            Phase_n1 := xp;
            state := unsigned'("00");
         when others =>
```

A|RT Builder finished successfully.

Figure 2 - A|RT Builder User Interface with Cross-highlighting

The Evaluation

First, the CORDIC algorithm was designed and simulated in a C-based environment such as Visual C++ or GNU. Floating-point variables in the CORDIC algorithm were easily specified in fixed-point form by using the A|RT Library data type "Fix<w,p>", where "w" is the fixed word width and "p" is the fixed number of precision bits. For example, a variable with an 8-bit word length and 5 bits of precision is specified as follows: "Fix<8,5>".

Fixed-point values can also be specified as unsigned fixed-point, Ufix<w,p>, or as signed or unsigned integers, Int<w> and Uint<w>. In addition, several overflow and quantization characteristics can be applied to C variables to model the potentially adverse effects of finite precision arithmetic.

Once the desired response was achieved in C simulation, A|RT Builder was used to automatically convert the fixed-point CORDIC algorithm to Verilog or VHDL. For the evaluation, a number of different C coding "styles" were employed to optimize the data path length, to introduce pipelining, and to share resources and hierarchy in the generated VHDL and Verilog files.

```
Int<32> MultA;
Int<32> MultB;
MultA = InpA * InpB;
MultB = InpC * InpD;
OutP = MultA + MultB;
```

Figure 3 - No Resource Sharing In C Code

```
COMPUTE_PROC: process(
.....
 begin
 InpA_n1 := signed(InpA);
 InpB_n1 := signed(InpB);
 InpC_n1 := signed(InpC);
 InpD_n1 := signed(InpD);
 OutP_n2 := signed(OutP);
 MultA := InpA_n1 * InpB_n1;
 MultB := InpC_n1 * InpD_n1;
 OutP_n2 := MultA + MultB;
 OutP_n1 <= std_logic_vector(OutP_n2);
 end process;
.....
```

Figure 4 - No Resource Sharing in VHDL Code

The examples in figures 3 and 4 illustrate two C coding styles: Figure 3 shows a non-resource shared operation coded in C and Figure 4 shows the corresponding VHDL code fragment generated by A|RT Builder to get a two multiplier, one clock-cycle, non resource shared implementation in VHDL.

```
static Uint<1> cycle = 0u;
 static Int<32> MultA;
 Int<32> MultB;

switch (cycle) {
            case 0:
             MultA = InpA * InpB;
             break;
            case 1:
             Int<32> MultB = InpC * InpD;
             OutP = MultA + MultB;
             break;
}
++ cycle;
```

Figure 5 - Resource Sharing In C Code

```
begin
 OutP_n2 := "——————————————————————";
 — copy state to local variables
 MultA := signed(MultA_r);
 — compute new state and outputs
 InpA_n1 := signed(InpA);
 InpB_n1 := signed(InpB);
 InpC_n1 := signed(InpC);
 InpD_n1 := signed(InpD);
 OutP_n2 := signed(OutP);
 cycle := unsigned'("0");
 case cycle is
 when "0" =>
 MultA := InpA_n1 * InpB_n1;
 when "1" =>
 MultB := InpC_n1 * InpD_n1;
 OutP_n2 := MultA + MultB;
 when others =>
 assert false
 report "Invalid (possibly unknown or dontcare)
value for cycle"
 severity warning;
 end case;
 OutP_n1 <= std_logic_vector(OutP_n2);
 — copy local variables to next value for state
 MultA_nxt <= std_logic_vector(MultA);
 end process;
.....
```

Figure 6 - Resource Sharing in VHDL Code

To get a resource-shared implementation in VHDL, a different C coding style is employed. Figure 5 illustrates the C code required to get a single multiplier, two clock-cycle implementation in VHDL. Figure 6 shows the resulting VHDL code fragment with only a single multiplier in the VHDL process description. Because a switch statement has been used in the C code, A|RT Builder generated a VHDL case statement (or switch in Verilog), indicating that the two branches are exclusive, so the operators used in them can be fully shared.

The Synplify FGPA synthesis tool was employed to synthesize the HDL code and the resulting netlist was input to the Xilinx Alliance series tools. Thus, the CORDIC algorithm was implemented in an XC4000 series FPGA. The design can also be implemented in Virtex FPGAs.

The evaluation proved the validity of the design flow from an algorithm written in C to an implementation in a Xilinx FPGA. The quality of VHDL or Verilog HDL generated by A|RT Builder is comparable to that of hand-crafted code. However, the HDL code generated by A|RT Builder is only as good as the C code source. You have full control over the code that is generated and by adhering to specific coding styles, generation of very efficient behavioral and RTL code by A|RT Builder can greatly decrease the time required to map a C algorithm into HDL code.

Importing Components

One of the most powerful capabilities of the A|RT methodology uncovered during the evaluation was the ability to import pre-defined, optimized components into the C code, such as those created by the Xilinx Core Generator, those offered by the Xilinx LogiCORE™ and AllianceCORE programs, or any custom-made modules. Then you can use A|RT Builder to automatically map the instantiated components in VHDL.

For example, to import a Xilinx CORE Generator module, the C code would contain a function call referencing the name of the specific module. The Xilinx CORE Generator tool can be used to create custom parameterized, pre-optimized modules which are then imported as black boxes into A|RT Builder as explained. All of the parameters and pin names are named the same in the C function as they are on the CORE Generator instance. To simulate in C, you write a behavioral model in C for the CORE Generator module. The behavioral code, intended to be used only for C simulation and HDL code generation for the module, is inhibited by encapsulating the behavioral code between "#ifndef __SYNTHESIS__" and "endif" C statements. A|RT Builder is then employed to translate the C code into HDL. The C function then becomes a component declaration in VHDL or a module declaration in Verilog HDL.

For logic synthesis, these black-boxes are defined as "don't touch" to the synthesis tool. If you want to perform behavioral simulation of the CORE Generator modules in HDL, you have two options:

- Use behavioral HDL that A|RT Builder generates from your behavioral C code without inhibiting the HDL translation. Replace the behavioral HDL with a black-box component or module declaration before synthesis.
- Use the behavioral HDL code generated by the CORE Generator for the module, and replace it with a black-box declaration before synthesis. The Xilinx Alliance Series tools can automatically merge the black-boxes after synthesis

The benefits to the DSP algorithm designer and hardware engineer are enormous because the LogiCORE and CORE Generator blocks are now usable by both the system engineer and the hardware engineer. The system engineer can now incorporate pre-defined hardware functionality directly in C code. The hardware engineer gets a VHDL or Verilog HDL model from A|RT Builder that directly instantiates area- and performance-optimized LogiCORE and CORE Generator DSP building blocks. Also, if a DSP algorithm requires extensive memory elements such as register files or block RAM, the same code technique can be employed to target the block RAM capabilities of the Virtex devices.

Conclusion

Hardware designers realize that they will have to move to higher abstraction levels to increase their productivity; there is just no question about this. A|RT Builder is an ideal first step; it allows you to use C, while still performing a WYWIWYG translation for maximum control over the generated HDL code. The main benefits of the A|RT methodology are:

- **Automatic conversion from C/C++ to HDL**-Eliminates error-prone manual re-coding from C to VHDL/Verilog.
- **Architectural design from C/C++**-C/C++ simulation can be substantially faster than HDL; richer and more attractive development environments; more-compact code.
- **Open System**-Easily include existing, pre-optimized modules like LogiCORE, CORE Generator, and AllianceCORE components.

For further information about Frontier Design and the A|RT tools, please contact Doug Johnson at: doug_johnson@frontierd.com.

Consult the following references for more information about the CORDIC algorithm:

(1) Jack E. Volder, "The CORDIC Trigonometric Computing Technique" IRE Transactions on Electronic Computers, Vol EC-8, pp. 330-334.

(2) Vladimir Baykov, CORDIC Bibliography, http://devil.ece.utexas.edu/cordic.html