

Handing Analog to the System Software

by Ian Macbeth

Chief Technology Officer and VP Engineering,
Anadigm

To address the growing complexity and shrinking time scales in system development, system design is moving to an even higher level of abstraction in which functions are described in new system languages such as System-C. These system languages promise to supplant hardware description languages like RTL as the design-entry point while encompassing real-time operating system (RTOS) source code *and* providing a means of modeling and verifying the design.

Abstracting Analog

At the periphery of every system is the need to interface to the analog world. Until now, however, the only choices that designers have had for the analog interface were discrete components, semi-custom arrays of analog components, mixed-signal ASICs, or - for a limited number of very large markets - standard ICs.

The control systems in which analog interfaces are used tend to differ significantly from application to application, and even within a single application (such as TEC control on a DWDM laser source, or remote sensing and control in a centrally-supervised industrial processing plant.) There is significant variation from implementation to implementation.

A sensor manufacturer, for example, that can offer a diversity of analog interfaces - possesses a substantial competitive advantage. However, the ability to provide such variety in an economical manner hinges upon the manufacturer's ability to create analog designs faster and with more flexibility.

In digital systems, abstraction-to-software and design-flow automation arise from two attributes:

1. The ability to translate designs into a simple set of low-level *functions* (logic gates, latches etc.)
2. The addition of *field programmable* (FP) logic, which allows for a very elaborate re-deployment and manipulation of the system under software control.

Field programmable analog arrays (FPAA) bring both these attributes to the analog arena. Exploiting the natural precision, generic form, and switching fabric of a CMOS-based switched-capacitor (SC) architecture, the addition of a programming layer allows the selection of wide varieties of signal processing *functions* using digital configuration data.

Examples of such functions are filter stages, gain stages, summing/difference stages, voltage multiplication, phase/voltage comparators, rectifiers, oscillators, and references. All have user-programmable and reprogrammable attributes.

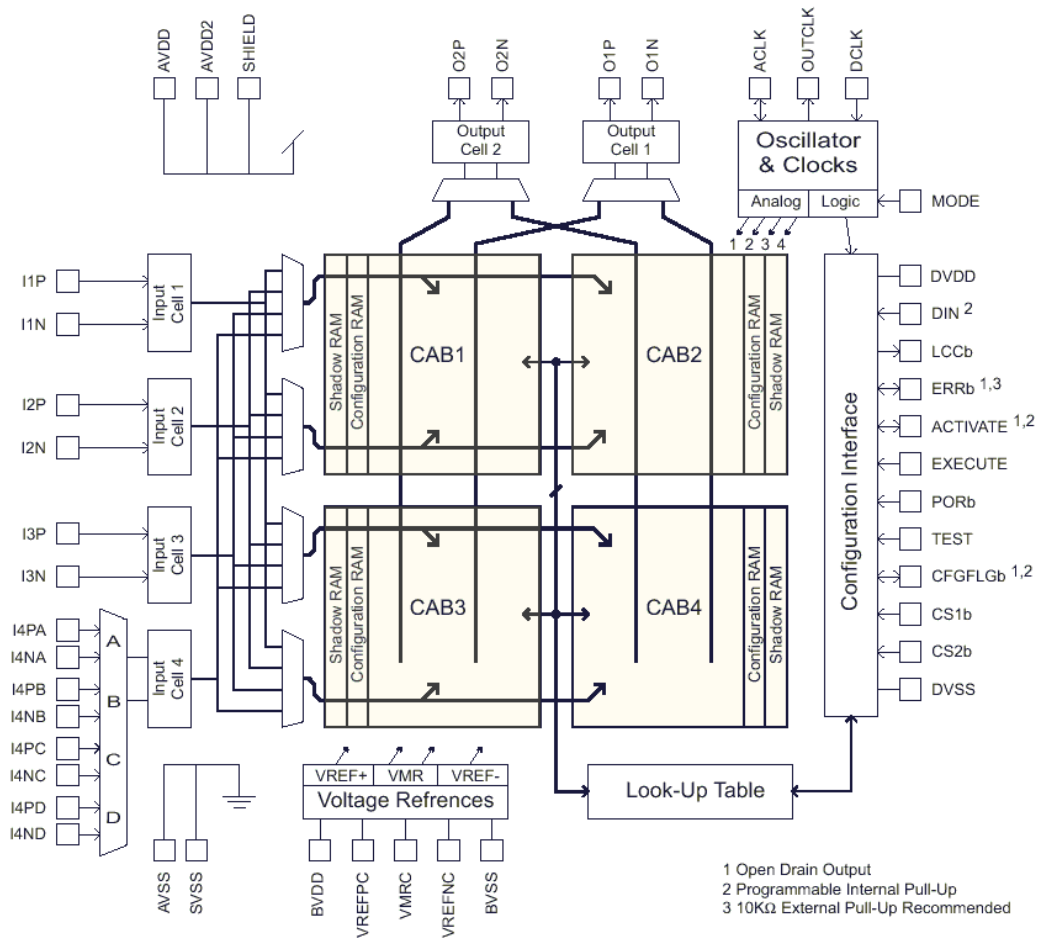


Fig. 1: The AN220E04, A Dynamically-Reconfigurable FPA

Furthermore, programming of the architecture can be highly selective. For example, in the latest-generation FPAs, each configuration byte can be reprogrammed individually without any effect on the operation of the remaining FPA.

For the first time designers have:

- The ability to represent low-level analog functions independent of the silicon and correct-by-construction: The equivalent of the logic gate.
- The ability to deliver these functions using software in the field.
- The ability to manipulate these functions under software control.

Active Rather Than Passive Configuration Data

Classically, field programmable products employ *static-reprogramming* mechanisms which incur a system interrupt while the device reconfigures and (for FPAs) can give rise to significant subsequent settling times. In contrast, *dynamic reprogramming* supports the selective, partial reprogramming of the device while it is operating, and can support continuity of operation with minimal - or no - settling effects.

For static reconfiguration, design tools classically deliver entire, passive, “dumb” configuration data sets ready for deployment in the field. Dynamic reconfiguration demands that the system software must:

- Determine the *new configuration data* segment to be applied. This is function-dependent, where structure and component values may be algorithmically-derived from high-level parameters.
- Know *where* to apply new configuration data within the memory map, which is unique to each design.
- Assemble the necessary data sequence to deliver to the configuration logic of the selected device.
- Maintain control over the timing of the update event - a critical capability for real-time systems.

All these requirements demand knowledge of the device architecture, the place-and-route decisions made by the software- and configuration-data formats. This knowledge must be conveyed from the FPAAs design tools to the system processor.

An EDA tool developed for use with dynamically reconfigurable FPAAs solves this problem by presenting device configuration in two forms: Classical configuration data and *active APIs*. The designer can choose which to use.

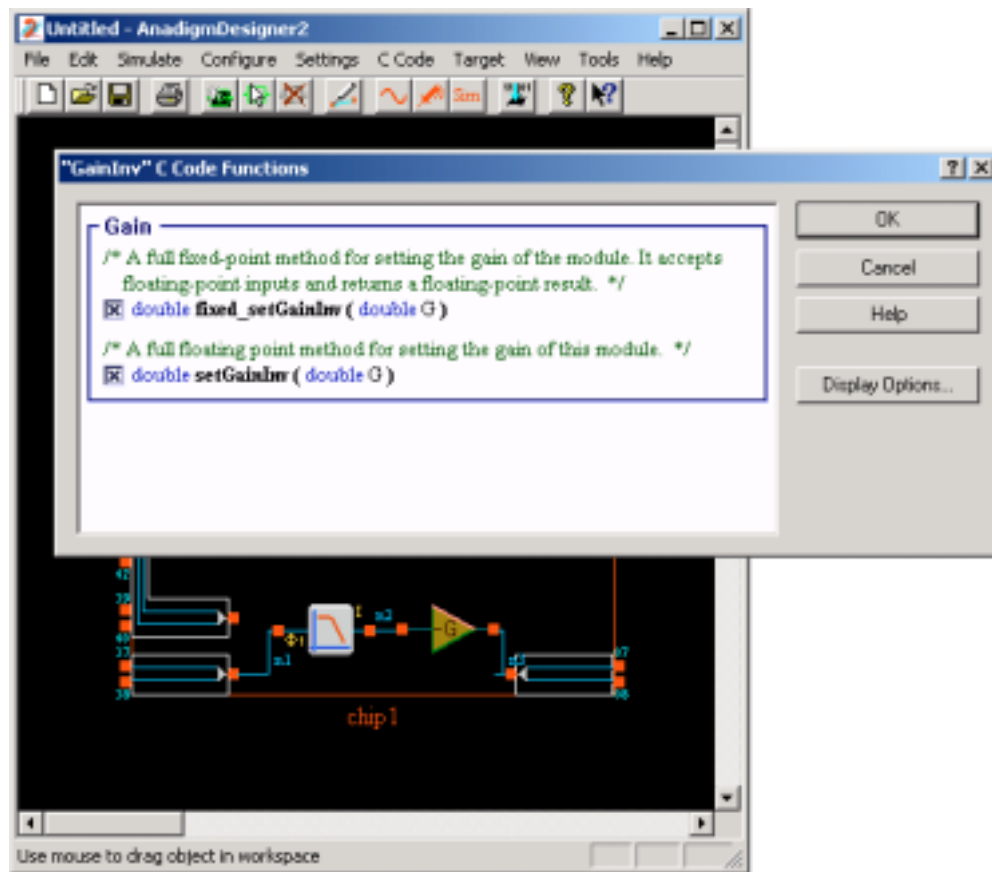


Fig. 2: Constructing An API Using The AnadigmDesigner2 EDA Tool

A simple design in the AnadigmDesigner2 EDA tool might consist of a filter and a gain stage in which the gain is to be varied under processor control (see Fig. 2.) Associated with the gain stage is at least one algorithm for calculating new circuit configurations from a high-level parameter, in this case gain, "G." This can be done for each circuit element that is to be dynamically controlled.

The design software will then export these algorithms in the form of an ANSI-C code library, which can be readily included into real-time operating system software. Additional functions are also exported. Among them is (PrimaryConfigChip), which will perform a full configuration of the selected device - a process equivalent to applying a full, passive configuration data set.

```
...
// Initialize the reconfiguration buffer for the chip
   an_initializeReconfigData(an_chip1);

// Do the primary configuration for the chip.
   PrimaryConfigChip(an_chip1);

...
if (update)
{
   // ...Change the Analog MUX if necessary...
   if ((RotateMode) || (keypressed == 5) || (keypressed == 6) || (keypressed == 2))
       an_setPadSelect(an_chip1_InputCell4, currentAnalogChannel);

   // ...Update the gain...
       an_setGainInv(an_chip1_GainInv, GainSetArray[currentAnalogChannel]);

   // ...And update the chip.
       UpdateChip(an_chip1);
}
...
```

Fig. 3: AN220E04 FPAA Dynamic Update Function Calls

Fig. 3 shows a sample application code segment. A selected FPAA is initialized, configured with a primary configuration, and then is primed conditionally using function calls to update a multiplexer (an_setPadSelect) and a gain stage (an_setGainInv.) Having delivered these updates a call to execute the update is issued (UpdateChip), applying both changes within a single system clock cycle.

The final result is a set of small run-time data segments, targeting specific areas within the FPAA devices and issued by the processor. The designer need not be concerned about memory maps, algorithm details, or bitstream segment construction. All this know-how is built into the code automatically. So API access is at the same high level as the initial drag-and-drop circuit design, and extremely fast and easy to use.

Time For Alternative Approaches

At a time when systems demand higher levels of abstraction and are increasingly self-contained the need for analog to be a part of the picture is clear.

Dynamically-reconfigurable FPAA's and their associated EDA tools support system integration for the FPAA. New tools and products such as these fill the “analog gap” in the programmable systems arena.

