FPGA'S MIDDLEWARE FOR SOFTWARE DEFINED RADIO APPLICATIONS

Xavier Revés, Vuk Marojevic, Ramon Ferrús, Antoni Gelonch

Universitat Politècnica de Catalunya - Signal Theory and Communications Department
Av. Canal Olímpic s/n, 08860 Castelldefels, Barcelona (Spain)
{xavier.reves, marojevic, ferrus, antoni}@tsc.upc.edu

ABSTRACT

The division in several layers of the implementation of systems is a solution adopted to avoid complexity, provide flexibility and improve portability and code reusability through different hardware. Middleware (intermediate layer between two other layers) implementations are based on the use of increasingly high-level languages and Application Programming Interfaces (API). The Field Programmable Gate Arrays (FPGA) world can also apply this approach to produce building blocks independent from hardware platforms and devices. This paper presents details of the implementation of a Middleware, called Platform and Hardware Abstraction Layer (P-HAL) when applied to FPGA devices. It was specially designed for radio applications and allows designing specific functions independently of the hardware context where they are applied, thus providing flexibility to the so-called Software Radios employing FPGA devices.

1. INTRODUCTION

The Flexibility parameter has more and more relevance in today's radio systems and has become part of the metrics that define the quality of a product, together with power consumption, sensibility, range, etc [1]. Flexibility is in this case applied to the management of the execution of the hard tasks that are in charge of processing the signal waveform and bit stream, the radio physical layer. It is well known that flexibility goes in the opposite sense than power consumption and size because optimised designs for a particular task always exhibit lower power demand than designs that can alternate different tasks [2]. However, the increasing need of flexibility necessarily forces the exploration of flexible solutions that can be applied to the programming of radio Layer 1 applications, and that can be extended to Layers 2 and 3 (hereafter, "application" will mean "radio application dealing with lower layers").

If the application is detached from the optimisation that represents a custom design and is moved towards the software level, those concepts that are applied in the general-purpose software design can also be employed. Among them, the most interesting here is the possibility to develop parts of the application software without knowing the hardware in advance. This requires an abstraction layer. For each layer an API provides access to the specific resources or functions that it hides. Despite the advantages that this model has demonstrated to have in GPP (General-Purpose Processing), its usage has not spanned over the radio processors technology. The reasons of this include power consumption, communications network security, manufacturer design confidentiality and tight dependency of programs on support hardware. However, these limitations can be overcome with technological evolutions and proper manufacturer agreements. As a conclusion, a software model to apply on radio technology that allows for open software development is not a utopia at present.

In this paper the translation of the software model to FPGA devices is discussed. Section 2 presents the main lines that can be applied to program Software Radio [3] applications. Section 3 provides an insight to the FPGA API part for a possible general device context. Section 4 deals with the usage of the structure described in section 3 on a concrete programmable hardware platform.

2. SOFTWARE MODEL TO BUILD RADIO APPLICATIONS

2.1. Requirements

The software model must encompass the utilisation of hardware platforms compound of different processing boards. Therefore, the model for the hardware platform is understood as an array of heterogeneous processors showing different interconnection mechanisms among them. Some of such processors will work under high-level software support, like an operating system (OS), or simply under some firmware. Because of that, our definition of hardware platform includes systems like a workstation or PC with its GPP and OS, but also devices well suited to radio applications like embedded arrays of DSPs (Digital Signal Processor) or FPGAs.

The model provides mechanisms to add as much platforms as required to the processing set, just to obtain enough computational performance for the target application. Some protocols and procedures are used to merge all the platforms into a single virtual one.

0-7803-9362-7/05/$20.00 ©2005 IEEE

In the Software Radio environment, all the processing devices that are connected together to build the hardware platform must cooperate to run in real-time the desired application. In this context, the ability to isochronously exchange data is crucial. Then, the requirements of having enough computing power and suitable communication and synchronisation mechanisms are obvious. In addition, reconfiguration demands the possibility to change, completely or partially, the running application whenever the system management requires it.

Another crucial aspect to be considered in the application area is the possibility to monitor the application behaviour or even information delivered by its processing entities. Such feature is mandatory in a system that expects to agree with the basic "rules" of Software Radio or Cognitive Radio [4]. As a consequence, a set of mechanisms to extract control data in real time from the application has to be contemplated. Not to mention that the transport of this information is bi-directional since the application management can alter the relative behaviour of some parts of the software, according to the possibilities of such software to accept modifications at runtime.

All the aforementioned abilities, to be portable and independent from platforms but also from software components, require the inclusion of rules to the design of software to be included in any application. Such rules only apply to the interface of a given software entity but not to the task that it is programmed to perform.

2.2. Architecture and elements

In the Fig. 1. a schematic representation of the software model appears. Four different layers have been identified. At the top there is the application, which is described as a set of independent objects exchanging data streams. This is the only connection between objects: a set of discrete-time ordered samples representing waveforms, bits, bytes, etc. Interfaces are viewed as mere pipes (FIFO-like) where the application can write to and read from.

The next level downwards represents the software support layer to the application, which offers an abstract environment to construct the application. The name that we assign to this layer is P-HAL (Platform and Hardware Abstraction Layer). In the third layer there are the different parts that compound the supporting software. Note that every single hardware platform (moved down to the fourth layer) contains identical functional units that build up the aforementioned abstraction layer. These entities exchange information to show up to the application as a single execution framework, independently from the actual platform composition and architecture.

The abstraction layer elements that are identified in the Fig. 1. are the following:

- BRIDGE. Move application data generated on a given platform by an application object to another platform.
- SYNC. Maintain the local time reference to a value close enough to the reference in other platforms.
- STATS. Retrieve/provide information from/to the application object for monitoring purposes.
- KERNEL. Maintain a stable union of all the platforms that are under the abstraction layer.

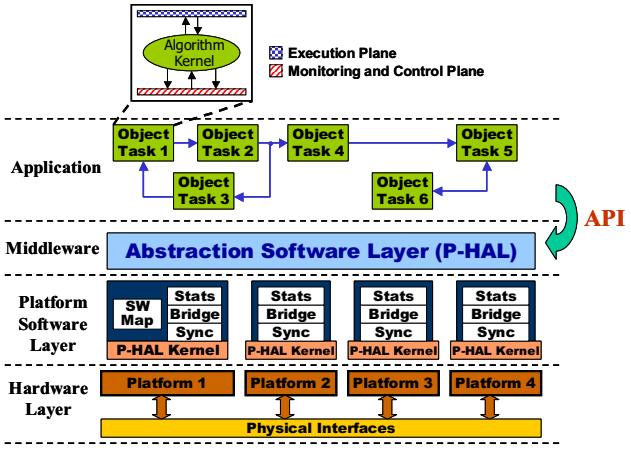

Fig. 1. The software model architecture

2.3. Programming applications

Using this simple approach it is possible to program the applications independently from hardware, following the traditional API programming approach. A set of functions is available to the object's programmer to interact with other objects and with the system. The novelty of the approach relies on using it for the focused application type and on using it as well for GPP or DSP as for FPGA. That is, the same API is implemented to program objects on these different families of processors. Although FPGAs are usually considered in a quite different manner than other processors, not doing any difference in advance can add some advantages. One result from this consideration is that the overall model fulfils the constraints that were imposed to cope with radio applications. For instance, the simplification of procedures to not excessively load the FPGA side result in low overhead in the processors side, and then, in high performance of the applications even running on the abstraction layer. To find other advantages of the equal treatment of processors and FPGA, consider, for example a set of hardware not containing FPGAs. The same algorithms can be executed on hardware sets with other processors or even accelerators (ASIC) matching the algorithm. In general, the objective is to have the application programmed using a single high-level language that can be ported to any device. Using adequate compilers and mapping and scheduling techniques the abstract application can be applied to any hardware set that provides the abstraction layer functionalities. However, having a given algorithm programmed using a single language to be

translated either to GPP, DSP or FPGA is not realistic at present. It is not because of syntax but because of structure. In the meanwhile, two separate languages are considered: C/C++ and VHDL for GPP/DSP and FPGA respectively.

3. THE FPGA API FUNCTIONS

The initial version of the API was firstly designed to run on GPP with a Linux OS. After seeing its conceptual validity for real-time applications it was moved to DSP and finally to FPGA. The translation did not offer much difficult since the basic features accessed with the API were defined so that they could be easily implemented on FPGAs. The following list summarises the actions that can be performed using the API:

- Read or Write data to one virtual interface.
- Acquire initialisation data (at start-up).
- Watch and change the value of parameters.
- Execution, sequencing and time control.

Whichever are the actions that one API allows, within a GPP the executable code organisation can be represented as well as it is shown in Fig. 2. . Different parts of code (surroundings) are accessed by means of a mechanism that allows executing those instructions that have knowledge of the platform and offer the requested service (or simply stating, a function call). From the point of view of the FPGA the procedure is exactly the same. The API is just an interface to interact with another piece of code that in this case is running concurrently with the algorithmic part of the object on the same FPGA. The number of objects within a single FPGA is not limited to one but each one must have its own API "routines". In partially reconfigurable FPGAs it is interesting to have the possibility to combine relatively small objects within a single device reusing part of the resources used by P-HAL to offer the services that are accessed through the API. The basic elements shown in Fig. 2. are for a general implementation. They are the interface traffic switch, the RAM adaptation, the control port and the execution and time control. Apart from the algorithm core, all the remaining pieces of code belong to the P-HAL library specific for a given device and hardware resources.
Within P-HAL data are conveyed using packets. The interface traffic switch is in charge of routing them in and out the object. The packet header contains information about origin and destination object and interfaces and origin time stamp, among others. This packet-oriented architecture is adequate for FPGA implementations where there are physical interfaces shared by multiple devices. Short packets stored within internal FPGA RAM are delivered to the object, which reads data as if it were reading from a FIFO. Conversely, data written by the object is treated as going to a FIFO. When enough data are available a packet is constructed and sent through the appropriate physical interface. The routing table, which contents are established by the local platform P-HAL "master" through the control port, indicates the right physical interface.

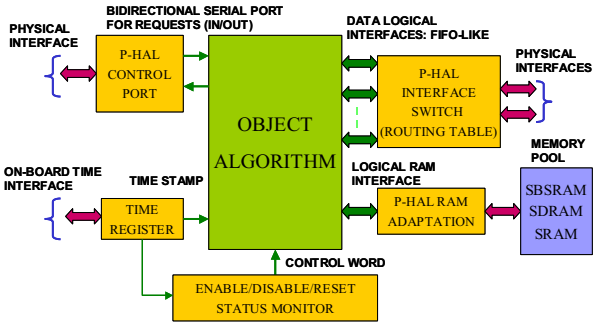

Fig. 2. Program parts in FPGA

The control port, moreover of serving for management purposes, has also a relationship with the algorithm part. With a serial interface, it may request to the object the value of a given parameter or even modify it. When an object has to request a given parameter (e.g. initialisation), it also does it by using the control port.

A special interface, the RAM adaptation, is introduced for those objects that request access to RAM. The object is designed to have a single-cycle synchronous access to RAM. The P-HAL adapter converts cycles to the actual available RAM, including possible wait states. It has been included in the case that FPGA internal RAM is not used or not enough for the application.

Finally, the timing and status resources are just used to allow the algorithm to run in the specific time intervals and under some given circumstances. It must be noted here that one important factor within P-HAL is the absolute control of the execution of the application. Since real-time has to be kept, it is mandatory that the packets arrive within some time bounds and that the objects finish their processing before some deadline.

4. AN FPGA API IMPLEMENTATION CASE

The boxes shown in Fig. 2. can translate to very different implementations depending on the concrete hardware interfaces and algorithm linked to P-HAL library. In the Fig. 3. a slightly more detailed diagram of an API implementation is shown. The FPGA device is inserted in a context where two daisy chain, one bus and one SRAM interfaces are available. Since the filter only has one input interface and another output interface just one input FIFO and an output FIFO are required. The dimensioning of the FIFOs is a key aspect to avoid data overruns. In this case, since the bus interface uses a synchronous protocol with bursts of up to 64 bytes, FIFOs have been dimensioned to have at least 128 bytes (64 words of 16 bits). Since the used FPGA device has some internal memory blocks (BlockRAM of Xilinx Virtex FPGA [5]) that can be used as 256-words of 16-bits FIFO, this has been the finally

selected configuration. Note that the algorithm is not necessarily working at a frequency that has any rational relationship with the sampling frequency. It uses a frequency that is related to the interfaces operation clock, thus avoiding multiple device clocks and synchronisation stages. The clock frequency has to be larger than the samples output rate to compensate the latencies in the interfaces. For low occupancy or dedicated interfaces the increase in clock frequency is not very relevant but in highly busy interfaces it can even twofold or threefold to compensate for inactivity periods. Such increase can be dramatic since the FPGA may not be able to perform computations at this high speed.

The sample algorithm is an in-phase down converter with carrier centred at 1/4, including a symmetric FIR filter of 51 taps (odd coefficients are zero except central one) and a down sampling by a factor of two. The non-zero coefficients are programmable provided that they decay at a minimum rate of 1/t (e.g. windowed *sinc* low pass filter). It computes an output sample every four cycles of clock by using internal RAM tables and distributed arithmetic partial multiplications. Input samples are 10-bits wide while coefficients and output are quantified using 16 bits.

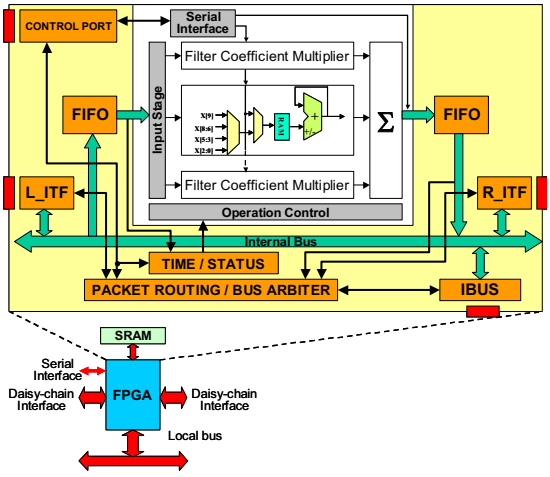

Fig. 3. Simplified diagram of an application of the API

The comparison of resources used by the two parts of the design says that filter requires about 550 Virtex slices (1 slice = 4-bit input 1-bit output LUT + 1 Flip-Flop) while the implementation of P-HAL API requires 2 Block RAM and 250 Virtex slices, both running at a maximum speed of about 64MHz in a Virtex 150 FPGA. This indicates that the complexity of implementing the API for the particular situation presented above (but interesting due to the quite generic approach) is comparable to the implementation of a digital down converter (DDC) without carrier adjust for a radio interface like that used in UTRAN. However, it employs, in this particular implementation, only about the 15% of the total FPGA resources. This figure can significantly change depending on the reference architecture that is taken and on the device being used. The variation is as significant as requiring about the 50% of total resources for API (when using small FPGAs and generic interface configurations, like in the example) to only 0.5% when using large FPGA (> 1M gates) with reduced interfaces.

5. CONCLUSIONS

In this paper it has been presented the translation of the typical software architecture in common processors to FPGA devices. This translation, applied to Software Radio implementations, is seamlessly linked to similar software structures for GPP or DSP to construct abstract radio applications. The abstract context is obtained from the usage of a middleware, called P-HAL, which enables the interaction of different application objects located on different processors of different platforms.

With the presented structure, a suitable framework to build and develop radio applications in a very flexible manner is obtained. The cost of flexibility is an extra utilisation of logical resources, but to confront to future radio access network requirements a high degree of flexibility is required, and FPGAs can incorporate it at a moderated cost. Extending the mechanisms that permit a short development time and low integration cost of software to FPGAs is an interesting path to explore. This represents that the FPGAs have to be considered as processors with their own entity and not mere accelerators attached to other processors.

6. REFERENCES

[1] Jun-Zhao Sun, J. Sauvola, D. Howie, “Features in future: 4G visions from a technical perspective”, IEEE Global Telecommunications Conference. 2001. pp. 3533-3537.

[2] A. K. Salkinitzis, Hong Nie, P. Takis Mathiopoulos, “ADC and DSP Challenges in the Development of Software Radio Base Stations”, IEEE Personal Communications, August 1999, pp. 47-55.

[3] J. Mitola III, “The Software Radio Architecture”, IEEE Communications Magazine, Vol. 33, No. 5, pp. 26-37, May 1995.

[4] J. Mitola, G. Q. Maguire Jr., “Cognitive Radio: Making Software Radios More Personal”, IEEE Communications Magazine, August 1999, pp. 13-18.

[5] Xilinx, Inc. “Virtex FPGAs Data Sheet”, 2002.

ACKNOLEDGEMENT: This work has been supported by CYCIT (Spanish National Science Council) under grant TIC2003-08609, which is partially financed from European Community through the FEDER program.