Implementation of Scalable Embedded FPGA for SOC

Hayder MRABET, Zied MARRAKCHI*, Habib MEHREZ* and André TISSOT***

**LIP6-ASIM Laboratory*
Université Paris 6, Pierre et Marie Curie.
4, Place Jussieu, 75252 Paris, France

***CEA-DAM IDF*
Commissariat à l'Energie Atomique
email:{hayder.mrabet,zied.marrakchi,habib.mehrez}@lip6.fr, andre.tissot@cea.fr

Abstract *—Integrating an embedded FPGA into SoC allows postfabrication changes. Thanks to their unlimited reconfigurability, eFPGAs are able to implement specific functions, thus improves the systems performance. In this paper we present an SRAM-based eFPGA architecture. We explore the hardware aspects of the eFPGA including internal structure and external coupling with a VCI interconnect. We also focus on the design flow for the implementation and the configuration.*

1 Introduction

Since its introduction in the 1980's, the reconfigurable logic has gone through many phases. Earlier Field Programmable Gate Arrays(FPGAs) provided a "sea" of Look-Up Tables(LUTs) and registers which are linked together using programmable interconnects. The programmable elements were built from SRAM cells, the device can be quickly reprogrammed to implement any desired functionality.
In the 90's, FPGAs had high logic capacities enough to implement entire signal processing functions. Since then, Reconfigurable Logic became a standard component in every digital electronic system.
With the enormous movement since 1990's toward System On Chip (SoC), new methods to explore programmable logic have been developped to suit this new environment. The previous year saw the emergence of the System On Programmable Chip (SoPC). SoPCs mix SoC with a configurable fabric that designers can manipulate after chip fabrication. The FPGA companies introduced SoPCs chips as combination of both soft and hard silicon cores embedded into programmble architectures. At this time ALTERA got outside the Excalibur SOPC in two versions ARM-based and MIPS32-based. Xilinx produces a platform with an embedded hard core PowerPC405 and ATMEL defined the RISC-based SoPC AT94K FPSLIC.
Due to the limitations of FPGA technology, these products have the disadvantage of higher power consumption and lower performance compared to the standard cell ASIC. This makes SoPCs mainly used for prototyping and applications requiring low volume production. SoCs using ASIC implementation technology are more attractive for high volume applications and offers higher performance and density and lower power. But this SoCs needs supleness. Once it is fabricated, it is impossible to introduce new functionalities or the slightest design modification.

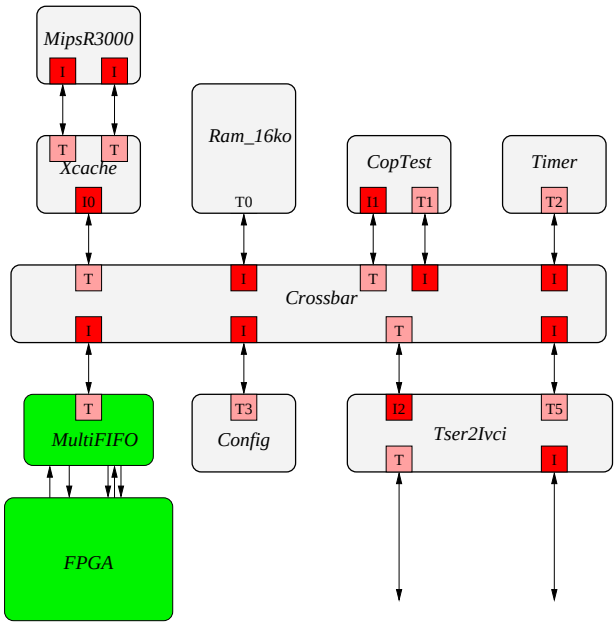

Figure 1: AOC (**A**sim **O**n **C**hip) platform

Recently we have seen the insertion of field programmble cells into ASICs. FPGA vendor [1] and new IP developers [2, 3] are now offering "hard" embedded FPGA cores (eFPGA) that can be incorporated into SoC design. Include an eFPGA into a SoC gives the possibility to reuse a portion of the chip for a variety of tasks and functionalities like image analysis and signal processing. Thanks to his

flexibility the eFPGA can cover design error or specification change after fabrication like communication standard.
A general platform that contain the necessary for any SoC design was developed in the ASIM laboratory. It is called AOC (Asim On Chip). It is based on a MIPS processor, and along with it, a VCI bus, some internal memeory and some external interface. We extand AOC by including a VCI target eFPGA core. An overall view of the AOC SoC is provided by figure 1.
In this paper we present a reconfigurable hardware designed specifically to be integrated in a SoC. Thanks to its scalable structure and its generic VCI interface, the eFPGA can be adapted exactly to designers specifications. This eFPGA can easily be integrated into a processor-based system in order to offload some the most computationally intensive tasks.

This paper is organized as follows: the following section describes the architecture of the eFPGA. Section 3 describes the entire system design flow for the eFPGA. Section 4 concludes this work.

2 Scalable eFPGA

To be effective, it is necessary to minimize any redesign of the base platform elements or add-on cores to create new products. To achieve this, platforms must have been built on a foundation of reusable Intellectual Property blocks designed to a standard interface.
Satandard VCI(Virtual Component Interface) [5] is adopted for designing and integrating our eFPGA as a virtual component.
The eFPGA architecture is composed of a programmable logic array, a VCI interface module and a loader for the configuration.

2.1 Programmable Logic Array

Most applications to be implemented into the eFPGA are divided to control and data sections, with control involving mostly single-bit random logic, and the datapath comprising larger wider operation. That is why we construct a simple model with fine granularity that can roughly emulate a larger granularity when desired.
The core of the eFPGA has island-style strucure (see figure 2). It is an array of Configurable Logic Blocks(CLBs) whose functionality is determined through multiple programmable bits. Each CLB contains two 4-input LUT (look Up Table), each LUT is followed by a bypass flip-flop. Bidirectional wires run vertically and horizontally within the array to connect CLBs. All wires in the network are grouped into horizontal and vertical routing channels which are inked together through programmables switch boxes. For the simplicity of the design we do not use long wires, all wires have a length one (span one logic block).
Each CLB has two outputs that can drive all the adjacent wires in the top and the right sides. It has also 6 inputs that can read from any adjacent wire in the four sides.

The regularity of the island-style simplifies the automatic structuring of the Programmable array Layout. In [4] we

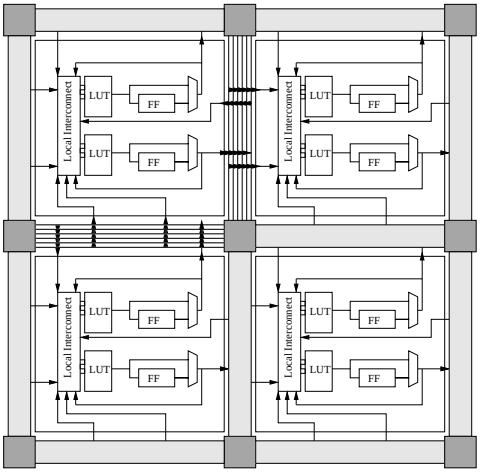

Figure 2: Programmable Logic Array Architecture

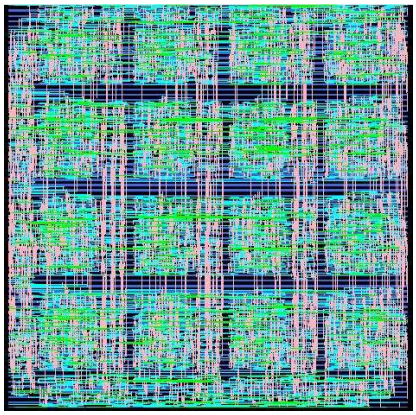
Figure 3: 4x4 eFPGA prototype

decribed a technique to automate the array layout generation. Using this technique we are capable to generate a large spectrum of different array sizes to obtain the best design fit on the SoC device. The parameters for the generator are the number of CLB per Row and the number of CLB per column. The final routed layout example of the programmable array is given in figure 3. It is an array of size 4x4, with an equivalent logic capacity of 384 gates. The layout measures $0.453mm$x$0.456mm$ in CMOS 0.13μ and is routed using 4 layers of metal. Table 1 illustrates details of different arrays generated automatically.

array size	Logic Capacity (gate)	number of transistor	Area in CMOS 0.13μ (mm^2)
4 x 2	192	25588	0.109
2 x 6	288	37728	0.159
4 x 4	384	48288	0.216
4 x 8	768	93688	0.402
8 x 8	1536	181328	0.786

2.2 Generic e-FPGA Interface

The efficiency of data transfer during communication between the processor and the coprocessor is often the dominant factor determining overall system performance.
Coprocessor that will be implemented on the Programmable array will use defined standard FIFO interface. A multififo module will include the VCI interface in the bus side. This module is called the VCI target Multififo. It is composed essentially of:

- Two sets of FIFOs for performing read-from-bus and write-on-bus connected respectively to inputs and outputs pins of the array. Their number can be varied between 1 and 4 FIFOs in each direction.
  One "Read FIFO" is also used to transfer the configuration data to the loader and the programmable array.
- A "configuration" register that can be programmed by the main processor to set the eFPGA's functional mode.
- A "status" register which indicates the status of the eFPGA.

An overview of the Multififo interface module with one read and one write FIFOs is given in the figure 4. The data size is considered as fixed to 32 bit, like the VCI cell size. Specific pins are reserved on the periphery of the array for the fifo control. Each read FIFO provides three ports for access by the programmable coprocessor: DOUT, ROK, READ. Each write Fifo provides three ports DIN, WOK, WRITE. The FIFO protocol is mapped on the programmable array to insure communication with the FIFOs. We note that a same read FIFO can insure the data transfer to the coprosessor and the configuration data transfer to the loader too.

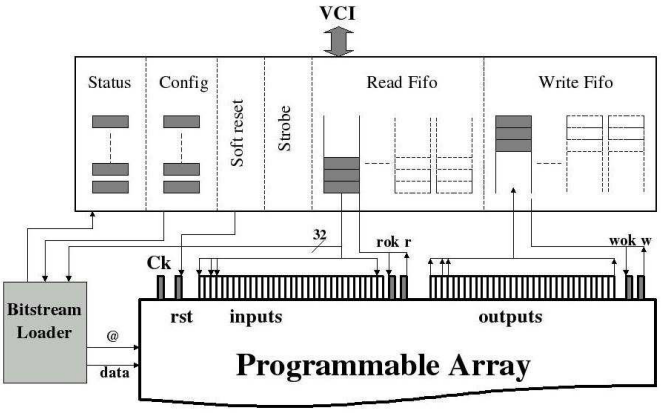

Figure 4: The eFPGA interface

2.3 Configuration Loading

Because reconfiguration can take milliseconds or longer, rapid and efficient configuration becomes a critical issue.
Frequently, programs that can be accelerated through a mapping on reconfigurable hardware are to numerous to be loaded simultaneously onto the available area. A number of different configurations may be loaded onto the same area at different times. In some cases only a part of the configuration requires modifications when the rest should remain intact.
It is benificial to be able to swap easily different configurations as they are needed during program execution.
In our eFPGA we use a seletive configuration using the RAM technique. The random access allows a partially and a fast dynamic reconfiguration.
Compared to other technique like the shift register configuration, this one requires an important number of pins dedicated to the configuration loading. For an embedded FPGA, the number of pins is not critical (as long as we have not a problem of packaging).
The configuration ressources are grouped as tiles of 16-bits words that can be randomly accessed and can be programmed on the fly. A small state machine named "The loader" is responsible of programming the array. It is connected to the programmable array and to the multififo interface module.
When the "configuration" register is set on the configuration mode, the loader recieves (through the "Read-FIFO") a sequence of 32 bit-words that corresponds to a series of consecutive 32-bit address and 32-bit data. The loader decodes the address and loads the adequate data (In this work the most signifigant 16 bits of each data word are ignored).

3 Design Flow

3.1 Exploration Environment

The starting point is a standard C code writen to describe a specific functionality. Simulations assert that the most execution time(90%) is generally consumed by about 10% of a program's code. The goal is to group those segments of codes that increase timing consuming and implement them on the reconfigurable hardware.
The programmer is responsible of extracting the critical segments and dividing a program between the main processor and the reconfigurable hardware. Those segments are gen-

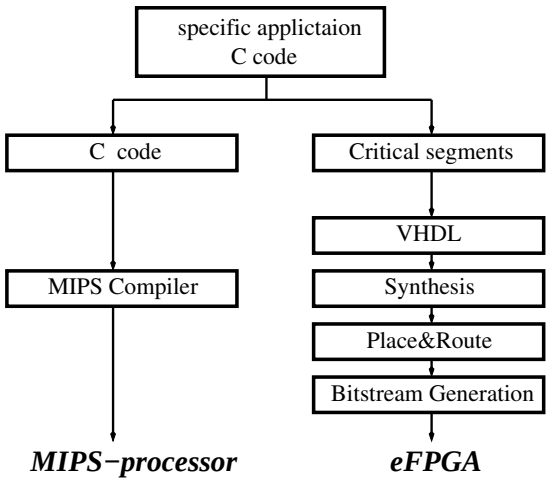

Figure 5: Configuration Flow

erally inner loops that have small static code but potentially large instructions count. The VHDL resulting from the refinement of those segments is easily mapped on the eFPGA.
An automatic flow shown in figure 5 generates the config-

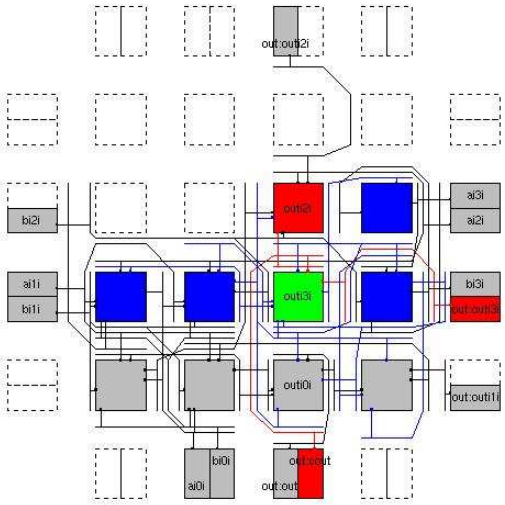

Figure 6: Place&route using VPR

uration Bitstream corresponding to the hardware mapped on the eFPGA.
A complete design flow for mapping the VHDL descriptions on the eFPGA has been based mainly on open source publicly available tools. The flow includes 'boog' [6] for logic synthesis, SIS [8] for mapping, T-vpack+VPR [7]for place and route. We developed a generic generator of bitstream. Figure 6 shows an example of the mapping phase results on 4x4 eFPGA.
As a final result, we obtain the same original functionality but partioned on :

- software C code to be executed by the processor.
- hardware mapped on the eFPGA.

3.2 RTL Level / Hardware design

SystemC Model : The full system of figure 1 has been modelised on SystemC first. All the AOC components, except the programmable array, are open models of SoCLib [9]. All simulation models writen in SystemC and respect the VCI communication protocol.
A variety of applications are simulated. Arithmetic and logic benchmarcks are implemented on the eFPGA. The binary Bitstream is stored in the system RAM (or an external RAM/ROM). Then it is loaded into the eFPGA through the system bus and the eFPGA interface. In this configuration stage the main processor(MIPS) make the role of a DMA to transfer configuration data.
System to layout: System Co-simulations of the same benchmarks are made with the eFPGA on VHDL (the rest is always on SystemC).
The eFPGA interface module and the loader are easily exported on standard VHDL and are synthesized using the target Sxlib [6] cells library.
The programmbale array netlist and layout (see Figure 3) are constructed simply by replicating the main tiles (CLB + Connection box + Switch box).

4 Conclusions

eFPGA coprocessor architecture greatly simplifies the process of offloading computationally intensive functions from a programmable processor(Mips) into hardware. Those applications can be greatly accelerated.
The reconfigurable structure, the interface module and the reconfiguration technique are all key points in the design of reconfigurable coprocessor. We define each module in order to perform a small, fast and scalable eFPGA.
We use flexible modules that allows to generate a large spectrum of architectures with different sizes. The scalability of the mesh architecture can be varied to obtain the best design fit. The generic multififo interface can be easily adapted to any size of the programmable array simply by modifing the number and the size of the FIFOs. Finally, the Random Acces insured by the loader module can cover a broad memories area and it does not need any modification.
This paper has described the implementation of an embedded FPGA, and has illustrated the flow to design and explore a programmable hardware core. This eFPGA has been integrated into System-On-Chip platform that will be realized on 0.13μ CMOS technology.

References

[1] Actel Corp, "VariCore Embedded Programmable Gate Array Core (EPGA) 0.18m Family". Datasheet, December 2001.

[2] eASIC Corp, "FlexASIC 0.13μ Core", http://www.eASIC.com.

[3] M2000 Inc , "M2000 FLEXEOStm Configurable IP Core", http://www.m2000.fr.

[4] H.Mrabet, Z.Marrakchi, H.Mehrez, A.Tissot,"Automatic Layout of Scalable Embedded Field Programmable Gate Array". International Conference on Electrical Electronic and Computer Engineering (ICEEC'04), Cairo, Egypt, September 2004, pp. 469-472

[5] Virtual Socket Interface Alliance, Virtual Component Interface Standard Draft Specification, v.2.2.0, http://www.vsia.com, August 1997.

[6] www-asim.lip6.fr/recherche/alliance, ALLIANCE CAD.

[7] www.eecg.toronto.edu/ vaughn/vpr/vpr.html

[8] www-cad.eecs.berkeley.edu/Software/software.html

[9] http://soclib.lip6.fr