Hardware and Software Infrastructure to Implement Many-core Systems in Modern FPGAs

Felipe T. Bortolon
PUCRS - Porto Alegre, Brazil
felipe.bortolon@acad.pucrs.br

Fernando G. Moraes
PUCRS - Porto Alegre, Brazil
fernando.moraes@pucrs.br

ABSTRACT

Many-core systems are increasingly popular in embedded systems due to their high-performance and flexibility to execute different workloads. These many-core systems provide a rich processing fabric but lack the flexibility to accelerate critical operations with dedicated hardware cores. Modern Field Programmable Gate-Arrays (FPGAs) evolved to more than reconfigurable devices, providing embedded hard-core processors with several IP cores. While FPGAs provide a rich reconfigurable hardware fabric, only one or two embedded hard-core processors are available to execute complex software applications. Therefore, modern FPGAs offer the possibility to merge the benefits of many-core systems with the reconfigurability of FPGAs. The goal of this paper is to present an infrastructure to implement many-core systems in modern FPGAs. An embedded hard-core processor is used to manage the many-core area, and to communicate with a host computer. A functional proof-of-concept system is presented, paving the way to connect dedicated hardware IPs into the NoC.

CCS CONCEPTS

•**Hardware** →*Programmable logic elements;*

KEYWORDS

FPGA, MPSoC, Reconfigurable Systems, System Management

ACM Reference format:
Felipe T. Bortolon and Fernando G. Moraes. 2017. Hardware and Software Infrastructure to Implement Many-core Systems in Modern FPGAs. In *Proceedings of SBCCI '17, Fortaleza - Ceará , Brazil, August 28-September 01, 2017,* 5 pages.
DOI: 10.1145/3109984.3109997

1 INTRODUCTION

The unending demand for higher performance allied to low energy consumption points to the adoption of multi/many-core systems, as the most promising solution to cope with the application requirements. Underlying embedded systems can execute different applications in parallel in order to improve system performance and energy-efficiency. Typically, applications designed to run in such systems are partitioned into different tasks to execute in different cores, enabling its parallel execution. While replicating processor elements improve the system performance, designing and managing such systems still a challenge.

Another worthwhile alternative to meet application requirements, while ensuring performance and energy-efficiency, is the use of FPGAs. In this case, application tasks may be implemented either in software or hardware, depending on its performance/power requirements. Due to the increasing performance requirements of emerging embedded applications, FPGAs are becoming more than reconfigurable devices. Modern FPGAs, besides the programmable logic (PL), provide hard-core processors with several subsystems, e.g. floating-point unit, cache memories, on-chip memory (OCM), and dedicated IP blocks. Different markets adopt FPGAs in their final application products, like medical, avionics, automotive, consumer electronics, data centers, security, and communication. While FPGAs provide a cost-effective programmability and a rich reconfigurable hardware fabric, only one or two embedded hard-core processors are available to execute complex software applications. Resulting scenario reduces the relevance of such devices to execute applications with scalable performance and energy-efficiency requirements.

More recently, in the attempt of combining the performance scalability of many-core systems with the design flexibility of FPGAs, authors have been exploring the characteristics of modern FPGAs to implement multi/many-core systems. In this regard, the benefits of both approaches could be tuned in order to meet application requirements. For instance, in [6] a homogenous soft-core (i.e. MB-lite) based multiprocessing system has been designed in a Spartan-3E device. A similar architecture is presented in [11]. In this work, the author creates a homogeneous multi-core system using a binary tree NoC and four OpenRisc1200 processors in a Virtex4 FPGA. The application executed in the system, however, is hardcoded in the memory, and thus, a new synthesis is required to change the test software. Moreover, promoted solution uses an application processor family that requires a non-negligible amount of area and power. Different from previous approaches, a heterogeneous multi-core system was implemented in Virtex-II Pro [7]. In this work, authors propose an FPGA-based middleware structure to manage communication and synchronization between heterogeneous processors.

An important issue in many-core systems is its runtime management and monitoring capabilities [9]. Different from above approaches, this paper *addresses* this issue by using a hard-core processor to manage a many-core system, implemented in the programmable logic (PL), in such a way to control the applications executing in the system at run-time.

The main *contribution* of this paper is twofold. The first one is the deployment of an FPGA-based many-core platform, enabling

SBCCI '17, Fortaleza - Ceará , Brazil

DOI: 10.1145/3109984.3109997

to explore the parallelism offered by several processing units to execute complex applications. The second one is the possibility to explore complex management techniques, like dynamic task mapping and QoS control, since the hard-core processor offers a larger computational power compared to the cores composing the many-core system.

This paper is organized as follows. Section 2 presents the Zynq-7000 All Programmable SoC. Section 3 presents the hardware and software inftrastructure. Section 4 details the MPSoC architecture. Section 5 details the main contribution, the implementation of the MPSoC in the Zynq-7000 platform. Section 6 concludes this paper.

2 THE ALL PROGRAMMABLE SOC

The Zynq-7000 All Programmable SoC [4][5] is a family of devices that combines the software programmability of processors with the hardware reconfigurability of an FPGA. Those devices contain two main regions, the Processing System (PS), which contains one or two ARM processors, and a 7-Series PL fabricated with 28-nm technology. The PS further contains instruction and data caches, OCM, DMA controller, input-output peripherals, among other subsystems. Figure 1 details this architecture.

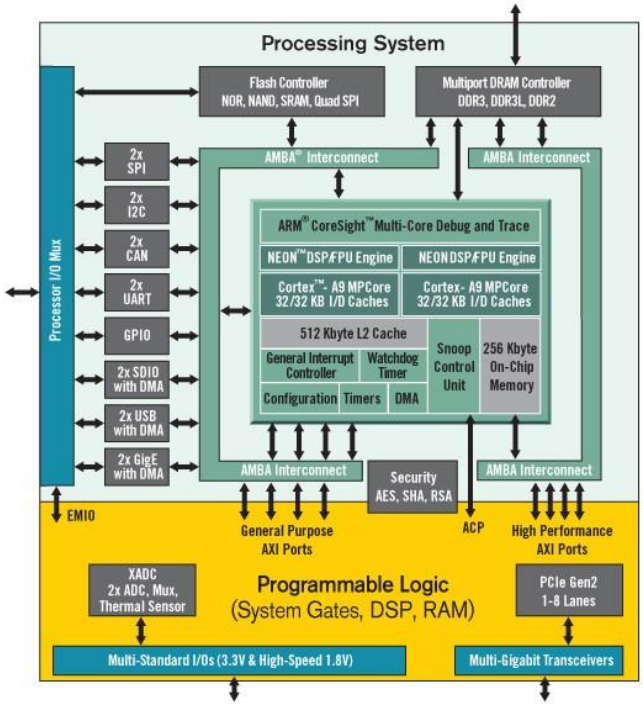

Figure 1: Zynq-7000 device family configuration [6].

While the PL section is appropriate for implementing parallel logic, the PS is suitable for implementing complex algorithms. To benefit from those characteristics, the Advanced Extensible Interface (AXI) bus, which is part of ARM AMBA [2] family creates the necessary connection between the PL and the PS. Therefore, the overall functionality of an embedded system can be partitioned into hardware and software parts [8].

3 HARDWARE AND SOFTWARE INFRASTRUCTURE

This Section presents the *main contribution* of this work: the infrastructure to implement many-core systems in FPGA. The promoted infrastructure includes a 3-layer software stack and additional hardware modules, which allow implementing and managing many-core systems in FPGAs. The software stack comprises: (*i*) the *software interface* - API to communicate with the host computer; (*ii*) the *management software*, responsible for the core functions to manage the system; and (*iii*) the *hardware interface* - API to communicate with the programmable logic, which also comprises additional hardware to establish the link between the PS and the PL.

Figure 2 depicts the overall architecture targeting the Zynq device. A *host computer* communicates with the ARM processor, which may either require to communicate with the PL (*user logic*), or to execute some internal processing (on the ARM processor). The PL may also trigger requests to the processor (e.g. management purposes) or transmit some information to the host computer. Therefore, the software layers are used to provide the necessary communication mechanisms between these three hardware components: the host computer, the ARM processor, and the programmable logic.

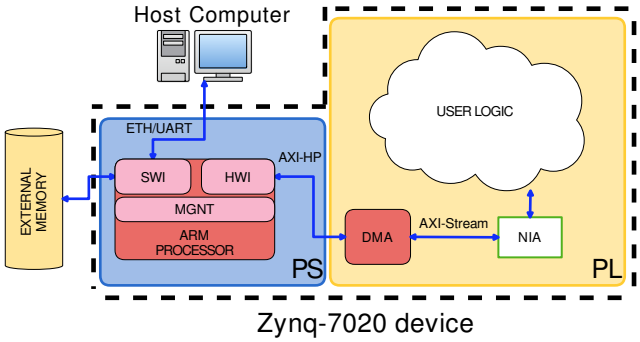

Figure 2: Zynq-7020 emulation setup.

3.1 Software Interface (SWI)

The Software Interface (SWI) provides means to configure the system, extract debug information, upload applications and retrieve results to/from the host computer. The SWI uses three methods to handle the communication between the ARM processor and the host: (*i*) command interface, (*ii*) file transfer, and (*iii*) debug interface.

The *command interface* method is used to handle the communication between the host and the system through Telnet (using an Ethernet connection). Upon reception of a message from the host, the SWI decodes the message. As the semantics of the messages are tightly related to the user logic prototyped in the PL, a generic method is adopted, with a pair *command name* and *command arguments*. Commands may send/receive data to/from the PL.

The SWI provides functions to the ARM processor to access an external memory and to the host to download/upload applications or data to the PL. The Trivial File Transfer Protocol (TFTP) is adopted due to its low memory footprint implementation, which is

important for embedded systems due to memory constraints. As mentioned in Section 1, using local storage in the system allows fast data transfer between PS and PL.

The *debug interface* uses the UART to send debug messages data to the host. The debug data may be issued from the PS or PL parts of the system.

3.2 Hardware Interface (SWI)

The Hardware Interface (HWI) provides the communication link between the management software and the programmable logic. To optimize the PS-PL communication performance, the HWI access a DMA core [3], implemented in the PL, through an API.

The DMA core is configured to operate using the AXI-Stream protocol because it is a simple and flexible interface, designed to transport arbitrary unidirectional data streams. This module is configured with four communication channels, two *slaves* (for data reception) and two *masters* (for data transmission). Figure 3 shows the protocol operation from the *slave* channel's perspective. The *TREADY* signal indicates when the slave is ready to receive new data. If this condition is true, the sender asserts *TVALID* to indicate that there is data available in *TDATA* bus. The *TSTRB* bits are set alongside *TVALID* to indicate whether *TDATA* bytes are regular data or transfer control information. This process is repeated until the end-of-transfer, which is signaled by *TLAST*. Figure 3 shows a data transfer with four words.

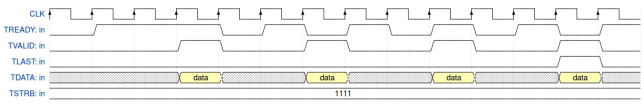

Figure 3: AXI-Stream protocol waveform.

The API provides functions to access the DMA core following the same principle of the SWI interface: generality. Nevertheless, the *user logic* varies according to the user design and may require additional hardware to convert its signals to the DMA standard. Figure 2 shows a module name *NIA* (Network Interface Adapter), which adapts the AXI-Stream to a network-on-chip (NoC) protocol. Different adapters must be designed, according to the *user logic* communication protocol.

3.3 Management Software

Contrarily to the SWI and the HWI, the management software (MGNT) is developed according to the design implemented in the PL. The MGNT uses both the SWI and the HWI APIs to implement the system management. Moreover, this software layer is explored to develop management algorithms, as detailed in Section 5.

4 MANY-CORE ARCHITECTURE MODEL

This section overviews the public-available HeMPS [1] many-core system, adopted as a case study in this work. HeMPS is a homogenous NoC-based MPSoC, which includes processing elements (PEs), a NoC, and an application repository. Figure 4 presents an instance of the HeMPS MPSoC with 16 PEs.

PEs may assume two distinct roles (the hardware is the same, the software differentiates the PEs): manager or slave. A manager PE (*MP*) controls the system, receiving applications from the external

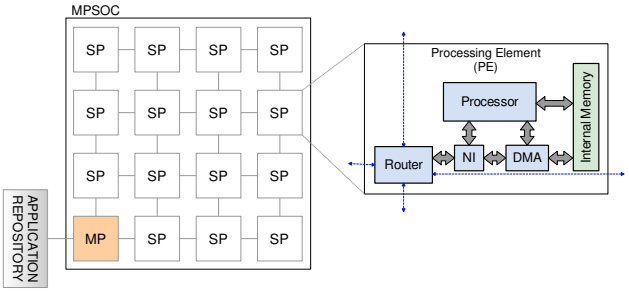

Figure 4: HeMPS instance with 16 processing elements.

memory (*application repository*), executes task mapping/migration, receives debug messages, among other functions. Slaves PEs (*SPs*) execute user tasks. Each *SP* contains a Plasma processor [10], and executes a small operating system (μkernel), enabling multi-tasking and inter-task communication through message passing.

The NoC router communicates with the PEs using a synchronous credit-based protocol. This protocol has a signal named *credit*, indicating that the router may receive new data. If this signal is asserted, the upstream router asserts the *tx* signal, and one 32-bit work (*flit*) is sent per clock cycle.

4.1 Application Model

An m-task application $A = \{t_1, t_2, ..., t_m\}$ is modeled by a directed acyclic task graph $G(T, E)$, with each vertex $t_i \in T$ representing a task and the directed edge (e_i, e_j), denoted as $e_{ij} \in E$, representing the communication between tasks t_i and t_j. Tasks communicate using a *Send* and *Receive* MPI-like primitives.

Tasks are classified as *initial* or *non-initial*. Initial tasks are those that do not have dependencies from other tasks to start executing and can generate data to other tasks. Non-initial tasks start their execution when another task sends data to it. The manager processor first allocates the initial tasks into the available SPs. Then, as the allocated tasks start to communicate with unallocated tasks, i.e. non-initial tasks, the slave SP request to the MP to map them.

5 CASE STUDY

This Section presents a proof-of-concept implementation using the infrastructure detailed in the previous Section. This implementation integrates the infrastructure presented in Section 3 with the MPSoC presented in Section 4. The *goal* is to use the ARM processor as the manager PE (MP), providing a *generic* and *flexible* environment to communicate with a host processor, and to execute the MPSoC management functions.

Figure 5 details the proposed system architecture. The PL receives the MPSoC, replacing the MP by the DMA and NIA cores. The ARM processor, in the PS, executes the SWI, the HWI and the management functions originally executed in the MP.

5.1 Hardware

Figure 6 presents the floorplanning of the system illustrated in Figure 5. Note the small area of the PEs (Plasma IPs) compared to other modules. The small area footprint of the selected processor enables to implement larger systems.

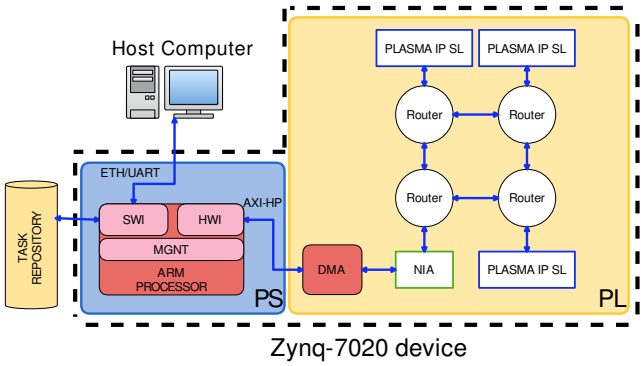

Figure 5: Architecture of the prototyped system.

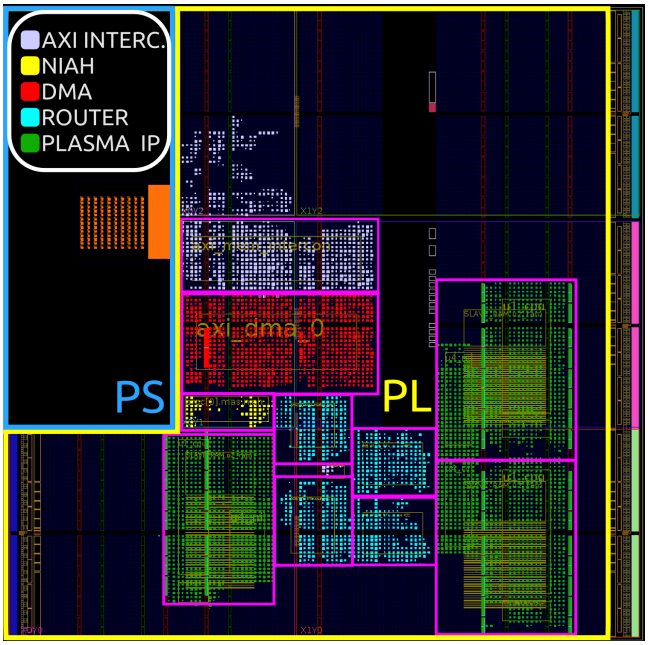

Figure 6: Programmable logic floorplanning - device Zynq Z-7020.

The internal memory of each SP contains 64 KB, requiring 16 36-Kb BRAMs (memory blocks available in the FPGA). The bitstream embeds the μkernel of each SP (by initializing the BRAMs). The code of the tasks to execute is transferred from the external memory (SD card), through the NoC, to the selected SP.

The NIA module converts DMA AXI-Stream to the synchronous credit-based protocol. Two independent finite state machines achieve this conversion (transmission and reception).

Additionally, the μkernel was extended with a *debug service*. When a given task executes a *print* command, it is generated a debug packet addressed to the NIA. The debug packet is received in the HWI and transmitted to the host (by the UART) by the SWI.

5.2 Software

The Software Interface (SWI) initially receives applications from the host computer, storing them in an external memory (SD card) using TFTP. Once applications are received, the host sends *commands* to the management software through the SWI, using telnet. The management software treats four command types:

- *hemps start*: initialization packets and the address of the NIA to each SP.
- *hemps reset*: resets the MPSoC.
- *hemps load*: has an application file as a parameter. The PS reads the application description (graph G(T, E)), enabling the PS to execute the application mapping when the application is requested to execute. Several applications' descriptions may be loaded simultaneously.
- *hemps run*: start to map the initial task into the available SPs. All applications loaded with *hemps load* are sequentially allocated the SPs.

Those commands trigger the HWI API to execute the required operations. After mapping the initial tasks into SPs, the PS waits for control flow requests, e.g. request to map non-initial tasks, or a message indicating that a slave finished the execution of a given task.

Besides these functions, the software also sends debugging messages (from the PS or PL) to the host computer using the UART. Those messages are related to the user applications and are explored for both software debugging, i.e. Plasma SP software, or to verify the correctness of the application execution.

5.3 Synthesis Results

The hardware design achieved 51 MHz clock frequency using optimized logical and physical synthesis settings, combined with floorplanning (Figure 6). The Processing System AXI interfaces may operate at 250 MHz. Hence, it is possible to enhance further the frequency in the reconfigurable area.

The Zynq-7020 programmable logic utilization report, for the complete design, is shown in Table 1. Special blocks, e.g. DSP and XADC, are not used and thus do not integrate this table.

Table 1: Complete design area utilization report.

Resource	Utilization	Available	Usage (%)
Slice LUTs	18,045	53,200	33.92
Slice Registers	10,467	106,400	9.84
N# BRAMs (36Kb)	50	140	35.71

The number of available BRAMs limits the size of the MPSoC. Moreover, the BRAMs column-wise organization forces the Plasma SP to extend its floorplan area to enclose the necessary memory elements and hence increase the design critical path. Using LUTs to implement the memory was discarded because it would require too many resources for 64KB. If the Zynq Z-7100 device is adopted, it may receive up to 47 PEs (755 36-Kb BRAMs / 16).

5.4 Execution of a Real Application

This section presents the execution of an MPEG decoder, with five tasks, as shown in Figure 7. The first task sends 8x8 frames to the ILVC task, and the application processes frames in a pipeline fashion.

Figure 7: Task graph of the MPEG decoder application.

On startup, the Management Software (MGNT) initializes the interface with both the host computer and the programmable logic. In Figure 8 is illustrated the interaction between the host computer and the MGNT from the device's perspective.

```
1.  TFTP: Write request: mpeg_up.txt
2.  hemps start: Initializing slaves.
3.  hemps load: application loaded
4.   hemps run: Allocate initial tasks.
5.  > task [00000003] allocated to [00000100]
6.  Wait for requests from slave PEs.
7.  > received request from task [00000003]
8.  > task [00000004] allocated to [00000001]
9.  > replied task request message to [00000100].
10. > replied task request message to [00000001].
11. > received request from task [00000004]
...
15.  > received request from task [00000002]
...
19. > received request from task [00000001]
...
24. > task [00000003] terminated.
25. > task [00000004] terminated.
26. > task [00000002] terminated.
27.  > task [00000001] terminated.
28. > task [00000000] terminated.
29. > application [00000000] terminated.
```

Figure 8: MPEG execution flow from Zynq system perspective (messages received from UART).

First, the user transfers an application file to the device through TFTP (line 1). Next, the user issues the previously described commands in the following order (lines 2-4): *hemps start*, *hemps load* and *hemps run*. After the run request, the MGNT maps all initial tasks to the processing elements and waits for requests from those (lines 5-6). The PEs requests will trigger the mapping of non-initial tasks. Moreover, the MGNT further controls where tasks are mapped to inform the location of tasks that need to communicate (line 9-10). This process is repeated as long as there is still tasks to execute and PEs to receive tasks (lines 11-23). Lines 24 to 29 are related to the reception of end of task messages.

For 512 frames, the simulation time (Intel Xeon 8 Core, 24 GB of RAM) the simulation of the application required 12,000 seconds, while in the board 2 seconds. A speedup of 3-orders of magnitude is an expected result but highlights the relevance to prototype complex systems to enable the evaluation the complex systems.

6 CONCLUSIONS

This paper presented a software and hardware infrastructure to implement many-core systems in FPGAs, with a hard-core processor (ARM) to manage the system and to communicate with a host computer. The proof-of-concept is simple in terms of numbers of cores but showed the feasibility to use the hard-core processor to manage the system. A hard-core processor brings flexibility to the design, easing the communication with the host processor.

This work may evolve in several directions. The first one is to deploy a large system in devices such as the Zynq Z-7100, enabling the evaluation of complex applications in systems with more than 40 processors. The second one is to evaluate different management techniques, as task mapping, QoS control and DVFS for power savings. A third research direction is to implement heterogeneous systems, with dedicated hardware accelerators connected to the NoC.

7 ACKNOWLEDGEMENT

The Author Fernando Gehm Moraes is supported by CNPq funding agency.

REFERENCES

[1] N. Calazans F. Moraes. E. Carara, R. Oliveira. 2008. HeMPS - A Framework for NoC-Based MPSoC Generation. In *ISCAS*. pp. 1345–1348.
[2] Xilinx Inc. 2012. AXI Reference Guide. In *UG761*.
[3] Xilinx Inc. 2015. AXI DMA v7.1 – LogiCORE IP Product Guide. Xilinx Inc., pp 77–81.
[4] Xilinx Inc. 2015. Zynq-7000 All Programmable SoC. 1863p.
[5] Xilinx Inc. 2015. Zynq-7000 All Programmable SoC – Technical Reference Manual. 116p.
[6] W. M. Jose. 2009. Multiprocessor system in an FPGA. In *International Conference on Reconfigurable Computing and FPGA's*. 273–278.
[7] F.Rousseau; F. Petrot Kouadri M, A.M. 2008. Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers. In *RSP*.
[8] M. Stewart. L. Crockett, R. Elliot. 2014. *The Zynq Book* (1 ed.). Strathclyde Academic Media, 484p.
[9] P. Liljeberg J. Plosila. M. Fattah, M. Daneshtalab. 2011. Exploration of MPSoC Monitoring and Management Systems. In *ReCoSoC*.
[10] S. Rhoads. Plasma CPU. http://plasmacpu.no-ip.org/. (????). [Online; accessed Aug-2015].
[11] N. D. Torring. 2007. Multiprocessor in a FPGA. In *Informatics and Mathematical Modelling*.