Ouessant: Flexible Integration of Dedicated Coprocessors in Systems On Chip

Pierre-Henri Horrein, Philip-Dylan Gleonec, Erwan Libessart, André Lalevée and Matthieu Arzel
LabSTICC/Télécom Bretagne
Brest, France
{ph.horrein, pd.gleonec, erwan.libessart, andre.lalevee and mathieu.arzel}@telecom-bretagne.eu

***Abstract*—Integration of hardware accelerators in System on Chips is often complex. When dealing with reconfigurable hardware, this greatly limits the attainable flexibility. In this paper, we propose an approach based on a dedicated instruction set designed to manage data transfer and execution. This approach, named Ouessant, is based on a very simple general purpose instruction set designed for close interaction with dedicated hardware accelerators. This instruction set is used to program a dedicated controler, which commands the accelerator's execution and data transfer with minimal CPU intervention. The resulting architecture is flexible, extensible, and can be easily integrated in System on Chips. Adding new accelerators is also made easier. Implementation of the architecture on different FPGA resources show very low footprint and a very small impact on attainable performance. Ouessant is freely available under an open-source license.**

I. Introduction

General-Purpose Processors (GPP) are designed to be usable for a wide range of applications. However, this flexibility comes at a cost. Dealing with compute-intensive tasks such as signal processing or multimedia presents challenging performance issues. In order to reach required computing throughputs without increasing power or surface consumption, a solution is to use Hardware(HW)/Software (SW) co-design. In this approach, compute-intensive parts are offloaded to dedicated coprocessors. Most modern Systems on Chip (SoCs) use this solution. For example, smartphones SoCs integrate hardware video decoders, in order to provide flawless High-Definition video playback, which can not be obtained with low-power GPP cores.

The design of dedicated coprocessors can be divided into two distinct problems: the design of an efficient computing architecture, and integration of this design in the complete SoC. In this paper, we focus on the integration part, which consists in offering control of the coprocessor and on managing communication with the complete SoC. Integration is often a constraint on coprocessor. The communication method of the system must be used, and a control interface for the central GPP must be provided. It can thus quickly become a bottleneck in coprocessor performance: even if the coprocessor is very fast once data is available, if data transmission and control from the processor is inefficient, attainable computing throughput will stay low. The development of reconfigurable computing is also leading to new solutions for HW/SW co-design, with more flexibility in the design. This flexibility must be supported by hardware integration techniques, which adds more constraints to the design.

In this paper, in order to tackle this hardware integration challenge, we propose a new microcontroller-based approach. This approach, named Ouessant, uses a small instruction set design to control accelerator usage. It is designed to be as independant of the system as possible. It provides a low-overhead, flexible, portable, and easy-to-use integration method for hardware accelerator in reconfigurable platforms. It comes with efficient software integration for either baremetal applications, or Linux programs. Ouessant is available [1] under a CeCILL-C license, which is Free Software and compatible with LGPL license.

The paper is organized as follows. In Section II, some works on coprocessor integration in SoC are presented, and the proposed approach is then described. In Section III, the resulting hardware architecture is detailed, before presenting software integration in Section IV. Implementation details and preliminary results on the architecture are given in Section V, before concluding in Section VI.

II. Existing work and proposed approach

A. Existing work

Integrating a coprocessor in a system can be done using different approaches, according to the requirements. The typical way is to connect coprocessors on a bus. They can thus be easily accessed by a processor. They are usually seen as slaves, with different registers for the configuration. Data access is done either through common access to memory, or through integrated First In/First Out (FIFO) communication devices. Communication can be offloaded to a Direct Memory Access (DMA) peripheral, in order to free GPP time.

Other approaches offer more flexibility and ease integration. Vuletic *et al.* [2] proposed to use virtual memory in order to share memory between hardware and software tasks. This is done to increase applications portability between platforms and to ease the application developer work, since no prior knowledge of platform-specific parameters is required. Andrews *et al.* [3][4] proposed the HThreads framework, in which the whole software thread machinery is reimplemented in hardware, with rather high-level communication mechanisms. This solution offers tight integration with a POSIX system, through the compatibility of HThreads with POSIX threads. However, it suffers from the complexity of overhead of threads. The Molen polymorphic processor [5][6] is based on a small dedicated instruction set. This set is defined whatever the accelerators will be in the system, thus limiting the number of instruction The coprocessor is then integrated between the processor and the bus, providing an extension to the instruction set of the GPP. This approach is completely transparent and provides acceleration with a very low time overhead. However, it requires access to the bus/processor interface, and it requires one accelarator per processor, making it inefficient in MultiProcessor System on Chips (MPSoC).

978-3-9815370-6-2/DATE16/©2016 EDAA

B. Ouessant approach and aims

Ouessant aims at providing a microcontroller approach to hardware integration. It uses the instruction set based approach of Molen in a different way. One of the main issues with Molen is the interface with the system. Interfacing the Molen between the processor and the bus is an interesting but complex choice. While it provides transparency and low latency access to the accelerator, it prevents parallelization between hardware and processor, and it cannot be used in hardcore processors such as the Zynq system designed by Xilinx.

With an Ouessantnt coprocessor (OCP), the aim is not to be completely transparent. Transparency for end user can be achieved through software libraries. One of the aims of the coprocessor is to be portable, meaning independant from the processor. In order to benefit from Molen advantages, while counterbalancing the previously described side effects, different choices have been made. The coprocessor still uses a basic instruction set to control the accelerator. However, more instructions are envisioned, in order to provide increased autonomy to the coprocessor. Ouessant is integrated in the system in a classical way, meaning as a regular peripheral (usually on the communication bus). It is explicitly controlled from the outside, with configuration and start/stop commands issued by a central processor. This choice, while slightly increasing the overhead, greatly decreases the constraints on the resulting coprocessor.

III. Detailed hardware architecture

In this section, hardware architecture of the OCP is detailed.

A. General architecture

The general architecture of an Ouessantnt coprocessor is described in Figure 1. It is divided into 3 main parts, which represent the different abstraction levels used to integrate the accelerator.

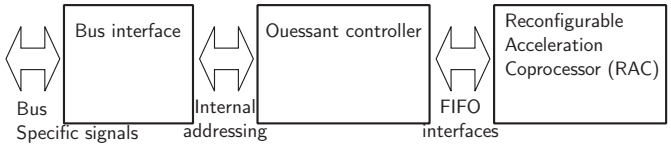

Figure 1: Global view of an Ouessantnt coprocessor

The Reconfigurable Acceleration Coprocessor (RAC) is the accelerator. It is user defined, and can be changed independantly from other components of the OCP. It uses FIFO-based communication, which is the easiest interfacing solution.

The next component is the controller. It is the central part of the OCP, providing instruction decoding and executing the instructions. Details on possible instructions and on the controller are provided further in the paper. The controller interacts with the RAC through the selected communication system (FIFO or memory access in future versions). It communicates with the complete system through the bus interface. The controller has no notion of the system type: it uses an internal address representation. The bus interface, which is the last component, is then used to translate the internal address representation in the required representation for the system.

B. RAC

The aim of Ouessant is to provide easy and flexible integration of dedicated accelerators in a System on Chip (SoC). In order to reach this goal, the Ouessant project provides variable width FIFOs, which can be used to interface with many accelerators. A complete RAC is described in Figure 2. These FIFOs are easy to use, with few signals to manage. They provide serializing and deserializing functionalities, and can thus serve as simple data formatting entities.

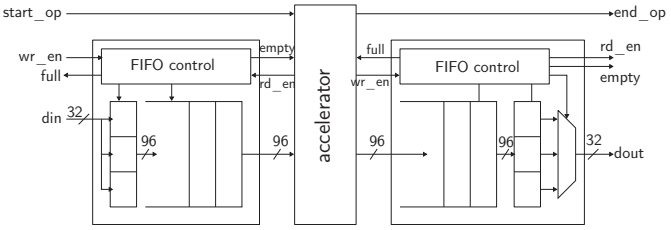

Figure 2: Ouessant RAC integration

The number of input and output interfaces can be adapted according to the accelerator requirements. For example, a dedicated configuration FIFO can be added if the accelerator requires additional configuration.

C. Interface

OCP interface is designed to translate Ouessant internal addressing mechanism to the SoC communication system. To ease the description, we make the hypothesis that the communication is managed through a bus. While this is a common configuration, it is not a requirement of Ouessant.

In the Ouessant approach, memory is divided in different banks. A memory bank is defined as a set of contiguous memory words. An internal address is a memory bank id with an offset inside this bank. This is a simple virtualization scheme, which is used to offer dynamic data management in Ouessant. Actual location of data is irrelevant when designing the coprocessor or writing the firmware. Banks location can then be configured at runtime.

The interface architecture is described in Figure 3. It is divided in two main parts: one which is dependant on the system bus, and one which is independant. The independant interface manages configuration and translates the internal bank/offset representation in a global address reprentation.

Configuration is stored on 10 registers. The first register is a control register. In the current version, only 3 bits are used, one for starting the coprocessor (bit S), one to enable interrupt (bit IE), and one to signal whether data processing is finished or not (bit D). The second register is the number of instructions in the program. The remaining registers are used to store memory banks location in the system.

The translation mechanism is quite simple. The controller sets a bank id and an offset when it requires data transfer. The interface selects the correct bank address in its configuration registers. It then adds the offset, in order to obtain the complete correct address in the system.

The system bus interface uses generic signals from the bus-independant interface and implements the real bus protocol. It must be implemented for each bus supported by Ouessant.

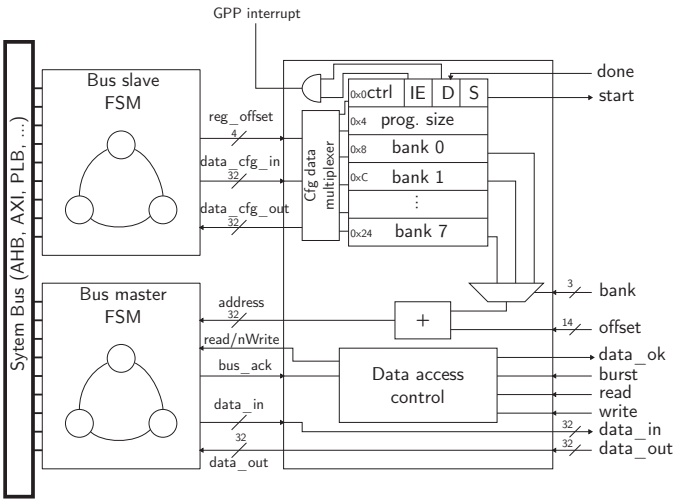

Figure 3: Ouessant interface architecture

D. Instruction set and controller

The instuction set for Ouessant is still a very simple and basic one. It will be extended in future versions. Operation code is stored on 5 bits, which allows up to 32 different instructions.

Currently, only 4 instructions are implemented, which can be divided in two different kinds:

- data transfers instructions, designed to copy data from/to memory. It provides burst transfer, and can be used as a simple Direct Memory Access component integrated in Ouessant. Two instructions are available: `mvtc` and `mvfc`, which respectively write and read data to and from coprocessor.
- execution management instructions. Two instructions can be used in this context: `exec`, which launches the coprocessor and waits for it to end, and `eop`, which signals the CPU.

Ouessant controller is responsible for instruction decoding and actual control of data transfer and coprocessor operations based on provided microcode. It is based on a classical unpipelined Fetch/Decode/Execute microcontroller architecture. It roughly consists of a Finite State Machine to control execution, and of registers to store the state it is in.

The resulting global Ouessant architecture is thus modular, and provide independant interfaces between each part. Interested users can thus focus on the part they are interested in, for example integration on a specific bus, without modifying anything else. Each part is optimized to be as efficient as possible, and to limit latency as much as possible.

IV. Software integration

One aim of Ouessant is to provide seamless hardware acceleration for end users. This means that it is necessary to provide efficient and easy to use software integration.

Integrating an hardware accelerator using an OCP in a software project requires very little modification. To illustrate the process, we describe how a Discrete Fourier Transform (DFT) operation would be used in a software program with different configurations. In a full-software implementation, the DFT is implemented in software, and called through a normal function call. The GPP executes the FFT, and the result is directly available in the output array.

```
// 64 words from offset 0 of bank 1
// to coprocessor FIFO 0
mvtc BANK1,0,DMA64,FIFO0
mvtc BANK1,64,DMA64,FIFO0
...
mvtc BANK1,448,DMA64,FIFO0
execs
mvfc BANK2,0,DMA64,FIFO0
mvfc BANK2,64,DMA64,FIFO0
...
mvfc BANK2,448,DMA64,FIFO0
eop
```

Figure 4: Example microcode for DFT execution

When using a typical bus-based hardware accelerator, the GPP sends data to the accelerator, waits for the computation, and then copies the results back to the `result` array. Data can be transferred through a Direct Memory Access (DMA) peripheral, however, the GPP is still responsible for scheduling transfers and launching operations.

In an Ouessantnt-accelerated application, the program configures the Ouessant, providing its parameters (pointers to arrays), lauches the computation and waits for the results. The OCP microcode is located in the memory and a very simple example is provided in Figure 4.

After the computation, the results are directly available in the output array. During computation, the GPP can process other tasks if required, as long as it does not involve data being processed by OCP.

The OCP is able to access memory, it is thus necessary to provide it with physical address. When no virtual memory is used, integration is quite easy. The only trick is to manage caches properly, which is often useless since current systems implement cache snooping.

Efficiently integrating Ouessant in a virtual-memory based environment such as Linux kernel is much more difficult. The strong isolation between kernel and user modes and the high overhead induced by the kernel can quickly decrease performance. In order to be able to benefit from Ouessant acceleration under a Linux system, it is necessary to provide an efficient driver. It is important to notice that this problem is true for any accelerator, not only for OCP. However, with OCP, the job will only need to be done once, and will then benefit all supported accelerators.

When studying Linux hardware integration, the main problem of drivers for high performance hardware lies in the kernel/user separation. In fact, each domain has its own virtual memory space, and it is not possible to directly access user space memory from kernel, or kernel space memory from user space applications. This is a very efficient memory protection mechanisms, but this means that data copies are required each time the user/kernel layer is crossed. Since data copies are performance killers, this is not acceptable in our case.

Several solutions are defined in the kernel to avoid these data copies [7]. In the Ouessant Linux driver, the `mmap` solution is used. This allows kernel space memory to be mapped in user space applications. Data is thus shared between both

domains. As will be seen in the Section V, this implementation allows performance gain even on small operations.

V. Implementation and results

A. Implementation status

The OCP described in this paper has been completely implemented, and integrated in a Leon3 [8] based system. The Leon3 is a Free Software implementation of the SPARCv8 architecture, available as a soft-core on many different FPGAs. It is based on an Amba2 bus. Linux integration has also been successfully used.

The OCP currently provides three main RACs. The first accelerator is a locally developped 2D Inverse Discrete Cosine Transform (IDCT) for JPEG decoding. The second one is the Spiral iterative DFT [9]. It can be configured to accept different DFT size, limited to the available FPGA size. In the following experiments, the previously described 256 points DFT was used. Any Spiral generated accelerator can be used.

Tests have been performed on a Digilent Nexys4 board, based on Xilinx Artix7 LX100T FPGA, with 16MB SRAM memory. Sytem clock frequency has been set to 50 MHz for all configurations, and no timing errors were left according to Xilinx tools.

B. Results

In order to validate the Ouessant approach and estimate its performances, the OCP overhead in FPGA resources has been computed. This has been done by synthesizing each accelerator alone, and with the OCP. Results have been obtained using the Xilinx synthesis tool, with the "Keep Hierarchy" option. This is not really an accurate measure, but it gives us a good idea of the hardware footprint.

Obtained results show that the actual OCP implementation consumes a reasonable amount of hardware resources (less than 1000 LUT and 750FF). This is for all OCP related parts: interface, controller and FIFO control. FIFO memory is inferred as BRAM, and strongly dependant on the accelerator. IDCT and DFT gives similar results except for the FIFO size and the RAC (actual accelerator size), which is independant from Ouessant.

Both RACs have then been used under a Linux system. Computing time has been measured through time markers in the software code. All the time results are summarized in Table I, and are given in cycles. For each available accelerator, the required number of cycles to process data has been computed (Lat. in the table). Data transfer time has not been considered in this estimation. The accelerator has then been integrated using an OCP in a Leon3 platform, and deployed on the FPGA. The computing time is given in the table in the HW column. A time-optimized software version (SW) for each operation has also been run for comparison. Finally, an acceleration factor (named Gain in the table) has been computed as the ratio between software execution time and hardware execution time.interrupt mode.

	Lat.	HW	SW	Gain
IDCT	18	3000	5000	1.67
DFT	2485	7000	600.10^3	85

Table I: Time results for OCP

We can see from the results that the overhead remains quite low. Given the fact that data must be transferred to and from the coprocessor, which is unavoidable, the Ouessant related overhead seems explainable. When running it without Linux, the DFT took 4000 cycles to compute, which gives an overhead of 3000 cycles coming from Linux. This comes from system calls. Given the computing time, we have roughly 1500 cycles needed for data transfer, and 1024 32-bits words to transfer. This means that around 1.5 cycles per word were required, which is quite a good result.

Finally, some results are hard to measure. Integrating the generated DFT took around 2 hours of work. The integration process was simplified since almost no control was required, and the result was easy to simulate, using the OCP. Since the OCP integration on the bus had already been validated, once it was functional in simulation, it worked on the board on the first try. This gain in integration time is non-negligible, at least when the original accelerator is adapted to a FIFO interface. Looking at the results in Table I, this means that an OCP is a good solution to benefit from hardware acceleration with little effort.

VI. Conclusion

In this paper, we presented Ouessant, a new coprocessor integration approach based on the Molen original idea. Ouessant provides easy, flexible, portable, and low-footprint integration of hardware accelerators in a SoC. It has been implemented and deployed on an Artix7 FPGA, with low resource and time overhead. Using the approach for single 2D-IDCT processing yields an acceleration factor of 1.67 in a Linux system. This factor rises up to 85 for 256 complex points DFT computation using the Spiral generated DFT core.

Ouessant is still under active development. Current work in progress includes complete Zynq (AXI4) integration, and Dynamic Partial Reconfiguration. Standalone operation is also studied, to provide control for processor-free designs. The instruction set is also being worked on, too provide higher flexibility. Finally, automatic generation of Ouessant interfaces for High-Level Synthesis of accelerators is under study. The Ouessant project is available as an open source project [1].

References

[1] "Ouessant Redmine server," https://redmine.telecom-bretagne.eu/ouessant.

[2] M. Vuletic, L. Pozzi, and P. Ienne, "Virtual memory window for application-specific reconfigurable coprocessors," *IEEE Transactions on VLSI Systems*, vol. 14, no. 8, pp. 910–915, 2006.

[3] D. Andrews, D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp, and P. Ashenden, "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *Micro, IEEE*, vol. 24, no. 4, pp. 42 – 53, jul. 2004.

[4] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A Computational Model for Reconfigurable Devices," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug 2006.

[5] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The MOLEN polymorphic processor," *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1363–1375, Nov 2004.

[6] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The Molen compiler for reconfigurable processors," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 1, 2007. [Online]. Available: http://doi.acm.org/10.1145/1210268.1210274

[7] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd edition*. O'Reilly, 2005.

[8] "Leon3," http://www.gaisler.com/index.php/products/processors/leon3.

[9] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.