GALS SoC Interconnect Bus for Wireless Sensor Network Processor Platforms

C. H. Fernández
School of Computer Science and Informatics
University College Dublin
Dublin 4, Ireland
carlos.hernandez@ucd.ie

Rajkumar K. Raval
School of Computer Science and Informatics
University College Dublin
Dublin 4, Ireland
rajkumar.raval@ucd.ie

C. J. Bleakley
School of Computer Science and Informatics
University College Dublin
Dublin 4, Ireland
chris.bleakley@ucd.ie

ABSTRACT

The purpose of this paper is to present a new System-on-Chip bus designed for the application specific requirements of Wireless Sensor Network motes.

The bus is designed to support Globally Asynchronous Locally Synchronous (GALS) systems. The bus is multi-rate with delay tolerance to support Dynamic Voltage and Frequency Scaled (DVFS) sub-systems.

Unlike traditional buses, the sub-systems operate as peers, rather than as master-slaves. Low power features include clock gating when inactive and burst transfers. The bus supports up to 255 interconnected resources.

Categories and Subject Descriptors:
B.5.0 Register-Transfer-Level Implementation: General
B.6.0 Logic Design: General

General Terms: Design.

Keywords: Wireless Sensor Network, WSN, System on Chip bus, SoC bus, low power, application specific bus, GALS..

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are comprised of large numbers of tiny battery powered motes which incorporate sensing, processing and wireless communications. Envisaged applications for WSNs include surveillance, precision agriculture, industrial plan monitoring and in-building energy management systems [**?**]. Current WSN nodes (or motes, as they are called because of their small size) are constructed from discrete off-the-shelf chips, typically including analog sensors, a microcontroller and an RF chip [**?**, **?**], as shown in Figure 1.

In order to reduce cost, the authors expect that, in time, these early prototypes will be replaced by System on Chip (SoC) solutions. The main remaining obstacle to wide spread introduction of these systems is to increase battery life from a few days, achieved using current motes, to 1-2 years.

GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy.

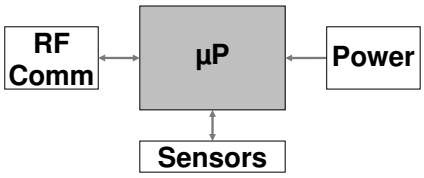

Figure 1: Centralized processing system

It has been estimated [**?**] that, for certain applications, the processor consumes up to 64% of the power of a mote during the execution of relatively simple applications. Presently, research is ongoing on WSN applications which require considerably more computational complexity and so greater processing power [**?**].

In order to achieve the battery lifetime targets, it is clear that radical approaches to reducing the power consumption of WSN mote processors are required. A number of research groups have proposed reducing the power consumption due to processing by utilizing hardware accelerators [**?**, **?**]. These accelerators perform regular tasks at much lower power consumption than would be obtained using a conventional processor, as shown in Figure 2. *So working with small and optimised processing units instead of using a conventional processor reduces the amount of energy consumed.*

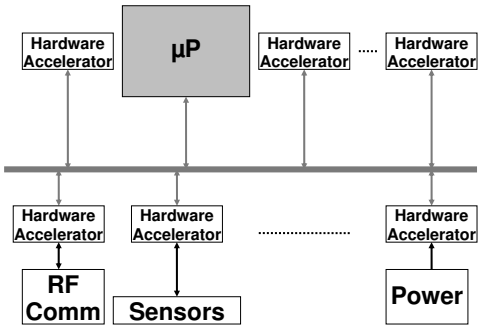

Figure 2: Distributed processing system including hardware accelerators

To maintain flexibility, a conventional processor is incorporated in the overall processor platform. In most cases, the processor and hardware accelerators communicate via a shared memory bus. We proposed that in order to minimize power consumption, it is desirable that:

1. Any Processing Units (PUs) (hardware accelerators or conventional processors) which are not in use should be supply gated.

2. All PUs should operate at the minimum voltage and clock frequency which allows the system to meet the requirements of the application.

It is clear that these requirements for an SoC, lead to a Globally Asynchronous Locally Synchronous (GALS) design methodology [?]. This approach has the advantage that PUs can operate at independent clock frequencies and switch on and off as needed. However, this approach requires a solution to the problem of providing low power, asynchronous, delay tolerant communication between the PUs within a SoC. It is this problem that the paper seeks to address.

The paper proposes a new design of SoC bus intended for use in GALS systems which allow for computational resources to be clocked asynchronously and to be powered down. The bus is designed to meet the key WSN mote requirements of low power, low bandwidth and low duty cycle.

The paper is structured as follows. In Section 2 previously published bus architectures and protocols are reviewed to assess their applicability to the problem. Section 3 describes the architecture, protocol and hardware implementation of the proposed SoC bus. Section 4 presents a description of such system implemented on an FPGA. In Section 5, ASIC implementation results are provided. Section 6 ends paper with the conclusions and the future work.

2. RELATED WORK

Many SoC buses, bus protocols and and IP blocks have been designed and implemented. Examples of SoC buses are *AMBA* [?], *Atlantic* [?], *Avalon* [?], *Wishbone* [?], *CoreConnect* [?, ?], *CoreFrame* and *Marble* [?], among others [?].

The requirements for the WSN SoC bus are as follows:

- Low power: bus active only when needed and low leakage power.
- Low data rate: Optimized for applications where data consists mainly of interrupts, configuration and signalling data.
- Simple layout: One bidirectional 8-bit data bus.
- GALS suppport: clock independent of PUs, selectable clock rate for each transmission.
- Peer-to-peer Logical multi-point without Master/Slave scheme.
- Delay tolerant: support for asleep/unavailable resources.
- Reusable: IP blocks with simple interface to PU logic.

On review, it was found that no previously defined bus meets these requirements. Following is a brief analysis of the problems associated with the use of existing buses. Not all of the drawbacks are listed, just the most important:

- *AMBA (AXI)*: separate buses for address/data, fixed rate synchronous bus.
- *Atlantic*: fixed rate synchronous Master/Slave bus.
- *Avalon*: separated buses, fixed rate synchronous and Master/Slave bus.
- *Wishbone*: separate buses and Master/Slave design.
- *CoreConnect (PBL)*: 32-64 bits data width, fixed rate synchronous bus, Master/Slave design.
- *CoreFrame*: unidirectional separated buses.
- *Marble*: separated buses, fully asynchronous design with handshaking protocol.

In summary, the proposed bus provides significant advantages in terms of functionality over previous buses for GALS systems. In addition, by facilitating a delay tolerant design, the bus allows for significant power reductions in the Processing Units.

3. PROPOSED SOC BUS

3.1 Architecture

Figure 3 shows the overall architecture of the proposed SoC bus.

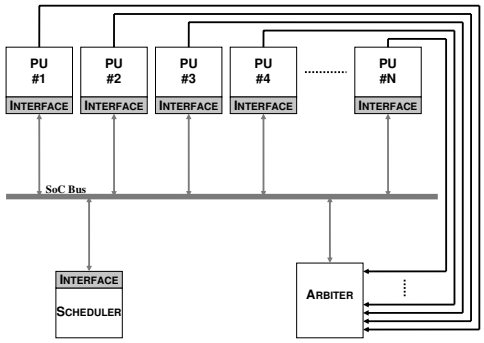

Figure 3: Layout of the system and connections to the bus

Each PU incorporates an IP block which interfaces with the bus. The bus and interface block are synchronised by a clock generated by the arbiter, whereas the main PU has an independent clock. Thus, communication is not tied to the clock used in any one PU. This provides flexibility and makes this bus suitable for use in GALS systems.

Because of the low duty cycle of WSN PU traffic transmitted along the bus, there is a low percentage of utilisation. Hence there is typically no need to segment the bus into smaller sections, although this can be done.

Transmission requests are signaled to the Arbiter via a dedicated line per PU. In the case of a large number of PUs, encoders and decoders may be used to reduce the number of lines. The remainder of the signals are shared between all of the resources. The complete list of signals is shown in Table 1.

3.2 Bus Protocol

The protocol is based on a bus request - grant scheme:

1. The resource requests the bus setting its `bus_request` line active high.

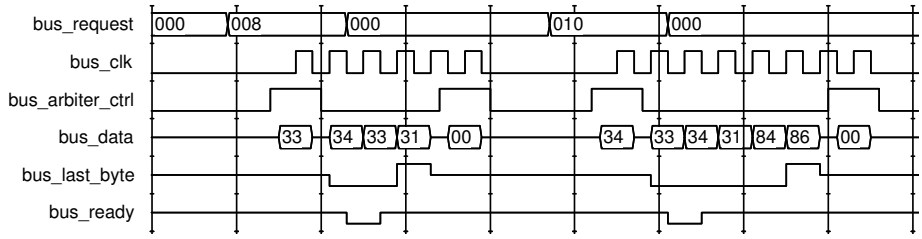

Figure 4: Simple bus transmissions

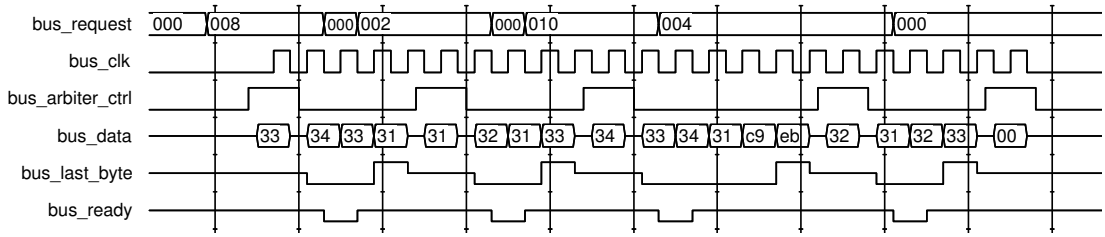

Figure 5: Concatenated bus transmissions

Table 1: Complete list of signals in the proposed bus

Name	Width	Function
`bus_request`	1 x n	Request line from the PU enquiring for transmission to the Arbiter. One per PU
`bus_clk`	1	Independent clock line controlled by the Arbiter
`bus_arbiter_ctrl`	1	Special line used by the arbiter to control the bus.
`bus_data`	8	Data bus. There is only one 8-bit data line to signal the data and addresses.
`bus_ready`	1	Line controlled by the receiver to indicate whether it has been possible to receive the message.
`bus_last_byte`	1	Line controlled by the sender to indicate the last byte of the transmission.

2. Using a pre-programmed priority table (or any other selection method) the Arbiter decides, among all the resources requesting the bus, which will be the next to transmit.

3. The Arbiter sets the `bus_arbiter_ctrl` line to initiate the transmission.

4. The Arbiter sends the identifier of the granted PU on the bus, and starts the clock. The resources requesting the bus have to be awake and have to read the granted identifier in the first rising edge of the clock.

5. The resource whose identifier matches the one sent by the Arbiter, takes control of the bus on the second rising edge of the clock. At that moment it sends the destination identifier. On the following positive edges of the clock, it sends the data until the last byte when it sets the `bus_last_byte` and it yields control of the bus to the Arbiter.

6. The arbiter regains control of the bus, setting `bus_arbiter_ctrl` line high.

7. If there are no other PUs requesting permission to transmit, the Arbiter sends the address `00h` to finish the transmission and stops the clock. Otherwise, the protocol returns to step 2.

Figures 4 and 5 show two examples of transmission. In Figure 4 two simple transactions are performed; the PU with ID number `33h` sends the message `'33h, 31h'` to the PU ID `34h` and the bus is put back in idle state (Arbiter sends `00h` address) until PU number `34h` responds to PU `33h` with message `'34h, 32h, 84h, 86h'`.

In Figure 5, four consecutive transmissions are executed by different PUs. During a transmission it is possible that another PU is waiting to transmit, so in this case the Arbiter can grant access to the next PU without setting the bus to idle state, therefore saving time and power.

3.3 Hardware Blocks

The bus comprises three main hardware blocks: the Arbiter, the Scheduler and the Interfaces. The Arbiter controls the bus, the Scheduler is used to store messages missed by asleep or busy PUs and Interfaces connect PUs to the bus.

3.3.1 Interface block

The Interface is divided in two separate blocks - reception and transmission. Each is used independently and has different resources, but they share the same pu identifier (ID).

Transmit block.

The transmit block of the interface is shown in Figure 6.

The data to be transmitted is stored in the PU and read asynchronously by the transmit block, that allows both parts to use different clocks and therefore be completely clock independent.

The transmit block is controlled by the PU with the interaction shown in Figure 7. When the PU requests that the transmit block send a message, the block requests a bus grant from the Arbiter using the dedicated line.

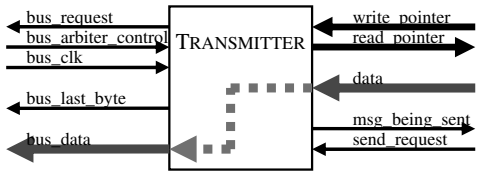

Figure 6: Transmit block of the interface

Once the Arbiter grants permission, the transmit block starts sending the message over the bus, indicating this to the PU by setting `message_being_sent`. The PU clears the request signal as soon as the transmission has begun. The transmission finishes when the last byte is sent, which is indicated with the `bus_last_byte` signal. At this moment, the transmitter clears `message_being_sent` to indicate that it is ready to send a new message.

The signals used by this block are listed in Table 2.

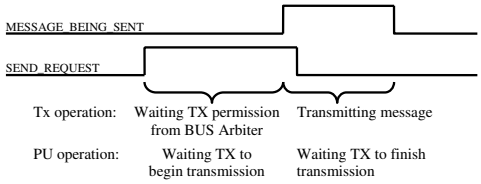

Figure 7: Handshaking protocol between TX block and PU

Table 2: List of signals of TX interface

Name	Width	Function
`write_pointer`	q	The write pointer signals the number of bytes stored in the memory (in this case, in the PU).
`read_pointer`	q	The read pointer is controlled by the transmitter, and it is used to read the memory asynchronously.
`data`	8	Data to transmit.
`message_being_sent`	1	Set when the transmission has been started.
`send_request`	1	Used by the PU to point the interface that there is a new message ready to transmit.

The width of the pointers, q, depends on the maximum length of the messages that the designer estimates will be transmitted. Pointers are used to read the data from a memory located in the PU, so their size is associated with the size of the memory.

Receive block.

Figure 8 depicts the architecture of the reception block and the signals involved:

The receive block performs very similarly to the transmit block, using handshaking (Figure 9) to assure clock independence. Active receive blocks always check the ID of the destination PU in the message that is being transmitted over the bus. If the destination identifier matches the block's ID, the block copies all of the contents of the message to its internal memory. Once the last byte of the transmission has been received, the block indicates to the PU that it has a valid message stored in the memory. The receiver remains in a busy state until the PU clears the receiver.

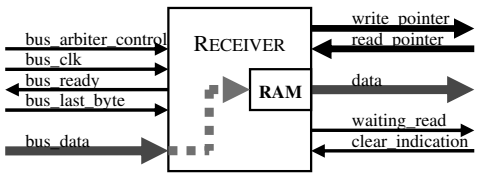

Figure 8: Receive block of the interface

The signals used to communicate between the Interface block and the PU are listed in Table 3.

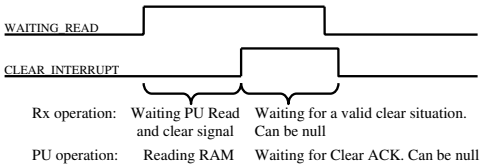

Figure 9: Handshaking protocol between RX block and PU

Table 3: List of signals of RX interface

Name	Width	Function
`write_pointer`	p	The write pointer signals the number of bytes stored in the memory (inside the receiver block)
`read_pointer`	p	The read pointer is controlled by the PU, and it is used to read the memory asynchronously.
`data`	8	Data received.
`waiting_read`	1	This signal is active when a complete message has been received.
`clear_indication`	1	Used by the PU to point the interface that the message has been reading and it is possible to receive another message.

The amount of RAM implemented in the receive block is decided by the designer, as it depends on the maximum length of the messages to receive. Because the memory is accessed via `write_pointer` and `read_pointer`, their width, p, must be enough to address the whole memory space.

3.3.2 Bus arbiter

The Arbiter is the block which controls bus operations. It selects which PU transmits at any time, with priority based on a pre-programmed look-up table. In the case of contention, a single PU is not granted bus control on consecutive requests. Other strategies can be applied to this

bus (i.e. lottery, priority, FIFO, etc.) at the designer's discretion.

The Arbiter also supplies the clock which synchronises the transmission. The clock rate is determined by look-up of a pre-programmed table in the Arbiter. Clock rate is expressed as an integer fraction of the main system clock, and is set according to the DVFS status of the PUs involved in the communication.

In case one of the PUs does not release the bus the Arbiter has the ability to regain control of the bus. This could happen in case the PU exceeds the maximum length allowed by transaction, specified by the designer, or it is switched off before it finishes it.

3.3.3 Scheduler

One important feature of the bus is the capacity to deal with busy or asleep PUs.

In the case that the receiving PU does not respond, the Scheduler receives the transmission, stores it and re-transmits it when the receiver is available. In the case that the intended receiving PU is asleep, the Scheduler also initiates PU wake up via a voltage control PU. The advantages of this scheme are two fold. Firstly, delay tolerance for all PUs is centralized in a single block reducing area overhead in the PU bus interface blocks. Secondly, the bus is free during the wake up or busy time of the receiving PU which may be hundreds of milliseconds.

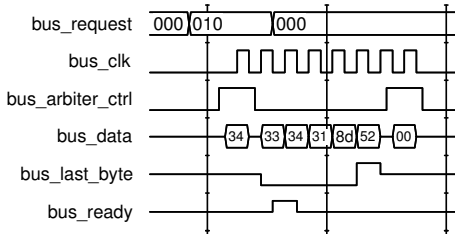

Figure 10: Transmission with unavailable receptor.

In the case that a receiver is busy and cannot receive messages, bus_ready is set high. For asleep Interfaces, the line bus_ready remains in the high impedance state. In both cases, the Scheduler will intervene and receive the transmission. The scheduler will store the data and resend it when the PU is available.

4. FPGA IMPLEMENTATION

In order to test the bus, the IP blocks and the protocol were implemented in Verilog and tested on motes designed by the Tyndall Institute [**?**], which incorporate Xilinx Spartan II-E FPGAs. The verification environment was as shown in Figure 11.

The functionality of the system was as follows:

- LEDS: controls the LEDs. LEDs are controlled by an internal variable. Every time the variable is changed (due to a message send to this block) the LEDs change their state. This block can also transmit the value of the LEDs when queried. *The goal of this PU is not to offload an hypothetical processor but to test multi-source communications towards one PU, using each of the 3 different LEDs to visually acknowledge messages received by this PU.*

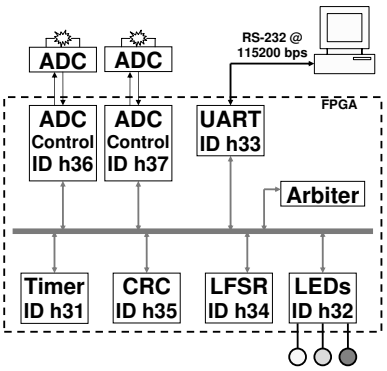

Figure 11: Test system

- Timer: sends a pre-defined message to the LEDs block every 0.5 seconds.
- CRC: performs a cyclic redundancy check on the message received ('CRC-16/CITT').
- LFSR: generates a random number and transmits it.
- ADC Control: *Tyndall motes are equipped with 2 one-line serial ADCs, so a communication protocol is needed to recover the data. The protocol is micro-programmed in the ADC control block instead of software-programmed.* When a query message is received the block reads the ADC input and returns the measurement to the querying PU.
- UART: connected to a PC. Exchanges messages between the bus and a terminal window.

First, the system was synthesized and tested using the Xilinx ISE environment [**?**] and simulated with ModelSim [**?**]. Complex tests, such as asynchronous transactions, different clock rates, simultaneous bus requests and communication with asleep or busy PUs were performed in simulation.

Afterwards the system was implemented in the FPGA and tested by using the Timer PU to toggle the LEDs on and off automatically. At the same time, test messages to the PUs were send manually from the PC via the UART and a terminal window, in order to execute the tasks supported by the LEDs, LFSR, ADC and CRC blocks and to read the parameters of the PUs.

More intensive tests were written in C. These tests were performed in the same way as the manual tests but were executed repeatedly over a long time, generating reports on the performance of the system.

Once the functionality of the system was successfully verified, an ASIC synthesis and simulation was carried out.

5. ASIC IMPLEMENTATION

The complete SOC interconnect system was synthesised using Synopsys Design Compiler [**?**] using the technology libraries provided by VTVT Group [**?**]. The design was simulated at the gate level using Modelsim [**?**]. Synopsys PrimePower [**?**] was used to estimate the power and energy consumption of the system. TCL Scripts were developed to run the EDA tools and generate reports.

The system consisted of two to eight PUs and an Arbiter. Bus Functional Models of PU functionality were implemented to generate transactions on the bus. The gate

level netlist of the complete system was simulated with SDF back annotation, generating the VCD file of the simulation to provide the circuit switching information. Along with the the VCD file, wire load model was provided to PrimePower to provide an estimation of the energy consumed.

Figure 12 shows relative percentages of power figures compared to the number of PUs. *The purpose of this figure is to show how the power used by the bus in idle state is much lower than in active state, increasing the difference with the number of PUs and remaining the idle power consumption almost constant, due to the clock stopping scheme depicted in figures 4 and 5. Because the fact that most of the time the system is in asleep state, this is considered to be a major advantage in terms of energy savings.*

The small difference between idle and leakage power is due to Arbiter, whose clock is always active. Most of the increase in active power compared to the number of PUs is due to the activity of the PUs that are not involved in message communication, because these PUs have to check the message identifier before returning to idle state.

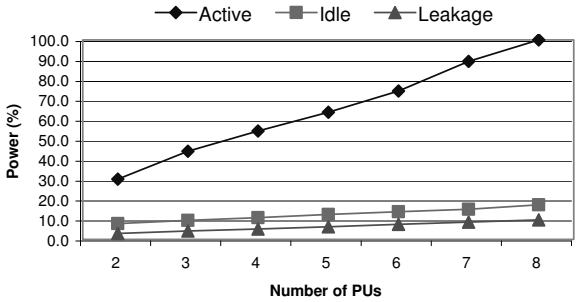

Figure 12: Power comparison; active, idle and leakage

The area of the synthesized blocks are given in Table 4. Values are referred to VTVT Group standard cell libraries [**?**].

Table 4: Area used for each synthesized module

Module	Cells	Area units
Arbiter	162	18942.33
Interface	222	31597.77
RX block	121	17227.73
TX block	101	14239.41

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a new System on Chip interconnection bus, specifically designed for GALS systems. The architecture and functionality of the bus were implemented and verified on an FPGA. The system was synthesised and simulated for ASIC to obtain power figures and area reports.

Extended power measurements and direct comparisons with other buses are planned once the general architecture of the platform is completed. Future work is focused in the application of the interconnection bus to the Wireless Sensor Network SoC we are currently developing.

7. ACKNOWLEDGMENTS

This work is supported by a research grant from Enterprise Ireland. The authors would like to thank Tyndall National Institute for their support and materials provided.

8. REFERENCES

[1] Altera Corporation. Atlantic interface. 2002.
[2] Altera Corporation. Avalon interface specification. 2005.
[3] J. Andersen. Wireless sensor networks and pervasive computing presentation. *University of Aarhus*, 2005.
[4] ARM Limited. AMBA AXI Protocol v1.0 Specification. 2003, 2004.
[5] S. Bainbridge, W.J.; Furber. Asynchronous macrocell interconnect using marble. *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–132, 1998.
[6] D. M. Chapiro. Globally-asynchronous locally-synchronous systems. *Ph.D. Thesis*, oct 1984.
[7] IBM. The coreconnect bus architecture. 1999.
[8] A. W. S. H. D. C. K. P. Jason Hill, Robert Szewczyk. System architecture directions for networked sensors. *ACM SIGOPS Operating Systems Review archive*, 34(5), 2000.
[9] R. K. Jason Hill, Mike Horton and L. Krishnamurthy. The platforms enabling wireless sensor network. *Communications of the ACM*, 47(6), 2004.
[10] M. Kohvakka, T. Arpinen, M. Hännikäinen, and T. D. Hämäläinen. High-performance multi-radio WSN platform. *REALMAN '06: Proceedings of the second international workshop on Multi-hop ad hoc networks: from theory to reality*, 1-59593-360-3:95–97, 2006.
[11] Mentor Graphics Corporation. www.mentorgraphics.com.
[12] W. D. Peterson. Wishbone system-on-chip (soc) interconnection architecture for portable ip cores. *OpenCores.org*, 2002.
[13] R. C. D. Polastre, J.; Szewczyk. Telos: enabling ultra-low power wireless research. *Fourth International Symposium on Information Processing in Sensor Networks*, 2005.
[14] B. D. Rick Hofmann. Next generation coreconnect processor local bus architecture. *Annual IEEE International ASIC/SOC Conference*, 2002.
[15] V. K. K. H. T. Salminen, E. Lahtinen. Overview of bus-based system-on-chip interconnections. *IEEE ISCAS International Symposium on Circuits and Systems*, vol.2:372 – 375, 2002.
[16] Synopsys, Inc. www.synopsys.com.
[17] Tyndall National Institute. www.tyndall.ie.
[18] R. W. A. N. P. J. Vijay Raghunathan, Trevor Pering. Experience with a low power wireless mobile computing platform. *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISLPED04)*.
[19] VTVT Group. http://www.ee.vt.edu/~ha/cell_library/distribution.html.
[20] Xilinx ISE Foundation design tools. www.xilinx.com.