Reconfigurable High-speed Asynchronous I/O Ports for Flexible Protocol Support

Suleiman Abu Kharmeh[†‡] and Simon Hollis[†]
[†]Department of Computer Science, University of Bristol; [‡]XMOS Semiconductor Ltd.
{kharmeh,simon}@cs.bris.ac.uk

***Abstract*—Many I/O protocols have elements in common, which may be exploited and shared between multiple on-chip protocol implementations, to yield area and energy savings. However, current implementations of I/O functionality in chips place each protocol in its own dedicated block, and so this potential advantage is never gained.**

We highlight some elements that may be usefully shared between protocols and explain how a flexible, reconfigurable GCU may be constructed out of functional blocks using these elements. An overview of the structure of the generator is given, and it is shown that implementation in asynchronous logic yields significant advantages for the design over a conventional synchronous implementation. Finally, we show how the use of asynchronous FIFOs greatly simplifies the design of the system and allows simultaneous support of a number of protocols via the time-sliced re-use of the GCU fabric.

I. Introduction

Throughout the past few years, while digital processing systems have been converging towards the parallel domain, digital communication systems on the other hand have been converging towards the serial domain. For example the Parallel Advanced Technology Attachment (ATA) - Previously IDE- has been replaced with the Serial ATA, USB Transceiver Macrocell Interface (UTMI) has been replaced by the UTMI+ Low Pin Interface (ULPI) and Serial Gigabit Media Independent Interface (GMII) is the serial counterpart of the original GMII specification and so on [1], [2], [3]. While in the processing domain, software on different levels of abstractions from high level programming languages down towards the hardware instructions and microcode offer a high-level of programability and flexibility, there has been little focus on offering the same level of flexibility for digital communication systems. A quick review of any state-of-the-art digital system would reveal a high number of communication standards and protocols used to interface between the different building blocks of the system. These range in speed, bus width, synchronisation mechanism, and protocol complexity. The most common approach in implementing such protocols in hardware, especially for the complex and high speed ones, is through the integration of a hardware controller. Referring to the OSI reference model, those controllers typically implement the Link Layer Control (LLC) and/or Media Access Control (MAC) functions (see Fig. 1) [4]. There is a basic set of functions that such controllers are required to implement. Regardless of the inherent similarities the OSI model impose on the communication standards, systems architects still approach each protocol separately through the development of a separate controller to interface to the physical Layer [4], [5].

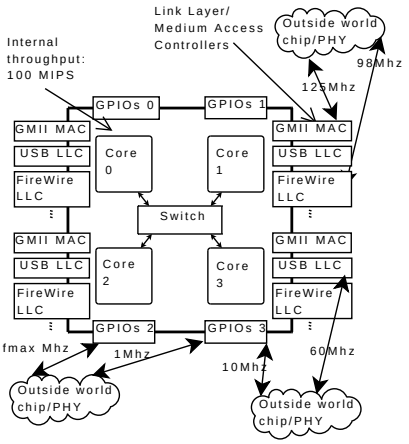

Fig. 1. Multi-core chip with multiple high-speed serial protocol controllers

Throughout this paper, we will present a highly programmable, high speed communication unit. Such unit will be integral to an embedded processing unit, and will provide the much needed flexibility in the I/O system, while still maintaining high data rates to support these protocols. While general purpose central processing units (CPU) have been evolving for decades, the industry is failing to provide a general purpose communication unit (GCU) for high speed interfaces. A GCU must handle signal rates comparable to the processing speeds of the internal core, while providing well defined standard interfaces to software.

II. Related work

In recent years, communication protocol interfaces to the higher level software have been provided through a few different approaches. The most commonly used approach is the on-chip integration of a well defined hardware module into the chip to perform the specific functions required for the desired protocol. The integration of a separate dedicated Media Access Controller or Link Layer Controller for each of the desired protocols is very popular amongst both Application Specific Integrated Circuits (ASIC) and Field Programmable Gate Arrays (FPGA) designers, and by definition is able to support a multitude of data rates and virtually any communication protocol. The disadvantages of such approach is that a separate LLC and/or MAC controller need to be integrated on chip

for each protocol the system uses, and this approach does not provide much flexibility in the implementation the desired protocols (a minor protocol specification change would require a system redesign and a chip re-spin). This lack of flexibility also does not help the evaluation and prototyping of any other communication protocol for future system development. Another approach for the simpler interfaces with lower throughput in the FPGA domain is to provide a soft implementation [6] of the desired protocol functionality which can be programmed into the FPGA at run time. While this approach may seem flexible to adapt to protocol changes and able to support multiple protocols, it is very costly in terms of silicon area and consequently power consumption. Also, the maximum signaling rates that such approach can handle is normally limited due to the inherent limitation on system operating frequencies compared to the equivalent ASIC [7].

Finally, and as a half way point between dedicated hardware and a standalone software implementation, a hybrid model is currently emerging in some systems [8] where basic general purpose hardware modules are introduces to perform low level protocol functions such as serialisation/de-serialisation of the data stream, and generation/verification of the Error Correction Codes (ECC) for the data packets, while leaving higher level data processing and control response to the ISA software layer.

This paper presents a hybrid style General purpose high speed Communication Unit (GCU) that is programmable at run-time and able to support a multitude of communication standards. This will provide the much needed flexibility for any digital communication system.

III. Case studies

In the following sections, we will review some of the most common communication standards in use today. This is done in order to provide a solid insight on the similarities between the standards and therefore form the foundation of our GCU.

Starting with the Ethernet networking standard, the Medium Independent Interface (MII) is used to connect between the MAC controller and the Physical Layer. The MII signals to the MAC includes a clock (30MHz for 100BASE-TX), receive and transmit data buses accompanied with data valid and various other control signals. Frames consist of preamble, a Start-of-Frame (SOF), the addresses of the frame's source and destination, a length or type field followed by the client data, the error correction code -Cyclic Redundancy Check (CRC)- and finally an extension field if required [3]

Next we give a quick evaluation of the USB Transceiver and Macrocell Tester Interface (UTMI) specification. This interface operates at a maximum clock frequency of 60MHz and the data buses are 8-16 parallel lines. Also packet and word framing signals are provided to the LLC [2]. The UTMI control interface provides hardware support for low power modes where the LLC shuts all blocks not necessary for resume operation. The UTMI frame consists of a synchronisation word (similar to the preamble in the Ethernet above) followed by packet ID, the data byes, CRC (up to 2 bytes) and finally the end of packet identifier. The USB standard enforces strict timing deadlines on the response to packets. Those deadlines must be carefully taken in consideration when designing our GCU. Triggered by the recent evolution in serial communication, and the package costs associated with the parallel interface (64 to 80 pins) the UTMI specification has been updated to accommodate for an extremely high speed serial version called UTMI+ Low Pin count Interface (ULPI) [2]. In this update, the separate parallel data buses were replaced with a single 4 or 8 bit bidirectional bus of differential lines for both the send and receive. This dropped the link interface to 8 or 12 signals, which consequently dropped the package size as small as 32 pins or less. It still enforces the hard deadline requirements of the USB2.0 specifications. It is claimed that the ULPI specification can be used for any Link interface including USB, Ethernet and Wireless protocols [2]. It also adds a bus direction switch time of one USB clock cycle.

Protocol /Property	Max Clock Freq. (MHz)	Packet length (bytes)	CRC (bits)	Data Throughput (Mbps)
MII	30	1522	32	100
USB	60	1024	5-16	480
GMII	125	1522	32	1000
SATA	300	8192	32	3000

TABLE I
LINK LAYER PARAMETERS FOR SELECTED COMMUNICATION PROTOCOLS

Finally, we review the Serial Advanced Technology Attachment (SATA) standard. The SATA has high data rates of up to 3Gbit/s. Link Layer clock has a maximum of 300MHz. The SATA frame format is very similar to the Ethernet frame in that it contains a SOF, payload, a 32bit CRC and finally End-of-Frame [1].

We have noticed that the above protocols have common elements such as CRC, serialisation/de-serialisation, similar data thruoghputs and clock frequencies... etc. Such commonalities we believe can be exploited to produce a reconfigurable GCU to support multiple protocols with minimum custom logic design. We explore this concept in the remainder of this paper.

IV. Exploiting commonality between protocols

Noting the commonality between various protocols, as outlined in the previous section, we can produce a general framework (a GCU) for supporting multiple protocols without the need for dedicated hardware blocks for each. Key to our proposed approach is the observation that many commonly-used protocols can be decomposed into a sequence of basic operations. Each operation can then be implemented individually and parametrised to make them usable across a range of protocols. By means of illustration, consider the basic Ethernet protocol [9]. The basic data transmission operation can be outlined as the following sequence of operations:

1) Create a packet of the correct length in an *output buffer*;
2) *Generate the header*, including a fixed *preamble*;
3) Insert the data into the buffer;
4) *Generate a check-sum*;
5) *Serialise the data stream* from the output buffer onto the *I/O pins*.

In the sequence above, we have highlighted in italics where we believe there are basic operations that could be shared between multiple protocols. A similar sequence can be drawn up for other protocols of interest, although many will be more complicated and require more steps.

We propose that these basic operations could be implemented in dedicated hardware blocks that may be shared across a range of protocols that require these components, and call them *F-blocks*, for "Functionality Block". Some addition examples of potential F-block in addition to those outlined above are ECCs, acknowledgment generation, input buffering, hop counters, rotation and permutation blocks, and even the use of a programmable lookup-table or Flash memory to implement arbitrary functionality. Obviously, the specific requirements for the F-blocks will vary between protocols. For example, whilst many protocols use error-detecting codes (such as CRCs), the length and algorithm used will be different (refer to Section III for examples). Therefore, F-blocks must be *parametrisable*. An interesting avenue for future investigation is to what extent this parametrised need be performed — will parametrising over CRC lengths be sufficient, for example, or would an CRC block need to be programmable to support a wide range of codes. In particular, Amdahl's Law [10] states that there is no point though, in implementing functionality in dedicated hardware blocks if it is not going to be used across the majority of protocols used in practice. Doing so would just use silicon that is idle (and potentially leaking power) the majority of the time, and give a net decrease in overall efficiency. Therefore we need to be sure that the addition of an F-block is a good idea. We propose evaluating the various protocols likely to be deployed both currently and in future I/O functionality, and determining exactly which elements should be dealt with by an F-block, and which are best implemented in software, or by a general-purpose co-processor. Similarly, the various functions will have different performance characteristics, and to addition of one particular component might adversely affect overall performance. When could this occur, and how is it best to deal with it? We hope to answer these questions with our future work.

V. Implementation of F-blocks

Our proposal is that F-blocks be dynamically combined to produce a sequence of logic capable of completely synthesising a protocol. We further propose that F-blocks be re-used between multiple protocols sinking data into multiple output buffers, one per I/O port by dynamically reconfiguring their interconnection (see Fig. 2). To guarantee the data rates and quality of service necessary to simultaneously support multiple protocol streams, we propose that F-blocks and their interconnections be *time-sliced* between the various I/O ports.

A. Composition of F-blocks

Since the goal of our system is that F-blocks may be combined to produce an arbitrarily complex path, the delay of the protocol path will be unique to every configuration. Under these circumstances, design-time analysis of compositional-correctness will be problematic. In a synchronous design, we must verify that inter-block timing requirements are met for every possible combination of F-block chain. Even with only a dozen F-blocks in the system, there are approximately $12\text{e}! = 1.3\times10^9$ possible combinations. This size of verification problem is clearly intractable. The use of an asynchronous design methodology, however, would allow this composability problem to be side-stepped. We propose to prove functional correctness for each individual F-block, implementing a *delay-insensitive encoding* [11]. The properties of this encoding guarantees that functionalites of the combination of several blocks will also be correct if the individual blocks are correct, and so the verification problem is solved.

Performance will, of course vary between protocol setups, and so we propose that timing analysis be performed at design time on the F-blocks and a worst case value ascertained. Logic in charge of composing the blocks (the scheduler we will present shortly) can then determine what the total path delay will be by considering the delays along the path. For a synchronous design, where all blocks share a common clock, every F-block enabled on a path would have to run at the maximum frequency of the slowest block f_{min}, and so the total path latency of n block would be $n \times (1/f_{min})$. However, asynchronous logic runs at the actual case speed for all components, so a path consisting of n blocks (labelled $i = 1..n$) each with worst-case frequencies f_i takes only $\sum_{i=1}^{n}(1/f_i)$ seconds; likely to be significantly less than for an equivalent synchronous design. Therefore, the use of asynchronous logic to implement our F-blocks gives us not only flexibility and tractability of implementation, but increased performance over a more synchronous design strategy.

B. Time-slicing

Since our aim is to greatly reduce the area overhead of supporting multiple protocols on a chip, we wish to re-use as much functionality as possible. Our choice of a composition of F-blocks allows us to produce an arbitrary protocol, but is more wasteful than a full custom implementation if only one I/O port uses it at once. Therefore, we introduce the concept of *time-slicing* the protocol generation fabric.

To support this, we include a scheduling unit, which controls the multiplexing (MUXing) and demultiplexing (DEMUXing) of data streams between the GCU and the various I/O ports sharing its functionality. It is also responsible for configuring the interconnection between F-blocks (i.e., configuring the GCU). It is anticipated that the GCU will be configured several times a milli-second.

To provide the throughput of multiple I/O ports, the GCU must run a at least their sum of throughputs. The choice of asynchronous logic for the F-blocks means that they will run as quickly as possible, independent of individual I/O port data rates. Therefore, the maximum possible number of I/O ports per F-block can be mapped. In between scheduled runs, the individual I/O ports are still expected to output data at a rate dictated by their particular protocol, and to support this, we

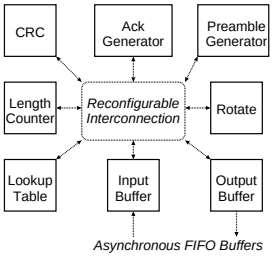

Fig. 2. F-blocks with their reconfigurable interconnection, creating a full GCU

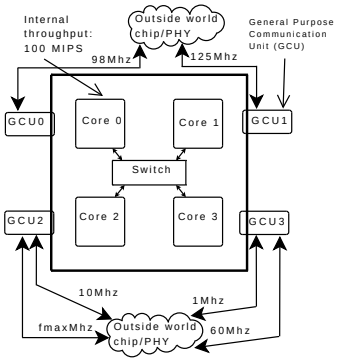

Fig. 3. Multi-core chip with multiple high-speed serial protocol controllers

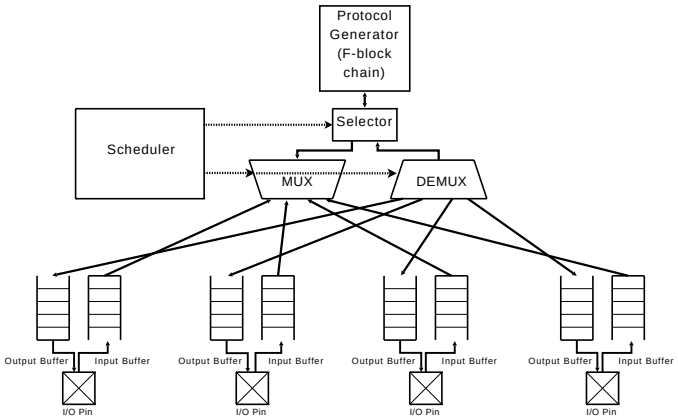

Fig. 4. Asynchronous FIFOs with multiple I/O pins allows GCU time-slicing

insert asynchronous buffers between them and the GCU, to buffer data for the inter-time-slice time.

C. Use of asynchronous FIFOs

As explained above, the data rate coming out of the GCU is likely to be much greater than that actually required by the I/O ports. Each I/O may also require a different data rate to all others (they all may run different protocols). Finally, the use of time-slicing means that data being sent to an I/O port from the protocol fabric will be bursty. This gives rise to the need for input/output buffering at the ports, and a demanding set of timing requirements on them. I/O buffers will have to tolerate very high frequency data rates on the protocol-generator side, a wide range from slow to very high (e.g. 1Mbit/s for USB1.1 to 1Gbit/s for Gigabit Ethernet) on the pin-side; all of which may have completely asynchronous clocking regimes.

Conventional FIFOs would have great difficulty in dealing with this range of constraints, and so we propose to use an asynchronous flavour of FIFO to perform buffering. Asynchronous FIFOs are well know for this kind of application [12], and are ideal for decoupling the timing regimes and various data rates. In particular, they can operate over this large, dynamic range of data rates without suffering the large synchronisation penalties and loss of in-step operation brought on by a synchronous design. Therefore, they give great performance and complexity advantages in our design.

VI. Conclusion

We have seen that there are many I/O protocols used by contemporary ASICs in interfacing with outside components. Their data rate requirements range from in the order of 10Mbit/s to several Gbit/s, potentially connected to the same set of ports. Multiple protocols are typically supported simultaneously by the implementation of a set of dedicated blocks inside a chip. However, this leads to inefficiencies since many of these are typically idle for a significant fraction of the operating life of the device.

We have presented a mechanism where this overhead may be removed, though, by the implementation of a reconfigurable GCU capable of implementing a wide range of protocols from basis building blocks, which we have identified as common to a wide range of contemporary protocols. Due to their dynamic configuration, and arbitrary arrangement, we have seen that a conventional synchronous design would pose intractable block-composition problems, and so implementation using a delay-insensitive logic style is preferable. High-speed operation and time-slicing of the fabric to support multiple protocols, connected to multiple I/O pins at once is made possible by the useful properties of asynchronous FIFOs in interfacing between wildly-varying data rates, and unpredictable and bursty traffic.

Overall, the proposed system is able to support all the protocols considered, should be capable of running reliably at high speed, and provides much-needed space efficiency gains for future generations of ASIC.

References

[1] Information Technology, "At attachment with packet interface - 7 volume 3 - serial transport protocols and physical interconnect (ata/atapi-7 v3)," Online, March 2004.

[2] ——, "Introduction to the utmi + low pin interface (ulpi)," Online, March 2004.

[3] ——, "Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications," Online, 2000. [Online]. Available: http://ieeexplore.ieee.org/iel5/7057/19017/00879000.pdf

[4] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, pp. 425 – 432, April 1980. [Online]. Available: http://ieeexplore.ieee.org/iel5/26/23925/01094702.pdf

[5] xilinx, "Virtex-5 family overview," Online, June 2008. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[6] L. Yuan, P. Pari, and Q. Gang, *Information Hiding - Soft IP Protection: Watermarking HDL Codes*. Springer Berlin / Heidelberg, 2005, vol. 3200/2005.

[7] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 12, no. 8, pp. 203 – 215, February 2007.

[8] E. Horta, J. Lockwood, D. Taylor, and D. Parlour, "Dynamic hardware plugins in an fpga with partial run-time reconfiguration," in *Design Automation Conference*, no. 39, 2002, pp. 343– 348.

[9] A. S. Tanenbaum, *Computer Networks*. Prentice Hall, 1981.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (Third Edition)*. Morgan Kaufmann, 2003.

[11] J. Sparsø and S. Furber, Eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, Boston, 2001.

[12] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 8, pp. 857–873, Aug. 2004.