# Interfacing Hardware and Software

M. Eisenring, J. Teich

Computer Engineering and Communication Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH)
Gloriastrasse 35, CH-8092 Zurich, Switzerland
email: {eisenring, teich}@tik.ee.ethz.ch

**Abstract.** The paper treats the problem of automatic generation of communication interfaces between hardware devices such as FPGAs and ASICs and software (programmable) devices such as microprocessors. In [2], we introduced an object-oriented approach to interface generation starting from a coarse-grain process graph specification level to the final device-dependent implementation. Here, we present generic templates of how to implement hardware/software interfaces, in particular in the scope of rapid prototyping environments that use reconfigurable hardware devices such as FPGAs. The major concerns here are 1) efficiency of communication protocols (speed), 2) low implementation complexity (area), 3) synthesizability, and 4) low power dissipation.
**Keywords: automatic interface synthesis, low power design, hardware/software codesign, rapid prototyping**

## 1 Introduction

Embedded systems impose strong constraints on design criteria such as area, speed and power dissipation. These often force a mixed implementation consisting of microcontrollers and microprocessors for implementing software functionality, and ASICs and FPGAs for hardware processes.

In order to find an appropriate solution which best fits the specification, design space exploration is neccessary. With the advent of configurable computing paradigms and the availability of powerful compiler and synthesis tools, the exploration of different design alternatives in the design space (rapid prototyping) has become a reachable goal. However, automatic interface synthesis, which generates a link between the communicating processes, becomes of increasing interest and importance and hence a key factor for rapid prototyping.

In [2], we introduced an object-oriented approach to automatic interface generation: Starting with a graphical specification in the form of a data flow graph (i.e. SDF graph [5]), and given a partition of tasks (nodes) onto hardware and software modules, we explained how the required communication channels as implied by the arcs in the graph may be refined such that the required interfaces in hardware and software (drivers) could be automatically synthesized. Due to the restricted communication semantics of the underlying data flow graph model (uni-source, uni-sink FIFO-buffered channels with blocking read, non-blocking write semantics), the refinement could be done in a completely automated manner.

Here, we want to focus on implementation aspects of hardware/software interfaces of a particular class of target architectures that use reconfigurable hardware devices such as FPGAs. In this area, the generated interface must satisfy the following requirements:

- *efficiency* of communication protocols (speed),
- *low implementation complexity* (area),
- *synthesizability*, and
- *low power dissipation*.

First, the communication should not slow down severely the application. Second, the interface should occupy the least amount of chip area possible. Also, the generated interface code should be synthesizable using existing compiler and hardware synthesis tools. Finally, there are sometimes power constraints that dictate a certain implementation. With these concerns in mind, we introduce 1) an efficient interface protocol, 2) an interface template for communication on and with reconfigurable hardware devices. 3) A gated-clock implementation is proposed for minimizing the power dissipation of the functionality that is implemented in hardware.

In the following, we present an efficient protocol for synchronizing two communicating partners in hardware (Section 2), a generic hardware template for interfacing a reconfigurable hardware device with a microprocessor, and an efficient and power saving implementation of hardware using a gated-clock technology (Section 3). Finally, we describe how these interface techniques are embedded into our object-oriented interface generation tool HASIS [2].

## 2 Templates for hardware/software communication

In [3], we described different interface models for communication between processors and hardware components. Here, we propose a special efficient interface protocol for two communication partners being mapped onto hardware devices being clocked with the same clock and a generic hardware/software interface template.

### 2.1 An efficient synchronization protocol

Between independent subsystems we use a handshake protocol as described in [7] for synchronisation. In the context of this paper, we assume that subsystems use the same clock and that the exchange of data is synchronized to this clock. Each subsystem has a request line for each channel to request a transmission and an acknowledge line to get the request of the communication partner. The events on that lines are described as follows:

*req+* : ready to transmit data at the next active clock edge
*req-* : no further data to transmit

The communication takes place at the next active clock edge if both communication partners have released an event req+. The protocol has the following properties: 1) symmetric, 2) communication within one clock cycle possible, 3) request remains valid until it is acknowledged, 4) easy to implement.

*Example 1.* Figure 1 presents the synchronisation between two synchronous subsystems A and B of unknown computation time. Each node signals a request for communication by emitting req+, e.g. reqA+. The communication takes place at the next rising clock edge where both events reqA+ and reqB+ have been emitted (see in Fig. 1 times X, Y and Z).

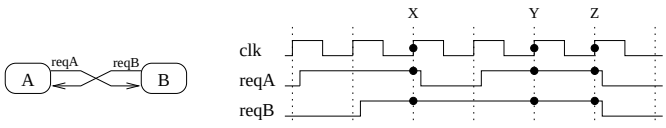


**Fig. 1.** Synchronisation between two synchronous subsystems

### 2.2 Generic hardware/software interface template

Similar to [6], we use a configurable interface template between hardware nodes and programmable software devices, e.g. microprocessors. It consists of three concurrent parts (see Fig. 2a) which allows to switch off power of unused circuit parts and is generated by HASIS [2] for various processors.

1. The *address decoder* decodes the addresses and enables parts of the controller logic if an appropriate address has been used by the microcontroller. This block contains combinational logic and an optional address register in case of a shared bus for address/data.
2. The *controller logic* contains state machines to implement a specific microcontroller bus protocol and the state machines for the request lines (see Section 2.1) of the connected hardware nodes.
3. The *data I/O* part is a (de)multiplexer for the access to the shared resource microcontroller data bus. There exist three different unidirectional transfer elements (see Fig. 2b) which may be instantiated as input or output. They are activated by the enable signal $e$ through the controller logic. The *TRI-STATE switch* realizes a direct link between the data bus and an internal channel. With the *asynchronous register*, data may be stored at any time. The *synchronous register* allows to save a data value at the next active clock edge.

## 3 Reduction of power consumption of hardware nodes

Power dissipation of sequential circuits is proportional to the used clock frequency and is a design constraint for embedded systems. Numerous approaches like [4] and [1] present solutions to reduce power. Here, we propose a *gated-clock* [1] (see Fig. 3a) which allows to switch off parts of a sequential circuit that are actually not in use.

Our implementation makes use of the handshake protocol between synchronous subsystems as proposed in Section 2.1. The idea is to stop the clock of a hardware node if there is no further computation to do. Therefore, a node has two different modes.

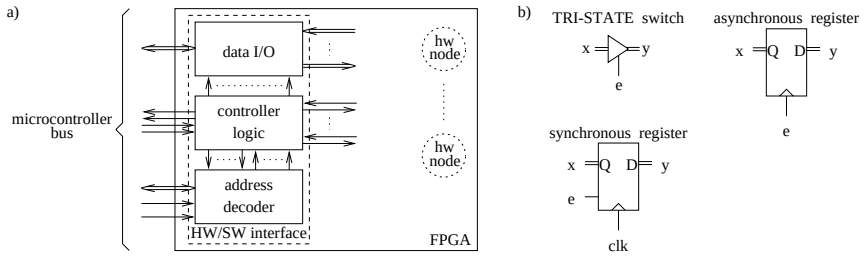


**Fig. 2.** a) HW/SW interface template, b) data I/O components

- *node enabled*: The node is running by executing its task.
- *node disabled*: The node is waiting for data transmission on all of its channels and therefore the clock is disabled. The clock is reenabled if at least one request for data transmission is acknowledged.

An implementation of a *gated-clock* power saving scheme may be described as follows. To each subsystem, a clock circuit is added which observes the node

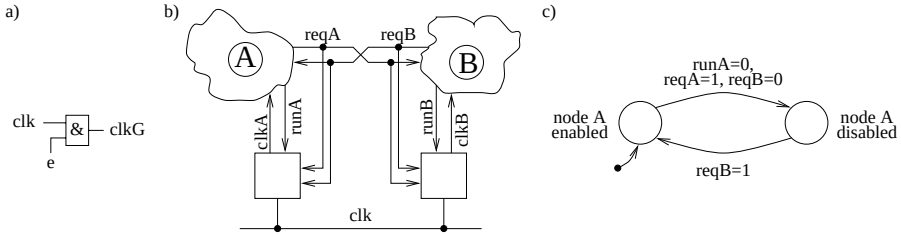


**Fig. 3.** a) Gated-clock, b) circuit structure, c) modes of node A

state and the request lines of all channels of the node, e.g. Fig. 3b).

*Example 2.* Figure 3b) presents two nodes A and B connected via one channel (only request lines shown). Each node has an additional clock circuit and an output signal (e.g. *runA*) which shows the node activity (i.e. *runA*=1: node A runs, *runA*=0: no further computation for node A). The clock circuit disables the node clock (e.g. *clkB*) if there is no computation to do.

*Example 3.* Figure 3c) presents the two node states of node *A* in Fig. 3b). The node remains disabled as long as its request isn't acknowledged. For simplicity, only conditions which force a state transition are shown.

*Example 4.* Figure 4a) shows a possible circuit implementation. If the system is implemented on an FPGA, e.g. the XILINX XC4062XL, each gated-clock circuit requires only one CLB (configurable logic block). If the node clock (e.g. *clkA*) is enabled, it has one gate delay compared to *clk*. Figure 4b) shows parts of the timing diagram for the gated-clock circuits in Fig. 4a). At time **X** node A

releases *reqA+* which is not acknowledged at the next clock edge. Therefore *clkA* has been disabled (see **Y**). It is reenabled if node B releases its *reqB+* (see **Z**).

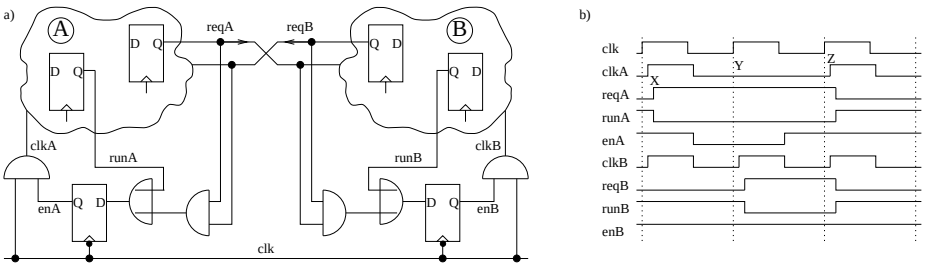


**Fig. 4.** a) Circuit for gated-clocks, b) timing diagramm for gated-clock

To satisfy the setup/hold conditions between a hardware/software interface and a hardware node, the *req+* event of the interface is delayed for a half clock period.

## 4 HASIS, a hardware/software interface generator

An object-oriented approach is taken for the generation of interfaces [3]. The major advantage of the object-oriented approach was to define classes for abstract communication interfaces such as via I/O ports, shared-memory communication, using DMA or interrupt facilities, etc., which are supported by almost any kind of microprocessor family on the market. In order to allow code generation for a new microprocessor, only few device-specific parts of the code generation classes have to be reconfigured or rewritten because of class inheritance, see [2],[3].

## References


1. L. Benini and G. De Micheli. Transformation and synthesis of FSMs for low-power gated-clock implementation. *Int. Symp. on Low Power Design*, pages 21–26, 1995.
2. M. Eisenring and J. Teich. Domain-specific interface generation from dataflow specifications. In *Proceedings of Sixth International Workshop on Hardware/Software Codesign, CODES 98*, pages 43–47, Seattle, Washington, March 15-18 1998.
3. M. Eisenring, J. Teich, and L. Thiele. Rapid prototyping of dataflow programs on hardware/software architectures. In *Proc. of HICSS-31, Proc. of the Hawai'i Int. Conf. on Syst. Sci.*, volume VII, pages 187–196, Kona, Hawaii, January 1998.
4. Shyh-Jye Jou and I-Yao Chuang. Low-power globally asynchronous local synchronous design using self-timed circuit technology. In *International Symposium on Circuits and Systems*, volume 3, pages 1808–1811, June 1997.
5. E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
6. B. Lin, S. Vercauteren, and Hugo De Man. Constructing application-specific heterogeneous embedded architectures for custom HW/SW applications. In *ACM/IEEE Design Automation Conference*, June 1996.
7. Ad M.G. Peeters. *Single-Rail Handshake Circuits.* PhD Thesis, Eindhoven University of Technology, 1996.