Using Synchronizers for Refining Synchronous Communication onto Hardware/Software Architectures

Zhonghai Lu, Jonas Sicking, Ingo Sander and Axel Jantsch
Royal Institute of Technology, Sweden
{zhonghai,ingo,axel}@kth.se, jonas@sicking.cc

Abstract

We have presented a formal set of synchronization components called synchronizers for refining synchronous communication onto HW/SW codesign architectures. Such an architecture imposes asynchronous communication between HW-HW, SW-SW and HW-SW components. The synchronizers enable local synchronization, thus satisfy the synchronization requirement of a typical IP core. In this paper, we present their implementations in HW, SW and HW/SW, as well as their application. To validate our concepts, we conduct a case study on a Nios FPGA that comprises a processor, memory and custom logic. The final HW/SW implementation achieves equivalent performance to pure HW implementation. Our prototyping experience suggests that the synchronizers can be standardized as library modules and effectively separate the design of computation from that of communication.

1 Introduction

Many System-on-Chip (SoC) designs start with a functional specification that assumes *synchronous Model of Computation* (MoC) [2, 12]. This design style is attractive since it allows one to separate timing from function. The designer can concentrate on the design of the system functionality without being distracted by low-level communication details. This also facilitates verification, which is a key activity at the system level. Later, the implementation details and design constraints can be gradually filled in and fulfilled by refinement.

To deal with increasing SoC design complexity, IP reuse is a key [5]. By using validated components, time-to-market can be shortened; design productivity can be increased; design quality can be better guaranteed. Communication refinement, which is a crucial step in a refinement-based design flow, must address IP reuse. However, given a huge number of IP components, which can be either hardware (HW) or software (SW), integrating them in a whole system presents challenges. First, most HW IP modules are synchronous and probably run with a different speed. Composing these synchronous components together has to bridge and arbitrate different clock domains. Second, refining communication between software IP modules must observe data and control dependency. Not only functionally correct, a refined system implementation must be efficient. Third, communication between HW and SW components needs to explicitly or implicitly pass data and control. While such problems as refining synchronous communication onto asynchronous architectures can be addressed by *ad hoc* approaches, providing a systematic approach is essential to maintain functional correctness, to enable automation, and to facilitate IP reuse.

We have proposed a set of well-defined synchronizers to glue synchronous components on asynchronous communication architectures [11]. In this paper, we focus on their *implementation* and *application*. The remainder of the paper is structured as follows. The related work is briefed in Section 2. We introduce the synchronous MoC and a digital equalizer model in Section 3. The concept of synchronizers is summarized in Section 4, followed by the implementations of the synchronizers in HW, SW and HW/SW in Section 5. In Section 6, we present the case study prototyped on an FPGA. Finally we conclude the paper in Section 7.

2 Related Work

Based on the separation of communication from computation and of function from architecture, a large body of work on communication refinement exists in the literature. Through the Virtual Component Interfaces (VCI) of the VSI Alliance [9], the COSY-VCC design flow [3] supports communication refinement from specification, to performance estimation and to implementation. IPSIM [6] developed on top of SystemC 3.0 supports an object-oriented methodology and establishes two inter-module communication layers. The message box layer concerns generic and system-specific communication, while the driver layer implements higher level application dependent communications. The SpecC methodology defines four levels of abstraction, namely at the specification, architecture, commu-

nication and implementation level, and the refinement transformations between them [7].

Unlike the above works, our work starts with a system model specified using synchronous MoC. The Latency Insensitive Design (LID) [4] assumes synchronous modeling paradigm as ours. It targets synchronized HW design when wires interconnecting IP blocks experience indefinite latencies. However, LID considers synchronized HW, therefore providing global clock and state. Our work targets architectures where neither global clock nor state exists. Second, LID is not applicable to parallel software architectures in which communication is offered by an operating system. Our approach is applicable to pure SW (single-threaded and multi-threaded) and mixed HW/SW in addition to pure HW.

3 Specification in Synchronous MoC

3.1 Functional system specification

The synchronous MoC [2, 12] is based on an elegant and simple mathematical model, the *perfect synchrony hypothesis*, i.e., both computation and communication take no observable time. It has been the ground of synchronous languages such as Esterel, Signal, Argos and Lustre.

A system is modeled as a set of concurrent communicating processes via signals. Processes use ideal data types and assume infinite buffers. Signals are ordered sequences of events. Each event has a time slot as a slot to convey data. If the data contains useful value, the event is *present* and called a *token*; otherwise, the event is *absent* and modeled as a $\sqcup$ representing a clock tick. Each signal can be related to the time slots of another signal in an unambiguous way. The output events of a process occur in the same time slot as the corresponding input events. Moreover, they are instantaneously distributed in the entire system and are available to all other processes in the same slot. Receiving processes in turn consume the events and emit output events again in the same time slot. A signal can thus be viewed as being transported on an *ideal communication channel* which has no delay for any event data types (unlimited bandwidth).

Two events are *synchronous* if they have the same tag. Two signals are *synchronous* if each event in one signal is synchronous with an event in the other signal and vice versa. A process is *synchronous* if every signal of the process is synchronous with every other signal of the process. A system is *synchronous* if all processes are synchronous locally and globally (synchronous with signals of other processes). A system specified in the synchronous paradigm is a synchronous system. For feedback loops, the perfect synchrony leads to cyclic dependency between an input signal and an output signal. If such cyclic communication is allowed in system behavior, some mechanism must be used to resolve it [8]. One possibility is to introduce a delay in the output signal. This breaks the deadlock, and the system model becomes deterministic, i.e., given the same input sequences of events, it generates the same output sequences of events.

3.2 The digital equalizer model

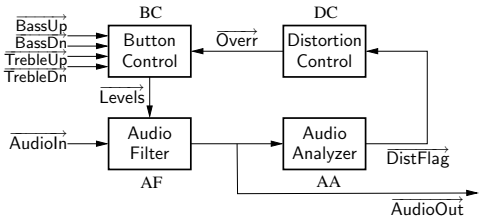

Figure 1. The digital equalizer.

As an example, we show the specification of an equalizer, whose functionality is to regulate the bass and treble components of an audio stream $\overrightarrow{\mathsf{AudioIn}}$ in response to the user input through the four button signals, namely, $\overrightarrow{\mathsf{BassUp}}$, $\overrightarrow{\mathsf{BassDn}}$, $\overrightarrow{\mathsf{TrebleUp}}$ and $\overrightarrow{\mathsf{TrebleDn}}$. The four signals are collectively called $\overrightarrow{\mathsf{Buttons}}$ signal. In addition, it has a control loop that prevents the bass level from exceeding a predefined threshold in order not to damage the speakers.

The digital equalizer is structurally decomposed into four functional blocks or subsystems shown in Figure 1. Its function is specified by the following set of equations:

$$
\begin{array}{lcl}
\mathit{AudioOut} & = & \mathit{Equalizer}(\mathit{Buttons}, \mathit{AudioIn}) \\
\text{where} & & \\
\mathit{AudioOut} & = & \mathit{AF}(\mathit{Levels}, \mathit{AudioIn}) \\
\mathit{Levels} & = & \mathit{BC}(\mathit{Buttons}, \mathit{init} : \mathit{Overrides}) \\
\mathit{DistortionFlag} & = & \mathit{AA}(\mathit{AudioOut}) \\
\mathit{Overrides} & = & \mathit{DC}(\mathit{DistortionFlag}) \\
\mathit{init} & = & \sqcup
\end{array}
\tag{1}
$$

The first equation represents the system layer. It takes two input signals $\overrightarrow{\mathsf{Buttons}}$ and $\overrightarrow{\mathsf{AudioIn}}$ as arguments, generating the output signal $\overrightarrow{\mathsf{AudioOut}}$. The evaluation of this equation calls for the evaluation of the next four equations that describe the subsystem layer. The final equation sets the initial value of the signal $\overrightarrow{\mathsf{Overrides}}$ ($\overrightarrow{\mathsf{Overr}}$) to $\sqcup$, which is used to resolve the cyclic dependency due to the feedback loop. The *Audio Filter (AF)* subsystem, as depicted in Figure 2, handles the bass and treble components of the digital audio input in response to the amplification level from the *Button Control (BC)*. The *Audio Analyzer (AA)* analyzes the audio output signal and checks if the bass exceeds a predefined threshold using a Fast Fourier Transform (FFT) component to determine the frequency spectrum. The *Distortion Control (DC)* determines if a violation occurs. It generates the corresponding commands for the *Button Control*. The *Button Control* monitors the button inputs and the override signal from the *Distortion Control*, in turn passing the amplification level to the *Audio Filter*.

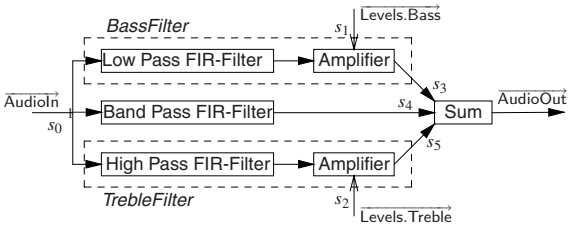

Figure 2. The *Audio Filter*'s internal structure

4 The Concept of Synchronizers

4.1 The synchronization problem

In the system model all signals of processes are synchronous. A process expects to receive and emit events containing values in a regular interval. However, when implementing the model in HW and SW, it is not always possible to guarantee this. There is no ideal communication. Instead the communication channels are typically bandwidth-limited due to resource sharing, and may have a stochastic delay. Since the correctness of a functional model is based on the ideal communication, any deviation from the *perfect* communication could lead to inconsistent behavior. Any implementation of a functional model must therefore maintain *synchronization consistency*, i.e., the synchronization semantics must be preserved. This is achieved through well-defined *semantic-preserving* communication refinement.

4.2 Process synchronization property

The perfect synchrony may overly specify a system due to globally synchronizing all processes and signals. In fact, whether or not the input signals of a process must be synchronous, i.e., the synchronization property of a process, is subject to the evaluation condition of the process, i.e., the condition(s) to *evaluate* its input events. Because of the tight synchronization in the model, some processes may be over specified, limiting the implementation alternatives. During the refinement, the designer(s) must inspect and determine the synchronization property of the processes.

In [11], we use *firing rules* to discuss the synchronization property of *synchronous processes*. For a synchronous process with n input signals, PI is a set of N input patterns, $PI = \{I_1, I_2, \cdots, I_N\}$. The input patterns of a synchronous process describe their firing rules, which give the conditions of evaluating input events at each event cycle . I_i ($i \in [1,N]$) constitutes a set of event patterns, one for each of n input signals, $I_i = \{I_{i,1}, I_{i,2}, \cdots, I_{i,n}\}$. A pattern $I_{i,j}$ contains only one element that can be either a wildcard $*$ or an absent value $\sqcup$, where $*$ does not include $\sqcup$. Based on the definition of firing rules, we have proposed four classes of process synchronization properties as follows:

- *Strict synchronization*: All the input events of a process must be present before the process evaluates and consumes them. The only rule that the process can fire is $PI = \{I_1\}$ where $I_1 = \{[*], [*], \cdots, [*]\}$.
- *Nonstrict synchronization*: Not all the input events of a process are absent before the process fires. The process can *not* fire with the pattern $I = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$. This class also includes processes that can not fire if one or more particular input events are $\sqcup$.
- *Strong synchronization*: All the input events of a process must be either present or absent in order to fire the process. The process has only two firing rules $PI = \{I_1, I_2\}$, where $I_1 = \{[*], [*], \cdots, [*]\}$ and $I_2 = \{[\sqcup], [\sqcup], \cdots, [\sqcup]\}$.
- *Weak synchronization*: The process can fire with any possible input patterns. For a 2-input process, its firing rules are $PI = \{I_1, I_2, I_3, I_4\}$ where $I_1 = \{[*], [*]\}$, $I_2 = \{[\sqcup], [\sqcup]\}$, $I_3 = \{[*], [\sqcup]\}$ and $I_4 = \{[\sqcup], [*]\}$.

We can identify processes with a *strict*, *strong*, and *weak* synchronization property in the equalizer (Figure 1 and 2). The $BassFilter$ (s_0 and s_1) and $TrebleFilter$ (s_0 and s_2) have a strict synchronization. Both filters are composed of a FIR filter and an amplifier. The FIR filter is a *sequential* process specified as an FSM. Its state transition is sensitive to timing, thus an $\sqcup$ value in an audio stream can change the values of its output sequence. Meanwhile, the amplifier must have an amplification level. An $\sqcup$ value makes the amplifier undefined. The *Sum* process (s_3, s_4 and s_5) has a strong synchronization. It is a *combinational* process and thus tolerable to events with an $\sqcup$ value. When it receives $\sqcup$ events, it will output $\sqcup$ events. However, the three events of s_3, s_4 and s_5 must be synchronized before being processed since they represent the low, medium and high frequency components of the same audio sample. The *Distortion Control* and *Button Control* processes generate amplification levels for the audio filters. They have a weak synchronization. They can fire even when either or both of their input events are absent ($\sqcup$) since pressing buttons happens irregularly and the bass level surpassing the threshold occurs aperiodically.

4.3 Achieving synchronization consistency

Apparently, for processes with a strict or strong synchronization, their synchronization properties can not be satisfied if any of their input signals passes through a nondeterministic channel since the delay via such a channel is stochastic. As a consequence, the signals of processes can not be globally synchronous. However, they can be locally synchronized by using adapters to satisfy their synchronization properties. To achieve strong synchronization, we use a synchronizer process *sync*; to achieve strict synchronization, we use three processes, *sync*, *deSync* and *addSync*. We use a two-input process to illustrate these processes in Figure 3. A synchronizer process *sync* aligns the tokens of its input events, as shown in Figure 3a. It does not change the time structure of the input signals. A desynchronizer *deSync* removes the absent values, as shown in Figure 3b. All its input signals must have the same token pattern, resembling the output signals of the *sync* process. Removing

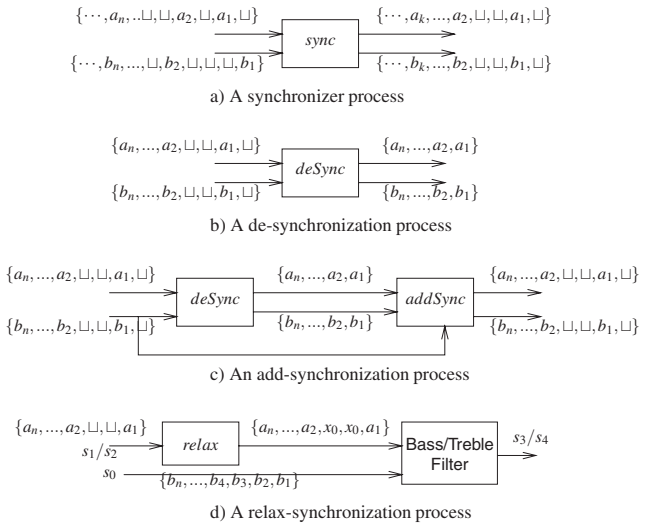

Figure 3. Processes for synchronization

absent values implies that the process is *stalled*. The desynchronizer changes the timing structure of the input signals, which must be recovered in order to prevent from incurring unexpected behavior of other processes that use the timing information. An add-synchronizer *addSync* adds the absent values to recover the timing structure, as shown in Figure 3c. It must be used in relation to a *deSync* process. If the input events of the *deSync* is a token, the *addSync* reads one event from its internal buffers for each output signal; otherwise, it outputs an $\sqcup$ event. As can be seen, the two processes *deSync* and *addSync* are used as a pair to assist processes to fulfill strictness.

For feedback loops, we use a *relax* synchronizer to generate synchronization events whenever necessary, as shown in Figure 3d. If the input event is a token, it outputs the token; otherwise, a token x_0 is emitted. The exact value of x_0 is application dependent. In this way the loop is virtually broken, and the system throughput can be greatly enhanced. Relaxing synchronization is a *design decision* to preserve synchronization semantics. It leads to value discrepancy between the specification and the refined model. This value discrepancy must be acceptable by the system requirements. Otherwise, we can not use the *relax*.

Next, we present the synchronizers' implementations in HW, SW and mixed HW/SW. As we shall see, the parameterizable implementations may be standardized in a library.

5 HW and SW Implementations

5.1 Synchronizers in HW

5.1.1 The *deSync* and *addSync* components

As we discussed in Section 4.3, the *deSync* component takes stochastically arriving events coming from a channel and feeds them into a synchronous process. It is used in a pair with an *addSync* component, as illustrated in Figure 4.

The *deSync* process determines when the synchronous process can be started and then fire it. One way is to raise a start signal connected to the process which will let the process execute until the output data is computed (*start-done* method). Another way is to have a clk_enable signal that triggers the process execution as long as the signal is raised (*clock gating* method). To detect whether the resulting output data is produced the process can either have a done output signal, or it could always produce data a certain number of cycles after it has fired.

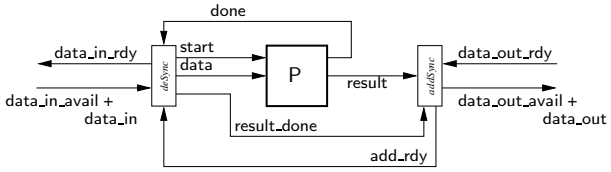

Figure 4. A deSync using *start-done* method.

Using the different signaling schemes results in different implementations of the *deSync* component. Figure 4 shows the input and output signals of the *deSync* using the start-done method. When the *addSync* is ready to store data it raises add_rdy which enables the *deSync* to fire the process by raising start. Once the process completes its computation, the result is presented on result and done is raised. The *deSync* component will then forward the done signal to the *addSync* component via result_done which stores and forwards the result. The reason why done is not connected directly to the *addSync* is that done might stay raised until the process is fired again. If done was connected directly to *addSync* it would be unable to differentiate a halted process from one that produces data every cycle. Using the clock gating method, the *deSync* implementation will interact with a process that is pipelined such that it takes new input data and produces new output data every cycle [13].

The *addSync* component is attached to the output of a synchronous process (Figure 4). In addition to recovering timing information, the *addSync* has two primary functions: store the output data of the synchronous process, and send a signal to the *deSync* when it is able to receive more data.

5.1.2 The *sync* component

The *sync* component synchronizes events from two or more stochastic channels. On the input channels, events can arrive independently of the other channels. But on the output channels, events are always sent out at the same time.

Conceptually the *sync* is connected as shown in Figure 3a. It has as many input channels as output channels. While implementing the *sync*, its output signals must never use separate channels if one of the channels can be faster than the other. This could make the events sent on the chan-

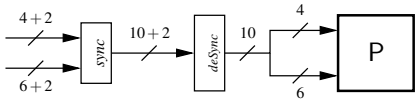

Figure 5. An output-merged *sync* component.

nels unsynchronized again reversing the effect of the *sync* component. Although this can be handled by using *deSync* components, it is better to merge the output signals of the *sync* and send them in parallel. This will also ensure that the events never get unsynchronized again, we could even send them over a stochastic channel before being handled by a *deSync* component. Figure 5 shows an example with the bit width of signals marked, where '+2' is the data_out_avail and data_in_rdy signals.

The signaling methods can work in a system in which events in different parts of the system may travel with a different speed. To support multiple clock domains, we only need to place a clock domain bridge, for example, a buffer, on the border between two domains. It receives events using one clock and sends events using the other [13].

5.2 Synchronizers in SW

While mapping a system model in pure software, the problem of maintaining synchronization consistency essentially turns into a dependent scheduling problem, because the execution of processes requires the token availability that is the pre-condition to fire a process. It is not necessarily to design specific software components to implement synchronizers. Instead an intelligent *scheduler*, which checks the conditions to fire a process and determines the firing sequence, *implicitly* implements the synchronizers.

In a single-thread environment, process execution must be serialized. It is therefore mandatory to compute a schedule. Such a schedule may be computed during compile-time or run-time. For the equalizer, at the system layer we can statically schedule the four parallel subsystem components, leading to a PASS (Periodic Admissible Sequential Schedule) as $PASS(Equalizer) = \{BC, AF, AA, DC\}$, where the *Button Control* should run first considering the initial token *init* on the override signal. For each individual process, we can further schedule its internal processes with finer granularity [10].

In a multi-threaded environment with the operating system support, process execution is virtually parallelized. Both dynamic and static scheduling methods can be used to compute a schedule and perform memory requirement analysis [1]. A static method generally costs less overhead but also less flexible. There are applications for which a PAPS (Periodic Admissible Parallel Schedule) may be computed off-line. But for most cases, a dynamic scheduler has to conditionally schedule the processes.

5.3 Synchronizers for mixed HW/SW

In a mixed HW/SW environment, the synchronizers can be implemented with *interface components* plus a *scheduler*. The interface components are HW components which enable a HW process to communicate with a SW process via, for instance, a shared bus, and vice versa. Specifically, a *HwSw* component receives events from a HW channel and sends it to a SW process; a *SwHw* component receives the result from a SW process and sends it to a HW channel. Data are sent via the bus through write and read transfers to the addresses of the interface components (memory-mapped I/O). Both polling and interrupt methods can be used to signal if the data is available to be accessed. In

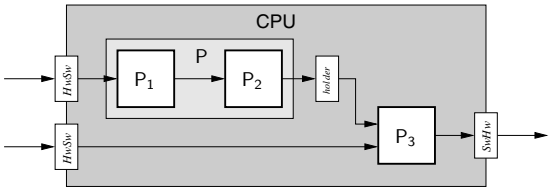

Figure 6. Mixed HW and SW processes.

addition, a scheduler has to serialize process execution for single-thread execution or parallelize process execution for multi-thread execution, and meanwhile resolves the synchronization requirement of processes.

Figure 6 shows an example of a mixture of HW and SW processes. To serialize the system we use a combination of static and dynamic serialization. P_1 and P_2 can be statically serialized within P as $PASS(P) = \{P_1, P_2\}$, while both P and P_3 would be dynamically serialized and fired by the scheduler. The *holder* component stores an event until it can be picked up by P_3. It decouples the execution of P_3 from P. The scheduler is implemented in SW and runs as the main program loop. It is responsible for calling all other functions by looping though all top-level software processes and checking if they are able to fire. As soon as one process has data available on all input *HwSw* components and all output *SwHw* components are able to accept data, the scheduler will call the function implementing the process. The resulting scheduler code is sketched as follows:

```
typedef volatile struct{int val;int set;
} usr_sync_data;
struct usr_sync_data holder={0, 0};
int P_func(int inval){
  return P2_func(P1_func(inval));
}
void run_scheduler(){
 while (!stop_scheduler()) {
 if(P1_in->set && !holder->set) {
    holder->val=P_func(P1_in->val);
    holder->set=1;}
 if(P3_in->set && holder->set && !P3_out->set){
    P3_out->val=P3_func(holder->val, P3->val);
    holder->set=0;}}}
```

6 Prototyping on NIOS FPGA

6.1 The synchronizer-based design flow

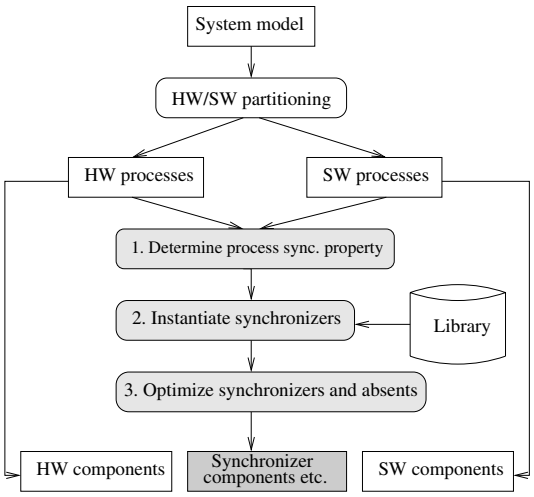

Figure 7. Synchronizer-based design flow

Figure 7 illustrates the synchronizer-based design flow. A system is first partitioned into hardware and software processes. The processes are *computational* processes. They are designed into hardware and software components, respectively, by, for example, reusing IP cores or customization. This design of computation can be developed in parallel with that of communication, which mainly consists of three steps:

1. Determine the synchronization property of processes: This is to identify whether a process has a strict, non-strict, strong or weak synchronization requirement according to the synchronization classification by designers.

2. Instantiate synchronizers: After the synchronization property of processes is identified, the instantiation of synchronizers from a library is straightforward. To wrap a strict process, we use three synchronizers: a *sync*, a *deSync* and an *addSync*. For a strong process, we use a *sync* synchronizer.

3. Optimize synchronizers and absents (⊔): Synchronizers can be used at a different level of process granularity. The same type of synchronizers may be merged to reduce overhead. Moreover, for feedback loops, the *relax* may be used to enhance the system throughput. Dealing with absents, which involves the analysis of system dependency on timing information, is also an optimization step. The optimization of absents and relaxed synchronization is design decision. It must be used without violating system requirements.

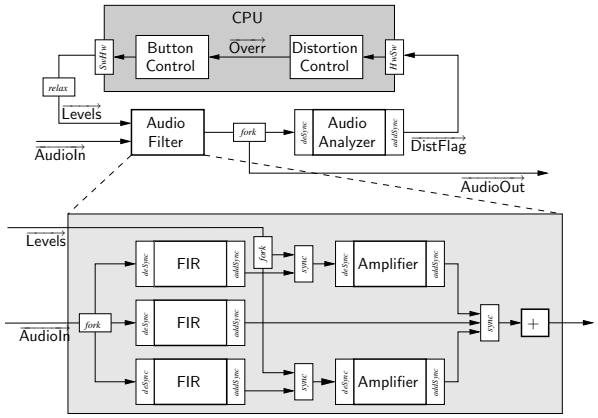

Figure 8. The equalizer with synchronizers.

In the following, we describe the prototyping of the equalizer example on a Nios FPGA [14] according to the design flow.

6.2 The mapped equalizer

We have prototyped the equalizer on an Altera Stratix FPGA [14], which is configured to comprise a Nios processor, memory and configurable logic. It uses the Avalon bus interconnect.

We first conduct an intuitive partitioning. Both the *Audio Filter* and *Audio Analyzer* contain relatively complex calculations. The *Audio Filter* contains a large number of multipliers in the FIR filters and the *Audio Analyzer* uses an FFT (256 points) transformation. We therefore choose to implement them in HW. The *Distortion Control* and *Button Control* processes contain simple operations that can be efficiently implemented in SW. These computational components are designed independently of communication.

After the computation blocks are designed, we assembled them together by inserting synchronizers into the system. The resulting system is shown in Figure 8. The *deSync* and *addSync* components are added before and after all strict processes. Two *sync* components wrap the amplifiers, and one *sync* wraps the *Sum* (+) process. The *HwSw* and *SwHw* components are placed surrounding the two software processes. As we discussed in Section 4.3, we use a *relax* component to optimize the feedback loop guarding from overloading the bass. We insert a *relax* between the *Button Control* and the *Audio Filter* and use a 'safe' level for the bass. As a safe level we reuse whatever level was used last time. This causes a small delay between that the *Audio Analyzer* detects that the bass is too high and that the bass volume is lowered. Since the feedback loop is so short the delay should be small enough and tolerable. The *relax* component will also insert the initial event to ensure the loop works. The *fork* components split one input stream into two or multiple identical output streams.

6.3 Further optimization and results

Dealing with absent (⊔) events is nontrivial. An ⊔ event does not contain a useful value but it carries timing information. For *combinational* processes, the timing information has no effect. However, to preserve synchronization semantics, it must be transmitted just like a token. To distinguish it from a token, we can send it as a special value or use a flag bit in the data. However, if the rate of ⊔ events is too high, too much communication overhead will be incurred. One optimization is aimed to reduce the rate of the ⊔ event. The *Audio Analyzer* sends events to the *Distortion Control*. 255 out of every 256 events will contain the ⊔ value. Since the *Distortion Control* will emit an event with the ⊔ value every time it receives such an event, the same will hold true for the $\overrightarrow{\text{Overr}}$ signal. However, the *Button Control* will not emit ⊔ events when ⊔ events are received, but it will emit an event with the same value as the last time. The *relax* component after the *Button Control* would repeat the same values anyway if no event was received. This means that we can skip sending the first 255 ⊔ events from the *Audio Analyzer* and reduce the rate of events on the feedback loop to 1/256th of the rate through the *Audio Filter*. As an optimization to reduce cost, we can merge the three pairs of *deSync* and *addSync* into one pair. The *Audio Filter* is then simplified as shown in Figure 9.

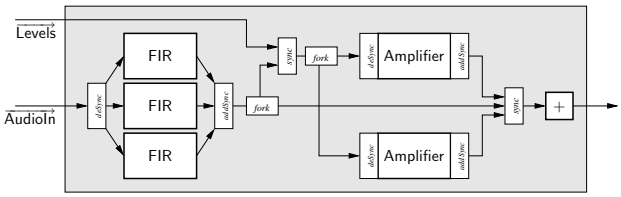

Figure 9. The optimized *Audio Filter***.**

The final HW/SW implementation achieves an average processing time of 19 cycles per 12-bit sample, which is the processing time of the *FIR* process (16 taps). This implies that *the performance is equivalent to pure hardware implementation* because, even if the system had been designed using pure HW, its performance still would have been limited by the same FIR process. The implementation consumes about 7000 logical elements, of which approximately 520 (7.4%) counts for the synchronizers (*deSync*, *addSync*, *sync* and *relax*).

7 Conclusion and Future Work

We have proposed a compact set of synchronization components to systematically glue synchronous components or IP blocks on asynchronous implementation architectures. They are used to maintain synchronization consistency from specification to implementation. We validated the concept and implementations of the synchronizers with an equalizer in an FPGA with mixed HW and SW. Our experience in the prototype suggests that using the synchronizers enables us to decouple the design of computation from communication. The insertion of the synchronizers is straightforward, and the resulting system is built correct-by-construction. Besides, their implementation overhead is small, enabling efficient design.

Our future work is to automate the synchronizer-based design flow. We are building more synchronizer implementations in the library for different application scenarios, for example, different bus width and clock rates of IP cores and different communication protocols between hardware and software. The process synchronization property may be annotated by designers. Alternatively, it may be embedded in a system specification if written in the pattern-match style [12], and therefore can be automatically extracted. For the optimization, heuristic algorithms may be developed.

References

[1] S. S. Battacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis From Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages 12 years later. *Proceedings of The IEEE*, 91(1):64–83, January 2003.

[3] J.-Y. Brunel, W. Kruijtzer, H. Kenter, F. Petrot, L. Pasquier, E. de Kock, and W. Smits. COSY communication IP's. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[4] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[5] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Springer, 2000.

[6] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. IP-SIM: SystemC 3.0 enhancements for communication refinement. In *Proceedings of Design Automation and Test in Europe*, 2003.

[7] R. Dömer, D. D. Gajski, and A. Gerstlauer. SpecC methodology for high-level modeling. In *Proceedings of the Ninth IEEE/DATC Electronic Design Processes Workshop*, April 2002.

[8] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded system: Formal models, validation and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[9] C. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee. Standards for system-level design: practical reality or solution in search of a question? In *Proceedings of Design Automation and Test in Europe*, March 2000.

[10] Z. Lu, I. Sander, and A. Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis*, October 2002.

[11] Z. Lu, I. Sander, and A. Jantsch. *Applications of Specification and Design Languages for SoCs*, chapter Refining synchronous communication onto network-on-chip best-effort services, pages 23–38. Springer, September 2006.

[12] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, Feb. 2004.

[13] J. Sicking. Implementation of asynchronous communication for ForSyDe in hardware and software. Master's thesis, Royal Institute of Technology, Sweden, IMIT/LECS/[2005-73].

[14] www.altera.com.