A Synchronization Algorithm for Local Temporal Refinements in Perfectly Synchronous Models with Nested Feedback Loops

Tarvo Raudvere, Ingo Sander, Axel Jantsch
Royal Institute of Technology
Stockholm, Sweden
tarvo,ingo,axel@imit.kth.se

ABSTRACT

Due to the abstract and simple computation and communication mechanism in the synchronous computational model it is easy to simulate synchronous systems and to apply formal verification methods. In synchronous models, a local temporal refinement that increases the delay in a single computation block may affect the functionality of the entire model. To preserve the system's functionality after temporal refinements we provide a synchronization algorithm that applies also to models with nested feedback loops. The algorithm adds pure delay elements to the model in order to balance the delay caused by refinement and to assure concurrent data arrival at computation blocks. It is done so that the refined model stays latency equivalent to the original model. The advantages of our approach are that (a) we remain fully within the synchronous model of computation, (b) we preserve the functionality of the existing computation blocks, and (c) we do not require additional computation resources, wrapper circuits or schedulers.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids; J.6 [**Computer-aided Engineering**]: Computer-aided Design (CAD)

General Terms

Design, Algorithms

Keywords

System Design, Design Refinement, Synchronization

1. INTRODUCTION

Synchronous computational models are popular in system design targeting safety critical applications and in the aerospace industry [13]. The synchronous hypothesis assumes that the computation in processes and communication between them takes no time. In this kind of models local temporal refinements like the introduction of pipelining and resource sharing are a potential source of errors due to changed time behavior, especially if models contain nested feedback loops. In this paper the term *local temporal refinement* means the replacement of a block A with an equivalent block B, which has longer delay than A. The additional delays cause mismatch in data arrival and therefore the system's correct behavior has to be restored.

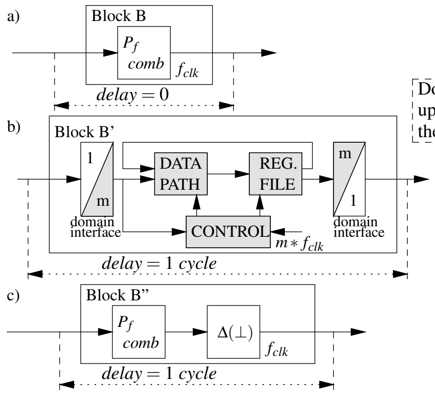

Figure 1: Resource sharing leads to several clock domains and adds an extra delay at the system level.

The problem is illustrated with a refinement in Figure 1, where a combinational function based on many identical operations in *Block B* is mapped onto *Block B'*, which implements resource sharing. We note that these two blocks do not have the same behavior. Due to the feedback signal from the register file to the data path, *Block B'* has internally a noticeable built-in delay. This delay is explicitly modeled in an equivalent *Block B''*. In the system perspective (a) the initial value in the additional delay generates unexpected output values and (b) concurrently arrived values in the original model will arrive at different time instants to multi-input processes.

In this paper we present a synchronization algorithm that is applied after local temporal refinements in order to balance the delay of all paths in the entire model. The algorithm preserves the system's functional behavior such that the refined model will be *latency equivalent* to the original one, i.e., the same output values appear in the same order but they are delayed. The refined model is correct-by-construction.

The general idea of the synchronization technique is illustrated in Figure 2. A refinement adds a delay to the process P_0. Any initial integer value of this delay will change the system's reaction to the input signal. Instead of integrating a controller to the model that executes processes when correct data is available, we use special

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy.
Copyright 2007 ACM 978-1-59593-605-9/07/0003 ...$5.00.

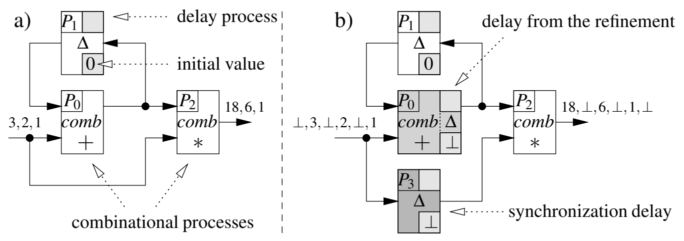

Figure 2: Problem introduction: a) initial model b) refined and synchronized model

absent values ($\perp$) for initialization of delays. To preserve concurrent data arrival at destinations (P_2) we add delay processes (P_3). In addition, we extend input signals with regular $\perp$-values. Although $\perp$-values appear in output signals as well, the refined model is latency equivalent to the original one.

2. RELATED WORK

Many researchers agree that a system design approach must start with a formal system model on a high level of abstraction [10, 7]. The initial model must then be refined manually or automatically [1] into a concrete implementation. Our synchronization technique supports design methodologies that use formal design transformations in a synchronous computational model. The latter model is the base for languages like Lustre [8] and Esterel [3], which have successfully been used for safety-critical industrial applications. The computational model has also similarities to the synthesizable subset in VHDL and Verilog languages that employ a clocked-synchronous computational model, which makes our approach very applicable in practice.

Retiming and pipelining are well-known techniques that address latency and data arrival problems. In order to reduce the circuit area or a critical path retiming algorithms [12, 11] relocate already existing memory elements. Although retiming techniques address synchronization problems, these problems are not caused by additional delays inserted to the model. On the other hand, the introduction of additional delays to the model is elaborated in pipelining transformations. In software pipelining different models than the perfectly synchronous one are used. In hardware the data synchronization is solved by a pipeline controller, derived from a high level system specification [15]. Our technique makes it possible to introduce pipelining in a synchronous model with nested feedback loops, without adding controllers or changing computational models.

In order to avoid synchronizations problems caused by temporal refinements desynchronization [2, 14] or latency insensitive design [5] (LID) techniques can be applied. The former technique transfers a synchronous model to an asynchronous one, which is less sensitive to delayed data arrival. LID targets the mapping of a synchronous model based on IP-blocks to hardware, where longer wires entail delayed data arrival. The synchronization problem is solved by (1) wrappers around IP-blocks stalling computation if input data is not available, and (2) by handshake channels and relay stations between IP-blocks that replace synchronous communication. The handshake mechanism distributes stalling messages and a relay station buffers data items if the destination process cannot consume them.

Although both of these techniques are common in practice, they have side effects that our synchronization algorithm avoids. We avoid unnecessary discontinuities in the design process caused by changes in the computational model. It is impractical to switch the computational model due to a single local temporal refinement. The use of the same computational model makes it much easier to verify refined models against each other. In addition, verification in deterministic synchronous models by using simulation or formal methods is simpler than in other models. On the other hand our synchronization technique is complimentary applicable within refinements in a synchronous island of GALS model or in an IP block of LID.

Our technique has similarities with the work in [9], where the labeling of delay processes is used to verify refined circuits with nested feedback loops. In our algorithm identical labels of delay processes show that the data from these processes reach a common destination process at the same time instant. If a refinement increases the delay on one path to the destination, it must be compensated on all others as well. The latter is similar to the work about software adaptation to reconfigurable hardware in [4], though this work does not address the synchronization of nested feedback loops in synchronous models.

3. SYNCHRONIZATION

3.1 System Model

We describe systems in the synchronous model of computation as a set of processes, which communicate through synchronous signals. Processes can be grouped into blocks of processes. Blocks may run at different clock speeds to model different clock domains. In this paper we assume that a system contains only combinational processes P_{comb} and simple delays P_{Δ}. More complex components can be constructed by combining combinational and delay processes. A signal is defined as a sequence of events $\{e_0, e_1, e_2, \dots e_i, \dots\}$, where an event e_i carries a value v and the index i is used as a time tag. All signals share the same set of tags for synchronization purposes. The signal direction is from the source process to the destination process, and every process has only one output signal. Every combinational process has a dedicated combinational function f. For each tag i, a combinational process consumes from its input signals $s_0, \dots, s_{n-1}$ events with the tag i carrying values $v_0, \dots, v_{n-1}$ and produces to its output signal s_k an event with the tag i and a value $v_n = f(v_0, \dots, v_{n-1})$.

There are two kinds of events: (1) *present events* that carry a value and (2) *absent events* that are used only for synchronization. An absent event e_j shows that a signal contains no value at a time instant j. We use the mark $\top$ as an abstract value if we refer to present values, and $\perp$ for absent values. For example, the abstract presentation of signal $\{1_1, 4_2, \perp_3, 2_4, \dots\}$ is $\{\top, \top, \perp, \top, \dots\}$. To extend a data type T to $T_{\perp}$ the $\perp$-value is added to its domain. We assume that the reaction of any n-input combinational process to input events is $P_{comb}(e^1, \dots, e^n) = \perp$, if for all i $(1 \le i \le n)$, $e^i = \perp$. A system is *$\perp$-consistent* if every process at every time instant receives only one type of values ($\top$ or $\perp$). We initialize the delay processes that temporal refinements introduces to the model with $\perp$-values. In order to keep the refined system $\perp$-consistent our algorithm synchronizes the model by inserting additional $\Delta(\perp)$ processes. We use the terms $\top$-delay process and $\perp$-delay process to indicate the initial value of delay elements, i.e., $\Delta(\top)$ and $\Delta(\perp)$ respectively. A *path* is a sequence of processes connected by signals that have the same direction from one end to the other. A *loop* is a cyclic path including no process twice. A *common source* is a process, whose output signal is connected to inputs of more than one process. A *common destination* is a multi-input process. A *pair of paths* contains two paths having one common source and one common destination, and no other process belongs to both of the paths.

3.2 Pattern Equivalence

In order to preserve the latency equivalence of synchronous models the delay of the paths that feed a common destination process (process P_4 in Figure 3.a) may be only increased equally. If a refinement adds a $\perp$-delay process to one path (P_5 in *path₁*), the rest of the paths have to be extended with $\perp$-delays as well (P_6 in *path₂*). The synchronization procedure gets much more complex if the system contains multiple feedback loops. If we insert a $\perp$-delay process (process P_{10} in Figure 3.b) to a path (*path₃*) that feeds a loop (*loop₁*), it is necessary to add a $\perp$-delay process (process P_{11}) to the loop as well. Otherwise the common process of the path and the loop receives in some clock cycles different types of events, which makes the system $\perp$-inconsistent. Since the $\perp$-values will be reproduced in the loop, the feeder system part has to deliver regularly $\perp$-values to the loop. The regularity is denoted as a *pattern*.

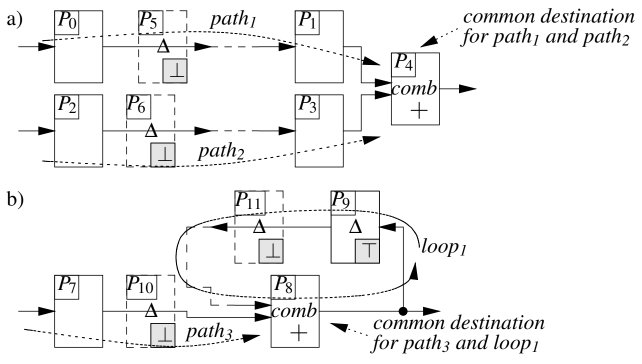

Figure 3: Synchronization of a) paths and b) loops

Pattern: *A pattern is defined as a minimal sequence* $V = \{v_0, \dots, v_i, \dots, v_{n-1}\}$ *of values* $\top$ *and* $\perp$*, which cannot be constructed by a single repetitive subpart* $\{v_0, \dots, v_i\}$.

For a process the pattern shows in which order $\perp$ and $\top$-values arrive.

Pattern Equivalence: *Two patterns* V^a *and* V^b *are equivalent if they have the same length* n*, and there exists an integer constant* k *such that for all* $i \in \{0, \dots, n-1\}$, $v_i^a = v_j^b$*, where* $j = ((i+k) \; mod \; n)$.

The pattern $V_1 = \{\top, \top, \perp, \perp\}$ is equivalent to $V_2 = \{\top, \perp, \perp, \top\}$, but not to the pattern $V_3 = \{\top, \perp, \top, \perp\}$. Elements in equivalent patterns can be rotated but not shuffled.

Let's consider a fragment of a loop where P_{comb1} sends data to P_{comb2} and there is only a $\Delta(\perp)$ process between them. If P_{comb1} has the pattern $\{\perp, \top, \top, \perp\}$, then the equivalent pattern $\{\perp, \perp, \top, \top\}$ corresponds to P_{comb2}. Due to the direct or indirect connections between all processes, the extension of one loop by a $\perp$-delay process leads to a scenario, where all processes have to operate with equivalent patterns, in order to preserve $\perp$-consistency. Since the delay of any loop divided by the pattern length has to give an integer result, we find the greatest common integer divider (*gcd*) of the loop delays in the original model. This is the basis of the pattern length (N). In addition, the pattern length has to correspond to the delay differences of all the pairs of paths that have a common source (fork) and a common destination (join). The delay difference of these paths has to be a multiple number of the pattern length. After the first temporal refinement, which adds a $\perp$-delay process to the model, the $N+1$ element pattern can be formed. According to this pattern the synchronization $\perp$-delay processes will be inserted to the rest of the model. For the pattern length calculation we do not need to take into account the loops, which are a combination of other loops since $gcd(n_1, n_2) = gcd(n_1, n_2, n_1 + n_2)$. Also we consider only pairs of paths from a fork to a join, i.e., the pairs that do not have a common prefix or infix.

3.3 Algorithm

The synchronization algorithm contains two parts: (1) to analyze the structure of the original model in terms of delay processes, (2) and based on the analysis to insert synchronization $\perp$-delays to the refined model. A system input can be imagined as a shift register, where one value enters to the system at every clock cycle. Similar to the changes in the system we extend the input signals with $\perp$-values.

ALGORITHM 1. *Labeling of delay Processes*

Step 1 Find the delays (D_i^{loop}) of all feedback loops.

Step 2 For all pairs of the paths ($path_a$,$path_b$) with a common source and a common destination, find the delay difference of the paths, $D_{a,b}^{path} = abs(D_a^{path} - D_b^{path})$.

Step 3 Calculate the greatest common integer divider N for the values found in the previous steps. N is the length of the equivalent patterns.

Step 4 Create an ordered set *Label* that contains N items - $\{L_0, L_1, \dots, L_{N-1}\}$. In the following we use capital letters $A, B, C, \dots$ as labels.

Step 5 Select a reference input and attribute a label L_j, $j = (D^{Pi} \; mod \; N)$, to every delay process P_i according to the delay D^{Pi} from the reference input to P_i. (The reference input is chosen arbitrarily without any influence to the synchronization result.)

Step 6 According to the delay D^{e_i} from the input event e_i to the reference input, assign the label L_j to e_i, $j = (-D^{e_i} \; mod \; N)$.

In the worst case the number of loops and pairs of paths grows exponentially with the number of processes n_p in the model, which gives an exponential complexity to the algorithm $O(k^{n_p}), (k > 2)$. In practice, there are not signals between every two processes, and the number of loops and pairs of paths is close to the number of processes in the model. Our heuristic implementation of the algorithm finds if there exists a loop with m delay processes in the model, but does not necessarily find all such loops.

The labeling can be illustrated on the system presented in Figure 4.a that contains eight combinational processes and four $\top$-delay processes. (`Step 1`) The given system contains two feedback loops: $loop_1 = \{p_1, p_2, \dots, p_{11}\}$ and $loop_2 = \{p_4, p_5, \dots, p_9\}$, having delays: $D_1^{loop} = 4$ and $D_2^{loop} = 2$. (`Step 2`) There are two pairs of paths: from P_0 to P_6, and from P_9 to P_4. $Path_1$ runs through $P_1, P_2, \dots, P_5$ and $path_2$ is the direct connection between P_0 and P_6. The delays of these two paths are two and zero clock cycles respectively and the difference of them is $D_{1,2}^{path} = 2$. The delay difference of *path₃*, $(P_{10}, P_{11}, P_1, P_2, P_3)$, and *path₄*, the direct connection from P_9 to P_4, is $D_{3,4}^{path} = 2$. (`Step 3`) The greatest common integer divider for the found values is $N = 2$. (`Step 4`) Thus the ordered set of labels contains two items, $L_0 = A$ and $L_1 = B$. (`Step 5`) The processes P_3 and P_8 get the label $L_0 = A$, since the delays from the system (reference) input are $D^{P3} \; mod \; N = 0 \; mod \; 2 = 0$ and $D^{P8} \; mod \; N = 2 \; mod \; 2 = 0$. In the same way P_5 and P_{11} get the

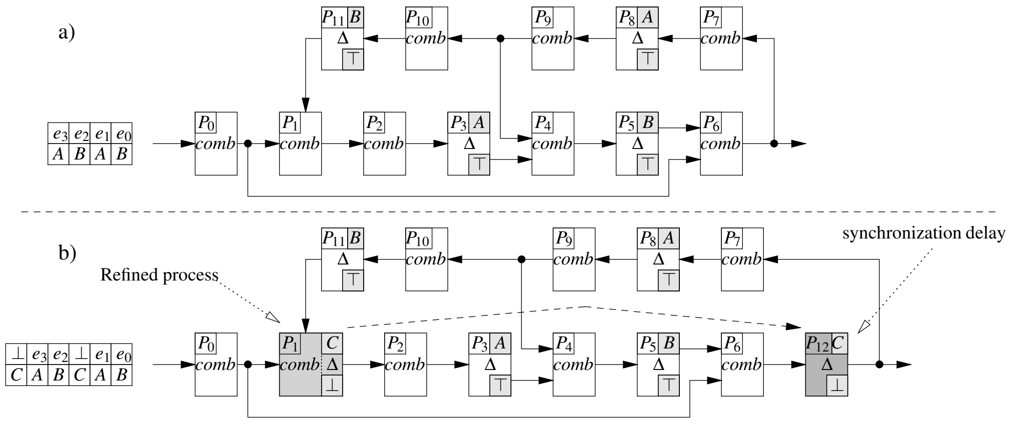
a)
b)
P_{11} B Δ $\top$
P_{10} comb
P_9 comb
P_8 A Δ $\top$
P_7 comb
e_3 e_2 e_1 e_0
A B A B
P_0 comb
P_1 comb
P_2 comb
P_3 A Δ $\top$
P_4 comb
P_5 B Δ $\top$
P_6 comb
synchronization delay
Refined process
$\perp$ e_3 e_2 $\perp$ e_1 e_0
C A B C A B
P_1 C comb Δ $\perp$
P_{12} C Δ $\perp$

Figure 4: Synchronization example : (a) initial model after labeling and (b) refined model

label $L_1 = B$. (`Step 6`) The input events are labeled following the given rule.

Algorithm 1 gives labels so that any multi-input combinational process (P_1 or P_4) operates at any clock cycle with values from delay processes, which have the same label. The latter condition holds also in the refined model, which is synchronized by Algorithm 2. The algorithm adds $\perp$-delay processes in order to make the refined model $\perp$-consistent. Since the $\top$-values are processed in the same order in the original model and in the $\perp$-consistent refined model after synchronization, these two models are latency equivalent.

ALGORITHM 2. *Balance the path delays*

Step 7 Find the delay $D^{new}_{\Delta(\perp)}$ from the system input to the inserted delay process $\Delta_{new}(\perp)$.

Step 8 A new label L_j will be associated with the delay element, where $j = D^{new}_{\Delta(\perp)}$ *mod* N.

Step 9 Increment the pattern length: $N = N + 1$.

Step 10 As the new label has got the position j in the set *Label*, shift all these labels in the set that had before the indexes $\{j, \ldots, i, \ldots, N-1\}$ such that $L^{new}_{i+1} = L^{old}_i$. According to the new positions of the labels a new pattern is formed, where $\perp$ corresponds to L_j.

Step 11 Insert a delay process $\Delta(\perp)$ with label L_j to every path in the system before the delay L_{j+1} (if $j = N$ then L_0) or after L_{j-1} (if $j = 0$ then L_N). Extend the input signal with $\perp$-values in the same way. In the new model the distances D^{Lj} from the system input to any process with label L_j have to satisfy the equation: D^{Lj} *mod* $N = j$.

In the example in Figure 4.b a temporal refinement adds a delay $\Delta(\perp)$ to process P_1. Algorithm 2 synchronizes the refined model as follows. (`Step 7`) The delay from the main input to the new delay in process P_1 is 0. (`Step 8`) The respective new label C will have the first position in the set *Label*, i.e., $L_0 = C$. (`Step 9`) $N = 3$ and (`Step 10`) the labels have new positions $Label = \{L_0 = C, L_1 = A, L_2 = B\}$. All processes will operate with patterns equivalent to $\{\perp, \top, \top\}$. (`Step 11`) Therefore we have to insert a delay process on every path after B-delays or before A-delays. Thus the process P_{12} can locate anywhere between P_5 and P_8. If it is before P_6, there has to be a C-delay on the lower input of P_6. According to the labels we extend the input signal as well.

In order to synchronize the system after further local temporal refinements we apply Algorithm 2. However, the previously added synchronization $\perp$-delay processes can be reused in refined processes without introducing new ones. It is allowed to shift $\perp$-delay processes in a non-branching structure between other delay processes. The shifting is valid since the sequential composition[1] $P_{\Delta(\perp)} \circ P_f = P_f \circ P_{\Delta(\perp)}$, if f is a combinational function and $f(\perp) = \perp$.

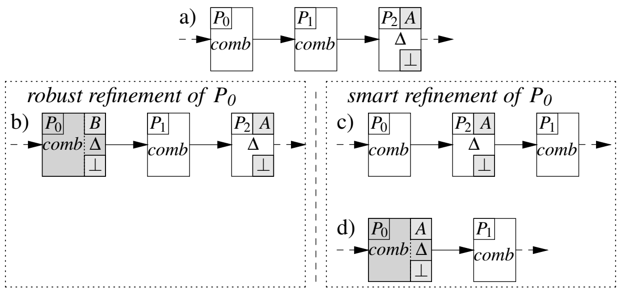
a)
P_0 comb
P_1 comb
P_2 A Δ $\perp$
robust refinement of P_0
b)
P_0 B comb Δ $\perp$
P_1 comb
P_2 A Δ $\perp$
smart refinement of P_0
c)
P_0 comb
P_2 A Δ $\perp$
P_1 comb
d)
P_0 A comb Δ $\perp$
P_1 comb

Figure 5: Delay processes can be relocated

Figure 5 presents two choices how to model a temporal refinement in P_0 (model a). In model b an additional delay is added to P_0. In the alternative case (models c and d), we shift the delay process P_2 next to P_0, and instead of introducing a new delay we merge P_0 and P_2. The latter choice performs the same refinement without increasing the system's delay.

In order to show that our algorithm can be applied to any system after a temporal refinement, we prove that there exists at least one pattern for any system and the system synchronized according to that pattern is $\perp$-consistent.

Assertion 1: There exists a pattern that fits to all loops and paths with a common source and a common destination.

Proof: According to Algorithm 1, the initial pattern length is calculated as the greatest common divider. Since the integer value *one* is a common divider for any set of integers, the respective one element pattern $\{\top\}$ exists for any system structure. In this case

[1]Sequential composition ($\circ$) of processes $P_1(x)$ and $P_2(x)$ is defined as $P_1(x) \circ P_2(x) = P_2(P_1(x))$

after a temporal refinement and synchronization all processes have to operate with patterns equivalent to $\{\perp, \top\}$.

Assertion 2: The system, which is synchronized according to the pattern $\{\perp, \top\}$ is $\perp$-consistent.

Proof: Let's replace in the original model every $\top$-delay process $\Delta(\top)$ by a sequential composition of delay processes $\Delta(\perp) \circ \Delta(\top)$. The modified model is $\perp$-consistent since at every second time instant all processes operate on $\perp$-values (the patterns are equivalent to $\{\top, \perp\}$). As we showed earlier (Figure 5), it is allowed to shift $\perp$-delays between two $\top$-delays [2]. In fact, all combinational processes in the modified model are between $\Delta(\top)$ and $\Delta(\perp)$. Thus the $\Delta(\perp)$ can be shifted to the combinatorial process for a temporal refinement. The system stays $\perp$-consistent and operates with the pattern $\{\perp, \top\}$. The described model is equivalent to the model synchronized by our algorithm.

The same assertion can be easily proved for any pattern length.

4. CASE STUDY: REFINEMENT OF A DIGITAL AUDIO EQUALIZER

We illustrate the synchronization algorithm within the design process of a digital audio equalizer (Figure 6.a). The equalizer divides the input signal into three frequency bands and amplifies two of them according to the button settings in the button control block. The sum of the amplified signals gives the equalizer output. The audio analyzer and the distortion control blocks adjust the bass amplification level to protect the speakers.

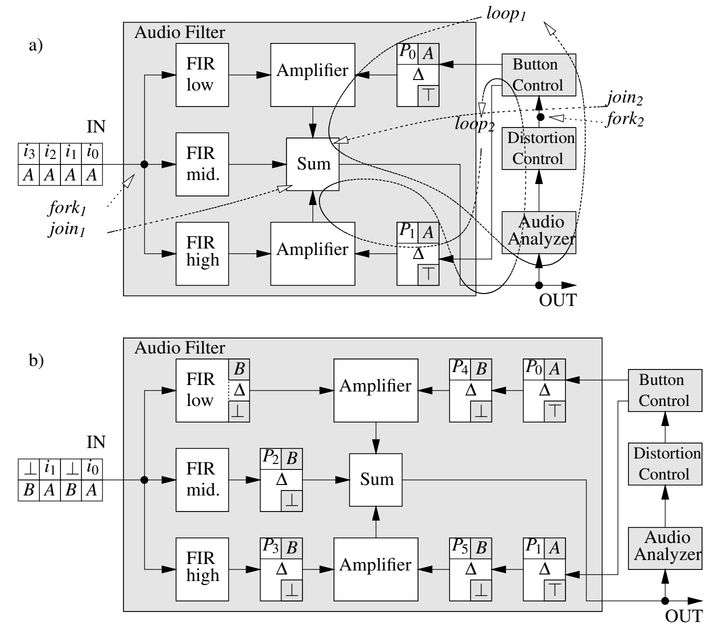

Figure 6: Audio equalizer: a) initial model b) after the clock domain refinement of the process FIR_{low}

The equalizer contains three order-k FIR-filters with polynomial functions $f(d,c) = \Sigma_{i=0}^{k-1}(d_i c_i)$ of delayed input values d and the filter coefficients c. Instead of implementing the function f in FIR_{low} as a large combinational circuit, we introduce resource sharing like in Figure 1. Due to the feedback signals in resource sharing, the refined block has an additional delay. In order to keep the refined models latency equivalent to the original one we apply the synchronization algorithm. In the following we shorten the explanation of the algorithm steps by viewing the FIR-filters as combinational processes and abstract from the existence of registers in the FIR-filter blocks[3].

The system contains two loops ($loop_1$ and $loop_2$ in Figure 6) with one delay process in each. The delay difference of common source/destination paths between respective fork and join points is in all cases equal to zero. Since the greatest common divider of the found values is one ($N = 1$), we create a one-element set *Label* $\{L_0 = A\}$ and mark the delay processes P_0 and P_1 and input events with A.

The $\perp$-delay added to the process FIR_{low} by the refinement gets a new label B. The delay from the system input to that process is zero. According to Algorithm 2 this gives a new content of *Label*, $\{L_0 = B, L_1 = A\}$ ($N = 2$), and processes operate with patterns equivalent to $\{\top, \perp\}$. In order to synchronize the system, we have to insert $\perp$-delay processes with label B between A-delays and between all input events. FIR_{mid} and FIR_{high} are refined in a similar way and the existing $\perp$-delay processes P_2 and P_3 are encapsulated in the refined blocks.

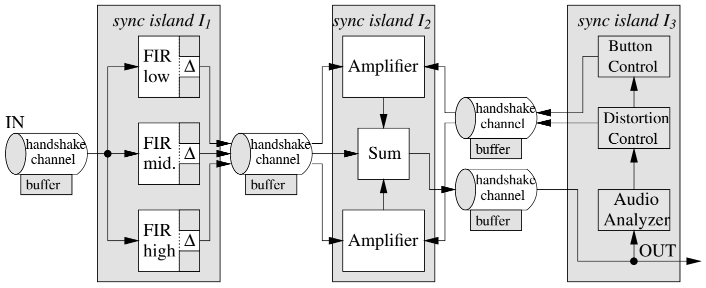

Figure 7: Implementation of audio equalizer as GALS or LID model

In Figure 7 an LID or GALS model based solution to the synchronization problem is showed. After adding delays to the FIR-filters, the data arrival from the filters and the button control block to the amplifiers is not synchronized. In order to restore the system functionality it is divided into three (1) asynchronously communicating synchronous islands or (2) synchronous blocks of LID, communicating via handshake channels. Computation in an island is executed when data is delivered by all its channels.

Although the LID method does not add any explicit delay process for synchronization, there are buffers in every channel, which increase the circuit area. In order to implement the absent extension at RT-level, we needed only one bit signal to inform processes about the current data type of an input value (absent or present). This is equivalent to the one bit signal in LID used to distribute stalling events between computation blocks. The input/output latency of LID and our model were the same, and both models had to work with two times more values than the original model. The LID relay stations in the channels are initialized with stalling values. Since the system contains feedback loops these values are reproduced [6]. Due to the feedback loops, the absent events are reproduced in our model as well.

The LID and GALS approaches are common in IP-block based design. Though they may not be the best candidates for refinements in small synchronous blocks, since create very small and

[2]In some extend shifting is also valid in branching structures.

[3]In the actual implementation all FIR-filters receive the delayed data values from a common block of shift registers. The synchronization algorithm labels all cells in the register by A, and inserts a $\perp$-delay between all of them. The rest of the system is identical to the described one

disproportionate islands, like I_1 and I_3. Due to the complex communication mechanism between synchronous islands, formal verification and formal refinements in this kind of models are considerably more complex. In addition, our model is more expressive, since the impact of refinements is explicit. The labels indicate relations between refined computation blocks and paths with increased delays. The explicit synchronization delay processes can be shifted to proper positions and reused for the further refinements. In the hardware model many of the $\perp$-delay processes can be mapped onto latches between combinational processes. Processes that feed latches with the same label have to have the same latency. Our synchronization algorithm does not point out the exact positions of $\perp$-delay processes or respective latches between combinational processes. However it is possible to find better places by using existing retiming techniques in order to reduce the number of memory elements or to maximize clock frequency.

5. CONCLUSION

The introduction of resource sharing and pipelining in computation blocks are only some examples of design refinements, which increase the delay in the refined blocks compared to the original ones. Although the explicit change is made in a single block, it influences the functional behavior of the entire system in the synchronous model of computation. The proposed algorithm solves the synchronization problem and allows to use the synchronous model even at late stages of the design process. On the other hand, the proposed method is also a good candidate to refine synchronous blocks in LID or GALS model.

Our synchronization algorithm (1) does not change combinational processes, (2) does not add schedulers or controllers like they are used for process execution in pipelined systems and in data-flow models, and (3) does not introduce wrappers and handshake communication channels. This leaves the blocks simpler to analyze, verify, and to apply the further design refinements. The only resources we add to the model are the regularly placed $\perp$-delay processes. We have implemented these delays as latches at the RT-level, where the real gate delays are considered instead of the ideal computation time at a high level of abstraction. The number of the synchronization delay processes can be further reduced to a minimum by using retiming techniques.

6. REFERENCES

[1] S. Abdi and D. Gajski. Automatic generation of equivalent architecture model from functional specification. In *Proceedings of the Design Automation Conference*, pages 608–613, 2004.

[2] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In *Proceedings of the International Conference on Concurrency Theory*, pages 162–177, 1999.

[3] G. Berry, M. Kishinevsky, and S. Singh. System level design and verification using a synchronous language. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, page 433, Washington, DC, USA, 2003. IEEE Computer Society.

[4] T. Callahan and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, 2000. ACM.

[5] L. Carloni, K. McMillan, and A. SangiovanniVincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design*, 20(9), 2001 2001.

[6] L. P. Carloni and A. L. Sangiovanni-Vincentelli. A framework for modeling the distributed deployment of synchronous designs. *Formal Methods in System Design*, 28(2):93–110, 2006.

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

[8] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *Software Engineering*, 18(9):785–793, 1992.

[9] S. Huang, K. Cheng, and K. Chen. On verifying the correctness of retimed circuits. In *Proceedings of the the Great Lakes Symposium on VLSI*, 1996.

[10] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[11] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[12] N. Maheshwari and S. S. Sapatnekar. Minimum area retiming with equivalent initial states. In *Proceedings of the International Conference on Computer-Aided Design*, pages 216–219, 1997.

[13] S. Nadjm-Tehrani and J.-E. Strömberg. Formal verification of dynamic properties in an aerospace application. *Formal Methods in System Design*, 14(2):135–169, March 1999.

[14] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the International Conference on Application of Concurrency to System Design*, St Malo, France, 2005.

[15] M. Weinhardt and W. Luk. Pipeline vectorization. *In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 234–248, Feb 2001.