Optimized System Synthesis of Complex RT Level Building Blocks from Multirate Dataflow Graphs

Jens Horstmannshoff and Heinrich Meyr
Integrated Signal Processing Systems
Aachen, Germany

Abstract

In order to cope with the ever increasing complexity of todays application specific integrated circuits, a building block based design methodology is established. The system is composed of high level building blocks of which some are reused from previous designs while others might have been created by behavioral synthesis. In data flow oriented designs, these blocks usually have complex non-matching interface properties, making it necessary to generate additional interfacing and controlling hardware to integrate them into an operable system.
In this paper, an RTL-HDL code generation from a synchronous data flow representations is introduced, that efficiently automates the generation of the required additional hardware. While existing code generation approaches provide strong limitations concerning the building block interfacing properties, our method enables the integration of components that access their ports periodically with arbitrary patterns. In order to reduce interface register cost, a minimum-area retiming approach is taken to determine optimum building block activation times, which is known to have polynomial time complexity. The code generation methodology is compared to an existing approach using a simple case study.

1 Introduction

Todays ASIC designers face the problem of an exploding design complexity. At the same time, the development cycles are getting shorter in order to achieve a minimum time-to-market of the product. This pressure leads to a shift in ASIC design paradigm to speed up productivity while guaranteeing that only one design iteration is necessary to develop an operable product (first-silicon-success).
This new paradigm is based on the combination of high level building blocks that are taken from different origins [3]. To a certain degree these blocks can be reused from previous designs or purchased from third party vendors. Other blocks have to be custom made for the specific application. This involves the usage of advanced block design methods like behavioral synthesis [1] or data-path generation. In general, the high-level building blocks have non-matching complex interfacing properties that require the designer to write additional interfacing and controlling hardware that combines all blocks into an operable system.
In order to ensure a first-silicon-success of the building block based design it is important to use a seamless design flow from the algorithmic system specification to hardware implementation, that enables the the joint development of algorithm and architecture and a stepwise refinement of the initial algorithmic model. This flow allows the verification of the refined model to the more abstract algorithmic specification.
The ideal design environment to automate the seamless development of building-block based data-path oriented ASIC designs is RTL-HDL code generation from synchronous data-flow graphs [6]. Here, the algorithmic input specification is given by the data flow graph representation of the system which contains no notion of time. In the course of the code generation, the purely functional data-flow actors are mapped to complex RTL building blocks that are taken from a library. Furthermore, additional interfaces and controllers are generated to glue the given building-blocks into an operable system.
In this paper, we present a new approach of generating a building-block based target RTL architecture from synchronous data flow graph specifications. Here, a minimum area retiming transformation is used to reduce interfacing register cost. The paper is organized as follows: Section 2 summarizes the existing HDL code generation approaches from synchronous data flow representations. In Section 3 an overview of the tasks performed by our HDL code generation tool is given. After discussing the timing specification of the building blocks in section 4, the algorithms involved in the code generation are presented in section 5. Finally, a simple case study is presented in section 6.

2 Existing Approaches

Several approaches of generating hardware from synchronous data flow graphs are known to date. In [10], portions of the data flow graph are grouped into hardware execution units for which asynchronous communication is generated. Here, the granularity of the actors is restricted, so the desired combination of complex building blocks is not supported.
In [11], a library based HDL code generation method is presented that enables the integration of more complex building blocks. This approach, however, is strongly restricted concerning the I/O properties of the building blocks. Each block is assumed to read and write samples *equidistantly*, meaning that a fixed number of clock cycles elapses between each sample being read or written. This assumption is not valid for a vast number of building-block architectures which are integrated in todays communication systems.
The code generation approach presented in this paper is based on the work discussed in [2], where the building blocks are allowed to have arbitrary periodic port access patterns. Here, a straightforward approach is taken to generate the additional interfacing and controlling hardware, which can lead to a high overhead in interfacing registers. In this paper we will present a code generation approach that tremendously reduces this overhead.

3 Code Generation Overview

Algorithmic simulation of communication systems can be performed very efficiently using synchronous dataflow [6]. In dataflow, a system is represented as a directed graph, in which the nodes represent computations and the edges represent FIFO channels. These channels queue data values, encapsulated in objects called tokens, which are passed from the output of one computation to the input of another. In the dataflow model, no notion of time is present. For our code generation tool we use dataflow graphs as offered by [9] as an input specification. In the course of the code generation, the computational models contained in the dataflow nodes are mapped to RTL building blocks. Furthermore, a cycle based time scale is introduced for the data tokens transmitted over the graph's edges. Therefore, detailed information is required about the timing of each RTL building block.

The building blocks are assumed to have complex *timing-patterns* for their ports. The port timing pattern consists of a periodic sequence of timesteps (specified in multiples of clock cycles) at which data samples are consumed or produced at this port. Most data-path oriented signal processing components can be described in this manner. A detailed description of the component timing model will follow in Section 4.

The main task in system integration is the generation of additional RTL-components to glue the single components together to a working system. As depicted in Figure 1, this is done by introducing interface components and controllers which provide reset and stall signals to ensure proper building block activation. These components are now discussed with respect to their tasks.

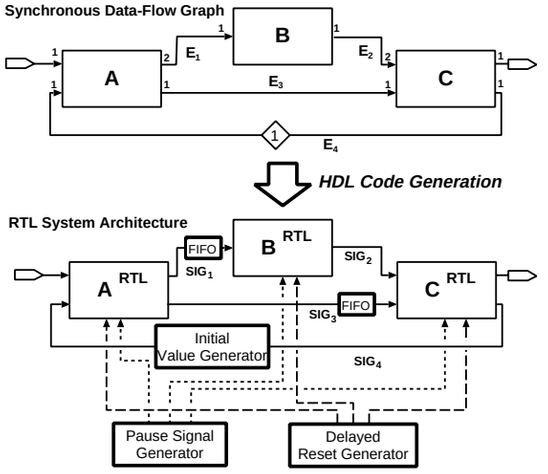

Figure 1. HDL Code Generation Scenario

- **Stall signal generation:**
  In synchronous dataflow, each component consumes and produces a fixed number of samples at its ports each time it is activated. This number is referred to as the port's *data rate*. In a block diagram with multiple rates, each component has to be activated a certain number of times within a global processing period in order to achieve *rate-consistency*. When the system is rate-consistent and free of deadlocks, it can go through an infinite number of iterations with finite memory requirements at the interfaces between its components [6]. If the number of clock cycles the building-blocks require for one system iteration is not constant for all building-blocks in the system, it becomes necessary to pause the faster ones. In the target architecture, this is realized by applying a stall signal to the corresponding blocks or by using gated clock signals.
- **FIFO buffers:**
  On certain signals in the RTL target architecture, FIFO buffers have to be inserted to adapt non-matching timing patterns of the connected building block ports or to balance the latency of merging paths. The registers required for this latency balancing are also called *shimming registers* [4].
- **Initial value generation:**
  In the system's initialization phase, some blocks have to start processing with determined initial values in order to avoid dead-locks in feedback loops. These values are produced by an initial value generator in the interface preceding those blocks.
- **Delayed reset signals:**
  After the system reset signal is deactivated, a component can start processing only after the preceding block starts to deliver valid data samples, i.e. after the writing block has finished its initialization. This delayed activation is implemented by disabling the reset signals for each component at a pre-determined point in time. Therefore, the target architecture contains a reset generator which delivers an independent reset-signal for each building-block.

In order to build this required additional hardware, the following data has to be determined:

- Number of pause cycles per building block
- Mapping of pause cycles to block schedules
- Reset deactivation delay of each building block
- Data token transfer delays

In section 5, we will present the algorithms to determine this data.

4 System Input Specification

The HDL code generation presented here is based on a data-flow representation of the system and a clock cycle true timing specification of the RTL building blocks. In the following, a formal model is introduced for this data.

4.1 Data-Flow Graph Terms

The system to be integrated is described by a directed synchronous data flow graph [6] $G^{\text{SDF}} = (\mathcal{N}^{\text{SDF}}, \mathcal{E}^{\text{SDF}})$, where $\mathcal{N}^{\text{SDF}}$ represents a set of nodes (blocks) and $\mathcal{E}^{\text{SDF}}$ stands for a set of edges. Figure 2 depicts two nodes N_i^{SDF} and N_j^{SDF} which are connected via edge E^{SDF}.

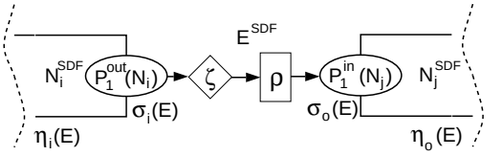

Figure 2. Definition of graph terms

Each node N_i^{SDF} has a number of input ports $P_j^{in}(N_i^{\text{SDF}})$ and output ports $P_j^{out}(N_i^{\text{SDF}})$. So, edge E^{SDF} in figure 2 can be represented by an ordered pair of ports $E^{\text{SDF}} = (P_1^{out}(N_i^{\text{SDF}}), P_1^{in}(N_j^{\text{SDF}}))$, where the input port of this edge is given by $\sigma^{in}(E^{\text{SDF}}) = P_1^{out}(N_i^{\text{SDF}})$ and the output port

by $\sigma^{out}(E^{\text{SDF}}) = P_1^{in}(N_j^{\text{SDF}})$. To denote the nodes with respect to a connected edge we introduce the input node of edge E as $\eta_{in}(E^{\text{SDF}}) = N_i^{\text{SDF}}$ and its output node as $\eta_{out}(E^{\text{SDF}}) = N_j^{\text{SDF}}$.
The number of samples which are consumed or produced on a port $P(N^{\text{SDF}})$ within one activation is given by its data-rate $r(P(N^{\text{SDF}}))$. The number of *initial values* $\zeta(E^{\text{SDF}})$ on edge E^{SDF} denotes the number of data tokens that are written to the edge output port $\sigma_o(E^{\text{SDF}})$ before the first data token is produced by the edge input port $\sigma_o(E^{\text{SDF}})$. In addition to these typical synchronous data-flow properties, the user can assign a minimum number of registers $\rho(E^{\text{SDF}})$ to an edge that will be placed on the corresponding signal in the RT level system architecture.

4.2 Building Block Interface Specification

As mentioned above, the building blocks are assumed to perform their calculations periodically after initialization. The duration of one processing period of node N_i^{SDF} in number of clock cycles is called *iteration-period* and will be denoted as $I_{node}(N_i^{\text{SDF}})$. During one iteration-period, the RTL building block consumes and writes the same number of data items at its ports as specified by the data-rates of the corresponding synchronous data flow model. In order to map these data items to the clock cycle true schedule of the RTL model, we introduce a *port time mapping vector* $\vec{\mu}(P)$ for every data port P to specify in which clock cycle within the iteration period it is accessed.If port P is an output port, the j-th element of $\vec{\mu}(P)$ represents the clock cycle index of the first valid appearance of the j-th sample in the iteration period of port P in the periodic processing phase. If P is an input port, the i-th element of $\vec{\mu}(P)$ contains the cycle in which the i-th data sample is read from the port. Since the first element of the port time mapping vector denotes the cycle index of the first access to this port, it can be seen as the latency of this port.
Some building blocks require a certain number of clock cycles between the deactivation of the reset signal and the start of the periodic I/O schedule to initialize their internal states. This duration is called *initialization-time* $\tau_{init}(N_i^{\text{SDF}})$.
Figure 3 depicts the waveforms of a resource shared down-sampling FIR filter block with registered output ports that was generated using behavioral synthesis. The numbers in the waveforms of the `INPUT` and `OUTPUT` port represent the data token index within the iteration period.

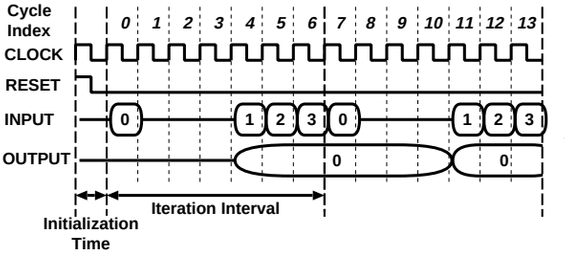

Figure 3. Waveforms of DFIR Filter

Following the deactivation of the reset signal the block requires $\tau_{init} = 1$ clock cycle to initialize its internal states. Then the block starts processing periodically with an iteration interval if $I = 7$ during which $r(\text{INPUT}) = 4$ data samples are consumed from the input port and $r(\text{OUTPUT}) = 1$ data sample is written to the output port. As depicted in figure 3, the port time mapping vectors are given by

$$\vec{\mu}(\text{INPUT}) = (0\ 4\ 5\ 6)^T \qquad \vec{\mu}(\text{OUTPUT}) = (4) \quad (1)$$

The periodicity model holds especially well for data-flow oriented components with resource sharing as they are generated by high level synthesis tools [1]. As we will see in section 5.2 it is also important to specify whether an output port is registered or whether there exists a combinatorial path from an input port. This property is crucial for the construction of the paused timing pattern.

5 Algorithms

5.1 Rate and Periodicity Consistency

The first step in synthesizing an operable RTL architecture from the synchronous data flow system is to determine the minimum *system iteration period* and the number of clock cycles each building block has to be paused in this period. The system iteration period I_{sys}^{min} describes the minimum number of clock cycles the RTL system requires for one system iteration. Therefore, it is necessary to calculate the number of iteration periods each block goes through during one system iteration. This information can be extracted from the synchronous data flow representation.
Let $q_{N_i^{\text{SDF}}}$ denote the number of times node N_i^{SDF} is activated per system iteration period, then we have to determine values for $q_{N_i^{\text{SDF}}}$ which fulfill the following *balance equations* for all edges $\forall E_j^{\text{SDF}} \in \mathcal{E}^{\text{SDF}}$

$$q_{\eta_i(E_j^{\text{SDF}})} r(\sigma_i(E_j^{\text{SDF}})) = q_{\eta_o(E_j^{\text{SDF}})} r(\sigma_o(E_j^{\text{SDF}})) \quad (2)$$

Equation 2 expresses the fact that within one system iteration period the same number of samples is produced and consumed by the ports connected to edge E_j^{SDF}. We can construct a *topology matrix* Γ that contains the integer $r(P_n^{out}(N_i^{\text{SDF}}))$ in position (j, i) if node N_i^{SDF} produces $r(P_n^{out}(N_i^{\text{SDF}}))$ samples from output port n on the edge E_j^{SDF}. If it contains the integer $-r(P_n^{in}(N_i^{\text{SDF}}))$ in position (j, i), node N_i^{SDF} consumes $r(P_n^{in}(N_i^{\text{SDF}}))$ samples on input port n from edge E_j^{SDF}. Then, the system of equations to be solved can be written as

$$\Gamma\ \vec{q}\ = \vec{0} \quad (3)$$

where $\vec{0}$ is a vector full of zeros, and $\vec{q}$ is the *repetition vector*. Now we are able to calculate the minimum system iteration period

$$I_{sys}^{min} = \max_{\forall N_i^{\text{SDF}} \in \mathcal{N}} (I_{node}(N_i^{\text{SDF}})\ q_i) \quad (4)$$

Please note that any system iteration period can be chosen depending on the system throughput requirements as long as the minimum system iteration interval is not violated. If the number of cycles a node N_i^{SDF} requires for q_i activations is smaller than the system iteration period, the component has to be paused for

$$p(N_I^{\text{SDF}}) = I_{sys}\ -\ I(N_i^{\text{SDF}})\ q_{N_i^{\text{SDF}}} \quad (5)$$

cycles in each system iteration. This measure has to be taken in order to achieve a global *periodicity consistency*.

5.2 System Timing Adjustment by Retiming

In this paper, we will use a minimum-area retiming approach to map the pause cycles $p(N_i^{\text{SDF}})$ to the building blocks I/O schedule and determine the block reset deactivation cycles, while minimizing the delay of all data token transfers in the system. This approach leads to a tremendous reduction in interface register area compared to the

approach presented in [2].
In the following, we will demonstrate how to construct a directed *retiming graph* G^{RET} from the input system specification on which the classic min-area retiming algorithm [7] can be applied.

5.2.1 Retiming Graph Construction

The directed retiming graph $G^{\text{RET}} = (\mathcal{N}^{\text{RET}}, \mathcal{E}^{\text{RET}})$ is a representation of the building block I/O schedules and the data transfers that take place in one system iteration. In general, the I/O schedule of a building block can be partitioned into multiple phases where each phase contains exactly one clock cycle in which ports are accessed. A retiming node $N_i^{\text{RET}}(N^{\text{SDF}}) \in \mathcal{N}^{\text{RET}}$ stands for the i-th phase in the I/O schedule of data flow block N^{SDF}. Every retiming node has a *reference cycle* $c(N_i^{\text{RET}}(N^{\text{SDF}}))$ that contains the index of the corresponding port access cycle in the periodic I/O schedule of building block N^{SDF}. The retiming edges $E^{\text{RET}} \in \mathcal{E}^{\text{RET}}$ are represented by ordered pairs of retiming nodes $E^{\text{RET}} = (N_i^{\text{RET}},\ N_j^{\text{RET}})$. The set of Retiming edges can be divided into two basic types $\mathcal{E}^{\text{RET}} = \mathcal{E}^{\text{RET}}(\mathcal{N}^{\text{SDF}}) \cup \mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}})$. A *schedule edge* $E^{\text{RET}}(N^{\text{SDF}}) \in \mathcal{E}^{\text{RET}}(\mathcal{N}^{\text{SDF}})$ stands for a direct sequential dependency between two phases in the I/O schedule of building block N^{SDF}. A *data transfer edge* $E_i^{\text{RET}}(E^{\text{SDF}}) \in \mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}})$ represents the transfer of the i-th data token over data flow edge E^{SDF} within one system iteration. Here, the feeding retiming node represents the I/O schedule phase in which the data token is written, while the consuming retiming node stands for the I/O schedule phase in which the token is read. Every retiming edge E^{RET} has a delay $D(E^{\text{RET}})$ and a width $w(E^{\text{RET}})$ assigned to it.
In order to construct the retiming graph $G^{\text{RET}}(G^{\text{SDF}})$ from a data flow graph G^{SDF}, we first partition the I/O schedules of all building blocks into retimable phases to determine the retiming nodes and the schedule edges. Inserting pause cycles into the periodic I/O schedule of a building block has the effect of stretching the port access patterns. Depending on where the pause cycles are inserted in the periodic I/O schedule of the block, different stretching scenarios occur.
As an example, the I/O schedule of the downsampling FIR filter in figure 3 shall be used, assuming that it has to be paused ($p(\text{dfir}) > 0$) and only activated once each system iteration ($q_{\text{dfir}} = 1$). This block can be paused by deactivating the load-enable signal of all internal registers. We partition the I/O schedule of this block into five phases where the first phase spans the first three cycles of its periodic I/O schedule. Pausing the component in any cycle of this phase will result in the same pattern stretching. The remaining four clock cycles of the I/O schedule are partitioned into four distinct phases since the pattern is stretched differently depending on which of these cycles is chosen for pausing. In the retiming graph, the five I/O phases are represented by retiming nodes $N_{0-4}^{\text{RET}}(\text{dfir})$ that are connected cyclically by retiming schedule edges $E_{0-4}^{\text{RET}}(\text{dfir})$ as depicted in figure 4. These edges represent the cyclic I/O schedule of the building block.

Figure 5 presents the algorithm that determines the set of retiming nodes $\mathcal{N}^{\text{RET}}$, their reference cycles $c(N^{\text{RET}})$ and the number of retiming nodes $\#N^{\text{RET}}(N^{\text{SDF}})$ for every building block. The algorithm iterates through all building blocks and generates a retiming node for every phase in the periodic I/O schedule of a block that contains a clock cycle in which a port access takes place. However, if a

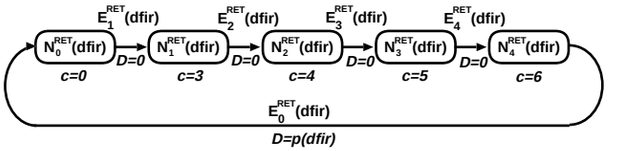

Figure 4. Retiming graph representing I/O schedule of block dfir

building block does not have to be paused ($p(N^{\text{SDF}}) = 0$), only one retiming node will be created. The function *is_port_access*(N^{SDF}, i) returns true if a port access takes place in the i-th cycle of the periodic I/O schedule of block N^{SDF}. In the case of registered output ports, the function returns true if a data item is written in cycle $i + 1$ in order to take the register delay into account.

The retiming nodes are cyclically connected by

```
1) N^RET ← ∅
2) for all N^SDF ∈ N^SDF
3)   if p(N^SDF) = 0
4)     N^RET ← N^RET ∪ {N_0^RET(N^SDF)}
6)     c(N_0^RET(N^SDF)) ← 0
7)     #N^RET(N^SDF) ← 1
8)   else
9)   j ← 0
10)    for i=0 to q_{N^SDF} I(N^SDF) − 1
11)      if is_port_access(N^SDF, i)
12)        j ← j + 1
13)        N^RET ← N^RET ∪ {N_{j−1}^RET(N^SDF)}
14)        c(N_{j−1}^RET(N^SDF)) ← i
15)      end
16)      #N^RET(N^SDF) ← j
17)    end
18)  end
19) end
```

Figure 5. Determination of retiming nodes

$\#N^{\text{RET}}(N^{\text{SDF}})$ schedule edges, where $E_0^{\text{RET}}(N^{\text{SDF}})$ always marks the schedule edge between the last $N_{\#N^{\text{RET}}(N^{\text{SDF}})}^{\text{RET}}(N^{\text{SDF}})$ and the first $N_0^{\text{RET}}(N^{\text{SDF}})$ retiming node. The pause cycles that are introduced in the I/O schedule of a building block are represented by the retiming edge delays $D(E_i^{\text{RET}}(N^{\text{SDF}}))$ assigned to schedule edge $E_i^{\text{RET}}(N^{\text{SDF}})$ of this block. As depicted in figure 4 the $p(N^{\text{SDF}})$ pause cycles are initially placed on schedule edge $E_0^{\text{RET}}(N^{\text{SDF}})$. So we can write

$$D(E_i^{\text{RET}}(N^{\text{SDF}})) = \begin{cases} 0 & : \quad i \neq 0 \\ p(N^{\text{SDF}}) & : \quad i = 0 \end{cases} \tag{6}$$

In the retiming transformation, these pause cycles are distributed over the I/O schedule to minimize data transfer delay. As we will in section 5.2.2, the width of all schedule retiming edges is

$$w(E_i^{\text{RET}}(N^{\text{SDF}})) = 0 \qquad \forall E_i^{\text{RET}}(N^{\text{SDF}}) \in \mathcal{E}^{\text{RET}}(\mathcal{N}^{\text{SDF}}) \tag{7}$$

This means that pausing the I/O schedule of a building block does not cause any register cost.
Now the set of data transfer retiming edges $\mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}})$ is constructed. The number of data tokens transferred over a data flow edge E^{SDF} in one system iteration is given by the *interface rate*

$$r_{int}(E^{\text{SDF}}) = q_{\eta_i(E^{\text{SDF}})}\ r(\sigma_i(E^{\text{SDF}})) \tag{8}$$

In the retiming graph, each data token transfer over a data flow edge E^{SDF} is represented by a data transfer edge $E^{\text{RET}}(E^{\text{SDF}})$. So the number of retiming edges per data flow edge is also determined by $r_{int}(E^{\text{SDF}})$. In order to determine the input and output retiming nodes of each data transfer edge, we need knowledge about the I/O schedule phase in which a data token is produced or consumed. This information is given in the *retiming variable mapping vector* $\vec{\phi}(P(N^{\text{SDF}}))$ that can easily be determined once the phase partitioning was performed. The i-th element $\phi_i(P(N^{\text{SDF}}))$ of this vector contains the I/O phase index in which the i-th data token is read or written on port $P(N^{\text{SDF}})$. For the down-sampling FIR filter example in figure 3, these vectors are

$$\vec{\phi}(\text{INPUT(dfir)}) = (0\ 2\ 3\ 4)^T\ ,\ \vec{\phi}(\text{OUTPUT(dfir)}) = (1)$$

Figure 6 contains the algorithm that determines the set of data transfer edges $\mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}})$, the data transfer delays $D(E^{\text{RET}}(E^{\text{SDF}}))$ and their width $w(E^{\text{RET}}(E^{\text{SDF}}))$. The algorithm generates a retiming data transfer edge for each data transfer that takes place within each system iteration. In line 5, the retiming edges are constructed by pairing up the retiming nodes that represent the producing and consuming I/O schedule phases of the corresponding data token. When determining the output retiming node of a data transfer edge $E^{\text{RET}}(E^{\text{SDF}})$, we have to consider that the first $\zeta(E^{\text{SDF}})$ input tokens that port $\sigma_o(E^{\text{SDF}})$ consumes are initial values assigned to data flow edge E^{SDF}. Therefore, the token correspondence has to be cyclically shifted by $\zeta(E^{\text{RET}})$ samples in line 4. The delay $D(E_j^{\text{RET}}(E^{\text{SDF}}))$ of a data transfer edge denotes the delay between the production of the j-th data token onto edge E^{SDF} and its consumption. The initial delay between token production and consumption is calculated in line 7, assuming that all building blocks are activated in the same clock cycle and that the pause cycles are inserted between the last and the first retiming node as stated in equation 6. Here, $d_{i,\,j}(E^{\text{SDF}})$ is the j-th element of the *edge input pattern*, while $d^{\zeta}_{o,\,j}(E^{\text{SDF}})$ stands for the j-th element of the *edge output pattern* cyclically shifted by ζ initial values. These patterns are constructed by duplicating the port time mapping vectors of the connected ports as often as the corresponding block is activated within each system iteration. The negative delay caused by the availability of initial values is represented by $\Delta\tau_{iv}(E^{\text{SDF}})$. The determination of these values is thoroughly discussed in [2]. The width of a data transfer edge equals the width of the corresponding data flow edge.

1) $\mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}}) \leftarrow \emptyset$
2) **for all** $E^{\text{SDF}} \in \mathcal{E}^{\text{SDF}}$
3) **for** j = 1 **to** $r_{int}(E^{\text{SDF}}) - 1$
4) $k \leftarrow (j + \zeta(E^{\text{SDF}}))\ \bmod r_{int}(E^{\text{SDF}})$
5) $E_j^{\text{RET}}(E^{\text{SDF}}) \leftarrow \Big(N^{\text{RET}}_{\phi_j(\sigma_i(E^{\text{SDF}}))}(\eta_i(E^{\text{SDF}})),\ N^{\text{RET}}_{\phi_k(\sigma_o(E^{\text{SDF}}))}(\eta_o(E^{\text{SDF}}))\Big)$
6) $\mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}}) \leftarrow \mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}}) \cup \{\ E_j^{\text{RET}}(E^{\text{SDF}})\}$
7) $D(E^{\text{RET}}(E^{\text{SDF}})) \leftarrow d^{\zeta}_{o,\,k} - d_{i,\,j}(E^{\text{SDF}}) + \Delta\tau_{iv}(E^{\text{SDF}}) + \tau_{init}(\eta_o(E^{\text{SDF}})) - \tau_{init}(\eta_i(E^{\text{SDF}}))$
8) $w(E^{\text{RET}}(E^{\text{SDF}})) \leftarrow w(E^{\text{SDF}})$
9) **end**
10) **end**

Figure 6. Determination of data transfer edges

5.2.2 Retiming Formulation

Based on the retiming graph constructed in the previous section, the standard min-area retiming problem can be formulated. A retiming is a labeling of the retiming graph vertices $\pi\ :\ N^{\text{RET}} \rightarrow Z$, where Z is the set of integers. The delay of retiming graph edge after retiming is denoted by $D_\pi(E^{\text{RET}})$ and given by

$$D_\pi(E^{\text{RET}}) = \pi(\eta_o(E^{\text{RET}})) + D(E^{\text{RET}}) - \pi(\eta_i(E^{\text{RET}})) \quad (9)$$

The retiming label $\pi(N^{\text{RET}})$ of a retiming node N^{RET} represents the number of delays moved from its outgoing retiming edges to its incoming retiming edges. Moving a delay to a data transfer edge $E_j^{\text{RET}}(E^{\text{SDF}})$ means that the delay between the production and consumption of the j-th data token via data flow edge E^{SDF} is enlarged. Moving a delay to a schedule edge $E^{\text{RET}}(N^{\text{SDF}})$ introduces a pause cycle between the I/O schedule phases embodied by the input and output retiming node of this edge.
It is our objective to minimize the data transfer delay weighted with the width of the transferred data tokens, in order to reduce interface register cost. A retiming $\pi(N^{\text{RET}}) = 1$ on retiming node N^{RET} contributes an increment of $\Delta C(N^{\text{RET}})$ to the cost function

$$\Delta C(N^{\text{RET}}) = \sum^{\mathcal{E}^{\text{RET}}_{in}(N^{\text{RET}})} w(E^{\text{RET}}) - \sum^{\mathcal{E}^{\text{RET}}_{out}(N^{\text{RET}})} w(E^{\text{RET}}) \quad (10)$$

Here, $\mathcal{E}^{\text{RET}}_{in}(N^{\text{RET}})$ represents the set of incoming retiming edges, while $\mathcal{E}^{\text{RET}}_{out}(N^{\text{RET}})$ stands for the set of outgoing retiming edges with respect to retiming node N^{RET}. Please note that all schedule edges are zero width as shown in equation 7, since no cost is caused by pausing a building block. When formulating the retiming as an ILP problem, the objective function to be minimized is given by

$$\min \sum^{\forall N^{\text{RET}} \in \mathcal{N}^{\text{RET}}} \Delta C(N^{\text{RET}})\pi(N^{\text{RET}}) \quad (11)$$

In order to achieve a *valid* retiming, it has to be ensured that the retiming graph edge delay after retiming does not become negative. A negative delay on a data transfer edge implies that a data token is read from a data flow edge before it has been produced, while the negative delay on a schedule edge means that the block schedule is shortened. For the I/O schedule edges this non-negativity constraint can written as

$$\pi(\eta_i(E^{\text{RET}})) - \pi(\eta_o(E^{\text{RET}})) \leq D(E^{\text{RET}})$$
$$\forall E^{\text{RET}} \in \mathcal{E}^{\text{RET}}(\mathcal{N}^{\text{SDF}}) \quad (12)$$

where $\eta_i(E^{\text{RET}})$ denotes the start retiming node of edge E^{RET} and $\eta_o(E^{\text{RET}})$ marks the end retiming node of this edge. When constructing the constraints for the data transfer edges $E^{\text{RET}}(E^{\text{SDF}})$, we have to consider that the delay must never become smaller than the minimum delay $\rho(E^{\text{SDF}})$ that the user assigned to the corresponding data flow edge E^{SDF}. This leads to

$$\pi(\eta_i(E^{\text{RET}})) - \pi(\eta_o(E^{\text{RET}})) \leq D(E^{\text{RET}}) - \rho(E^{\text{SDF}})$$
$$\forall E^{\text{RET}} \in \mathcal{E}^{\text{RET}}(\mathcal{E}^{\text{SDF}}) \quad (13)$$

The ILP problem given by equations 11, 12 and 13, resembles the unconstrained min-area retiming problem as formulated in [7]. The dual of this ILP problem is a min-cost flow network problem which can be solved efficiently in polynomial time [5].

In order to ensure feasibility of the retiming ILP problem, it has to be guaranteed that the sum of delays along all data-transfer cycles in the retiming graph is smaller or equal zero. If this is the case, the network optimization algorithm will always find a valid retiming solution.

5.2.3 Retiming Results

After solving the retiming problem formulated in section 5.2.2, we are able to determine all data that is necessary to generate the required additional hardware. The *reset deactivation time* $t_{ro}(N^{\text{SDF}})$ denotes the delay in clock cycles between system reset deactivation and the reset deactivation of block N^{SDF}. This value is given by the retiming labeling of the first retiming node of the corresponding block

$$t_{ro}(N^{\text{SDF}}) = \pi(N_0^{\text{RET}}(N^{\text{SDF}})) \quad (14)$$

In the RTL target architecture, the reset generator will provide an independent reset signal for each block N^{SDF}, which is delayed by $t_{ro}(N^{\text{SDF}})$ clock cycles.

The periodic pausing pattern that the pause signal generator has to provide for each building block N^{SDF} can be determined from the schedule edge delays after retiming $D_\pi(E^{\text{RET}}(N^{\text{SDF}}))$ as given by equation 9 and the reference cycles of the retiming nodes $c(N^{\text{RET}}(N^{\text{SDF}}))$. A schedule edge delay of $D_\pi(E_j^{\text{RET}}(N^{\text{SDF}}))$ after retiming means that the periodic schedule of block N^{SDF} has to be paused for $D_\pi(E_j^{\text{RET}}(N^{\text{SDF}}))$ cycles in the $c(N_j^{\text{RET}}(N^{\text{SDF}}))$-th clock cycle of its periodic processing phase.

The number of registers required to implement the FIFO buffer on a data flow edge is given by the maximum number of overlapping token transfers. By minimizing the delays of these transfers in the retiming transformation, the maximum overlap is tremendously reduced compared to existing approaches.

6 Case Study

The carrier synchronization unit [8] depicted in figure 7 is used as an example to compare the register cost produced by the approach presented here to the approach taken in [2]. The RTL architectures that correspond to the functional models of the data flow models have non-matching port I/O patterns and different processing periodicities, requiring the RTL-HDL code generation to produce additional interfacing and controlling hardware in order to synthesize all blocks into an operable system. In table 1 the number

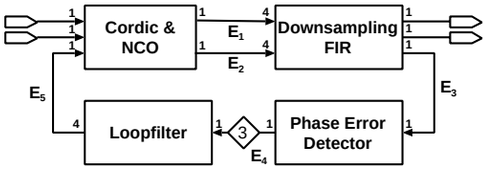

Figure 7. SDF Graph of Carrier Synchronizer

of interfacing registers inserted between the building block ports are compared for two code generation approaches. The variable N_R represents the number of word level registers, while $N_R\ w$ denotes the register cost in one-bit wide registers. In [2], periodicity adjustment is performed by arbitrarily inserting the pause cycles into the building block I/O schedules, while the reset deactivation time and shimming register cost is determined by using a shortest path algorithm on a timed graph. As we can see, interfacing registers have to be inserted on the signals that correspond to

	Approach I		Approach II	
	N_R	$N_R\ w$	N_R	$N_R\ w$
E_1	3	12	0	0
E_2	2	8	0	0
E_3	0	0	0	0
E_4	0	0	1	5
E_5	4	68	0	0
Total	9	88	1	5

Table 1. Register Cost of Carriersynchronizer

data flow edges E_1, E_2 and E_5, leading to a total register cost of 88 single-bit registers. Approach II represents the optimized code generation algorithm presented in this paper. By using a retiming transformation, to map the pause cycles to the building blocks I/O schedules and to determine the block reset deactivation times, register cost is decreased tremendously. Only one 5-bit wide register is required on edge E_4, resulting in a interfacing register cost reduction by a factor of 17. Please note that the complexity of the generated controller does not significantly increase when changing the pause cycle mapping or the delayed reset deactivation pattern.

7 Summary

In this paper, we presented an HDL code generation approach from synchronous data flow graphs that enables the seamless building block based design of data-flow oriented systems. By employing a min-area retiming technique to minimize data transfer delays, the interface register cost is significantly reduced compared to existing approaches.

In future work, we will focus on the support of data-dependent communication and on providing an interface to top level control dominated systems.

References

[1] R. Camposano. A review of hardware synthesis techniques - behavioral synthesis. In G. D. Micheli, editor, *Hardware/Software Co-Design*, 1996.

[2] J. Horstmannshoff, T. Grötker, and H. Meyr. Mapping Multirate Dataflow to Complex RT Level Hardware Models. In *ASAP*. IEEE, 1997.

[3] M. Hunt and J. Rowson. Blocking in a system on a chip. *IEEE Spectrum*, November 1996.

[4] H. Jagadish and T. Kailath. Obtaining Schedules for Digital Systems. *IEEE Transactions on Signal Processing*, 39, November 1991.

[5] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Rinehart & Winston, 1976.

[6] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, September 1987.

[7] C. Leiserson, F. Rose, and J. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Proceedings of the 3rd Caltech Conference on VLSI*, pages 87–116. ACM, 1991.

[8] H. Meyr, M. Moeneclaey, and S. Fechtel. *Digital Communication Receivers*. John Wiley and Sons, 1998.

[9] SYNOPSYS. *COSSAP Block Diagram Editor Users Guide*, 1999.

[10] M. C. Williamson and E. A. Lee. Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications. November 3-6 1996.

[11] P. Zepter, T. Grötker, and H. Meyr. Digital Receiver Design using VHDL Generation from Data Flow Graphs. In *Proc. 32nd Design Automation Conf.*, June 1995.