Efficient Building Block Based RTL Code Generation from Synchronous Data Flow Graphs

Jens Horstmannshoff
Integrated Signal Processing Systems
RWTH Aachen, Germany

Heinrich Meyr
Integrated Signal Processing Systems
RWTH Aachen, Germany

ABSTRACT

This paper presents a RTL-HDL code generation from synchronous data-flow graphs which supports the building block based design of data-flow oriented ASIC systems. Here, additional interfacing and controlling hardware is generated to adapt non-matching interfacing properties. In order to reduce interface register cost, a retiming approach is taken to schedule optimum building block activation times. The code generation methodology is compared to an existing approach using different case studies.

1. INTRODUCTION

As the complexity of todays application specific integrated circuits steadily increases, a building block based design methodology is established in order to meet aggressive time-to-market constraints [2]. Here, the design is partitioned into several functional units called building blocks. Each functional building block is independently implemented using different paths to implementation. Eventually, all blocks are combined into the desired system architecture. Due to the increasing block reuse the I/O properties of connected building blocks might be non-matching, requiring the system designer to create additional interfacing hardware to combine all blocks into an operable system.

A very efficient way, of automating the seamless development of building-block based data-path oriented ASIC designs is RTL-HDL code generation from synchronous data-flow graphs as depicted in figure 1. Synchronous data-flow graphs [4] are widely used for algorithm design of data flow oriented designs like wireless digital receiver structures [6]. At this level of specification, no notion of time is present which enables fast algorithmic simulation. 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. Due to the possibly non-matching interfacing properties of the blocks, additional interfacing and controlling hardware has to be generated to compose all blocks into an operable system. Here, the interfaces contain FIFO queues to adapt non-matching data-transfer pattern and provide initial values in the system startup phase. The controller provides independent enable signals for each building block.

In this paper, we present an RTL-HDL code generation from synchronous data-flow graphs that supports the building block based design of data-flow oriented ASIC systems. Here, a retiming approach is used to schedule building block activations which leads to a tremendous reduction in interface register area compared to previous work [1].

The paper is organized as follows: Section 2 summarizes the existing HDL code generation approaches from synchronous data flow representations. In section 3, the input specifications for our HDL code generation are presented. Section 4 discusses the algorithms involved in the code generation . Finally, some experimental results are shown in section 5.

2. PREVIOUS WORK

Several approaches of generating hardware from synchronous data flow graphs are known to date. In [8], 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 [9], 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 [1], 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.

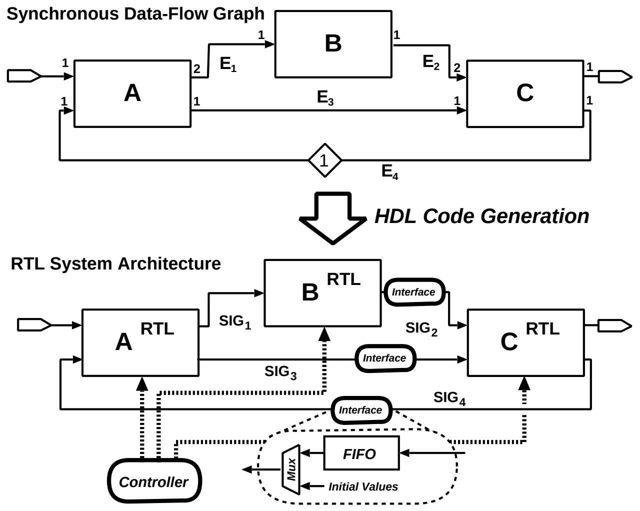

Figure 1: Code Generation Scenario

3. INPUT SPECIFICATION

3.1 Synchronous Data-flow Graphs

At the algorithmic level, the system is represented by a synchronous data-flow graph [4][7] $G^{\text{SDF}} = (\mathcal{N}^{\text{SDF}}, \mathcal{E}^{\text{SDF}})$, which serves as an input specification for our code generation. In data-flow, a system is represented as a directed graph, in which the set of nodes $\mathcal{N}^{\text{SDF}}$ stands for computations, while the set of edges $\mathcal{E}^{\text{SDF}}$ represents 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. To clarify our terminology, figure 2 depicts two nodes N_i^{SDF} and N_j^{SDF} which are connected via edge E^{SDF}. Each node N_i^{SDF} has a number of input ports $P_j^{in}(N_i^{\text{SDF}})$ and output

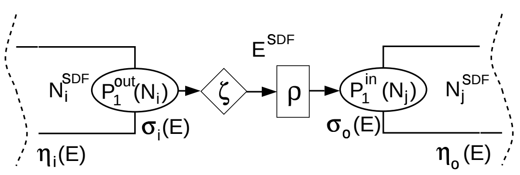

Figure 2: Definition of graph terms

ports $P_j^{out}(N_i^{SDF})$. So, edge E^{SDF} in figure 2 can be represented by an ordered pair of ports $E^{SDF} = (P_1^{out}(N_i^{SDF}), P_1^{in}(N_j^{SDF}))$, where the input port of this edge is given by $\sigma^{in}(E^{SDF}) = P_1^{out}(N_i^{SDF})$ and the output port by $\sigma^{out}(E^{SDF}) = P_1^{in}(N_j^{SDF})$. To denote the nodes with respect to a connected edge we introduce the input node of edge E as $\eta_{in}(E^{SDF}) = N_i^{SDF}$ and its output node as $\eta_{out}(E^{SDF}) = N_j^{SDF}$.

In synchronous data-flow, 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* $r(P(N^{SDF}))$. The number of *initial values* $\zeta(E^{SDF})$ on edge E^{SDF} denotes the number of data tokens that are written to the edge output port $\sigma_o(E^{SDF})$ before the first data token is produced by the edge input port $\sigma_o(E^{SDF})$.

3.2 Building Block I/O Model

The synchronous data-flow model of the system contains no notion of time. When the purely functional data-flow actors are mapped to RTL architectures from a building block library, we have to map the block communication to a clock cycle true timescale. This enables us to perform the communication analysis necessary for HDL code generation.

The building blocks are assumed to access their ports in *port timing-patterns.* A port timing pattern consists of a periodic sequence of time steps (specified in multiples of clock cycles) at which data samples are consumed or produced at this port. Figure 3 depicts the port access waveforms of a resource shared downsampling FIR filter architecture, from which the port timing patterns can be derived.

The duration of one pattern period is given by the *block iteration period* and is denoted by $I_{node}(N^{SDF})$. In figure 3 the iteration period is given by $I_{node}(DFIR2) = 7$. During one block 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. The j-th element of $\vec{\mu}(P)$ represents the clock cycle index of the first valid access to port port P after block initialization. The DFIR2 block in figure 3, consumes $r(\text{INPUT}) = 4$ data samples and produces $r(\text{OUTPUT}) = 1$ data samples per block iteration period. The waveforms of the INPUT and OUTPUT ports in figure 3 are marked by the index of the currently transmitted data token within the block iteration period. So the port mapping vectors are given by

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

In the following, we will graphically represent the port timing patterns by a line of boxes, where each box stands for a clock cycle in the processing of the I/O schedule of the corresponding port. Figure 4 c) shows this representation for the waveforms displayed in figure 3. A shaded box implies a port access in the corresponding cycle. Different box shadings are used to differentiate between the production or consuption of separate data tokens on a single data port.

4. ALGORITHMS

In this section, we discuss the algorithms that are necessary to calculate all parameters for the efficient generation of the the additional top level glue hardware as shown in figure 1. This includes a controller that supplies independent initialization and pausing signals for each building block. Furthermore, interfaces have to be

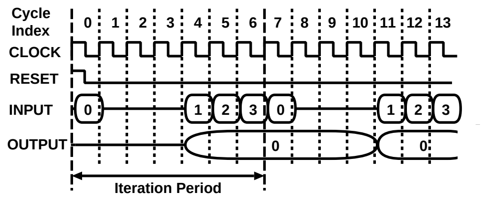

Figure 3: Waveforms of DFIR2 Filter Architecture

generated that contain FIFO registers for mismatched timing patterns and unbalanced merging paths. If a data-flow edge contains initial values, the corresponding interfaces also have to provide these values before the first valid output sample of the feeding block is available.

All algorithmic steps are demonstrated using the simple example system whose synchronous data flow graph model is depicted in figure 4 a). This system contains two downsampling filter blocks DFIR1 and DFIR2. The I/O schedules of the architectures that are used to implement the data-flow actors are depicted in figure 4 b) and 4 c). The I/O schedule of block DFIR2 has already been discussed in section 3.2.

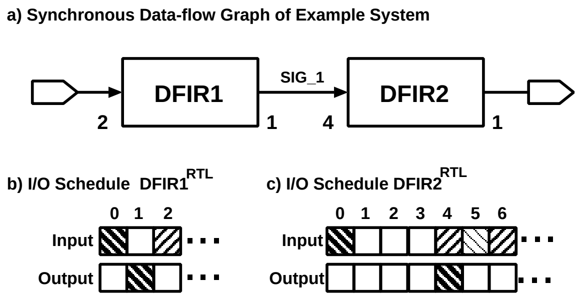
a) Synchronous Data-flow Graph of Example System
DFIR1
SIG_1
DFIR2
2
1
4
1
b) I/O Schedule DFIR1^RTL
c) I/O Schedule DFIR2^RTL
0 1 2
0 1 2 3 4 5 6
Input
Output
Input
Output

Figure 4: Example System

4.1 Block Pausing

The first step in analyzing the RTL system communication 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 minimum system iteration period I_{sys}^{min} describes the minimum number of clock cycles the RTL system requires for one system iteration. This value also marks the upper throughput bound of the RTL system architecture with the given building blocks. To calculate I_{sys}^{min}, we first have to determine the number of iteration periods each block goes through during one system iteration. This information can be extracted from the synchronous data-flow graph [4].

Let $q_{N_i^{SDF}}$ denote the number of times node N_i^{SDF} is activated per system iteration period, then we have to determine values for $q_{N_i^{SDF}}$ which fulfil the set of equations given by

$$\Gamma\,\vec{q} = \vec{0} \tag{2}$$

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

$$I_{sys}^{min} = \max_{\forall N_i^{SDF} \in \mathcal{N}} \left(I_{node}(N_i^{SDF})\, q_i\right) \tag{3}$$

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

$$p(N_I^{SDF}) = I_{sys} - I(N_i^{SDF})\, q_{N_i^{SDF}} \tag{4}$$

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

In the example system depicted in figure 4, block DFIR1 has to be activated three times each system iteration while DFIR2 only needs to be activated once. This leads to a minimum system iteration period of $I_{sys}^{min} = 12$. When choosing the minimum system iteration period, block DFIR2 has to be paused for 5 clock cycles each system iteration.

4.2 Scheduling by Retiming

For the generation of the controller, we have to schedule the building block activations. This involves the determination of the *block initialization time* $t_{ro}(N^{\text{SDF}})$, which denotes the number of clock cycles that elapse between system initialization and the initialization of the building block that implements data-flow actor N^{SDF}. Furthermore, we have to determine a mapping for the $p(N^{\text{SDF}})$ pause cycles to the I/O schedule of the building block. In this paper, we use a retiming approach [5] to perform this scheduling task.

In the following, we will demonstrate how to construct a directed *retiming graph* G^{RET} from the data flow input system specification that represents the periodic communication of the system. On this graph representation, retiming will be performed.

4.2.1 Phase Partitioning

The retiming graph $G^{\text{RET}} = (\mathcal{N}^{\text{RET}}, \mathcal{E}^{\text{RET}})$ is a directed graph, where each node N^{RET} stands for a phase of a block I/O schedule that has to be scheduled. So the first step in constructing the retiming graph is to find a phase partitioning for each building block that reflects the scheduling possibilities for this block. Here, we have to differ between two cases

- Unpaused Building Blocks $p(N^{\text{SDF}}) = 0$
  The I/O schedule of these blocks only consists of one phase, since only the block initialization time has to be determined. This is the case for block DFIR1 in figure 4.

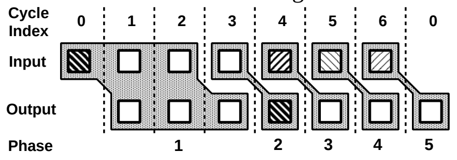

Figure 5: Phase Partitioning of Block DFIR2

- Paused Building Blocks $p(N^{\text{SDF}}) \neq 0$
  In these cases, we have to determine a pause mapping in addition to the block initialization time. So a phase partitioning has to be found that reflects all possible pause cycle mappings. This phase partitioning is done by iterating through the q times replicated I/O schedule of the paused building blocks and creating a new schedule phase for each *port access cycle*.
  For block DFIR2 in the example system of figure 4, this results in the phase partitioning depicted in figure 5. The I/O schedule of DFIR2 is partitioned into five phases that separate its port access cycles. Please note that the OUTPUT port is registered, which means that the port access cycle of the output data sample is cycle 3.

4.2.2 Construction of Retiming Graph

The set of retiming nodes $\mathcal{N}^{\text{RET}}$ is composed of the block schedule nodes and an initialization node $N^{\text{RET}}_{\text{init}}$ that represents the system initialization. The nodes are connected by a set of retiming edges $\mathcal{E}^{\text{RET}}$. The retiming edges $E^{\text{RET}} \in \mathcal{E}^{\text{RET}}$ are represented by ordered pairs of retiming nodes $E^{\text{RET}} = (N^{\text{RET}}_i, N^{\text{RET}}_j)$. All retiming edges are labeled with the edge delays $D(E^{\text{RET}})$ and the edge widths $w(E^{\text{RET}})$. The edge set $\mathcal{E}^{\text{RET}}$ can be partitioned into three subsets

- Schedule edges $\mathcal{E}^{\text{RET}}_{\text{sch}}$
  A schedule edge $E^{\text{RET}}_i(N^{\text{SDF}}) \in \mathcal{E}^{\text{RET}}_{\text{sch}}$ stands for a direct sequential dependency between two phases of an I/O schedule. All schedule nodes of a building block N^{SDF} are cyclically connected by schedule edges to reflect the periodic execution of the I/O schedule phases. The delay of a schedule edge represents the number of pause cycles that are placed between the schedule phases represented by the connected nodes. The width of a schedule edge is always zero.
- Data transfer edges $\mathcal{E}^{\text{RET}}_{\text{dt}}$
  A data transfer edge $E^{\text{RET}}_i(E^{\text{SDF}}) = (N^{\text{RET}}_i, N^{\text{RET}}_j)$ stands for a data token transfer between the I/O schedule phases represented by nodes N^{RET}_i and N^{RET}_j. Here, the edge delay denotes the delay between the production of a data token from the feeding block port $\sigma_i(E^{\text{SDF}})$ onto edge E^{SDF} and its consumption from the consuming block port $\sigma_o(E^{\text{SDF}})$. The width of a data transfer edge represents the word length of the data items that are being transfered. Please note that initial values assigned to a data-flow edge lead to a cyclic shift in the consuming retiming node correspondence.
- Initialization edges $\mathcal{E}^{\text{RET}}_{\text{init}}$
  An initialization edge $E^{\text{RET}}_{\text{init}}(N^{\text{SDF}}) = (N^{\text{RET}}_{\text{init}}, N^{\text{RET}}_0(N^{\text{SDF}}))$ represents the sequential dependency between system initialization and the first schedule phase of block N^{SDF}. The delay of these edges represent the initialization time of block N^{SDF}, so we can write
  $$t_{ro}(N^{\text{SDF}}) = D(E^{\text{RET}}_{\text{init}}(N^{\text{SDF}})) \qquad (5)$$
  The width of these edges is always zero.

Figure 6 depicts the retiming graph of the example system from figure 4. Here, the node label DFIR*i*_*j* represents the *j*-th schedule phase of block DFIR*i*. Initialization edges and schedule edges are solid, while data transfer edges are dotted.

For retiming we arbitrarily choose an initial system schedule where

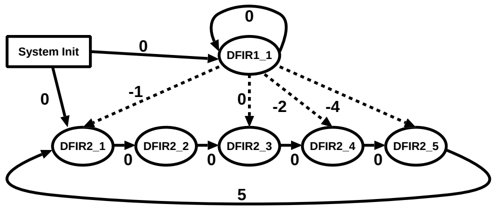

Figure 6: Retiming Graph of Initial Setup

all building blocks are activated at system initialization time. So the edge delay for all initialization edges is zero. Furthermore, we insert all pause cycles at the end of the $q_{N^{\text{SDF}}}$ block iterations within each system iteration period. This implies that the schedule edge which connects the last schedule phase of each block N^{SDF} to the first schedule phase has a delay of $p(N^{\text{SDF}})$, while all other schedule edge delays are zero. The initial data transfer delay of the i-th data transfer over edge E^{SDF} is given by

$$\begin{aligned} D(E^{\text{RET}}_i(E^{\text{SDF}})) &= \left\lfloor \frac{i+\zeta}{r^{out}} \right\rfloor I^{out} + \mu^{out}_{(i+\zeta) \bmod r^{out}} \\ &- \left\lfloor \frac{i}{r^{in}} \right\rfloor I^{in} - \mu^{in}_{i \bmod r^{in}} \qquad (6) \end{aligned}$$

where r^{in}/r^{out} represents the data rate of the edge input/output port while I^{in}/I^{out} stand for the iteration period of the edge input/output node. The vectors $\vec{\mu}^{in}/\vec{\mu}^{out}$ denote the port timing patterns of the edge input/output ports.

For the example system in figure 4, this results in the pattern correspondence on SIG_1 as depicted in figure 7.

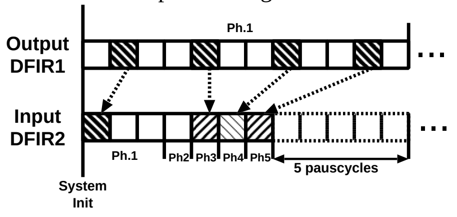

Figure 7: Initial Communication over Edge SIG_1

4.2.3 Retiming Formulation

Based on the retiming graph constructed in the previous section, the retiming problem [5] 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 a retiming graph edge after retiming is denoted by $D_\pi(E^{\text{RET}})$ and given by

$$D_\pi(E^{\text{RET}}) = \pi(\eta_o(E^{\text{RET}})) + D(E^{\text{RET}}) - \pi(\eta_i(E^{\text{RET}})) \qquad (7)$$

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. 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.
It is our objective to minimize the data transfer delay weighted with the width of the transfered 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

$$\Delta C(N^{\text{RET}}) = \sum^{\mathcal{E}_{in}^{\text{RET}}(N^{\text{RET}})} w(E^{\text{RET}}) - \sum^{\mathcal{E}_{out}^{\text{RET}}(N^{\text{RET}})} w(E^{\text{RET}}) \qquad (8)$$

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

$$\min \sum^{\forall N^{\text{RET}} \in \mathcal{N}^{\text{RET}}} \Delta C(N^{\text{RET}}) \pi(N^{\text{RET}}) \qquad (9)$$

In order to achieve a *valid* retiming, it has to be ensured that the retiming graph edge delay after retiming does not become negative.

$$\pi(\eta_i(E^{\text{RET}})) - \pi(\eta_o(E^{\text{RET}})) \leq D(E^{\text{RET}}) \qquad (10)$$

The ILP problem given by equations 9 and 10, resembles the unconstrained min-area retiming problem as formulated in [5]. The dual of this ILP problem is a min-cost flow network problem which can be solved efficiently in polynomial time [3]. In order to ensure feasibility of the retiming ILP problem, it has to be guaranteed that the sum of delays along all cycles in the retiming graph is smaller or equal zero.

4.2.4 Retiming Solution

After solving the retiming problem formulated in section 4.2.3, we are able to determine all data that is necessary to generate the required additional hardware. The block initialization times are given by the initialization edge delays $D_\pi(E_{\text{init}}^{\text{RET}}(N^{\text{SDF}}))$ as shown in equation 5. The pause cycle mapping can directly be derived from the schedule edge delays $D_\pi(E_i^{\text{RET}}(N^{\text{SDF}}))$. 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.
The pattern correspondence on SIG_1 of the retimed example system in figure 4 is shown in figure 8. Block DFIR2 is initialized one clock cycle after system initialization. One pause cycle was moved between phase 3 and phase 4, while two pause cycle were place between phase 4 and phase 5. In this schedule, no registers are required on edge SIG_1 since no data transfer overlap takes place.

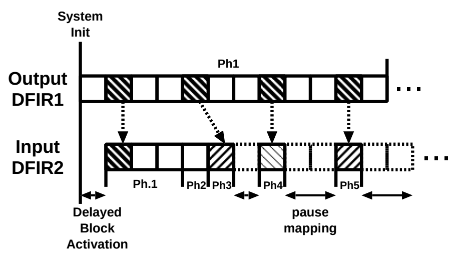

Figure 8: Retimed Communication over Edge SIG_1

5. EXPERIMENTAL RESULTS

Table 1 shows the code generation cost for different designs. Here, C denotes the total number of one bit wide registers used in the interface FIFOS, while ΔA represents the area overhead imposed by the generated controller and interfaces. The retiming based approach presented in this paper is compared to two other scheduling methods. Approach [1] performs periodicity adjustment by arbitrarily inserting the pause cycles into the building block I/O schedules, while initialization time and register placement is determined using a shortest path algorithm on a timed graph. Furthermore, a schedule with minimum register cost was determined using a branch-and-bound algorithm. Due to the large search space of the scheduling problem this method leads to intolerable run times.
In the Carriersync design, the retimed scheduling results in an interface register cost reduction by a factor of 8 compared to approach [1]. Although the optimum solution is not reached, the increase in area overhead is just 0.8%. In design DmodFX, the retiming approach leads to an optimum solution with an area overhead of 0.6%. In the lattice design, interface register cost was reduced by two orders of magnitude compared to approach [1]. It is interesting to notice that approach [1] could not find a feasible solution in the Triplefeedback design, due to the fixed pause locations.
In general, we can say that the retiming based code generation approach produces very efficient designs in polynomial time that are close to the optimum solution.

Design	Appr. [1]		Retiming		Optimum	
	C	ΔA	C	ΔA	C	ΔA
Carriersync	42	11.9%	5	4.0%	0	3.2%
DmodFX	448	18.5%	0	0.6%	0	0.6%
Lattice	3832	-	36	-	12	-
Tiplefeedback	infeasible		24	-	11	-
$C =$ Interface Register Cost, $\Delta A = A_{int\&contr}/A_{blocks}$						

Table 1: Experimental Results

6. 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. After mapping purely functional data-flow actors to RTL building blocks from an existing library, additional interfacing and controlling hardware is created to integrate an efficient RTL top level architecture. By employing a retiming technique to minimize data transfer delays, the interface register cost is significantly reduced compared to existing approaches.

7. REFERENCES

[1] J. Horstmannshoff, T. Grötker, and H. Meyr. Mapping Multirate Dataflow to Complex RT Level Hardware Models. In *ASAP*. IEEE, 1997.
[2] M. Hunt and J. Rowson. Blocking in a system on a chip. *IEEE Spectrum*, November 1996.
[3] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Rinehart & Winston, 1976.
[4] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, September 1987.
[5] 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.
[6] H. Meyr, M. Moeneclaey, and S. Fechtel. *Digital Communication Receivers*. John Wiley and Sons, 1998.
[7] SYNOPSYS. *COSSAP Block Diagram Editor Users Guide*, 1999.
[8] M. C. Williamson and E. A. Lee. Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications. November 3-6 1996.
[9] 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.