2011 14th Euromicro Conference on Digital System Design

A Scalable Distributed Asynchronous Control Network for High Level Synthesis of Digital Circuits

Tom van Leeuwen, Rene van Leuken
Circuits and Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, The Netherlands
t.g.r.m.vanleuken@tudelft.nl

***Abstract*—This paper presents a scalable asynchronous distributed control network. The control circuit allows for true asynchronous operation of all digital resources and as a result of its scalable distributed topology allows unlimited resource sharing. We start with the description of a data flow graph, and using traditional scheduling algorithms, generate an asynchronous distributed control network and the asynchronous data path. The distributed controllers are implemented such that they can be created by connecting a small number of pre-designed sub-controllers which are presented in this paper. Prototype IP-blocks of these sub-controller circuits have been designed in a 90nm ASIC design process. To prove the effectiveness of our method, we present some key performance parameters: area and power under timing constraints.**

***Index Terms*—Asynchronous circuits, Application Specific Integrated Circuits, Design Automation, Logic Design, Controller Network, De-synchronization**

I. Introduction

MOST digital circuits use a clock signal to synchronize operations, called synchronous circuits. Although this clock signal makes the design convenient, especially since practically all commercial synthesis tools assume synchronous designs, some advantages can be exploited when using asynchronous circuits. Those advantages can include typical case performance, low power consumption, less sensitive to variability, lower EMI admittance and protection against power analysis attacks. Disadvantages of asynchronous circuits include the lack of synthesis tools, their sensitivity to hazards and in some cases performance loss.

We present a controller network for asynchronous circuits which can be created automatically, including the asynchronous data-path, given a data flow description.

II. Related work

Behavioral synthesis is widely explored in the past, mostly targeting synchronous circuits. Scheduling, the process of allocating operations to time slots, is a well-known method for behavioral synthesis. A large number of scheduling algorithms are available, as well as control network topologies. For behavioral synthesis of asynchronous circuits, a number of methods for scheduling and resource allocation are published [1] [2]. However, these publications do not include the synthesis of the control network. Behavioral synthesis methods for asynchronous circuits including the control network synthesis are also published. In [3], distributed controllers for asynchronous scheduled data flow graphs are proposed, similar to our method, but each distributed controller is specified in a separate Signal Transition Graph (STG). STG's are hard to synthesize because they should operate hazard free. As a consequence, this method fails already on medium size circuits. In our method, only a few small STG's have to be synthesized, which can then be reused to create the larger complex distributed controller.

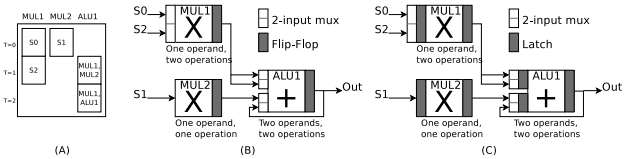

Fig. 1. FIR3 filter, a: Resource mapping, b: Data-path with flip-flops, c: Data-path with latches

III. Background

A. Data Flow Graph

A Data Flow Graph (DFG) represent operations and their data-dependencies. We use a DFG as input for our synthesis method. Using existing scheduling and resource allocation algorithms, the operations are assigned to time slots and resources. In Figure 1a, the resource allocation of a FIR filter (FIR3) is shown. Note that scheduling algorithms can assign multiple cycles to an operation, approximating asynchronous behavior where the delays of operations are independent from each other [2].

Each resource is scheduled to execute a number of different operations from the DFG. In synchronous circuits, this is handled by a multiplexer (MUX) at the input of each resource. A flip-flop with an enable signal on its output makes sure the data is available as long as required by the scheduling result. If two operations are scheduled to resource A, ax and ay, during cycle 1 and 3, the results of operation ax is available during

978-0-7695-4494-6/11 $26.00 © 2011 IEEE
DOI 10.1109/DSD.2011.114

IEEE computer society

cycle 2 and 3, and the result of operation ay is available from cycle 4 onwards.

There are a number of required properties on the scheduling result, to make the synchronous data-path possible:

- A result of an operation can only be used after it is produced. An operation X that has a data-dependency from operation Y in the DFG should be scheduled at least one time slot later than operation Y.
- A result should be available until the last operation that depends on it has consumed it. If the results from operation X have data-dependencies to Y, the resource which executes operation X cannot execute a new operation in a time slot earlier than the time slot in which Y is executed. (unless a register is used, which acts as a new resource)

B. Signal Transition Graphs

Signal Transition Graphs (STG's) are a subset of Petri nets where all transitions are signal transitions [4]. STG's are used to model Speed-Independent controllers. Speed-Independent circuits operate hazard-free under certain assumptions [5]. A STG contains transitions, places and directed edges which can connect a transition and a place in both directions. A directed edge cannot connect two places or two transitions. Every place can contain a token. A transition is enabled when all input places (places with an edge to the transition) contain a token. A transition can be an input transition which *can* be fired by the environment when enabled, or a transition of an output or internal signal (non-input transition) which *will* be fired by the circuit when it is enabled. If a transition is fired, the tokens from the input places are removed and a token is added to each of its output places.

To simplify the drawings, a place can be made implicit when it has exactly one incoming and one outgoing edge. The two edges and the place are then replaced by one edge between two transitions. This edge can now contain a token.

Marked Graphs (MG) are a subset of STG's, where each place has exactly one incoming and one outgoing edge. When drawing a MG, all places are usually implicit.

Directed circuits are a closed cycle in a MG where the direction of the arcs is respected.

A strongly connected MG is a MG which is strongly connected when there is a path from each transition in the graph to every other transition.

C. De-synchronization

De-synchronization is the process of replacing all flip-flops for latches and the clock tree for latch controllers. This method is proposed by Cortadella et all [6]. Replacing the flip-flops for latches is a technique also used in synchronous designs.

We use Marked Graphs to model the operation of the latches. Marked Graphs are also used by Cortadella to prove that the de-synchronization method is valid. We use Marked Graphs to prove the two properties, liveness and flow equivalence, which together show that the circuit is a valid implementation for the synchronous scheduling results[6].

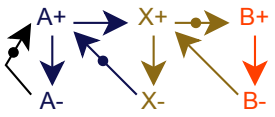

Fig. 2. Fall decoupled latch controllers (A and B are even, X is odd)

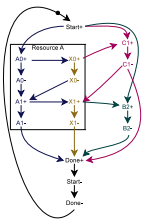

Fig. 3. Marked Graph of Fall-decoupled model extended with resource sharing

1) Liveness: Liveness indicates that the circuit cannot enter a deadlock state, a state which it cannot leave. A strongly connected Marked Graph is live *if* each directed circuit contains at least one token.

2) Flow equivalence: An asynchronous circuit is flow-equivalent to the synchronous counterpart, or in our case the scheduling results, if the data in each latch of the asynchronous circuit is equal to the data of the corresponding latch in the synchronous counterpart.

IV. Proposed method

A. Data-path

For the conversion of the synchronous scheduling results to a latch-based design, the flip-flop is replaced by two latches. Using re-timing, one latch can be placed between the MUX and the input of the operation, as shown in Figure 1 b (flip-flop) and c (latch). Since the second input to the multiplier is a constant in the FIR filter, these are hard-coded in the multiplier and not shown in the data-path.

Now the latches are located at the output of the MUX, the MUX and latch can be combined using dynamic logic. In our simulations, dynamic logic is used for the combination of the input MUX and input latch.

B. Control Network

The starting point for behavioral synthesis is a behavioral description of the circuit. Our method uses a State Sequencing Graph (SSG), which is converted to a bundled-data asynchronous circuit.

The controllers are based on the fall-decoupled model from [6]. This model is live and flow-equivalent to synchronous circuits. However, this model does not allow hardware reuse, so a new model is created which allows hardware reuse, but is still live and flow-equivalent to the synchronous scheduling results.

1) Fall-decoupled model: In Figure 2, the fall-decoupled model is shown. A and X indicate even- and odd latch control signals. The $A+$ transition will make latch A transparent, while $A-$ will make latch A opaque. In this model, even and odd latches alternate. In [6], it is proven that this model live and flow-equivalent to a synchronous counterpart when each flip-flop is replaced by two latches and latch controllers.

2) Resource sharing: To be able to implement the scheduling results, the Fall-decoupled model has to be extended to implement resource sharing. For each operand, a separate handshake is introduced unless the data is an input from the environment. The communication with the environment should also implement handshaking to indicate that new input data is available and that the output data is ready. The start and done signal are introduced to represent the validity of inputs to and outputs from the asynchronous circuit respectively. The start signal indicates valid input data while the done signal indicates valid output data.

Combining the Fall-decouple model with resource sharing results in the controller model shown in Figure 3. In this marked graph, each transition is a latch control signal except Start and Done. The letter A, B and C indicate the input latch control signals for three different resources, while X represents the output latch control signal for the resource with input latch A. The numbers associated with the latch control signal represent the time slot in which the operation is scheduled. If a resource has no operation scheduled for a certain time slot, the numbers will not be subsequent, but the numbers are always strictly increasing, e.g. no two operations can be scheduled on one resource at the same time and the order in time is honored by the marked graph. In the rest of this section, we focus on the implementation of the Control Network.

C. Liveness and flow equivalence

In the proposed control network model, initially there is only a token at the start signal. Considering valid scheduling results as stated in section III-A it can be proven that any directed circuit includes the start signal. Thus, the proposed control network is always live. Similarly, it can be proven that our method is flow-equivalent to the scheduling results.

D. Handshaking

To implement the proposed controllers in a circuit, handshaking is used to communicate between latch controllers. Each operand is coupled with one set of handshake signals. Inside each resource, the even latch controller and odd latch controller also communicate with one set of handshake signals. If output data for a certain operation is used more than once, the handshake is forked to all succeeding operations.

A delay element is required for each latch, which results in two delay elements per resource. The required delay for the operation can be added to one of those delays. To save area and improve delay matching, the handshake signals for all operations scheduled on a particular resource should share the same delay element.

E. Handshake blocks

To create an automated design flow which can implement the proposed handshaking, a number of IP-blocks have been designed. In this section, the topology of the IP-blocks is explained. There are three main blocks (inputselect, odd latch controller and outputselect) and a few support blocks (fake request, fake acknowledge, fork). The topology of those blocks can be found in Figure 4.

1) Inputselect: The inputselect block controls the even latch and input MUX. The active inputselect block which has control over the resource, indicated by the start signal, will send a request out to the odd latch controller and make the even latch transparent when a request in is received. After the delay, the inputselect block will receive an acknowledge out from the latch controller and the even latch will be made opaque again. When the odd latch is also opaque (indicated by a low acknowledge out), a finish signal is send, to hand over control to the next inputselect block.

2) Odd latch controller: The odd latch controller makes the odd latch transparent when a request in is received and the old data is latched by all succeeding resources, indicated by a low output acknowledge. When the latch is transparent, a request out is send and after the delay, an acknowledged out is received, indicating that the data has propagated through the odd latch, so it can be made opaque again. Also, when the request in is high, an acknowledge in is send immediately to indicate that the data has propagated through the even latch. The acknowledge in can only go low when the odd latch is transparent.

3) Outputselect: The outputselect block does not control any latch or MUX, but is used to unfold the subsequent requests from the odd latch controller. The odd latch controller has only one output request signal, but the resource is shared so output data should be coupled with different handshake signals, which is taken care of by the outputselect block. When a request arrives at the active outputselect block, an acknowledge is send to indicate that the odd latch can go opaque again, and an output request is send. The acknowledge is made low when data is latched by the subsequent resource, indicated by a high acknowledge out signal. When the request in is low again, the next outputselect block is activated.

4) Fake request: When an operand is provided by the environment, there is no handshake associated with the data and the data is valid during the entire operation of the circuit. For these cases, a fake request block is designed, which replaces the inputselect block.

5) Fake acknowledge: When a result is not an operand for any operation, e.g. when it is merely an output to the environment, there is no handshake associated with the result. For these cases, a fake acknowledge block is designed, which replaces the outputselect block.

6) Fork: When a result is an operand for multiple operations, the handshake should be forked. A fork is created for this purpose, which forks the request out to all destinations and uses a muller-C element to join all acknowledge signals.

7) Join: When a resource has more than one operand, each operation is assigned two inputselects block and delay elements. The data-path includes an extra MUX and even latch for the second operand. The odd latch controller is modified to include an extra handshake input.

V. Results

To test the asynchronous control flow, a number of DFG's were used. The implemented circuits include a 5th order LWDF low-pass filter, an 11th order WDF filter and an 18-point IMDCT (Figure 6).

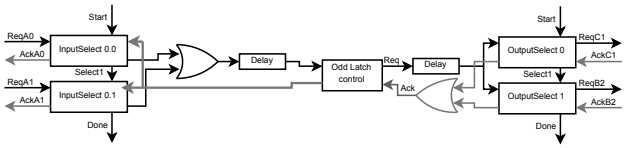

Fig. 4. Handshake blocks implementing a distributed controller

TABLE I
POWER CONSUMPTION AND GATE AREA

Circuit	Synchronous		Asynchronous	
	Power (mW)	Gate area (um^2)	Power (mW)	Gate area (um^2)
5th order LWDF filter	1.46	37687	1.72	93049
11th order WDF filter	3.37	332590	2.70	493602
18-point IMDCT core	13.72	86973	11.43	138622

The circuits were scheduled using the List scheduling algorithm. It is assumed that an ALU with two latches and a MUX has 70% of the delay of an MUL with two latches and a MUX, so during scheduling the ALU was assigned 7 cycles and the MUL was assigned 10 cycles. During synthesis, the delay constraints for the ALU was set to 3.5 ns and the delay for the MUL was set to 5 ns.

The circuits were implemented in UMC90 with a gate library produced by the Faraday corporation. The netlist of the IP-blocks was created using the Technology Mapping function in Petrify and the layout of the IP-blocks is designed using Cadence Encounter. The data-path is compiled by Synopsys Design Compiler. The power consumption and gate area of the three example circuits can be found in Table I.

The longest path of the LWDF circuit at different multiplier latencies is shown in Figure 5. Again, the delay of the ALU is set at 70% of the multiplier delay. The absolute value of the controller overhead increases when the delay of operations decrease. This is a result of controller paths which are not delayed by the delay element that will become part of the critical path, while they would normally be shorter than a different path delayed by the delay element with the same endpoint. The slope of the synchronous circuit is equal to the number of cycles times the multiplier delay, since the multiplier delay fixes the clock period. The latency of the asynchronous circuit is the controller overhead plus the data-path latency.

From Figure 5, it can be concluded that the control overhead is a significant part of the critical path. To outperform a synchronous design when using multiplier latencies of 5ns, the control overhead should be reduced to 30% of its current value.

VI. CONCLUSION

An asynchronous distributed control network based on a number of pre-designed IP-blocks has been presented. A VHDL code generator has been implemented in our scheduling toolbox and a number of circuits are shown to prove the effectiveness of the implementation. Compared to synchronous designs, a significant reduction (upto 20%) in power consumption can be achieved for larger circuits, while maintaining a

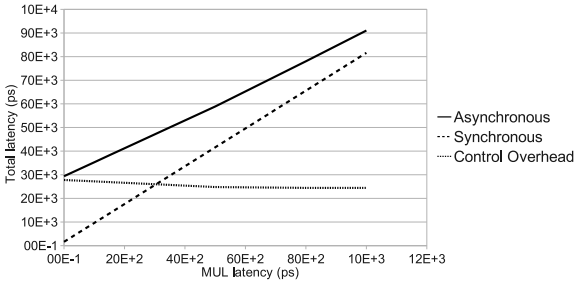

Fig. 5. Latency of asynchronous and synchronous LWDF filter with different multiplier latencies

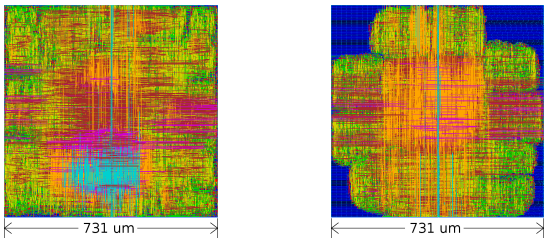

Fig. 6. The asynchronous design of a IMDCT (left) and the synchronous version at the right side

reasonable additional area size cost. At the same time the latency of the asynchronous circuits is kept the same as of the synchronous circuits. More importantly, the design of the asynchronous digital circuits has become a lot easier, since our scheduling toolbox automatically generates asynchronous circuit implementations of a given set of data flow graphs.

More realistic delay matching could improve the performance of the resulting asynchronous circuits. Also, different completion detection methods can be implemented.

REFERENCES

[1] B. Bachman, H. Zheng, and C. Myers, "Architectural synthesis of timed asynchronous systems," in *Computer Design, 1999. (ICCD '99) International Conference on*, 1999, pp. 354 –363.

[2] H. Saito, N. Hamada, N. Jindapetch, T. Yoneda, C. Myers, and T. Nanya, "Scheduling methods for asynchronous circuits with bundled-data implementations based on the approximation of start times," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E90-A, pp. 2790–2799, December 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1521680.1521697

[3] J. Cortadella and R. Badia, "An asynchronous architecture model for behavioral synthesis," in *Design Automation, 1992. Proceedings., [3rd] European Conference on*, march 1992, pp. 307–311.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," 1996.

[5] J. Sparsøand S. Furber, *Principles of Asynchronous Circuit Design*. Kluwer Academic publishers, 2001.

[6] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake protocols for de-synchronization," in *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, april 2004, pp. 149 – 158.