High Level Synthesis of Asynchronous Circuits from Data Flow Graphs

Rene van Leuken, Tom van Leeuwen, and Huib Lincklaen Arriens

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 toolbox for the automatic generation of asynchronous circuits starting from a data flow graph description. The toolbox consists of a scheduling and code generation tool. We use traditional scheduling algorithms as for synchronous circuits, but have replaced the implied synchronous controller for an 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. The distributed controllers 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. Our toolbox is a capable to generate large complex asynchronous solutions, with upto 20 percent power saving, and as least as good latency performance as of synchronous solutions.

1 Introduction

Digital circuits use a clock signal to synchronize operations, the so called synchronous circuits. Although this clock signal makes the design convenient, especially since practically all commercial synthesis tools assume a synchronous design, some advantages can be exploited when using asynchronous circuits (circuits without clock signal). Those advantages can include typical case performance, low power consumption, less sensitive to variability, lower EMI admittance and protection against differential power analysis attacks. Disadvantages of asynchronous circuits include the lack of synthesis tools, their sensitivity to hazards and in some cases performance loss. To assist a designer in his/her attempts to convert a behavior level description of a compute function to be implemented in digital hardware, we have developed an toolbox, which is capable of scheduling and mapping operations on hardware resources. These operations are currently limited to ALU functions like multiplication, subtraction, comparison and addition. However, since many computational and signal processing functions consist of only these functions, many of them can be implemented. Our design methodology uses traditional scheduling algorithms as for synchronous

J.L. Ayala et al. (Eds.): PATMOS 2011, LNCS 6951, pp. 317–330, 2011.

circuits to generate a asynchronous solution. To achieve this, we replace the by the scheduling software implied synchronous controller with an asynchronous distributed control network.

2 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. In this paper, standard scheduling algorithms for synchronous circuits are used, but a new control network is created, targeting asynchronous circuits. For behavioral synthesis of asynchronous circuits, a number of methods for scheduling and resource allocation are published [1] [8]. However, these publications do not include the synthesis of the control network. Also, a number of behavioral synthesis methods for asynchronous circuits including the control network synthesis are 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. In our method, only a few small STG's have to be synthesized, which can then be reused to create the larger distributed controller. In [5], a high level synthesis method using a bundled-data centralized controller is proposed. The centralized controller neglects some of the advantages of asynchronous operation, since all operations are synchronized by the controller. Also, their method is limited to bundled-data implementations, while our method can easily be converted to any completion detection method. In [2], Cortadella et. al. propose a method for de-synchronization. De-synchronization is the process of converting a (synthesized) synchronous circuit to an asynchronous circuit. Although this method does not target high-level synthesis and prevents resource sharing, the theory of de-synchronization is used in this paper since our method uses scheduling results for synchronous circuits.

3 Background

The starting point for behavioral synthesis is a behavioral description of the circuit. Our method uses a State Sequencing Graph (SSG) as input, which is then converted to a bundled-data asynchronous circuit. The conversion from a behavioral description to an SSG is not explained in this paper in detail, since the same method is used for synchronous designs, so only the relevant issues are explained in this paper. More details about scheduling and resource allocation, the process of converting the behavioral description to a SSG, can be found in [7].

3.1 Data Flow Graph

A Data Flow Graph (DFG) is a graph of operations, represented by nodes, and data-dependencies represented by directed edges. Additionally, there are

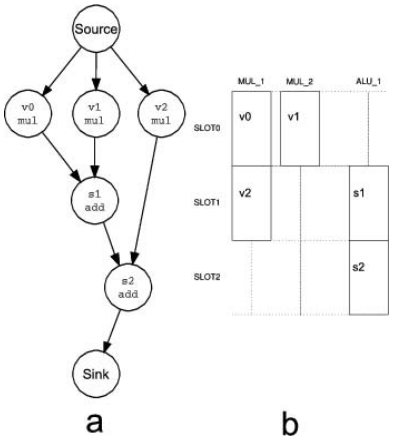

Fig. 1. a: Data Flow Graph of a 3rd order FIR filter, b: Resource Mapping

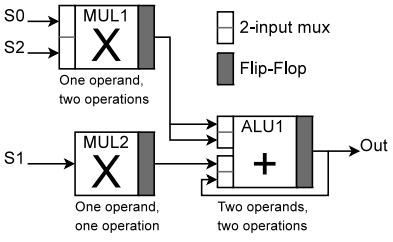

Fig. 2. Datapath of FIR3 filter with flip-flop

two extra nodes, the *source* and *sink* node. Those are used to represent data-dependencies from and to the environment. When an operation processes input data, it depends on the source node, and when the output of an operation is used by the environment, the sink node depends on this operations. An example is shown in Figure 1-a, where the DFG of a 3rd order Finite Impulse Response (FIR) filter is depicted.

3.2 Scheduling and Resource Sharing

When the data-dependencies are identified using the DFG, a scheduling algorithm can map each operation to a time slot. Then, operations can be allocated to resources like Multipliers and ALU's. Each resource can execute a number of operations from the DFG, but it can only execute one operation per time slot. Each operation is scheduled on a resource that is able to execute the operation. Data can be saved for more than one cycle in the flip-flop of a resource, but when the data needs to be saved after a new operation is executed, a register is used which is also considered a resource. This paper will not go into detail about scheduling and resource allocation, since well-known algorithms for synchronous circuits are used. The results of scheduling and resource allocation for the FIR filter can be found in Figure 1-b. It should be noted that most scheduling algorithms support multiple clock cycles per operation. Using a large number of clock cycles for each type of resource allows the synchronous scheduling algorithm to approximate asynchronous behavior at the cost of computational time [8]. As stated, each resource is scheduled to execute a number of different operations from the DFG. In the intended synchronous circuit, 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 intended. The

flip-flop of a resource loads new data at the end of an operation scheduled to that resource. For example, if resource A is scheduled to do two operations, ax and ay, during cycle 1 and 3 respectively, then the results of operation ax is available during cycle 2 and 3, and the result of operation ay is available from cycle 4 to the last cycle. The datapath of the intended synchronous implementation of the FIR filter can be found in Figure 2. Since the second input to the multiplier is a constant in the FIR, these are hardcoded in the multiplier and not shown in the datapath. There are a number of requirements which have to be satisfied in order to create a valid scheduling for the intended synchronous datapath. These requirements are used later on:

- 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).

3.3 Bundled-Data

Asynchronous circuits indicate themselves when an operation is finished. There are several ways for an operation to indicate the completion, but the most common ways is by a matched delay element. If the operation starts, the input of the delay element is toggled. When the output of the delay element also toggles, the operation is assumed to be finished. The output of the delay element can thus be used to indicate that the succeeding operation can start [9]. A matched delay element is not data-dependent, and thus the delay is matched to the longest path in the operation. Although average-case performance can not be achieved with bundled data, performance improvements can be achieved by delay matching between the delay element and the operation since the variation between gates within a (part of a) chip is smaller than the maximum variation taken into account by the design of synchronous circuits [6]. Note that our method is not limited to Bundled-data, it can be converted to any completion detection method.

3.4 Signal Transition Graphs

Signal Transition Graphs (STG's) are a subset of Petri nets where all transitions are signal transitions [4]. In this paper, STG's are used to model Speed-Independent controllers. Speed-Independent circuits operate hazard-free under certain assumptions [9]. An 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 marked graph, all places are usually implicit. *Directed circuits* are a closed cycle in a marked graph 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.

3.5 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 [2]. Replacing the flip-flops for latches is a technique also used in synchronous designs. The process is trivial since a flip-flop, which is normally composed of two latches, is now explicitly created with two latches. Additionally, when the circuit is latch-based, the circuit can be retimed since the two latches are independent, i.e. the latches can be moved through logic blocks as long as the timing requirements are met. For de-synchronization, the clock signal for the latches is replaced by latch controllers. The controllers for de-synchronization, discussed in detail by Cortadella, ensure that the circuit is equivalent to the synchronous counterpart. Since we use scheduling results which are valid for synchronous circuits, we can use the theory of de-synchronization to prove that our asynchronous circuit is able to implement any valid scheduling results. However, the controllers proposed by Cortadella do not allow resource sharing, so new controllers are designed based on the theory of de-synchronization. In this paper, 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 replacement for the synchronous counterpart [2], in our case the intended synchronous circuit represented by the scheduling results.

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.

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.

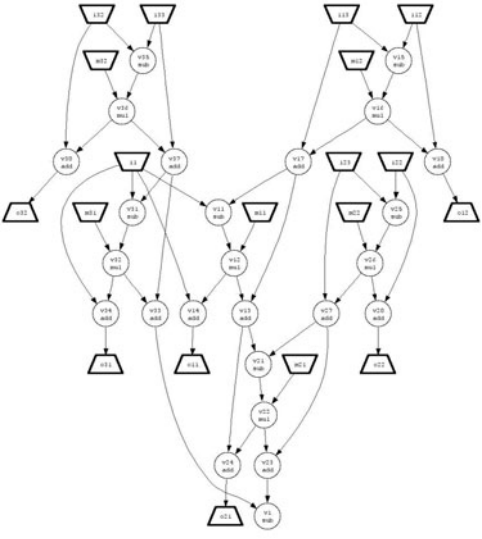

Fig. 3. A DFG of an example (6th order Lattice Wave Digital Filter) designers may use to experiment with scheduling tools and implementation

4 Proposed Method

4.1 Toolbox

To assist a designer in his/her attempts to convert a behavior level description of a compute function to be implemented in digital hardware, we have developed an toolbox, which is capable of scheduling and mapping operations on hardware resources. These operations are currently limited to some ALU functions, such as multiplication, addition, subtraction and comparison. The tool lets designers enter a data flow graph description and after specifying some attributes like type and number of resources, the tool creates a scheduled data flow graph (Figure 3) and maps the operations to resources. The tool is set up as a collection of (Matlab) functions, some of which are accessible through a GUI (Figure 4). The functions can be used for testing an algorithm both in the Matlab environment as a reference, as well as for supplying the VHDL test benches. Currently, designers can choose from the ASAP and ALAP scheduling methods (minimum number of clock states, unlimited resources), a Force Directed scheduling method (minimum number of clock states, minimum number of resources which are optimally distributed over the available clock states) and a List scheduling method (user defined number of resources that determine the number of clock states needed).

4.2 Datapath

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

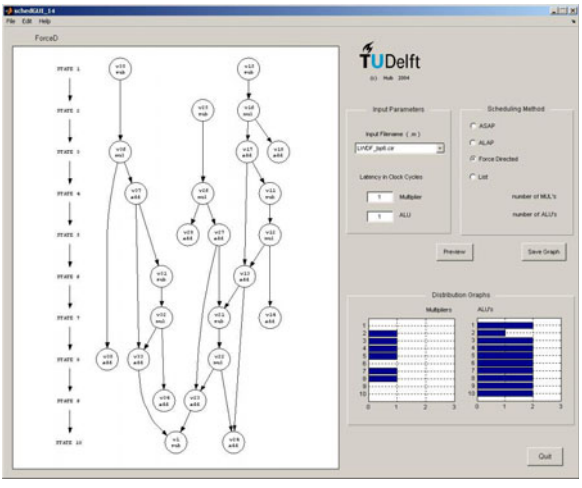

Fig. 4. The user interface of the scheduling and mapping tool

between the MUX and the input of the operation, as shown in Figure 5. A latch is located at the output of the MUX, the MUX and input latch can be combined using dynamic logic. In our simulations, dynamic logic is used for the combination of the input MUX and input latch.

4.3 Control Network

The controllers are based on the fall-decoupled model from. 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.

Fall-decoupled Model. In Figure 6, 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 [2], 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.

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 signal is introduced unless the data is an input from the environment or a constant, which are available during the entire operation of the circuit. The communication with the environment should also contain a form of 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

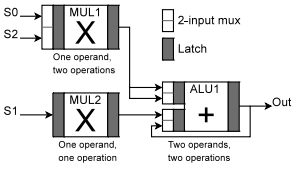

Fig. 5. Datapath of FIR3 filter with latches

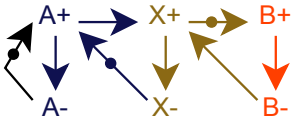

Fig. 6. Fall decoupled latch controllers (A and B are even, X is odd)

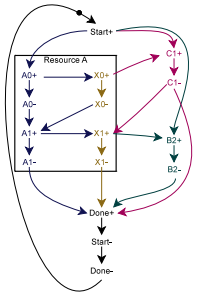

Fig. 7. Marked Graph of Fall-decoupled model extended with resource sharing

validity of inputs to and outputs from the asynchronous circuit respectively. When the start signal goes high, it indicates that all input data is valid, and when the done signal goes high, it indicates that the output data is available. The signals have to go low in the same order to reset the handshake signals to the initial state, i.e. it is a 4-phase handshake. Combining the Fall-decouple model with resource sharing results in the controller model shown in Figure 7. 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.

4.4 Liveness

Initially, there is only a token at the positive event of the start signal. To prove that the model is Live, we have to prove that any directed circuit includes the start signal. To prove that all directed circuits include the positive event of the start signal, the edges are followed backwards from any given event in the circuit:

1. The positive event of the start signal is preceded by the positive event of the done signal via the two negative events of those signals.
2. The positive event of the done signal is preceded by either the last negative event of an odd latch, or the last negative event of an even latch.
3. Any negative event of an odd latch control signal (Xn-) is always fired by the positive event of the odd latch control signal from the same cycle (Xn+).

4. The positive event of an odd latch (Xn+) is always fired by an event of an preceding (An+) or succeeding (Bn-m- where $m >= 0$) even latch at the same cycle or a lower cycle; The even latch from the same resource belongs to the same operation, and thus the same cycle. The negative event from the succeeding even latch (Bn-m-) has to be from the same cycle or a lower cycle, because the negative event indicates that the data in the odd latch from the previous cycle can be overwritten, because it is saved in the succeeding even latch. In the synchronous scheduling results, it is also assumed that previous output data is also available until the end of the next operation.
5. A negative event of an even latch (An-) is always fired by the positive event of that even latch (An+) from the same cycle.
6. A positive event of an even latch is either fired by the start signal containing a token, or by a latch event *at least one cycle earlier*. The positive event of an even latch (An+) is preceded by the negative event of the same latch from the previous operation (An-m- where $m >= 1$) which is at least one cycle earlier, from the negative event of the odd latch (Xn-m-) of the previous operation on the same resource, or from the positive event of a preceding resource (In Figure 7 shown as Xn-m+ with respect to Bn). The positive event of the preceding odd latch indicates that data is ready for an operation. The data for every operation should originate from an operation at least one cycle earlier, because the scheduling assumes that a result of an operation can only be consumed after it is produced.

Following the directed circuit backwards as indicated will always end in event 6, where the cycle number is decreased by one, from where it can be traced back to item 3, 4 or 5 where the cycle number stays equal or decreases and ends in event 6 again. Consequently, any directed circuit ends in cycle 0 of an even latch. Cycle 0 of an even latch is fired only by the start signal which contains a token. Thus, every directed circuit contains a token and the model is live.

4.5 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.

4.6 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 8.

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 and make the even latch is transparent when input data is ready, which is indicated by an input request. 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.

Odd Latch Controller. The odd latch controller makes the odd latch transparent when new data is ready and the old data is latched by all succeeding resources, indicated by a request in and low output acknowledge respectively. 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.

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 (activated by the start signal), an acknowledge is send immediately to indicate that the odd latch can go opaque again, and an output request is send. The acknowledge is made low when the 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 using the finish signal.

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 when the operand is an input from the environment.

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.

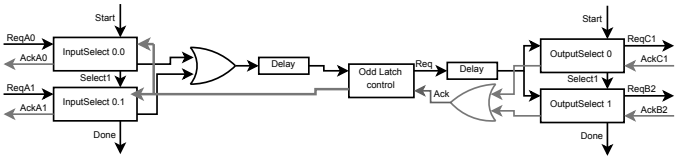

Fig. 8. Handshake blocks implementing a distributed controller

Fork. When a result of 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.

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

5 Results

To test the asynchronous control flow, a number of high-level descriptions were synthesized. The implemented circuits include a 5th order LWDF low-pass filter (Figure 11) and an 18-point IMDCT (Figure 10). 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 IP-blocks are then used to implement the control network for the scheduling results using our scheduling toolbox. Synopsys Design Compiler is used to compile the datapath and select delay elements to match the datapath latency. Then, the datapath, control network and delays are combined in Cadence Encounter and the placement and routing of the IP-blocks and datapath completes the layout. The delay of the data operations were distributed over the latch delay elements instead of using an extra delay element. The typical delay of the controllers and delay elements were matched to the worst-case delay of the datapath, while the simulations were run in typical case conditions. For each circuit, a synchronous counterpart was also generated. The same scheduling algorithm and resources are used, however, one clock cycle of 5ns is assigned to both the MUL and ALU.

Power simulations (Table 1) were performed using Cadence Encounter with back-annotated activity from netlist simulations. The same input was used for the synchronous and asynchronous circuits, except for the clock signal. The speed of the synchronous circuit was matched to that of the asynchronous circuit. For

Table 1. 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

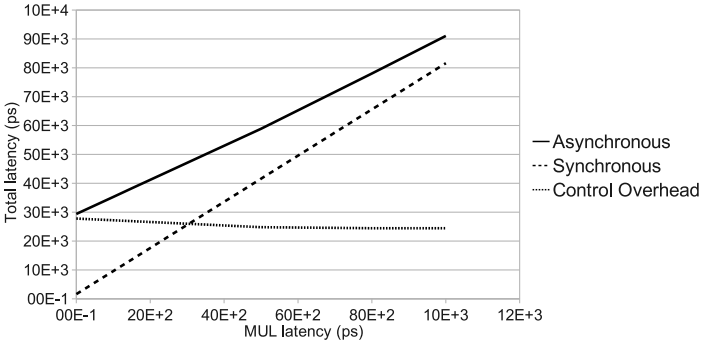

Fig. 9. Latency of asynchronous and synchronous LWDF filter with different multiplier latencies

a small 5th order LWDF filter with 32-bit operations, the synchronous circuit consumes less power. The extra power consumption for the asynchronous circuit can be attributed to the power consumption of the control network, which consumes more power than the synchronous control network per resource. For larger circuits, where more operations are scheduled per resource, the power of the control network becomes less significant and the asynchronous circuits use considerably less power than the synchronous counterparts. Table I also shows the area size of the designs. The numbers show that the additional area required to implement the distributed control network is substantional for small digital designs. The longest path of the LWDF circuit at different multiplier latencies is shown in Figure 9. The delay of the ALU is set at 70% of the multiplier delay. It can be observed that 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 datapath latency. The datapath latency is equal to the results of the fine-grained scheduling. From Figure 9, it can be concluded that the control overhead is a significant part of the critical path when reasonable values for

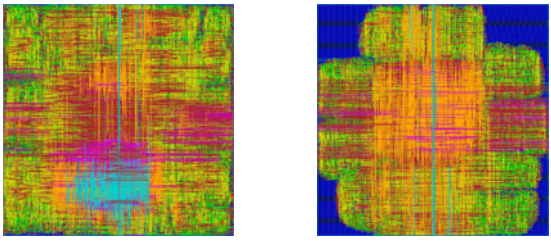

Fig. 10. The asynchronous design of a IMDCT (left) and the synchronous version at the right side

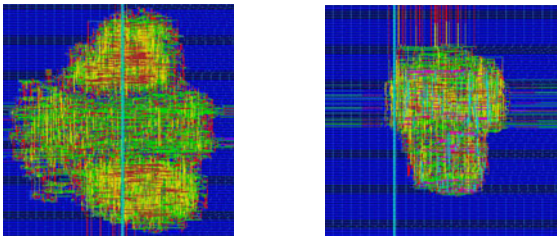

Fig. 11. The asynchronous design of a LWDF (left) and the synchronous version at the right side

the multiplier delay are used. To outperform a synchronous design when using multiplier latencies of 5ns, the control overhead should be reduced to 30% of its current value.

6 Conclusion

In this paper we have presented a toolbox for the automatic generation of asynchronous circuits starting from van data flow graph description. The toolbox consists of a scheduling and code generation tool. We use traditional scheduling algorithms as for synchronous circuits, but have replaced the implied synchronous controller for an asynchronous distributed control network. We have also presented an asynchronous distributed control network based which based upon a number of pre-designed and optimized IP-blocks. Compared to synchronous designs, a significant reduction (upto 20 percent) in power consumption can be achieved for larger circuits, while maintaining good latency figures. The area size cost is still high. However, some improvements are still possible such as flip-flop based register file designs. More importantly, the design asynchronous digital circuits has become a lot easier, since our high level synthesis toolbox automatically generates asynchronous circuit implementations of a given set of data flow graphs. But also, our toolbox is capable to synthesize large and very large complex circuits. To our knowledge, this was not possible before.

References

1. Bachman, B., Zheng, H., Myers, C.: Architectural synthesis of timed asynchronous systems. In: International Conference on Computer Design, ICCD 1999, pp. 354–363 (1999)
2. Blunno, I., Cortadella, J., Kondratyev, A., Lavagno, L., Lwin, K., Sotiriou, C.: Handshake protocols for de-synchronization. In: Proceedings of 10th International Symposium on Asynchronous Circuits and Systems, pp. 149–158 (April 2004)
3. Cortadella, J., Badia, R.: An asynchronous architecture model for behavioral synthesis. In: Proceedings of 3rd European Conference on Design Automation, pp. 307–311 (March 1992)
4. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers (1996)
5. Hamada, N., Shiga, Y., Saito, H., Yoneda, T., Myers, C., Nanya, T.: A behavioral synthesis method for asynchronous circuits with bundled-data implementation (tool paper). In: 8th International Conference on Application of Concurrency to System Design, ACSD 2008, pp. 50–55 (June 2008)
6. Imai, M., Nanya, T.: A novel design method for asynchronous bundled-data transfer circuits considering characteristics of delay variations. In: 12th IEEE International Symposium on Asynchronous Circuits and Systems, pp. 10–77 (2006)
7. de Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education, New York (1994)
8. Saito, H., Hamada, N., Jindapetch, N., Yoneda, T., Myers, C., Nanya, T.: Scheduling methods for asynchronous circuits with bundled-data implementations based on the approximation of start times. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E90-A, 2790–2799 (2007), http://portal.acm.org/citation.cfm?id=1521680.1521697
9. Sparsø, J., Furber, S.: Principles of Asynchronous Circuit Design. Kluwer Academic Publishers, Dordrecht (2001)