Downloaded from orbit.dtu.dk on: Nov 19, 2019

DTU Library

# Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation

**Nielsen, Sune Fallgaard; Sparsø, Jens; Madsen, Jan; Selvaraj, Henry**

*Published in:*
EUROMICRO Symposium on Digital System Design

*Link to article, DOI:*
10.1109/DSD.2004.1333290

*Publication date:*
2004

*Document Version*
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

*Citation (APA):*
Nielsen, S. F., Sparsø, J., Madsen, J., & Selvaraj, H. (Ed.) (2004). Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation. In *EUROMICRO Symposium on Digital System Design* IEEE. https://doi.org/10.1109/DSD.2004.1333290

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

# Towards behavioral synthesis of asynchronous circuits – an implementation template targeting syntax directed compilation.

S. F. Nielsen J. Sparsø J. Madsen

Technical University of Denmark, Informatics and Mathematical Modelling
Richard Petersens Plads, Bldg. 322, DK-2800 Kgs. Lyngby, Denmark

e-mail: {sfn,jsp,jan}@imm.dtu.dk

## Abstract

*This paper presents a method for behavioral synthesis of asynchronous circuits. Our approach aims at providing a synthesis flow which is very similar to what is found in existing synchronous design tools. We adapt the synchronous behavioral synthesis abstraction into the asynchronous handshake domain by introducing a computation model, which resembles the synchronous datapath and control architecture, but which is completely asynchronous. The datapath and control architecture is then expressed in the Balsa-language, and using syntax directed compilation a corresponding handshake circuit implementation is produced. The paper also reports area, speed and power figures for a couple of benchmark circuits, which have been synthesized to layout.*

## 1 Introduction

Asynchronous circuits have a number characteristics that can be exploited to advantage in the design of current and future submicron integrated circuits, and the design and implementation of asynchronous circuits is by now well understood [8, 10, 16, 19]; However, in order to enable a more widespread adaptation of asynchronous design, access to efficient high level synthesis tools is crucial and unfortunately such tools are largely lacking. In this paper we outline a *complete* behavioral synthesis flow, and present some important steps of this flow which uses traditional front-end behavioral synthesis techniques and which uses an existing asynchronous synthesis tool as the backend.

Figure 1 illustrates the synchronous and asynchronous design flows that are typical of today, and it shows where the work presented in this paper fits in. The details will be explained below and in the following section.

Synthesis of synchronous circuits, which is illustrated in the left column of figure 1, has succeeded in raising the level of abstraction to that of specifying circuits at the behavioral level. From a behavioral description in a language like VHDL, Verilog or System-C some intermediate representation is extracted – often a control data flow graph (CDFG). From the CDFG the classic synthesis tasks [15] of scheduling, allocation, and binding is performed resulting in a RTL level circuit description which is then synthesized into gate level circuits and eventually a layout.

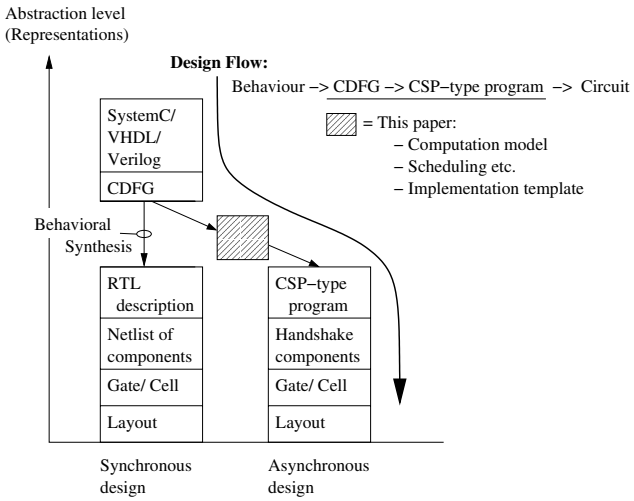


Figure 1: Existing synchronous and asynchronous design flows and the design flow addressed in this paper.

Synthesis of asynchronous circuits is illustrated in the right column of figure 1. It is less mature and several somewhat different approaches is being pursued. The most influential of the available synthesis tools falls in two categories: (i) synthesis of large-scale RTL level circuits based on syntax directed compilation from CSP-like languages: Tangram [3, 20], OCCAM [4], Balsa [2], ACK [14] and TAST [18], and (ii) syn-

Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04)
0-7695-2203-3/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.

thesis of small-scale sequential control circuits [9, 11]. The tools that perform syntax directed compilation target a library of so-called handshake components; some examples will appear in section 5. The handshake components can be designed using in principle any of the sequential control circuit synthesis tools.

The syntax directed compilation approach is radically different from the behavioral synthesis flow used by designers of synchronous circuits; the compiler merely performs a one-to-one mapping of the program text into a corresponding circuit structure. Although syntax directed compilation does allow the designer to work at a relatively high level it does not provide any optimizations; "what you program is what you get". In some situations this can be considered an advantage but in general it puts more burden on the designer: exploring alternative implementations requires actually programming these, whereas in a traditional synchronous synthesis flow, the designer can quickly and easily experiment with different constraints and goals and in this way create alternative implementations from the same program text. In our work we use Balsa as a back-end and take advantage of the one-to-one mapping which allow us to describe *specific* implementations at a high level.

It is interesting to note that the internal representation of circuit behavior used in synchronous behavioral synthesis is actually based on an asynchronous model – a CDFG, i.e., a dependency graph expressing the control- and data-flow of the application. This naturally raises the question, addressed in this paper: Is it possible to apply the transformations and optimizations used in synchronous synthesis, for asynchronous design as well?

The design flow that we target in our work is illustrated in figure 1, and as illustrated this paper focus on behavioral synthesis, i.e. transforming a CDFG representation into a structural netlist of handshake components (represented as a Balsa program). In this way we leverage existing and mature tools and techniques for both high level design of synchronous circuits and (back end) synthesis tools for asynchronous design. The contribution of this paper is the addition of behavioral synthesis to asynchronous circuit design in the form of automatic resource sharing and constraint based design space exploration. In particular our contributions are: (1) an abstract event based computation model, (2) synthesis algorithms for scheduling, allocation and binding and (3) a suitable target implementation template. We have previously studied scheduling algorithms usable in this context [17] and there is nothing preventing the use of scheduling algorithms developed by other researchers [5, 12].

The paper is organized as follows: Section 3 introduces the concept which allows us to adapth the techniques from synchronous behavioral synthesis into behavioral synthesis of asynchronous design. Section 4 describes details of the asynchronous datapaths. Section 5 briefly explains the Balsa templates, and finally section 6 presents and discusses some results on the efficiency of the approach.

## 2 Related work

The introduction mentioned a number of asynchronous high level synthesis tools. Tangram [3, 20] is a proprietary tool of Phillips. It is quite mature and has been used to design circuits which are currently in production. Balsa [2] is a somewhat similar tool which has been developed by the University of Manchester and which is available in the public domain. These tools are based on syntax directed compilation where there is a one-to-one correspondence between the program source and the resulting circuit and where the control is highly distributed. TAST [18] and in particular ACK [14], involve the generation of a datapath and one or more centralized controllers. ACK is no longer supported and TAST is not available in the public domain.

A number of papers have presented work on synthesizing asynchronous circuits from DFG or CDFG representations, but they are surprisingly few and they have a different and/or more limited scope [1, 6, 7, 13]. The first paper limits itself to DFGs and focus mostly on a synthesis algorithm and its runtime. The remaining papers address synthesis from a CDFG representation and they target solutions where a centralized controller or a distributed structure of controllers are specified at the level of individual signal transitions (in the form of signal transition graphs or burst-mode state graphs).

Our approach is different in that it targets handshake components and syntax directed compilation. This makes it both simpler and more powerful: Simpler because the controller is synthesized implicitly in a distributed fashion whereas in the previously published approaches it represents a major task of the synthesis. And more powerful because Balsa allows very large circuits to be synthesized.

Some research seems to indicate that the distributed control and the handshake signaling, which characterize circuits produced by syntax directed compilation, results in poor speed. To alleviate this a number of low-level post-synthesis techniques are being used. One approach is peephole optimization which replaces common structures of handshake components


IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.

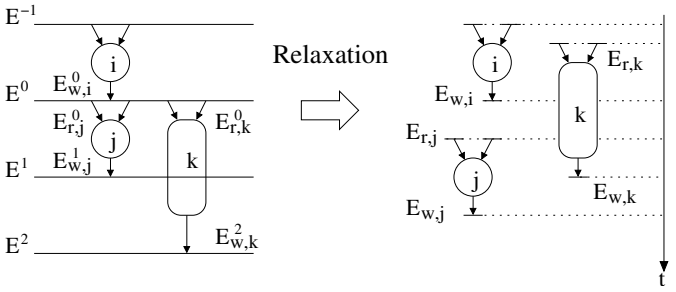


Figure 2: Adapting synchronous synthesis (left) into the asynchronous handshake domain (right).

with simpler ones [20, 12] and other approaches involve re-synthesis from a specification of the behavior of one or more handshake components into a more efficient implementation [5]. In any case this work is orthogonal to the work presented in this paper where focus is on high level synthesis.

## 3 From synchronous to asynchronous behavioral synthesis

Let us first review and analyze the elements of synchronous behavioral synthesis. The target for behavioral synthesis is a hardware architecture consisting of a datapath which is able to perform a set of operations, and a controller which controls the execution sequence of these operations in order to perform a given application. A key issue in behavioral synthesis is to reuse hardware resources for the different operations in order to minimize area, and to explore possible parallelism by executing several hardware resources concurrently in order to increase performance.

Most behavioral synthesis tools make optimizations based on a CDFG which is extracted from a behavioral specification of the circuit behavior. This specification may be expressed in a hardware description language such as VHDL, Verilog or SystemC, or in a traditional programming language such as C, C++ or Java. Behavioral synthesis-tools for synchronous systems use modern compiler techniques to translate source code into some variant of a CDFG as part of their front-end. This process is well understood [15] and will not be addressed in this paper.

Based on the CDFG, synchronous behavioral synthesis tools perform three sets of transformations in order to create a suitable hardware architecture;

- Scheduling, in which operator nodes of the CDFG are grouped into operation-groups or time-slots, and where the execution of the next operation-group is handled by a synchronization event, $E^i$, where $i$ strictly orders the events in time. In the case of synchronous behavioral synthesis $E^i$ is controlled by the system clock.

- Allocation, in which the minimum hardware resources/ functional units (FUs), required for execution of the operation-groups are determined.

- Binding (or assignment), where individual operator nodes are tied to specific hardware resources.

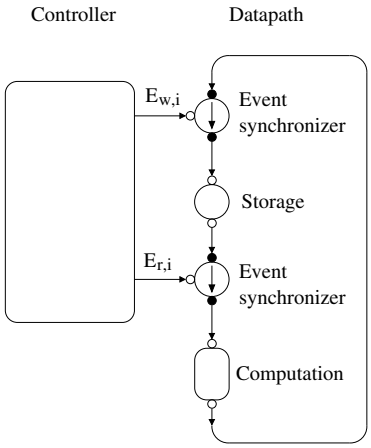


Figure 3: Computation model in the asynchronous handshake domain, where the labeling refers to the role the handshake components play in our model.

The synchronization events determine (i) the beginning of executing an operation (ii) writing the result of an operation.

The CDFG extracted in the synchronous behavioral synthesis is a 1-bounded colored Petri net, where colors represents data values, edges represent places, and nodes represent transitions. Interestingly, the Petri net model is based on an asynchronous execution semantics which should make it a obvious model for asynchronous synthesis as well. In the synchronous synthesis, figure 2 (left), operations are ordered according to a global synchronization event, $E^i$, i.e., read events ($E_{r,j}$) for operator $j$ happens at the same point in time as the write events ($E_{w,i}$) for operator $i$ in the previous operation-group: $E^0_{w,i} = E^0_{r,j} = E^0$, and furthermore all operations in an operation-group are executed simultaneously: $E^0_{r,j} = E^0_{r,k} = E^0$.

If we relax these assumptions: $E_{w,i} \neq E_{r,j}$ and $E_{r,j} \neq E_{r,k}$ as shown in figure 2 (right), and if we make these synchronization events controlled by the controller, we can create a hardware architecture consisting of a datapath and a controller as shown figure 3. It resembles the synchronous architecture but it is completely asynchronous. For this model to oper-

Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04)
0-7695-2203-3/04 $ 20.00 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.

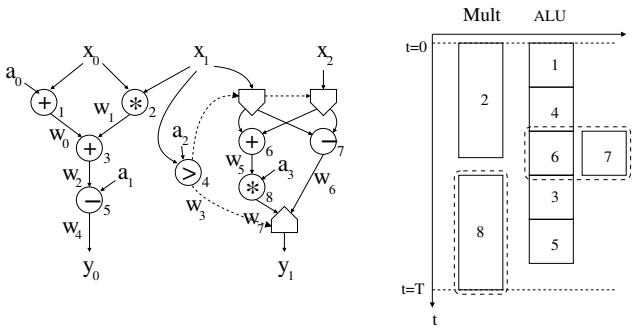


Figure 4: (Right) Our example CDFG with labels on temporary data. (Left) Scheduling of our CDFG.

ate with arbitrary synchronization events, the computation part (functional units) has to act as an independent process, with its own local control, decoupled from the storage. In this way we have adopted the synchronous abstraction to the asynchronous handshake domain.

This idea allows us to use any of, but not restricted to, the many synchronous behavioral synthesis techniques to obtain a hardware architecture (datapath and controller) and then to implement this architecture using asynchronous circuit techniques. At the same time, this idea allows the use of behavioral synthesis techniques operating in continuous time.

## 4 Datapath synthesis

Lets assume we are given a CDFG and that scheduling, allocation and assignment has been performed as shown in figure 4 using the FU library shown in table 1. The FU library have been normalized with respect to the ALU component. We will consider the schedule to operate in continuous time. However it is of no importance whether the schedule have been obtained using an asynchronous scheduling method or through a synchronous method which have been relaxed into continuous time, as discussed in the previous section. Note that the operator nodes have been labeled: 1,2,..,8 and temporary data: $w_0$,$w_1$,...,$w_7$. The branch part of the CDFG, nodes $\{6, 7, 8\}$, gives rise to two paths in the schedule. Determined by the execution of node 4, either 6 and then 8, or 7.

The scheduling in figure 4 results in the fastest execution of the CDFG on a datapath containing only one Mult and one ALU component.

The general structure of the asynchronous datapath is shown in figure 5 and it follows the computation model presented in the previous section. The internal variables (L0...Ln) in our datapath are implemented as latches. The functional units (FU0...FUm) are implemented as independent processing units, with local control, wrapping the computation part with latches on both input and output ports. The functional units can be simple combinatorial blocks or they can be augmented with input and output latches. This choice has consequences on circuit area, lifetime of the variables, speed and power consumption. In this paper we assume that the functional units have normally opaque latches on input and output ports. This is a somewhat arbitrary choice and has no fundamental implications on the approach or the synthesis algorithms. The use of input and output latches tends to increase speed and to reduce power consumption by preventing spurious signal transitions to propagate beyond latch boundaries. If input and output latches are not used, more variable latches may be needed in the datapath in order to accommodate the longer lifetime requirements and in order to avoid auto assignments.

| FU | $\sigma$ | t | A | E |
|---|---|---|---|---|
| ALU | $\{+, -, >\}$ | 1 | 1 | 1 |
| Mult | $\{*\}$ | 2.6 | 10 | 13 |

Table 1: Simple example normalized FU library.

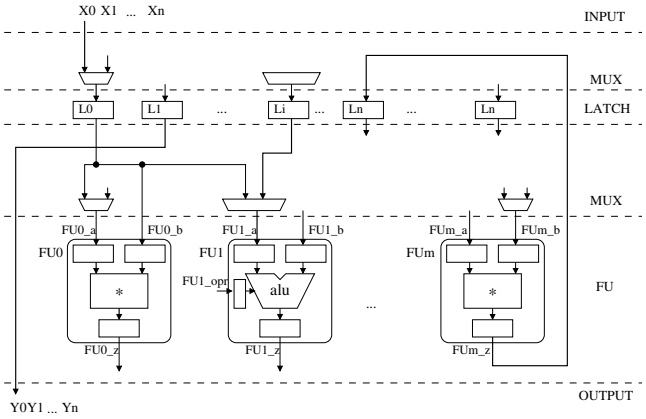


Figure 5: General structure of the datapaths our asynchronous circuits have.

To compute the life times we need to determine how long a variable is to be kept. Since our FUs have input latches we only need to hold the variable until it have been read for the last time, at the start of the last computation. This reduces the variable life time requirements, leading to a possible reduction in the number of variables needed. We estimate the overhead for reading and writing a result to a variable latch to be $t_\Delta = 1/3t_{ALU}$, which is added to the variable lifetime.

Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04)
0-7695-2203-3/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.

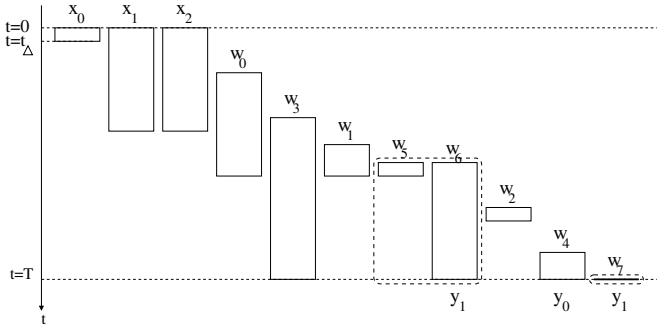


Figure 6: Variable lifetime for our scheduled CDFG.

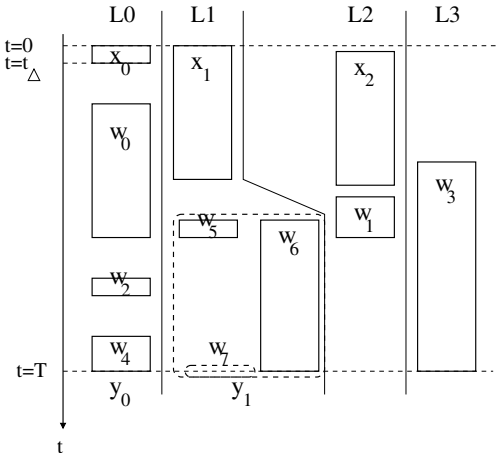


Figure 7: Latch assignments for our scheduled CDFG.

For our example, the variable lifetime using this latch convention is shown in figure 6. We can use the left-edge algorithm [15] to find the minimum number of latches required in the datapath, which in this case is 4 latches. The resulting variable to latch assignment is shown in figure 7.

With the FU allocation, operator to FU assignment and variable latch assignment the datapath can be constructed by connecting the components through multiplexors. The datapath for our example is shown in figure 8. The controller to this circuit implements the schedule and starts the FUs with the right data at their designated times.

## 5 Balsa implementation

For implementing the controller and datapath in asynchronous hardware, we are utilizing the Balsa CAD framework. In figure 9 is shown the Balsa handshake circuit equivalent to our datapath from figure 8. Such a Balsa handshake circuit is built from handshake components which implements the equivalent RTL operations as latching data, multiplexing data,

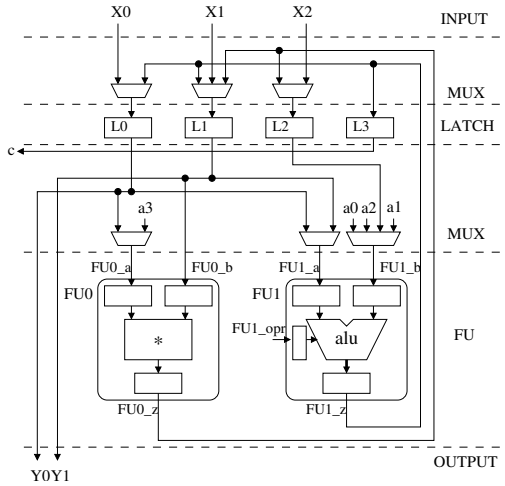


Figure 8: Datapath for our scheduled CDFG.

addition etc. Each of these handshake components has its own local asynchronous control to ensure proper asynchronous functionality and to handle the asynchronous handshake communication protocol [19].

Besides these asynchronous handshake components which have their equivalent RTL counter parts, there are the demux components which handles "wire-forks", and more importantly the transfer handshake components connecting the asynchronous controller with the datapath; the latter play the role of event synchronizers, refer to figure 3, controlling the computation. These extra components augments the mux layers with sublayers of demux and transfer components. Notice the mux components implement a merge functionality and is not directly connected to the controller, neither are the latches, demuxes or FUs (except the opr control signal), only the transfer components are connected to the controller. The FUs are autonomous components which start computing when all their input data is present. Using these components and our computation model, there is a one to one correspondence between the datapath of figure 8 and figure 9.

In our design we use a bundled data 4-phase protocol where signals contain a 1 bit request and a 1 bit acknowledge wire additional to the data wires. Furthermore, the transfer components degenerate to simple wire connections containing no logic.

As an example of how the datapath is constructed using the Balsa-language consider the assignment of subtraction operator 3 (figure 4) to ALU FU1 (figure 9). This subtraction operator has inputs $w_0$ $w_1$ and output $w_2$ ($w_2 = w_0 - w_1$), assigned to variables L0 L1 and L0 respectively. Starting the computation is


IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.

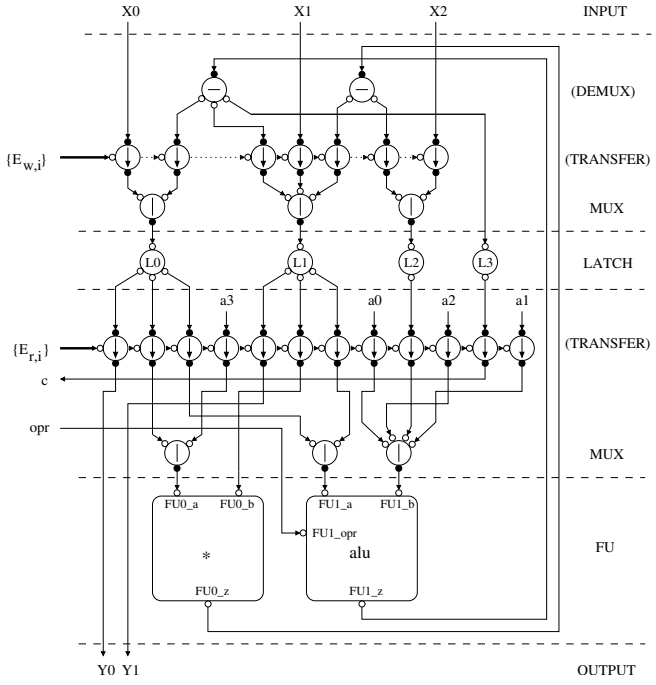


Figure 9: Final datapath for our scheduled CDFG using Balsa/Tangram handshake components.

performed by executing the following parallel Balsa-statement:

```
FU1_opr<-alu_sub || FU1_a<-L0 || FU1_b<-L1
```

This set of parallel channel assignment statements tells FU1 to perform a subtraction, and to use the data of L0 and L1. The result $w_2$ of the computation is written to L0 using the following Balsa-statement:

```
FU1_z->L0
```

Both statements will synchronize the controller with the ALU using the transfer components. Due to the design of the FUs with both input and output latches, the controller and the rest of the datapath is free to do other work while FU1 computes. The reading of input X0 to the internal variable L0 and placing the results of internal variable L0 on output channels Y0 is executed in a similar way. These Balsa-statements: (i) starting a computation, (ii) writing the result of computation or (iii) communicating with the outside world, implements the events described in section 3. These are then sequenced in the right order, using the Balsa-sequence operator ";" implementing our schedule.

The full details of implementing the circuits in Balsa with conditional computation, as well as explaining the optimizations which can be performed

```
import [Balsa.types.basic]
import [FU_types]
import [FU_lib]

procedure Ex(input X0,X1,X2:word;
             output Y0,Y1:word) is
  variable L0,L1,L2,L3:word
  channel FU0_a,FU0_b,FU0_z:word
  channel FU1_a,FU1_b,FU1_z:word
  channel FU1_opr:alu_operation
  constant a0= 255
  constant a1= 255
  constant a2= 255
  constant a3= 255
begin
  Mult(FU0_a,FU0_b,FU0_z) ||
  ALU(FU1_opr,FU1_a,FU1_b,FU1_z) ||
  loop
    X0->L0 || X1->L1 || X2->L2 ;
    FU0_a<-L0 || FU0_b<-L1 || FU1_a<-L0
    || FU1_b<-a0 || FU1_opr<-alu_add ;
    FU1_z->L0 || FU1_a<-L1 || FU1_b<-a2
    || FU1_opr<-alu_gre ;
    FU1_z->L3 ;
    if L3=0 then FU1_a<-L1 || FU1_b<-L2
    || FU1_opr<-alu_add
    else FU1_a<-L1 || FU1_b<-L2 ||
    FU1_opr<-alu_sub end ;
    FU0_z->L2 || FU1_z->L1 ;
    if L3=0 then FU0_a<-a3 || FU0_b<-L1
    end || FU1_a<-L0 || FU1_b<-L2 ||
    FU1_opr<-alu_add ;
    FU1_z->L0 ;
    FU1_a<-L0 || FU1_b<-a1 ||
    FU1_opr<-alu_sub ;
    if L3=0 then FU0_z->L1
    end || FU1_z->L0 ;
    Y0<-L0 || Y1<-L1
  end
end
```

Figure 10: Balsa program for our example.

both to improve the computation speed, reduce temporary variables, and to decouple the control circuit to take advantage of possible variable computation times is beyond the scope of this paper. The full Balsa program of our running example, implementing the controller and datapath, is shown in figure 10.

## 6 Results

In order to demonstrate the feasibility of the proposed approach and in order to evaluate the efficiency of the proposed implementation template we have synthesized different versions of a couple of benchmark circuits, FIR and HAL, and we have simulated the post place-and-route netlists. In this way we are reporting speed, area and energy figures for actual circuit implementations.

It is important to stress the results do not represent


IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.

| id | Alg. | $*$ | ALU | t $[ns]$ | A $[mm^2]$ | E $[nJ]$ |
|---|---|---|---|---|---|---|
| 1 | FIR | 8 | 7 | 124.7 | 0.877 | 2.95 |
| 2 | FIR | 2 | 1 | 284.8 | 0.282 | 2.80 |
| 3 | HAL | 5 | 5 | 171.2 | 0.667 | 2.03 |
| 4 | HAL | 2 | 1 | 309.6 | 0.260 | 1.89 |
| 5 | HAL | 1 | 1 | 397.4 | 0.151 | 2.01 |

Table 2: Layout results.

| FU | $\sigma$ | t$[ns]$ | A $[mm^2]$ | E $[nJ]$ |
|---|---|---|---|---|
| ALU | $\{+,-,>\}$ | 25.5 | 0.0112 | 0.0266 |
| Mult | $\{*\}$ | 56.3 | 0.105 | 0.314 |

Table 3: FU library (16-bit) based on layout in $0.18\mu m$ technology, used by our synthesis algorithm.

an attempt to evaluate the asynchronous implementations against corresponding synchronous ones; our focus is on the efficiency of the automated resource sharing within the asynchronous domain.

The simulation results have been obtained using the following steps: (1) Given a CDFG and circuit constraints in the form of a maximum resource allocation our tool produces a corresponding Balsa program. In this process we target an operator-library consisting of an ALU and a multiplier, and these operators are themselves implemented as small Balsa program modules. (2) The Balsa CAD-tools are then used to generate a Verilog netlist of the asynchronous circuit (single rail 4-phase early protocol) and the Cadence CAD tools are used to generate the corresponding layout. We are using the $0.18\mu m$ STM standard-cell technology, which have been augmented with standard cell components for implementing various special asynchronous components such as Muller C-elements. (3) Finally simulation results are obtained by simulating the Verilog netlist together with extracted layout information in NanoSim. We simulate 200 computations, using random numbers with out any correlation. All the circuits are implemented using 16-bit variables and are simulated at 1.8V and at a temperature of $25^oC$.

The benchmark results are shown in table 2, where $t$ is the average time to do one computation, $A$ is the layout area and $E$ is the average energy consumption per computation. In a similar way we have characterized the ALU and multiplier operators, see table 3. The speed figures in table 3 have been used in the above mentioned step 1 to calculate the schedules.

Implementations 1 and 3 in table 2 are the direct non-resource-shared circuit implementations of the computations. These have also been designed using latches on the input and output of the multipliers. Although this gives an extra area overhead it is insignificant compared to the area of the multiplier. The important fact is that it reduces the combinatorial depth of the circuit and thus reduces the power consumption, which leads to a more fair comparison. The speed figures in table 2 includes a $20ns$ handshake delay in the testbench used to simulate the layouts.

| id | Alg. | $*$ | ALU | t $[ns]$ | A $[mm^2]$ | E $[nJ]$ |
|---|---|---|---|---|---|---|
| 1 | FIR | 8 | 7 | 121.8 | 0.916 | 2.91 |
| 2 | FIR | 2 | 1 | 285.4 | 0.221 | 2.91 |
| 3 | HAL | 5 | 5 | 169.5 | 0.580 | 1.84 |
| 4 | HAL | 2 | 1 | 269.2 | 0.221 | 1.84 |
| 5 | HAL | 1 | 1 | 381.7 | 0.116 | 1.84 |

Table 4: Model results.

The results in table 2 shows that resource sharing saves area at the expense of reduced speed. This is as could be expected. Concerning energy consumption it is interesting to note that it remains constant. Given that resource sharing leads to more control circuitry for the same computation, an increase in energy consumption could be expected. It seems that the smaller size of the layout and the reduced wirelength, which results from this leads to a power saving which corresponds to the increase caused by the added control.

In order to estimate the overhead of the control circuitry which is introduced by resource sharing, we can use the figures in table 3 and estimate the cost of an *ideal* resource shared implementation, e.g. an implementation in which the added control has zero area, latency and energy consumption. Such ideal figures are shown in table 4. Comparing tables 4 and 2 it is seen that the control circuitry introduced by resource sharing accounts for 10-30 % of the area and 0-15 % of the speed, whereas it does not affect energy consumption (as discussed above). It should be noted that the are figure conforms with, figures reported by the tool Balsa-cost which provides cost estimates at the handshake-component level.

We find these results encouraging and in support of the design flow, the implementation template, and the approach to resource sharing, which is proposed by this paper.

## 7 Conclusion

The paper presented a design-flow for behavioral synthesis of asynchronous circuits and it makes the


IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.

following contributions: (i) A method for synthesizing a CDFG to a Balsa-description have been developed using a methodology closely related to, but not restricted to, traditional synchronous behavioral synthesis. This allows the designer to perform design space exploration by adding physical constraints to the circuit. (ii) Using this method and the Balsa and Cadence design tools five layouts have been designed and simulated. The results show that it is possible to do tradeoffs between area and circuit delay for asynchronous circuits. We find there is a $10-30\,\%$ area overhead and a $0-15\,\%$ time overhead and no power overhead implementing this method. We find these results encouraging and in support of the design flow, the implementation template, and the approach to resource sharing, which is proposed by this paper. Future work will include automating the front-end part of the flow, exploration and adaption of more synthesis algorithms, misc. optimizations at the circuit level and last but not least, more and larger benchmarks.

## References


[1] B. M. Bachman, H. Zheng, and C. J. Myers. Architectural synthesis of timed asynchronous systems. In *Proc. ICCD'99 (IEEE International Conference on Computer Design: VLSI in Computers and Processors)*, pages 354–363, October 1999.

[2] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

[3] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150–166. IEEE Computer Society Press, 1988.

[4] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[5] T. Chelcea and S. M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, June 2002.

[6] J. Cortadella and R. M. Badia. An asynchronous architecture model for behavioral synthesis. In *Proc. European Conference on Design Automation (EDAC)*, pages 307–311. IEEE Computer Society Press, 1992.

[7] J. Cortadella, R. M. Badia, E. Pastor, and a: Pardo. Achilles: a high-level synthesis system for asynchronous circuits. In D. D. Gajski, editor, *Proc. 6th International Workshop on High-Level Synthesis*, pages 87–94. Univ. California, 1992.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[9] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[10] R. M. Fuhrer and S. M. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines Algorithms and Tools*. Kluwer Academic Publishers, June 2001. ISBN 0-7923-7425-8.

[11] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[12] G. Gopalakrishnan, P. Kudva, and E. Brunvand. Peephole optimization of asynchronous macromodule networks. In *Proc. International Conf. Computer Design (ICCD)*, pages 442–446. IEEE Computer Society Press, October 1994.

[13] Euiseok Kim, Jeong-Gun Lee, and Dong-Ik Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 104–113. IEEE Computer Society Press, April 2000.

[14] P. Kudva, G. Gopalakrishnan, and V. Akella. High level synthesis of asynchronous circuit targeting state machine controllers. In *Asia-Pacific Conference on Hardware Description Languages (APCHDL)*, pages 605–610, 1995.

[15] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[16] Chris J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, July 2001. ISBN: 0-471-41543-X.

[17] S. Nielsen and J. Madsen. Power Constrained High-level Synthesis of Battery Powered Digital Systems. Proceedings Design Automation and Test Europe, March 2003

[18] M. Renaudin, P. Vivet, and F. Robin. A design framework for asynchronous/synchronous circuits based on CHP to HDL translation. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 135–144, April 1999.

[19] J. Sparsø and S. Furber, editors. *Principles of asynchronous circuit design – A systems perspective*. Kluwer Academic Publishers, 2001.

[20] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.





IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on February 17,2010 at 08:05:48 EST from IEEE Xplore. Restrictions apply.