Behavioral Synthesis of Asynchronous Circuits Using Syntax Directed Translation as Backend

Nielsen, Sune Fallgaard; Sparsø, Jens; Madsen, Jan

Published in:
IEEE Transactions on Very Large Scale Integration Systems

Link to article, DOI:
10.1109/TVLSI.2008.2005285

Publication date:
2009

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

Behavioral Synthesis of Asynchronous Circuits Using Syntax Directed Translation as Backend

Sune Fallgaard Nielsen, Jens Sparsø, *Member, IEEE*, and Jan Madsen, *Member, IEEE*

***Abstract*—The current state-of-the art in high-level synthesis of asynchronous circuits is syntax directed translation, which performs a one-to-one mapping of an HDL-description into a corresponding circuit. This paper presents a method for behavioral synthesis of asynchronous circuits which builds on top of syntax directed translation, and which allows the designer to perform automatic design space exploration guided by area or speed constraints. This paper presents an asynchronous implementation template consisting of a data-path and a control unit and its implementation using the asynchronous hardware description language Balsa. This "conventional" template architecture allows us to adapt traditional synchronous synthesis techniques for resource sharing, scheduling, binding, etc., to the domain of asynchronous circuits. A prototype tool has been implemented on top of the Balsa framework, and the method is illustrated through the implementation of a set of example circuits. The main contributions of this paper are the fundamental idea, the template architecture and its implementation using asynchronous handshake components, and the implementation of a prototype tool.**

***Index Terms*—Asynchronous circuits, behavioral synthesis.**

I. Introduction

MOST of the tools for synthesis of large-scale asynchronous circuits are based on a technique known as syntax-directed translation; a process in which a description in a language similar to communicating sequential processes (CSP, [2]) is mapped directly into a hardware implementation composed of so-called handshake components, each implementing a syntactic element of the program text. Examples of such languages and tools are Haste [3], [4], OCCAM [5], Balsa [1], ACK [6], and TAST [7]. At least one of these tools (Haste) has been used to design significant industrial-quality chips.

It is interesting that these tools are fundamentally different from existing synthesis tools used in industry to design synchronous circuits, in that they perform a one-to-one mapping of a program text into a corresponding circuit. This transparency is both an advantage and a disadvantage. The advantage is that the designer has full control over the resulting circuit. The disadvantage is that in order to explore alternative implementations, the designer is required to actually program these. In behavioral synthesis of synchronous circuits the designer need only express the desired behavior, and based on constraints on area, speed or energy, the synthesis tool then automatically produces an implementation. In this way, the same behavioral description may be used to synthesize circuits with very different area, speed, and energy characteristics. Fig. 1 illustrates the two design flows.

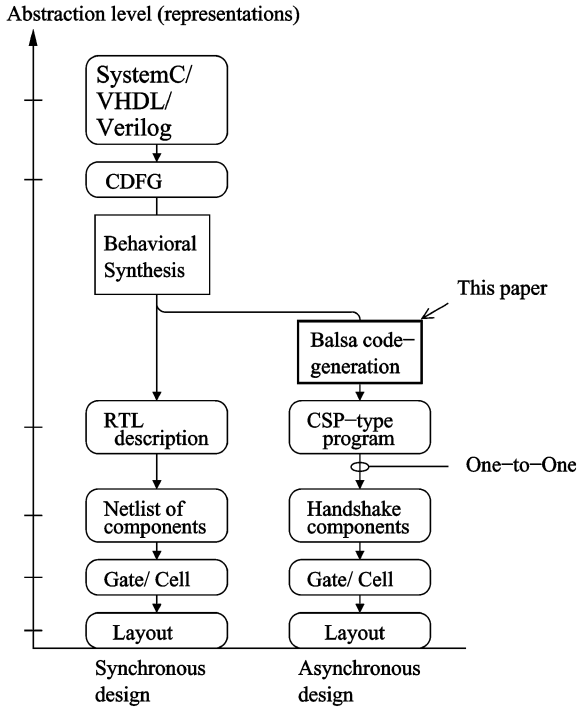

Fig. 1. Existing synchronous and asynchronous design flows and the synthesis process addressed in this paper.

The first step in behavioral synthesis is to extract from the program text an intermediate representation in which nonessential constraints (such as sequencing of independent statements in the program text) have been omitted. Some form of control data flow graph (CDFG) is normally used for this. From the CDFG, the classic synthesis tasks [8] of scheduling, allocation, and binding are performed, resulting in an register transfer level (RTL) circuit description which is then synthesized into gate-level circuits and eventually to layout.

The work presented in this paper represents an attempt to provide a similar behavioral synthesis flow for asynchronous circuits, where optimizations are performed by the synthesis tool rather than by the designer.

The novelty of the approach is: 1) the idea of adapting traditional synchronous circuit synthesis techniques to an asynchronous circuit design flow based on syntax-directed translation and 2) the implementation templates for the control unit and the datapath, and the method which is used to derive these.

Our work takes advantage of the one-to-one mapping from a Balsa-description to circuit, because it allows us to express the

Manuscript received November 29, 2006; revised June 04, 2007, September 14, 2007, and December 18, 2007. Current version published January 14, 2009.

The authors are with the Department of Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark (e-mail: sfn@imm.dtu.dk; jsp@imm.dtu.dk; jan@imm.dtu.dk).

Digital Object Identifier 10.1109/TVLSI.2008.2005285

implementation at a high level avoiding altogether the complex problem of specifying and synthesizing an asynchronous controller [9]–[13] and avoiding the many low-level details involved in generating a standard cell netlist of the implementation. The techniques described in this paper can therefore also be seen as a front-end tool for Balsa and similar tools.

The CDFG and its extraction from a behavioral description is well understood [8] and will not be addressed in this paper—the existence of a CDFG is assumed. This paper reviews how the classic synthesis algorithms used for scheduling, allocation, and binding are modified to a fine-grain discrete-time domain suitable for asynchronous design and it elaborates on how we generate and describe implementation of the datapath and the associated controller in Balsa. Early work on our approach was published in [14]. This paper provides a much more comprehensive description of the approach and in particular the implementation template and the procedure used to produce the Balsa program resulting from the synthesis. The current synthesis flow also implements an important optimization resulting in less control overhead in the synthesized circuits. Furthermore, this paper provides a larger set of benchmark results, including a couple of circuits for which standard-cell implementations have been generated and characterized.

This paper is organized as follows. Section II discusses related work. Section III gives a review of our model, the input format, and the implementation template. Section IV presents the procedure for the Balsa code generation. Section V presents a range of benchmarks and results and finally Section VI concludes the paper.

II. Related Work

The introduction mentioned a number of asynchronous high-level synthesis tools. Haste [3], [4] is a proprietary tool of Handshake solutions, a spin-off from Phillips Semiconductor. This is quite mature and has been used to design circuits which are currently in production. Balsa [15] is a somewhat similar tool which has been developed at the University of Manchester and which is available in the public domain. These tools are based on syntax-directed translation where there is a one-to-one correspondence between the program source and the resulting circuit and where the control is highly distributed. A related approach is used in [16] where a program source is decomposed into a set of basic processes for which there is an asynchronous circuit representation. TAST [7] and in particular ACK [6], 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 [17]–[20]. The first paper limits itself to DFGs and focuses 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 is specified at the level of individual signal transitions (in the form of signal transition graphs or burst-mode state graphs). The partitioning and optimization of such a structure of distributed controllers is the subject for [21].

The work presented in [22] targets high-performance rather than resource sharing and thus has the opposite goal to our project. Their approach is based on mapping each node (or group of nodes) of the CDFG onto a micro-pipeline stage, creating a highly pipelined circuit.

A somewhat recent paper [23] presented a method with a very similar aim to ours. The work extends an existing synchronous behavioral synthesis system, MOODS, so that it can handle asynchronous circuits as well. However their focus is mainly on the opportunity for average-case performance and on the implementation of the asynchronous components used in the datapath and in the controller. The paper provides little insight into the underlying synthesis procedure.

Our approach is different from the above related works 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 in the synthesis process; 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 translation, 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 with simpler ones [4], [24], while other approaches involve resynthesis from a specification of the behavior of one or more handshake components into a more efficient implementation [25]. At all events, this work is orthogonal to the work presented in this paper where focus is on high-level synthesis.

III. Behavioral Synthesis

This section briefly reviews the elements of traditional synchronous behavioral synthesis and discusses the implications for adapting these methods to the synthesis of asynchronous circuits.

A. Review of the Classic Approach

Most behavioral synthesis tools make optimizations based on a CDFG which is extracted from a behavioral specification of the circuit behavior [8].

Our CDFG format is similar to that found in [8], [26], and [27]. Fig. 2 shows an example fragment of behavioral code (in Balsa syntax, augmented with a multiplier operator) and the corresponding CDFG. Here we are only considering a single process and limiting ourselves to a form where external synchronization is not modeled by the CDFG, i.e., input data has to be ready at the input edges to the CDFG and output results are taken from the output edges of the CDFG. The extraction of the CDFG from such a behavioral description is discussed in [26] and [28] and is outside the scope of this paper.

```
import [balsa.types.basic]
type word is 16 bits

procedure EX(input X0,X1,X2:word;
       output Y0,Y1:word) is
  variable x0,x1,x2,y0,y1:word
  constant a0=255
  constant a1=255
  constant a2=255
  constant a3=255
begin
  loop
    X0->x0 || X1->x1 || X2->x0;
    y0:=((a0+x0)+(x0*x1)-a1) as word;
    if (x1>a2) then
      y1:=(a3*(x1+x2)) as word
    else
      y1:=(x1-x2) as word
    end;
    Y0<-y0 || Y1<-y1
  end
end
```

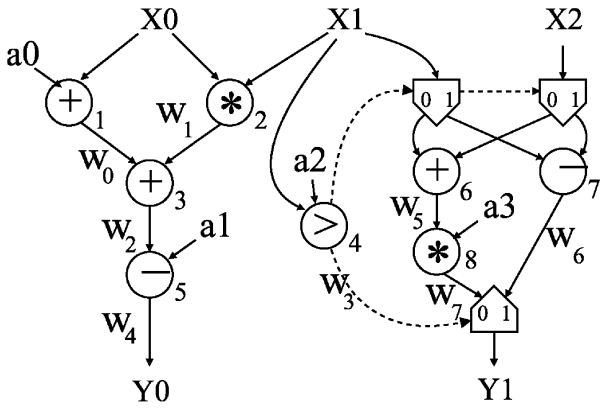

Fig. 2. Example program text and the corresponding CDFG. In the CDFG, the different operators are identified by an index $(1, \ldots, 8)$ and the different intermediate results by labels $(w_1, \ldots, w_7)$.

The target for behavioral synthesis is a hardware architecture as shown in Fig. 3 consisting of: 1) a datapath which is able to perform a set of operations on variables stored in a set of registers and 2) a controller which controls the execution sequence of these operations and the reading from and writing into registers in order to perform a given computation. The selection logic consists of a multiplexor selecting the proper inputs for the functional units and the registers.

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.

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

- Scheduling, in which operator nodes of the CDFG are scheduled into the time-slots defined by the clock signal. Fig. 4(a) shows a possible schedule for the example assuming a datapath with an arithmetic logic unit (ALU) and a multi-cycle multiplier.
- Allocation, in which the set of functional units (FUs) required for execution of the operator nodes and the number of registers needed to store input operands and intermediate results are determined.
- Binding (or assignment), where individual operator nodes and (intermediate) variables are tied to specific hardware resources, also shown in Fig. 4(a).

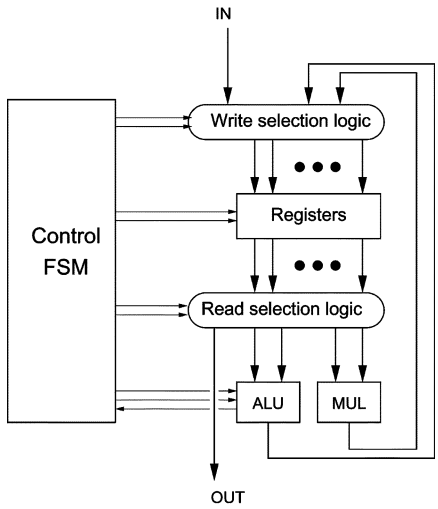

Fig. 3. Generic (synchronous) datapath.

B. Asynchronous Template Architecture

This paper presents an asynchronous implementation template in which the structure of the datapath is very similar to the synchronous implementation in Fig. 3. This is what enables us to use the same high-level synthesis approach, which for the running example results in the solution illustrated in Fig. 4. Fig. 5 shows an example with only two registers/variables and one functional unit, in order to allow a complete illustration of the read and write selection logic (cf. Fig. 3). The details are explained in the following.

Despite the structural similarity between the synchronous and the asynchronous datapath, the asynchronous template architecture is fundamentally different in the way the sequencing of operations is controlled, and the way in which the controller is implemented.

In a traditional synchronous implementation (see Fig. 3) a central controller state machine orchestrates the necessary operations in an *implicit* or *indirect* way. Operands or temporary results, which are to be written into registers, are assumed to be present at the end of the scheduled clock cycle and operands and function codes for functional units are assumed to be available from the beginning of the scheduled clock period. Fig. 4(b) illustrates this by showing a fraction of an "unrolled implementation."

In an asynchronous circuit, there is no notion of a global and discrete time. Instead all communication and synchronization

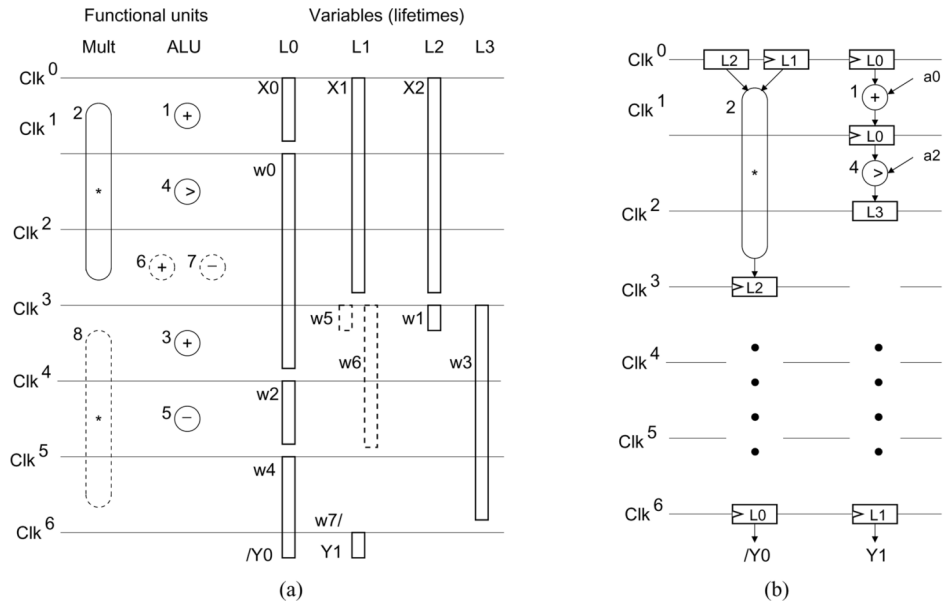

Fig. 4. (a) Possible schedule of the operations and intermediate variables in Fig. 2. The dashed nodes indicate conditionally executed operations. (b) Partial illustration of an "unrolled implementation".

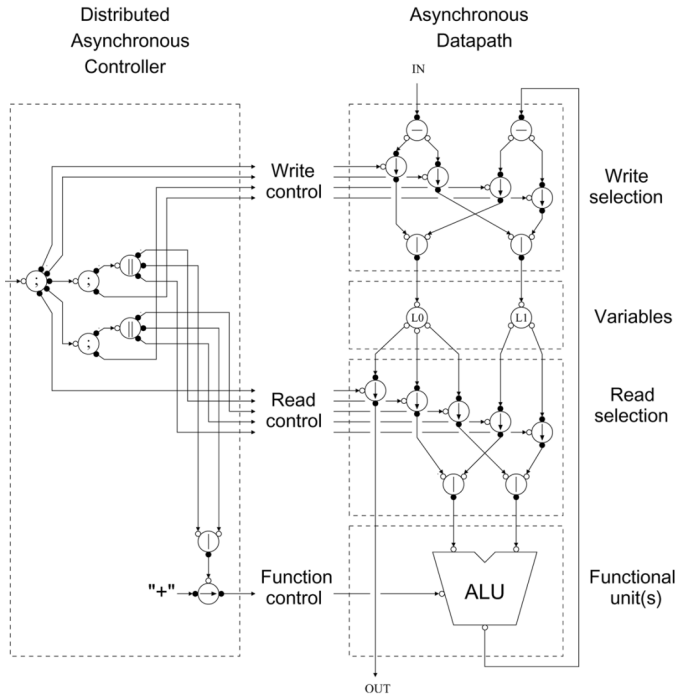

Fig. 5. Our asynchronous datapath template.

occurs using local point-to-point handshake channels. Functional units may start to compute at any time when their inputs are ready, and likewise they may finish at any time. In a similar way, relevant registers are written when the relevant operands are ready. Another difference is that the controller in Fig. 5 is implemented in a distributed fashion using handshake components (see the Appendix) such as: sequencer, parallelizer, join, fork, and miscellaneous conditional sequencing elements.

The essence of this is that all the required (individual) partial orderings of reads, operations, and writes are implemented *directly*, and it is thus the collective behavior of the handshake components which implements the schedule. This gives the asynchronous implementation a larger set of possible behaviors, as it adapts to the actual (and possibly data-dependent) latencies of the different operations; only the required partial orderings are enforced and in a (timeless) fashion.

Let us now revert to the detailed implementation of the asynchronous template. The example in Fig. 5 shows a datapath with only two registers/variables and one functional unit in order to allow a complete illustration of the selection logic. The asynchronous datapath is composed of only five types of handshake components: variables, operators, transferrers, mergers, and demultiplexors, as shown in Fig. 5. Readers who are not familiar with handshake channels (push/pull, protocol, etc.) and handshake components are referred to the Appendix. Note that the demultiplexor does not have a control input, it is more like an "inverse" merge. Note also the elegant way in which handshake components mix push and pull channels in such a way that the transfer-components alone control the flow of data. The template uses a lot of transfer-components, but it should be kept in mind that these are implemented using wires only! The datapath implementation is thus quite efficient. The controller can start an operation by transferring operands to its inputs and it can control the writing of results into the variables by activating the proper transferrer. Notice that all signaling among components (including data-less control) is implemented using handshake channels.

In conclusion our asynchronous implementation template has a number of interesting properties.

- It is possible to use existing synthesis techniques developed for synchronous design almost directly.
- It implements the schedule directly by explicitly implementing the necessary partial orderings of operand reads, operations and operand writes.

- It allows us to describe the implementation of the circuit at a high-level by taking advantage of the one-to-one mapping performed by Balsa and thus avoiding interference between high-level and low-level optimizations.
- The controller is implemented as a structure of simple handshake components. In this way, our method avoids the difficulty of synthesizing a centralized controller whose state space would typically be very large due to the many possible orderings of the read and write control signals to which it must respond.

C. Adaptation to Asynchronous Design

Our current work focuses on the classic synthesis problem of minimizing circuit area under a certain execution time constraint T, but there is nothing which prevents us from optimizing for other goals (energy or execution time).

Our synthesis flow from CDFG to a schedule follows the standard synthesis flow. First, the algorithm enters an analysis phase in which a compatibility graph [8] is created in order to deal with conditional execution, and the as soon as possible (ASAP) and as late as possible (ALAP) schedules are computed [8] in order to reduce the search space.

To perform behavioral synthesis our method uses the metaheuristic Simulated Annealing algorithm [29], [30]. To represent solutions for the Simulated Annealing algorithm, a solution vector containing n tuples (one for each operator), consisting of the pair: *(StartTime,FUassignment)*, is used. The objective function driving our algorithms is based on an estimate of the area cost, which is currently approximated by

$$A \approx A_{\mathrm{FUs}} + A_{\mathrm{Latches}} + A_{\mathrm{Muxes}}.$$

Our method ignores A_{Control} as it is substantially smaller than the others, since we are limiting ourselves to data-processing dominant circuits with a single repetition. Our method does not yet estimate $A_{\mathrm{Interconnect}}$ (wiring). However with the shrinking of gates, this becomes an important part of the total area, especially in connection with shared resources which gives rise to an increase in wire count.

Our method assumes a library of asynchronous handshake components from which to build our circuit. In this library, the FUs may be implemented as simple combinatorial circuits or they may include (normally opaque) input and output isolation latches. This choice has consequences for the lifetime of the intermediate variables. 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. In this paper, it is assumed that the functional units normally have opaque latches on input and output ports. This is a somewhat arbitrary choice and has no fundamental implications for the approach or the synthesis algorithms. The use of input and output latches tends to increase speed and to reduce energy consumption by preventing spurious signal transitions from propagating beyond latch boundaries. Synthesis algorithms for both situations are presented in [31].

In the rest of this paper, it is assumed that behavioral synthesis has been performed and thus the starting point is an allocation of resources and a scheduled and resource-assigned CDFG.

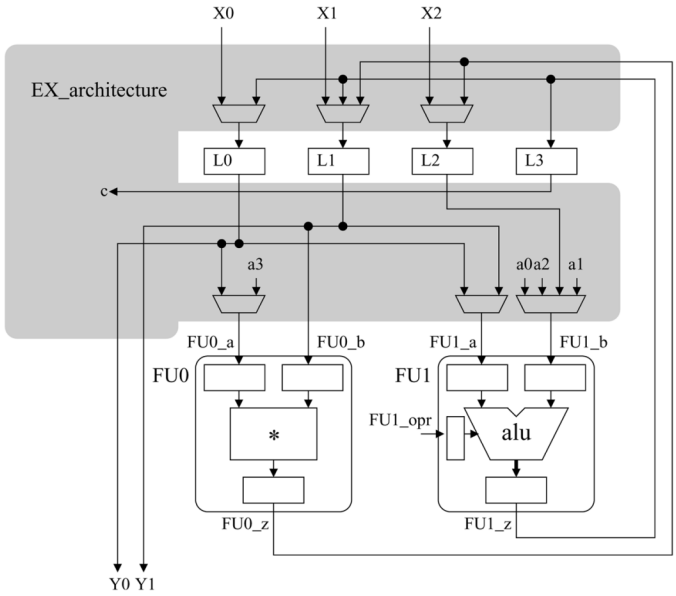

Fig. 6. Datapath model for our running example.

IV. Balsa Code Generation

The central step in our asynchronous synthesis flow is to (automatically) generate a Balsa program describing the circuit to be implemented using the template shown in Fig. 5.

The synchronous schedule, operator to functional unit assignment, and the variable to latch assignment generated by our behavioral synthesis algorithm, shown in Fig. 4(a) corresponds to the data-path model, shown in Fig. 6. This is the "machine" the behavioral synthesis algorithm has found to be optimal to execute the CDFG under the given constraints. The controller for this circuit implements the schedule and starts the FUs with the right data at their designated times. Our approach combines the controller and the routing into a component which is designated EX_architecture. Before presenting the algorithm for code generation, the Balsa code generated by this algorithm is discussed.

A. Balsa Program for our Example

The resulting Balsa program for the scheduled CDFG is shown in Fig. 7 and is explained in the following. "Labels" have been added to support the text.

The program starts by including a library of functional units (label A). These FUs are themselves implemented as Balsa procedures (a procedure in Balsa is like an entity in VHDL).

After the include statements, there is a procedure EX, which implements the synthesized circuit (label B). It has input channels X0, X1, and X2 and output channels Y0 and Y1. Next, follows a declaration part in which the internal variables, the internal channels to and from the FUs and possible constants are declared. The variables synthesize into a corresponding set of variable components (a.k.a. handshake latches), cf. Fig. 5.

Inside the EX-procedure (at label C), the functional units of our datapath together with the EX_architecture are instantiated (in parallel). The symbol "||" denotes the parallel compo-

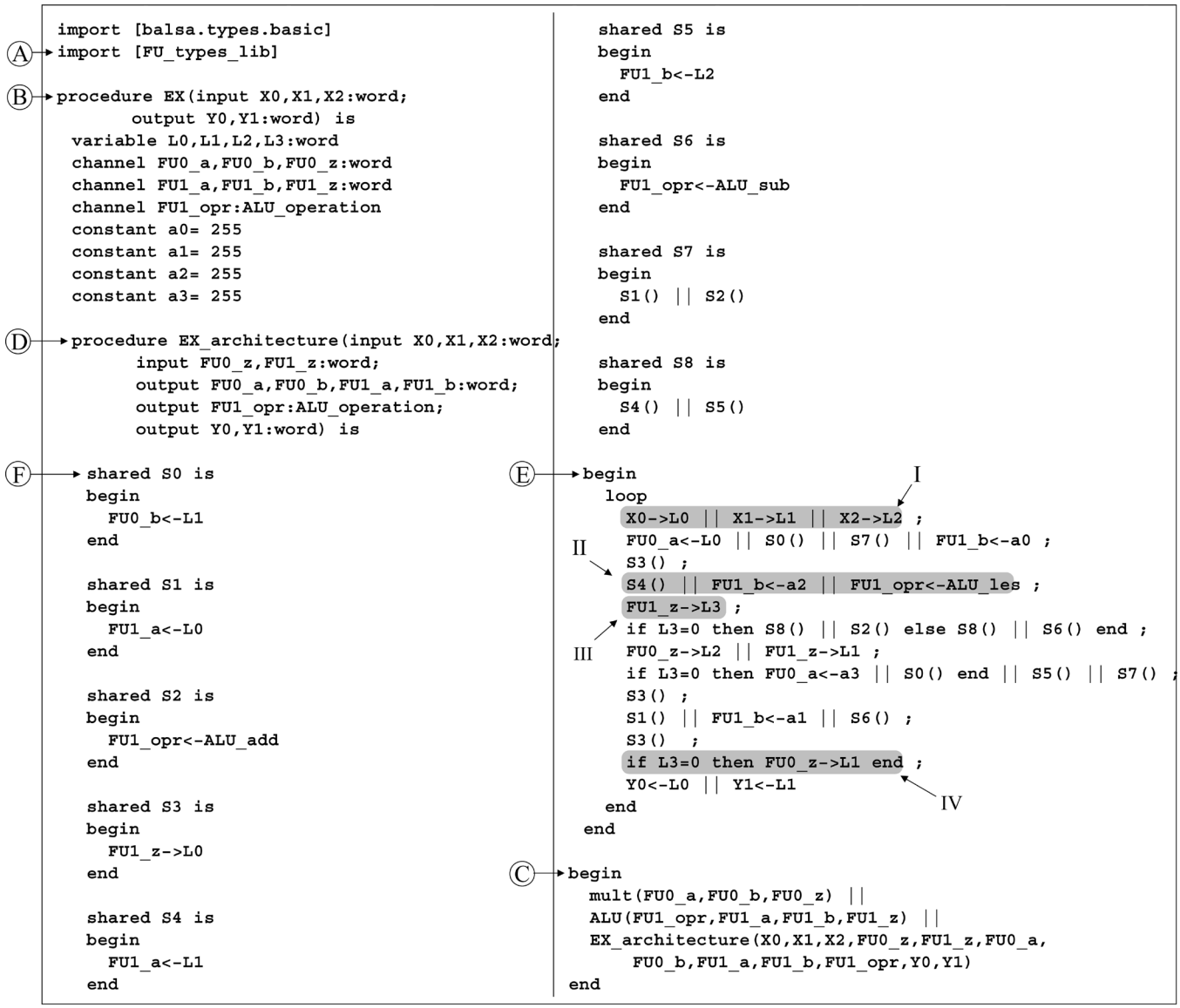

Fig. 7. Synthesized Balsa program for our running example.

sition operator. The result is three physical circuits operating in parallel.

The EX_architecture procedure (label D) implements the controller and the read and write selection handshake components of the datapath connecting the handshake FUs with handshake inputs, handshake latches and handshake outputs. The body of the loop (label E) consists of simple assignment statements composed by sequential (";"), parallel ("||") and if-then-else constructs. This part represents the combined operation of the datapath and its associated controller. There are two types of assignments: one transferring operands from the variable latches, L0, L1, L2, and L3 to the inputs (input latches) of the FUs and another which transfers results from the outputs (output latches) of the FUs to the variable latches. All these simple assignments are "implemented" by the transfer components in the write selection and operand selection parts of Fig. 5. The transfer components are controlled by the control unit which is represented by (and synthesized from) the structure of the code in the loop body, i.e., the way in which the assignment statements are composed using sequential (";"), parallel ("||") and if-then-else constructs.

If more than one assignment uses the same handshake component as target, the Balsa compiler automatically synthesizes a merge component to handle the "joining" of channels to that handshake component. Likewise, if more than one assignment uses the same handshake component as a source, the Balsa compiler automatically synthesizes a demultiplexor component to handle the "forking" of channels from that handshake component.

During the execution of the loop body it is possible that the same assignment occurs several times, involving the same pair of source and target handshake components. This would lead to the synthesis of multiple channels between the same components, adding unnecessary (demultiplexor and) merge components, as illustrated in Fig. 8 (left). This is an unwanted consequence of the syntax-directed translation. Here a more efficient

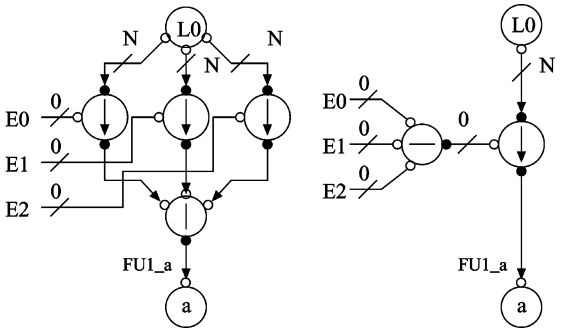

Fig. 8. Repeated use of hardware resources (left) without and (right) with shared construct.

topology is used where the control channels to the transferrers are merged first, as illustrated in Fig. 8 (right). For each assignment that occurs several times, the algorithm creates a shared construct and replaces the assignment with a call to this shared construct. The shared constructs are placed before the body of procedure EX_architecture (label F).

Let us now take a closer look at some representative sections of the code in the loop body. The assignments in the highlighted area labeled I, receive in parallel (the "||" symbol) inputs X0, X1, and X2 into latches L0, L1, and L2. The assignments in II initiate the ALU operation (the conditional test of) $L1 > a2$? by reading from L1 (using a shared construct S4) and the constant a2 and giving the ALU a perform "les" command, corresponding to the start of operator 4 in the CDFG. The assignment in III reads the output of the ALU and stores the result in latch L3, corresponding to the end of operator 4 in the CDFG.

The statements in IV implement a conditional construct. Depending on the result of a previously performed comparison whose result is in L3, the code performs the following: if $L3 = 0$, it stores the result of multiplication $L1{*}a3$ in L1.

Fig. 9 shows the datapath implemented by the Balsa program in Fig. 7, and Fig. 10 shows the structure of the controller. The output-channels on the right-hand side of Fig. 10 are the read, write, and FU control channels, which come in on the left-hand side of Fig. 9. For these, the corresponding statements can be easily identified in the source code. For conditional statements the controller needs to read and compare variables in the datapath and therefore some channels are going from the datapath to the controller. In the program, there are several occasions of comparisons involving L3 and in the figures the associated channel has been labeled c. This channel does not explicitly appear in the source code but is a label which is manually added.

The corresponding handshake components to the selected statements (I, II, III, and IV) have been highlighted in Fig. 10 and the selected statements (II and III) have been highlighted in Fig. 9.

Algorithm for Code Generation

The Balsa code generation is a mapping of the read and write assignments found in the schedule, using the template, to a sequence of Balsa statements. No design-space exploration is performed, only a finite set of local optimizations.

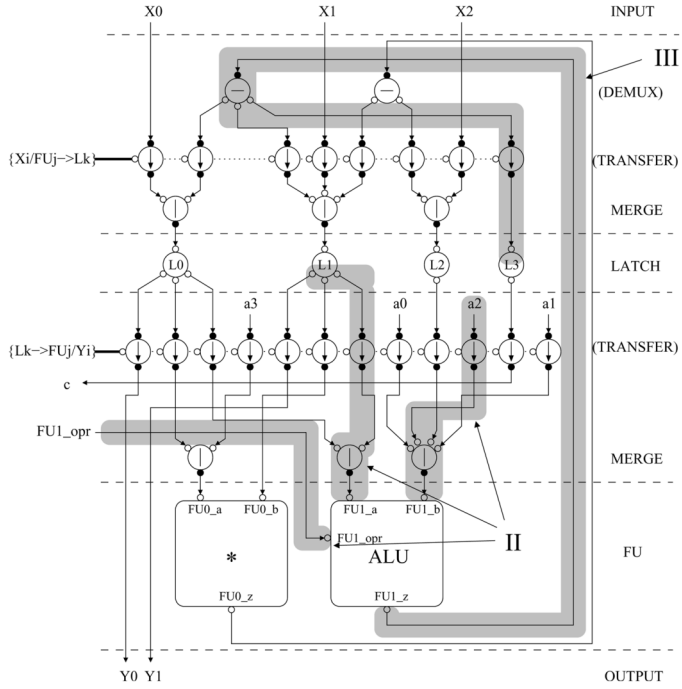

Fig. 9. Final datapath for our scheduled CDFG using handshake components.

Input to the code generation algorithm is as follows.

i) A description of the external input and output channels to the procedure.

ii) A description of the machine, shown in Fig. 6, in the form of the number and type of functional units and the required set of latches.

iii) An annotated CDFG, where each *node* has been augmented with a start time and an FU assignment and each *arc* has a latch assignment and a port-mapping; the nodes which are conditionally executed have information about which arc contains the Boolean condition for it to be executed; the nodes are grouped into re-executional body's (i.e., groups of nodes which may be re-executed) and the information of the arc containing the condition for this is also specified.

The pseudo-code for generating the Balsa-code is shown in Fig. 11 and explained in the following.

Step 1) Start by writing the include for the FU_lib and write the main procedure head with the inputs and outputs for the CDFG (found in `ExternalChannels`). Functions starting with `Output...` are functions which write the output Balsa-program text.

Step 2) Instantiate the required variable latches (found in the `LatchAllocation` list) and for functional units declare the internal channels to and from these (using the `FUallocation` list and the FU libraries), and state the constants used in the CDFG (`OutputConstants` searches through the CDFG and declares them individually). Then write the procedure architecture head with the previously mentioned internal channels and using the name of the procedure.

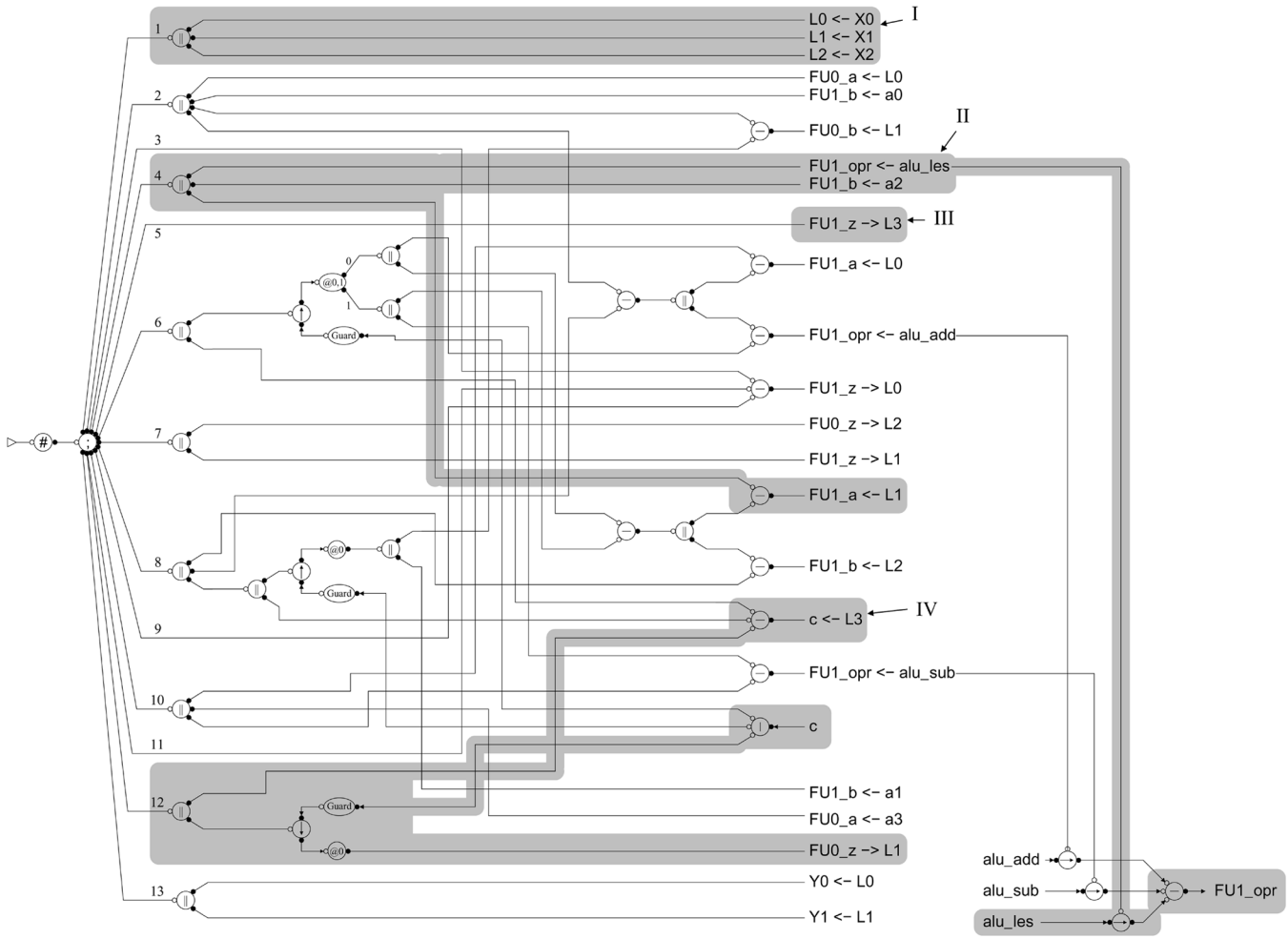

Fig. 10. Final control unit for the scheduled CDFG using handshake components.

Step 3) Run through the schedule and resource allocations (each node of the annotated CDFG) and construct the time-ordered list of assignments (read and write) that are occurring. Each node has a list of input arcs and only one output arc for which there is an assignment. Read-assignment for input arcs and write-assignment for the output arc. The `AssignmentList` is a list of tuples in the form of *(Time, Assignment)*. The time for read-assignments is the scheduled start time of the node and the time for a write assignment is the node start time plus the delay of the FU it has been assigned to.

Step 4) Run through the generated list of assignments and for each reoccurring assignment declare the corresponding shared construct. This is extended to include pairs of time-simultaneous reoccurring assignments. The reoccurrence is found by running through all assignments and counting their use. Those which are used more than once are shared. The `SharePairList` is a list of tuples of the form: *(Time, AssignmentA, AssignmentB)*. `OutputSharedProcedure` is responsible for declaring a shared procedure with a single assignment and `OutputSharedPairProcedure` for pairs.

Step 5) The architecture consists of a "statemachine" wherein each executable body is a "state". The running example only has a single (re-)executable body so this is not seen there. Each executable body consists of a sequence of conditional Balsa-statements. First the list of assignments is partitioned into these bodies. If there is more than one, declare the state-variable needed and output the begin of the architecture. For each of the bodies run through the list of assignments in time order and declare the statements, using calls to the shared constructs if applicable. Assignments, that are simultaneous in time are declared using the "||"-operator, other assignments are sequenced by the ";"-operator. `TimeSortAssignments` partitions the nodes in the CDFG into a sequence of assignments to be executed in parallel. Some of these group assignments have common conditional execution and must therefore be grouped together in the Balsa-statement. `PartitionConditional` takes care of this partition. `OutputConditionalAssignments` is a

```
OutputLibraries();
OutputProcedureHead(Name,ExternalChannels);                                  Step 1

OutputLatches(LatchAllocation);
OutputChannels(FUAllocation);
OutputConstants(CDFG);
OutputArchitectureHead(Name,ExternalChannels,FUAllocation);                  Step 2

AssignmentList = ConstructAssignmentList(CDFG);                              Step 3

ShareList = ConstructShareList(AssignmentList);
SharePairList = ConstructSharePairList(AssignmentList);                      Step 4

OutputSharedProcedure(ShareList);
OutputSharedPairProcedure(SharePairList);

BodyList = ExecutableBlockPartition(AssignmentList);                         Step 5

OutputStateVariables(BodyList);
OutputArchitectureBegin();
OutputStatemachine(BodyList);

Foreach (Body in BodyList) {

     OutputCaseStatement(Body);

     Sequence  = TimeSortAssignments(Body);

     Foreach (GroupAssignment in Sequence) {

          ConditionalAssignment = PartitionConditionally(GroupAssignment);

          Foreach (ParallelAssignment in ConditionalAssignment) {

               OutputParallelAssignment(ParallelAssignment);
               OutputParallel(ParallelAssignment,ConditionalAssignment);
          }

          OutputSeqencer(GroupAssignment,Sequence);
     }

     OutputStateChange(Body);
}
OutputStatemachineEnd();
OutputArchitectureEnd();

OutputBegin();                                                               Step 6
OutputFUCalls(FUAllocation);
OutputArchitectureCall(Name,Externalchannels,FUAllocation);
OutputEnd();
```

Fig. 11. Pseudo-algorithm for code generation.

function which declares a group of parallel assignments and puts them inside a conditional statement depending on the common node execution condition. As an example assume the condition-code is "$w3 = 0$" (with w3 mapped to L0) for FU0_z assigning to L1 then the function declares the statement: "if $L0 = 0$ then $FU0\_z- > L1$ end". If the condition-code is the "unconditional" element the assignments are simply declared.

Step 6) Finally, construct the main procedure using instantiations of the allocated functional units and the architecture.

V. Results

In order to demonstrate the feasibility of the proposed approach, and in order to evaluate the efficiency of the proposed implementation template, we present results for different synthesized versions of a range of benchmark circuits: GCD, FIR, HAL (a.k.a. DIFFEQ), ELLIPTIC, and COSINE. GCD are reported on and described in the Section V-A. The main purpose is to illustrate the benefits of behavioral synthesis over plain syntax-directed translation. Section V-B presents results for the remaining benchmarks. For FIR and HAL, we have produced layouts and in Section V-C we report on the area, speed, and energy figures.

TABLE I
FU Library (16 Bit) Based on Layout in 0.18-μm Technology. The Normalized Figures are Used by the Scheduling Algorithm

FU	*Operation*	Latency	Area	Energy
ALU	$\{+,-,>\}$	25.5	0.0112	0.0266
Mult	$\{*\}$	56.3	0.105	0.314
ALU	$\{+,-,>\}$	1	1	1
Mult	$\{*\}$	2.6	10	13

A library of asynchronous FUs with the worst-case latency, area and energy parameters shown in Table I is used. The multiplier is a radix-2 Booth-multiplier which has been manually coded in Balsa. The figures are normalized with respect to the ALU.

But first, this paper reports on the area cost of our running example. A one-to-one circuit-mapping of the CDFG in Fig. 2 would have an area estimate of 79 846.0 ("Balsa-cost"), whereas the synthesized Balsa-code, shown in the previous section, has

TABLE II
AREA "BREAKDOWN" IN PERCENTAGE OF OUR RUNNING EXAMPLE FOR BOTH THE ORIGINAL SOURCE CODE AND THE SYNTHESIZED CODE

EX_org		
Controller	Datapath	
3.01%	Variables	2.48%
	Routing	0.76%
	FUs	93.8%

EX_synt		
Controller	Datapath	
6.78%	Variables	7.93%
	Routing	11.9%
	FUs	73.4%

an area estimate of 49 959.0 ("Balsa-cost"); leading to an area reduction of 37%. "Balsa-cost" is an area measure reported by the Balsa tool and represents a dual rail implementation in an 0.36-μm technology. Table II shows area "breakdowns" of both of these circuits into control, variables, routing, and FUs.

For the original circuit without any resource sharing, it is the FUs which are the major area contribution. This is largely due to the type of CDFG, which is dominated by computation. The contributing factor of the controller is the compare circuit in the conditional expression.

For the synthesized version, we see what we would expect from behavioral synthesis of synchronous circuits. Resource sharing the FUs leads to an increase of the storage and routing in the datapath, and a modest increase in control area. The FUs are still the major component and this is due to the multiplier. If the Balsa-coded multiplier is replaced by a more efficient implementation, we would expect the FUs area contribution to be somewhat closer to the combined variable and routing components. We also observe that the area used by the controller still constitutes a small part of the overall circuit area, as is the case in behavioral synthesis of synchronous circuits. The control area is dominated by the sequencer, the guard, and the choice handshake components.

It should be emphasized that the area reduction obtained in this example is primarily gained by sharing the two multipliers. Resource sharing two adders literally means replacing one of the adders with a multiplexor and a latch which in combination have almost the same area as the adder our method is trying to optimize away. We therefore do not expect large area reductions in circuits which are fully composed of the simple logic-arithmetic operations; addition, shift, compare, subtraction, AND, OR, XOR, and NOT.

A. GCD

In [32, sec. 13.2.3] the process of syntax-directed translation and optimizations at the source code level (using Tangram) is illustrated using GCD as an example. Fig. 12 shows the well known algorithm expressed in Balsa code. The problem is that the source code contains 4 operator symbols, and that the corresponding circuit will have four functional units as well. In order to optimize the area, the designer has to rewrite the code. Fig. 13 shows one such optimized design. It is slightly different from the Tangram code in [32] as Balsa does not support exactly the same constructs as Tangram, but the ideas underlying the optimization are the same. Even this simple example hints that the process

```
import [balsa.types.basic]
type word is 16 bits

procedure gcd(input a,b: word ; output c: word) is
  variable ai,bi : word
begin
  loop
     a -> ai || b -> bi ;
     while ai/=bi then
       if ai>bi then
         ai:=(ai-bi as word)
       else
         bi:=(bi-ai as word)
       end
     end ;
     c<-ai
  end
end
```

Fig. 12. GCD-algorithm.

```
import [balsa.types.basic]
type word is 16 bits

type twoword is record
  a,b:word
end

procedure gcd(input ab: twoword ; output c: word) is
  variable data : twoword
begin
  loop
     ab->data ;
     while data.a/=data.b then
       if data.a>data.b then
         data:=(twoword {((data.a-data.b) as word),
                                    data.b as word)})
       else
         data:=(twoword {data.b,data.a})
       end
     end ;
     c<-data.a
  end
end
```

Fig. 13. Optimized version of GCD.

TABLE III
COMPARISON OF THE PLAIN ORIGINAL GCD, THE "HAND"-OPTIMIZED GCD AND THE SYNTHESIZED GCD

Program	Balsa-cost
gcd_org	7435.25
gcd_hand	7161.75
gcd_synt	6846.00

of optimizing the circuit and exploring alternatives *can* be tedious. In behavioral synthesis one would take the basic code in Fig. 12 and synthesize it with area minimization as the constraint. The work presented in this paper does exactly this, i.e., from a CDFG extracted from the basic code in Fig. 12, we automatically synthesize a circuit containing only one functional unit, an ALU. Table III shows the area estimates ("Balsa-cost") reported by Balsa for the different versions of the circuit. It is seen that behavioral synthesis in this example actually outperforms the manually optimized design.

B. Benchmarks

In a similar way, we have synthesized a range of benchmarks as shown in Table IV. FIR is an eight-tap FIR filter. HAL is an

TABLE IV
BENCHMARK RESULTS. COLUMNS ALU, MUL, AND LT LISTS THE NUMBER OF ALUS, MULTIPLIERS, AND LATCHES IN THE CIRCUITS

	Original code				Synth. code in/output latch				Synth. code no in/output latch			
Program	ALU	Mul	Lt	Balsa-cost	ALU	Mul	Lt	Balsa-cost	ALU	Mul	Lt	Balsa-cost
FIR	7	8	0	459749.25	1	2	12	140535.00	1	2	9	132635.75
HAL	5	5	0	348093.75	1	1	16	81381.75	1	1	11	78601.75
ELLIPTIC	26	8	0	518017.75	2	2	23	143248.75	2	2	19	150256.00
COSINE	26	16	0	964470.25	4	2	32	160889.25	4	2	17	155626.75

TABLE V
LAYOUT RESULTS

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 VI
MODEL RESULTS

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

iterative Euler integration of a differential equation. ELLIPTIC is a fifth-order elliptic wave filter. COSINE is a part of the DCT algorithm. Again, the area is expressed in terms of the "cost" reported by Balsa. As seen, it is possible to automatically synthesize implementations using the smallest possible number of functional units. The next question is how efficient these implementations are. To answer this question layouts for FIR and HAL are produced.

C. Layout Results

For the benchmarks FIR and HAL, we have used the backend part of the Balsa tools and actually produced a layout targeting handshake components using the single-rail four-phase early protocol. We have used the 0.18-μm STM standard-cell technology, which has been augmented with standard cell components for implementing various special asynchronous components such as Muller C-elements. Simulation results are obtained by simulating the post place-and-route Verilog netlist together with extracted layout information in NanoSim. We simulate 200 computations using random numbers without any correlation. NanoSim reports the total time for execution and the total energy used. All the circuits are implemented using 16-bit variables and are simulated at 1.8 V and at a temperature of 25 °C.

It is important to stress the results do not represent 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 benchmark results are shown in Table V, 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 I. The speed figures in Table I have been used in calculating the schedules.

Implementations 1 and 3 in Table V are the direct non-resource shared circuit implementations of the computations and the others are with resource sharing. 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 combinational depth of the circuit and thus reduces the energy consumption, which leads to a more fair comparison. The speed figures in Table V include a 20-ns handshake delay in the testbench used to simulate the layouts.

The results in Table V show 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. But here the energy reducing effect of the opaque latches surrounding the functional units is countering this.

In order to estimate the overhead of the control circuitry which is introduced by resource sharing, we have computed the cost of an *ideal* resource shared implementation, i.e., a model in which the added control has zero area. Such ideal figures are shown in Table VI and the difference between these figures and the actual layout figures in Table I is an estimate of the overhead.

Comparing the energy consumption of model and layout, the model does not take into account the "use" of the FUs, i.e., actual data and their correlations but assumes worst-case random number correlations everywhere. This leads to inaccuracies, e.g., energy consumption comparing for (id 2) FIR resource shared where the model (without overhead) exceeds the layout (with overhead). The FIR computation is a large number of multiplications with constants, these constant multiplications have less switching activity.

Overall comparing Tables V and VI 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 before). We find these results encouraging.

VI. CONCLUSION

This paper presented a design-flow for behavioral synthesis of asynchronous circuits; and more specifically, a method for synthesizing from a scheduled and resource-assigned CDFG representation into an optimized Balsa implementation. Key elements in this process are the implementation template and procedure used to produce the Balsa implementation. The

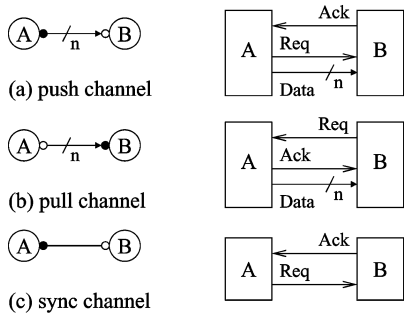

Fig. 14. Asynchronous handshake channels.

design-flow allows the designer to perform design space exploration by constraining the synthesis to using few(er) functional units.

Using this method and the Balsa and Cadence design tools, a number of benchmark circuits have been synthesized and for five of these, actual layouts have been produced and simulated. The results show that it is possible to do tradeoffs between area and circuit delay for asynchronous circuits. We find that there is a 10%–30% area overhead and a 0%–15% time overhead and no energy 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 includes automating the front-end part of the flow (i.e., automatic extraction of a CDFG from a behavioral description in some language), exploration and adaption of more synthesis algorithms, miscellaneous optimizations at the circuit level and last but not least, more and larger benchmarks once the flow is fully automated.

Finally, it is important to emphasize that the described design flow could be used in a number of ways. One is to synthesize asynchronous circuit implementations from behavioral descriptions expressed using existing industry-standard languages like: C, VHDL, Verilog, etc. Another use would be as a front-end for Balsa, which would allow the designer to automatically produce optimized implementations of some or all of the procedures in the design.

APPENDIX
HANDSHAKE COMPONENTS AND HANDSHAKE CHANNELS

In asynchronous circuits, components synchronize and exchange data on a local basis using handshake channels. This appendix briefly reviews various forms of handshake protocols and a basic set of handshake components. In doing so, the appendix also introduces the component symbols used in the schematics shown in Section IV. For further details refer to [4], [33], or to manuals for the Balsa and Haste/Tangram languages.

Handshake Protocols: Fig. 14 shows two components, A and B, connected by a handshake channel. The figure also shows the actual wires for a bundled data implementation of the different handshake protocols. Other encodings like dual-rail are possible.

A channel connects a port on one component to a port on another component. The sense of the port (active or passive) indicates the direction of the handshake. A filled circle indicates an active port and an non-filled circle indicates a passive port. An active port initiates a handshake (sends the request) and the passive port acknowledges requests. Channels can carry data in either the same direction as the handshake (a push channel) or in the opposite direction (a pull channel). The direction of the data flow is indicated by the arrow. Channels that carry no data are known as sync channels or activation channels as they are often used to start the operation of other handshake components.

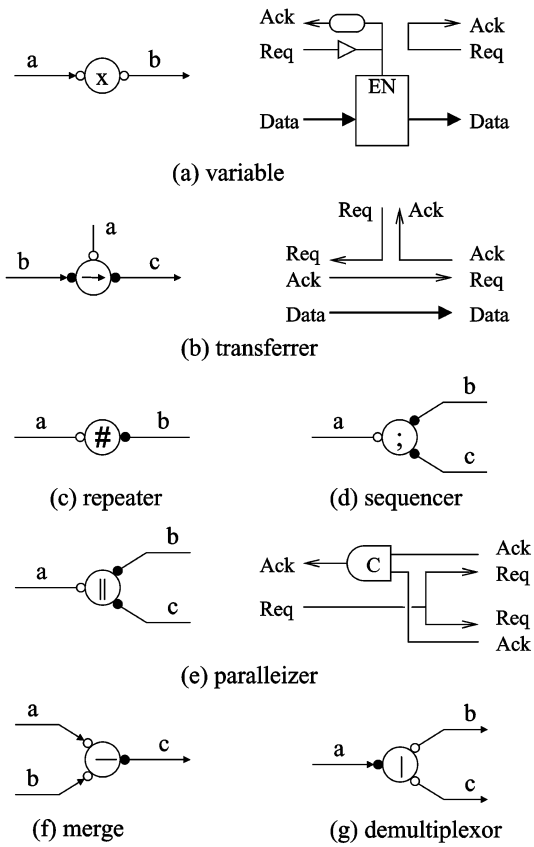

Fig. 15. Most important and frequently used handshake components. For components (a), (b), and (e) there is also shown a possible realization in four-phase bundled data early protocol.

To illustrate this, Fig. 14(a) shows a push channel, Fig. 14(b) shows a pull channel, and Fig. 14(c) shows a sync channel. In the push channel, the request signal indicates validity of the data and in the pull channel data validity is indicated by the acknowledge signal. A fourth type of handshake channel not shown in Fig. 14 is a bi-put channel where data is exchanged in both directions.

Handshake Components: EDA tools based on syntax-directed translation using CSP-like languages like Balsa and Haste perform one-to-one mappings from a program text into a corresponding structure of handshake components. In principle there is a handshake component for every syntactic element of the language. Fig. 15 shows the most important and most frequently used handshake components. The total number of different handshake components in the Balsa- and Haste-based tools is around 30–40.

Fig. 15(a) shows a *variable*. It has passive input and output ports. The environment is supposed to perform mutually exclusive reads and writes. The variable is implemented using a (transparent) latch and a small control circuit. A variable component implements a variable in the source language.

Fig. 15(b) shows a *transferrer* which can be used to write to or read from a variable. A handshake on the activation port *a* causes the transferrer to transfer data from its input port *b* to its output port *c*. The handshake on the activation port encloses the transfer. Note the use of active and passive ports. The transferrer implements assignments and channels communication in the source language.

Fig. 15(c) shows a *repeater*. Initiating a handshake on port *a* causes an unbounded sequence of handshakes on port *b*. The repeater component implements loop-forever constructs in the source language. All channels are dataless activation channels.

Fig. 15(d) shows a *sequencer*. Initiating a handshake on port *a* causes a handshake on port *b* followed by a handshake on port *c*. The handshake on port *a* encloses the sequence of handshakes on ports *b* and *c*. All channels are dataless activation channels. The sequencer component implements sequential execution of program statements in the source language.

Fig. 15(e) shows a *parallelizer*. It is similar to the sequencer, except that handshakes on ports *b* and *c* are performed concurrently.

Fig. 15(f) shows a *merge*. Handshakes on passive ports *a* and *b* are assumed to be mutually exclusive and the mixer simply relays a handshake on one of its input channels to the output channel *c*. Very often different statements in the source program assign values to the same variable. A merge component is used to merge these (mutually exclusive) assignments to the variable.

Fig. 15(g) shows a *demultiplexor*. Its function is to pass data from the input port *a*, to one of its two output ports *b* and *c*. The output ports *b* and *c* are passive and handshakes on these ports are assumed to be mutually exclusive. The input port is active. In this way input data is simply pulled through the demultiplexor to the relevant output port. Very often different assignment statements in the source program reads from the same output port. A demultiplexor component is used to provide this value to the relevant components.

We will limit the introduction of handshake components to the above set, and just mention that the final control and datapath circuits shown in Figs. 9 and 10 also make use of the following components: *guard* and *choice*.

REFERENCES

[1] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *Computer J.*, vol. 45, no. 1, pp. 12–18, 2002.

[2] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[3] C. H. K. v. Berkel, C. Niessen, M. Rem, and R. W. J. J. Saeijs, "VLSI programming and silicon compilation," in *Proc. Int. Conf. Comput. Des. (ICCD)*, 1988, pp. 150–166.

[4] K. van Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, ser. International Series on Parallel Computation. Cambridge, MA: Cambridge University Press, 1993, vol. 5.

[5] E. Brunvand, "Translating concurrent communicating programs into asynchronous circuits," Ph.D. dissertation, Comput. Sci. Dept., Carnegie Mellon Univ., Pittsburgh, PA, 1991.

[6] P. Kudva, G. Gopalakrishnan, and V. Akella, "High level synthesis of asynchronous circuit targeting state machine controllers," in *Proc. Asia-Pacific Conf. Hardw. Description Lang. (APCHDL)*, 1995, pp. 605–610.

[7] M. Renaudin, P. Vivet, and F. Robin, "A design framework for asynchronous/synchronous circuits based on CHP to HDL translation," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits Syst.*, Apr. 1999, pp. 135–144.

[8] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[9] 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," Columbia Univ., NY, Tech. Rep. TR CUCS-020-99, 1999.

[10] H. Jacobson, E. Brunvand, G. Gopalakrishnan, and P. Kudva, "High-level asynchronous system design using the ACK framework," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits Syst.*, Apr. 2000, pp. 93–103.

[11] P. Kudva, G. Gopalakrishnan, and H. Jacobson, "A technique for synthesizing distributed burst-mode circuits," in *Proc. ACM/IEEE Des. Autom. Conf.*, 1996, pp. 67–70.

[12] S. M. Nowick, K. Y. Yun, and D. L. Dill, "Practical asynchronous controller design," in *Proc. Int. Conf. Comput. Des. (ICCD)*, Oct. 1992, pp. 341–345.

[13] K. Y. Yun, D. L. Dill, and S. M. Nowick, "Synthesis of 3D asynchronous state machines," in *Proc. Int. Conf. Comput. Des. (ICCD)*, Oct. 1992, pp. 346–350 [Online]. Available: ftp://snooze.stanford.edu/pub/papers/async/YDN92.ps

[14] S. F. N. J. Sparsø and J. Madsen, "Towards behavioral synthesis of asynchronous circuits—An implementation template targeting syntax directed compilation," in *Proc. EUROMICRO DSC*, Aug. 2004, pp. 298–305.

[15] A. Bardsley and D. A. Edwards, "The Balsa asynchronous circuit synthesis system," presented at the Forum Des. Lang., University of Tuebingen, Germany, Sep. 2000.

[16] S. M. Burns, "Automated compilation of concurrent programs into self-timed circuits," M.S. thesis, Comput. Sci. Dept., California Inst. Technol., Pasadena, 1988.

[17] B. M. Bachman, H. Zheng, and C. J. Myers, "Architectural synthesis of timed asynchronous systems," in *Proc. IEEE Int. Conf. Comput. Des.: VLSI Comput. Processors (ICCD)*, Oct. 1999, pp. 354–363.

[18] J. Cortadella and R. M. Badia, "An asynchronous architecture model for behavioral synthesis," in *Proc. Euro. Conf. Des. Autom. (EDAC)*, 1992, pp. 307–311.

[19] J. Cortadella, R. M. Badia, E. Pastor, and A. Pardo, D. D. Gajski, Ed., "Achilles: a high-level synthesis system for asynchronous circuits," in *Proc. 6th Int. Workshop High-Level Synth.*, 1992, pp. 87–94.

[20] E. Kim, J.-G. Lee, and D.-I. Lee, "Automatic process-oriented control circuit generation for asynchronous high-level synthesis," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits Syst.*, Apr. 2000, pp. 104–113.

[21] M. Theobald and S. M. Nowick, "Transformations for the synthesis and optimization of asynchronous distributed control," in *Proc. ACM/IEEE Des. Autom. Conf.*, Jun. 2001, pp. 263–268.

[22] M. Budiu and E. A. G. Venkataramani, "Spatial computation," in *Int. Conf. Arch. Support for Program. Lang. Operat. Syst. (ASPLOS)*, Oct. 2004, pp. 14–26.

[23] M. Sacker, A. Brown, P. Wilson, and A. Rushton, "A general purpose behavioural asynchronous synthesis system," in *Proc. Int. Symp. Adv. Res. Asynch. Circuits Syst.*, Apr. 2004, pp. 125–134.

[24] G. Gopalakrishnan, P. Kudva, and E. Brunvand, "Peephole optimization of asynchronous macromodule networks," in *Proc. Int. Conf. Comput. Des. (ICCD)*, Oct. 1994, pp. 442–446.

[25] T. Chelcea and S. M. Nowick, "Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems," in *Proc. ACM/IEEE Des. Autom. Conf.*, Jun. 2002, pp. 405–410.

[26] L. Stok, "Architectural synthesis and optimization of digital systems," Ph.D. dissertation, Dept. Math. Comput. Sci., Eindhoven Univ. Technol., Eindhoven, The Netherlands, 1991.

[27] J. B. Dennis, "Data flow computation," in *Control Flow and Data Flow—Concepts of Distributed Programming, International Summer School*. Marktoberdorf, West Germany: Springer, 1984, pp. 343–398.

[28] J. Brage, "Foundations of a high-level synthesis system," Ph.D. dissertation, Techn. Univ. Denmark (DTU), Lyngby, Denmark, 1993.

[29] S. K. C. Gelatt and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[30] P. J. M. Van Laarhoven and E. H. L. Aarts, Simulated annealing: Theory and practice Kluwer, Norwell, MA, 1987.

[31] S. F. Nielsen, "Behavioral synthesis of asynchronous circuits," Ph.D. dissertation, Inform. Math. Modelling, Techn. Univ. Denmark (DTU), Lyngby, Denmark, 2005.

[32] J. Sparsø and S. E. Furber, *Principles of Asynchronous Circuit Design—A Systems Perspective*. Norwell, MA: Kluwer, 2001.

[33] A. M. G. Peeters, "Single-rail handshake circuits," Ph.D. dissertation, Dept. Math. Comput. Sci., Eindhoven Univ. Technol., Eindhoven, The Netherlands, 1996.

Sune Fallgaard Nielsen received the M.Sc. degree in physics and the Ph.D. degree in computer science from the Technical University of Denmark, Lyngby, Denmark, in 1999 and 2005, respectively.

He is currently a postdoctoral with the Department of Information Technology, Technical University of Denmark. His research interests include behavioral synthesis of asynchronous circuits, combinatorial optimization, modeling of nonlinear systems by the use of periodic-orbit theory, and extending the computational power of quantum computers.

Jens Sparsø (M'98) received the M.Sc. degree from the Technical University of Denmark (DTU), Lyngby, Denmark, in 1981.

Since 1982, he has been with the Section for Computer Science and Engineering, Department of Informatics and Mathematical Modelling, DTU, first, as an Assistant Professor and then was appointed as an Associate Professor in 1986, and Professor in 2007. His research interests include architecture and design of VLSI systems, application specific computing structures and processors, low power design techniques, design of asynchronous circuits and systems, and communication structures for systems-on-chip (i.e., networks-on-chip). He is the coauthor of the book *Principles of Asynchronous Circuit Design-A Systems Perspective* (Kluwer, 2001).

Prof. Sparsø was the recipient of the Radio-Parts Award and the Reinholdt W. Jorck Award in 1992 and 2003, in recognition of his research on integrated circuits and systems. He was the recipient of the Best Paper Award at the IEEE International Symposium on Asynchronous Circuits and Systems in 2005. He has been on the steering committees and technical program committees for several conferences. He was the General Chair for PATMOS 1998 and the Program Chair for PATMOS 1999 and ASYNC 2006. He was the Director and Local Organizer of a summer school on asynchronous circuit design at DTU in 1997.

Jan Madsen (S'83–M'90) received the M.Sc. degree in electrical engineering and the Ph.D. degree in computer science from the Technical University of Denmark, Lyngby, Denmark, in 1986 and 1992, respectively.

From 1992 to 1996, he was an Assistant Professor and from 1996 to 2002 he was an Associate Professor. Since 2002, he has been a Full Professor of computer-based systems with the Department of Informatics and Mathematical Modelling, DTU. He is currently head of the section on Embedded Systems Engineering. His research interests include modelling, analysis and design of embedded systems, particularly system-level tools for performance analysis and verification, hardware/software codesign, and wireless sensor networks. He has published more than 80 papers in peer-reviewed international journals and conferences, 7 book chapters, and edited 1 book.

Prof. Madsen has been the Program Chair and Vice Chair of the Design Automation, and Test in Europe Conference, and Program Chair and General Chair for the Hardware/Software Codesign Conference, CODES. He has been a member of the technical program committee and organizing committee of several technical conferences, including the Symposium on Hardware/Software Codesign, the International Symposium on System Synthesis, the Design Automation Conference, the Real-Time System Symposium, and the International Symposium on Industrial Embedded Systems.