2010 IEEE Symposium on Asynchronous Circuits and Systems

Click Elements
An Implementation Style for Data-Driven Compilation

Ad Peeters, Frank te Beest, Mark de Wit
Philips Incubators
High Tech Campus, 5656 AE Eindhoven, the Netherlands
{*ad.peeters,frank.te.beest,mark.de.wit*}*@philips.com*

Willem Mallon
NXP
High Tech Campus, 5656 AE Eindhoven, the Netherlands
w.mallon@nxp.com

***Abstract*—This paper presents a new design template and design flow for the implementation of data-driven asynchronous circuits. It relies on the use of edge-triggered flip-flops as the only storage elements, not only for the datapaths, but also for the control circuits; latches and C-elements that are common in many asynchronous circuit design styles are not required. The design template uses a two-phase handshake protocol for inter-component communication. In a pipeline structure, these circuits operate near the speed of Mousetrap circuits, but the required design-flow is simpler. The implementation style — which we refer to as Click elements — has been chosen to resemble synchronous circuits as much as possible. This allows for the use of conventional optimization and timing tools in the design flow and for a cheaper design-for-test implementation. The click templates are well suited for a data-flow driven compilation flow, which avoids much of the control overhead of traditional syntax-directed compilation. The two-phase circuits show a significant improvement in performance and energy efficiency compared to four-phase single-rail circuits.**

***Keywords*-asynchronous circuits, pipelines, two-phase circuits, control-data-flow circuits.**

I. Introduction

Syntax-directed compilation of asynchronous circuits has been established as a practical way of designing robust asynchronous circuits, with hundreds of millions of ICs in the market today in the smartcard, identification, and in-vehicle networking domains. Although this macro-modular approach has an inherent elegance, it certainly also has its limitations. Perhaps the strongest point of syntax-directed compilation is that it allows a modular approach in which handshake protocols at component interfaces help in localizing timing assumptions. Four-phase handshake protocols appear to lead to the simplest circuits, yet — in combination with the resulting distributed but interconnected handshake controllers — this tends to limit the maximum achievable performance. The performance limitation may not be an issue for the implementation of reactive state machines and simple microcontrollers, but it does impact the applicability in pipelined microprocessors and digital signal processing functions.

Part of this work has been supported by CATRENE under project CT302 TOETS.

Pipelined designs have become ever more important in the experiments and products we have been involved in. These designs turned out to hardly benefit from the expressive power of the design language, since they are based on the replication of a few simple templates.

Our initial approach to improve pipelined design was to limit the design to the use of these templates and to provide an optimized implementation thereof. This however leads to clumsy and bulky code that is difficult to maintain and in a way reduces the abstraction level of the source code. The optimization in case of a syntax-directed compilation not only requires careful tweaks of the source code, it may even include setting compiler directives on parts of the design.

To restore a high level of abstraction for pipeline designs, we moved from syntax-directed compilation to dataflow compilation. Dataflow techniques are well known in compiler design and are now rapidly gaining popularity in asynchronous circuit design [1], [2], [3], [4].

One of the bottlenecks in the performance optimization of asynchronous circuits is static timing analysis in the presence of combinational loops. In order to verify timing assumptions (such as delay matching and isochronic forks) in control circuits, static timing analysis needs to be performed at various stages during the optimization and sign-off. Combinational loops, which for instance exist inside C-elements and in more global asynchronous control loops, have to be broken first in order to enable the analysis to be done. This limits optimization, especially since different set of timing breaks are needed for different modes of operation of the circuit (such as initialization, functional, and scan modes).

A side-effect of the omnipresence of combinational loops in asynchronous circuits is that it not only complicates static timing analysis, it also turns out to make performance estimation by designers a challenge, even when supported by transparent compilation. Apparently, the plug and play simplicity of handshake circuits has an associated price in terms of complexity and performance.

In working with standard (third-party) EDA tools, especially for physical synthesis and static timing analysis, we have learned that using flip-flops rather than latches as state-holding elements greatly simplifies the design flow. In

978-0-7695-4032-0/10 $26.00 © 2010 IEEE
DOI 10.1109/ASYNC.2010.11

IEEE computer society

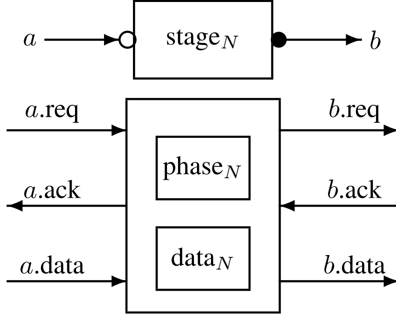

Figure 1. Handshake (top) and signal (bottom) representation of a pipeline stage, where variables have been indexed with stage number N.

addition, today's standard-cell libraries have many optimized implementation for flip-flops and only a few latches, if any. A third reason to prefer flip-flops over latches in datapaths is the demand for design-for-test, which is typically implemented using scan test. Scan-testing in the presence of latches requires the addition of slave latches that add an overhead that more-than-eliminates any advantage that latches may have from a functional perspective, in terms of area, energy-efficiency, and performance. Even optimizations like L_1L_2* testing [5] are not able to completely remove this overhead.

In this paper we present two contributions, the combination of which is especially interesting. The first is a new pipeline template named Click based on two-phase handshake protocols and a circuit implementation that only uses flip-flops to store state, both for control and data. When taking design-for-test into account, these circuits are more efficient than Mousetrap circuits [6]. The second contribution is a data-driven compiler and associated design flow that enables high-level design and allows the exploitation of software compiler technology. The combination of the two leads to circuits that are faster, are more straightforward for performance analysis and reduce the burden of the designer to optimize the source code, when compared to four-phase single-rail syntax-directed compilation.

II. Click elements

We use the name Click elements to refer to a data-driven two-phase implementation of a class of handshake circuits in which control information is forwarded with the data rather than via independent control paths. In this implementation, all input handshakes are passive and all output handshakes are active. The channels used to connect the elements are therefore of the push type (or nonput for control only channels). This makes these elements suitable for use in a data-driven compilation scheme. We will first introduce the implementation of a simple pipeline stage before presenting generalized Click elements.

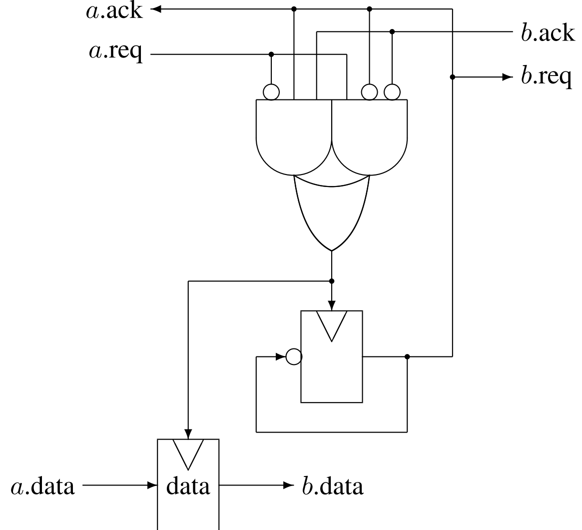

Figure 2. Click implementation of simple pipeline stage.

A. Simple pipeline stage

The simplest Click element is a pipeline stage with a single input handshake and a single output handshake, as shown in Fig. 1. Such a stage element will repeatedly accept new data from its passive input (denoted by an open circle) and present this data at its active output (denoted by a filled dot), acting as a buffer. The critical operation of the stage is the copying of new data, which is triggered when (i) new data is available at its input, and (ii) the current data-item has been accepted at its output, for instance by a next pipeline stage. This precondition for accepting new data can be defined precisely if we introduce a state-holding element that defines the phase of the data, assuming that this will change at every cycle of operation.

With reference to Fig. 1, the triggering moment and associated action can be defined as:

$\texttt{if}\ \text{phase}_{N-1} \neq \text{phase}_N\ \texttt{and}\ \text{phase}_N = \text{phase}_{N+1}$
$\texttt{then}\ \text{phase}_N,\text{data}_N := \text{phase}_{N-1},\text{data}_{N-1}$

Assuming a two-phase handshake protocol on channels a and b, this can be implemented as:

$\texttt{if}\ a.\text{req} \neq \text{phase}_N\ \texttt{and}\ \text{phase}_N = b.\text{ack}$
$\texttt{then}\ b.\text{req},\ a.\text{ack},\ \text{phase}_N,\ \text{data}_N :=$
$-b.\text{req},\ a.\text{req},\ \text{phase}_{N-1},\ \text{data}_{N-1}$

If we then choose a Boolean value for the phase, then we can select it such that it encodes a.ack, and we arrive at our Click implementation of such a pipeline stage:

$\texttt{if}\ a.\text{req} \neq a.\text{ack}\ \texttt{and}\ a.\text{ack} = b.\text{ack}$
$\texttt{then}\ b.\text{req},\ a.\text{ack},\ \text{data}_N := -b.\text{req},\ a.\text{req},\ \text{data}_{N-1}$

The trigger condition is equivalent to:

$a.\text{req} * -a.\text{ack} * -b.\text{ack} + -a.\text{req} * a.\text{ack} * b.\text{ack}$

Such a stage requires a single control flip-flop to store the control state (a.ack), as shown in Fig. 2. For the generation of the next state, this implementation locally inverts the

control signal. Naturally, making a copy of a.req is also an option, as specified above.

One of the reasons for calling such circuits Click elements is that the control elements have a precisely defined moment at which the state is updated, much like in a synchronous circuit. In contrast, in most asynchronous circuits the control and datapath states more or less evolve autonomously, albeit under control of handshake signals.

An important requirement of this circuit is the generation of a well defined clock pulse for the registers. In the inactive state, the outputs of both AND gates are zero. The circuit then waits for changes on the incoming handshake signals. When this happens, the output of one of the AND gates goes high and this produces the rising edge of the clock. As a response to the clock, the registers update their state. For the control register this inverts the state bit. Now the active AND gate goes back to zero, the inactive AND gate stays zero. The result is the falling edge on the clock signal. Two constraints need to be satisfied for the correct operation of this circuit.

1) The clock pulse width must be larger than the minimum pulse width defined in the technology library. This can be checked by a static timing analysis tool.
2) During the active phase of the clock, the input handshakes have to remain stable. The handshake signals can only change after a transition on the corresponding output handshake signal.

Obviously, glitches on the handshake signals are not allowed. Since all handshake signals are driven by flip-flops, these are safe.

The correct operation of the circuit is not dependent on the potential differences in arrival time of the branches of the forks driving the AND gates. The AND gates need three valid inputs to fire. Only one of the AND gates is 'selected' by the internal state bit. Transitions on the other AND gate are masked. Once activated, the state bit changes. This affects both AND gates, however, since during this time the handshake signals don't change, the only possible result is that both AND gates go back to the zero state.

The implementation of Fig. 2 has also been described in [7]. We will later introduce a number of generalizations of this circuit (with multiple inputs, outputs and state variables) that are not covered by this publication. A dual-rail version of this circuit has also been used in asynchronous interconnect design [8], [9]. This version also uses flip-flops as state-holding elements but requires more XOR gates in the control logic.

Two-phase control circuits have been introduced before. In Fig. 3 our circuit is compared to two of these existing approaches. The simplest control circuit is based on a so-called Muller pipeline [10], in which a Muller C-element and inverter together form the control circuit. Sutherland's Micropipelines [11] are based on this control circuit. Fig. 3(a) shows an implementation where the feedback-loop inside

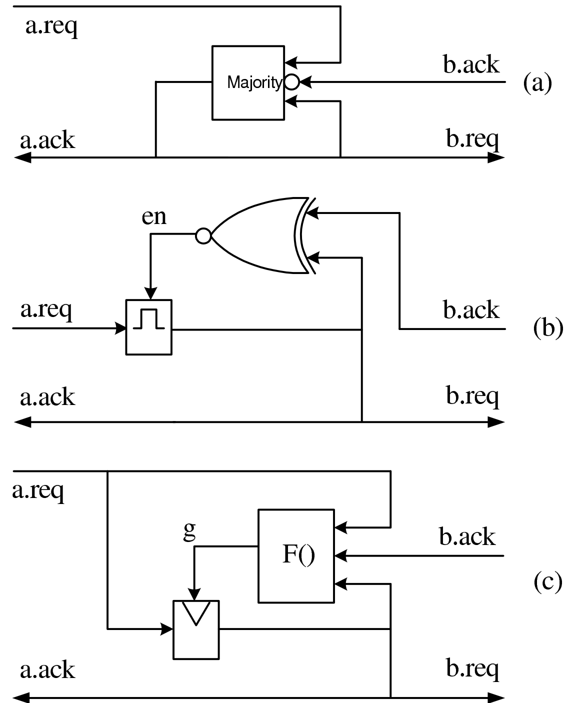

Figure 3. Three two-phase pipeline implementations, with feedback-loop based on C-element (a), latch (b) and flip-flop (c).

the C-element is made explicit by showing a possible implementation of the C-element based on a majority-gate. This circuit does not contain a four-phase signal that can be easily used to control datapath latches or flip-flops. It can be combined however with double-edge-triggered flip-flops or toggle-latches, both at the price of extra multiplexer delay in the datapath.

A more attractive two-phase pipeline control circuit is Mousetrap [6], shown in Fig. 3(b), where the control feedback-loop includes a four-phase signal (labeled 'en' in the picture) that can be used to enable both the control latch and the datapath latches (not shown in the diagram). In contrast to the Muller control, the feedback loop in the Mousetrap circuit is broken by a state-holding element, namely the latch. Since this is a level-sensitive element, this still leads to complications for timing analysis and design-for-test.

Fig. 3(c) shows the Click implementation introduced above, which also includes a four-phase signal (labeled 'g' in the picture), that is used to clock both the control and datapath flip-flops. One may observe that in the Click circuit, the control feedback loop is broken by an edge-triggered element, which facilitates both timing analysis and design-for-test. Compared to the Mousetrap circuit, one may note that the combinational function in the control feedback loop is slightly more complex (it has three instead of two inputs), which is likely to lead to a somewhat reduced performance

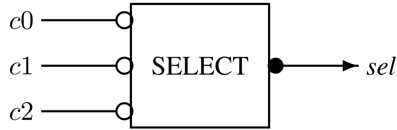

Figure 4. Handshake symbol SELECT with three inputs.

before design-for-test is introduced.

The Click implementation presented in Fig. 3(c) can be easily generalized to control circuits with multiple inputs, multiple outputs, and multiple control flip-flops, simply by changing the feedback function F(). The Click template can thus be used to implement handshake elements with at least one passive input and at least one active output handshake port.

In general, a Click implementation consists of:

1) A control circuit, consisting of combinational logic and control flip-flops in which the combinational logic generates the clock signal for the flip-flops. A general version of the control circuit is shown in Fig. 3(c).
2) A datapath circuit (optional), consisting of combinational logic and datapath flip-flops (registers), for which the clock signal is generated by (and shared with) the control circuit.

The combinational function in the control circuit precisely defines the trigger moment for the datapath. This trigger point leads to a rising clock edge on the local clock, which updates both the control and datapath flip-flops, and as a side affect also generates the (redundant) falling edge of that same clock. In the following sections some examples of generalized Click elements are presented.

B. Example: SELECT

The SELECT is an example of a handshake element that can be implemented with the generic Click template. It has an arbitrary number of passive inputs, for which the environment will guarantee mutual-exclusive activation (if not, then this can be arranged by prefixing the element with an arbiter component to implement this mutual exclusion), and one output port. The SELECT component waits for a handshake on one of its inputs to be activated, and will subsequently send on its output a one-hot encoded data item indicating which port activated the select. Once this information is acknowledged, the input will also be acknowledged. An example of this component for three inputs ($c0$, $c1$, and $c2$) is shown in Fig. 4.

Fig. 5 shows the implementation of the SELECT with three channels. This circuit contains three flip-flops that maintain control status, namely which handshake port (0, 1, or 2) initiated the handshake. This state is needed to generate the acknowledge signal and to generate the one-hot encoded data for the *sel* output port.

One may observe that in this circuit, the next state of the control flip-flops is not always the inverse of the previous state. Only for the then-active control channel, the clock-event generated by sel.ack will change the state of the flip-flop; the other two state bits will remain unchanged. Therefore the next-state of the control flip-flops cannot be generated using the inverter feedback, as was the case for the simple Click element presented in Fig. 2.

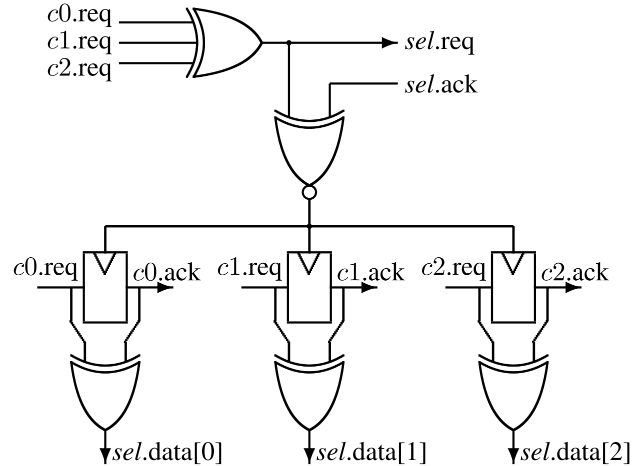

Figure 5. Implementation of Select element.

C. Example: JOIN stage

The implementation of a Join stage is presented to illustrate that the Click template indeed allows the elimination of C-elements in pipelines, such in contrast to the Mousetrap template.

Figure 6. Handshake symbol for JOIN element.

A Join component with two inputs is shown in Fig. 6. It will repeatedly wait until new data has arrived on inputs a and b, and then forward the combination of the two on output c, and store it internally.

The implementation of this Join component would traditionally require a C-element to synchronize the incoming requests on a and b, before activating a standard pipeline state that would capture the data and forward it to c. An example of such an implementation can be found in the literature on Mousetrap [6].

In the Click implementation of a Join element, we can simply make the combinational feedback function more complex, so as to precisely define the capture moment in terms of the inputs. This implementation is shown in Fig. 7, in which function F() stands for:

$$(a.\text{req} \neq c.\text{ack}) \texttt{ and } (b.\text{req} \neq c.\text{ack}) \texttt{ and } (c.\text{ack} = c.\text{req})$$

which is equivalent to:

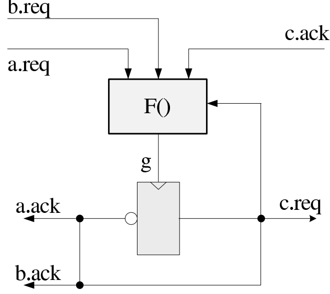

Figure 7. Click implementation of control circuit for JOIN element.

a.req*b.req*$-c$.req*$-c$.ack $+$
$-a$.req*$-b$.req*c.req*c.ack

D. Example: Arbitrated Merge

No asynchronous circuit solution is perfect, and the Arbitrated Merge component is an example of this. The Click template does eliminate the need for C-elements in control circuits, but it fails to eliminate the use of the elusive mutual-exclusion element.

The Arbitrated Merge is a handshake element suited for Click implementation. Its handshake symbol is shown in Fig. 8. The element shown has two input channels (a and b), and an output channel c. Data that arrives at the element via handshake on a or b is propagated to channel c. The data on c will be interleaved; so when there is a conflict (data arriving on a and b simultaneously), the Merge will make a choice on which channel to serve first.

Figure 8. Handshake symbol for MERGE element.

A Click implementation of the Merge is shown in Fig. 9. The element that generates signals a.sel and b.sel is a so-called mutual-exclusion element, which guarantees that only one output will go high, even if both inputs are high. When either a.sel or b.sel goes high, the clock signal for the Click pipeline is generated, which latches both the three control signals (a.ack, b.ack, and c.req), and the datapath bits (c.i, generated from either a.i or b.i, depending on which input is selected by the mutual-exclusion element).

One may observe that control signal c.req will be inverted unconditionally, upon each activation. In contrast, control signals a.ack and b.ack are inverted conditionally, depending on which channel is selected by the mutual-exclusion element.

E. Example: DEMUX

The examples shown so far all generate an internal clock signal that is directly distributed to all control and datapath

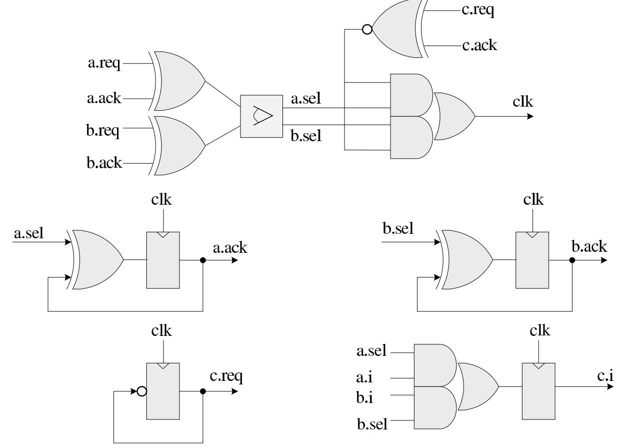

Figure 9. Click implementation of arbitrated MERGE handshake component.

flip-flops. Selective updating of control flip-flops is then implemented in the combinational logic, by augmenting the feedback-function for the control element, such as in the implementation of the Merge element shown in Fig. 9.

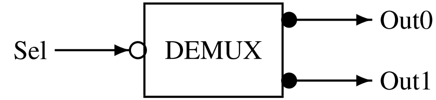

Figure 10. Handshake symbol Demux with two outputs.

The Demux component is an example of a component that can also be implemented using clock gating in the control. The Demux handshake element, whose symbol is shown in Fig. 10, is a data-driven demultiplexer that repeatedly receives on its input data-items that are propagated to one of its outputs, where the destination is encoded in the input.

Fig. 11 shows a Click implementation of the Demux, in which the destination of incoming data is one-hot encoded as part of the incoming data on Sel. The destination can then be used as a clock-gating condition to selectively update either the request on either channel Out0 or channel Out1, depending on which output is selected.

One may observe that only one of Out0.req or Out1.req will toggle per activation. Therefore, signal Sel.ack can also be generated with an XOR function of these output request signals, thereby avoiding the use of a flip-flop. For a generic Demux element with more than two channels this is more costly than using the extra register, since the complexity of the XOR function would increase, whereas the flip-flop implementation shown here suits all.

It is also possible to implement this element in an unbuffered way, where the data flip-flops are omitted. In such an implementation, the Sel.ack signal would have to be delayed until the output that was addressed has acknowledged completion. This would require an XOR function on the

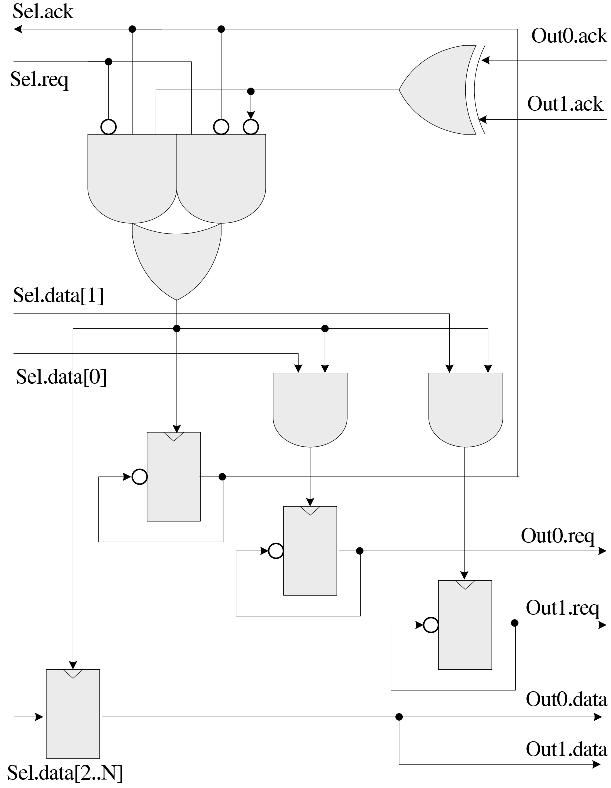

Figure 11. Handshake element Demux with two outputs.

incoming acknowledge signals of the outputs.

III. Design flow

A complete high-level synthesis flow to create Click networks has been largely created from scratch. The only part that has remained the same is the front-end of the compiler. This has been done to ensure source code compatibility with the syntax-directed flow. The source code language is Haste, although it should be straightforward to replace this by a SystemVerilog or SystemC front-end, or even add support for design languages like Simulink [12].

An overview of the design flow is given in Fig. 12. It consists of three steps before handing the design over to a standard third-party backend flow for placement, routing, and sign-off. The three steps are:

- Compilation (tool name: htcomp)
- Mapping (tool name: htmap)
- Processing for Optimization, Scan & Timing (POST) (tool name: htpost)

The interfaces between the handshake tools are based on standard file formats, but with proprietary extensions used for functionality that is not available in the basic format. For the handover to third-party tools, most design information is passed on using standard file formats. For the additional requirements, custom constraints are used together with tool-specific scripting.

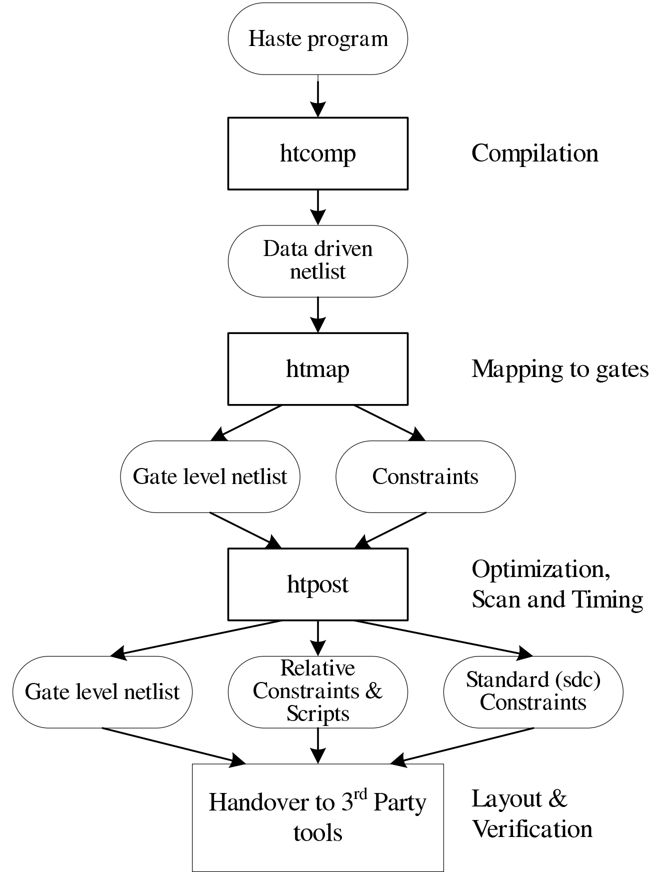

Figure 12. Handshake element Demux with two outputs.

A. Compilation

The compiler converts a Haste program into a data-driven handshake circuit. One of the biggest differences is that its compilation is not syntax-directed like the old compiler, which means that there is no longer a one-to-one relation between Haste constructs and handshake components. With syntax-directed compilation the designer decided where and when sequencers and storage elements are introduced by using semicolons and variables. In the new data-driven compilation scheme these decisions are made by the compiler (which can be guided by constraints).

This approach not only enables more compiler optimizations but it also keeps the Haste code clean from implementation details. For example, Fig. 13 and Fig. 14 show the Haste code for the two compilers of a process that performs conditional inputs without using registers to store the input value. The syntax-directed version requires probes because using a variable (as x and y in Fig. 14) would introduce storage in the circuit. This is costly both in terms of area and time to read and write the extra variable. Probes are low-level constructs that are intrinsically unstable and require compiler directives to suppress warnings about potentially unsafe code. The probe construct and compiler directive

```
mux:proc(
  S?chan bool & IN1,IN2?chan T & OUT!chan T).
  forever do
    wait(outprobe(S))
  ; if {$STABLE} dataprobe(S) {$} then
      wait(outprobe(IN1))
    ; OUT!{$STABLE} dataprobe(IN1) {$}
    ; IN1?~
    else
      wait(outprobe(IN2))
    ; OUT!{$STABLE} dataprobe(IN2) {$}
    ; IN2?~
    fi
  ; S?~
  od
```

Figure 13. Syntax-directed Haste code for multiplexer without storage

```
{$NR_BUF '0'} mux:proc(
  S?chan bool & IN1,IN2?chan T & OUT!chan T).
begin
  x:var bool & y:var T
| forever do
    S?x
  ; if x then IN1?y else IN2?y fi
  ; OUT!y
  od
end {$}
```

Figure 14. Data-driven Haste code for multiplexer without storage

make the code harder to write and understand. This version also has explicit sequencing of actions, further reducing its speed.

The data-driven version nicely separates the functional description of the multiplexer and the number of storage elements used for its implementation (which is set with a compiler directive). Using data-flow analysis, the compiler can determine that no storage is required. The current default in the compiler is to insert a single pipeline stage (buffer) in each procedure. This can be overruled by specifying a compiler directive for the procedure.

The new data-driven compiler is source code compatible with the previous one. However, non-handshaked expressions, like probes and wires, do not fit well in the data-driven compilation scheme. The values of these expressions are always observable, which makes it more difficult to create a pipelined implementation. When compiling these expressions the compiler defaults to syntax-directed compilation, which usually results in complex controllers with multiple sequential steps (instead of the single-sequencer controllers generated by the data-driven approach). Haste designs that have been optimized for syntax-directed compilation typically use many non-handshaked expressions, which reduces the potential for optimization by the compiler. Therefore a rewrite or cleanup of the Haste source code is typically needed to truly benefit from the data-driven compilation scheme.

The compilation flow from Haste to a handshake circuit is done in three steps, two of which are shown in Fig. 16 for the Haste program in Fig. 15. First Haste is translated to a control-data flow graph (CDFG), which is a Petri net that contains only the essential dependencies between atomic actions and computations in the Haste code. The CDFG represents the complete interface between the Haste-specific front-end of the compiler and the data-driven back-end. It is based on the format defined in [13], which is also used in [14]. The nodes in the CDFG represent the operations in the circuits, while the edges represent the data transfers between the nodes. Each data transfer can be seen as a movement of tokens, where a node with a valid set of tokens on its inputs consumes these and produces a set of tokens on its outputs (typically one). Edges can also have an initial token directly after reset.

```
forever do
  IN?x
; if x.valid then
    result := f(x.data)
  else
    result := 0
  fi
; OUT!result
od
```

Figure 15. Simple Haste program

Every CDFG node results in a unique handshake component instantiation (i.e., there is no hardware sharing between node implementations). In the initial CDFG every edge represents the value transfer of a single Haste variable. This will result in an implementation with fine-grained parallelism. However, more parallelism does not always give higher performance if extra synchronization is required. Therefore, before proceeding to the next step, the CDFG is transformed by merging edges and nodes to reduce the number of incoming and outgoing edges of each node (which determines the amount of synchronization).

In the second step (not shown in Fig. 16) the compiler decides where to insert buffer (pipeline) components. A buffer is a storage component that alternates between receiving a value on its passive input and sending that value via its active output. Buffers can be inserted on any edge in the CDFG. The cost of a buffer (i.e., the number of registers) depends on the bit-width of the edge on which it is placed. During buffer insertion the compiler tries to minimize the number of registers while satisfying the following three constraints:

1) The architecture defines that every edge with an initialization token must be implemented with a buffer that starts with sending the initialization value.
2) Every cycle in the CDFG must contain at least two buffers (otherwise deadlock would occur because a single buffer alternates between inputs and outputs).

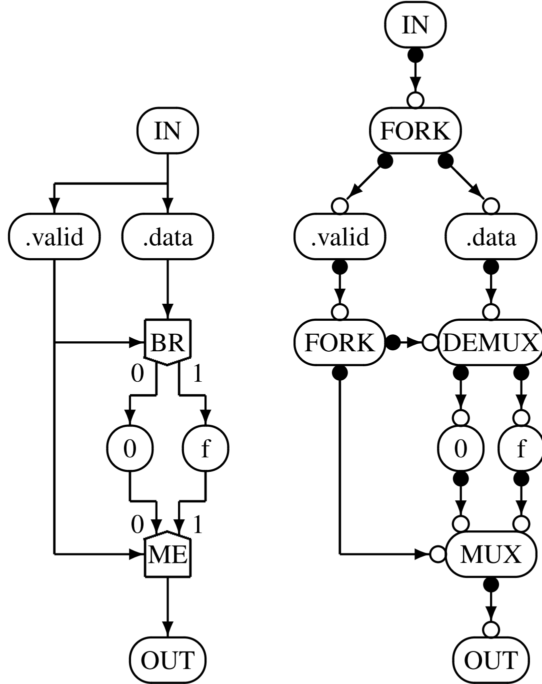

Figure 16. CDFG and handshake circuit for Haste program of Fig. 15.

3) There must be at least N buffers on every path from any input to any output of the CDFG (where N specifies the number of pipeline stages). By default N is one, but this can be overridden for every Haste procedure using a compiler directive (e.g., as shown in Fig. 14).

This optimization problem is solved by a heuristic algorithm, which falls outside the scope of this paper.

The third and last step converts the CDFG to a handshake circuit. This step is relatively straightforward because the operation of a Petri net closely matches the behavior of a data-driven handshake circuit. Every type of node in the CDFG can be directly mapped onto a corresponding handshake component with passive inputs and active outputs.

The generated handshake circuit is written into a new SystemVerilog-based format. The main advantages of this new format are:

- Combinational logic in the data-path is translated into RTL Verilog, which can be synthesized with standard EDA tools. This enables the flow to take advantage of the latest developments in logic synthesis. It also makes it easier to port the flow to different standard-cell libraries. Finally it allows for RT-level simulations (which are faster than gate-level simulations).
- The hierarchy of the Haste design (i.e. procedure call graph) is preserved in the handshake circuit. This facilitates analysis and constraint generation during later stages in the design flow.
- It allows annotations for each channel and component, which enables mixing of different implementation styles in a single handshake circuit. This can for example be used to specify two-phase or four-phase operation.

B. Mapping

The mapper takes the data-driven handshake representation of a design and combines this with a library of component implementations to create a gate-level netlist and a set of timing constraints. Initial mapping is a straightforward substitution of handshake components by their respective gate-level implementations. The RTL parts of the design are given to a third-party logic synthesizer and the resulting logic is included in the final netlist.

An example of a gate level handshake component description is shown in Fig. 17. This example shows the Click implementation of the buffer (pipeline) component that was introduced in Fig. 2. The component description is again in SystemVerilog [15] with a custom extension for the behavioral and constraint information. Nets that are referenced inside a port instantiation are postfixed with `.in` or `.out` (so-called 'modports') to denote if the port is consuming or driving the net, respectively. When parameterized components are used, htmap will first create a set of component instantiations with resolved parameters.

Since the generated circuit uses conventional logic (in the asynchronous world also known as single rail), the timing relation between the control circuit and the data path logic is not automatically correct. For correct operation, the control circuit must be slower than the data path. The old synthesis method achieved this by inserting delay chains at predefined places in the control circuit. This can however lead to an excessive number of delay chains that are not always required.

The new synthesis method does not include delay chains upfront but inserts them later during timing verification at those places where they are required. To guide this process relative-timing constraints [16] are generated by htmap, based on the constraint information in the handshake component description. Relative-timing constraints are in general not limited to a single handshake component. In most cases multiple components need to be analyzed together to create complete constraints. Two types of constraints are used:

- Constraints that start and end in the same component.
- Constraints that start in one component and end in another component.

An example of the second kind is included in Fig. 17. The three dots (`...`) are used to indicate paths that are external to the handshake component. Both setup_start and setup_end are incomplete constraints that are analyzed by htmap. If htmap finds a timing path starting with the fast part of setup_start in one component and ending with the fast part of setup_end in another component it combines the two into a single complete constraint. While analyzing timing

```
module HC_BUF(input reset,
              push_chan.pas #(.width(2), .prot("2phase")) in,
              push_chan.act #(.width(2), .prot("2phase")) out );

    assign in.ack=toggle;
    assign out.req=toggle;

    INV     inv(.a(in.req.in), .z(ai.out));
    NAND3   na3(.a(ai.in), .b(out.ack.in), .c(toggle.in), .z(wi.out));
    OAI31   oai(.a(ai.in), .b(out.ack.in), .c(toggle.in), .d(wi.in), .z(clk.out));
    FFRESET ff(.d(wi_ff.in), .q(toggle.out), .clk(clk.in), .r(reset.in));
    INV     inv_ff(.a(toggle.in), .z(wi_ff.out));
    FF      var_0(.d(in.data.in[0]), .q(out.data.out[0]), .clk(clk.in));
    FF      var_1(.d(in.data.in[1]), .q(out.data.out[1]), .clk(clk.in));

  timing
    behavior stg( [ ( in.req~ || out.ack~ ) ; ( out.req~ || in.ack~ || out.data~ ) ] );

    relative setup_start(
       .fast(oai.z+ ; ( var_0.q~ || var_1.q~ ) ; out.data~ ; ...),
       .slow(oai.z+ ; out.req~ ; ...) );

    relative setup_end(
       .fast(... ; in.data~ ; ( var_0.d~ || var_1.d~ )),
       .slow(... ; in.req~ ; ( var_0.clk+ || var_1.clk+ )) );

  endtiming
endmodule
```

Figure 17. Specification of the Click element of Fig. 2 based on SystemVerilog.

paths, htmap uses the behavioral description, also shown in Fig. 17, to determine how to trace transitions through the components (e.g., to eliminate false paths).

C. Post processing

During post processing the last modifications are done to the netlist, before the handover to the layout process. These modifications consist of a final logic optimization, the insertion of scan test (optional) and the tuning of the circuit timing to make sure that all timing constraints are met.

To limit the number of tools and increase flexibility, all these tasks are combined in a single tool, htpost, which is controlled by a tcl script. This tool will read standard liberty (.lib) library files, verilog gate-level netlists and the constraints generated by htmap. Additional constraints, such as for performance, net loads, and transitions times, can be specified using the standard sdc syntax.

Since htmap does not fix design rule violations, an optimizer is included in htpost that does constant propagation and buffer insertion to create a violation free circuit. This is only intended as a preliminary optimization to enable error-free simulation of the output netlist. Final optimization takes place during layout.

The optional scan test is based on the synchronous multiplexer-based testing method introduced in [17]. One major improvement is the use of clock gating to introduce the synchronous clock into the circuit. This clock-gating logic can in most cases be integrated with the pipeline control logic to reduce the cost of scan, as shown in Fig. 18 for a basic Click stage. The clock gating is of the so-called OR-type, which means that after the rising edge of the clock, the circuit has time to stabilize until the falling edge. The scan-enable input to the AND gates is used to ensure that the clock is always enabled during scan shift.

The circuit also allows for some advanced behavior to switch the circuit between synchronous mode and asynchronous mode. For example, it is possible to use the synchronous scan mode to bring the circuit into a certain state by scanning the appropriate bits into the control state registers; from which it is possible to continue operation in asynchronous mode by first making the clock signal zero followed by making the scan enable zero. This behavior can form the basis of future advanced delay-fault test methods to directly validate the relative timing constraints. Circuits that can switch between synchronous and asynchronous operation have been reported earlier by Grass [18] in a variant that uses an explicit synchronous/asynchronous mode signal.

The main task of htpost is to make sure that the circuit meets all relative timing constraints. To solve timing violations, htpost will insert delay chains in the circuit. To minimize the number and size of the delay chains all constraints are analyzed together, to find the optimal

Table I
BENCHMARK RESULTS FOR KEY HANDSHAKE COMPONENTS.

	Syntax-directed, four-phase			Data-driven, two-phase		
Benchmark	Throughput (MHz)	Area (μm^2)	Power	Throughput (MHz)	Area (μm^2)	Power
Fifo	193	549	52.29	603	487	12.27
Feedback	145	1507	193.93	522	1225	34.17
Demux	164	606	76.93	520	664	21.10
Non buf demux	224	164	32.44	470	221	14.61
Mux	100	1143	156.13	424	1020	54.61
Non buf Mux	84	1020	334.58	400	573	36.36
Select	90	1470	311.59	138	1634	234.38
Static for	121	1331	297.03	245	2499	177.39
Dynamix for	94	2429	533.73	168	3981	615.81
Clock gate	173	717	80.02	611	602	13.63
Ram	111	1495	1625.04	259	1712	1424.22

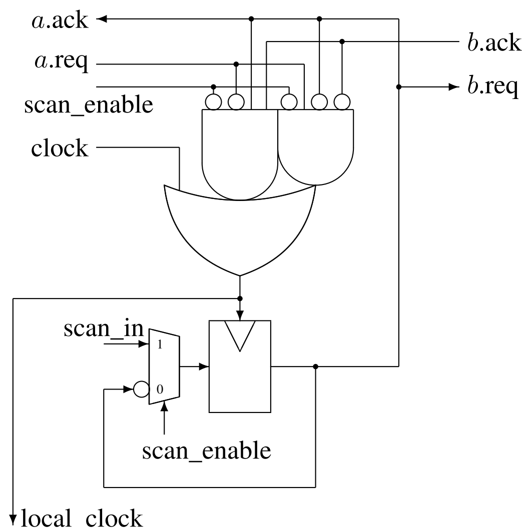

Figure 18. Scan-testable control circuit of Click pipeline stage of Fig. 2.

placement of the delay chains. This analysis consists of the following two steps:

1) For each gate affected by a constraint path the slack is calculated using standard static timing analysis. If the slack is negative, the value is annotated on the path.
2) The slacks are combined to determine the optimal size and location of the delay chains required to solve all violations.
   - For each constraint the "minimal flow" is calculated, this is the minimal number of delay elements that are required to solve the constraint.
   - Considering the delay elements required for one constraint, the effect of these elements on the total negative slack for all constraints is determined. The set of delay elements with the best total negative slack reduction is inserted in the circuit and the analysis is started again.

The method of introducing delay chains only where needed and where most effective, based on global analysis, is a significant improvement over the method implemented in the syntax-directed compiler, which introduces delay chains in predefined locations based on local timing requirements only.

IV. RESULTS

During the development of the data-driven design flow, a number of small designs were used to validate the components and flow, and to get an initial idea of the performance and power characteristics. For comparison, the designs were also processed using the syntax-directed compiler and associated design flow. The Haste source code that is used as input for the syntax-directed compiler has been optimized to get the best result, especially in terms of data-path registers and sharing of control, whereas the code provided to the data-driven compiler is more behavioral, see for instance the code in Fig. 13 and Fig. 14. Since the compilers are source-code compatible both versions of the design could be processed by both compilers. However, code optimized for the syntax-directed compiler will typically result in suboptimal data-driven implementation since it disables analysis and optimization by the compiler by the use of non-handshaked expressions, probes and compiler directives. Vice versa, input for the data-driven compiler is clean, which enables optimization by the data-driven compiler, but produces suboptimal (typically larger and slower) results in the syntax-directed flow, which does not include these data-flow analysis and optimization steps.

The results of the benchmarks are shown in Table I. It shows the pre-layout area, power, and throughput results in a 0.18 μm technology. The throughput numbers are also shown in Fig. 19. The selection of designs has been chosen to include all basic data-driven components that are included in the flow, and 8-bit datapaths are used in all designs. The benchmark also includes a design (Select) that is well suited for a syntax-directed flow but not very well suited for data-driven compilation. As a result this example hardly benefits from data-driven compilation. Most of the other designs show significant improvements, up to a 3x improvement of the throughput in a basic fifo stage. The power efficiency is improved even more than the throughput, indicating that

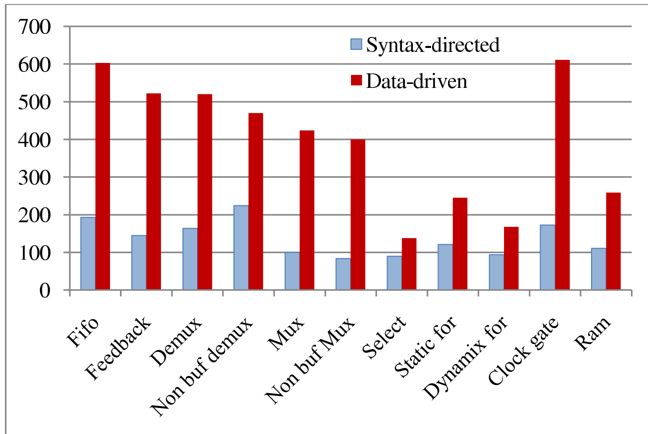

Figure 19. Throughput (in MHz.) for syntax-directed and data-driven implementations.

even per MHz the new circuit is more efficient.

The previous experiments were all done in a rather old 0.18 μm technology. To show future potential, we also did an experiment to indicate the maximum performance for a range of more advanced technologies. The circuit used for this was the Fifo benchmark. The results are shown in Table II. The results show that a throughput of around 1 GHz is possible using 90 nm process technology.

Table II
BENCHMARK RESULTS FOR KEY HANDSHAKE COMPONENTS.

Three stage fifo	Throughput (MHz)		
Technology Node	Worst Case	Nominal Case	Best Case
0.18 μm^2	603	998	1650
90 nm (High Vt)	656	1121	1767
90 nm (Standard Vt)	1085	1692	2496
65 nm (High Vt)	663	1210	1938
65 nm (Standard Vt)	1121	1849	3024

After the initial testing, work started on a larger benchmark. This was complicated however by the fact that most large benchmarks were heavily optimized for syntax-directed compilation. Without some rework to remove these optimizations, the data-driven compilation would be limited in its operation and not give the best possible results. The first design that was used was a custom-specific 16-bit microcontroller. After a rewrite that removed all arb/narb specifiers, STABLE directives and wire statements, the data-driven version of this circuit is twice as fast as the optimized syntax-directed version. Furthermore, the new Haste code is much cleaner and easier to read without the low-level constructs. In addition, the area of the data-driven design is equivalent to the optimized four-phase syntax-directed design. The prototype data-driven flow performs significantly better than the mature syntax-directed flow. This shows that the new approach has a great future potential.

V. CONCLUSION

This paper has introduced two contributions that can help in making asynchronous circuits more acceptable by making both the design flow and the designs themselves less disruptive, that is, more synchronous alike.

On the design-flow side, this has resulted in the implementation of a data-driven compiler for Haste, which has been set up in such a way that a connection to more standard design languages can be established via the internal control-data flow representation. The data-driven style of our handshake circuits is very similar (i.e. passive inputs, active outputs) and is based on the same insights as in the work of Taylor for Balsa [1], [2], [3]. Taylor has chosen for a data-driven subset of Balsa, which is then compiled in a syntax-directed way into a data-flow circuit. In contrast, we have chosen to stick with a more traditional imperative language, and have implemented a data-flow compilation. Thereby we have avoided restrictions on the input language, which would otherwise be needed because not all language constructs can be transparently mapped onto a pipeline template. In the design flow that we have presented here, any Haste program is compiled into a data-driven implementation through an intermediate control-data flow representation [13]. Advantages of our approach are that the input language is more intuitive and that compatibility between syntax-directed and data-driven designs can be maintained. In addition, data-driven designs described in a different source language, such as Simulink [12], can be compiled into Haste for data-driven compilation straightforwardly.

Nielsen [14], [19], [20] and Hansen [21], [4] have presented alternative approaches to improve on strict syntax-driven compilation. In both approaches, Haste or Balsa code is taken as a behavioral starting point, and different analysis and transformation steps are implemented, after which optimized Haste or Balsa is generated. The transformations described by Nielsen typically aim at improving area (reducing cost by sharing resources), although performance enhancing optimizations are also reported. The framework supports design-space exploration guided by area or speed constraints, and uses a specific template to map the optimized source code. The source-to-source transformations described by Hansen are aimed at enhancing performance by increasing parallelism through automatic parallelization and pipelining, arithmetic transformations and reordering of channel communications. Both approaches are very effective in relieving the burden for the designer. Since the standard compilers for Balsa and Haste are syntax driven, they demand optimized source code as input, which especially reduces readability and maintainability. The work of Nielsen and Hansen simplifies the design phase, but since the result is fed through the same compiler, the performance is still limited by the syntax-tree-like handshake control structure and the associated handshake components. In contrast, both

Taylor's work and this paper make use of a new class of handshake components and a new compilation scheme (based on data flow rather than control flow) to enable substantially faster handshake circuits.

On the circuit-implementation side, Click implementation has been introduced, which is based on two-phase handshake protocols (in line with Mousetrap [6]), but in contrast uses only flip-flops as state-holding elements, both in the control and in the datapath elements. This not only eliminates the need for C-elements, it thereby also facilitates the reuse of existing synchronous design tools. Click implementation can be used independent of a data-driven compiler. The arbitrated Merge element, for instance, is typically not used in a data-driven compiler.

Although data-driven compilation does not dictate the use of Click implementation of the associated handshake components, we believe this is the combination that leads to the highest performing circuits in an industrial standard-cell based design flow. The two-phase control circuits are efficient in terms of area and power, and most importantly, have proven to outperform the four-phase circuits in terms of speed. On the one hand this is due to the simplicity of the protocol and its flip-flop based implementation. On the other hand this is also facilitated by the fact that the Click implementation lends itself very well for performance optimization and physical synthesis using standard synchronous design tools.

Acknowledgements

Thanks are due to Rene Engbers and Rob Knubben (of TASS), and Frits Schalij (Philips Incubators) for their work on the implementation of the data-flow compiler, and to Marc Verra (Philips Incubators) for running the benchmarks.

References

[1] S. Taylor, "Data-driven handshake circuit synthesis," Ph.D. dissertation, University of Manchester, 2007.

[2] S. Taylor, D. Edwards, and L. Plana, "Automatic compilation of data-driven circuits," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, Apr. 2008, pp. 3–14.

[3] S. Taylor, D. Edwards, L. A. Plana, and L. A. Tarazona D., "Asynchronous data-driven circuit synthesis," *IEEE Trans. VLSI Syst.*, 2010.

[4] J. Hansen and M. Singh, "Concurrency-enhancing transformations for asynchronous behavioral specifications: A data-driven approach," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, Apr. 2008, pp. 15–25.

[5] S. DasGupta, P. Goel, R. G. Walther, and T. W. Williams, "A variation of LSSD and its implicationson design and test pattern generation in VLSI," in *Proc. Int. Test Conference (ITC)*, 1982, pp. 63–66.

[6] M. Singh and S. M. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Trans. VLSI Syst.*, vol. 15, no. 6, pp. 684–698, 2007.

[7] R. L. Traylor, "Self-timed data pipeline apparatus using asynchronous stages having toggle flip-flops," U.S. Patent 5,386,585, Jan. 31, 1995.

[8] B. R. Quinton, M. R. Greenstreet, and S. J. E. Wilton, "Asynchronous IC interconnect network design and implementation using a standard ASIC flow," in *Proc. Int. Conf. Computer Design (ICCD)*, Oct. 2005, pp. 267–274.

[9] B. R. Quinton, M. R. Greenstreet, and S. J. E. Wilton, "Practical asynchronous interconnect network design," *IEEE Trans. VLSI Syst.*, vol. 16, no. 5, pp. 579–588, May 2008.

[10] J. Sparsø and S. Furber, Eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[11] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989.

[12] M. Tranchero, L. Reyneri, A. Bink, and M. de Wit, "An automatic approach to generate Haste code from Simulink specifications," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, May 2009, pp. 175–184.

[13] J. T. J. van Eindhoven and L. Stok, "A data flow graph exchange standard," in *Proc. European Conf. on Design Automation (EDAC)*, Mar. 1992, pp. 193–199.

[14] S. F. Nielsen, "Behavioral synthesis of asynchronous circuits," Ph.D. dissertation, Technical University of Denmark, 2005.

[15] *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*, IEEE Computer Society Std. 1800-2005, Nov. 2005.

[16] K. S. Stevens, R. Ginosar, and S. Rotem, "Relative timing," *IEEE Trans. VLSI Syst.*, vol. 11, no. 1, pp. 129–140, Feb. 2003.

[17] F. te Beest and A. Peeters, "A multiplexer based test method for self-timed circuits," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, 2005, pp. 166–175.

[18] E. Grass, B. Sarker, and K. Maharatna, "A dual-mode synchronous/asynchronous CORDIC processor," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, Apr. 2002, pp. 76–83.

[19] S. F. Nielsen, J. Sparsø, and J. Madsen, "Behavioral synthesis of asynchronous circuits using syntax directed translation as backend," *IEEE Trans. VLSI Syst.*, vol. 17, no. 2, pp. 268–281, Feb. 2009.

[20] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the Haste/TiDE design flow," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, May 2009, pp. 185–194.

[21] J. B. Hansen, "Concurrency-enhancing transformations for asynchronous behavioral specifications," Master's thesis, University of North Carolina at Chapel Hill, 2007.