Asynchronous Data-Driven Circuit Synthesis

Sam Taylor, Doug A. Edwards, Luis A. Plana, *Senior Member, IEEE*, and Luis A. Tarazona D., *Student Member, IEEE*

***Abstract*—A method is described for synthesizing asynchronous circuits based on the Handshake Circuit paradigm but employing a data-driven, rather than a control-driven, style. This approach attempts to combine the performance advantages of data-driven asynchronous design styles with the handshake circuit style of construction used in existing syntax-directed synthesis. The method is demonstrated on a significant design—a 32-bit microprocessor. This example shows that the data-driven circuit style provides better performance than control-driven synthesized circuits. This paper extends previous reported work by illustrating how conditional execution, oft-cited as a problem for data-driven descriptions, is handled within the system, and by a more detailed analysis of the design example.**

***Index Terms*—Asynchronous design, Balsa, circuit synthesis, data-driven, digital design, handshake circuits, syntax-directed.**

I. Introduction

BEFORE asynchronous synthesis techniques will be seriously considered over their synchronous counterparts, they must demonstrate that they can achieve competitive performance. The research reported here aims to improve the performance of large synthesized asynchronous circuits. The focus of the approach is on a handshake circuit representation of the circuit; that is to say, an abstract representation of the structure of the circuit which is independent of technologies, protocols, data encodings, or any other details of the actual circuit implementation.

The handshake circuit paradigm allows the construction of large scale circuits by the composition of small handshake components that are straightforward to implement in isolation. Hardware descriptions are written in a high-level language and compiled in a syntax-directed fashion into the handshake circuit representation. This means the structure of the resulting circuit is directly related to the source code, allowing optimizations and tradeoffs to be made at the source code level. Furthermore, writing circuit descriptions in languages such as Balsa [1], [2], [8], and Haste[1] is relatively straightforward, even for novices. However, control overhead in the conventional control-driven style of handshake circuit synthesis is a major obstacle to performance.

Manuscript received June 27, 2008; revised December 17, 2008. This work was supported by EPSRC.

The authors are with the Advanced Processor Technologies Group, School of Computer Science, The University of Manchester, Manchester M13 9PL, U.K. (e-mail: sam@sam.taylor.name; doug@cs.man.ac.uk; plana@cs.man.ac.uk; tarazona@cs.man.ac.uk).

Digital Object Identifier 10.1109/TVLSI.2009.2020168

[1][Online]. Available: http://www.handshakesolutions.com/Technology/Haste/

Previous work [32], [33] gives details of how the control overhead arises in a conventional control-driven synthesis style, an overview of previous attempts to mitigate its effects [4]–[6], [10], [13], [22] and an in-depth justification for a data-driven approach. For definitive background material on handshake circuit compilation, see [3].

Essentially there are the following three principal effects contributing to the control overhead.

- All inputs are synchronized with each other before any operations within a block can proceed because only control may activate processing operations and therefore it is necessary for the control to know that the inputs it will use are available.
- Reads and writes are sequenced to ensure that variables are not written and read concurrently.
- Data processing operations only begin after the control initiates them due to the pull style of operation. If the data processing were to operate in parallel with the control then the overhead of the control should be reduced.

Data-driven asynchronous design styles are thus much less prone to the problem of control overhead, however, attempts to automate transformations to existing Balsa handshake circuits to produce more efficient structures along the lines of existing dataflow style compilation strategies [34], [35], [37] were not successful because it was not clear what the result of such optimization should look like and it was very difficult to guarantee the resulting circuit would behave in the same fashion as the original. Techniques such as data-driven decomposition (DDD) [37] rely on pipelining sequential programs and produce modified circuit behavior. A Balsa designer may depend on the design behaving in the manner it was written which could easily not be the case after optimization; indeed if a DDD-type strategy were to have been applied to the SPA processor [21] the memory interface would have broken.

The approach presented here attempts to combine the benefits of a data-driven style with the convenience and flexibility of the handshake circuit paradigm which allows the robust synthesis of large circuits. To this end, the handshake circuit structures of the control-driven Balsa synthesis method have been examined and data-driven alternatives are proposed. To generate these structures, a data-driven description style is proposed and a compiler has been developed to compile these description into a handshake circuit representation. This compiler is integrated into the Balsa design flow enabling the use of existing Balsa tools for moving from the handshake circuit representation to a gate-level circuit.

The organization of this paper is as follows. Section II reviews other related work. Section III examines how "classic" Balsa handshake circuit templates can be more efficiently replaced by data-driven variants. Section IV examines problems

with conditional structures and introduces new handshake components required for a data-driven system. Section V introduces a new data-driven Balsa language. Section VI describes the implementation of a significant design example. Finally, the performance, area and power consumption are compared for the design example synthesized both by "classic" Balsa and the new data-driven Balsa.

II. Related Work

A. Data-Driven Approach

The syntax-directed synthesis approach generates control-driven structures, but it is noticeable that in clocked design much emphasis is placed on pipelined datapath design; similarly most asynchronous techniques are based on dataflow pipelined style approaches.

Muller [18], [19] described the first asynchronous pipelines using C-elements as latches with combinational logic between the latch stages. The latency of each pipeline stage can be reduced by incorporating logic within registration stages. This has led to the concept of pipeline templates which define the registration control of a stage and allow for transistor-level designs to be incorporated within them. Williams [36] developed the PC0 dual-rail pipeline. Subsequently Lines [15] developed the concept of templates based on precharge half- and full- buffers. Other implementation styles, aimed at high performance, often requiring careful timing control include GasP from Sun Laboratories [25], [30] and the IPCMOS pipelines from IBM [25]. Singh and Nowick [26]–[28] have developed a range of high performance pipeline styles with a variety of tradeoffs; these papers contain an excellent review of asynchronous pipeline techniques.

While the works described above are indicative of the interest in pipelining techniques, they are not directly applicable to the problem addressed in this paper: they are concerned with circuit level implementations of pipeline architectures rather than the problems of how to specify the composition of data-driven circuits from a behavioral synthesis language.

Sparsø [29] analyzed the performance of a variety of pipeline topologies in terms of token flow through the structures and quantified performance in terms of forward and reverse latency, and cycle time. Again, although this material leads to a good understanding of the complexity of pipeline structures, it is not directly applicable to automated synthesis techniques.

B. Other Synthesis Systems

Automated synthesis techniques for large scale systems do not have to be restricted to a syntax-directed handshake circuit approach.

1) Desynchronization-Based Methods: These involve converting conventional synchronous design descriptions into asynchronous designs [7], [14]. Typically existing CAD tools are used for much of the datapath synthesis and asynchronous control synthesis tools are used to produce controllers that replace the global clock. This approach has the advantage that designers need little specialist knowledge of asynchronous techniques. A drawback is that by using a design targeted at a synchronous implementation, potential advantages of asynchronous techniques are not exploited. For example, concurrency is restricted to the synchronous pipeline structure and so the fine-grained concurrency possible in asynchronous design is not exploited. It is also difficult to exploit the possibility for asynchronous designs to use data-dependent delays instead of the worst-case delays of synchronous design.

2) Communicating Hardware Processes (CHP)-Based Methods: The CSP [12]-based CHP language is the basis of some asynchronous synthesis systems [17], [23], [31]. These systems use manual or automatic program transformations to refine a design into a more concurrent version. The final program is then translated into a production-rule set which is used to generate a transistor implementation of the design.

The Caltech synthesis tools (CAST) have been used to produce some high performance circuits [16] but these rely on significant manual intervention in the synthesis flow to arrive at the most effective program transformations and also rely on the use of the PCHB (precharge half-buffer) circuit style. This circuit style is not widely used and requires a specialized cell library.

The automatic program transformations employed in CAST are not behavior preserving and are only correct for designs that meet particular requirements. An inexperienced designer may struggle to understand and meet these requirements.

C. Handshake Circuit Optimizations

Attempts have been made to apply control resynthesis to the control of both Tangram/Haste [13] and Balsa [4], [5]. Control resynthesis attempts to improve the performance of the control tree by clustering sections of the tree, determining the overall behavior, and synthesizing a new controller to implement this behavior using a controller synthesis tool [6], [9], [10]. By removing the communications between clusters of components, the resulting controller should improve performance over the original control tree.

Control resynthesis is effective but limited. Improving the speed of the control tree will obviously help reduce control overhead but only so much improvement can be gained. The control still synchronizes with data at the same points and so the sequential operation of the control-driven structure is still maintained. Control resynthesis is complementary to other approaches to improving control overhead including the data-driven style introduced in this paper

Hansen and Singh [11] describe source-to-source transformation of the original specification into a new one using a variety of concurrency-enhancing optimizations: automatic parallelization, automatic pipelining, arithmetic optimization, and reordering of channel communication. Considerable speedups are claimed. However, some of the examples used start with extremely naive code sequences, so it is easy to obtain significant improvements. Unlike the techniques described in this paper, their work is not guaranteed to preserve the original behavior of the hardware and frequently the designer is given responsibility for ensuring that the behavioral changes are acceptable. This reduces the usefulness of an "automated" approach as it is necessary for the designer to understand the nature of the transformations to ensure they are safe, In contrast, the work described in this paper maintains the source-level predictability of the description—"what you write is what you get" and the

performance improvements are made over highly optimized and efficient control-driven descriptions.

Nielsen [20] has explored optimizations of circuits synthesized by Balsa; the work is concerned with resource allocation and bindings to explore the optimization space but within the confines of the conventional control-driven approach offered by Balsa. The work has recently been extended to Haste. A similar approach taken by the Moods system [24]. No attempt is made to generate data-driven circuits in these approaches although it is noted that the control structure arising from syntax-directed translation results in poor performance.

III. Data-Driven Circuit Structures

In the context of handshake circuits, a data-driven style should achieve greater speed performance because of the following:

- all control is activated in parallel;
- sequencing is localised to storage elements; read and write sections of control can operate in parallel as the localised sequencing ensures that storage elements are not concurrently read and written;
- control and datapath can operate in parallel because of the push nature of data processing.

There are possible disadvantages to data-driven circuits; they are likely to require more area and to consume more energy. The localized control of the data-driven style consumes more area than the control-driven tree as instead of appearing once, the control is distributed in many places. This effect is exaggerated in delay-insensitive implementations where an increased amount of completion detection is required and the implementation of push-style variables is particularly expensive. However, the increased concurrency in this distributed control is a major factor in the increased performance. Energy consumption due to switching can also be expected to increase as a result of the increase in concurrent activity. Speculation can also be expected to have an impact on energy consumption as this involves extra switching activity in the datapath that need not occur in the control-driven style.

The data-driven circuit style will be introduced in this section by comparison with conventional handshake circuits generated by the Balsa system because the source code is freely available making it feasible to experiment with the system. However, it should be emphasized that the results here apply to any system using a similar syntax directed compilation method such as Haste. The data-driven style was largely developed by examining and adapting handshake circuit structures so comparison provides the most instructive method of introduction. Some of the eight new handshake components required to support the new style are mentioned. More details can be found in [32].

A. Input

The conventional Balsa input structure is shown in Fig. 1. This structure is produced by the active enclosure construct shown below.

```
a, b -> then
     <body - a used once, b used twice>
end
```

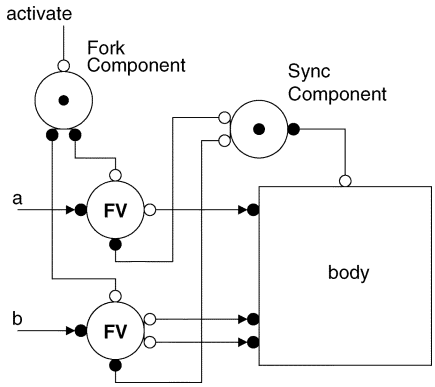

Fig. 1. Balsa input structure.

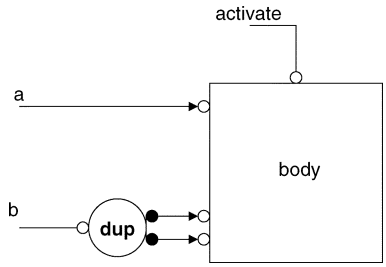

Fig. 2. Data-driven input structure.

In Balsa, channels (that is data wrapped in a request/acknowledge handshake) stand on the left-hand side of an expression and the symbol -> denotes a read from the channels, either into a variable or into another channel, or to a code block that has channel-like interfaces. Similarly the symbol <- denotes writing to the channel on the left hand side of an expression. In this example, the activation of the input command is used to initiate pulling data from the environment on the input channels `a` and `b`. The *Fork* component passes the activation request in parallel to the two *FalseVariable* (FV) components. These are used to hold open data (the data on the channels `a` and `b` is not released until the handshakes on those channels complete) and implement multicast on the input channels. The body of the structure is activated following the signal ports of the *FalseVariable* component being synchronized at the *Sync* component. This activation indicates the availability of the data for the body to then pull it from the read ports of the FV when required.

The data-driven style makes use only of push structures. Instead of using the FV to implement multicast, an alternative push structure must be used. As the input channels are now push channels, there is no need to pull the input data. For inputs that are used in only one place, the data can be pushed directly to the body. For inputs that are used more than once, a duplicate of the data must be sent to all the required places. The *Dup* component is used to implement this broadcast behaviour. Fig. 2 shows the data-driven version of the circuit example given in Fig. 1.

An advantage of this approach is that the input channels do not need to be synchronized before activating the body as the body no longer needs an activation to indicate the availability of the data; the data will be pushed to the required places at some point.

The obvious drawback with this approach is that, as the original structure implemented multicast, the body was free to select

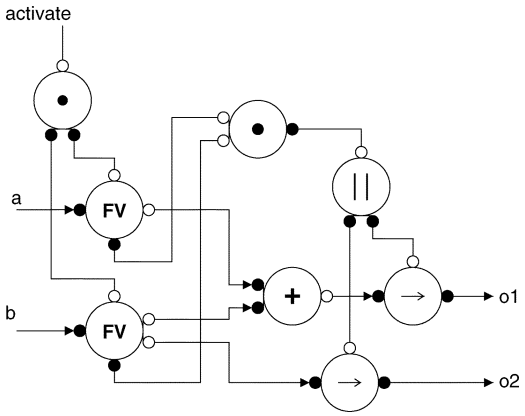

Fig. 3. Balsa data processing structure.

which read ports, if any, of the FV to use. Where conditional structures are used, the data is only conditionally required. In the broadcast structure, the data is sent to all possible destinations whether they need it or not. The resolution of this problem is discussed in Section IV.

B. Variables

They are implemented by the *Variable* handshake component. This component has a passive input known as the write port and one or more passive outputs known as the read ports. The control-driven approach allows data to be written to the *Variable* component by pushing to the write port and read from the variable by pulling from the read ports. The language ensures that the variable is not written at the same time it is read. To the designer, a Balsa variable therefore looks very much like a variable found in most imperative programming languages.

In the data-driven style, the storage component is called the *VariablePush* and has active push "read" ports. Unlike the original *Variable* component, this component has a write-once, read-once behavior; each time a data value is written it is automatically pushed on all read ports and the handshake on all read ports must then complete before the next write data is accepted. This makes a data-driven variable much more akin to a channel that has storage, thereby allowing each end of the channel to complete independently. In the control-driven style, one is free to use each read port one or more times or not at all. The drawback of the write-once read-once nature of data-driven variables is that each port must be used once and only once. This allows more flexibility in the control-driven style but where the designer uses this flexibility, they do so at the cost of performance.

In common with the data-driven input structure from the previous section, the drawback of this approach is that the data that is pushed on the read ports of the variable may not actually be required by the destination.

C. Data processing

The original Balsa data-processing structure is a pull structure implemented using the *Fetch* component to initiate a read of the required data from the required *Variable* or *FalseVariable* components, pull it through pass-through data components, and then push it to the destination. The following Balsa code produces the example handshake circuit structure shown in Fig. 3.

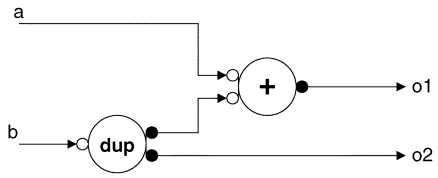

Fig. 4. Data-driven data processing.

TABLE I
CONTROL COMPLEXITY FOR N-BIT DATAPATH

Parameter	Balsa	Data-driven
Gate Count	8n+6 AND/OR gates 2n+8 C-gates	5n AND/OR gates n+2 C-gates
Forward Latency	4 AND/OR gates 1 C-gate	1 AND gate
Reverse Latency	2 AND/OR gates 3 C-gates	1 AND/OR gate 1 C-gate

```
a, b -> then
   o1 <- a + b ||
   o2 <- b
end
```

The input channels `a` and `b` are read (`->`) by the code block that follows. Here the sum of `a` and `b` is written (`<-`) to the destination channel `o1` and concurrently (`||`) `b` is written to channel `o2`.

As shown in the preceding sections, in conventional Balsa, *Variable*s and *FalseVariable*s have passive read ports whereas in the data-driven style, data is always pushed to all places where it may be required. In the data-driven style this data is pushed straight through the push datapath components to the destination as shown in Fig. 4.

The handshake circuit graph for the data-driven circuit is certainly a lot smaller but what impact does it have on the control part of the circuit? Table I summarizes the key parameters in a typical dual-rail implementation of the control circuitry for the two approaches.

IV. CONDITIONAL STRUCTURES

A. Conditional Execution

Conditional execution is supported by the `case` and `if` structures in Balsa. This section will take the `case` construct as an example as it is more commonly used than `if`, and the implementation of `if` is fundamentally the same as that of `case` with a few extensions.

The following Balsa code is an example of the use of the `case` construct. The control input `c` is used to determine whether to send the sum of `a` and `b` or just `b` to the output `o1`. This code is compiled into the handshake circuit shown in Fig. 5.

```
a, b, c -> then
    case c of
      1 then
        o1 <- a + b
    else
        o1 <- b
    end
end
```

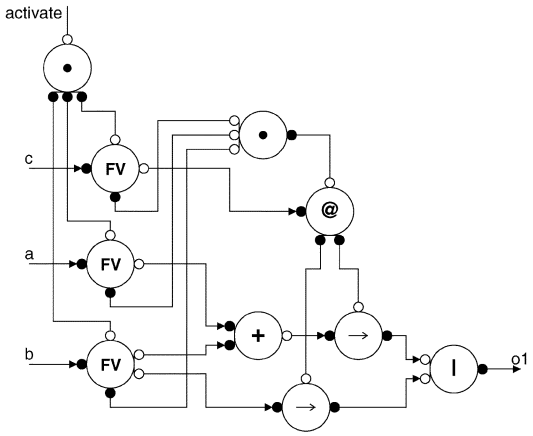

Fig. 5. Balsa conditional structure.

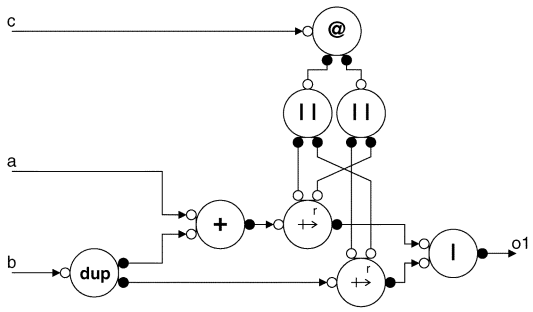

Fig. 6. Data-driven conditional structure.

As usual, the handshake circuit operates by requesting the three inputs, synchronizing on their arrival and then activating the body. The body pulls `c` from the *FalseVariable* into a *Case* component (@) that decides which of its sync outputs to activate based on the value of the control data that has been input. The standard data-processing structure is then used to pull the required data and send it to the output. Additionally in this example, the *CallMux* component (|) merges the two possible sources for output `o1` onto a single output channel. As the *Case* component will only activate one of its outputs at any time the *CallMux* will only receive an input on one input channel at a time, thereby avoiding any hazards.

The data-driven equivalent of this circuit is shown in Fig. 6. The difference between the data-driven style and the control-driven style is that as all inputs are pushed (see Sections III-A and III-B) all the data processing operations are initiated, even though the result may not be required. In order for the circuit to operate correctly these extra results must not be allowed to propagate. The *FetchReject* component ($|\!\!\rightarrow$) is introduced to "reject" the unwanted data. *FetchReject* is so named because it is rather like a push version of the *Fetch* component. Instead of pulling data and sending it to the output, it waits for pushed data to arrive on the input and then either passes it through to the output or completes on the input channel without sending anything on the output, thereby "rejecting" the data. Two sync ports are provided on the component, the activation port which is used to instruct that the data should be passed and the reject port which is used to instruct that the data should be rejected.

Once the *FetchReject* components are in place, all that remains is to connect the activation and reject ports to the correct outputs of the *Case* component. In this simple example, one is activated while the other is rejected. This arrangement allows the *CallMux* component to be used as in the original Balsa circuit because concurrent input handshakes are avoided by correctly using the *FetchReject* components.

As the data-driven style does not require synchronization of the inputs, there is potential for performance improvements over the control-driven circuit. The logic in the *Case* component is able to proceed as soon as the control data arrives, and in parallel with the data processing rather than always having to complete before initiating the pull data processing. However, the data-driven style is essentially speculating on needing the results of all operations. When using a conditional structure the unwanted results must be rejected and the overhead of this operation may harm performance. However, it is believed that generally this overhead should rarely be significant for the following reasons.

As the rejection will often occur in parallel with other useful operations, its effect on the overall performance should be limited. Only where the reject takes longer than useful processing will it reduce the overall performance as both must be completed before the next "cycle" of the operation. The reject operation itself is quite efficient but if the arrival of the data is slow then the overall impact may be greater.

In cases where there is no operation in parallel with the reject, it may often be the case that the data will arrive in advance of the reject signal and the rejection will therefore be concluded quite swiftly. Note that in the Balsa circuit, it is still necessary for all the inputs to arrive before the operation can complete even if no data processing is actually performed. Furthermore, in the Balsa circuit, the logic in the *Case* component does not begin evaluating until all the inputs have arrived whereas in the data-driven approach the evaluation can occur in parallel with the arrival of the inputs and so the *FetchReject* may have received the reject by the time the data arrives so it will at least be immediately rejected, albeit following a possible additional delay through some data processing logic. In the conventional Balsa case, all the inputs must arrive before the process of deciding what to do with them can begin.

Even so, it may be the case that unbalanced datapaths could cause a problem. Consider the example shown in Fig. 6. Here one of the operations is an addition while the other is simply passing through the data from input `b` unchanged. The addition is most likely to incur a significantly longer delay than the pass-through operation. If the second operation is selected frequently, and assuming the environment can supply inputs and consume outputs quickly enough, there is the potential for the rejection of the add operation to reduce the throughput of the overall circuit.

However, experience in designing with Balsa has shown that the delay of the control nearly always exceeds that of the datapath so it is reasonable to be optimistic that many datapath delays incurred as a result of speculation will be entirely masked by the delay of the control that works out whether or not to reject. Additionally, the inputs needed for the datapath operation may arrive earlier than those for the control allowing the datapath to complete before the control signals arrive at the *FetchRejects*.

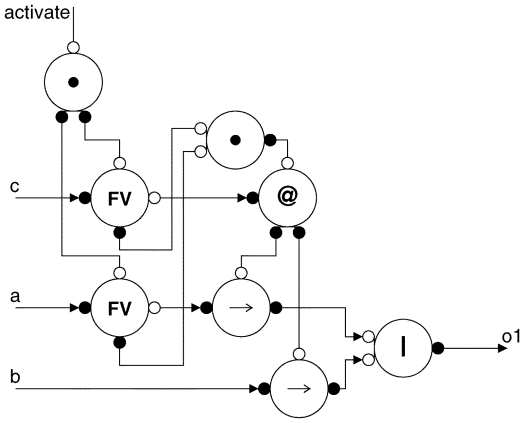

Fig. 7. Balsa conditional input structure.

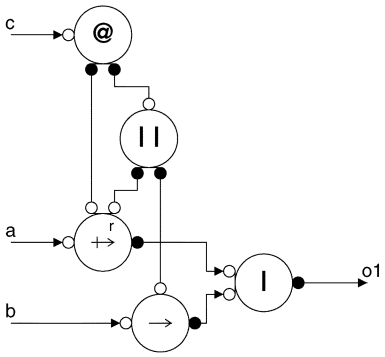

Fig. 8. Data-driven conditional input structure.

B. Conditional Input

Conditional inputs may occur in Balsa code when an input is made as part of the body of a conditional structure. For example, in the code below, channel b is a conditional input in the else clause of the case construct.

```
a, c -> then
    case c of
      1 then
        o1 <- a
    else
        b ->  then o1 <- b end
    end
end
```

During the operation of this code, data is only pulled on input channel b if the else clause is activated. Otherwise no communication occurs on channel b. This code is compiled into the handshake circuit shown in Fig. 7.

The important thing to notice when this circuit is converted to the data-driven style is that when data arrives on input b, it is always used; there is no need to reject any data if the else clause of the case is not executed, as the input never arrives. Of course, in a data-driven style there may be a request pending on channel b but this should be acknowledged by a subsequent cycle of the circuit when the else clause is executed. It is important that, until the else clause is taken, this request is not propagated to a downstream component which may acknowledge out of sequence possibly causing erroneous behavior. To avoid this possibility the *FetchPush* component is used. This component can be considered as a push version of *Fetch*, or a version of *FetchReject* without a reject.

To further explain the above, consider the example in Fig. 8 which is the data-driven equivalent of the example in Fig. 7. The *FetchPush* component is used on channel b to ensure any request on b is not passed to the *CallMux* component before the *Case* has decided that that operation should occur. This ensures the inputs to the *CallMux* cannot occur concurrently.

Combining conditional and unconditional inputs in an expression is more challenging. Consider the following code example, only a small modification to the last example given above: the data on one of the channels (a) is used twice.

```
a, c -> then
    case c of
      1 then
        o1 <- a
    else
        b -> then o1 <- a + b  end
    end
end
```

In this example, if the else clause is not taken then any data that is pending on b is not to be rejected but data on a into the adder must be rejected. In general, this problem will occur any time conditional inputs are combined in an expression with unconditional inputs. In order to avoid this problem, a scheme could be devised to reject the unconditional inputs before they are combined with the conditional inputs [32]. However, such a scheme reverses part of the advantage of adopting a push style as the datapath operations are once again stalled waiting for control to decide whether the result of the operation is required, instead of control and datapath operating in parallel. Furthermore this scheme presents additional complexity in compilation as the placing of rejections is now much less straightforward. For these reasons, such a scheme has not been used. Instead, combinations of conditional and unconditional inputs within expressions are considered invalid by the compiler, avoiding the need to produce an implementation at the expense of some reduction in flexibility. However, as discussed in Section V-G, users are still able to implement this scheme in the source description if they choose to.

C. Nested Conditionals

Conditional structures in Balsa can be nested within one another as demonstrated by the following code.

```
c, d -> then
    case c of
      1 then
        case d of
          1 then <body X> end
    else
         e -> then
            case e of
              1 then <body Y> end
        end
    end
end
```

In the control-driven style the output activations from one conditional structure are simply used to activate the nested conditional. In the data-driven style, the evaluation of the logic in

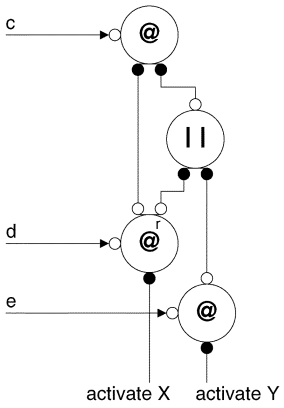

Fig. 9. Data-driven nested conditional structure.

```
-- One-place buffer
procedure buf (input  i : 1 bits;
               output o : 1 bits ) is
   variable x : 1 bits
begin
   input  i
   output x
   during
      x <- i
   end

  input x
   output o
   during
      o <- x
   end
end
```

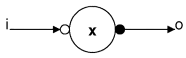

Fig. 10. Data-driven one-place buffer description and handshake circuit.

all *Case* components proceeds concurrently, but the output activations of nested conditionals must be delayed pending an activation from the outer structure as shown in Fig. 9. This example demonstrates the use of the *CasePush* and *CasePushR* components.

CasePush is used where it is necessary to synchronise with an activation before output activations are made from the *Case* component. This is true for the *Case* component whose input is channel `e` as data will only arrive on `e` when it is required.

It may be necessary to reject the input to a *CasePush* if data will arrive that is not required, as in this example with the *Case* component whose input is channel `d`. *CasePushR* is simply a *CasePush* with a reject input that upon activation will discard the input data without activating any outputs. The reject port is then activated on all conditions where the activate port is not.

V. New Input Language

This section will briefly introduce the high-level language that is translated in a syntax-directed fashion into the new circuit structures. Note that the language was primarily conceived as a means to an end; that is, to generate the data-driven handshake circuits. In the same way that all valid Balsa descriptions may be compiled to functional circuits, so all data-driven descriptions may similarly be compiled to data-driven structures. This means that the language reflects the dataflow style of the circuits and is less flexible and less familiar than the sequential programming language style possible in Balsa.

```
-- One-place buffer
procedure buf (input  i : 1 bits;
               output o : 1 bits) is
variable x : 1 bits
begin
   loop
      i -> x -- Input communication
      ;
      o <- x -- Output communication
   end
end
```

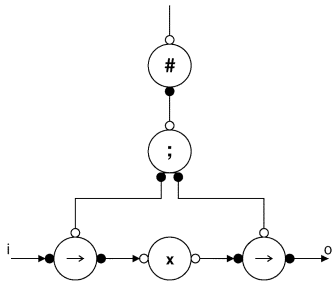

Fig. 11. Balsa one-place buffer description and handshake circuit.

The language is designed to resemble conventional Balsa wherever possible. Unlike Balsa where a circuit consists of commands linked by sequential or parallel control, the data-driven approach consists of lists of commands that operate independently and in parallel. Unlike the control-driven approach, control sections of the circuit do not wait for an activation but proceed as far as they are able, pausing only when awaiting data.

A. Hello World!

The equivalent of a Hello World program in Balsa is the one-place buffer. This serves equally well as an introduction to the data-driven language and is shown in Fig. 10.

It can be seen from this small example that much of the language is very similar to conventional Balsa. The declaration of the procedure and the input and output ports is identical. Unlike conventional Balsa, the procedure input ports will always be passive due to the push style of implementation. Internally to the procedure the input ports are treated as read-only channels and the output ports as write-only channels.

The main new feature in evidence here is the division of the procedure into blocks consisting of input and output declarations and a body containing the commands that use the inputs and generate the outputs. Unlike Balsa, the control structures of the circuit are largely implicit. Blocks implicitly operate in parallel, as do the list of commands within the blocks. The only synchronization between the two blocks in this example takes place at the variable; the read must complete before the next write can overwrite the data in the variable. This allows the variable reads and writes to overlap to the largest possible extent.

The handshake circuit for this buffer is simply a *VariablePush* component; this should be compared with the control-heavy Balsa generated circuit of Fig. 11.

B. Variables

The control-driven style of Balsa allows variables to be accessed in a very general fashion, so as to appear very similar to variables in a standard programming language. Variables can be

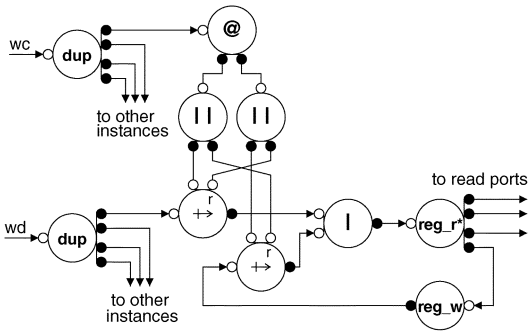

Fig. 12. Simplified register cell.

read and written in any arbitrary sequence. The *Variable* component has passive read and write ports and the control tree initiates communication on these as required. In the data-driven approach, the *VariablePush* immediately pushes any data written to it out of its active read ports. This means that a variable must always be read after it has been written. Variables therefore resemble less those of standard programming languages and are much more similar to channels. In fact, it may be more helpful to think of a variable in the data-driven style as a channel that contains storage, or even as a type of channel which each communicant can use at different times, rather than having to synchronize like a normal channel.

Reflecting this, variables are specified as inputs and outputs (to blocks—procedure ports only connect using channels) in precisely the same fashion as channels. In the following discussion use of the term channel generally implies a channel or variable except where otherwise stated.

C. Input Control

In a control driven approach, it is necessary for the control and data to synchronize to release the data once all required reads had been completed on the channel. As reads are now to be pushed, this synchronization is unnecessary as the release of all the read ports will indicate that all reads on the channel are completed. In the data-driven approach, therefore, inputs are merely specified as arriving at some point during the operation of the commands; the control waits for the arrival of inputs at any points where they are read (if they have not arrived already).

D. Write Command

The write command (e.g., `x <- i`) is used to output the result of an expression to an output channel (or variable). The channels written to must have been declared as an output from the block.

Compilation of the write command involves compiling the expression into appropriate push datapath components and connecting the result to the destination.

E. Arrays

Channels and variables can be arrayed in a similar fashion to Balsa. However there are some differences in the semantics of variable arrays. The code below demonstrates the full flexibility offered by Balsa for using arrayed variables.

```
input  i : array 0..3 of 2 bits
input  c : 3 bits
input  d : 2 bits
output o : 2 bits
output p : array 0..3 of 2 bits

variable v : array 0..3 of 2 bits

i -> v ;
loop
   c -> then
      case c of
        0b1xx then
          o <- v[(#c[0..1] as 2 bits)]
      | 0b0xx then
          d -> v[#c[0..1] as 2 bits)]
      end
   end
   ;
   p <- v
end
```

First, a single value is written to the entire array, then an individual element is read or written, and then the entire array is read as a single value. The strategy adopted by Balsa is to implement the arrayed variable using multiple *Variable* components, one for each element in the array. The control can then initiate reads and writes of the passive ported *Variable*s individually or as a group, splitting the write data and combining the read data as required.

A data-driven equivalent of this circuit structure presents substantial problems. Once each *Variable* component has been converted to *VariablePush* components, it is necessary to write to each *VariablePush* before it is read. After writing to a single element in the array, only that element would be available to read.

One option is to leave the management of the structure to the user, who must only attempt to read elements of the array that are written. Alternatively the user could be restricted to always writing to every element if they wish to use runtime indexing or an elaborate scheme to write-back the original data to those variable elements that are not written could be devised. This would ensure that every time any element in the array is written, all the other elements are also written (with unchanged data). To the read side, the arrayed variable always appears as if the entire array has been written.

None of these suggestions have been fully adopted. An arrayed variable declared in the data-driven language in the same fashion as a Balsa variable generates a single *VariablePush* that holds an entire value of the array type. The whole of the array must therefore be written to at once.

Variables can also be declared in a similar fashion to arrayed channels producing multiple variables in the implementation. Each of these variables must be written individually; the whole array may not be written by a single command. This second type of variable can be used by the user to generate a fairly close approximation of the functionality of the multi-variable Balsa structure by implementing, in the source description, the second of the schemes offered above. Although the functionality may be similar, the area used is substantially greater.

F. Structural Iteration

Structural iteration is a very useful language feature especially when combined with arrayed channels and variables. Essentially it allows the same code to be compiled multiple times with different channel and variable connections. For example, the following code is a simplified excerpt from the data-driven description of the register bank of the nanoSpa processor.

```
constant REGNUM = log REGCOUNT bits

array REGCOUNT of variable reg_usrw
array REGCOUNT of variable reg_usrr

input  reg_usrr
output reg_usrw
during
   for i in 0..REGCOUNT - 1
      reg_usrw[i] <- reg_usrr[i]
   end
end

input  reg_usrw, wc, wd
output reg_usrr, reg_svcr
during
   foreach i in reg_usrr
      case wc of
        (i as REGNUM) then
          reg_usrr[i] <- wd
      else
          reg_usrr[i] <- reg_usrw[i]
      end
    end
end
```

The code generates REGCOUNT instances of the circuit in Fig. 12. (The position of the channels that take data to the read ports are indicated on the diagram but the code for the read ports is not given above.)

Effectively this code generates a register "cell" for each register. In each "cycle" of operation the write control (`wc`) and data (`wd`) is duplicated to each cell and that cell compares the register address in the control against its own index. If they match then the write back data is written to that register, otherwise the original value from the register is written.

G. Restrictions

1) Combining Inputs: A conditional input is an input that is part of the body of a conditional structure. As explained in Section IV-B, such inputs cannot be combined with unconditional inputs in any expression: the following code will produce a compiler error since the operation `a + b` cannot be used where the input of `a` is conditional and `b` is not.

```
input  b, c
output o1
during
    case c of
      1 then
        o1 <- b
    else
        input a during
            o1 <- a + b
        end
    end
end
```

This restriction can be worked around by declaring another channel and making both inputs to the expression conditional as follows.

```
channel t
input  b, c
output o1, t
during
    case c of
      1 then
        o1 <- b
    else
        t <- b
        input a, t during
            o1 <- a + t
        end
    end
end
```

Note that by using this technique, less advantage is taken of the speculation as the case must be resolved before the channel `t` is written and the expression begins evaluation. Note also however, that the speculative evaluation of the addition is avoided in the case where the else clause is not chosen. This may be exploited for the purposes of improving performance or reducing energy consumption.

2) All Inputs and Outputs Must Be Used: All inputs and outputs that are declared must appear in the body of the block. (They must also be declared if they appear.) It is only necessary for the possibility to exist for each output to be produced. It is not necessary for every, or indeed any, output to actually be produced by the block when it is operating. Once an input is declared it will be assumed that a value will arrive from that channel or variable, but an output declaration means only that the block is the one that writes to the channel/variable, not that a value will definitely be written in any particular "cycle" of the block.

3) Output to Input Dependencies Must Not Be Disjoint: An output depends on an input if the input must arrive before the output can be produced. For example in the following code `o1` depends on `c` and `a`, `t` depends on `c` and `b` and `o2` depends on `a` and `t`.

```
channel t
input  a, b, c, t
output o1, o2, t
during
    case c of
      1 then
        o1 <- a
    else
        t <- b
    end
    o2 <- a + t
end
```

This gives three sets of input dependencies for each output: {c, a}, {c, b}, {a, t}. These are not disjoint as c appears in the first two and a appears in the first and third. This code is therefore valid, however the following is not valid.

```
channel t
input  a, b, c, t
output o1, o2, t
during
    case c of
      1 then
        o1 <- b
    else
        t <- b
    end
    o2 <- a + t
end
```

The sets of input dependencies for this code are: {c, b}, {c, b}, and {a, t}. The set containing a and t is disjoint from the other two sets. A separate block should be used to produce o2.

```
channel t
input  b, c
output o1, t
during
    case c of
      1 then
        o1 <- b
    else
        t <- b
    end
end

input  a, t
output o2
during
    o2 <- a + t
end
```

This rule helps to ensure the design is understandable as each block has a single "cycle" of operation due to the fact that all inputs are synchronized somewhere, though not necessarily with all others. For example, if the following code were valid then its meaning would be open to question but presumably, following the method of operation so far defined, o1 will be written every time a arrives, o2 would be written every time b arrives and there would be no synchronization between the two operations.

```
input  a, b
output o1, o2
during
    o1 <- a
    o2 <- b
end
```

In Balsa, if one were to write: o1 <- a||o2 <- b, then there is an explicit synchronization that takes place in the control. The data-driven style is designed to avoid making such synchronizations. In Balsa, there will be one communication on o1 and one on o2 before another takes place on either channel. In the data-driven style there could be infinite communications on o1 before any occur on o2 or vice versa. This could make designs much more difficult to understand.

VI. Design Example—nanoSpa

The benefits and drawbacks of the data-driven style have been explored using a large design example—nanoSpa which is a 32-bit microprocessor implementing what is essentially a slightly cut-down version of the ARM instruction set and which is a development of SPA [21], the first large scale design described in Balsa.

The nanoSpa has been gradually developed with the sole objective of making a Balsa synthesized asynchronous ARM of the maximum possible performance. Development has reached the stage where the processor implements all the main features of the instruction set and benchmark programs can be run in simulation to produce a good idea of the performance (which is almost ten times that of the original SPA). This makes it an excellent example in demonstrating whether a data-driven circuit can offer performance improvements over the best available conventional Balsa circuit. The demonstrator was chosen to do the following:

- demonstrate that the data-driven synthesis flow can be used to construct a significant design;
- compare the performance of a high performance Balsa design with the closest possible equivalent in the data-driven style;
- demonstrate the integration into the existing Balsa design-flow and the use of mixed Balsa and data-driven designs;
- attempt some level of qualitative comparison between the features and flexibility offered to the designer in both description styles.

A. Data-Driven nanoSpa

The data-driven nanoSpa has been described in the new data-driven input language. The description is roughly the same length as the Balsa original (∼3000 lines). As far as possible, the micro-architecture of the processor has been precisely copied from the Balsa description. As a consequence, most of the synthesized datapath logic is the same as the Balsa nanoSpa, and the control contains most of the significant differences.

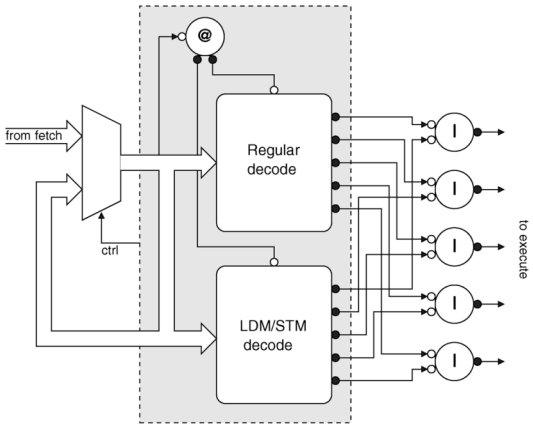

Fig. 13. Data-driven nanoSpa decode structure.

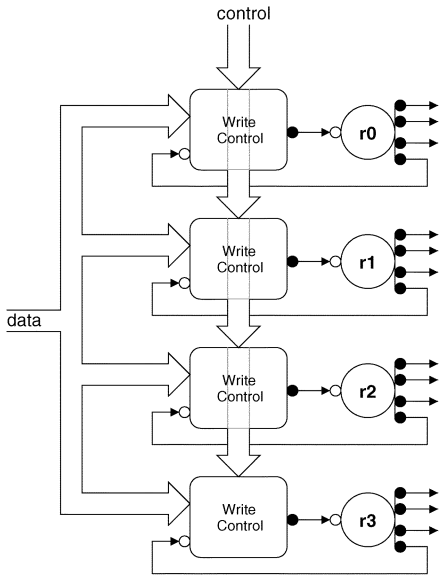

Fig. 14. Data-driven nanoSpa register structure.

The intention is to attempt to explore the advantage gained by using the data-driven style in describing a design that is as close as possible to a Balsa description, rather than by tailoring the design specifically to suit the data-driven style.

The two major exceptions where it was necessary to make significant changes to the architecture are in the decode unit, due to its use of (temporal) iteration, and the register bank, due to its reliance on Balsa-style variables. These issues are discussed below.

B. Decode

Unusually for a RISC-style processor, the ARM instruction set contains support for multi-cycle load and store instructions. These load and store multiple (ldm/stm) instructions allow any given subset of registers to be loaded from or stored to contiguous words in memory using a single instruction. The nanoSpa implements these instructions in the decode stage by simply generating and issuing multiple single memory transfer operations to the execute unit. The iterative decode for ldm/stm instructions makes use of the Balsa while loop structure to repeatedly generate memory transfer operations. In the control-driven style the handshake for the inputs to the decode can enclose all of this iterative operation allowing the inputs to be read repeatedly by each iteration.

In a data-driven style, iteration is implemented in a different way: instead of enclosing several variable and channel reads within a single input handshake, the input handshake is repeated as many times as required, each time reading variables and channels only once. It is quite straightforward to rearrange the structure of the decode to implement the multi-cycle instructions as shown in Fig. 13. In this structure the whole decode can be viewed as iterative with regular instructions simply being a special case requiring only a single iteration. When an instruction arrives at decode it is passed through the multiplexer to the decode logic. If the instruction is an ldm/stm, the necessary data for the next iteration is passed back to the multiplexer and the control signal is set so as to reinject the data as the next instruction. When the ldm/stm is finished, or after a single cycle if the instruction is a regular instruction, the multiplexer is signalled to inject the next instruction being sent from fetch. Although this may not be the most efficient implementation, it has the important advantage that the two blocks shown in the shaded area in Fig. 13 (for regular or ldm/stm instructions) can be copied directly to the data-driven description.

C. Register Bank

The Balsa nanoSpa register bank uses the general read and write structure for variable arrays discussed previously. The passive-ported *Variable* component allows reads and writes to occur to variables in any arbitrary order. As discussed earlier, it is not so easy to provide this general structure when using push style variables. In order to read from any variable, it is necessary for that variable to push its data. Therefore, in order to implement the register bank in the data-driven style it is necessary to write to every variable (i.e., register) during every cycle. The data-driven register bank write structure is illustrated in Fig. 14. The write control and data are here duplicated to individual write control units belonging to each register. These individual units decide whether to write the data to their respective register. If they do not write the data, they recycle the existing value and write this to the register instead. The subsequent read may therefore pick the appropriate data from any register as all registers will push data.

The data-driven register bank structure results in an individual cell for each register that controls the writes to that particular register (see Fig. 14). A read unit is generated for each read port. This structure results in improved performance but also significantly increases the area over the Balsa counterpart. It will also significantly increase the energy consumption as every register is written on every cycle.

D. Simulation Results

1) Individual Modules: Rather than simulating the entire processor, it is more instructive to examine the results from simulating individual modules within the processor. This avoids

TABLE II
PERFORMANCE OF DUAL-RAIL AND SINGLE-RAIL NANOSPA MODULES

Module	Test	Dual-rail Gates/Cycle			Single-rail Gates/Cycle		
		Balsa	Data-driven	Improvement	Balsa	Data-driven	Improvement
Fetch		59	29	2.0	47	33	1.4
Decode	regular	52	39	1.3	179	76	1.0
	ldm/stm (5 registers)	604	254	2.4	578	311	1.9
Register bank	1 write	134	69	1.9	82	61	1.3
	2 writes	182	74	2.5	99	61	1.6
ALU	and	74	41	1.8	57	33	1.7
	add 0 carry	85	74	1.1	75	42	1.8
	add 5 carry	86	74	1.2	83	52	1.6
	add 32 carry	107	65	1.6	137	106	1.3
	mov	77	57	1.4	56	32	1.8
ExecuteControl	non-memory	44	24	1.8	41	32	1.3
	memory store	57	30	1.9	46	32	1.4
	memory load	64	32	2.0	50	32	1.6
Execute	nop	83	58	1.4	70	56	1.3
	and	93	58	1.6	91	59	1.5
	and with shift	133	65	2.0	129	63	2.0
	ands (update flags)	95	58	1.6	91	59	1.5
	ldr/str	116	65	1.8	150	106	1.4
	branch	92	74	1.2	88	56	1.6

issues associated with the pipelining and processor architecture and demonstrates the performance improvements gained by using the data-driven logic style. The environments used in the test benches for these simulations all have zero delay. Generally, this favours the control-driven approach as, for example, the cost of synchronizing inputs that all arrive simultaneously is minimal. The results then show (to a close approximation) the minimum improvement achieved by the data-driven style. The fact that the data-driven style does not synchronize all inputs before beginning the operation and does not wait until the operation is complete before releasing them can potentially further improve performance.

The fetch, decode, and execute units together with some individual modules from the execute unit have been simulated. These latter modules were simulated, where appropriate, with different input data to demonstrate the data-dependent variation in performance. The relative performance of the modules in nanoSpa for both dual-rail and single-rail implementations are shown in Table II. The *Gates/cycle* figure is the number of gates the request-in signal goes through before emerging as an acknowledge signal to its data provider. It is a measure of maximum throughput of each module in the design. The relative areas of the units within nanoSpa (as measured by transistor count) are shown in Table III.

TABLE III
DUAL-RAIL AND SINGLE-RAIL NANOSPA AREA

Module	Dual-rail Transistor Count			
	Balsa		Data-driven	
Fetch		7667		17957
Decode		64394		271369
Register bank		68456		376914
Execute		143750		265707
ExecuteControl	5073		5754	
ALU	38687		53492	
Shifter	28987		85431	
Other execute	64311		107642	
Other		36482		24806
Total		**320749**		**956753**

Module	Single-rail Transistor Count			
	Balsa		Data-driven	
Fetch		5741		5280
Decode		37114		58825
Register bank		31468		71999
Execute		56754		72492
ExecuteControl	2887		3470	
ALU	9179		11270	
Shifter	13677		26249	
Other execute	28261		27720	
Other		20380		11234
Total		**151457**		**219830**

2) Register Bank Hybrid Design: The register bank has been highlighted as a particular problem in terms of area and energy consumption. A possible solution that may be easily implemented is to use the conventional Balsa register bank in place of the data-driven register bank. As the interface to both register bank designs is the same and the two design styles are integrated into the same flow, it is trivial to produce this hybrid design. This provides an excellent example of how designs with mixed Balsa and data-driven modules can be used. The lower area and energy requirements of the control-driven style can be exploited for noncritical modules, while the performance of the data-driven style is exploited for others.

3) Processor Performance: The processor was simulated at the transistor level using nanoSim to measure both speed and energy consumption with the processor running the Dhrystone benchmark. Table IV compares the energy consumption for a single Dhrystone loop and the performance of the control-driven design, the data-driven design and a hybrid design with a conventional Balsa register bank.

The dual-rail control-driven nanoSpa achieves 54 Dhrystone MIPS. The data-driven version achieves 85 Dhrystone MIPS, an improvement of 1.6 times the original. As can be seen from Table III, the area is significantly increased, from 320749 to 956753 transistors. As anticipated, a significant proportion of this increase is found in the register bank (from 68456 to 376914 transistors). If the increase in register bank area is ignored, then the data-driven nanoSpa is just over twice the size of the original Balsa version. As expected (for the reasons given in Section III),

TABLE IV
PERFORMANCE AND ENERGY CONSUMPTION PER DHRYSTONE LOOP

Style	Dual Rail DMIPS		Energy (μJ)	
Balsa	54.2	(1.00x)	0.36	(1.00x)
Data-Driven	85.2	(1.57x)	1.59	(4.44x)
Hybrid	67.9	(1.25x)	0.82	(2.29x)

Style	Single Rail DMIPS		Energy (μJ)	
Balsa	65.3	(1.00x)	0.15	(1.00x)
Data-Driven	92.7	(1.42x)	0.28	(1.85x)
Hybrid	80.9	(1.24x)	0.19	(1.23x)

TABLE V
ENERGY CONTRIBUTION BY NANOSPA MODULE

Module	Dual-rail Energy (%) Balsa	Data-Driven	Hybrid
Fetch	6	3	5
Decode	23	28	56
Register bank	4	50	3
Execute	44	16	32
Other	22	2	4

Module	Single-rail Energy (%) Balsa	Data-Driven	Hybrid
Fetch	8	4	6
Decode	30	28	43
Register bank	5	38	5
Execute	39	25	39
Other	18	5	7

the energy consumption is considerably greater for the fully data-driven version. Table V shows the contribution of the various modules to the energy consumption. It confirms that the largest energy consumer in the data-driven approach is the register bank, whereas for the Balsa implementation, the execute unit is the most significant. The dual-rail hybrid design achieves 68 Dhrystone MIPS (a speed improvement of 1.25) in dual-rail and uses 651057 transistors. Performance has been traded for reductions in area and energy consumption. In this case, the decode unit is now the most significant energy consumer. A lesson that can be drawn from these figures is that a careful analysis is required to identify those parts of the design contributing to bad performance in speed, energy, or area.

4) Single-Rail Performance: Tables II–V also show the results for the single-rail back-end. The performance improvement for the data-driven single-rail back-end (1.4×) is somewhat less than for dual-rail, probably because the control is more complex in dual-rail, but offers better opportunity for optimization in the data-driven case. It can be seen that the area penalty is much smaller for single-rail than that for dual-rail. Again, much of the increase is in the register bank (from 30480 to 79480 transistors). If the increase in register bank area is ignored then the data-driven design is only approximately 18% larger. The energy figures for the dual-rail back-end are considerably worse than for single-rail, one reason being that in dual-rail, data is pushed by twice switching one wire for each bit, whereas in single-rail, only one wire for each bit is switched (once) and only if the value of that bit changes.

VII. CONCLUSION

This paper has described contributions in the field of asynchronous digital circuit synthesis. The existing handshake circuit synthesis method has been examined and performance has been postulated as a major weakness. The overhead of the control-driven style of compilation has been identified as a significant contributing factor to the shortcomings in performance of the existing synthesis method. However, the handshake circuit paradigm is attractive because it is both flexible and robust, independent of any particular implementation style, straightforward to understand, and the transparent compilation allows source-level optimization.

A data-driven style of circuit would seem to offer potential for increased performance. Therefore an alternative data-driven style of handshake circuit structure has been proposed along with a language from which this circuit style may be compiled. The compiler to translate this language into handshake circuits has been implemented and integrated into the existing Balsa framework.

The data-driven style has been successfully demonstrated by the implementation of a complex 32-bit microprocessor design using the Balsa synthesis system. The potential improvements over the control-driven style have been demonstrated by comparison of this design with the equivalent control-driven implementation. The results are transferable to other synthesis systems based on handshake circuits such as Haste.

The increased area and energy requirements of the data-driven style have been briefly noted but these are unlikely to be disproportionate to the performance gains and could be decreased by further work on modified or alternative back-end implementation styles. Future work should also address new structures to better support the register banks.

Due to the variables and sequential and iterative control structures, it is possible in Balsa to write a naive sequential program that appears very similar to a conventional programming language. Such a program will compile and produce a functioning (but slow) circuit. In the data-driven style, it is necessary for the programmer to think in a different, more "asynchronous" manner as such sequential descriptions are not possible. It is also similarly necessary to do so when using conventional Balsa if good performance is required. The rewards of adopting a data-driven style with respect to performance are clear but the method introduced herein, being intentionally designed to be data-driven, is superior to adopting a data-driven approach with control-driven compilation. By using the handshake circuit paradigm and integrating the new style into the Balsa framework, it is straightforward to combine both styles in the same design-flow and so greater flexibility is offered to the designer.

The data-driven style has addressed the issue of the structure of handshake circuits and control overhead. In general, the performance of synthesized asynchronous circuits is still not competitive with their synchronous counterparts. More work is required at all levels of the design-flow before competitive performance is achieved.

REFERENCES

[1] A. Bardsley, "Balsa: An asynchronous circuit synthesis system," M.Phil. thesis, Sch. Comput. Sci., Univ. Manchester, Manchester, U.K., 1998.

[2] A. Bardsley, "Implementing balsa handshake circuits," Ph.D. dissertation, Sch. Comput. Sci., Univ. Manchester, Manchester, U.K., 2000.

[3] Kees van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge, U.K.: Cambridge University Press, 1993, vol. 5, Int. Series on Parallel Computation.

[4] T. Chelcea, S. Nowick, A. Bardsley, and D. Edwards, "A burst-mode oriented back-end for the balsa synthesis system," in *Proc. Des., Autom. Test Eur. Conf.*, 2002, pp. 330–337.

[5] 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.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Inf. Syst.*, vol. E80-D, no. 3, pp. 315–325, Mar. 1997.

[7] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006.

[8] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, *Balsa: A Tutorial Guide*. Manchester, U.K.: The University of Manchester, 2006.

[9] F. Fernández-Nogueira and J. Carmona, "Logic synthesis of handshake components using structural clustering techniques," *Integr. Circuit Syst. Des. Power Tim. Model., Opt. Simulation*, pp. 188–198, 2009.

[10] 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 University, New York, Tech. Rep. TR CUCS-020-99, Jul. 1999.

[11] J. Hansen and M. Singh, "Concurrency-enhancing transformations for asynchronous behavioral specifications: A data-driven approach," in *Proc. Int. Symp. Asynch. Circuits Syst.*, 2008, pp. 15–25.

[12] C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[13] T. Kolks, S. Vercauteren, and B. Lin, "Control resynthesis for control-dominated asynchronous designs," in *Proc. Int. Symp. Async. Circuits Syst.*, Mar. 1996, pp. 233–243.

[14] A. Kondratyev and K. Lwin, "Design of asynchronous circuits using synchronous CAD tools," *IEEE Des. Test Comput.*, vol. 19, no. 4, pp. 107–117, 2002.

[15] A. Lines, "Pipelined asynchronous circuits," Ph.D. dissertation, Dept. Comput. Sci., California Inst. Technol., Pasadena, 1995.

[16] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Pénzes, R. Southworth, and U. Cummings, "The design of an asynchronous MIPS R3000 microprocessor," *Adv. Res. VLSI*, pp. 164–181, 1997.

[17] A. J. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Developments in Concurrency and Communication*, ser. UT Year of Programming Series, C. A. R. Hoare, Ed. New York: Addison-Wesley, 1990, pp. 1–64.

[18] D. E. Muller, "Asynchronous logics and application to information processing," in *Symposium on Application Switching Theory to Space Technology*. Stanford, CA: Stanford University Press, 1962, pp. 289–297.

[19] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," in *Proc. Int. Symp. Theory of Switch.*, Apr. 1959, pp. 204–243.

[20] S. F. Nielsen, "Behavioral synthesis of asynchronous circuits," Ph.D. dissertation, Dept. Inf. Math. Model., Tech. Univ. Denmark, Lyngby, 2005.

[21] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple, "SPA—A synthesisable amulet core for smartcard applications," in *Proc. Int. Symp. Async. Circuits Syst.*, Apr. 2002, pp. 201–210.

[22] L. A. Plana and S. M. Nowick, "Architectural optimization for low-power nonpipelined asynchronous systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 6, no. 1, pp. 56–65, Mar. 1998.

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

[24] M. Sacker, A. Brown, P. Wilson, and A. Rushton, "Data-driven asynchronous circuits," in *Proc. IEEE Int. Symp. Async. Circuits Syst.*, Apr. 2004, pp. 125–134.

[25] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins, "Asynchronous interlocked pipelined cmos circuits operating at 3.34.5 GHz," in *Proc. ISSCC*, 2000, pp. 292–293.

[26] M. Singh and S. M. Nowick, "The design of high-performance dynamic asynchronous pipelines: High-capacity style," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 11, pp. 1270–1283, Nov. 2007.

[27] M. Singh and S. M. Nowick, "The design of high-performance dynamic asynchronous pipelines: Lookahead style," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 11, pp. 1256–1269, Nov. 2007.

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

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

[30] I. Sutherland and S. Fairbanks, "GasP: A minimal FIFO control," in *Proc. Int. Symp. Adv. Res. Asynch. Circuits Syst.*, Mar. 2001, pp. 46–53.

[31] TIMA Laboratories, Grenoble, France, "TIMA asynchronous synthesis tools," 2002. [Online]. Available: http://tima.imag.fr/cis/

[32] S. M. Taylor, "Data-driven handshake circuit synthesis," Ph.D. dissertation, Sch. Comput. Sci., Univ. Manchester, Manchester, U.K., 2007.

[33] S. Taylor, D. Edwards, and L. Plana, "Data-driven asynchronous circuits," in *IEEE Int. Symp. Asynch. Circuits Syst.*, Newcastle upon Tyne, U.K., Apr. 2008, pp. 3–14.

[34] J. Teifel and R. Manohar, " Static tokens: Using dataflow to automate concurrent pipeline synthesis," in *Proc. Int. Symp. Asynch. Circuits Syst.*, Apr. 2004, pp. 17–27.

[35] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, "C to asynchronous dataflow circuits: An end-to-end toolflow," presented at the IEEE Int. Workshop Logic Synth., Temecula, CA, Jun. 2004.

[36] T. E. Williams, "Latency and throughput tradeoffs in self-timed asynchronous pipelines and rings," Stanford Univ., Stanford, CA, Tech. Rep. CSL-TR-90-431, Aug. 1990.

[37] C. G. Wong and A. J. Martin, "High-level synthesis of asynchronous systems by data-driven decomposition," in *Proc. ACM/IEEE Des. Autom. Conf.*, Jun. 2003, pp. 508–513.

Sam Taylor received the B.Sc. and Ph.D. degrees in computer science from the University of Manchester, Manchester, U.K., in 2003 and 2007, respectively. His Ph.D. dissertation, from which much of the work in this paper was drawn, was on the synthesis of data-driven asynchronous circuits using a handshake circuit approach.

He was a Senior Engineer with Silistix Ltd., U.K., until August 2008. He is currently with the Wellcome Trust/Cancer Research UK Gurdon Institute, Cambridge, U.K.

Doug A. Edwards received the B.Sc. degree in physics and electronic engineering and the M.Sc. and Ph.D. degrees studying the properties of Zinc Sulphide Silicon Heterojunctions from the University of Manchester, Manchester, U.K.

He is currently a Reader with the School of Computer Science, the University of Manchester. After a period with Ferranti Ltd. as a Process Engineer improving the yield of CDI integrated circuits, he joined the School of Computer Science researching high density memory technology, high speed optical networks and hardware accelerators for printed circuit layout. His current research interests include computer-aided design for the synthesis of asynchronous circuits and has led the team which has developed the Balsa synthesis system.

Luis A. Plana (M'97-SM'07) received the Ingeniero Electrónico degree from Universidad Simón Bolívar, Venezuela, the M.S. degree in electrical engineering from Stanford University, Stanford, CA, and the Ph.D. degree in computer science from Columbia University, New York.

He is a Research Fellow with the School of Computer Science, University of Manchester. Before coming to Manchester, he worked with Universidad Politécnica, Venezuela, for over 20 years, where he was a Professor and Head of the Department of Electronic Engineering. His research interests include the design and synthesis of asynchronous, embedded, and globally asynchronous, locally synchronous (GALS) systems.

Luis A. Tarazona D. (S'07) received the Ingeniero Electrónico degree from Universidad Nacional Experimental Politénica, Venezuela, and the M.S. degree in communications and signal processing from the University of Bristol, Bristol, U.K. He is currently pursuing the Ph.D. degree from the School of Computer Science, University of Manchester, Manchester, U.K.

He is on leave from his post as a Lecturer with Universidad Politécnica, Venezuela. His research interests include the design and synthesis of asynchronous systems, digital circuits, and digital signal processing.