Automatic Compilation of Data-Driven Circuits

Sam Taylor*, Doug Edwards, Luis Plana
School of Computer Science, University of Manchester
Oxford Road, Manchester, M13 9PL, UK
{*smtaylor|doug|lplana*}*@cs.manchester.ac.uk*

Abstract

This paper describes a method of synthesising asynchronous circuits based on the Handshake Circuit paradigm but employing a data-driven, rather than the control-driven, style. This approach attempts to combine the performance advantages of data-driven asynchronous design styles with the handshake circuit style of construction.

The integration into the existing Balsa design flow of a compiler for descriptions written in a new data-driven language is described. The method is demonstrated using a significant design example — a 32 bit microprocessor. This example shows that the data-driven circuit style provides better performance than conventional control-driven Balsa circuits.

1. 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 synthesised 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. Control overhead in the conventional control-driven style of handshake circuit synthesis (as used in the Balsa [1, 5] and Haste [8] systems) is identified as the major obstacle to performance.

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 optimisations and trade-offs to be made at the source code level.

```
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
```

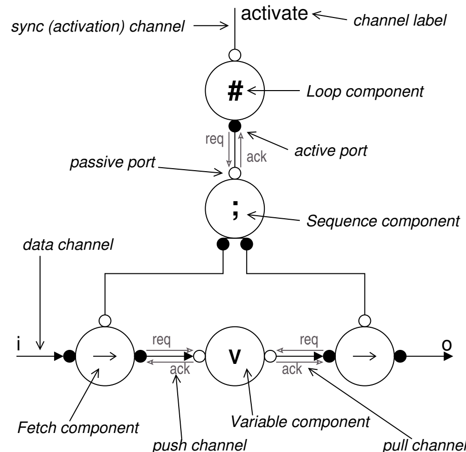

Figure 1. Balsa one-place buffer description & handshake circuit

A Balsa description of a one-place buffer and the equivalent handshake circuit graph is shown in figure 1. Handshake components are connected by channels. Each channel links an active port which sends the request to a passive port (which responds with an acknowledge) on another component. Data channels are represented by a line with an arrow that denotes the direction of the data flow. Data channels may be *push* meaning the data moves in the same direction as the request or *pull* meaning data moves in the direction of the acknowledge. *Sync* or *activation* channels that do not carry data are mainly used to activate the operation of various handshake components in the control part of the circuit.

*Sam Taylor is now with Silistix Ltd. Manchester, UK
This work was supported by EPSRC.

1522-8681/08 $25.00 © 2008 IEEE
DOI 10.1109/ASYNC.2008.14

IEEE computer society

Even in this small example, it is easy to see how the tree-like structure of the handshake circuit closely mirrors the control flow of the source description. This style of translation is described as control-driven. The inherent overhead of this style is discussed in the next section.

Data-driven asynchronous design styles are much less prone to excessive control overhead and so the approach of this research is 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 benefits of the new style are demonstrated by the manual translation of an existing Balsa design of significant size and complexity directly into the data-driven style.

2. Control Overhead

Compiled Balsa handshake circuits may be split into control and datapath. The datapath consists of *Variable* components, data processing structures and data channels. The control consists of a tree of control components connected with *sync* channels, which direct the movement of data around the datapath by activating interface components such as *Fetch*, *FalseVariable*, and *While*. This style of translation is described as control-driven meaning that the control tree is responsible for initiating all datapath operations. This approach is robust and flexible but there is a significant drawback: the control is nearly always slower than the data and as control and data are frequently synchronised, the data is frequently stalled waiting for the control to catch up.

An example will now be given that attempts to demonstrate how the control-driven structure contributes to control overhead. Figure 2 generalises the structure of a control-driven procedure which produces an output (*O*) and requires an input (*A*). Internally the process uses two variables (*V0* and *V1*). The operation of this structure is extremely sequential. Firstly, the portion of control labelled write is activated. The control decides whether to write some data to the *Variable* components. Once any data is written, it can be considered as being available for reading from the *Variable* components. However, the control must then complete

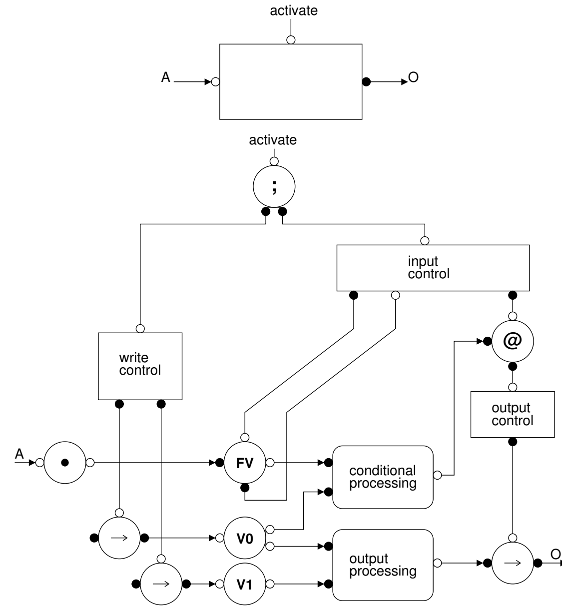

Figure 2. General control-driven structure

its handshake before the right-hand side of the tree is activated.

As well as data stored in variables, data from channels may also be used by means of input enclosure. The control (here labelled input) must activate the pulling of any such data to the *FalseVariable* (*FV*) component. The input control waits for the signal from the *FV* indicating the arrival of the data. Once again, it is not unlikely that this data has been available for some time on channel *A* which is awaiting synchronisation in order to deliver it.

All the required operands for the data processing operations have now been collected. The control may then initiate data processing operations. It may be necessary to decide what operations should be performed based on some of the data (e.g. if a case construct is used). Therefore the control may initiate some data processing operations using a *Case* (@) component for the purpose of making conditional choices. Following this, the final data processing operations that actually produce the outputs are initiated. These outputs are are written to variables or communicated on output channels.

When the variable writes and/or output communications are complete, the data in the *Variable*s and on input channels may be considered as being no longer required. However, all the handshaking in the control for inputs, conditionals and outputs must be completed before the write control is even activated again to begin the process of deciding whether to overwrite the data in the *Variable*s.

Note that for different processing operations, only a subset of the inputs may be required but all the inputs are synchronised with each other and the control before any operations begin. Furthermore, no inputs are released until after all data operations have completed, even though some may not be required after some operations have completed. If data were released sooner then other parts of the circuit will be allowed to proceed sooner as well.

The three principal problems in the structure of the handshake circuit that contribute to the control overhead are:

- All inputs are synchronised with each other before any further operations are begun. Data is available in *Variable* components before the read control is even activated. If control were operating in parallel with the arrival of data, then data may not be stalled as long waiting for the control to decide what to do. The control may even have resolved itself before the data arrives. If there were no need to synchronise all inputs before any operation can proceed, then processing, and control that relies on part of the data, can get a head start and operate concurrently with the arrival of the remaining data.
- The sequential activation of the read and write halves of the control tree. This sequencing is needed to ensure the *Variable* is not written and read concurrently. However, the location of the sequencing in the control tree is far away from the *Variable* leading to sequential operation of the two halves of the control tree. More concurrent operation of the two halves of the tree should increase performance.
- 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.

3. Combating Control Overhead

3.1. Control re-synthesis

Attempts have been made to apply control re-synthesis to the control of both Haste [9] and Balsa [3, 2]. Control re-synthesis attempts to improve the performance of the control tree by clustering sections of the control tree, determining the overall behaviour, and synthesising a new controller to implement this behaviour using a controller synthesis tool [4, 6]. By removing the communications between clusters of components, the resulting controller should improve performance over the original control tree.

Control re-synthesis is effective but limited as only so much improvement can be gained using the existing structure. The control still synchronises with data at the same points and so the sequential operation of the control-driven structure is still maintained. Control re-synthesis is complementary to other approaches to improving control overhead including the data-driven style introduced in this paper.

3.2. Concurrent sequencer

This technique specifically addresses the second of the three issues identified above by applying concurrent sequencing [15]. The original Balsa *Sequence* component has been enhanced to include concurrent sequence behaviour implemented using the ‘T-element’ [13]. The concurrent sequencer allows some overlapping between the write and read halves of the control. The read half of the control may be activated at the start of the return-to-zero phase of the write half, instead of waiting for the entire handshake to complete. This allows the write return-to-zero phase to operate concurrently with the read control. Unfortunately this cannot be done if the control is reversed so that reads precede writes as a write-after-read hazard would be inserted [13]. In some situations a write-after-write hazard may also prevent the use of concurrent sequencing. The Balsa compiler has been modified to automatically insert concurrent sequencing where it is safe to use it [17]. Concurrent sequencing provides performance improvements but it is limited in where it can be safely applied and only allows partial overlapping

3.3. Eager inputs

Eager inputs use modified *FalseVariable* components that activate the control without waiting for the data to arrive. The control is able to proceed up until the point where synchronisation with the data is required and there it stalls until the data does arrive [17]. Since the original publication, further work has shown that there are more conditions than originally thought where eager inputs may not be used. There is the possibility of automatically detecting when it is safe to use eager inputs, or allowing the designer to decide where they should be used by modifications to the source language, or a mixture of both these options. At this time, neither of these options have been fully integrated into the design flow.

While eager inputs allow control to get a head start before the arrival of data, it is still necessary to synchronise all the data and control before releasing the data. In a data-driven style, inputs allow early control activation without suffering from having to synchronise before re-

leasing the data.

3.4. Source description style

Due to the syntax-directed compilation, the source description is an important factor in the performance of conventional Balsa designs. The transparent compilation from language to handshake circuit structures gives the designer flexibility at the language level to optimise the resulting circuit. The nanoSpa processor which will be used as an example has been specifically designed to try and achieve the best possible performance from conventional Balsa. The techniques used in the source description are interesting. The use of variables, in most cases, is restricted to the pipeline registers of the processor. The pipeline registers are implemented using simple one-place buffer circuits. This is a very small handshake circuit with the sequencer located as close as possible to the *Variable* and, when combined with the concurrent sequencing, the performance of the pipeline register compares favourably to highly optimised controllers [11].

The logic within the pipeline stages is split into small modules that operate concurrently. Each module, therefore, has a relatively small control tree which helps to reduce control overhead. The control tree of each module has a *Loop* component at the head and so operates independently from other modules. Each module waits for data on its inputs, processes it and produces outputs. Instead of using a large monolithic control tree to direct the movement of data, small steering and merging modules are used to direct the flow of data. Apart from when modules must synchronise on channels to exchange data, they operate in parallel with one another. Within the constraints of the control-driven system, an experienced Balsa developer attempts to reduce control overhead by describing a system that is data-driven [14]. At least for an experienced designer, the data-driven style introduced in this paper is probably more suitable for describing what is desired than the control-driven style.

4. Data-Driven Approach

The data-driven style has been designed to reduce the impact of all three of the problems identified above. This is achieved by three particular facets of the design style:

- Data-driven control activation. Control is all activated in parallel, synchronising with data only when it is absolutely necessary and releasing it as soon as it has been used.

- Localised sequencing. Sequencing is located local to the variable component. The read and write sections of control can thereby operate entirely in parallel as the localised sequencing ensures that the variable is not concurrently read and written.

- Parallel operation of control and datapath due to a push data processing style.

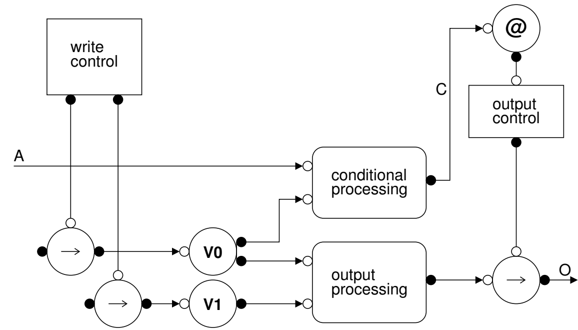

Figure 3. General data-driven structure

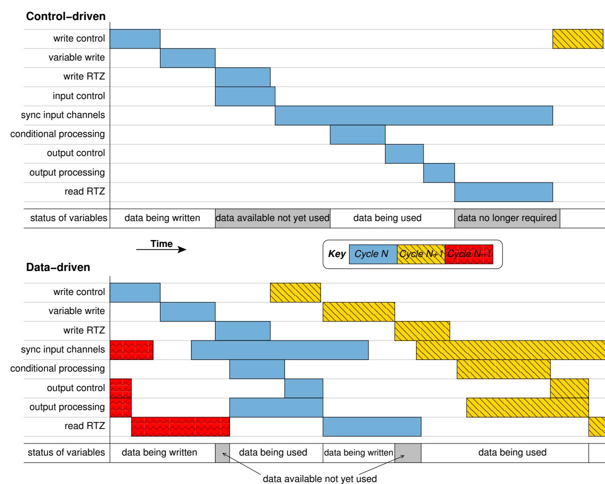

Figure 4. Control-driven vs. data-driven timings

The data-driven equivalent to the structure in figure 2 might be pictured as in figure 3. Figure 4 attempts to give a very rough example of how the data-driven structure enables much greater concurrent operation than the control-driven structure. Note how consecutive cycles of operation are overlapped due to localised sequencing and how the push structure enables processing to begin earlier as it does not need to wait for the control to activate it. Note also that the periods where data is stored in variables but no use is being made of it are much shorter. This figure is not based on real timings or drawn to an accurate scale and may be too optimistic but it shows, in general, how the data-driven style aims to reduce the impact of control overhead.

4.1. Why a data-driven approach?

- A data-driven approach is more commonly used in asynchronous circuit design styles. There are several examples of high performance data-driven style circuits such as the AMULET microprocessors [7] which were based on the Micropipeline paradigm and the Caltech MIPS [12] which used the CAST synthesis system.

- The data-driven approach should suffer from less control overhead than the control-driven style of Balsa for the reasons outlined in the preceding section. More parallelism is exploited between data and control by a data-driven style as there is less synchronisation between control and data.

4.2. Why a handshake circuit style approach?

- The handshake circuit approach is not specific to any particular implementation style. Many back-end implementations styles are possible. The compilation does not map direct to transistors or use unusual circuits such as PCHB [10] so it is much more flexible than some synthesis approaches. Standard-cell implementations are possible.

- Transparent compilation allows the designer to modify properties of the final circuit at the source level. This direct synthesis approach is relatively straightforward to understand. Any valid language description can be compiled into an implementation and there are no complex restrictions placed on the designer.

- The new data-driven style fits into an existing, proven design flow. This saved time and effort in the development of the style as existing tools and components are re-used.

4.3. Why not a data-driven approach?

- Due to the more restrictive data-driven style, data-driven descriptions are less flexible than those of conventional Balsa. In particular, the nature of Balsa variables means they can be used in a fashion familiar to most programmers but data-driven variables cannot. Additionally, no conditional iterative control structure is available in the data-driven style although these are less frequently used.

- Circuits in the data-driven style are likely to require more area and to consume more energy. The localised 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.

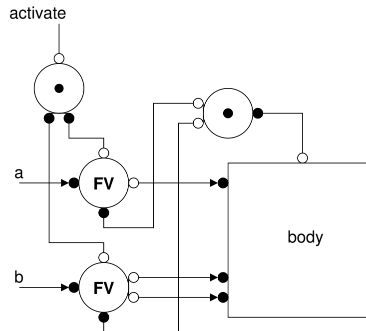

Figure 5. Balsa input structure

5. Data-driven Circuit Structures

The data-driven circuit style will be introduced in this section by comparison with the conventional Balsa handshake circuit style. The data-driven style was largely developed by examining and adapting Balsa 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 [18].

5.1. Input

The conventional Balsa input structure is shown in figure 5. This structure is produced by the active enclosure construct shown below.

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

The activation of the input command is used to initiate pulling data from the environment on the input channels, (a and b). The *FV* component is used to implement multicast on the input channels. The body of the structure is activated following the signal ports of the *FalseVariable* component being synchronised 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,

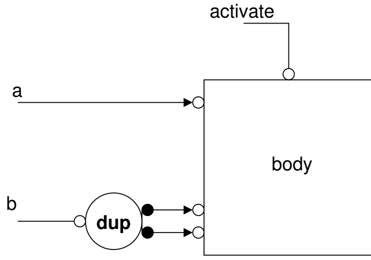

Figure 6. Data-driven input structure

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. See figure 6 for the data-driven version of the circuit example shown in figure 5.

An advantage of this approach is that the input channels do not need to be synchronised 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 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 outside the scope of this paper and is fully discussed in [18].

5.2. Variables

Variables provide data storage within the Balsa language. 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. This component allows variables to be very flexible. 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 pull structures are not used so this type of variable is not available. The replacement 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

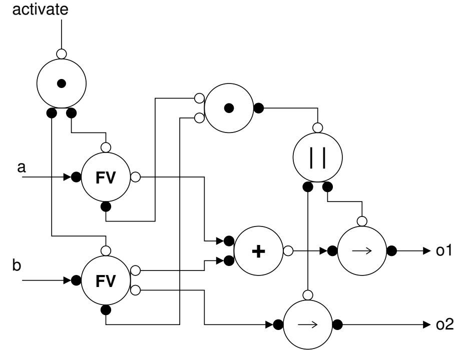

Figure 7. Balsa data processing structure

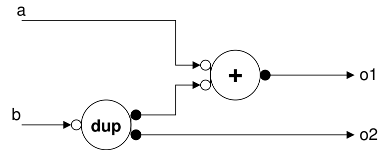

Figure 8. Data-driven data processing

behaviour; 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. Instead of a conventional variable, 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. This restricted behaviour is a major factor in the somewhat reduced flexibility of the data-driven descriptions over conventional Balsa.

In common with the 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.

5.3. 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 figure 7.

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

This code sends the sum of `a` and `b` to the destination channel `o1` and sends `b` to channel `o2`. As shown in the preceding sections, in conventional Balsa, *Variable*s and *FalseVariable*s had passive read ports

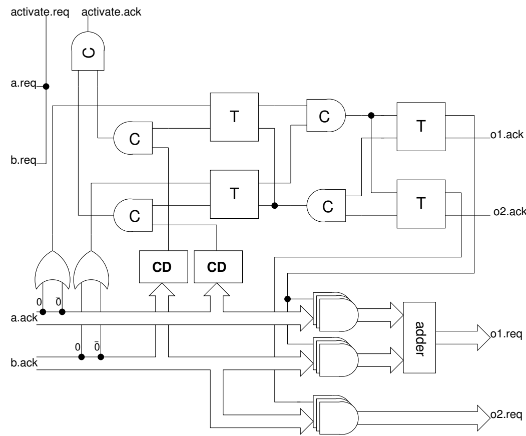

Figure 9. Balsa dual-rail processing circuit

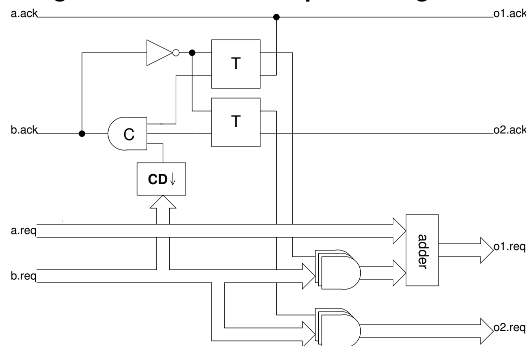

Figure 10. Data-driven dual-rail processing circuit

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 figure 8.

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? Figures 9 and 10 show the control for two dual-rail implementations. It is clear that the data-driven circuit is both smaller and faster (the same is true, but to a lesser extent, for a bundled data implementation). Note how the AND gates are opened early (quite probably before the arrival of data) allowing the data to proceed directly through the datapath logic (the adder in this case). No synchronisation is required between the inputs before they can be processed through the datapath logic and furthermore, the remaining significant control path dealing with the return-to-zero on the inputs has been substantially reduced.

6. 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 the language reflects the data-flow style of the circuits and is less flexible and less familiar than the sequential programming language style possible in Balsa. Note however (as described in section 3.4) that in order to achieve anything close to reasonable performance, a Balsa programmer must abandon sequential style, in favour of a data-flow type description.

This work originally grew from a desire to automate transformations to existing Balsa handshake circuits to produce more efficient structures along the lines of existing data-flow style compilation strategies [21, 20, 19]. This work was not continued for two reasons: it was not clear what the result of such optimisation 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) [21] rely on pipelining sequential programs and produce modified circuit behaviour. A Balsa designer may depend on the design behaving in the manner it was written which could easily not be the case after optimisation; indeed if a DDD-type strategy were to have been applied to the SPA processor the memory interface would have broken. The data-driven style addresses what an optimised circuit might look like, the second issue is an important area for future work. The data-driven language could provide a target for an optimisation strategy and a means for experienced designers to manually tweak the results or add their own data-driven style modules if desired.

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.

6.1. 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 figure 11.

It can be seen from this small example that much of the language is very similar to conventional Balsa (c.f. figure 1). The declaration of the procedure and the input and output ports is identical. Unlike conventional Balsa, the procedure input ports will always be passive

```
-- 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
```

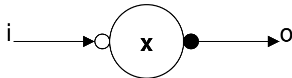

Figure 11. Data-driven one-place buffer & handshake circuit

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 block. The only synchronisation 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.

Incidentally, compared with the circuit of figure 1, the handshake circuit for this buffer is simply a *VariablePush* component.

6.2. 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 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 synchronise 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.

6.3. Input control

The semantics of the input are similar to the eager inputs described earlier in that the control is activated early. However, in the eager semantics, it was still necessary for the control and data to synchronise to release the data once all required reads had been completed on the channel. As reads are now to be pushed, this synchronisation 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).

6.4. 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.

6.5. 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 in figure 12 demonstrates the full flexibility offered by Balsa for using arrayed variables. Firstly, 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 the *Variables* have been converted to *VariablePush* components, it is necessary to write to each *VariablePush* before it is read.

```
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
```

Figure 12. Balsa read and write of arrayed variables

After writing to a single element in the array, only that element would be available to read.

An 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 run-time indexing. Alternatively, 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.

Neither 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 one time.

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.

7. A 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 [16], the first large scale design described in Balsa.

The nanoSpa has been gradually developed with the sole objective of making a Balsa synthesised 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.

7.1. Objectives of this example

- To demonstrate that the data-driven synthesis flow can be used to construct a significant design.
- To compare the performance of a high performance Balsa design with the closest possible equivalent in the data-driven style.
- To demonstrate the integration into the existing Balsa design-flow and the use of mixed Balsa and data-driven designs.
- To attempt some level of qualitative comparison between the features and flexibility offered to the designer in both description styles. In particular, it is believed, that this example demonstrates that the data-driven description differs very little from the style of Balsa code that an experienced Balsa developer would write. Indeed, converting the Balsa nanoSpa into a data-driven description provided very few challenges.

7.2. 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 synthesised datapath logic is the same as the Balsa nanoSpa, and the control contains most of the significant differences. 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

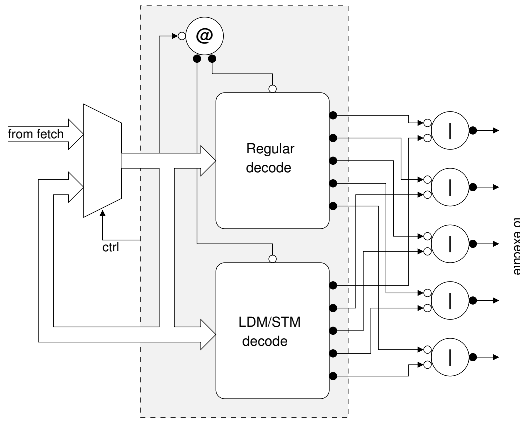

Figure 13. Data-driven nanoSpa decode structure

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.

7.3. 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. Attempting to replicate this structure presents some difficulty in the data-driven style. 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.

An iterative structure of this nature is not available in the data-driven style. However, it is quite straightforward to rearrange the structure of the decode to implement the multi-cycle instructions as shown in figure 13. In this structure the *ldm/stm* decode is no longer itself iterative. Instead 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 multiplexor to the decode logic. If the instruction is an *ldm/stm*, the necessary data for the next iteration is passed back to the multiplexor and the control signal is set so as to re-inject 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 multiplexor is signalled to inject the next instruction being sent from

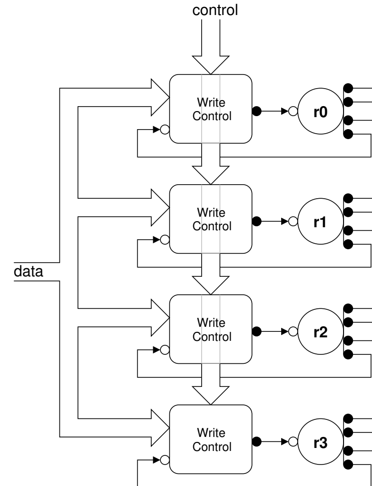

Figure 14. Data-driven nanoSpa register structure

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 figure 13 (for regular or *ldm/stm* instructions) can be copied directly to the data-driven description.

7.4. 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 when using push style variables it is not so easy to provide this general structure. 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 figure 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 phase 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 (figure 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.

7.5. Simulation Results

The results of simulations of the control-driven and data-driven nanoSpa processors are presented here. This paper only gives a superficial summary of the overall performance so as to justify the approach taken; a more detailed analysis, with deeper insight, is available [18].

All simulations are performed at gate-level using fixed gate delays. This does not provide a highly accurate estimate of absolute performance although experience has shown that the results of these simulations closely approximate transistor-level simulations in a 180nm technology. As a relative measure for the comparison of the two styles this level of simulation is more than sufficient. The control-driven nanoSpa has previously been simulated at transistor-level and both bundled data and dual-rail implementations achieve approximately 55 Dhrystone MIPS. It can be seen from the results in the next sections that the gate-level simulations slightly under-estimate the transistor level performance.

Precise transistor-level simulations are not possible for the data-driven design because the example cell library used has been designed locally and only contains transistor level models for the precise cells needed to implement original Balsa components. Some data-driven components use cells that are not currently provided and these would need to be added to the cell library. It is not believed that these results would yield any greater insight apart from giving a more accurate absolute performance estimate of the processor.

Dual-rail implementation. The dual-rail control-driven nanoSpa achieves 50 Dhrystone MIPS. The data-driven version achieves 79 Dhrystone MIPS, an improvement of 1.6 times the original. As expected the area is significantly increased, from 315373 to 938669 transistors. As anticipated, a significant proportion of this increase is found in the register bank (from 67036 to 370368 transistors). If the increase in register bank area is ignored, then the data-driven nanoSpa is roughly twice the size of the original Balsa version. The area overhead for dual-rail is particularly large. This is mainly due to the large size of the *VariablePush* component and the increased number of completion detectors. Note that no attempt has been made to optimise the back-end component implementations for area so there is future potential for reducing the area overhead although given the magnitude of the performance gains, the area overhead is not excessive.

Bundled data implementation. The bundled data control-driven nanoSpa achieves 52 Dhrystone MIPS. The data-driven version achieves 81 Dhrystone MIPS, an improvement of 1.5 times the original. The improvements in throughput of the individual modules are fairly similar to those shown in the dual-rail implementation. The difference in area for the bundled data implementation is much smaller than that for dual-rail (form 147561 to 223383 transistors). 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. When compared with the magnitude of the performance improvement, this area overhead can be considered as low.

7.6. 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 non-critical modules, while the performance of the data-driven style is exploited for others.

The new hybrid design achieves 62 MIPS in dual-rail and uses 637119 transistors. In bundled data the performance is 67 MIPS and the size is 175635 transistors. Performance has been traded for reductions in area and energy consumption.

8. Conclusions

This paper has described contributions in the field of asynchronous digital circuit synthesis. The existing Balsa synthesis method has been examined and performance has been identified 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 optimisation.

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. The potential performance improvements over the control-driven style have been convincingly demonstrated by comparison of this design with the equivalent control-driven implementation.

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 implementation of register banks.

A drawback of the data-driven style is that the descriptions are less flexible and not as familiar to a general hardware designer as those that are possible in Balsa.

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 clearly 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. This is a very useful contribution but is by no means the end of the story. In general, the performance of synthesised 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. *Implementing Balsa Handshake Circuits*. PhD thesis, University of Manchester, 2000.

[2] T. Chelcea, S. Nowick, A. Bardsley, and D. Edwards. A burst-mode oriented back-end for the balsa synthesis system. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 330. IEEE Computer Society, 2002.

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

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, Mar. 1997.

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

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

[7] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, Apr. 1997.

[8] http://www.handshakesolutions.com/Technology/Haste/.

[9] T. Kolks, S. Vercauteren, and B. Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Asynchronous Circuits and Systems*, Mar. 1996.

[10] A. Lines. *Pipelined Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1995.

[11] J. Liu. *Arithmetic and control components for an asynchronous microprocessor*. PhD thesis, University of Manchester, 1997.

[12] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Pénzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[13] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[14] L. Plana, D. Edwards, S. Taylor, L. Tarazona, and A. Bardsley. Performance-driven syntax-directed synthesis of asynchronous processors. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES)*, Sept. 2007.

[15] L. A. Plana and S. M. Nowick. Architectural optimization for low-power nonpipelined asynchronous systems. *IEEE Transactions on VLSI Systems*, 6(1):56–65, Mar. 1998.

[16] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, S. Temple, J. D. Garside, and Z. C. Yu. SPA— a secure Amulet core for smartcard applications. *Microprocessors and Microsystems*, 27(9):431–446, Oct. 2003.

[17] L. A. Plana, S. Taylor, and D. Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *Proc. International Conf. Computer Design (ICCD)*, pages 703–710. IEEE Computer Society Press, Oct. 2005.

[18] S. M. Taylor. *Data-Driven Handshake Circuit Synthesis*. PhD thesis, University of Manchester, 2007.

[19] J. Teifel and R. Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 17–27. IEEE Computer Society Press, Apr. 2004.

[20] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *IEEE 13th International Workshop on Logic Synthesis (IWLS)*, Temecula, CA, June 2004.

[21] C. G. Wong and A. J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *Proc. ACM/IEEE Design Automation Conference*, pages 508–513, June 2003.