Teak: A Token-Flow Implementation for the Balsa Language

Andrew Bardsley Luis Tarazona Doug Edwards
School of Computer Science, The University of Manchester,
Oxford Road, Manchester, M13 9PL, UK
bardsley@cs.man.ac.uk tarazonl@cs.man.ac.uk doug@cs.man.ac.uk

***Abstract*—This paper describes a new target component set and synthesis scheme for the Balsa asynchronous hardware description language. This new scheme removes the reliance on precise handshake interleaving and enclosure by separating out control 'go' and 'done' signalling into separate channels rather than using different phases of the asynchronous handshake. This leads to circuits in which optimisation and control overhead mitigation can be carried out by merging/separating control and data channels and by introducing handshake-decoupling latches. This work aims to make Balsa descriptions implementable in the more widely used and understood higher performance token-based asynchronous circuit styles.**

Keywords–asynchronous logic; logic synthesis

I. Introduction

The Balsa synthesis system[2] produces Handshake Component[12] implementation of circuits described in the Balsa language. The components produced are substantially similar to those produced by the Tangram system, the precursor of Handshake Solutions' TiDE system[8]. Balsa has been used to implement two substantial fabricated designs: the DMA controller for the Amulet3h asynchronous processor subsystem of the DRACO chip and the SPA ARM-compatible processor (and peripherals) for the Buxton smartcard chip. Balsa was developed to allow optimisation opportunities in Handshake Circuit designs to be explored. In particular, the *FalseVariable* component and input-enclosure language construct[1] have allowed pipelined descriptions with alternating latch and combinatorial handshake processing stages to be more naturally described.

The work described in this paper extends the degree to which the Balsa language can sympathetically be used to describe pipelined systems by proposing a new set of components, synthesis rules and Balsa language synthesiser collectively called the Teak system. The aim is to provide a path for future performance increases in Balsa synthesis by exploiting high performance pipelined asynchronous circuit styles.

II. Handshake Components and Circuits

Handshake Circuits[12] (HCs) are compositions of Handshake Components connected exclusively by 'channels'. Channels carry data from one component to another under the control of 'handshakes'. In implementation, channels often take the form of a 'request' wire, flowing from the communication-initiating component to the target component, and an 'acknowledgement' wire, flowing back from the target to initiator, and accompanying data-carrying wires. Data transfer takes place by the exchange of events on those two wires between initiator and target. Many choices of data encoding schemes and handshake signalling protocols are possible when devising gate-level implementations for channels and components. In the evaluations given at the end of this paper, 'dual-rail' encoding is used. In dual-rail, data is encoded onto pairs of wires (signalling data 0 or 1 for each data bit) which form one of either the 'request' or 'acknowledgement' portion of the channel. Dual-rail encoding allows circuits to be built in which data transfer is delay-insensitive[10].

A. Handshake Components and synthesis

In HC synthesis, templates for Handshake Components are selected from a small predefined set designed to map directly onto source-language constructs such as control sequencing (in the Balsa system: the *Sequence* component), or variables (the *Variable* component). Each template Handshake Component has a parameterised gate-level implementation allowing gate-level netlists to be rapidly generated by applying the parameters (for properties such as data width or fan-out/in) for each component instance and composing the resulting gate-level netlists. Control in HCs is handled by channels consisting (in most signalling protocols) of only 'request' and 'acknowledgement' wires. Handshakes on these data-less channels 'encloses' data-processing activity. This is explained further in the below Balsa example.

HC-based designs have been shown to have good energy usage characteristics[9] but poor performance[16]. Both of these characteristics can be attributed to the control-heavy nature of HCs. Explicit control channels mediate much of the data transfers between parts of a circuit. A number of methods have been used to mitigate this cost. These include reimplementing control circuits using Petri nets and Burst-Mode Machines[5], the replacement of critical components with more 'permissive' versions[16] and the wholesale replacement of the language, synthesis method and many of the components to produce a more pipeline-like implementation style[17].

1550-4808/09 $25.00 © 2009 IEEE
DOI 10.1109/ACSD.2009.15

III. A BALSA SYNTHESIS EXAMPLE

Balsa is a CSP[3] derived hardware description language with explicit constructs for language command/statement-level parallelism. Data can be transferred between parallel processes or commands using 'channels' which synchronise sender and receiver. These channels map naturally onto the HC-level channels which are used to implement them.

This is a simple one-place buffer for one bit of data written in Balsa (comments look like `-- comment`):

```
procedure buffer (
  input i  : bit;
  output o : bit
) is
   variable v : bit -- storage
begin
   loop       -- repeat forever
      i -> v -- chan. input to var.
      ;      -- explicit sequencing
      o <- v -- chan. output
   end
end
```

The body of `buffer` has two commands: a channel input to a variable, and a channel output from the same variable. These commands are explicitly sequenced using '`;`'. The **`loop`** … **`end`** construct repeats this behaviour indefinitely. Fig. 1 shows a HC implementation for this description (as generated by the 'balsa-c' compiler from the Balsa system). Circuit activity is started by beginning a handshake on the 'activate' channel. The **`loop`** is implemented by the *Loop* component connected to activate, and the sequencing by the *Sequence* component below that. The *Sequence*, when activated by the 'request' on the channel connected to the port with the open bubble, performs a sequential handshake on each of its output ports before completing the activation handshake (with an 'acknowledgement'). In this example, the left hand output initiates the first such handshake. In this way, the activation handshake encloses the action of the component, and so the activity of components connected downstream of it. The *Sequence* outputs are each connected to a *Fetch* component. *Fetch* uses its activation handshake to mediate the transfer of data from one data port (on its left) to the other (on its right). These data transfers link the input channel, variable and output channel to form the `buffer`'s data transferring behaviour.

IV. THE TEAK SYSTEM

Teak replaces the data-less activation channel (used to enclose the behaviour of program fragments in HC synthesis) with separate 'go' and 'done' channels. Control/datapath interactions using components which exploit signal-level event interleaving are replaced by the rendezvous/forking of control and data channels with local handshaking to complete control interactions. This separation of 'go' and 'done' makes Teak much more like the Macromodules system [15] than Handshake Circuits, albeit with more flexibility in the elimination of control channels through merging with data channels.

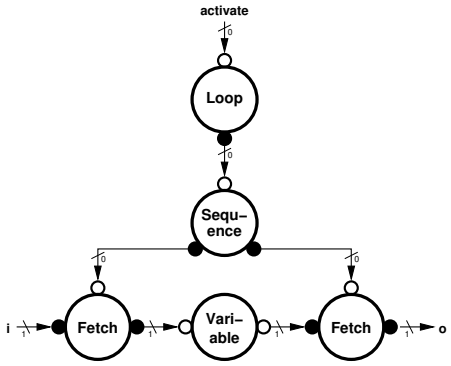

Figure 1. Handshake Circuit of buffer

Explicit buffering is used to decouple one component from another and to introduce the desired degree of token storage to enable the circuit to function and, looking beyond the work in this paper, to allow more transforming synthesis methods to increase circuit parallelism.

Treating control channels in this way allows all the optimisation techniques usable with pipelined asynchronous systems (i.e. those with input-enclosing-output processing stages and decoupling buffering stages) to be used on Teak circuits whilst still allowing local sequenced behaviour by using control channels.

V. TEAK COMPONENTS

There are currently eight Teak components (as shown in Fig. 2):

Steer (S) – conditional steer of input to exactly one output. Parameterised with disjoint match conditions for each output and bit ranges to carry to outputs. With 0 bits carried to outputs, *Steer* works like the Balsa *Case* component.

Fork (F) – unconditional *n*-way fork. *Fork* can be parameterised by which (if any) bits of the input are carried to each output. A two-way *Fork* of *n* and 0 bits can be used to generate a control token from moving data.

Merge (M) – input on one of the input ports is multiplexed towards the output. Inputs must be mutually exclusive. In some configurations, *Merge* may have to cope with second input arrival during first input activity.

Arbiter (A) – merge with arbitration between inputs.

Join (J) – unconditional *n*-way join. Concatenates data bits of arriving inputs.

Variable (V) – persistent storage. Separate write and read sections allow arbitrary control ordering/conditionality of reads. *Variables* allow complicated control activity without incurring the cost of always moving data along with control around a circuit. 'wg/wd' and 'rg/rd' (go/done) pairs make all writes data initiated and control token completed, all reads control token initiated and data delivery terminated.

Operator (O) – any and all data transforming operations. Inputs are formed into a single word. Internally an *Operator* is organised into interconnected terms allowing Operators to be amalgamated or separated to allow cheaper implementation or *Buffer* insertion.

Buffer (B) – data storage and channel handshake decoupling.

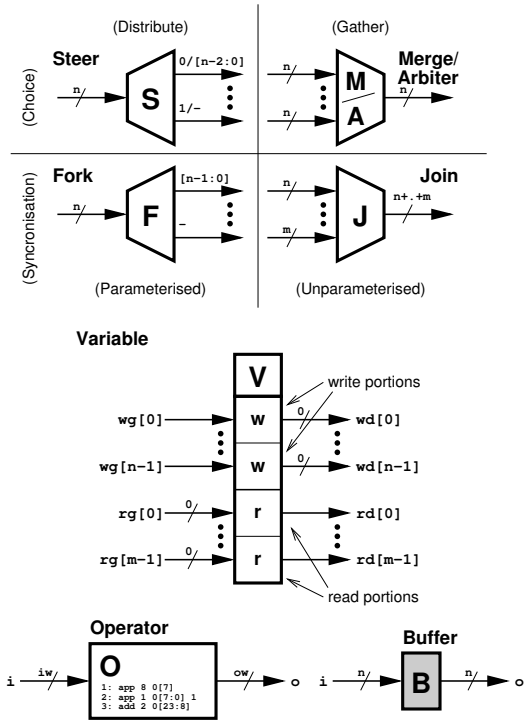

Figure 2. Teak components

All of the components, except *Buffer*, can be implemented with any chosen degree of input to output channel coupling (i.e. concurrency of handshaking events). *Buffer* must provide at least some decoupling so that it can be used to separate pipeline tokens. In this way, Teak components resemble the components of other elastic token pipeline systems.

The *Variable* is included in this component set in order to allow sequential, storage-centric descriptions to be mapped directly into hardware. This is in contrast to other token flow approaches to asynchronous synthesis [4][11] which perform single assignment analysis on the input language to allow variables to be eliminated in favour of pipeline buffers. This decision was made to allow the exploration of the possible power and area implications of retaining 'fixed' variables. Also, pipeline buffer-only approaches find it difficult to handle descriptions of persistent register banks without messy 'register refreshing' loops.

Fig. 3 shows the `buffer` example from Section III constructed from Teak components using synthesis rules from Section VI. Notice that the *Loop* component has become a loop comprised of a *Merge* (to introduce the 'go' token), a *Join* (to meet incoming data), and a *Fork* (to return a token back around the loop, through the *Merge,* after the output command) rather a composition of enclosing control components.

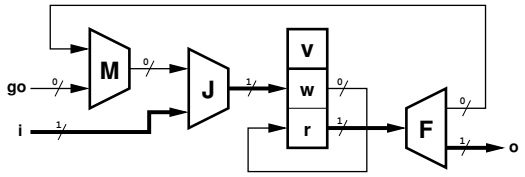

Figure 3. Teak circuit of buffer

VI. Teak synthesis

Teak synthesis is initially syntax directed. Optimisations can then be performed on the generated Teak component netlists (Teak circuits). Each command in a Balsa description is mapped into components with dangling 'go' and 'done' control channels (a few commands never terminate and have no 'done'). Expressions, channel accesses and assignment left-hand sides similarly have a pair of dangling channels: one bearing data and the other a control initiating/completion channel. Control can be sequenced by joining commands 'done' to 'go' in a chain. Data and control usually meet with *Fork* and *Join* components.

As with Balsa intermediate (Breeze) netlists, there are many possible choices of data encoding and signalling protocols on the channels between components. As Teak deals in the flow of tokens rather than enclosing handshakes, Teak component implementations also have choices of the degree of interleaving between input to output handshakes, the use of weak-condition behaviour and storage within components.

A. Channels

Channels in Balsa have no capacity. Inputs and outputs on a channel form a synchronisation where either party can delay the transaction until both are ready for data to be transferred. In Balsa, the **`select`** command (which allows choice based on order of arrival of data on a number of channels) and the 'enclosed' channel input command can be used to exploit the non-atomic nature of asynchronous channel construction to allow latch-less implementations of data processing stages to be described. In such stages, data processing and outgoing channel outputs are enclosed within the input handshake. Alternating such stages with latch-containing pipelining stages allows push pipeline-like structures to be built.

Fig. 4 shows a single output, single input channel implemented using Teak components, with a Balsa-style channel. The pair of data and acknowledging channels between output and input commands form a synchronisation and limit to a single token the capacity of the loop formed from output command (as data), through input command and back to the output command (as an acknowledging 'done'). Note the use of *Forks* and *Joins* between data and control.

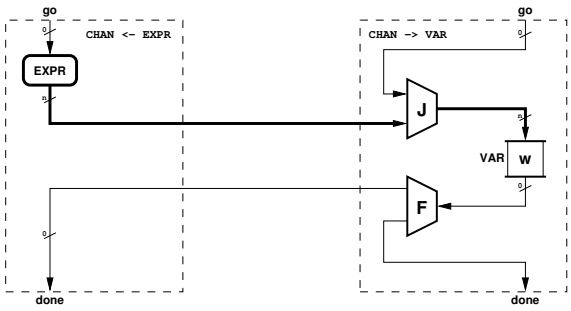

Figure 4. Balsa-style channel implementation

Unfortunately, Balsa's channel implementation doesn't allow the capacity of buffered Teak channels to be exploited. Instead we have chosen to change the semantics of Balsa channel to make writes 'fire and forget'. Channel outputs and inputs are no longer synchronised and enclosure inside a sending handshake can no longer be relied upon. In practice, this reduces the utility of the **`select`** language construct but also allows descriptions to be formed which exploit (or possibly rely upon) non-zero channel capacity. This introduces an incompatibility with the Balsa system's interpretation of Balsa descriptions.

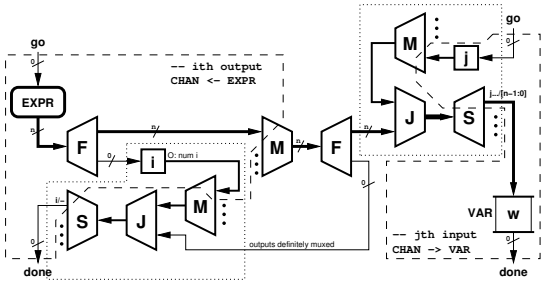

Figure 5. Multiple-output channel implementation

Fig. 5 shows how channel read and write commands are combined to form a complete Teak style language-level channel. The i and j constant-valued *Operator* components 'tag' the request channels from different input/output commands so that once those requests are merged, with the following *Merge* component, the source of the request is encoded on the *Merge* output. This common request is then *Join*ed to a token *Fork*ed from the outgoing data *Merge* (or, for inputs, the incoming data itself) and *Steer*ed to provide the local command acknowledgements.

The combination of tagging *Operators,* the following *Merge* and the *Join*/*Steer* combination (the two dotted boxes in Fig. 5) plays a similar role to the Balsa *DecisionWait*[1] Handshake Component. This involves steering an incoming token (in this case the acknowledgement from the data-bearing merge) to the correct output based on the arrival of a single token on one of a group of input tokens (in this case, the choice of output command site). In Teak, we have chosen to separate out the component parts of the *DecisionWait,* rather than provide a single component, to allow for flexibility of *Buffer* insertion.

In cases where acknowledgement tokens need not be steered (e.g. where there is only one read or write to a channel in the description) much of the control/data interaction can be optimised away (as shown in Fig. 6). This implementation is similar to that of Fig. 4, but without the sequencing of variable write to the output command's 'done'.

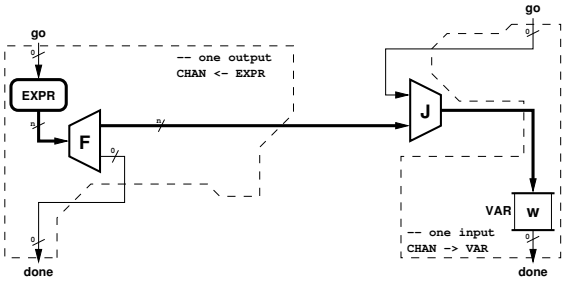

Figure 6. Channel component optimisation

B. Commands

Fig. 7 shows sequential and parallel composition of commands. Command 'go' and 'done' channels can be connected in series to form sequencing, and so no explicit *Sequence* component is required. Parallel composition requires two components (*Fork* and *Join*) in contrast to Balsa's *Concur* component which contains both functions in one component. Fig. 7's presentation of command composition is very similar to that used in non-return-to-zero (2-phase) signalled handshaking. This is illustrated well by Brunvand[6]. Note, however, that here we are using handshake **channels** rather than individual wire signals for each of 'go' and 'done'. On a channel, the token recipient can stall a handshake (by denying an acknowledgement) and so the token capacity of a string of commands is not necessarily limited to one (i.e. the strict alternation, from the ends of the string, of 'go', 'done', 'go', 'done' …). Where resources are not shared between sequentially composed commands, this property allows pipelining to naturally arise.

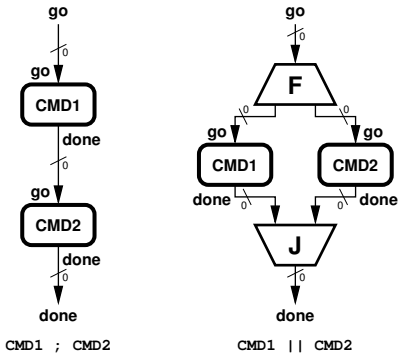

Figure 7. Sequential/parallel composition

Fig. 8 shows the structure of a `while` loop. The *Steer* component provides the control choice at the top of the loop. Without the *Steer*, a similar control loop with an incoming

Merge can be used to implement a non-terminating loop with the 'go' token providing initialisation. Note that the loop formed by the *Merge* and *Steer* components must have at least some buffering to prevent deadlock. Insertion of *Buffers* will, obviously, affect circuit performance. The examples examined in Section VIII have had *Buffers* manually inserted as this process is not yet automated.

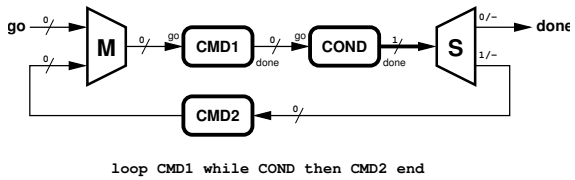

Figure 8. While loop implementation

C. Expressions

Expressions are compiled from variable reads by adding pairs of 'rg' (read go) and 'rd' (read done) ports on variable components, and *Operator* components to process data. Reading from channels within expressions (when within **select** commands or enclosed input commands, e.g. `chan -> then var := chan + 1 end`) is achieved by inserting *Variable* components to capture channel read data, and then using read port pairs on those variables to use that data. These variables can then often be removed if data is unconditionally used within the body command.

VII. Optimisation strategies

This section introduces some of the possible optimisations that can be performed over Teak-compiled circuits by taking advantage of some properties of either individual components or groups of them. Optimisations will be presented using simplified practical descriptions extracted from the design examples used in section VIII.

A. Variables

As stated in section V, in cases where channel reads are performed unconditionally after a write, the *Variable* can be removed (for single-read channels) or replaced by a cheaper and faster *Fork* component (for multiple-read channels), provided they are not used to enforce sequencing. Fig. 9 shows a single-write, unconditional single-read after write channel structure before and after optimisation.

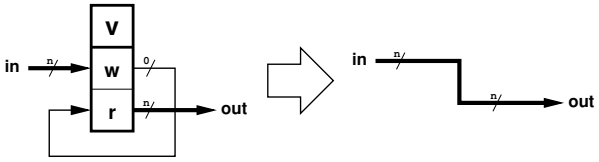

Figure 9. Variable read-after-write optimisation.

As an example, consider the following description whose implementation will contain *Variables* implementing the inputs on `a`, `b` and `c` as described in Section VI.C. This is a description of a full adder whose output is separated into sum and carry-out portions:

```
procedure adder (
  input a, b   : N bits;
  output sum   : N bits;
  output carry : bit
) is
   channel cs : N+1 bits
begin
   loop
      a, b -> then
         cs <- (a + b as N+1 bits)
      end ||
      cs -> then
         sum <- (#cs[0..N-1] as
            N bits) ||
         carry <- (#cs[N] as bit)
      end
   end
end
```

For simplicity, let's consider only the part of the circuit that provides the `sum` and `carry` outputs as shown in Fig. 10(a). The *Variable* that implements the channel `cs` has a single write port and two read ports for `sum` and `carry`. Reads are initiated as soon as the `cs` *Variable* 'wd' (write done) port indicates that new data has been stored. The *Fork* component at the top provides tokens for both read ports. As a write operation is directly followed by a read, the *Variable* can be substituted by a *Fork* that provides 'sum', 'carry' and 'done' results as shown in Fig. 10(d). In dual-rail circuits, an additional benefit of this type of optimisation is that the forked channels would only need to wait for the arrival of those input bits that will be carried to the output. This optimisation can be viewed as a 3-step process:

(i) The *Fork* labelled 1 is displaced 'downstream' in the datapath, after the *Variable* `cs`, leaving a single write, single read *Variable,* as shown in Fig. 10(b).

(ii) *Variable* `cs` can be removed as a write is directly followed by a read and the three *Forks* can be merged into a four-way *Fork*, leading to the circuit in Fig. 10(c).

(iii) Now, the *Join* component in Fig. 10(c) is redundant because both inputs come from the same fork. The inputs of the *Join* can be merged and the final circuit is shown in Fig. 10(d).

Similar kinds of optimisations based in component displacement will be presented in the following sections.

B. Fork displacement

In some circumstances, *Fork* components can also be displaced 'upstream' in a data or control path to allow for more concurrent operation. Consider the following segment of code where the results generated by the two output commands must be written sequentially to a common channel `out`:

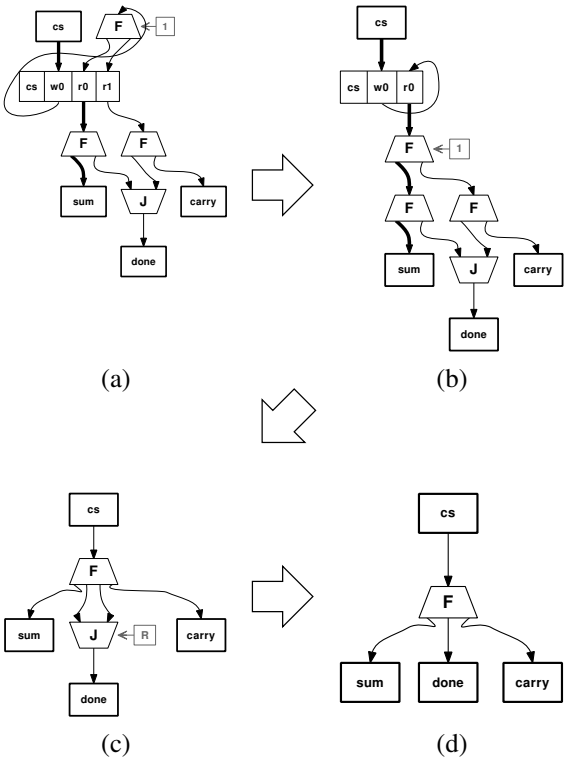

Figure 10. Variable substitution example

```
procedure tenFifteen (
   output out : 4 bits
) is
begin
   loop
      out <- 10; -- exprA
      out <- 15  -- exprB
   end
end
```

The resulting circuit is shown in Fig. 11. Note how the *Forks* labelled U fork the result of exprA and exprB to generate the output and the 'tag' constants (cA and cB). As explained in Section VI.A, those constants indicate which of the expressions will be output in the next iteration. If those *Forks* are moved upwards through the expression generators, as in Fig. 12, the constant that steers the control for the next iteration will be generated concurrently with the output. This kind of displacement can be done through any data transforming operation or even single-input command blocks.

C. Join displacement

If circuit conditions allow it, *Join* components can also be moved upstream resulting in faster operation and smaller circuits.

Consider the following segment of code which is a simplified version of a 'sign adjust' unit for the multiplicand input of the Booth multiplier in the nanoSpa processor [13][14]. The circuit takes an N bit input word `b` and, depending of the type of multiplication specified by the `mType` input, either appends M zeroes after the most significant bit of input `b` or sign-extends it to $N + M$ bits to generate the adjusted output `ba`. For clarity, let $N = 8$ and $M = 3$ in this example:

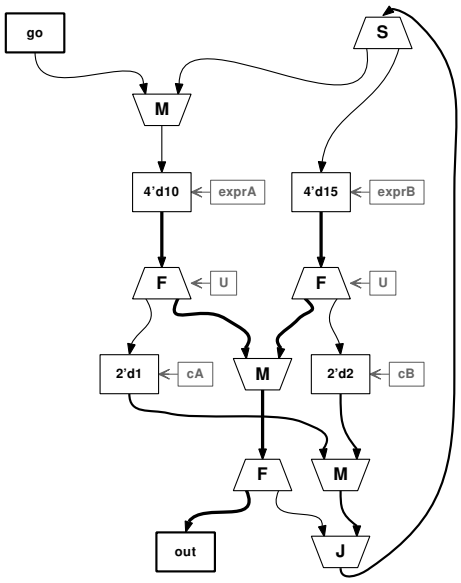

Figure 11. Sequenced channel write example.

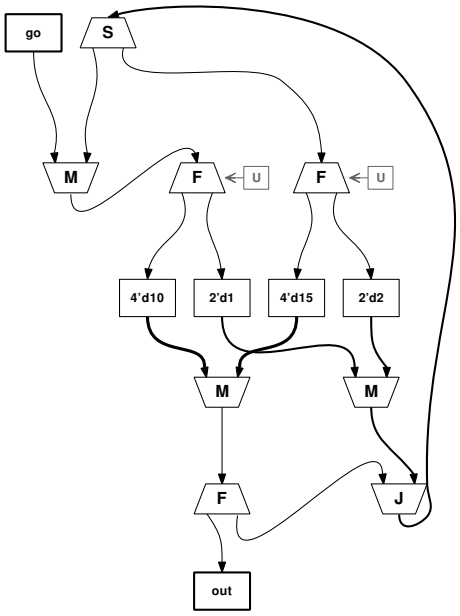

Figure 12. Fig. 11 after Fork displacement.

```
loop
   mType, b -> then
      case mType of MUL,UMULL,UMLAL
      then -- unsigned, pad with 0s
         ba <- zeroExtend (8, 11, b)
      else -- signed, sign-extend
         ba <- signExtend (8, 11, b)
      end
   end
end
```

The resulting circuit is shown in Fig. 13. In this figure, the dotted blocks on top, labelled inB and inM, contain the implementation of the two input channels reads and writes. The optimisation for blocks inB and inM works as follows:

(i) The *Join* labelled j0 connected to *Variable* `mType` is moved upstream through variables `b` and `mType` transforming them into a single-write, single-read variable `b_mType` whose width is now `(sizeof mType) + 8` bits. The size of the inputs and outputs of the *Steer* within inB are resized accordingly. The modified circuit for inB and inM blocks is labelled inMB in Fig. 14.

(ii) The previous transformation has exposed new optimisation opportunities: *Variable* `b_mType` can be removed (as explained in Section VII.A) and the three *Joins* can be merged into one, which in turn makes the *Fork* at the top of the circuit redundant. The resulting circuit is shown in Fig. 15.

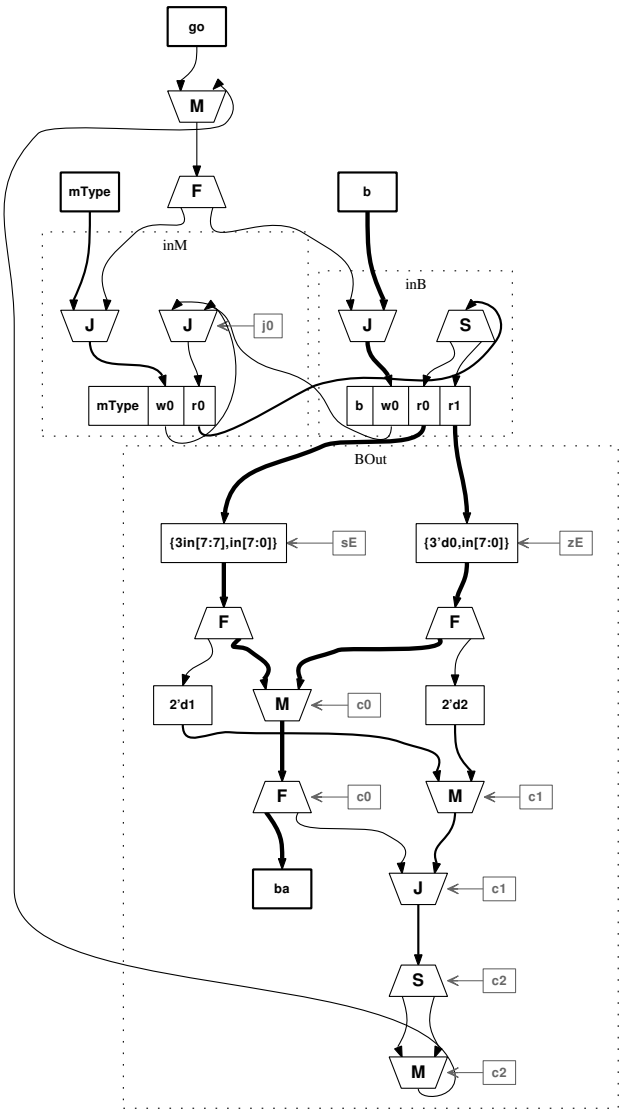

Figure 13. Unoptimised sign adjust circuit.

D. Fork-Merge-Join and Steer-Merge

Another target for optimisation are *Steer-Merge* and *Fork-Merge-Join* compositions. In Teak circuits, *Forks* are used in a datapath to generate a control token from a data token, to either synchronise or sequence operations. If the token sources of a *Merge* are derived from the control branches of a set of data *Forks* whose data channels are merged at some point ahead and then *Join*ed with the control-derived output of the *Merge,* this *Fork-Merge-Join* composition can be simplified and the final control token can be derived from the merged data.

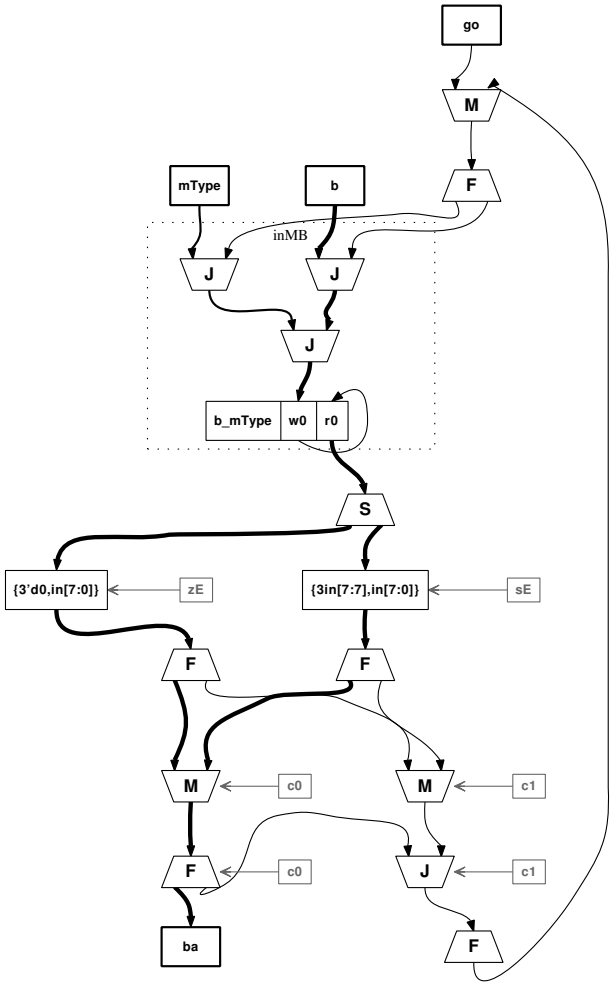

Figure 14. First optimisation steps of Fig. 13.

This *Fork-Merge-Join* optimisation can be seen more clearly referring to the bottom part of Fig. 13: The output data from Operators zE (zeroExtend) and sE (signExtend) are forked to produce control (thin lines) and data (wide lines) tokens. The data tokens are merged and then forked to produce the output `ba` an a new control token (*Merge* and *Fork* labelled c0). The control tokens from the top *Forks* in block BOut generate tag values (constant Operators 2'd1 and 2d'2) required to steer the control to the correct source in the next iteration (components labelled c1). As both data tokens are derived from a common source, the outputs of

the *Steer* are the only inputs to the *Merge* that generates the control token for the next iteration of the loop (components labelled c2). The simplification steps are as follows:

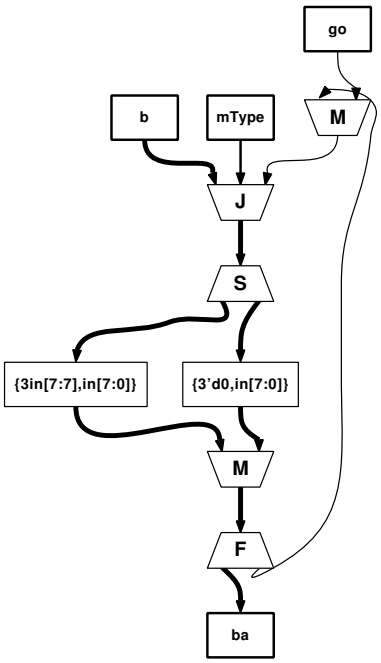

Figure 15. Optimised version of Fig. 13.

(i) The bottom *Steer-Merge* can be simplified into a single-output *Fork* that acts as an adapter that generates a control token from a data token as shown in Fig. 14.

(ii) As the generation of the control token in the new *Fork* is independent of the data value, the data channels that carry the constants can be simplified into control channels, making the constant blocks redundant. These are simplified in Fig. 14.

(iii) Now the control tokens from the *Forks* are redundant because each one will always synchronise with its sibling data token at the *Join* c1, hence those *Forks* and the *Merge* and *Join* with labels c1 can be reduced. The new *single-input* Fork inserted in step (i) can also be removed. The final circuit is shown in Fig. 15.

VIII. Design examples and results

This section presents the experimental results of a number of non-trivial, medium-complexity designs and compares Teak-style versus Balsa-style implementations. The main purpose of these experiments was to have an initial estimation of the performance of Teak circuits. For each design, the same Balsa language description was used to generate both Balsa and Teak circuits, targeting a dual-rail data-encoding implementation and mapped to an example gate library. The results shown here are for fixed-gate delay, gate-level simulation. For the Balsa circuits, all performance-driven optimisations described in [13] were applied. Teak circuits are produced by compiling Balsa language descriptions with the 'teak' compiler, producing component netlists which are then expanded into gate-level netlists using 'balsa-netlist' from the Balsa system. Component descriptions are written in the Balsa system's ABS language. A simple *Buffer* insertion strategy is applied to the component netlists that ensures no deadlock will occur was used. At the time of writing the optimisations presented in Section VII, and the *Buffer* insertion, have not been automated. These were applied by hand only to critical sections of the resulting network of Teak components.

TABLE I. Experimental results

Design	Balsa (ns)	Teak (ns)	Teak overhead
Mod-100 SC	19.52	23.11	18.39%
VDec	20.36	21.29	4.60%
nMult32	111.66	131.44	17.71%

The designs used as examples used are:

Mod-100 SC – A modulo-100 systolic counter in the style described in [12]. For this design we compared the average tick period of the counters.

VDec – A Viterbi decoder with architecture similar to that described in chapter 14 of [10] and [7] with the following parameters: code rate = $^1/_2$, constraint length $k = 3$ (4 states), 3 bit soft-decision decoding and 16 time slots of backtracking memory. The parameter used to compare the designs was the data output cycle time.

nMult32 – The 32×32 multiplier with 32 bit accumulate implemented using a radix-8 Booth algorithm [14] used in the nanoSpa processor. The parameter used to compare the designs was the average cycle time of signed multiply-and-accumulate operations.

The experimental results are shown in Table I. The results show that a non-fully optimised, initial token flow implementation of Balsa can produce circuits with comparable performance (within 5% – 18% slower).

IX. Conclusions and future work

Implementations of Teak circuits currently have worse performances than those of Balsa circuits. This is unfortunate but not unexpected. The implementations for Teak components are at an early stage of development, and the analysis tool required to perform satisfactory *Buffer* insertion is not present. We do, however, see a lot of headroom in the Teak approach as its small, regular component set allows the freedom to merge and split data and control much more naturally than is the case in Handshake Components. This feature will allow a greater degree of freedom in optimising Teak circuits than is possible with Handshake Components.

There is still much work that can be done to improve the

optimisation of Teak-generated circuits. This includes: the creation of effective buffering schemes, the implementation of components with different data encodings (e.g. one-hot codes running up to *Steer* inputs) as well as extensions and automation of the optimisations described in Section VII.

X. Acknowledgements

The authors would like to thank Fabien Gavant (of Grenoble INP-ENSERG, FR) for the use of his Viterbi decoder Balsa description.

References

[1] A. Bardsley. Balsa: An Asynchronous Circuit Synthesis System. Master's thesis (1998), Department of Computer Science, The University of Manchester, UK.

[2] Balsa project homepage at The University of Manchester, UK. URL *http://intranet.cs.man.ac.uk/apt/projects/tools/balsa/*.

[3] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM* **21(8)**, Pages 666–677 (August 1978).

[4] C.G. Wong, A.J. Martin. Data-Driven Process Decomposition for Circuit Synthesis. In *ICECS 2003: Proceedings of the IEEE Conference of Electronic Circuits and Systems*, 2001.

[5] T. Chelcea, A. Bardsley, D. A. Edwards, S. M. Nowick. A Burst-Mode Oriented Back-End for the Balsa Synthesis System. In *Proceedings of DATE'02.*

[6] E. Brunvard. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

[7] F. Gavant. Asynchronous Viterbi decoder described in Balsa language (unpublished report), School of Computer Science, The University of Manchester, UK and Grenoble INP-ENSERG, FR, 2008.

[8] Handshake Solutions company website. URL *http://www.handshakesolutions.com.*

[9] H. van Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, G. Stegmann. An asynchronous low-power 80C51 microcontroller. In *4th International Symposium on Asynchronous Circuits and Systems, ASYNC'98*, March 1998.

[10] J. Sparsø, S. Furber (eds.). *Principles of asynchronous circuit design - A systems perspective.* Kluwer Academic Publishers, 2001.

[11] J. Teifel, R. Manohar. Static Tokens: Using Dataflow to Automate Concurrent Pipeline Synthesis. In *10th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC'04*, 2004.

[12] Kees van Berkel. *Handshake Circuits - An asynchronous architecture for VLSI programming*. Cambridge International Series on Parallel Computers 5. Cambridge University Press, 1993.

[13] L.A. Plana, D. Edwards, S. Taylor, L. Tarazona, A. Bardsley. Performance-Driven Syntax-Directed Synthesis of Asynchronous Processors. In *CASES'07: Compilers, Architecture, and Synthesis for Embedded Systems.* School of Computer Science, The University of Manchester, UK, October 2007.

[14] L.A. Tarazona, L.A. Plana, D.A. Edwards. Architectural enhancements for a synthesised self-timed processor. In *Proceedings of the 19th UK Asynchronous Forum.* School of Computer Science, The University of Manchester, UK, 2007.

[15] M. J. Stucki, S. M. Ornstein, W. A. Clark. Logical Design of Macromodules. In *AFIPS Spring Joint Computer Conference 1967*, pages 357–364, 1967.

[16] L.A. Plana, S. Taylor, D. Edwards. Attacking Control Overhead to Improve Synthesised Asynchronous Circuit Performance. In *Proceedings IEEE International Conference on Computer Design ICCD-2005*, pages Pages 703–710, October 2005.

[17] S. Taylor, D. Edwards, L.A. Plana. Automatic Compilation of Data-Driven Circuits. In *14th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC'08*. School of Computer Science, The University of Manchester, UK.