Attacking Control Overhead to Improve Synthesised Asynchronous Circuit Performance

Luis A. Plana Sam Taylor Doug Edwards
School of Computer Science, The University of Manchester
{lplana, smtaylor, dedwards}@cs.manchester.ac.uk

Abstract

The development of robust synthesis techniques and tools is important if asynchronous design is to gain more widespread acceptance. Handshake circuits are a method of constructing asynchronous circuits from a set of modular components connected by handshake channels. They offer a level of abstraction above a particular target technology or implementation style. The Balsa system employs the handshake circuit approach and has demonstrated that it can be used to rapidly generate large, robust circuits.

This speed and flexibility is currently achieved at the cost of performance. This paper examines the problem of control overhead in handshake circuits and proposes new handshake component specifications and implementations that significantly reduce this overhead.

These changes are incorporated into the Balsa synthesis system and are shown to produce a doubling of the performance of a 32-bit processor without making any changes to the original description.

1. Introduction

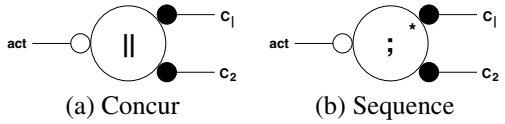
Balsa [2] is a synthesis system that generates purely asynchronous macromodular circuits. It was used in the development of the SPA processor [7]. SPA is a fully synthesised, 100% ARM-compatible processor core and was originally designed for a prototype smart card. Both QDI dual-rail and four-phase bundled data implementations were synthesised by the Balsa system from the same source language description. SPA is the largest and most complex design undertaken using the Balsa system to date and will be used as an example in this paper. For the smart card application, performance was not a significant requirement but nevertheless the performance of SPA was significantly lower than expected. There is no single reason for the poor performance of the SPA processor but a contributing factor is overheads due to the nature of the handshake circuit synthesis method. This paper addresses the handshake circuit overheads and proposes improved components that improve the performance of SPA significantly.

Originally proposed by Van Berkel [8] for use with the Tangram [8, 5] language (upon which Balsa is heavily based), handshake circuits offer an attractive paradigm for circuit synthesis. Complex descriptions written in the source language can be translated into a circuit implementation consisting of instances of components taken from a relatively small set of different handshake components that in isolation are straightforward to implement. The first stage of Balsa synthesis involves compiling descriptions written in the Balsa language into handshake circuits. The translation approach from language to handshake circuits is that of syntax-directed translation. This approach gives a 'transparent' compilation; there is a one-to-one mapping from language constructs to the handshake component network that implements it. This gives the designer flexibility at the language level to optimise the resulting circuit in terms of performance, area or power. Small changes at the language level result in predictable changes in the implementation.

A drawback of this approach is that of the overhead imposed on the circuits by the control-driven approach to translation. A handshake circuit can be considered as a large monolithic tree of control components that direct the movement of data through datapath components. The data and control are frequently synchronised and often the control is slower than the data reducing the performance of the circuit as the data flow is stalled while the control catches up. Previous work on improving the control overhead has concentrated on 'control re-synthesis' [1, 3]. The main focus of this paper is several replacement handshake components designed to reduce the latency of control structures and improve overall performance. These new components are complementary to the control re-synthesis techniques and both can be used with the re-synthesis being applied to the new components.

A handshake circuit is constructed from a small set of handshake components that are composed in a macromodular style. Each circuit consists of a network of instances of these components connected by channels. Each chan-

nel connects an *active* port on one component to a *passive* port on another. The sense of the port (active or passive) indicates the direction of the handshake. An active port initiates a handshake (sends the request) and the passive port acknowledges requests. Channels can carry data and this can flow in either the same direction as the handshake (a *push* channel) or in the opposite direction (a *pull* channel). Channels that carry no data are known as *sync* channels or frequently as *activation* channels as they are used to start the operation of many components when connected to an activation port.



(a) Concur (b) Sequence

Figure 1. Control Handshake Components.

Handshake components are commonly represented diagrammatically as labelled circles, where the label identifies the operation of the component. Ports are represented as smaller circles; closed circles are used for active ports and open circles for passive ports. Active ports are connected to passive ports by arcs that represent channels, with an arrow used to indicate the direction of data flow. Figure 1 gives an example of this notation for two common components, Concur and Sequence. These two components have a significant impact on the performance of handshake circuits. Optimisations to these components are discussed in sections 2.1 and 2.2.

2. Handshake Circuit Control

This section examines four different control structures commonly generated by Balsa. For each structure, the conventional approach is described followed by a description of the proposed improvement and the new component required to implement it.

The Balsa compiler has been modified to automatically sythesise circuits using these improved components where appropriate.

The basic commands supported by Balsa are operations such as reading and writing from channels, arithmetic and logic operations, and synchronise on a sync channel. Complex commands are formed by composing commands in parallel or sequentially. These are implemented using the Concur and Sequence components (figure 1). The components are parameterisable in respect of the number of composed commands.

2.1. Parallel Control

The operation of the Concur component is shown in figure 2. The component will wait for a request on the activation channel and will trigger the two commands (C_1 and C_2) concurrently. As shown in the STG, the component will go through the complete four-phase handshakes on the activations of both commands before acknowledging the activation channel.

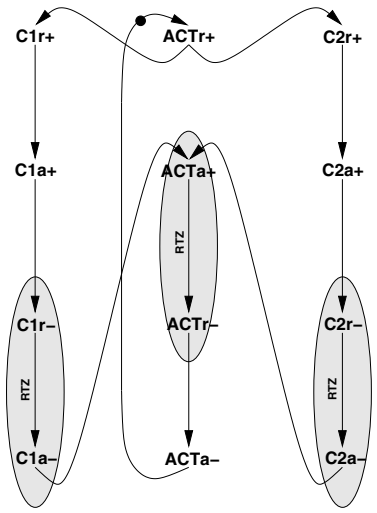


Figure 2. Conventional Concur Behaviour.

Figure 3 shows the conventional implementation of the Concur component. The component uses two S-elements to control the execution of the two commands. In the parameterised component, an S-element will be used for each command being controlled.

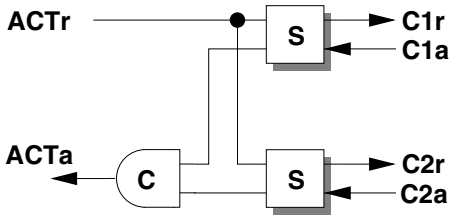
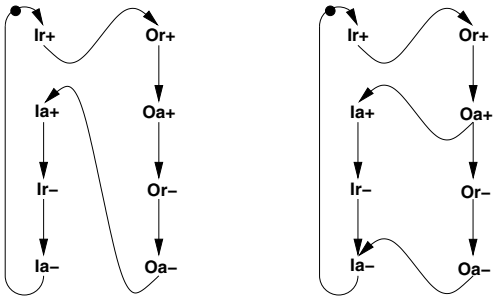


Figure 3. Original Concur Implementation.

The behaviour of the S-element is presented in figure 4(a). The figure shows that the output port will go through the complete four-phase handshake before the S-element acknowledges on the input channel.

The use of S-elements in the Concur component results in a simple implementation but introduces unnecessary overhead. Figure 2 highlights the return-to-zero (RTZ)



(a) S-element (b) T-element

Figure 4. Alternative Sequencing Behaviours.

phases of the three commands connected to the component. The correct operation of the circuit requires that the processing phase of the controlling command, connected to the activate port, is executed before the processing phases of the controlled commands. However, the RTZ phases can execute concurrently since, in both environments, the data is returning to the spacer state. This observation allows the opportunity for performance improvements wherever Concur is used.

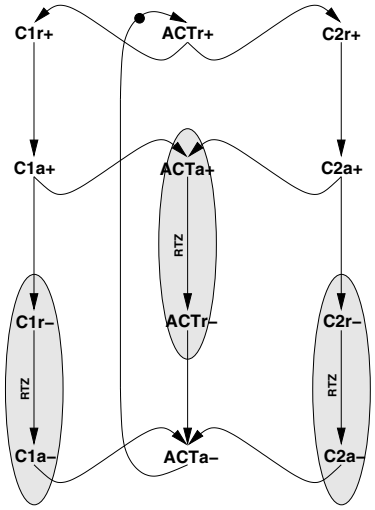


Figure 5. New Concur Behaviour.

Figure 5 introduces a replacement Concur component with more concurrent behaviour. In this case, the controlling command start its RTZ phase as soon as the processing phases of the controlled commands are finished, allowing all the RTZ phases to execute concurrently. This optimisation technique, that reorders the handshake events of a specified behaviour to obtain greater concurrency, is sometimes called reshuffling [4] and has been used successfully in other asynchronous design methods.

The new Concur behaviour cannot be implemented using S-elements. Instead, an alternative sequencing component, the T-element, is used. The operation of the T-element is shown in figure 4(b). The T-element will acknowledge on the input port as soon as the output port has completed the processing phase, allowing the two RTZ phases to execute concurrently.

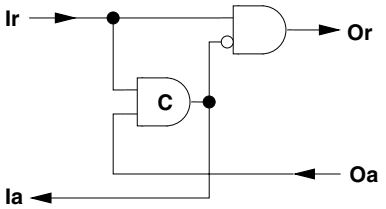


Figure 6. T-Element Implementation.

The implementation of the T-element is shown in figure 6. The T-element allows a more concurrent operation and is also smaller and faster than the S-element. The new Concur component is implemented by replacing the S-elements shown in figure 3 with T-elements.

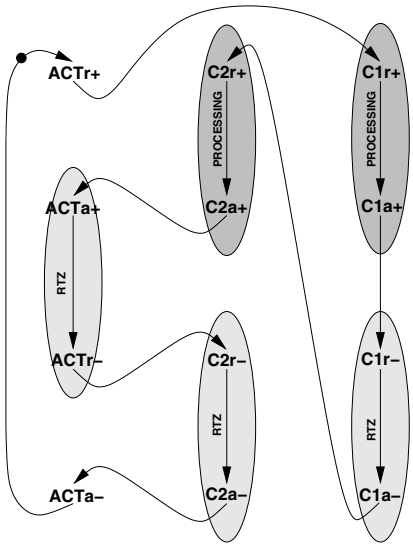


Figure 7. Conventional Sequence Behaviour.

2.2. Sequential Control

The operation of the Sequence component is shown in figure 7. On receipt of an activation, the component will trigger the two commands (C_1 and C_2) in sequence. As shown in the STG, the component will go through the com-

plete four-phase handshake in the first command before activating the second one.

Figure 8 shows the conventional implementation of the Sequence component. The component uses an S-element to control the execution of the first command. The acknowledge on the input port of the S-element is used to trigger the second command. In the parameterised component, S-elements are used to control the first $n-1$ commands.

As with the Concur component, the behaviour of the Sequence component introduces unnecessary overhead. The correct operation of the circuit requires that the processing phases of all the commands be executed in sequence. This requirement does not apply to the RTZ phases.

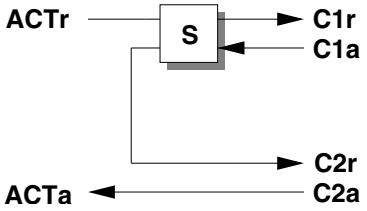


Figure 8. Original Sequence Implementation.

Figure 9 shows an alternative behaviour for the Sequence component. In this case, the controlling command start the processing phase of the second command as soon as the processing phase of the first one is finished, reducing latency and allowing the concurrent execution of RTZ phases.

The new Sequence component is implemented by replacing the S-element in figure 8 with a T-element. In contrast with the Concur component, care must be taken when using the new Sequence component: the new behaviour introduces the possibility of data hazards.

In QDI circuits such as those generated by Balsa, the correct operation of the circuit cannot rely on timing assumptions. This requirement may not be met if the new Sequence component is used and the same variable or channel is accessed by the two sequenced commands. In particular, the new component may introduce write-after-read (WAR) or write-after-write (WAW) hazards.

The WAR hazard is caused by the RTZ phase of the first command trying to close the variable read port concurrently with the second command trying to write new data. If the new data arrives first it will appear at the output of the read port before it closes, potentially altering the result of the first command.

A WAW hazard is related to the multiplexing of different writes to the same channel or variable. If the first command is executing the RTZ phase concurrently with the processing phase of the second command, the multiplexer may not be allowed to complete the four-phase handshake protocol on the output, leading to incorrect operation of the circuit.

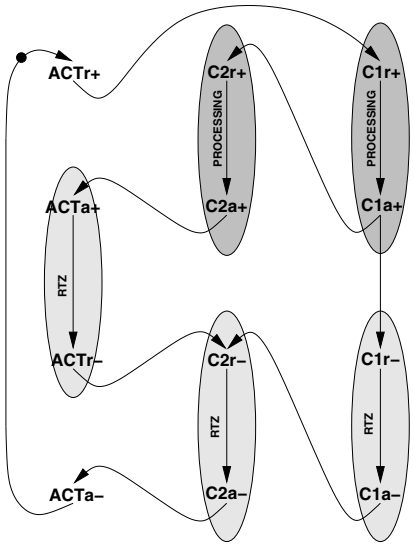


Figure 9. New Sequence Behaviour.

These hazards could be avoided by using interlock mechanisms, as shown in [6] but these were not used. Instead, the Balsa compiler was modified to identify potential WAR and WAW hazards and use the conventional sequence component in hazardous cases.

2.3. Passive Input Control

Handshake circuits are composed of fully asynchronous modules that communicate with each other. Each module operates at its own speed and can produce data at any time, independently of the speed of the module that consumes that data. This introduces the need for input control: a module that receives data must be able to decide when to accept it, how long the data should be kept valid and when to release it. Figure 10 shows how passive inputs are implemented in a module. Clearly, different input channels can originate from various independent source modules.

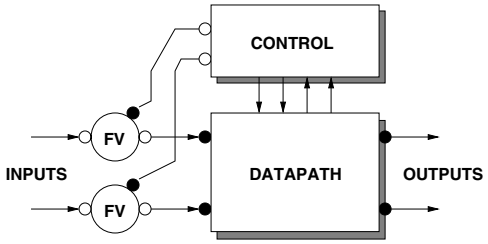


Figure 10. Passive Input Control.

A FalseVariable (FV) handshake component is used in every input to detect the arrival of new data, report it to

the control unit and allow the datapath to access the data as many times and for as long as needed. From the point of view of the module, the FalseVariable operates similarly to variable, *i*.e., the module can read the data many times.

Passive inputs are usually used when the environment has a choice as to which module input is activated. A simple example of the use of passive input control is an unbuffered multiplexer which accepts one, and only one, of several inputs and transfers it to its output.

The FalseVariable component, shown in figure 11, resembles a normal handshake Variable with one passive write port WD and a number of passive read ports RD_i. It differs, however, in the presence of an active 'probe' port S and in that it does not store data.

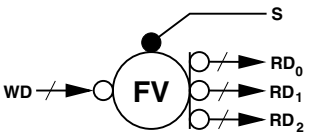


Figure 11. FalseVariable Component.

The behaviour of the FV component can be described as follows: A writer produces data that is pushed on channel WD. One or more readers consume data by pulling it on channels RD_i. The readers must wait until valid data has arrived on WD before reading. Channel S is used by FV to indicate the arrival of valid data on channel WD. Since FV does not store data, the writer is allowed to take the data away only after all the readers have consumed it.

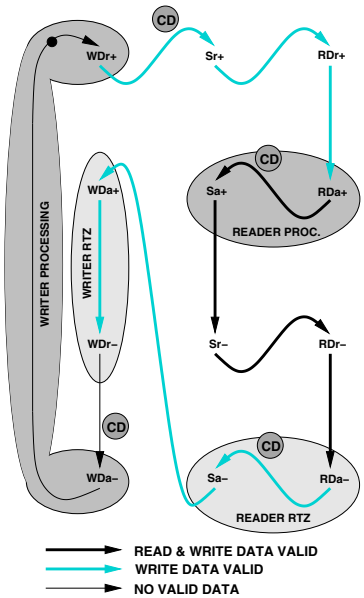


Figure 12. Conventional FV Behaviour.

Figure 12 shows the behaviour of the FV component. For simplicity, the STG shows a single reader. The STG has been annotated indicating when data is valid on the WD and RD channels. Clearly, the specified behaviour is safe since data-validity on WD completely encloses data-validity on RD.

The figure highlights the processing and return-to-zero (RTZ) phases of the writer and the reader. It is clear from the figure that these four phases are sequenced. Safe operation requires that the two processing phases be sequenced: the writer must produce data before readers can consume it. However, the RTZ phases can execute concurrently since, in both environments, the data is returning to the spacer state. Clearly, there is room for improved performance.

In a dual-rail implementation, further performance improvements are possible by modifying the way completion detection (CD) is done in FV. The readers must do CD in both the processing and RTZ phases to guarantee DI operation. In general, reader data CD would make writer data CD redundant. However, as indicated earlier, the readers are not required to consume data or they may do so conditionally. To guarantee DI operation in all cases, FV must do CD on writer data.

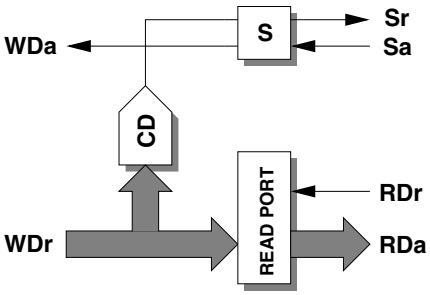


Figure 13. Original FV Implementation.

Figure 12 also shows how data is checked for completion in the dual-rail implementation: writer data is checked in the FV component after both processing and RTZ phases while reader data CD is carried out by the reader. It is clear in the STG that CD is done sequentially and is always in the critical path. If the data is wide, CD can be very expensive both in time and area.

Figure 13 shows the current implementation of FV. This implementation uses an S-element to achieve the sequential interleaving of events in the WD and S channels.

Figure 14 shows a new, more concurrent behaviour for FV. In this case, the writer is allowed to withdraw its data after the reader has closed the $read\ port$ $[RD_r-]$. This behaviour is safe and allows the two RTZ phases to execute concurrently.

The new behaviour also improves the way CD is carried out: Firstly, the two RTZ phases are executed concurrently,

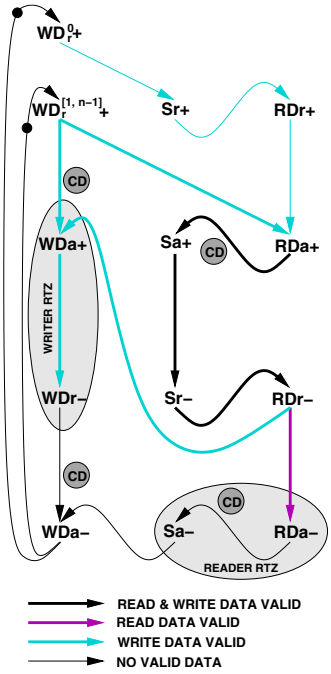


Figure 14. New FV Concurrent Behaviour.

which will result in concurrent CD on these phases. Secondly, the new behaviour allows concurrent CD also on the processing phases. A single writer data bit is used to trigger the events in S_r, allowing the reader to start its operation. Completion detection is carried out in FV while the reader is processing it. This new design is based on the Reverse Path Completion technique used successfully to implement Q^nDI combinational logic in Balsa.

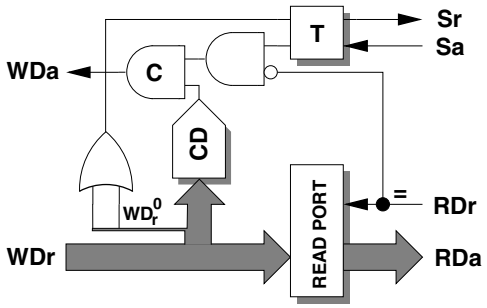


Figure 15. New FV Implementation.

The new FV implementation is shown in figure 15. The figure shows that the correct operation of the new implementation requires an isochronic fork (identified by =). This is a very reasonable assumption since the branch of the fork that has to 'win' the race is a local wire while the other branch corresponds to a signal that goes out to the writer and returns after the execution of its RTZ phase.

2.4. Active Input Control

Balsa allows the specification of active input control when there is no choice, *i.e.*, when the command module has a single input channel or a set of input channels which are activated concurrently. Figure 16 shows how active input control is implemented as a handshake circuit, for a single input channel.

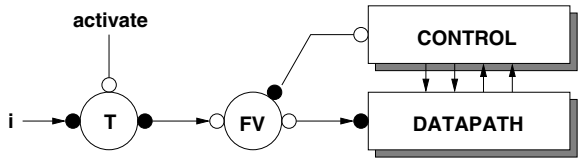


Figure 16. Active Input Control.

A FalseVariable component is also used to implement active input control. The Transferrer component (T) actively fetches data from its input channel and pushes it on its output channel. Clearly, this type of input control can also benefit from the improved FV design presented in the previous section. However, the absence of input choice introduces new possibilities for improving performance.

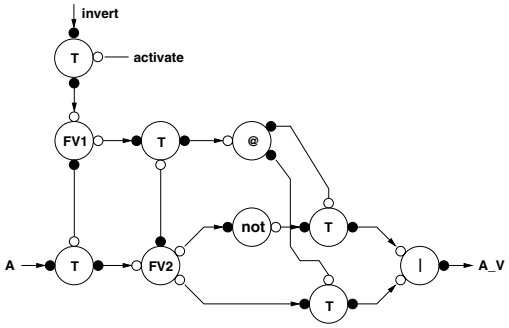


Figure 17. ALU Input Control.

Figure 17 shows a small section of the ALU of the SPA processor. A is one of the data input channels and, depending on the value of the $invert$ signal, either A or its complement, provided by the not component, is passed to the next stage.

The operation of the circuit is as follows: the $activate$ signal starts the operation by requesting the $invert$ input. When this signal arrives, FV1 will trigger the transferrer that pulls A. When this signal arrives FV2 will send the value of the $invert$ signal to the Case component, denoted by @, to decide if A is complemented or not. Unfortunately,

although the $invert$ signal arrived earlier, the circuit has to wait until A arrives to decide which path to take.

Clearly, there is unnecessary synchronisation between control and data signals that reduces the performance of the circuit. The FV component triggers the command only after data has arrived. In the absence of choice, there is a single command to be activated in the datapath and there is no need to wait for data to arrive to identify it. The command can be activated as soon as possible and it will wait until valid data arrives before processing it. In the ALU example above, the arrival of the $invert$ signal should be enough to trigger the Case component so that the path to be followed by A is selected as soon as possible, reducing the control overhead.

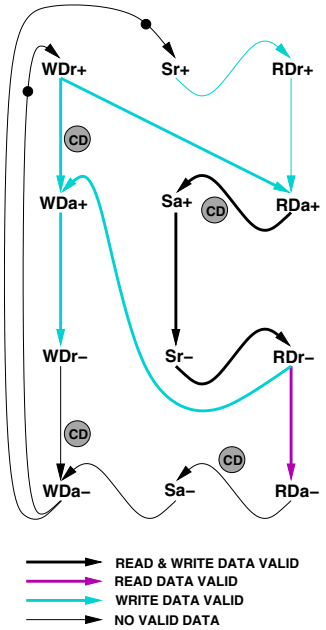


Figure 18. Eager FalseVariable Behaviour.

There are several possible ways to eliminate the unnecessary synchronisation detected in the handshake circuit above. A simple and efficient one is to modify the behaviour of the FV component, allowing it to trigger the command without waiting for the data to arrive.

Figure 18 shows the behaviour of this *eager* FalseVariable component. The behaviour shown in the figure is similar to that in figure 14, allowing the concurrent RTZ phases and implementing reverse path completion detection, with a significant difference: the command is triggered, denoted by S_r+, as soon as the previous input operation has completed. Basically, the command gets a head start by not having to wait for the data to initiate the control operations.

Figure 19 shows the implementation of the Eager FV component. It is very similar to the implementation shown in figure 15 and has the same isochronic fork requirement. The difference is that the input data is not used to trigger the T-element. Instead, the T-element is triggered as soon as the previous operation is completed.

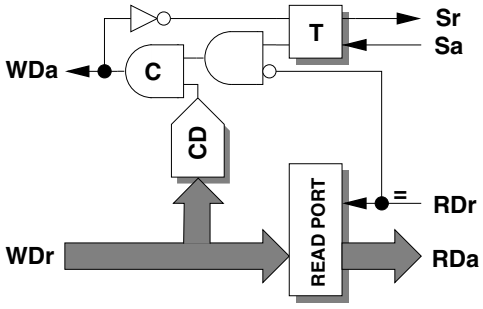


Figure 19. Eager FV Implementation.

Unfortunately, the eager FV component may, in some cases, trigger the command too soon. If the command is 'shared', *i.e.*, activated by more than one section of the circuit, several sections could try to activate it concurrently, resulting in incorrect operation. To guarantee correct operation, the command should not be triggered before the arrival of the $activate$ signal (shown in figure 16).

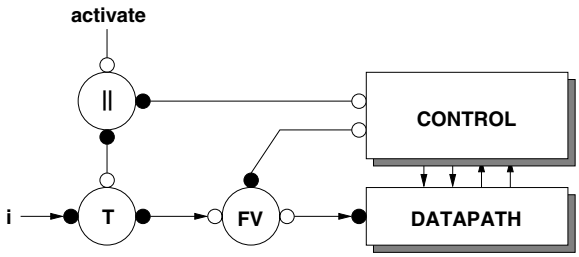


Figure 20. New Active Input Control.

Figure 20 presents a new handshake circuit implementation of the active input control that allows the safe use of the Eager FV component. The control unit must synchronise the FalseVariable component signal with the $activate$ signal, making sure that the command is triggered as early as possible but not too early.

3. Simulation Results

Table 1 shows the results of simulating the execution of a benchmark program in different implementations of the SPA processor, generated by selectively introducing the new handshake component implementations presented earlier. It should be noted that all the processor implementations used the original Balsa code. The performance of the original SPA is set as the reference.

Processor	Relative Performance
SPA	1.00
SPA1 (SPA + new Concur and FV)	1.51
SPA2 (SPA1 + Eager FV)	1.69
SPA3 (SPA2 + new Sequence)	2.06

Table 1. Simulation Results.

SPA1, which incorporates the new Concur and FV components, obtains a very significant 51% improvement in performance. These two replacement components can be substituted for the originals without restriction.

SPA2, which incorporates the use of Eager FVs into SPA1, obtains an even more significant 69% performance improvement. This component can be used by the Balsa compiler wherever active inputs or passive inputs without choice are specified by the designer. In SPA, 420 FV components are used and 378 could be replaced by the eager version.

Finally, SPA3 obtains an impressive 106% performance improvement, doubling the performance of the original SPA processor. SPA3 incorporates the use of the new Sequence component in addition to the previous changes. As mentioned in the text, this component must be used selectively to avoid introducing hazards. SPA contains 82 Sequence components (66 are 2-step, 12 are 3-step and 4 are 4-step sequencers) and only 2 of them cannot be replaced with the improved design. These two sequencers are used to send data sequentially through the same channel and would introduce WAW hazards.

4. Conclusions

Balsa has demonstrated that it is capable of dealing with the complexity of a full-featured, 32-bit processor, requiring only 25% of the design effort of equivalent non-synthesised processors. Unfortunately, the price of this rapid development is reduced performance.

The work presented in this paper shows that careful specification and design of a set of control handshake components can have a large impact on the performance of Balsa-synthesised circuits. In particular, the introduction of new Concur, Sequence and FalseVariable components, used to implement parallel, sequential and input control, results in a processor implementation with twice the performance of the original one.

References

[1] 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*, 2002.

[2] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *Computing Journal*, 45(1):12–18, 2002.

[3] T. Kolks, S. Vercauteren, and B. Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 233–243. IEEE Computer Society Press, Mar. 1996.

[4] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[5] A. Peeters and K. van Berkel. Single-rail handshake circuits. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 53–62, May 1995.

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

[7] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple. SPA – a synthesisable Amulet core for smartcard applications. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 201–210. IEEE Computer Society Press, Apr. 2002.

[8] K. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.