Implementing Balsa Handshake Circuits

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy in the
Faculty of Science & Engineering

2000

Andrew Bardsley

Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

Two major additions to the Balsa asynchronous circuit synthesis system are presented.

Firstly, a new back-end for generating VLSI and FPGA implementations of Balsa descriptions is described. This back-end allows parameterised generation of handshake components from template descriptions.

Secondly, a new method for producing handshake circuits which are better optimised during construction is described. A number of new handshake components are introduced. Each of these components, by virtue of their greater degree of parameterisation, can replace clusters of existing handshake components. These parameterised components are used to implement bespoke synchronisation, encoding/decoding and bitwise word division/construction operations using component specific optimisations which replace the, potentially dangerous, gate level optimisations previously used.

The design and VLSI implementation of a substantial test design using Balsa, a 32 channel DMA controller, is also presented. This DMA controller was constructed as part of the AMULET3i asynchronous processor macrocell. It is a hybrid synchronous/asynchronous design using a combination of full custom, hand designed standard cell and Balsa synthesis. The AMULET3i macrocell has been fabricated as part of the DRACO communications controller IC.

A simplified, fully asynchronous, version of this DMA controller is also presented in order to illustrate the use of the optimisations presented in the thesis.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Thanks to my supervisor, Dr. Doug Edwards, and to the whole of AMULET for their help, tolerance and suspension of disbelief. Thanks to the University for its flexible attitude to acceptable time keeping. Special thanks to my proof readers: Dr. John Bainbridge, Dr. David Gilbert, Dr. David Lloyd, Peter Riocreux for their fast turn-around times.

Special thanks also to Chatchai Jantaraprim for ceding his project with such good grace.

Grudging acknowledgement of anyone who ever said “Oh, you could do that in Balsa”.

Thanks also go to the authors of many fine free software packages used in the implementation of this project and the preparation of this thesis.

This thesis was prepared using Basser Lout 3.23 on both Athlon and Pentium-II based PCs running the Linux operating system 2.4.0. Figures were prepared using xfig 3.2.3 and imported into Lout as Encapsulated Postscript.

An Portable Document Format version of this thesis available at:
ftp://ftp.cs.man.ac.uk/pub/amulet/theses/bardsley_phd.pdf

Chapter 1. Introduction

This thesis extends previous work on the asynchronous circuit synthesis language Balsa [6] and on Handshake Circuits [9] (both Balsa and handshake circuits are introduced in chapter 3).

These extensions include the development of a new synthesis back-end from the Balsa handshake circuit intermediate file format Breeze (Breeze is described in §4.1.1) into CMOS and FPGA implementations. This back-end replaces the simple back-end for the language Tangram [64] developed as part of the EXACT project [25]. That older back-end was specific to the Cadence CAD environment and the (locally developed) AMULET low-power cell library [2].

Also presented are a number of optimisations for producing implementations of handshake circuits without requiring gate level optimisation. These optimisations centre around the use of a number of new handshake components. These components have a higher degree of parameterisation than has previously been described and are capable of incorporating a number of existing optimisation schemes: handshake circuit control resynthesis of the sort presented by Kilks, Vercauteren and Lin [45], encoder/decoder construction such as the Balsa Case component, bespoke synchronisation components and better support for bitfield extraction and word reconstruction.

The aim is to be able to synthesise Balsa circuit descriptions into handshake circuits which have simpler structures than those previously possible. These simpler structures reduce the number of interconnecting channels and superfluous synchronisations inherent in implementations which use handshake channels to communicate between all components. These improvements are sought without resorting to the potentially unsafe gate-level circuit optimisation.

Two design examples illustrate the use of Balsa to construct practical circuits. The first, described in chapter 6, is a 32 channel DMA controller constructed using a hybrid of synchronous and asynchronous techniques with the majority of the asynchronous portion constructed

using the new Balsa back-end but without the new components described in chapter 5. This DMA controller has been fabricated as part of the AMULET3i portion of the DRACO communications IC.

The second example is a simpler, fully asynchronous, DMA controller based on the same register structure as the AMULET3i DMA controller. This controller is used to illustrate the improvements in Balsa's optimisation strategy described herein. Justifications for the choice of a DMA controller as a typical Balsa design are given in §6.1. The design of this simple controller and the analysis of the controller's implementation are given in chapter 7.

The bulk of the work described in this thesis is concerned with the construction of the new Balsa back-end and the use of that back-end to construct the DMA controller integrated into AMULET3i. The aim of this design example is to show that a direct, optimised-during-construction, approach to asynchronous macromodular logic synthesis can be effective at producing practical circuit implementations. The new components presented in chapter 5 have (for the most part) not been implemented and as such represent the next step in producing an improved direct-synthesis back-end for Balsa. These new components and optimisations should, therefore, be considered to be a less significant part of this work than the back-end or DMA controller designs.

The remainder of this introductory chapter introduces some basic concepts of asynchronous design which may not be familiar to some readers. The final section of this chapter (§1.2) gives a short overview of the structure of the remainder of the thesis.

1.1. Asynchronous design

The design implementation and synthesis techniques described in this thesis are all asynchronous in nature. The work described herein does not try to measure the advantages of asynchronous techniques over conventional synchronous approaches. That being said, there are a number of properties which asynchronous circuits possess which are claimed to bring benefits over their synchronous counterparts. These include:

Easier/more intrinsic modularity – Modularity involves designing circuit units which

can be easily interfaced to each other to construct larger circuits. Modular interfaces allow connected units running at different rates to communicate without undue additional control complexity. This is easier to achieve using asynchronous handshaking (detailed in §1.1.1) than using clock synchronised transfers between units. Asynchronous units can be made to wait for each other by delaying the transitions on handshake control signals without incurring either delays quantised to the clock period or failures caused by metastability when trying to synchronise data. The use of asynchronous handshaking can be applied to very small circuit elements to allow the benefits of modular construction to be exploited within asynchronous circuits. This is the major reason for using asynchrony in the work described in this thesis.

Reduced energy consumption – Driving the clock distribution tree and the transistors which gate the clock onto latches can consume a considerable proportion of the energy supplied to a synchronous circuit. Well designed asynchronous circuits can avoid consuming as much energy as synchronous circuits where parts of the circuit are idle for a portion of the time. Synchronous designs can use techniques like clock gating, voltage scaling and clock frequency scaling to achieve similar results, albeit with more complicated implementations.

Improved electro-magnetic compatibility – As a large proportion of the energy dissipated by synchronous circuits is from gates/interconnect connected to the clock, E-M radiation emitted by synchronous circuits tends to be correlated to the clock's frequency and harmonics thereof. This correlated radio emission can make it difficult to operate radio receivers, meet emissions standards and provide enough current to the circuit at clock edges. It has been observed that although an asynchronous circuit performing similar work may emit a similar total amount of E-M radiation, that radiation is much less correlated to the circuit cycle time due to the variation in cycle time caused by changes in interface timings, data-dependent operation timing and thermal and supply voltage variation. Spreading the E-M emissions over a larger range of frequencies makes it much easier to operate radio receivers near asynchronous circuits and to reduce the amount of shielding required. Reduced EMI is a major reason for the adoption of the AMULET3i macrocell as part of the DRACO chip described in §6.3.

Architectural benefits – It is possible to implement some structures using asynchronous techniques which may not be possible when using a single clock. Asynchronous circuits can contain portions which operate at a speed appreciably higher than the 'ambient' cycle

time by relying on locally looping control synchronised with more global control only when necessary. This principle may be used to implement cheaper forms of circuit elements like multipliers by allowing a fast iterative multiplier to replace a more expensive combinatorial (or more slowly iterating) multiplier. The multiplier in the AMULET3i is built using this principle [47]. Another example is the boundary between software and hardware. The Philips Myna pager [60][44] exploits the improved EMI characteristics of asynchronous circuits to run the radio receiver and controlling microprocessor at the same time. This allows all three of the dominant pager standards to be implemented using the same hardware. Previous synchronous pagers must contain additional hardware for each pager standard to receive the different data packet formats from the radio receiver while the microprocessor is sleeping.

The majority of modern asynchronous circuit design techniques are based on the use of *handshaking* to communicate between units. The form of handshaking and its role is a defining parameter of a particular design technique. The majority of asynchronous circuits are not designed with arbitrarily placed timing constraints and so techniques are also often characterised by their approach to delay management. Delay management allows the designer to be sure that a circuit will function under changes of circuit delays due to temperature fluctuations, variations in fabrication process and implementation technology retargeting (e.g. process migration). The consistent application of a system of timing assumptions or more easily validated timing constraints is termed a *delay model*. The way that data is transported around a system and the way that data corresponds to control is determined by the *data encoding*.

The remainder of this introduction to asynchronous circuit design techniques outlines the common handshaking protocols, delay models and data encodings used in the design techniques described in chapter 2. All of these techniques and common asynchronous terms are relevant in the discussion of handshake circuits in chapter 3. Two component, commonly used in the implementation of asynchronous circuits, also described: the ubiquitous Muller C-element (in §1.1.4) and the S-element (in §1.1.5).

1.1.1. The handshake

A large part of the control in asynchronous circuits is devoted to the communication of data and control signalling from one part of a circuit to another. Without a global clock, there is

no global periodic source of events which can be used by communicating units to signal when data is provided and taken in a communication. Bespoke solutions to this problem involving control signals which convey readiness to communicate, data validity and the receipt of data are possible in each case where a communication is formed. To simplify the task of building such communication control systems, the notion of the *handshake* and the data *channel* were introduced. Handshakes are used where two or more units require control of a synchronous transfer of data between them. Handshakes can also be used as a mechanism for synchronising two units without the explicit transfer of data (e.g. to implement token passing schemes or control shared resources).

Where the data and handshake control signalling for a communication are considered a single conceptual unit, that communicating connection is often called a *channel*. Of the two units connected by a channel, one unit is *active,* it initiates the handshake by issuing a *request* to the other unit, and the other is *passive,* it receives the request and replies (when it is ready) with an *acknowledgement*. A handshake which consists of the exchange of only two tokens (one request, one acknowledgement) is the simplest form of handshake.

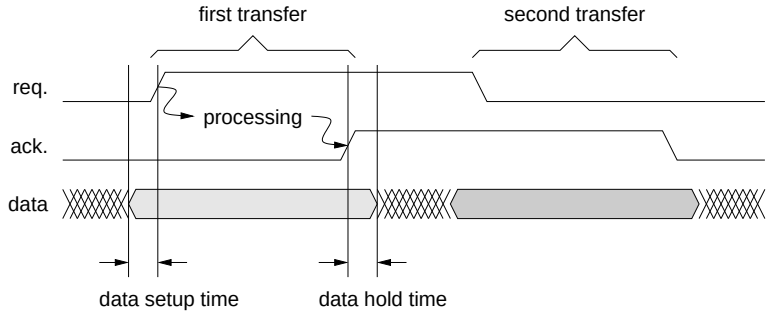


Figure 1.1. 2-phase handshaking

When two symbol handshaking is implemented using wires representing request and acknowledgement, with transitions on those signals communicating the tokens, this is called *2-phase handshaking* or *transition signalling.* The two phases are the periods between a request and its acknowledgement, forming the handshake itself, and the period between an acknowledgement and the next request, forming the idle phase. Figure 1.1 shows an example of 2-phase handshaking with both request and acknowledge signals starting low, both going high after the first handshake, then both falling again after the second handshake. Note that rising and falling signal transitions are equivalent in this scheme. Data validity is signalled by the ar-

rival of the request signal at the target. Data is allowed to become invalid at the initiator on receipt of the acknowledgement by the initiator. 2-phase handshakes can also be constructed using a single signalling wire which the active unit raises to indicate a request and the passive unit drops to signal an acknowledgement. Control of the driving of the control signal must be passed between units in between events. This form of handshaking is called *single track handshaking* [11] but is not considered in the remainder of this thesis.

In the 2-phase example, data is shown as a set of wires accompanying the signalling wires with matched delays to those wires. This way of attributing data to a pair of handshake signalling wires is termed *bundled data* and is a very common method for implementing cheap but non-DI (see §1.1.3) channels.

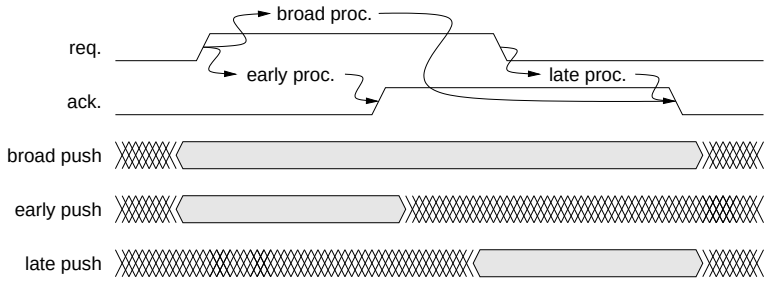


Figure 1.2. 4-phase push handshaking

Figure 1.2 shows another common form of handshaking – 4-phase handshaking. 4-phase handshaking can be simpler to implement in CMOS than 2-phase as the absolute levels of signals can be used to determine the phase of the handshake not just the difference in request and acknowledge. Unfortunately, to restore the signalling wires to their initial states at the end of each communication, two extra phases need be added to the handshake in order to sequence the return-to-zero of the two signalling wires. Data validity in 4-phase handshakes can be signalled a number of ways, the most common of which (*early*, *late* and *broad* data validity) are shown in figure 1.2.

The direction of a communication-initiating request need not be the same direction as that of the data. The examples given so far illustrate only communications where the data flows with the request. These are called *push* communications. Another form of communication is possible, that where the data flows from target to initiator under control of the acknowledgement. Figure 1.3 shows the common data validity schemes used with pull communications in both 2-phase and 4-phase handshaking. Note that in 2-phase pull and 4-phase broad pull the cost

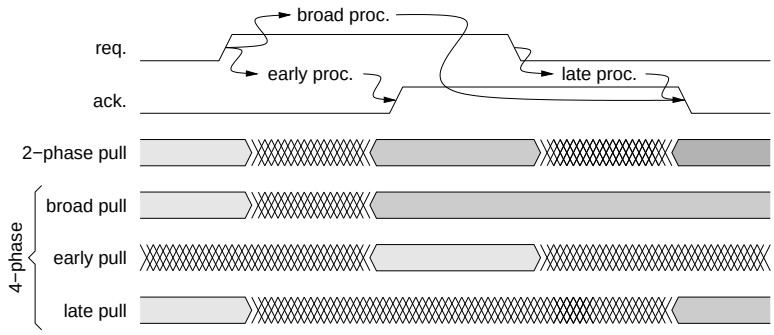


Figure 1.3. 2/4-phase pull handshaking

of implementing the channel may be high as data must be held valid **between** handshakes. This may involve adding latches to an implementation for each pull channel. For this reason, in 4-phase handshaking, *reduced* protocols [58] are useful.

In reduced signalling schemes, the data communicated may become invalid before the event which would ordinarily signal the completed receipt of that data is sent or received. For example, the reduced broad 4-phase push protocol allows the data to become invalid before the falling edge of the acknowledgement. This is similar to the early 4-phase push protocol (and as such early 4-phase inputs can be connected to 4-phase reduced broad outputs) but a reduced protocol may imply other constraints which allow the data receiving unit to be implemented more easily/cheaply. Figure 1.4 shows a simple implementation of a handshake controlled write port to a transparent latch. The acknowledgement for the write is formed from the delayed request and so acts as a latch strobe. This being the case, a reduced broad input may be presented to the latch in place of a broad input where the data on that input remains valid until at least the point where the latch closes.

Reduced data validity is most useful when used with 4-phase broad pull signalling. Using the acknowledgement signal as a latch control signal with 4-phase broad pull at the initiator is very convenient. When reduced broad signalling is used, the constraint that data remains valid whilst the acknowledge signal remains high and remains valid for some period after that in order to allow its use as a data strobe. Read ports are shown on the latch example in figure 1.4 as channels consisting of the output of the latch and a delay element in the control path for delay matching. Where a read is performed on this latch, which is then followed by a write, the read data becomes invalid before another read can be made. For the latch to

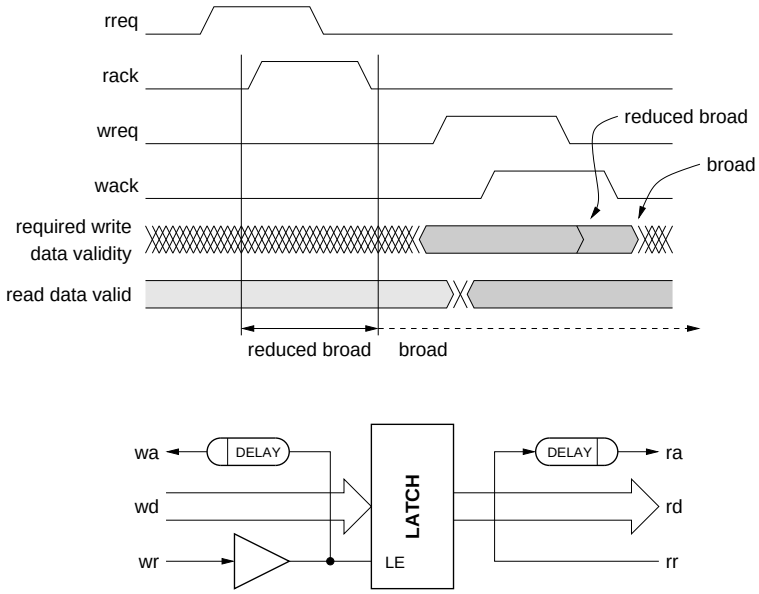


Figure 1.4. A latch using reduced broad push and pull protocols

support broad pull reads, the data on this read port must be maintained until the next read. If, however, it is understood that the latch will not be written until the recipient of the latch data has completed its read (i.e. either totally finished with the data on that port or latched a copy of it), the read port can be considered to support a form of reduced broad protocol. The acknowledgement from this reduced broad handshake can, in turn, be used as a latch control or validity indication for the data on that read port. Using these broader validity signals can make minimum latch enable pulse width constraints easier to meet than when early signalling is used. Early signalling would entail using a latch control pulse which only spans the rising acknowledge to falling request portion of the pull handshake (as shown in figure 1.4). Using the reduced protocol (with its two events on the acknowledgement occurring before the end of handshake data validity), a delay placed in the acknowledgement path is used twice so allowing smaller, easier to implement, delay elements to be inserted.

The cost of using reduced protocols in these ways is a greater reliance on timing validation to ensure correct circuit behaviour. This may not involve any additional design steps for circuits which use bundled data (in which considerable validation may already be necessary).

1.1.2. Delay models

Most design techniques make at least some assumptions about the timing characteristics of

components and wires. In synchronous design, assumptions are made (which need to be validated in layout) about the arrival time of the clock signal at clocked components. Further assumptions are made about the relationship between propagation delay and data hold times when using edge triggered devices, the propagation delay of combinatorial elements between latches and the period, duty and timing relationship between clocks. The global nature of the synchronisation formed by clock and latched control/data makes connecting circuits running at different rates more difficult than in asynchronous systems where the handshake and partially (or wholly) delay-insensitive communications make interconnection of circuits easier.

As clock rates increase and circuit feature sizes shrink (leading to a dominance of interconnect delays over gate delays), these timing assumptions become less and less realistic so making global synchrony difficult to achieve. Asynchronous circuit design abandons the use of global synchronisation in favour of local synchronisation between communication/participating units. The use of localised control in this way requires different approaches to delay management than those used in synchronous design.

Asynchronous circuits must meet similar critical-path constraints as synchronous circuits in order to be useful in their intended applications. In addition, timing relationships between local communication signals internal to asynchronous circuits must be validated to ensure correct circuit functionality under similar worst case conditions as required for synchronous validation. In synchronous design, it is often possible to achieve timing closure by lowering the target clock rate of the system. For an asynchronous circuit, however, it may not be possible to rescue a design by such a simple constraint relaxation. Adjusting the cycle rate of the circuit by changing the timing of the environment is often easy to achieve. Unfortunately, the timing of internal circuit stages, which may affect circuit functionality, is rarely easy to control in this way. For this reason, the problem of achieving timing closure for an asynchronous circuit can be potentially greater than that of a similar synchronous circuit.

Delay models help in simplifying the specification and validation of timing constraints. Circuits designed with a particular delay model in mind are often simpler to design and are easier to validate by comparison/cosimulation with their specifications because it is easier to identify places where timing constraints need to be met. This is particularly true of the design of synchronous circuits. At moderate circuit speeds, the requirement to distribute the

global clock to all parts of a circuit with only a small amount of skew is not difficult to meet. Circuit area can be reduced by migrating the local communication control logic required in asynchronous circuits into global, clock synchronised control with data transfers quantised to multiples of the clock period.

In the design of asynchronous circuits, there are a number of commonly applied delay models. Delay models may include delay/timing *assumptions* and *constraints.* Delay assumptions are globally applied simplified models about the nature of delays within a circuit. Popular delay assumptions include those about the relative significance of interconnect over gate delays (for example, in speed-independent circuits), the behaviour of wire forks (in quasi-delay-insensitive circuits) and the minimum response time of a circuit's environment (e.g. the fundamental mode, described in §2.4.1). Constraints are timing requirements which are applied at specific points in a circuit in a response to a functional need for that requirement at that point in the circuit. A good example of timing constraints are the data bundling requirements in systems with single rail bundled-data interfaces.

Circuits built using timing assumptions or containing timing constraints need to be tested to ensure that the assumptions are realistic and that the constraints are met. Timing validation in the current Balsa system is discussed in §4.4.2.

Popular delay models include:

Delay-insensitive (DI) circuits

DI circuits require no timing constraints or assumptions between gates to be preserved for circuit functionality to be guaranteed. Unfortunately, at gate level, few interesting circuits conform to the delay-insensitive ideal. For this reason, delay-insensitivity is most often applied to larger, more coarsely grained, units constructed using other timing regimes in order to make them easier to compose.

Quasi-delay-insensitive (QDI) circuits

QDI circuits are like DI circuits except that forks in wires may be assumed to be *isochronic.* Isochronic wire forks are forks which result in transitions at the leaves of the fork arriving

at the same time (or within some acceptably short period of time of each). This property is commonly used for the demultiplexing of request signals to a number of units, where only one unit is ready to acknowledge at a time. The initiator of the request can be certain that the request has been withdrawn when it receives an acknowledgement to the falling request (for a 4-phase circuit). The QDI assumption can also be extended to include assumptions of isochronic propagation after a number of logic gates' distance from the fork (the extended isochronic fork and Q^nDI delay models [10]). Although QDI is an attractively simple model, it can sometimes be over-restricting, especially where the 'probings' at the far ends of a fork can be guaranteed to occur at suitably well separated times [8].

Speed-independent (SI) circuits

In SI circuits, interconnect delays are assumed to be insignificant, only components exhibit appreciable delays. Forks in wires are assumed to be isochronic and so unacknowledged forked signals are assumed to have changed based on the observation at a single point on a fork.

Scalable Delay-insensitive (SDI) circuits

Scalable delay-insensitive circuits allow the principles of speed-independent circuits to be applied to larger designs than may be justified by their, potentially unrealistic, delay assumptions. SDI was developed to allow regions of a circuit which are small enough for the speed-independent assumption to hold to be identified so allowing a design to be partitioned. SDI was used to implement the TITAC-1 and TITAC-2 [68] microprocessors.

1.1.3. Data encoding

The way in which data is communicated from one part of a circuit to another is also an important design decision. In synchronous design, data is usually binary encoded where 2^n distinct symbols can be represented by boolean logic symbols $\{0, 1\}$ on n wires with all of the possible combinations of boolean symbols on those wires representing a symbol. The clock is used to signal the validity of both these data signals and the circuit's control signals.

Bundled data

Bundled data (also known as single-rail) communication involves transmitting control and data tokens on separate wires to which fixed timing relationships are applied. The use of binary encoding with request/acknowledge signalling is the most commonly used form of bundled data. Bundled data is popular due to the ability to separate control from datapath. Unfortunately, this separation often requires the use of explicit delay components or matched paths for control and data to bring control and datapath timings back together where data signals pass through processing units. Timing validation is necessary to ensure that bundling constraints are met and delays are of appropriate sizes.

One-hot codes

In cases where the number of symbols which can be communicated is small, *one-hot* (or unary) encoding may be used. One-hot encoding entails using n wires to encode n symbols, one wire per symbol. One of the n wires is held high (at 1) and the other $n - 1$ are held low (0) to encode a particular symbol.

With one-hot encoding, there is an obvious extra state for the n wires which can be used to indicate that no symbol is being communicated, this is when all n wires are held low. Making use of this quiescent or NULL state (NULL Convention Logic, see §2.4.5, calls the quiescent state NULL) allows the wires communicating a one-hot value to also communicate the validity of that value. Encoding data validity in with data is the key to what are termed *delay-insensitive codes.*

Delay-insensitive codes

DI codes are encodings of data on wires in which rules can be written for when data is valid (in the case of one-hot encoding, when any wire becomes high) and for when the communicating wires are quiescent (in one-hot: when all the wires are low). These rules must define how data symbols map onto the signal levels of the wires which physically implement communications. This mapping must also include a representation for the quiescent state. The simplest extensions to one-hot encoding, in the domain of DI codes, are the n-of-m codes. In these codes, symbols are formed by raising exactly n of the m wires for each communication (so one-hot

encoding on n wires is 1-of-n encoding). One of C_m^n symbols (plus the quiescent symbol) may be transmitted with each communication making n-of-m codes cheaper to implement than one-hot encoding (in terms of number of wires) at the expense of more complicated encoding/decoding to/from binary.

It is important that choices of data validity/symbol encoding rules can be decoded in a delay-insensitive manner. For communications which make use of the quiescent state to separate symbols, it is only necessary to ensure that no symbol's set of significant wire levels is a subset of another symbol's. Where the number of significant wire levels (usually high wires) for each symbol is the same (as in n-of-m encoding), this delay-insensitive requirement is always met.

Handshaking communications can be formed using DI codes by adding an extra control signal. For push communications, the completion/validity operation on the data wires provides a request, the extra control signal forms the acknowledge flowing in the opposite direction to the data. For a pull communication, the control signal carries the request, with the validity of encoded data signalling acknowledgement of that request. 4-phase DI coded handshakes use the quiescent state as the lowered control signal state.

Delay-insensitive codes can be used in transition signalling by using rules for data validity based on a number of transitions having taken place on the signalling wires. No quiescent state is necessary and the communication will be very energy efficient due to the low number of signal transitions. Unfortunately, transition signalled (2-phase) DI codes can be expensive in circuit area to decode/encode making them most suitable for applications where communication energy must be kept to a minimum but sparse encodings such as one-hot encoding may be too expensive in wires. Off-chip communications is an example of such an application [28].

Communications can be made of several bundles of wires, each with its own data encoding. Validity of data on the whole communication occurs when each of the individual bundle's data is valid. The most common of these schemes is *dual-rail* encoding.

Dual-rail encoding

Dual-rail encoding is a 1-of-2 encoding where each bit of data is carried by 2 wires. A handshake constructed using dual-rail encoding will take $2n + 1$ wires for n bits of data. A data valid signal for a dual-rail communication can be formed by ORing the 2 wires of each bit and then combining those individual bit valid signals using an AND operation (or, more often, a tree of Muller C-elements as described in §1.1.4).

Dual-rail solutions have been the most popular way of implementing DI interconnect due to the ease with which binary data can be encoded to/decoded from dual-rail communications. Many of the systems described in chapter 2 use, or have used dual-rail encoding to implement interconnect. The cost of the C-element tree required for synchronisation of data with control signals can become a major cost in dual-rail datapaths.

1-of-4 encoding

Recently, 1-of-4 encoding (of pairs of bits in a communication) has become popular for asynchronous datapath implementations. 1-of-4 encoding takes the same number of wires to encode as dual-rail encoding and has similarly simple encoding/decoding circuits for translation to binary encoding. The advantages of 1-of-4 over dual-rail encoding include reduced power and simplified data operations. In 1-of-4 encoding, only 2 transitions are required on 1 wire (or 1 transition on 1 wire for 2-phase signalling) for each 2b of data communicated. Dual-rail encoding requires 4 transitions for each 2b of data communicated (or 2 transitions in 2-phase signalling), twice as many transitions as for 1-of-4. Data operations are simpler because DI implementation of combinatorial functions tend to find sum-of-products implementations where the product terms are minterms of the data bits input to the operator. In dual-rail encoding, a minterm over a pair of 2b arguments involves a function of 4 inputs (2 inputs per argument, 1 of those per bit). 2b minterms in 1-of-4 encoding, however, consist of only 2 inputs (1 input per argument, 1 input for each 2b of argument). Any function implemented as a sum of minterms for 1-of-4 encoded data will, therefore, be smaller than a similar expression in dual-rail encoding.

1.1.4. The Muller C-element

The Muller C-element is one of the most common additions to the basic set of logic gates

made in order to make the implementation of asynchronous circuits easier. It is described here as it forms part of the implementations of other, larger components later in this thesis. The C-element is also presented as it is used to implement control and data synchronisations in many asynchronous design styles (specifically, many of the styles described in chapter 3). It is such a commonly used component that, where possible, the C-element is implemented as a standard cell in order to make its operation as fast as possible.

The C-element provides an AND function for signal transitions. Figure 1.5 shows the circuit symbol, operational waveforms and a typical transistor level implementation for a 2-input C-element. Starting with both of its inputs low, the 2-input C-element's output will also be low. When all its inputs have gone high, the output then goes high. This behaviour is similar to an AND gate. Unlike an AND gate, both inputs must then fall for the output to fall. To implement this function, the C-element must contain some storage element (the weakly maintained node 'nQ' in the figure).

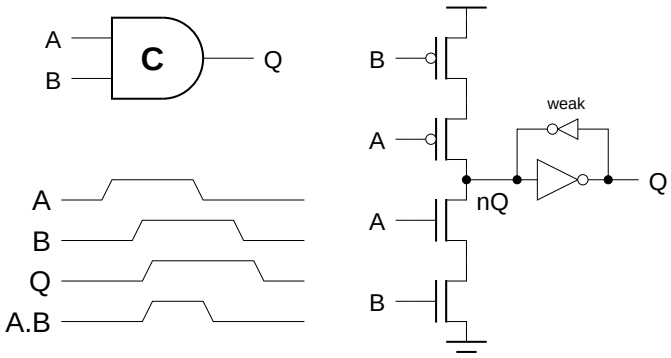


Figure 1.5. The Muller C-element

C-elements are useful for synchronising control signals by waiting for all of them to transition in the same direction. They can be used to implement DI joins of forked control where an AND gate would not synchronise the return-to-zero transitions of the signals making up that control. A bubble on the input of a C-element indicates that that input is pre-fired at circuit reset time. Pre-fired inputs can be useful for marking initial states in systems built using networks of C-elements. A common example is a FIFO with C-element control. Pre-fired inputs are placed on acknowledging inputs to the latch control to allow previous acknowledgements (or reset) to trigger the start of the next control cycle. Bubbled inputs can be implemented by inverting the input signal in question (giving the bubble its conventional meaning). A C-

element with all its inputs pre-fired/inverted is equivalent to a C-element with an inverted output. Moving inversions around systems of C-elements (with the correct reset states) can give similar inversion state saving benefits as in circuits of conventional logic gates.

When used in QDI or SI circuits, C-elements are also often required to allow pulses on one input to be ignored if the other input(s) have not made a successful transition (not a pulse) since the last output firing. C-elements can also be constructed to respond to transitions in only one direction on particular inputs. These forms of C-element are known as generalised C-elements or asymmetric C-elements.

1.1.5. The S-element

The S-element [9](also known as a Q-module [49]) is a circuit element commonly found in the implementation of handshake components. An S-element has 4 connections forming 2 request/acknowledge handshake pairs – 'Ar'/'Aa' and 'Br'/'Ba'.

The 'A' pair of signals form the initiating handshake. Raising 'Ar' causes the S-element to raise 'Br'. A complete 4-phase handshake can now take place on 'B' before 'Aa' is raised to signal acknowledgement of the initiating handshake. The 'A' handshake may now take part in a return-to-zero. Figure 1.6 shows the symbol, behaviour and gate level implementation of an S-element[1].

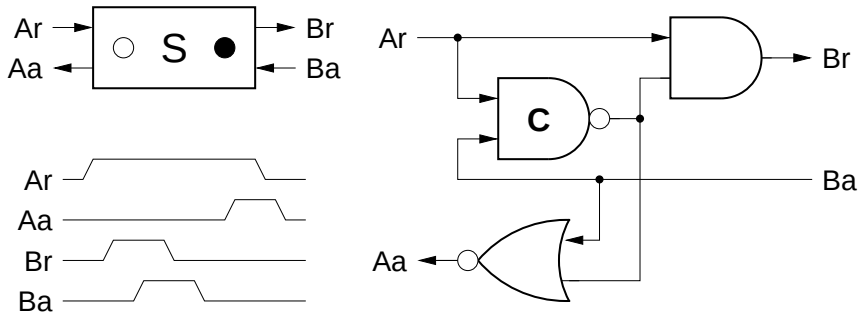


Figure 1.6. The S-element

Embedding one handshake within a phase (or across a number of phases of) another handshake is known as *enclosure.* Enclosure is essential to the construction of handshake circuit control components as it allows a handshake to represent the activity of a task using the re-

[1]The empty and filled circles on the handshake ports identify 'A' as a *passive* port and 'B' as an *active* port. This distinction is explained in §3.3.

quest to start the task and the returning acknowledgement to signal its completion. The S-element's role is to allow full 4-phase handshakes to be enclosed within the request rising to acknowledgement rising phase of another handshake. The rising acknowledgement of this enclosing handshake can then be used to stimulate another S-element, so forming a chain. The 4-phase handshakes on the 'B' ports of these chained elements will thereby be sequenced without overlapping. S-elements can also be used to issue concurrent, independent 4-phase handshakes by sourcing the 'Ar' inputs of a number of elements from a single request signal. A tree of C-elements can then be used to gather the 'Aa' signals to form an acknowledgement to that source request.

1.2. Thesis Structure

The remainder of this thesis is divided into seven chapters:

Chapter 2. Asynchronous Synthesis

This chapter presents existing asynchronous design methodologies and discusses their properties which make automated circuit synthesis possible.

Chapter 3. Handshake Circuits, Tangram and Balsa

Handshake circuits form the basis for the synthesis method used by Balsa. The extent of the Balsa system (and the Philips Tangram system to which it is related) as described by my M.Phil. thesis [6] is presented in order to separate improvements made to Balsa by this work from previous Balsa development. Chapters 2 and 3 form the background/previous work descriptions.

Chapter 4. The Balsa back-end

The Balsa system described previously omitted a means for producing circuits from handshake circuit intermediate descriptions. The design and implementation of such a back-end is described here. The improved use of the LARD asynchronous modelling language as a Balsa simulation engine is also described.

Chapter 5. New Handshake Components

New components are proposed to improve Balsa synthesis. The impact of these components on speed/area efficiency of synthesised designs is described qualitatively here and quantitatively in chapter 7.

Chapter 6. The AMULET3i DMA Controller

The design and implementation of the AMULET3i DMA controller is presented. This controller is offered as an existence proof of design flow for Balsa and represents a reasonable sized design challenge.

Chapter 7. Evaluating the New Handshake Components

A second, simpler, DMA controller design is presented and used to evaluate the new handshake components described in chapter 5.

Chapter 8. Conclusions

Chapter 2. Asynchronous Synthesis

The majority of large synchronous designs undertaken today include a degree of synthesis. Synthesis frees the circuit designer from the repetitive and error-prone task of hand constructing each part of a circuit from available cell library gates and macros. Synthesis also allows designs to be more quickly revised and retargeted to meet changes in specification and improvements in process technology.

In order to be successful, asynchronous design approaches must offer the designer not only the concrete benefits described in §1.1 but must also allow synthesis in the same way as conventional synchronous design allows. To this end, many of the approaches used to design asynchronous circuits can be (or have been) automated to some degree. This chapter describes a number of of these existing asynchronous design approaches and enumerates their strengths and weaknesses. Some of the ideas embodied in these other approaches, which are not used in handshake circuits, are revisited in chapter 5.

2.1. Design flows for asynchronous synthesis

All synthesis systems share the problems of fitting into existing design flows. As synthesis systems often have their own design entry method, integration is often limited to the realisation of a *back-end* for those systems. This back-end interfaces the synthesis tool with the existing design entry methods of the CAD system used to implement designs. This design entry usually takes the form of importing netlists into commercial CAD tool based design flows. Commercial tools are often the only solution for CMOS implementation as the design kits necessary to describe the target CMOS technology to the CAD system are specific to particular CAD systems and often contain encrypted or hidden information which make it difficult to port those kits onto other systems. FPGA programming is often similarly closed due to unpublished programming bitstream formats. More positive reasons to use existing commercial tools involve making use of commercial synthesisers as back-ends to home-grown systems, using the design management facilities of those systems or integrating synthesised designs with circuits designed using other design entry methods.

Figure 2.1 shows a typical synthesis design flow with the synthesiser, back-end CAD tools and possible design feedback paths shown. Not all of the asynchronous design methods described in this chapter are, strictly speaking, synthesis systems (where this term is taken to refer to automated systems for design implementation) although many of the stages shown in figure 2.1 are applicable to un-automated systems too (albeit with processing performed by hand).

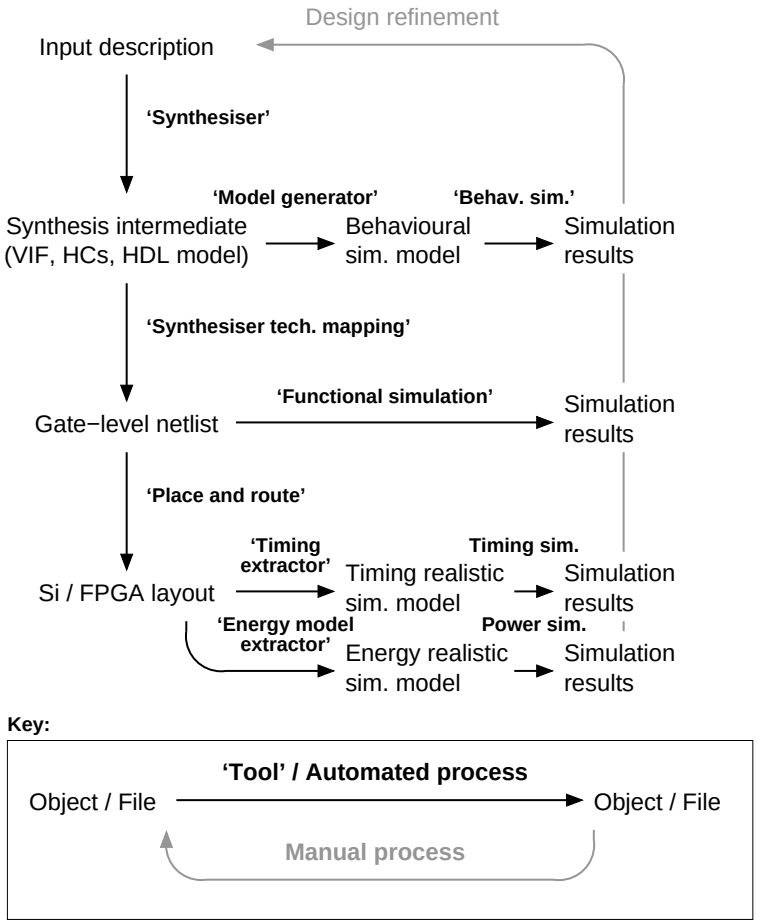


Figure 2.1. Typical synthesis design flow

The design flow starts with the design description. This is entered using the language, schematic or graphical description mechanism specific to the synthesis method. Many systems allow this description to by simulated to allow the design to be functionally validated. A 'correct' design can then be presented to the synthesiser to produce an implementation. Knowledge of the target technology is required to perform this synthesis. Technology specific choices of target cells, timing, area and power trade-offs can also be made. Once implemented, a design can then be simulated again. This simulation can be used to confirm that the implementation behaves in the same way as the original description and also to provide timing-realistic simu-

lation of the implementation.

The implementation can then be committed to CMOS layout or to an FPGA using place and route tools (although place and route is, increasingly, becoming part of synthesis). Once committed to layout, further simulation can confirm that the implementation is functionally correct and obeys timing constraints necessary to ensure reliable operation under temperature, voltage, noise and process variation conditions. Timing extraction tools (or equivalent capacitance and resistance extraction) are used to construct models for this simulation. In parallel with simulation of the implementation, static timing analysis can be used to confirm correct timing operation in all cases (not just those which the simulation covered). Formal verification of the correctness of the implementation process may also be undertaken, although this is not shown on figure 2.1.

At any stage in the implementation, simulation and timing validation of the design, it may be necessary to make changes to the design by changing either the original design description or one of the forms the design takes at later stages of synthesis (right down to the final layout). The implementation process may involve a number of cycles around a design – synthesise – evaluate design loop. The role of user intervention in the synthesis process and the choice of how many of the implementation steps this looping should incorporate is a function of the synthesis method used and the extent to which the user can affect that method's outcome by modifying the input description.

2.2. Directness and user intervention

Directness is a claimed advantage of handshake circuits and other synthesis methods which involves mapping, one-to-one, constructs in the input description into modules of the implementation. Directness allows the user to affect the implementation's properties directly by changing the input description. The output of all synthesis systems is affected by the 'appropriateness' of the input description to that system. Systems which make state encoding decisions (such as synchronous FSMs and the graphical asynchronous state machine approaches described later in §2.4) are often affected by the regularity and symmetry of descriptions which can result in pathologically bad implementations for descriptions which are almost, but

not quite, regular. Direct synthesis methods allow these cases to be avoided.

Unfortunately, directness can also result in larger and slower circuits due to an inability to exploit the tighter implementations possible for circuits which do exhibit symmetry and regularity. Directly implemented control often involves a more sparse state encoding due to the control being dispersed over a number of components. The macromodular synthesis methods described in the next section each suffer from aspects of this inefficiency caused by their direct nature.

2.3. Macromodules and DI interconnect

In order to simplify the timing closure problem, many systems are implemented with delay-insensitive interconnect between units with some internal timing constraints. The units in such systems can be separately layed-out in order to make timing constraints easier to handle. Pre-placed units can then be composed with the interconnect overlayed to produce the entire system. Applying delay-insensitive interconnect at a system top-level leads to the notion of asynchronous on-chip and off-chip buses. This system level use of asynchronous technologies is increasingly becoming a target for system design research as it allows asynchronous techniques to be introduced into otherwise conventional synchronous systems. The MARBLE macrocell bus described in chapter 6 is an example of such an asynchronous system level bus.

Taking the delay-insensitive interconnect approach close to its fine limit for granularity gives rise to the *macromodular* design methodologies (after the macromodules system [66] which used a similar, LEGO™-brick approach to circuit construction). Any design methodology which makes use of handshaking channels and small (although greater than gate sized) modules to construct circuits is here considered to be macromodular. Modules in these systems are usually constructed from small numbers of logic gates and may have internal timing constraints resolvable in layout by locality of placement. These internal constraints are usually QDI or equipotential-region (regions of interconnect which can be assumed to act like a single node, modelled by a single voltage) like in nature. Where data processing is required, either delay-insensitive or bundled data interconnect can be used. Using bundled data interconnect does, however, incur extra complexity in layout or layout timing validation to ensure that

bundling constraints are met.

Macromodular design styles exist for hand construction of circuits and for automated circuit synthesis. Existing systems include: Clark's macromodules, Sutherland's micropipelines, Brunvand's OCCAM-based system and, of course, handshake circuits. The micropipelines approach is presented first as it is a commonly used, relatively recent design style which incorporates many of the common control structures of older systems. Micropipelines are also presented first as the AMULET microprocessors are based on micropipeline ideas.

2.3.1. Sutherland's micropipelines

Micropipelines [67] are predominantly used to construct hand-built circuits. The design style is useful for constructing circuits consisting of pipeline stages with or without processing logic between stages. Asynchronous control of the latches between stages allows them to behave in an elastic manner, allowing data to flow straight through FIFO stages when the pipeline is empty and ripple through (being latched) when the pipeline is busier.

Pipeline stages are constructed using either completion signalling in data operations or by the use of bulk delays. The use of such delays is shown, in parallel with the functional unit, in figure 2.4. These stages can then be combined by connecting their incoming/outgoing channels. Signalling is performed using 2-phase handshakes with bundled data. Micropipeline macromodules act on signals, not on request/acknowledge channel signalling pairs. Channels only become well defined at a higher level of abstraction where micropipeline stages are composed.

Channel forks and combines are the simplest control structures used in Micropipelines and can be implemented using the elements shown in figure 2.2. These are the common macromodules of micropipelined systems and include the familiar Muller C-element for signal synchronisation and the XOR gate used as a signal merge component. The basic control elements used in Micropipelines are:

XOR gate and C-element – XOR gates are commonly used to implement OR functions for events (often known as a merge operation) to combine two mutually exclusively transitioning 2-phase control signals into one. C-elements are used in Micropipelines to implement a

signalling join (synchronisation) operation. This synchronisation is an AND function applied to both up and down-going signal transitions.

call – Call allows channel multiplexing based on request signal arrival order. Requests on the call element's inputs ('R1' and 'R2') are forwarded to the request output 'R'. Acknowledgements to the request output are carried back to the acknowledgement corresponding to the requesting input ('D1' and 'D2'). A call must receive mutually-exclusive requests to function correctly.

arbiter – The arbiter can be used to provide a decision between possibly simultaneous request inputs. It is often found connected to the call element to form an arbitrated call (often provided as an element in its own right as many optimisations are possible where call follows an arbiter). On receiving requests on 'R1' or 'R2' or both, a decision is made and one request is granted by transitioning either 'G1' or 'G2'. 'G1'/'D1' and 'G2'/'D2' form handshake pairs with a transition on 'D1' or 'D2' freeing the arbiter to service further requests. The arbiter can be a difficult element to implement as it requires the use of a mutual-exclusion element to prevent possible metastability present in the element from propagating to the outputs 'G1' and 'G2' while a decision is made.

select – The select component is used to direct an incoming signal to one of two outputs in the manner of a control demultiplexer.

toggle – The toggle element emits transitions alternately on the dotted and undotted outputs with a single output transition occurring for each input transition. The dotted output is the first output to emit a transition.

capture-pass latch – Pipeline latches can be built around the *capture-pass latch* a latch structure which incorporates the latch-opening delay as a delay between input and output signal pairs. Data is captured by the latch on receipt of a transition on the capture input and is invalidated (the latch made transparent) by a transition on the pass input.

A common addition to the micropipeline macromodule set is the *decision-wait* [41] (also shown in figure 2.3). Decision-wait implements a 'gate' for a number of input signals under the control of a common 'trigger' signal. Decision-wait implementations for 2 and 4-phase

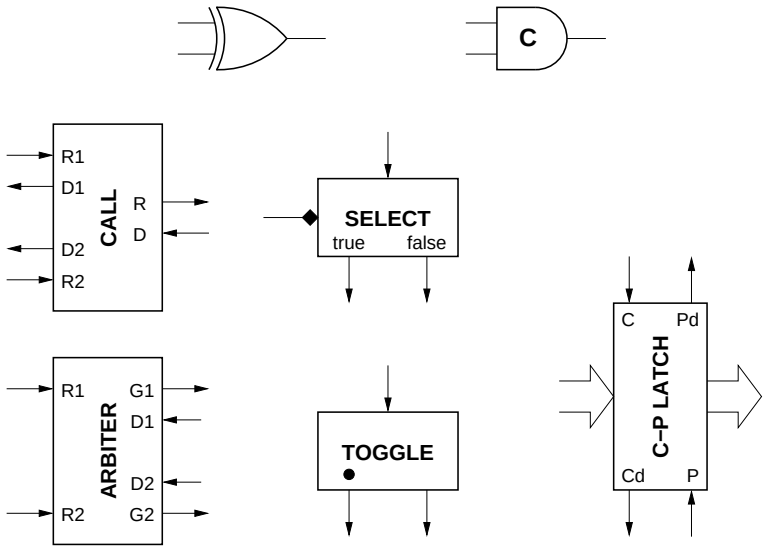


Figure 2.2. Micropipeline elements

signalling are shown in figure 2.3. Notice that the 4-phase decision-wait is just a means to encapsulate an isochronic fork between a number of synchronising C-elements. The 2-phase version is similar but uses two XOR gates to restore the level of the trigger input of the C-element which isn't fired back to its pre-trigger state.

Decision-waits can be used to implement the micropipeline call elements. An XOR gate combines input requests to form the output request and the decision-wait is used to track the input to which the acknowledgement must be returned. The fork necessary to implement this choice is encapsulated by the decision-wait. This fork-encapsulation is especially useful in similar 4-phase circuits where clean return-to-zero between activations is required. The 2-input, single trigger decision-wait can also be generalised to a n-input (rows) $\times$ m-trigger (columns) device. In this nomenclature, the common n-input decision-waits have an $n \times 1$ geometry.

Much of the detail of micropipelined circuits is concerned with the control of transparent latches. The capture-pass latch is a useful element as it implements the control necessary to open and close a latch under two sources of 2-phase control. The capture-pass latch does this by using both control signals (capture – 'C' and pass – 'P') to drive a latch cell and generate 'done' signals (capture-done – 'Cd' and pass-done – 'Pd') to pass back as acknowledgements. A two stage FIFO composed of capture-pass latch stages is shown in figure 2.4. This FIFO

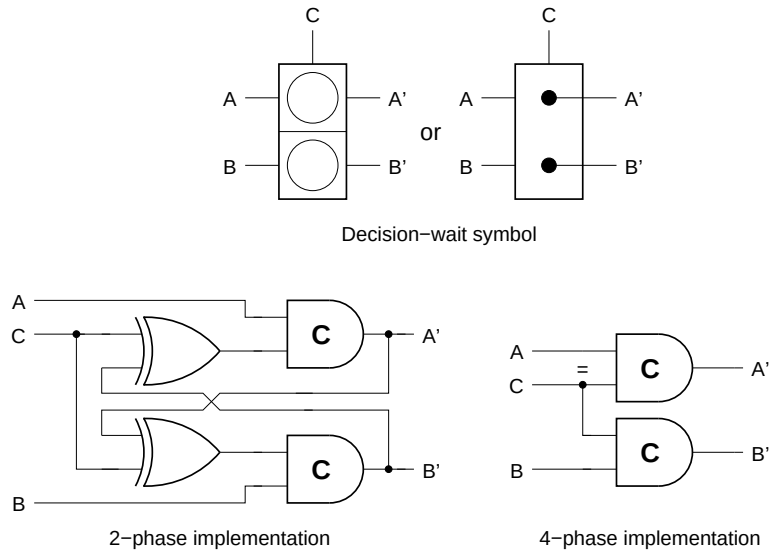
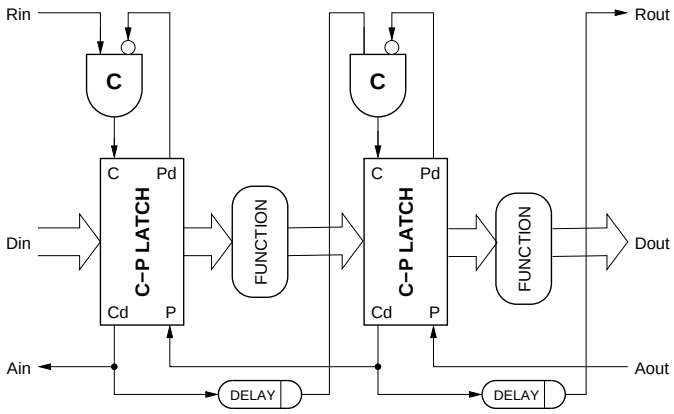


Figure 2.3. Decision-wait element and implementations

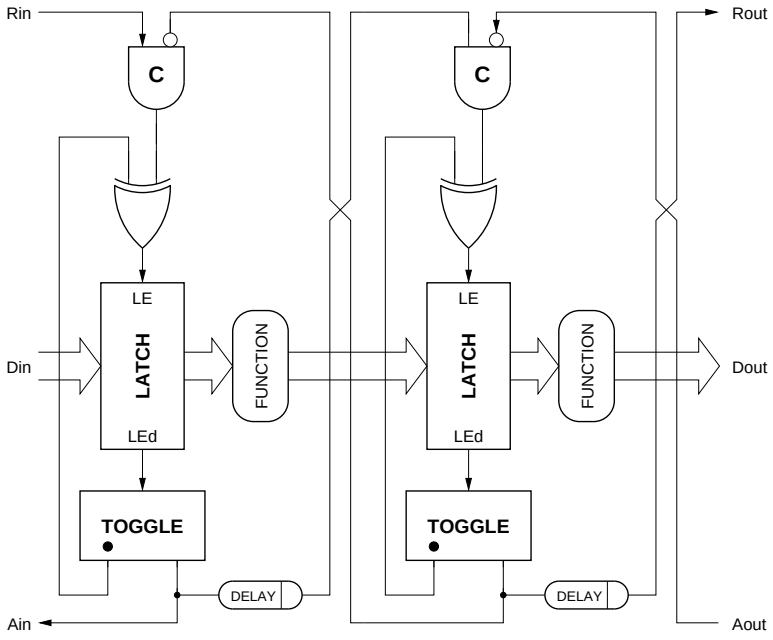
shows the general push driven nature of data movement in a micropipeline and the use of inverted input C-elements to provide circuit initialisation. A similar latch controller (also shown in figure 2.4) can be implemented using a conventional transparent latch with the use of a merge and toggle to generate a latch transparent signal ('LE', the 'LEd' signal is the 'LE' delayed by driving the latches) from the capture and pass signals of the first controller.

Pipelines may be forked and joined either conditionally (multiplexing/demultiplexing) or with forked paths coming back together in a synchronisation (the more usual use of the terms fork and join). Simple channel forks may be implemented with a C-element and a wire fork. Demultiplexing (forking with only one destination channel taking the data, completing the communication) can be implemented with a select element and a merge. Channel joins/combines are possible with merge or C-elements in the request path. Where a channel is forked, delays in the control paths may be necessary to ensure that the bundling constraints of control and the demultiplexing of forked data portions of the channel are met. The use of bundled data in the pipeline stages themselves also makes timing closure more difficult to achieve. In some cases, as Micropipelines are commonly used to describe processing pipelines, the physical layout of adjacent pipeline stages may make timing constraints easier to meet. The fork and C-element structures are common in other design methodologies which use channel communications.

Sutherland's original Micropipelines made use of 2-phase control signalling. 2-phase sig-



Capture−pass latch based FIFO



Transparent latch based FIFO

Figure 2.4. FIFOs with capture-pass and transparent latches

nalling makes the construction of pipeline latch controllers conceptually easier as considerations of return-to-zero phases in control do not arise. The complexity of individual macromodules is greater for 2-phase signalling solutions than for simple 4-phase implementations. This added complexity is due to the need to maintain state in those macromodules to track the levels of output signals between handshakes. 2-phase signalling also makes the implementation of pull channel transactions impractical by requiring extra latches to be introduced to hold data values between handshakes. In similar 4-phase implementations, the request return-to-zero can be used to signal to the data source that the data has been taken and may subsequently

be removed from a channel. Pull channels find applications in bus design [5][52] and in DI implementation of channel forks. In Micropipelines, these applications may require either isochronic forks or the breaking of the 'rules' of the design style.

When using 4-phase signalling with Micropipelines, the merge element is implemented with an OR gate, the select element becomes a simple demultiplexer and the toggle element becomes largely redundant. Latch control also becomes a major concern with 4-phase signalling. Controllers using different 4-phase protocols, varying degrees of parallelism of the input and output handshakes and other tricks are possible [29][63] and need to be chosen for particular pipeline applications.

2.3.2. Macromodules

Macromodules [66] were developed by Clark at Washington University during the late 1960s as a system for constructing large digital circuits which were composed of pre-built blocks of asynchronously communicating functional units – macromodules. Circuits built using these blocks needed no timing validation. The blocks themselves and interconnecting cables were built to preserve bundling constraints between data and control. Data was encoded in binary with two control signals: initiation (request) and completion (acknowledge). Pulses on these control signals constituted request and acknowledge events.

The macromodules were physically constructed using rack mounted, plug in modules built using off-the-shelf SSI logic gate ICs. Interconnect consisted of connections made by the juxtaposition of the modules in the rack and the addition of patch cables to join one macromodule to another. Data connections had a fixed width of 12b which could be extended by the use of additional macromodules by connecting the internal carry using the rack-based interconnect. The control structures possible using macromodules are very similar to those of micropipelines. Merge and rendezvous elements combine signals in the same ways as XOR gates and C-elements do for transition signalling. A call element also exists and has the same behaviour as its micropipeline successor. Data is communicated by push handshakes with control gating of data possible using a *data gate* module. The data gate provides 4 handshake ports: a pair of data inputs, one for input data to be gated by this module and the other which passes one input data ungated (this input can be used to cascade data gates to produce control

driven data multiplexers), a control input which enables the transfer and a data output. A data gate connected to a register write is very similar in structure to a micropipelined control/data synchronisation feeding a latch.

The data gate also serves a similar function to the transferrer component in handshake circuits (described as the Balsa Fetch component in §3.5.1). The main difference is that a transferrer pulls its input data from the sourcing module rather than allowing for data communications to drive inputs (such as latch write ports directly). In control terms, micropipelines represent a more dataflow-like approach, with control following data. Macromodules encourage gating of flowing data using the data gate module, similar gating in micropipelines must be built with ad-hoc synchronisation. Handshake circuits are more control driven with the transferrer forming part of most communications.

2.3.3. Brunvand's OCCAM synthesis

Brunvand's macromodular synthesis [13] system makes use of the channel-based, CSP-like programming language OCCAM [38] to describe circuits. Descriptions are automatically synthesised into compositions of control, variable read/write and datapath macrocells implemented with 2-phase signalling with bundled data. The connections between components are expressed as separate request, acknowledge and data signals with components placed directly to form final, gate level circuit netlists. Peephole optimisation of those netlists is used to trim unused components. The boundaries of the macrocells are used to restrict optimisations from removing hazard-eliminating redundancy from designs.

2.3.4. Plana's pulse handshaking modules

Plana [61] describes a system of macromodules for constructing circuits using pulse-mode handshaking. Pulse-mode signalling has been used in other systems in early asynchronous systems such as Macromodules and MU5 [53], an asynchronous mainframe developed at Manchester University. Pulse-mode signalling has, more recently, been used to implement arbiters [35] and in systems where pulsed operation is more appropriate to the implementation technology, such as Josephson junction based RSFQ logic [55]. Plana's example circuits built using the pulse-mode components are hand constructed but a pseudo-code is used to describe

circuits' specifications which could be used as the basis for a synthesis system.

The macromodules themselves are described using petri-nets with signal pulse labelled transitions. These petri-net descriptions are flow-table synthesised into gate level implementations. Protocol adapters allow the more usual 2 and 4-phase handshaking circuits to connect to ports of pulse macromodules. In construction, pulse handshaking combines the conceptual simplicity and lack of redundant phases of 2-phase design with a quiescent state which can be determined by signal levels (request low, acknowledge low) which 4-phase designs enjoy. Sequencing control components can be implemented with just wires and merging of signals requires only an OR gate. Unfortunately, a join element requires three C-elements in its construction where only one is necessary in 2 or 4-phase signalling. Optimising away joins becomes a more important optimisation than may be the case with conventional handshaking.

Timing validation for pulse-mode circuits must ensure that pulses are wide enough to be observed by destination macromodules after inertial interconnect delays are taken into consideration. Pulses must also be suitably separated to prevent delays from coalescing adjacent pulses. These constraints could be difficult to ensure in newer, sub-μm, CMOS technologies where the ratio of interconnect delays to gate delays is rapidly changing.

Plana additionally describes some improved sequencers for use in 4-phase level-sensitive signalling. These improvements concentrate on overlapping the return-to-zero phases of adjacent sequenced operations. Conflicts caused by these overlappings can be resolved by including extra gates to re-introduce the complete sequencing on a case-by-case basis.

2.4. Other asynchronous synthesis approaches

Macromodular circuit composition is not the only commonly used approach to the design of asynchronous circuits. Methods which synthesise asynchronous descriptions directly to conventional logic gates or transistor level implementations also exist. Amongst these are the use of classical asynchronous state machines, petri-net/STG based synthesis and burst-mode asynchronous state machines. These approaches require the user to describe the circuit's behaviour in terms of individual signal transitions using state-diagram-like input descriptions. These approaches will henceforth be referred to as the graph based approaches.

Graph based approaches can be unnecessarily complicated to work with for large circuit descriptions. Language based interfaces which use graph descriptions as an intermediate representation make it much easier for existing, synchronous industrial designers to make use of asynchronous design (as well as increasing productivity for experienced asynchronous designers).

Two other approaches are described below: NULL Convention Logic (NCL) and Communicating Hardware Processes (CHP). NCL is a method for describing DI implementation of functions using threshold logic gates in a way which allows the simple addition of pipelining to functional units (much like in Micropipelines). CHP is a CSP-like language for which a synthesis method exists to produce QDI implementation by language level design refinement.

2.4.1. 'Classical' asynchronous state machines

The formalisation of asynchronous methodologies began with synthesis of asynchronous state machines. Machines were described using the same kind of state diagram notation used for synchronous Mealy state machines and were implemented with conventional, boolean logic gates. Methods such as flow-table synthesis [72] exist to derive implementations from state diagrams. These asynchronous methods have the same kinds of computationally expensive problems of state encoding and logic optimisation common in synchronous state machine synthesis.

Asynchronous state machines work by generating output and 'new' state signals as functions of the machine's inputs and 'old' state signals. The difference between asynchronous and synchronous state machines is that the new state signals are connected directly to the old state signals without intervening latches resulting in blocks of combinatorial logic with feedback. The lack of latches and a clock to provide discrete points of state change means that a transition from one state to another may also pass through many intermediate, transient states with one or more state or input bits changing between states. The implementation of such a machine must allow all possible paths from one stable state to another to be allowed but without causing hazards on the circuit's outputs. For example, for an output of a machine which is held high in two states, a transition between those states must not be allowed to pulse that output low while switching from one sum term of that output's function being true to another. Redundant

terms must be included to maintain that output during the state transition.

The (required) redundancy introduced to prevent hazards may be removed by zealous gate level optimisation. This makes the use of the function-preserving logic optimisation methods used for synchronous state machine synthesis unsafe. In order to keep the complexity of the state assignment and state change/output hazard elimination under control (in order to make synthesis computationally tractable), a number of behavioural assumptions are commonly imposed on the behaviour of the environment in which these circuits are placed. The most common two assumptions are the *fundamental-mode assumption* and the *single input change assumption.* The fundamental-mode of operation assumes that no input signal changes occur while the state of a machine is settling after a previous period of input change activity. The single input change assumption requires that input changes occur one at a time in order to reduce the number of paths between stable states. The two assumptions are often used together with classical asynchronous state machines, this restricts such machines to single input changes separated by enough time for internal state to settle. The single input change assumption can make this class of circuits difficult to work with as the function of many asynchronous controllers is to coordinate asynchronously arriving signals which, obviously, may occur simultaneously.

The difficulties of working with classical asynchronous state machines and with the tractability of the synthesis route have led to a decline in their use as a circuit description approach. This is evident from the decline in the teaching of the approach to engineering students. The design of asynchronous state machines formed a common part of digital design textbooks in the 1960s and 1970s as a method for constructing latches and flip-flops. By the late 1970s and 1980s, most such textbooks instead concentrated on the practicalities of synchronous design and the use of the new VLSI technologies. The use of burst-mode and petri-net synthesis have largely supplanted them amongst asynchronous design advocates.

2.4.2. Petri-net synthesis

Petri-nets were devised as a formal, graphical means for describing the behaviour of concurrent systems ([12] makes some interesting comparisons of petri-nets with other concurrency formalisms). Their use in circuit design makes it easier to describe systems which are concur-

rent at a very fine level (when used to design signal level circuits). This kind of concurrent operation can be more difficult to express in state diagrams without drawing multiple diagrams or resorting to expanding the Cartesian product of the states of all the concurrent portions of a design.

A petri-net consists of *places* and *transitions* connected together by directed arcs. State is represented by the flow of tokens around a system with the tokens residing in the places between actions. Each directed arc connects a place to a transition or a transition to a place. Place to transition arcs provide tokens for transitions which may allow that transition to *fire*. Transition to place arcs carry tokens away from firing transitions towards other transitions.

A transition may fire when all the arcs leading to it come from places which hold at least one token. A fired transition will emit a token into each of the outgoing arcs from it and so into each of the places connected to those arcs. Tokens are not necessarily preserved by the transition firing action: the use of multiple output arcs from a transition allow tokens to be replicated (a fork). Multiple outputs from a place allow a choice of actions between the target transitions (an arbitration or choice made by another place's token) and multiple inputs to a place allow merging of token paths. Transitions are usually annotated with actions to take place when that transition fires or a condition (in addition to the presence of a full set of tokens) which must be satisfied for a transition to fire.

Table 2.1 shows five ways of using petri-net fragments to describe common control structures. Fork and join can be used together to perform a number of parallel paths through the net which, synchronise and join back into a single path. Choice and merge can be used in a similar way to allow a choice between paths. Drawing all the places for petri-nets which are composed mostly of single input/single output places becomes tedious. For this reason, places are often omitted where they are of little interest. Petri-net fragments for this abbreviated form are shown in the second row of figures in table 2.1.

Places may, in general, contain zero or more tokens although the subset of petri-nets which only allow 0 or 1 tokens to occupy a place are particularly useful. These are called *safe* petri-nets.

The position of tokens in a petri-net is called the *marking* of that net. At the start of its life,

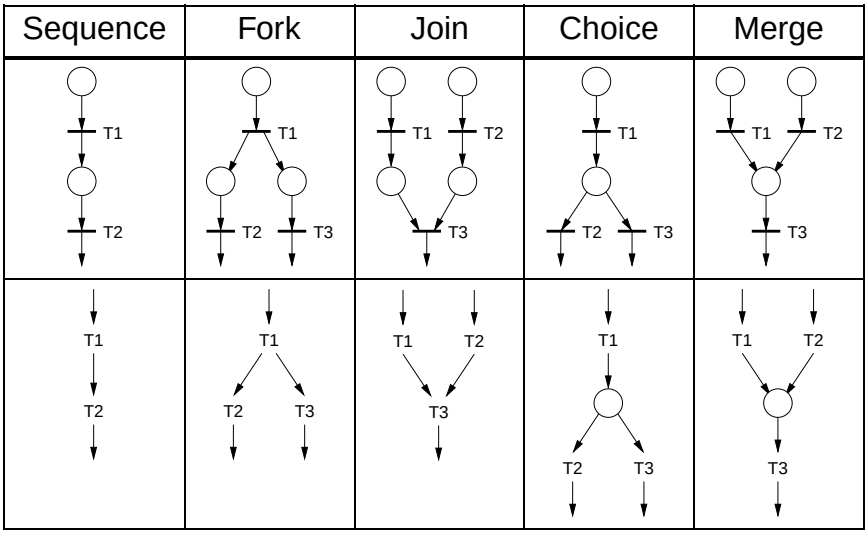


Table 2.1. Petri-net fragments

a net has an *initial marking* of tokens in places representing the 'reset' state of the system. Figure 2.5 gives an example petri-net (using the abbreviated form of petri-net diagram) with 2 tokens in its initial marking. This petri-net represents the 2-input decision-wait element described in §2.3.1, it is safe and has level changes on input and output signals annotating its transitions. Transitions with output signal changes[1] are actions which the circuit must perform, transitions with input signal changes are firing conditions for those transitions.

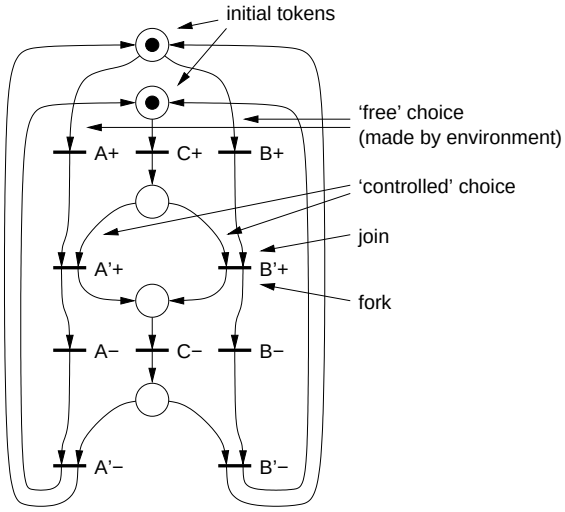


Figure 2.5. 2-way decision-wait petri-net

[1]To differentiate between signal transitions and petri-net transitions or state transitions, the term *signal change* will be used to refer to a change in the level of physical, wire borne, signal.

This type of signal-transition-describing interpreted petri-net is known as a Signal Transition Graph (STG) [17]. STGs are the basis for the circuit synthesis approaches used by the tools SIS [65], Petrify [19] and CASCADE [7]. Many of the synthesis algorithms provided with SIS will only synthesise a subset of these descriptions which contain no choice. Petrify, on the other hand, can synthesise safe nets with choice and also nets with boolean transitions guards in addition to signal transitions. CASCADE takes descriptions in a extended form of STG known as a generalised STG (gSTG). CASCADE makes use of the tools Petrify and 3D (a burst-mode tool, burst-mode machines are described in the next section) as synthesis back-ends to produce circuits from transformed versions of its input gSTG descriptions.

The complexity of petri-net synthesis lies mainly in the allocation of state signals to represent the state of the signals and marking in order to distinguish markings which have the same input signal pattern. This state encoding can be derived by hand by inserting extra transitions into a net and annotating those transitions as signal transitions on *internal* signals. These internal signals are the feedback signals making up the state holding in that net's implementation. The intention in adding transitions for these signals is to partition the circuit into regions in which like input signals patterns in states with different outputs/successor states can be distinguished by differences in the state of internal signals. The automation of this *complete state coding* makes the use of petri-net synthesis as an intermediate form of circuit description for higher-level synthesis systems possible.

Petri-nets have been used by Kilks, Vercauteren and Lin [45] to reimplement control in handshake circuits. Fragments of petri-nets representing the control paths through the handshake circuits control components were composed and the resulting petri-nets presented to a petri-net synthesis tool for state assignment and circuit implementation. Only the handshake channels at the root and leaves of the tree were left intact by this mapping process. This form of optimisation of a cluster of handshake components into a more tightly state encoded form (with the reduced area and possible performance increase this implies) is suggestive of the form of handshake circuit optimisation methods presented in chapter 5.

2.4.3. Burst-mode machines

Burst-mode machines [54] are very similar to classical asynchronous state machines. Beha-

viour is described with a state diagram annotated with input change/output change signal changes on the state-to-state transition arcs. Figure 2.6 shows the 2-way decision-wait element described as a burst-mode machine. The double ringed state S1 is the initial machine state. States S2 and S3 reachable by raising C together with either A or B respectively. Notice that more than one input signal change can be part of the input signal change of a state-to-state transition. More than one output change can occur at each state transition (although this is not shown in the given example).

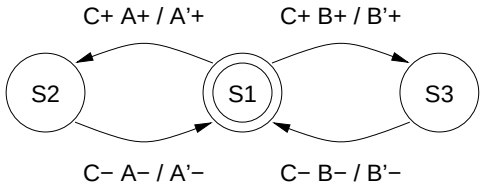


Figure 2.6. 2-input decision-wait burst-mode machine

The concurrent *bursts* of input or output changes at state transitions give this form of state machines their name. The signal changes making up a burst may occur in any order allowing a single input burst to express the input event reordering which is so often required in asynchronous machines. Unfortunately, this is the only form of concurrency available with burst-mode machines unless a number of machines are connected together. A back-end for Balsa which maps onto burst-mode machines has been developed by Chelcea [16]. This back-end amalgamates streams of sequenced control into single burst-mode machines but requires a fork/join structure and a separation into multiple machines to implement concurrency.

Two popular tools exist to implement burst-mode machines: 3D [75] and MINIMALIST [26]. Both tools produce fundamental-mode implementations with SI timing assumptions. Fundamental-mode operation places a requirement on the environment to not respond to output changes by changing circuit inputs until the internal (fed back) state of the circuit has settled. This mode of operation does, however, make state coding easier by not having to consider state changes due to input change whilst the internal state of the machine is settling. Fewer restrictions are, as a consequence of this simplification, placed on the optimisations which may be performed on an implementation without compromising hazard safety.

Simplifying the state coding problem makes larger (sequential) descriptions tractable in burst-mode machines than in STG synthesis. It could be argued that when partitioning a specific-

ations into communicating burst-mode machines, the state coding problems is replaced with the problem of picking where to place the transitions on the wires connecting the burst-mode machines together. A recent synthesis system, ACK [46], uses burst-mode machines for synthesis with automatic partitioning of the input description.

2.4.4. Communicating Hardware Processes – CHP

CHP is another CSP-like language. Its main distinguishing features from macromodular systems like Brunvand's OCCAM synthesis (§2.3.3), Tangram and Balsa are the use of language level signalling expansion and the *probe* primitive to make implementable circuit descriptions from user entered descriptions.

Signalling expansion allows the synthesiser to re-shuffle handshakes to allow the choice of protocol to be varied on a communication-by-communication basis in order to produce a better implementation. The probe allows the value of an input signal to be sampled in order to implement choice between input communications. The probe can also be used to write descriptions which use individual signals (rather than handshake channels) to form communications.

Burns [14] describes an early CHP-like language with one-hot encoded data and the probe construct. Martin goes on to apply his *production rule* based transistor level synthesis approach to the implementation of CHP descriptions [50]. CHP has been used to implement a number of substantial test designs [62][51]. More recently, Tanner [69] has begun developing commercial CAD tools using the synthesis methods used with CHP.

2.4.5. NULL Conventional Logic – NCL

NULL Convention Logic was devised by Fant [24] and is promoted by Theseus Logic Inc. [70] as a complete design solution for designing delay-insensitive asynchronous circuits. The NULL convention in the name is the use of the quiescent state (usually with all signals of a communication low) to separate token in data communications and control handshakes. Data is encoded in a delay-insensitive fashion using dual-rail or 1-of-4 encodings using 4-phase encodings to allow the NULL state to be discerned by signal levels rather than transition history. Control is also 4-phase. The same completion signal construction costs which are present in

other DI systems are also present in NCL (and are similarly expensive to implement).

What distinguishes NCL from other DI approaches is the use of threshold gates to implement logic. Threshold gates are logic gates which have outputs which fire when the sum of the weights on active inputs exceeds a fixed gate threshold. For threshold gates with weights of 1 on all inputs, the threshold is just the number of inputs to be active for the output to fire. For example, a 2-input OR gate is a 2-input threshold gate with a threshold of 1 and a 2-input AND gate is a similar threshold gate with a threshold of 2.

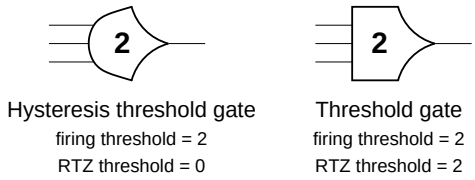


Figure 2.7. NCL threshold gates

Threshold gates can be generalised to include two thresholds: the firing threshold and the threshold below which the output will return to zero after firing. NCL implementations are made up of such gates but with fixed return-to-zero thresholds of zero. These are known in NCL's terminology as *hysteresis threshold gates.* The symbols used by NCL for these components are shown in figure 2.7. Hysteresis threshold gates make it easier to build circuits in which the NULL state is propagated but where gates hold their state while their inputs change from fired to NULL states. A 2-input, threshold of 2 hysteresis threshold gate is equivalent to a 2-input C-element. Note that replacing plain threshold gates in a combinatorial circuit with hysteresis gates results in a circuit with the same logical function (e.g. AND gates to C-elements) but with latching present. Hysteresis gates can be more expensive to implement than plain threshold gates although it is easy to make use of the storage inherent in the gates to form pipelined processing stages with a little extra control.

Synthesis of NCL circuits from logical descriptions can be performed by mapping two level boolean implementations of those functions into minterms implemented with C-elements and OR gates to implement AND and OR planes using DIMS [37]. The C-elements and OR gates of DIMS can then be mapped onto their threshold gate analogues. The next stage is the difficult, currently unautomated, part of the NCL synthesis path. Simple hysteresis threshold gates can be optimised into threshold gates with more complicated input weightings. These

optimisations are not easily automated and are the subject of further research by Theseus. Figure 2.8 shows a typical, fully optimised, NCL implementation of a (dual-rail) adder cell. Notice that more than one input of the 5-threshold gates is connected to the same signal so forming weighted inputs.

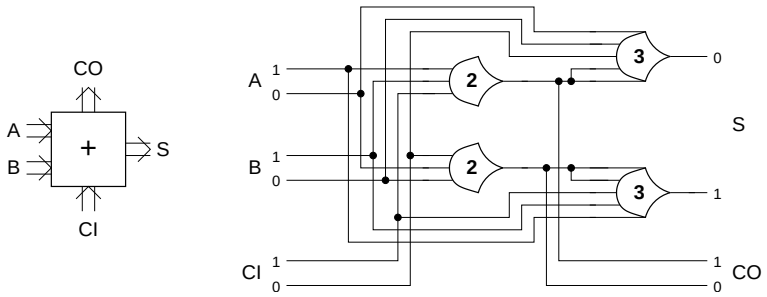


Figure 2.8. NCL hysteresis threshold gate adder

A large number of possible hysteresis and plain threshold gates are possible with varying numbers of inputs and threshold values. Only a subset of these gates are easily implementable as CMOS gates due to the limits of transistor stack sizes in modern, low V_{dd} CMOS. In optimising threshold gates of an implementation, these small, implementable gates must be considered the technology mapping target. In technologies such as FPGA's, larger gates may be possible or it may be advantageous in either technology to produce larger gates by expansion into boolean functions with feedback (such as might be generated with flow-table synthesis). This expansion compromises the delay-insensitivity on the NCL approach albeit using similar SI or QDI assumptions that may be applied in macromodular approaches.

2.5. Chapter summary

A number of different approaches to the design of asynchronous circuits have been presented. Each of these approaches either uses or can make use of automated synthesis to generate circuit implementations. Some features of these systems are already used with the Handshake Circuits implementation approach – such as macromodular construction and peephole optimisation. Others have been applied to the optimisation of certain components or clusters of components – such as petri-net and burst-mode machine synthesis.

Chapter 3. Handshake Circuits, Tangram and Balsa

The Balsa system uses the *Handshake Circuits* asynchronous macrocell based design paradigm as an intermediate representation for synthesised Balsa designs. Handshake circuits were created for use in the synthesis of the language Tangram created by Philips Research. They are intended to complement design description styles based around the use of synchronous[1] channel communications in the manner of CSP [36].

This chapter is intended as an introduction for readers unfamiliar with handshake circuits and Balsa. As such, it is a slightly disjointed collection of sections covering the topics:

- The Balsa language.
- The handshake circuits paradigm.
- The structure of handshake components (the components of handshake circuits).
- A notation for describing handshake component behaviours.
- The Balsa handshake component set.
- Some aspects of Balsa compilation.

A fuller explanation of Balsa synthesis is available elsewhere [6], as is a more formal introduction to the original handshake circuits methodology [9]. A longer description of the Balsa language can be found in the Balsa User Manual [20] and (for comparison) the Tangram language description can be found in the Tangram Manual [64].

3.1. Handshake circuits

Handshake circuits combine macromodular design with delay-insensitive communications to produce a design methodology in which entire designs are described using macromodules connected together by asynchronous communication channels. Each *handshake component* (macromodule) in a design is an instance of a library cell which may be parameterised to a

[1] in the sense of both sender and receiver being synchronised by a communication

limited degree. There are only a small number of such cells defined in a particular *handshake component set*.

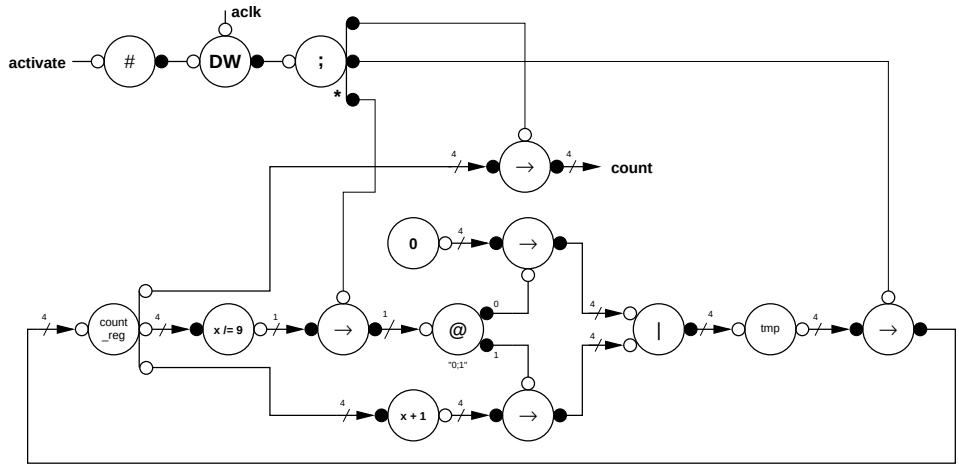


Figure 3.1. A simple handshake circuit (a modulo-10 counter)

3.2. The Balsa language

The Balsa language was created to provide a source language for compiling handshake circuits. For this reason, it is very similar to Tangram. Circuits are described by *procedures* which contain descriptions of processes initialised by an activation port. Procedures communicate with other procedures by means of handshake ports. Communications between parallel composed procedure instantiations are combined using the channel combination components described in the previous section. Most procedures consist of a body command whose behaviour is perpetually repeated using a `loop` ... `end` loop. This is the source Balsa description for the counter example given in figure 3.1:

```
import [balsa.types.basic] -- 1
public

type C_size is nybble -- 2
constant max_count = 9

procedure mod10 (sync aclk; output count: C_size) is -- 3
local -- 4
  variable count_reg : C_size
  variable tmp : C_size
begin
  loop -- 5
    select aclk then -- 6
       if count_reg /= max_count then -- 7
```

```
            tmp := (count_reg + 1 as C_size) -- 8
         else
            tmp := 0
         end ;
         count <- count_reg ; -- 9
         count_reg := tmp
      end
    end
  end
```

This example contains examples the use of most of the Balsa features necessary to read later examples:

1. Pre-compiled module inclusion. In this case `[balsa.types.basic]` only defines some common types: byte, nybble … The `public` keyword separates imports and file-local definitions (delimited by the keyword `private`) from those exported from this modules.
2. Type and constant declaration.
3. Procedure declaration with sync and output ports.
4. Local variables/latches.
5. Indefinite repetition with `loop`. Once activated a `loop` never terminates.
6. Passive input enclosure using `select`. The commands inside the `select` are enclosed in the handshake on `aclk`, `aclk` is effectively the activation for these commands.
7. `if` … `then` … `else` … `end` statements.
8. Assignment, expressions and type casting.
9. Output synchronising communication. An input communication looks like this: `channel -> variable`.

Other Balsa features are explained as necessary where descriptions using those features are used. A short reference guide to Balsa syntax and semantics is given in appendix 1.

3.3. Handshake components, ports and channels

Each handshake component has one or more *ports* with which it can be connected point-to-point to a port of another handshake component by a *channel*. Figure 3.1 shows a simple handshake circuit composed of *n* handshake components linked by *m* channels. Each channel carries request and acknowledgement signalling as well as an optional data payload with

requests flowing from the *active* component ports (filled circles) towards *passive* component ports (open circles). Acknowledgements flow in the opposite direction to requests in the usual fashion. Where a channel carries data, the direction of that data is indicated by an arrow on that channel's arc. Note that the direction of data may be different from the direction of signalling to support push and pull port and channels (all of these terms are defined in §1.1.1).

To complete the channel/port terminology:

- A *sync* or *nonput* channel/port is one which carries no data.
- The *direction* of a port is the direction of data on the channel connected to that port with respect to the port's component. Valid directions are *input* and *output*.
- The *sense* of a port is the signalling direction on that port. *Active* ports source requests on channels connected to them and *passive* channel receive requests from their channels. The combination of sense and direction on ports gives rise to four different port natures for data bearing ports: active input, active output, passive input and passive output. Passive input and active output channels can be connected using only push channels. Active input and passive output channels can be connected using only pull channels. Nonput ports also have a sense, giving the port natures: active sync and passive sync.
- The *type* of a channel/port is the data type to which values communicated via that channel/port are members. For most purposes, the type of handshake component ports are considered to be just simple bit vectors.

Although the handshake circuits in this thesis are implemented using the bundled data convention, it is not necessary to provide an explicit request (or acknowledgement in the case of pull channels) where that event is coded in the data on a channel (as is the case with the delay-insensitive codes introduced in §1.1.3). In particular, earlier versions of the Tangram design flow targetted dual-rail implementations.

3.4. Notation

Each of the handshake components described in the next section is accompanied by a description of that component's behaviour. This behaviour is expressed in a new notation similar to van Berkel's *handshake circuit calculus.*

Van Berkel's handshake circuit calculus describes the behaviour of components in terms of the sequencing, concurrency and enclosure of their handshakes. The ports of a component are assumed to be receptive to incoming requests at all times and as such, under the assumption of a DI environment, where bursts of inputs or outputs may validly occur in any order. These assumptions make the handshake circuit calculus convenient for constructing succinct, valid behaviours for handshake component but do not allow subtleties of implementation such as precise enclosure and concurrency semantics to be expressed. The calculus allows for many implementations, handshake expansions and refinements to a description to be derived from a single description and as such provides a minimum degree of specification for handshake components. Component behaviour must be described in other ways (trace descriptions, STGs/petri-nets, state graphs) in order to allow implementation details to be expressed.

This new notation is introduced to allow the overlappings of handshakes required to implement handshake components efficient to be explicitly expressed. For example, to allow the distinction between situations where handshakes are concurrently composed and are independent of each other and cases where those handshakes have synchronised return-to-zero phases. This new notation is, like van Berkel's calculus, directly mappable onto 4-phase refined (signalling ordering/interleaving expanded) descriptions of those components. Although van Berkel's notation is intended to be mapped into a refined implementation by less direct means. Unlike van Berkel's notation, data communications are included in the notation in a more complete way (van Berkel [9] tends to use Tangram-like descriptions rather than the handshake circuit calculus to express data operations).

This short introduction to the new notation will concentrate on the expansion of terms in the notation where chosen handshaking protocols are assumed on the various channels.

3.4.1. Term expansion

To produce an implementation, a choice of handshake protocols must be made and term expansions for those protocols applied to terms of the notation. Each term expands into two handshake phases: the up phase, indicated by the operator '$\triangle$' and the down phase, indicated by the operator '$\triangledown$'. A complete expansion of a description becomes a complete circuit behaviour by sequentially composing up and down phases to form a whole handshake. The simplest

expansion is for an active sync communication, denoted by the name of the port/channel on which the communication takes place. For example, a 4-phase phase expansion (as a trace description, albeit with the assumption of a receptive environment and the reorderability of input/output bursts) of an active sync is (channels denoted by lower case letters, commands by uppercase letters):

$\triangle$(c) $= c_r \uparrow ; c_a \uparrow$

$\triangledown$(c) $= c_r \downarrow ; c_a \downarrow$

The operators given in the expansions here compose a trace description of the expanded operations albeit with the assumption of a receptive, reorderring environment. Expansions into petri-net fragments are similarly direct.

Command combining operators also have expansions into two phases. The phases of the combined commands may be composed into parts of either up or down phases of the combined expansion. The three simple, command combining operators (and their expansions in 4-phase broad signalling) are:

Sequencing – A ; B

$\triangle$(A ; B) = $\triangle(A); \triangledown(A); \triangle(B)$

$\triangledown$(A ; B) = $\triangledown(B)$

Concurrency – A || B

$\triangle$(A || B) = $(\triangle(A); \triangledown(A)) \,||\, (\triangledown(B); \triangledown(B))$

$\triangledown$(A || B) = ε (empty, $\triangle$ and $\triangledown$ of A and B are not separated)

Concurrency with synchronised phases – A , B

$\triangle$(A , B) = $\triangle(A) \,||\, \triangle(B)$

$\triangledown$(A , B) = $\triangledown(A) \,||\, \triangledown(B)$

Enclosure of a command inside a passive sync communication (between request and acknowledge) is a very common operation. The handshake circuit calculus allows the choice of expansion to be made where several expansions exist for a particular protocol e.g. a 4-phase broad push handshake may enclose a command like an early handshake (between '$r \uparrow$' and '$a \uparrow$'), like a late handshake (between '$r \downarrow$' and '$a \downarrow$' or by enclosing two, sequenced, portions of the command between both early and late of these event pairs (i.e. '$r \uparrow ; \triangle(C); a \uparrow ; r \downarrow ; \triangledown(C); a \downarrow$'). Using more concurrency, the enclosure could be between '$r \uparrow$' and '$a \downarrow$'. The

new notation uses the differences in the way that phases of subcommands are packed into the phases of the compound command when using the control composition operators to make this enclosure choice. The enclosure (symbolized by the operator ' : ') provided by this notation (in this case, in all 4-phase protocols) is:

$\triangle$(c : C) = $c_r \uparrow ; \triangle(C); c_a \uparrow$

$\triangledown$(c : C) = $c_r \downarrow ; \triangledown(C); c_a \downarrow$

The use of the two parallel operators and the placing of the final down phase of a sequencing operator in the down phase of its expansion allows useful enclosures to be expressed using this 'phase-by-phase' enclosure operator. (c : [B , C] describes a coupled fork operation (as in the component Fork) and 'c : [B || C]' describes two independent parallel operation (as in the component Concur). Similar sequential operations are possible (in 4-phase broad): 'c : [B ; C]' is a sequencing with shared/parallel return-to-zero of 'C' and 'c'; 'c : [B ; C ; skip]' is the enclosure of 'B' and 'C' in the up phase of 'c' with sequenced 'C' and 'c' return-to-zero. Note that ' ; ' associates right-to-left and ' : ' binds tighter than ' || ', ' , ' and ' ; ' so that 'c : B ; C ; C4' means '[[[c : B] ; C] ; C4]'.

The phase-by-phase enclosure operator makes both the distinction between the different forms of parallel operator necessary but does avoid a notation with many complicated enclosure operators in which it would be more difficult to write protocol independent descriptions. There are a number of other combination operators providing control operations:

Precedence overriding/grouping – [**C**]

$\triangle$([C]) = $\triangle(C)$

$\triangledown$([C]) = $\triangledown(C)$

Indefinite repetition – #[**C**]

$\triangle$(#[C]) = $(\triangle(C); \triangledown(C))*$

$\triangledown$(#[C]) = ε

Communication choice – [**a** : **A** | **b** : **B**] **…**

$\triangle$([a : A | b : B] …) = $a_r \uparrow ; \triangle(A)); a_a \uparrow \mid (b_r \uparrow ; \triangle(B)); b_a \uparrow \mid \ldots$

$\triangledown$([a : A | b : B] …) = $a_r \downarrow ; \triangledown(A)); a_a \downarrow \mid (b_r \downarrow ; \triangledown(B)); b_a \downarrow \mid \ldots$

(The passive input/output operators ' !° ' and ' ?° ' can be used in place of ' : '.)

Guarded choice – [**e** $\rightarrow$ **A** | **f** $\rightarrow$ **B** | **…**]

$$\triangle([\ e \rightarrow A \mid f \rightarrow B \mid \ldots]) = \textit{if } e \textit{ then } \triangle(A) \mid f \textit{ then } \triangle(B) \mid \ldots \textit{ fi}$$
$$\triangledown([\ e \rightarrow A \mid f \rightarrow B \mid \ldots]) = \textit{if } e \textit{ then } \triangledown(A) \mid f \textit{ then } \triangledown(B) \mid \ldots \textit{ fi}$$

3.4.2. Data operations

Data operations include input/output communications, expressions and assignment to variables. Expressions are largely uninteresting and usually expand to just a piece of logic between request and acknowledge of the activating handshake. Assignment is considered to be a special case of a communication between expression and variable with pull handshaking used to demand a result from an expression. Input/output communications are the most interesting operations. Tangram treats these operations like their CSP counterparts with atomic expression to port/channel transfers for outputs and port/channel to variable transfers for inputs. The components used to implement these communications expand the transfer into handshakes and exploit control sequencings which overlap data-valid phases of handshakes to form a communications.

There are two forms of each of the input and output commands: passive and active forms. The passive forms allow commands to be enclosed between request and acknowledge in a similar way to ‘ : ’. Active output also involves an enclosure of the output value expression’s evaluation with the output signalling. Only active input resembles the Tangram communication, its right hand side term must be a variable. The forms of the four communication commands are:

Passive input

channel $?^{\circ}$ *command*

Active input

channel $?^{\bullet}$ *variable*

Passive output

channel $!^{\circ}$ *expression*

Active output

channel $!^{\bullet}$ *expression*

N.B. *command*s and *expression*s are the same form of term except that *expression*s may have

a value which is used by the communication operator.

With the passive input, this command is enclosed in the handshake making up the communication. This enclosure is a both-ways enclosure where the expansion is for mixtures of the same protocols on the channel and within the command. Where the command communications are broad and the channel is early, this enclosure is an up enclosure. The effect is to ensure that the command is fully enclosed in the data-valid phase of the incoming handshake. Active output has a similar, data-valid-overlapping, expansion. The expression is activated first with its acknowledgement providing the request for the communication. Where the expression uses early communications (it is effectively a pull communication), enclosure of the phases of the output communication is between up and down phases of the expression (the early data-valid period of a pull handshake). Using broad active output, data is only invalid during the up phase and so the down phases of expression and channel may run concurrently. Passive output works in a similar way but with enclosure of the expression in the communication handshake rather than interleaving of phases. An expression in these expansions is either a command returning a value or a more usual expression in terms of bound variables/values on channels in which the expression is enclosed.

All commands which communicate data are considered to return that data as a value and so can be used as expressions. This allows deeper enclosures of communication commands without requiring intermediate variables. 'Pure' expression terms (channel names, variable names, operators on values …) are shown in italics for emphasis. Consider the definition of the transferrer component (called Fetch in Balsa) which is used to transfer data from an active pull handshake port ('inp') to an active push port ('out') under the direction of a passive sync port ('activate'): It's handshake circuit calculus specification (signalling only) is '#[activate : [inp ; out]]'. This specification mirrors the component's 2-phase refinement well but is not suggestive of the cheap, latch-free implementations with control-interleaving which are used with 4-phase protocols. The Balsa Fetch component's description is:

#[activate : [out $!^{\bullet}$ inp $?^{\bullet}$ *inp*]]

Activate encloses the action. The active output on 'out' cannot occur until the active input on 'inp' has returned a value. On receipt of an acknowledgment on 'inp', the communication on 'out' is begun and completes with an enclosure of handshakes between 'inp' and 'out'.

The handshake on activate is similarly intertwined with the ‘inp’/‘out’ communication. The expansion of this component with 4-phase broad push and 4-phase reduced broad pull channels is:

$A = \#[\ \text{activate} :\ \text{B}\ ]$

$B = \text{out}\ !^{\bullet}\ \text{inp}\ ?^{\bullet}\ inp$

$\triangle(\text{B}) = inp \uparrow ; out \uparrow$

$\triangledown(\text{B}) = inp \downarrow ; out \downarrow$

$\triangle(\text{A}); \triangledown(\text{A}) = (activate_r \uparrow ; inp_r \uparrow ; inp_a \uparrow ; out_r \uparrow ; out_a \uparrow ; activate_a \uparrow ;$
$activate_r \downarrow ; inp_r \downarrow ; inp_a \downarrow ; out_r \downarrow ; out_a \downarrow)*$

The obvious implementation of this specification is a group of three wires (plus the data connections) between ‘activate$_r$’/‘inp$_r$’, ‘inp$_a$’/‘out$_r$’ and ‘out$_a$’/‘activate$_a$’. This is exactly the implementation commonly applied for this component. Using the data-valid enclosure expansion for outputs allows control interleavings to be directly read from the descriptions for data communication components. Where such enclosures are not possible (with 2-phase pull/4-phase broad outputs for instance) latches can be specified in the datapath of the expansion to extend the data-validity of signals.

3.5. Types of components

The handshake component sets used by Tangram and Balsa are very similar. The Balsa component set presented here contains most of the Tangram components, although some may have different names. Various improvements to the Tangram component set have been discussed in other places [58][4] and a number of different names have been applied to same component or the same name applied to components with different ports structures or behaviours. In order to avoid confusion, Tangram terms (variable, transferrer, sequencer, concursor and so on) are used to refer to common component types between Balsa and Tangram. Balsa component names (Variable, Fetch, Sequence, Concur respectively) are used to refer to those components with their Balsa specific port structures. The Balsa component names have initial capital letters to differentiate them from the more general Tangram terms. The names

of Tangram handshake components are given along with the component definitions of those Balsa components which are identical or similar to their Tangram antecedents in the component descriptions in the next section. Note that Balsa components are parameterisable by port types and arrayed port cardinalities but not by port senses. For this reason, some Tangram components map onto more than one Balsa component type with different port sense in each component.

The component descriptions below reflect the complete Balsa component set as it stood before the work described in chapter 5 of this thesis. They are grouped into a number of component classes by the part they most commonly play in handshake circuits. The reasons for this division are explained later (§3.6) along with options for reducing the number of components in each class.

3.5.1. Activation driven control components

The control components provide the events used by other components to sequence their activities. Each control component has a passive sync *activation* port and optionally a number of active sync *output activation* ports. Connecting the output activation port of a component to the activation port of another allows control trees to be constructed in which activity at the leaf ports is controlled by a single collective activation port on the root component. Activity on the output activations is enclosed within handshakes on the activation port and so leaf activity is enclosed within handshakes on the root component's activation. As these components are used primarily to implement command composition in handshake circuit HDLs, the term *command* is used below to refer to activation triggered sub circuits connected to control components' output activation channels. The Balsa control components are: Continue, Halt, Loop, Sequence, Concur and Fork.

Continue and Halt

Continue and Halt are the simplest handshake components. Continue's only function is to acknowledge all requests presented to it and Halt's is to never acknowledges requests. These two components are usually only used to tie off unused activations (usually before applying optimisations which simplify the component connected to the Continue/Halt). The Balsa

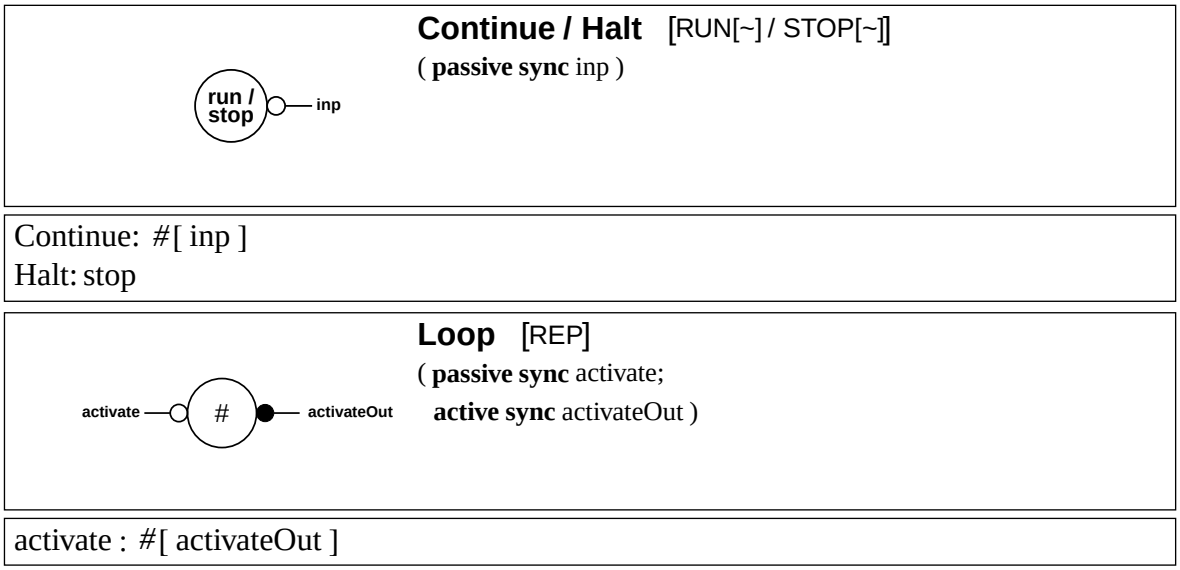


Figure 3.2. Continue, Halt and Loop handshake components

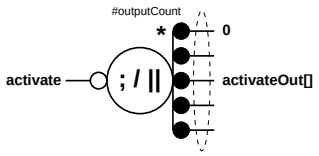
commands `continue` and `halt` map directly onto these components albeit with the Continue almost always being removed by the compiler.

Loop

Loop provides unbounded repetition of a command connected to the active 'activateOut' enclosed in a handshake on 'activate'. As this handshake can never be acknowledged, the acknowledgement path leading from the 'activate' port back up the control tree can be pruned. The term *permanent* is used to describe handshake circuit fragments (such as those with Loop bearing activations) which return no activation acknowledgement. Use of the activation request in permanent circuits is similar. The use of reset signals in other methodologies in which components have no explicit activations. Loop implements the Balsa `loop` command.

Sequence and Concur

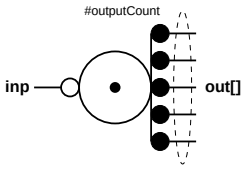
Sequence and Concur form a large part of handshake circuit control trees. They are used to activate a number of commands (either in sequence or in parallel) under the control of the activate handshake. In simple implementations, handshakes on Sequence 'activateOut' ports don't overlap and those on Concur 'activateOut' ports are independent (i.e. don't have synchronisation of return-to-zero phases for 4-phase implementations). This handshake inde-



Sequence / Concur [SEQ / PAR]

(**parameter** outputCount : cardinal;
passive sync activate;
array outputCount **of active sync** activateOut)

Sequence: #[activate : [$activateOut_0$; … ; $activateOut_{outputCount-1}$]]
Concur: #[activate : [$activateOut_0$ || … || $activateOut_{outputCount-1}$]]



Fork [FORK]

(**parameter** outputCount : cardinal;
passive sync activate;
array outputCount **of active sync** out)

#[activate : [out_0 , … , $out_{outputCount-1}$]]

Figure 3.3. Sequence, Concur and Fork handshake components

pendence may be unnecessary where commands connected to a component's 'activateOut' ports can safely overlap/be synchronised without deadlock in Sequence/Concur respectively. Performance increases can be gained in circuits using Sequence components if the use of sequencing is analysed and partially overlapping handshakes are allowed. A number of improved sequencers and a method of analysing macromodular circuits have been devised by Plana [61]. The use of more variants of Concur component can produce improvements in circuit area (Concurs with independent outputs are quite expensive, the alternatives may be slower in some cases but are generally smaller). A common optimisation is to use a Fork instead of a Concur where all the commands of that Concur are permanent.

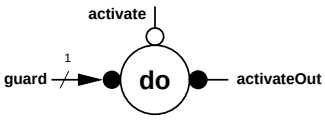
Fork

The Fork is used as both a replacement for Concur as described above and also as a connection component for multicast sync channels. Forks can be implemented just with an *outputCount*-way wire fork and an *outputCount*-input C-element where as Concurs cost an extra S-element for each activateOut port. In 2-phase implementations, Fork and Concur are identical.

3.5.2. Control to datapath interface components

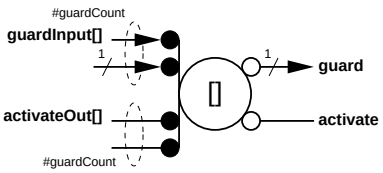
A small number of components allow control sync channels to interact with data transactions. The *transferrer* is the most common of these components, it controls the transfer of data from an active input port to an active output port under the control (and enclosure) of a passive activation port. Components with activations implementing looping and condition control operations as well as the Case component (which translates data values on a passive input activation port into activity on one of a number of active sync ports) also fall in this component class. The complete set of components is: While, Bar, Fetch, FalseVariable, Case.

While



While [DO]
(**passive sync** activate;
active input guard : 1 **bits**;
active sync activateOut)

#[activate : [guard ?• g ; [g → activateOut]]]



Bar [BAR]
(**parameter** guardCount : cardinal;
passive output guard : 1 **bits**;
passive sync activate;
array guardCount **of active input** guardInput : 1 **bits**;
array guardCount **of active sync** activateOut)

#[guard !° (c := [guardInput_0 ?• … ?• $\text{guardInput}_{guardCount-1}$?•
$choose(guardInput_0, \ldots, guardInput_{guardCount-1})$]) != -1)] ||
#[activate : activateOut_c]

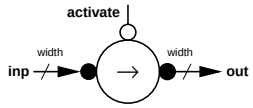
Figure 3.4. While and Bar handshake components

On activation, the While component requests a guard variable on the channel ‘guard’. If this guard value is true then the command connected to ‘activateOut’ is activated. While continues to request guards and activate the command connected to ‘activeOut’ until the guard value becomes 0. The activation is then acknowledged. The While component is used to implement the Balsa `while` command with a single guard and command. Bar can be used to extend the While to implement `while` commands with multiple guards and commands (like CSP guarded commands).

Bar

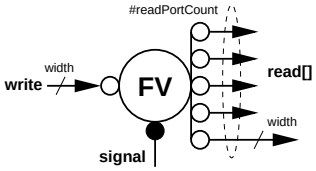
Bar gathers guards from its 'guardInput' ports on receipt of a guard request. The OR of these guards is returned on the 'guard' port. If the Bar component then receives a request on 'activate', one of the commands which corresponds to a true guard is activated. In common implementations, this behaviour requires Bar to contain state (held between guard and activate handshakes) identifying the command to activate. In the case that more than one guard input is true, an arbitrary choice is made about which command to execute. In practical implementations, the commands have a fixed priority and so the choice of command to execute can be determined. The function *choose* in the given behaviour (in figure 3.4) performs the choice, returning -1 to indicate that no output was to be activated.

Fetch



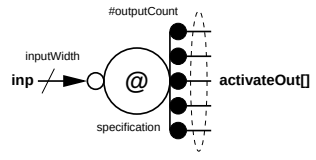
Fetch [TFR]
(**parameter** width : cardinal;
passive sync activate;
active input inp : width **bits**;
active output out : width **bits**)

#[activate : out $!^{\bullet}$ inp $?^{\bullet}$ *inp*]



FalseVariable
(**parameter** width : cardinal;
parameter readPortCount : cardinal;
passive input write : width **bits**;
active sync signal;
array readPortCount **of passive output** read)

#[write $?^{\circ}$ [v := *write* ; signal]] ||
#[read_0 $!^{\circ}$ v] || … || #[$\text{read}_{readPortCount\,-\,1}$ $!^{\circ}$ v]



Case [CASE]
(**parameter** inputWidth : cardinal;
parameter outputCount : cardinal;
parameter specification : string;
passive input inp : inputWidth **bits**;
array outputCount **of active sync** activateOut)

#[inp $?^{\circ}$ [$decode(outputCount, specification, inp) \neq -1 \rightarrow$
activateOut$_{decode(outputCount,\, specification,\, inp)\, \neq\, -1}$]]

Figure 3.5. Fetch, FalseVariable and Case handshake components

Fetch is the Balsa name for the transferrer. This component is the most common way of controlling a datapath from a control tree. Transferrers are used to implement assignment, input and output channel operations in Balsa by transferring a data value from a pull datapath and pushing it towards a push datapath. The components on the 'inp' port are either datapath components (forming expressions) or connection components (forming input communications on ports/language level channels). The components connected to the 'out' port are either variables or connection components forming output communications on language level channels. For example, Balsa implements `if` commands by transferring the guard value (from a pull datapath representing the guard expression) onto a Case component. In 4-phase implementation, there are many choices of handshake expansion for Fetch, particularly if a variety of protocols are used by connected control trees. Fetch implementations typically consist solely of port-to-port wire connections. The problems associated with optimising wire-only components without flattening a netlist are discussed later in this chapter.

FalseVariable

The FalseVariable is used to implement the passive input communication commands `arbitrate` and `select` in Balsa. The arrival of a request on the 'write' port triggers a handshake on the 'signal' port. A command connected to 'signal' is then free to read the value of the communication on 'write' by using the 'read' ports. In implementation, a FalseVariable contains no latches and so reading from the 'read' ports only makes sense while the handshake is taking place on the 'write' port (and so the 'signal' port). FalseVariable can also be considered to be a connection component and is usually used with the DecisionWait component to introduce a control activation to the passive input operation.

Case

The Case component is used to implement data to control decoders for use in Balsa `case` and `if` commands. On receipt of a request and data value on the port 'inp', Case triggers exactly one or none of the output 'activateOut' ports. The mapping of inp values to choice of activateOut port is determined by the specification parameter (the choice is made by the *decode* function in the behaviour given in figure 3.5). As an example, a specification of "0;1..2;3" defines a decoder with three outputs, the first triggered by an input value of 0, the second by

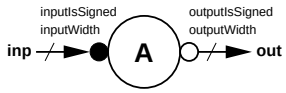
inputs of either 1 or 2 and the third by an input value of 3. As a decoder, Case is also used to implement assignments to arrays with indexed variables on their left-hand sides. The Balsa Case component can be used to replace a tree of the simpler Tangram CASE components (as described in [6]) and is the first Balsa handshake component to make use of specification string to allow a greater degree of parameterisation than just arrayed port cardinalities and types.

3.5.3. Pull datapath components

Compiled data operations ($+$, $-$, ...) in Tangram and Balsa consist of a sync channel meeting a transferrer causing a result to be requested from a tree of pull datapath components implementing the required function and pushing that result onto an output channel or into a *variable* (variables are the components which implement HDL level variables as latches). The pull datapath components form an activation driven tree in the same way as control components but with variables or input channels at the leaves. The activations of these components are pull ports with the incoming request flowing (and forking) towards the leaves of the tree with the result flowing (and joining) back to the root forming the result acknowledgement. Improved datapath compilation is a major item in the 'future work' section (§8.1) and consequently this class of components has remained largely unchanged from Tangram and through the rest of this thesis. The datapath components are: Adapt, Mask, Constant, UnaryFunc, BinaryFunc, Combine and CaseFetch.

Adapt and Mask

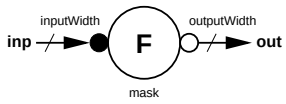
Adapt and Mask have very similar functions. They are used to modify the value coming in on the 'inp' port and present it on the 'out' port in a slightly modified form (the functions *adapt* and *mask* stand in for these operations in the behaviours given in figure 3.6). In the case of Adapt, this involves truncating the upper bits of or zero-padding/sign-extending the input value. Adapt is used to implement type coercions on numeric values. Mask is used to select arbitrary bits from an incoming value, the bit positions being determined by the bit select parameter mask. Adapt and Mask implement just enough mapping functionality to implement Tangram and Balsa data manipulations. It is easy to see that many of the components' parameterised forms consist simply of wires and tied-off/dangling signals. Removing these types of glue components is a major optimisation aim.



Adapt [ADAPT]

(**parameter** outputWidth, inputWidth : cardinal;
parameter outputIsSigned, inputIsSigned : boolean;
passive output out : outputWidth **bits**;
active input inp : inputWidth **bits**)

#[out !° inp ?• *adapt*(*outputWidth*, *inputWidth*, *outputIsSigned*, *inputIsSigned*, *inp*)]



Mask [FILTER]

(**parameter** outputWidth, inputWidth : cardinal;
parameter mask : inputWidth **bits**;
passive output out : outputWidth **bits**;
active input inp : inputWidth **bits**)

#[out !° inp ?• *mask*(*outputWidth*, *mask*, *inp*)]

Figure 3.6. Adapt and Mask handshake components

Constant

Constants provide constant data values to pull channels at the leaves of datapath trees. Constants provide the same kinds of obvious optimisation opportunities as Adapt and Mask.

UnaryFunc

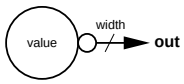
UnaryFunc implements the negate (2's complement) and complement operations in pull datapaths. Separate inputWidth and outputWidth parameters allow results to have different type to the arguments and allow a limited form of optimisation of Adapt/Mask components on UnaryFunc inputs and outputs.

BinaryFunc

BinaryFunc implements the binary datapath operations ($+$, $-$, $=$, $\neq$, $<$, $>$, $\leq$, $\geq$, $\vee$, $\wedge$) on two inputs. A total of seven parameters are provided to configure the types and signedness of the inputs and outputs allowing similar Adapt/Mask optimisations as with UnaryFunc.

Combine

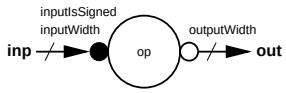
Combine components are used to bit-wise concatentate the values coming in from the two



Constant [CST]

(**parameter** width : cardinal;
parameter value : width **bits**;
passive output out : width **bits**)

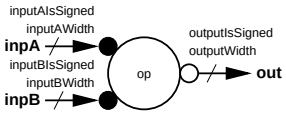
#[out !° *value*]



UnaryFunc [UN]

(**parameter** outputWidth, inputWidth : cardinal;
parameter op : UnaryOperator;
parameter inputIsSigned : boolean;
passive output out : outputWidth **bits**;
active input inp : inputWidth **bits**)

#[out !° inp ?• *op*(*outputWidth*, *inputIsSigned*, *op*, *inp*)]



BinaryFunc [BIN]

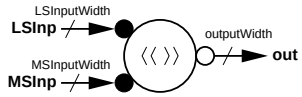
(**parameter** outputWidth : cardinal;
parameter inputAWidth, inputBWidth : cardinal;
parameter op : BinaryOperator;
parameter outputIsSigned : boolean;
parameter inputAIsSigned, inputBIsSigned : boolean;
passive output out : outputWidth **bits**;
active input inpA : inputAWidth **bits**;
active input inpB : inputBWidth **bits**)

#[out !° inpA ?• inpB ?• *op*(*outputWidth*, *outputIsSigned*, *inputAIsSigned*, *inputBIsSigned*, *op*, *inpA*, *inpB*)]

Figure 3.7. Constant, UnaryFunc and BinaryFunc handshake components

input ports ‘LSInp’ and ‘MSInp’. Combines implement the tupling expressions of Tangram and the record and array construction expressions of Balsa. The Balsa compiler also inserts Combines to reconstruct variable read ports for variables which become split during compilation. Writes to Variable components cannot select only a portion of bits to write and so single HDL level variables become multiple Variable components with Split components in the write paths and Combines in the read paths. Combines are, like other glue components, simply wires with a request wire fork and acknowledge synchronisation.

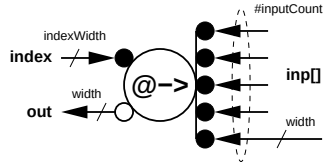
CaseFetch



Combine [COMBINE]

(**parameter** outputWidth : cardinal;
parameter LSInputWidth, MSInputWidth : cardinal;
passive output out : outputWidth **bits**;
active input LSInp : LSInputWidth **bits**;
active input MSInp : MSInputWidth **bits**)

#[out !° LSInp ?• MSInp ?• *combine*(*LSInp*, *MSInp*)]



CaseFetch

(**parameter** width, indexWidth : cardinal;
parameter inputCount : cardinal;
parameter specification : string;
passive output out : width **bits**;
active input index : indexWidth **bits**;
array inputCount **of active input** inp : width **bits**)

#[out !° index ?• inp_{index} ?• inp_{index}]

Figure 3.8. Combine and CaseFetch handshake components

CaseFetch is used to implement indexed array reads. On receipt of an ‘out’ request, an index value is pulled on the ‘index’ port. That value is used to select one of the ‘inp’ ports from which to provide the value back along the port ‘out’.

3.5.4. Connection components

This class includes components used to connect together channels of the same sense, provide synchronisation between multiple channels and combine the activity of a number of channels to allow multiplexing and resource sharing. This class also includes variables as they occupy the same positions in a handshake circuit as other types of channel connection component. Other than variables, the connection components in a handshake circuit are the only components whose presence isn’t explicitly described in the HDL source for that handshake circuit. This is because they are usually present as glue to implement HDL level channels and in particular, the multicast nature of Tangram and Balsa channels. The greater part of connection components implementations consist of just port-to-port wire connections. For this reason, optimising and combining connection components gives us better control of the location of troublesome wire forks which can cause wire load and drive strength management problems

in implementation.

The collection of synchronising and resource sharing connection components is mostly borrowed from the Tangram component set with the addition of parameterised arrayed ports. The connection components are: ContinuePush, HaltPush, Call, CallMux, CallDemux, Passivator, PassivatorPush, ForkPush, Synch, SynchPull, SynchPush, DecisionWait, Split, Arbiter and Variable.

ContinuePush and HaltPush

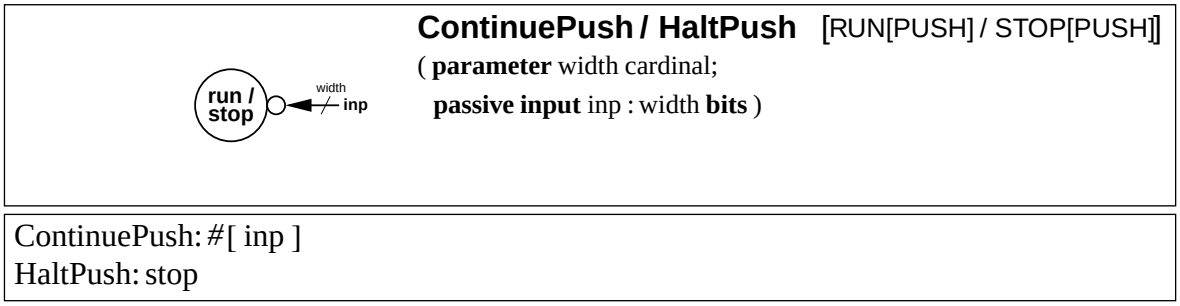
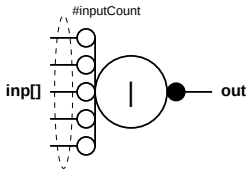


Figure 3.9. ContinuePush and HaltPush handshake components

ContinuePush and HaltPush have identical behaviours to Continue and Halt except they consume incoming data values. They are used mostly to cap push datapaths before optimisation and have just as boring implementations as their control counterparts.

Call{,Mux,Demux}

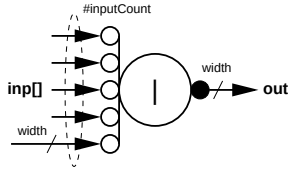
The Call components are all used to provide a resource connected to the common, active port to circuits connection to the passive ports. The signalling required to achieve this resource sharing is provided by a micropipeline style ‘call’ element. Call components expect one of the passive ports to be requesting the resource at a time and provide no hardware in their implementations to enforce this condition. The CallMux and CallDemux components provide datapath steering as well as control signalling. CallMux operates as a multiplexing element, useful in push pipelines for combining sequenced output communications/variable assignments into single output bundles/write port bundles. CallDemux serves a similar demultiplexing role for use with shared input ports. In bundled data implementations, CallDemux components can be realised with the control signalling of a Call component and a number of



Call [MIX[~]]

(**parameter** inputCount : cardinal;
array inputCount **of passive sync** inp;
active sync out)

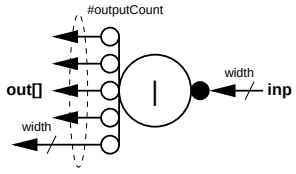
#[[inp_0 : out | … | $inp_{inputCount-1}$: out]]



CallMux [MIX[PUSH]]

(**parameter** width : cardinal;
parameter inputCount : cardinal;
array inputCount **of passive input** inp : width **bits**;
active output out : width **bits**)

#[[out !• inp_0 | … | out !• $inp_{inputCount-1}$]]



CallDemux [MIX[PULL]]

(**parameter** width : cardinal;
parameter outputCount : cardinal;
array outputCount **of passive output** out : width **bits**;
active input inp : width **bits**)

#[[out_0 !° inp ?• *inp* | out_1 !° inp ?• *inp* | $out_{outputCount-1}$!° inp ?• *inp*]]

Figure 3.10. Call, CallMux, CallDemux handshake components

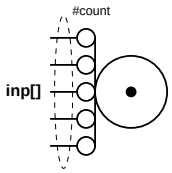
wire forks for the datapath.

Passivator{,Push}

Passivators allow the synchronisation of handshakes on channels connected to active ports. The sync Passivator is usually used to implement multicast sync channels where all the channels are local (i.e. no active ‘expansion’ port is required to pass the handshake on to another synchronising component). PassivatorPushs with outputCount of 1 implement connections of active output to active input ports in command/procedure composition. PassivatorPushs with greater outputCounts are used in the same context to connected multicast channels with data, again without expansion ports.

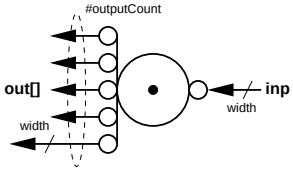
ForkPush

ForkPush is similar to the Fork component. A value pushed in on port ‘inp’ is sent out along



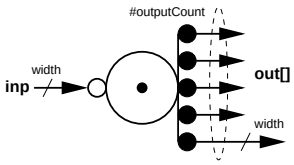
Passivator [PAS[~]]
(**parameter** count : cardinal;
array count **of passive sync** inp)

#[inp_0 : inp_1 : … : $inp_{count-1}$]



PassivatorPush [PAS[PUSH]]
(**parameter** width, outputCount : cardinal;
array outputCount **of passive output** out : width **bits** ;
passive input inp : width **bits**)

#[out_0 !° out_1 !° … !° $out_{outputCount-1}$!° inp ?° *inp*]



ForkPush [FORK[PUSH]]
(**parameter** width, outputCount : cardinal;
passive input inp : width **bits**;
array outputCount **of active output** out : width **bits**)

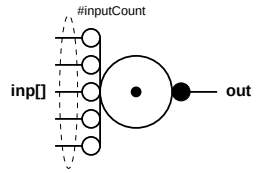
#[inp ?° [out_0 !• *inp* , … , $out_{outputCount-1}$!• *inp*]]

Figure 3.11. Passivator, PassivatorPush, ForkPush handshake components

each of the ‘out’ ports; returning acknowledgements are synchronised. ForkPush is a surprisingly rarely use component, it can be used on its own to implement multicasts from an active output port to a number of passive input ports. Where any active inputs are present, ForkPush can be combined with SynchPush to form a complete multicast communication tree.

Synch{,Pull,Push}

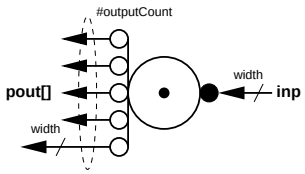
The Synch components implement synchronisation of a number of inputs, perform a handshake on an active output and then acknowledge each of those inputs. The active output port can be used as an ‘expansion’ port to pass the synchronisation on to another component or to a port of the circuit as a whole. The SynchPull component draws data from its active port which is distributed to its inputs during acknowledgement. This can be used to implement multicast inputs from circuit ports. The SynchPush is very similar to a ForkPush but has the active expansion port aout. SynchPush could have parameterisable number of these expan-



Synch [JOIN[~]]

(**parameter** inputCount : cardinal;
array inputCount **of passive sync** inp;
active sync out)

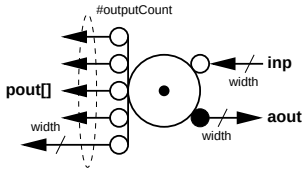
#[inp_0 : inp_1 : … : out]



SynchPull [JOIN[PULL]]

(**parameter** width, outputCount : cardinal;
array outputCount **of passive output** pout : width **bits**;
active input inp : width **bits**)

#[$pout_0$!° $pout_1$!° … !° $pout_{outputCount-1}$!° inp ?• *inp*]



SynchPush [JOIN[PUSH]]

(**parameter** width, outputCount : cardinal;
passive input inp : width **bits**;
array outputCount **of passive output** pout : width **bits**;
active output aout : width **bits**)

#[$pout_0$!° $pout_1$!° … !° $pout_{outputCount-1}$!° inp ?° aout !• *inp*]

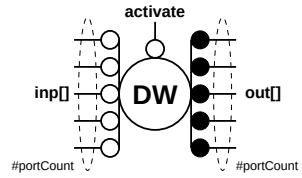
Figure 3.12. Synch, SynchPull and SynchPush handshake components

sion ports to produce a general multicast synchronisation component with a single passive input and a selection of passive and active output ports. This was not done initially as SynchPush is more often used to connect to circuit ports rather than to connect passive inputs to a multicast channel.

DecisionWait

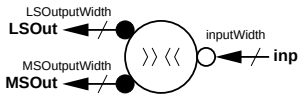
The DecisionWait component is used to synchronise one of the input requests of a Balsa `select` command's channels with the activation channel. The 'inp' and 'out' ports match one-to-one; an incoming request on one of the 'inp' ports is fed through to the corresponding out port on receipt of the activation request. The acknowledgement to that 'out' port is sent back to the activation port as well as its 'inp' ports.

The DecisionWait gets its name from the use of a decision-wait element in its implementation. Decision-wait encapsulates the isochronic fork on the activation request required to im-



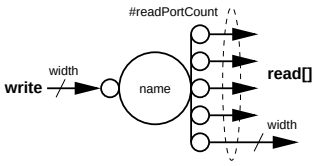
DecisionWait
(**parameter** portCount : cardinal;
passive sync activate;
array portCount **of passive sync** inp;
array portCount **of active sync** out)

#[activate : [inp_0 : out_0 | … | $inp_{portCount-1}$: $out_{portCount-1}$]]



Split [SPLIT]
(**parameter** inputWidth : cardinal;
parameter LSOutputWidth, MSOutputWidth : cardinal;
passive input inp : inputWidth **bits**;
active output LSOut : LSOutputWidth **bits**;
active output MSOut : MSOutputWidth **bits**)

#[inp ?° [LSOut !• *bitfield*(0, *LSOutputWidth* − 1, *inp*) ||
MSOut !• *bitfield*(*LSOutputWidth*, *inputWidth* − 1, *inp*)]]



Variable [VAR]
(**parameter** width, readPortCount : cardinal;
parameter name : string;
passive input write : width **bits**;
array readPortCount **of output** read : width **bits**)

#[write ?° *v* := *write*] ||
#[$read_0$!° *v*] || … || #[$read_{readPortCount-1}$!° *v*]

Figure 3.13. DecisionWait, Split and Variable handshake components

plement such an input selection operation. This encapsulation allows circuits to be built with passive input selection without requiring channels to act as 'probes' by distributing this fork. Using a DecisionWait to implement input 'filtering' is similar to the one-hot encoded channel input of Burns [14].

Split

Split is use mostly to split channels approaching a variable write port. Bitfield extraction on variable read ports is performed by the Mask and Adapt components and so no pull variant of the Split component is necessary.

Variable

Variables implement HDL level variables. Although the above description suggests that

parallel reads and writes are possible, this is not the case. The compilation mechanism for HDLs must ensure that activity on the write port and read ports are mutually exclusive. Concurrent reads are allowed and are not synchronised with each other.

Arbiter

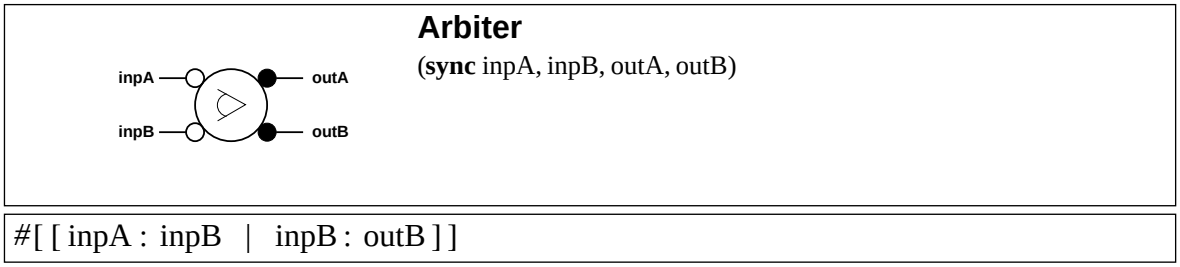


Figure 3.14. Arbiter handshake component

The Arbiter implements a decision between 2 incoming sync channels. A request arriving on one of the inputs will be passed onto the corresponding output (and a handshake will be performed) if only one input is active. If both inputs are simultaneously active, a decision between inputs is made and only a single output is activated.

Arbiter is necessary in places where decisions between a pair of sync handshakes which may overlap is necessary. The Arbiter contains a mutual-exclusion circuit element which contains a filter circuit allowing this decision to be made without causing an metastable signal levels to be exposed to that element's environment.

3.6. Compiling into handshake circuits

Handshake circuits are usually generated by synthesis from a description made in a hardware description language rather than constructed by hand. Tangram was the original language of handshake circuits and Balsa is, superficially, a replacement for Tangram in this role. Tangram and Balsa are both imperative, channel orientated languages not unlike OCCAM. The static process declaration and channel communication model map well onto hardware instantiation and asynchronous control. The handshake component set chosen to implement these languages are also tailored to allow language constructs to be mapped (by synthesis) one-to-one onto particular handshake components. The basis for this mapping is the Tangram compilation function (described in more detail in chapter 7 of van Berkel's thesis [9]). The direct

nature of this command to component mapping is claimed as an advantage by Tangram over less direct systems (such as synchronous VHDL synthesis), allowing the designer to influence the implementation directly through the source description. Maintaining directness under optimisation regimes is a major aim of the synthesis method work embodied in chapter 5.

As well as defining the language to handshake circuit mappings for individual (leaf) commands, the compilation function also describes the ways to combine control activations and connect data-bearing channels to form sequential and parallel compositions of commands. The control components described in §3.5.1 are used to describe the activation control of composed commands, the datapath components are used to implement expressions and the connection components are used to combine channels left dangling by communications, expression arguments and assignment destinations. Variables are placed to terminate dangling read/write channels corresponding to the language level variables when command combination passes back up the parse tree through their declarations.

3.7. Chapter summary

This chapter has introduced some of the features of previous work on Balsa and Balsa's handshake component set. The next chapter describes how the components of this set are mapped into implementations by the back-end of the Balsa design flow. Chapter 5 will then describe a number of new components intended to improve on the component set described here.

The key feature of handshake circuits over other macromodular design styles is the consistent use of channels to connect all the components in a design. It is assumed in the following chapters that this is a desirable property and that the advantages of directness and fine grain DI implementation (or, more usually, DI control signalling with bundled data) are attractive enough for this channel-rich arrangement to be preserved.

Chapter 4. The Balsa Back-end

This chapter describes the new tools added to the Balsa system to allow handshake circuits generated by the balsa-c Balsa compiler to be implemented and simulated. The *back-end* of the Balsa system is the path from a Breeze netlist to a silicon or FPGA implementation of that netlist. This path involves the Balsa tool balsa-netlist, a target (commercial) CAD system and a number of CAD system specific technology descriptions and scripts provided by the Balsa system.

The simulation system described in this chapter uses the tool breeze2hcs to convert Breeze netlists into models in the asynchronous modelling language LARD [22]. LARD models allow behavioural simulation of Balsa designs before technology mapping by the Balsa back-end. The version of breeze2lard described here is a much improved and expanded version of the tool described in earlier work [6].

4.1. The Balsa design flow

Design flows based on handshake circuits use the directness of handshake circuit based HDLs to handshake circuit compilation to allow the design – synthesise – evaluate cycle of design revision by user intervention (as described in §2.1) to be as fast as possible. Using behavioural simulation to perform the evaluation phase of this design loop allows very fast design revision albeit at the cost of realistic timing and (possibly) a compromise in the faithfulness of the modelling of individual handshake components.

In order to allow practical design revision when simulation at gate or layout levels is used, the execution times of parts of the design flow following Balsa compilation into handshake circuits must be suitably fast. This implies that those back-end tools be constructed with a similar regard to directness of implementation as the Balsa compiler. Figure 4.1 shows a design flow for Balsa using both LARD behavioural simulation (which is discussed in §4.5.1) and CAD system native simulation methods. Three design loops are shown in the design flow. These design loops use LARD (behavioural), gate level (functional) and final layout (timing

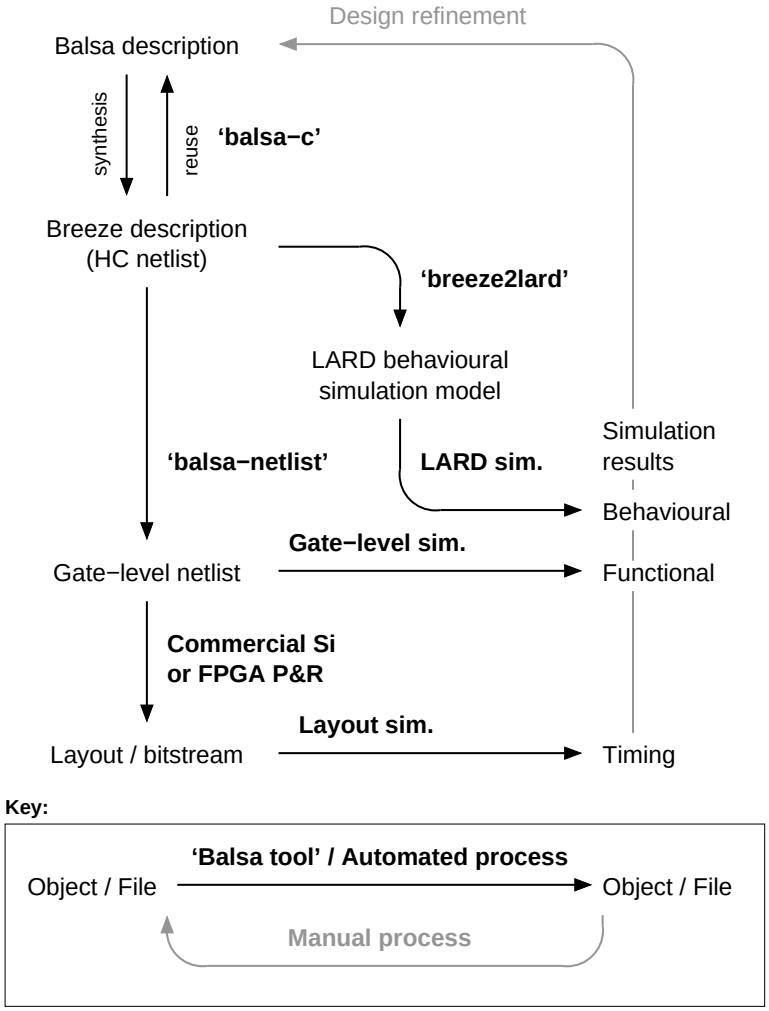


Figure 4.1. Balsa design flow

accurate) simulation tools respectively to allow simulation to be performed at different points in the design flow. In a typical design, all three levels of simulation complexity are used as the design approaches its final version.

A number of tools are involved in this back-end:

Balsa-c: Balsa to Breeze compiler

The Balsa compiler produces handshake circuits from Balsa descriptions. These handshake circuits are expressed in the file format Breeze (which is described in the next section). Balsa compilation is performed using the direct compilation method described in §3.6.

Breeze2lard: LARD behavioural model generation

Breeze2lard generates models written in LARD which can be compiled and run to perform behavioural level simulations of the input Breeze file (and so the source Balsa description). Breeze2lard can also generate simple test harnesses for simulated designs. The test harnesses can be augmented by the user using the LARD language.

Balsa-netlist: Handshake component netlist generator

In order to implement the netlist described by a Breeze file (Breeze is described in §4.1.1), versions of the handshake components used within that file must be generated and imported into the CAD system used to implement the design. Balsa-netlist performs this parameterised component expansion and generates netlists for these components. It also generates the netlist which connects these components. This 'top level' netlist will be equivalent to the netlist described by the Breeze file but with the handshake channels between components decomposed into signal level connections. Balsa-netlist is described in §4.2.

Balsa-{pv, ihdl, xi}: CAD system netlist entry

The netlists generated by balsa-netlist are imported into the target CAD systems by a number of scripts, one per CAD system, which drive the netlist import tools of those systems. This process is not very interesting and so no further explanation is included.

Target CAD system

Balsa designs are, ultimately, implemented using existing CAD tools. The choice of target CAD system affects the netlist generation of balsa-netlist and the template used by balsa-netlist to generate handshake component netlists. The CAD systems used with Balsa are described in §4.2.5.

All the Balsa tools described in this chapter begin with a handshake circuit netlist in Breeze. A short description of the Breeze file format is given below.

4.1.1. The Breeze handshake circuit netlist format

The file format *Breeze* is used as both a compiled library format for balsa-c and as an input format for the (design implementing) back-end and the other handshake circuit manipulating tools described in this chapter. A Breeze file contains handshake circuit netlists with parameterised handshake components as the component instances and handshake channels as the interconnections between these components. Each compiled Balsa procedure is present

in the output Breeze file as a separate handshake circuit netlist described as a Breeze *part*. As an example, the Breeze file for the decade counter example given in figure 3.1 is (with some of the less interesting portions removed):

```
-- Breeze intermediate file (balsa format)
-- Created: Sun Sep  3 18:06:18 2000
-- By: bardslea@Bliss.cs.man.ac.uk (Linux)
-- With balsa-c version: 3.1
-- Command: balsa-c mod10

import [balsa.types.synthesis]
import [balsa.types.basic]

type C_size is 4 bits
constant max_count = (9 as 4 bits)

part mod10 (
    passive sync activate;
    passive sync aclk;
    active output count : 4 bits ) is
attributes (
    isProcedure,isPermanent,
    noOfChannels=24,line=9,column=1 )
local
    sync "@14:3" #1
    sync "aclk" #2
    push channel "count" #3 : 4 bits
    pull channel "tmp" #4 : 4 bits
    . . . -- omitting some channel declarations
begin
    $BrzVariable ( 4,3,count_reg[0..3] :
        #5,{#13,#18,#7} )
    $BrzVariable ( 4,1,tmp[0..3] : #24,{#4} )
    $BrzCallMux ( 4,2 : {#10,#15},#24 )
    $BrzLoop ( #1,#23 )
    $BrzDecisionWait ( 1 : #23,{#2},{#22} )
    $BrzSequence ( 3 : #22,{#20,#8,#6} )
    $BrzCase ( 1,2,0;1 : #21,{#11,#16} )
    $BrzFetch ( 1 : #20,#19,#21 )
    $BrzBinaryFunc ( 1,4,4,NotEquals,false,false,false :
        #19,#18,#17 )
    $BrzConstant ( 4,9 : #17 )
    $BrzFetch ( 4 : #16,#14,#15 )
    $BrzBinaryFunc ( 4,4,1,Add,false,false,false :
        #14,#13,#12 )
    $BrzConstant ( 1,1 : #12 )
    $BrzFetch ( 4 : #11,#9,#10 )
    $BrzConstant ( 4,0 : #9 )
    $BrzFetch ( 4 : #8,#7,#3 )
    $BrzFetch ( 4 : #6,#4,#5 )
```

```
end
```

A Breeze file contains the `import`, `type` and `constant` declarations present in the source Balsa file from which it was compiled. Where parameterised Balsa procedures were described in the source Balsa file, the Balsa code necessary to generate parameterised versions of those procedures is also included in the Breeze file (parameterised procedures are a new addition to the Balsa language, described in §4.5.2). These declarations and pieces of code are used when the file is re-imported into balsa-c by an `import` line in another Balsa file.

The only Breeze part described in this example is `mod10`, the modulo-10 counter example's procedure. The port list of `mod10` contains three port declarations with senses, directions and Balsa types for each one. A Breeze part has the same port structure as its source Balsa procedure with the addition of an explicit activation port.

A part's `attributes` section gives some details of the part which make subsequent parsing easier. The attribute `isProcedure` identifies this as a compiled Balsa procedure, so guaranteeing that the first port of the part is a conventional procedure activation. The attribute `isPermanent` specifies that `mod10` once activated, will not return an activation acknowledgement (this information can be used to optimise the path from the activation of a circuit which includes `mod10` to the activation of `mod10`). The `noOfChannels` attribute is used by balsa-c to allocate space ahead of reading a (potentially long) channel list in order to improve performance. The final two attributes, `line` and `column`, identify the position in the source Balsa file at which this part's corresponding Balsa procedure was defined. This location is used by breeze2lard to help with source level debugging.

The main body of the part consist of two portions: the channel declarations and the instantiated handshake components.

Part channels

The part channel declarations specify the direction, push/pull sense and type of each of the ports and internal channels. The channels in a Breeze file are numbered sequentially rather than named. The first few of these channels correspond to the ports of the part. In this case, channels `#1`, `#2` and `#3` correspond to the ports `activate`, `aclk` and `count` respectively. Numbering the channels and ports this way makes processing of the components easier and

makes determining whether a channel is connected to a port easier (a number comparison as opposed to a name search).

It is not obvious from this example, but the types of data bearing channels are specified using only the form *n* `bits` (where $n \geq 1$). This simple typing, along with the overlapping of channels and ports, makes it possible to process Breeze files without knowledge of Balsa typing. All the internal channels of a part are typed in a similar way.

The names which appear in quotes in the channel declarations are the names which should appear in behavioural simulation of this part. These names specify which port or variable this channel flows to or from, or in the case of sync activation channels, the position in the source file of the command which this channel activates.

Part components

The component instantiations inside a part can contain parameterised handshake components from the handshake component set (identifiable by their `$Brz` prefix) and also instantiations of previously declared Breeze parts. Handshake component instances have their parameters specified at the beginning of their argument list. The parameters for handshake components are restricted in type to integers, strings and enumeration values (the boolean values `true` and `false` are enumeration values). The parameters are specified in the order in which they are present in the component definitions given in §3.5.

The numbers following the parameters (after the separating '`:`') specify the channels to which this component connects (again in the same order as the component's definition). Where arrays of ports are present on a component, channel number lists are enclosed in braces to delimit individual arrayed ports. The cardinality of arrayed ports is, in all of the defined handshake components, stated explicitly in one or other parameter of that component and so a tool processing a Breeze file can ignore the braces if it so wishes and use the parameters alone to determine the ports to which that components channel connections should be made.

Breeze part instances have the same form as handshake component instances except that no name prefix is present and no parameters are allowed. A command line option to balsa-c can be used to flatten instances of Breeze parts used within other parts into their constituent

handshake components.

4.2. Handshake component generation – balsa-netlist

Gate level implementations of a Balsa handshake component can be generated by applying the parameters from a Breeze instantiation of that component to the template for that component held by the Balsa back-end. The gates generated by these templates are not target technology cells but are generic logic gates with unlimited numbers of inputs. All the handshake components available to Balsa are described by a common set of templates (the 'common' technology) with special, target technology specific, versions only used in special cases. The templates are written in a handshake component description language which is processed to produce those generic gates. Target technology implementations are then generated by technology-mapping those generic gates. These technology specific netlists can then be written to disc (in CAD system native netlist formats) and imported into the CAD databases of the back-end CAD tools.

Figure 4.2 shows the progress of a design through this technology mapping back-end design flow. The 'gen' stage of component generation performs the template application into generic gates. The generic gates are then mapped onto gates with realistic numbers of inputs (to match the numbers of inputs available on gates of the target technology) by the 'map1' process. The 'map2' process then replaces these smaller gates with cells from the target cell library so producing the final implementation. The generated netlist is then output in a format appropriate to the target CAD system by the 'net' process.

Balsa-netlist is written in Scheme [43], a dialect of Lisp, and executed using the GNU Guile Scheme interpreter [48]. This choice of implementation languages allows new additions to the tool to be rapidly developed and for the CAD interface scripting and heavy data structure processing of balsa-netlist to share a common codebase.

4.2.1. Handshake component templates – gen

The templates describing parameterised handshake component generation contain descrip-

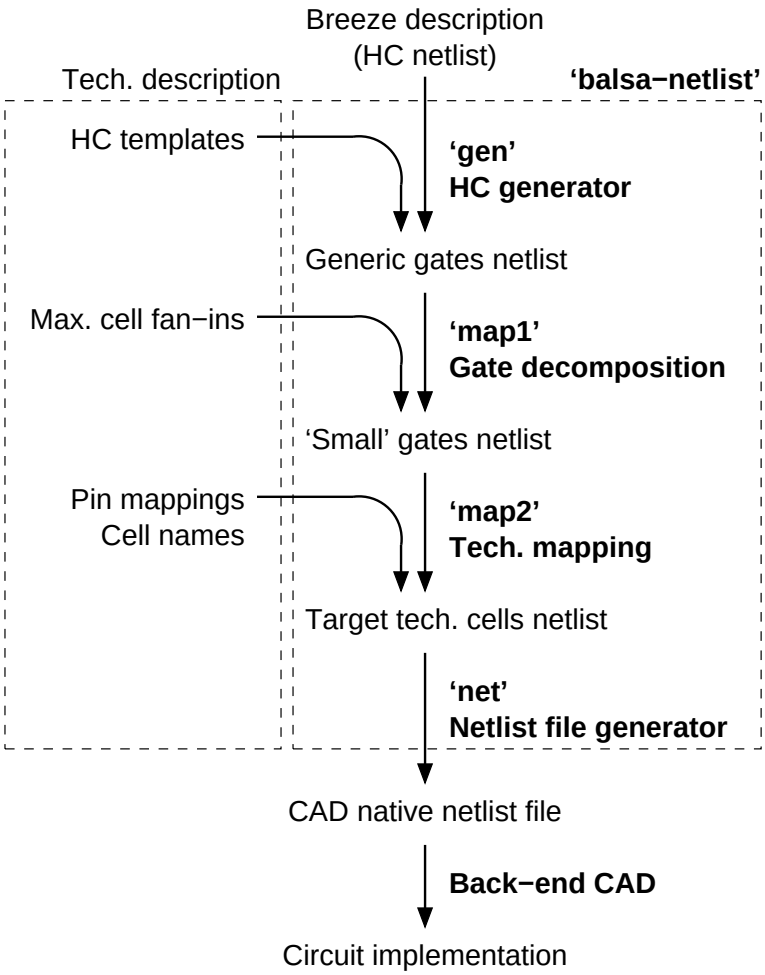


Figure 4.2. Balsa back-end design flow

tions of those components as combinations of the generic gates as described above. As an example, consider the expansion of a 4-way Fork component (Fork is described in §3.5.1) which appears in a Breeze file as:

```
Fork (4 : #1, {#2, #3, #4, #5})
```

Fork has a template in the 'common' technology which describes an implementation of Fork in which an incoming request, on port `inp`, is forked to form outgoing requests on each of the requests of the channels of port `out`. A tree of C-elements gathers acknowledgments from the `out` ports and returns an acknowledgement to `inp`. The template for this implementation is:

```
(primitive-part "Fork"
  (parameters
    ("outputCount" (named-type "cardinal"))
  )
  (ports
```

```
      (sync-port "inp" passive)
      (arrayed-sync-port "out" active 0
        (param "outputCount"))
    )
    (symbol
      (centre-string ".")
    )
    (implementation
      (technology "four_b_rb"
        (nodes)
        (gates
          (c-element (ack "inp") (ack (each "out")))
        )
        (connections
          (connect (req "inp") (req (each "out")))
        )
      )
    )
  )
```

This template contains not only an implementation for Fork (in protocol combination `four_b_rb` – 4-phase broad push/reduced broad pull, see §4.3), but also the order, types and cardinalities of that component's parameters and ports. Notice that the `out` port is arrayed over `outputCount` port elements matching the four channels (#2, #3, #4. #5) used in the example instance. The symbol section of the template gives a description of a string to be used by the breeze2ps handshake circuit 'pretty printer' (described in [6]). The implementation of Fork is split into three parts: the internal nodes to the component (`nodes`), the gates internal to the component (`gates`) and the connections between ports of the component (`connections`). Separating the internal gates (which may include internal connections/wire renamings) from the external port cross-connections allows handshake circuits to be partially flattened to remove port-to-port connections which are difficult to represent in many netlist formats and make signal load management difficult.

When converted to target technology cells, two circuits are described to implement this form of Fork: `Brc_Fork4` and `Brz_Fork4`. `Brc_Fork4` contains the cell instances described in the `(gates …)` section of the circuit description. These cells form the functional bulk of the circuit and present an external interface which consists of a subset of the ports of the Fork component. The `Brz_Fork4` circuit contains the whole circuit implementation in the form of an instance of `Brc_Fork4`) and the port-to-port connections implemented with logical buffer components. These buffers must be removed during the back-end before

circuit implementation.

The `Brz_Fork4` component's external interface is a set of ports to match the 5-ported Fork giving in the Breeze examples, i.e. a 10-ported circuit with alternating request and acknowledge ports.

The single gate in Fork's `(gates …)` section is:

```
(c-element (ack "inp") (ack (each "out")))
```

Gate descriptions of this form may be instantiated as a number of generic gates. In this case, a tree of C-elements is generated to join the 4 acknowledgements on the `out` ports to the acknowledgement on the `inp` port. Ports and arrays of ports are identified by their names and the `ack`, `req` and `data` operators split off the relevant wire portions of the ports given as arguments to them. The `each` operator expands the arrayed ports passed to it into a list of all the constituent ports of those arrays which can then be passed to one of the wire-selecting operators. In this case, the wires resulting from the `(ack (each …))` term form the input to a multi-way C-element implemented as a balanced tree of three 3-input C-elements. The `connect` command which is used in the `(connections …)` section works in a similar fashion to the `c-element` gate and expands into a set of connecting buffer generic gates from the request of `inp` to the requests of each of the ports of `out`. Where the arguments to the gate generating commands consist of more than one bit, a number of similar sets of gates are generated to implement the same function on respective bits of each argument giving rise to a 2 dimensional arraying of gate generation.

This 2 dimensional arraying of gates (by cardinality of arguments and bitwise width of each argument) allows all the connection components to be implemented very concisely. Operators for slicing, bit-reversing, bitfield-extracting and slice-combining allow the internals of other useful components (such as the many forms of BinaryFunc) to be described. Operators also exist to describe conditional expansion of gates in order to make implementation decisions based on component parameters. The language does not, however, include explicit operators to iteratively generate gates. The bit-rearranging operators allow all the current Balsa handshake components to be described using only the iteration implicit in the 2 dimensional expansion of gates.

4.2.2. Generic component decomposition – map1

The single generic gate produced by the expansion of a 4-way Fork is:

```
(c-element "inp_0a" "out_0a" "out_1a" "out_2a" "out_3a")
```

This is a 4-input C-element with output connection to signal `inp_0a`. Notice that only one gate is generated as all the arguments to the template gate had a cardinality of one. Also note that signal names have been technology mapped into compounds of port name, port index (0 for un-arrayed ports) and the portion of the channel which this signal represents (`r` for request, `a` for acknowledgement and `d` for data in this technology).

The mapping of this gate into smaller, implementable gates is performed by 'map1' under the direction of maximum gate fan-in information provided by the technology description files. This technology (the example is using the 'ams035' technology) has only 2-input C-elements and so 'map1' returns the set of gates:

```
(c-element2 ("internal_0d" 0) "out_0a" "out_1a")
(c-element2 ("internal_0d" 1) "out_2a" "out_3a")
(c-element2 "inp_0a" ("internal_0d" 0)
    ("internal_0d" 1))
```

Two internal nodes are introduced as bits of the bus signal `internal_0d`. Decomposition of C-elements this way produces a tree of C-elements which is not capable of recovering from the removal of an input transition before the tree's output has transitioned as a consequence of that signal. These decompositions are acceptable for this application. Applications which which require a C-element to tolerate pulses on inputs must make use of simple 2-input C-elements.

AND, OR, NAND, NOR and 2-level (AND/OR) implementations of decoders/encoders are handled with similar, tree building, gate decomposition mechanisms.

4.2.3. Target technology gate mapping – map2

The second stage of the technology mapping process produces target technology cells from the simple gates produced by 'map1'. Connections on the simple gates are mapped onto the pin order of the target technology cells and the names of the target cells are substituted for the simple gate names. The cells produced for the 4-way Fork are (in the Balsa netlist format):

```
(instance c2 ("internal_0d" 0) "out_0a" "out_1a")
(instance c2 ("internal_0d" 1) "out_2a" "out_3a")
(instance c2 "inp_0a" ("internal_0d" 0)
    ("internal_0d" 1))
```

4.2.4. Netlist generation – net

Balsa internal netlists for generated circuits are output in CAD system native netlist formats using the 'net' netlist generators. Netlist generation involves gathering prototypes for cells used by the generated netlist, pruning unused signals and adding connections to power components from power signal references in the Balsa netlists. The `Brc_Fork4` component definition is output, in structural Verilog, as:

```
module BrcFork_4(out_3a, out_2a, out_1a, out_0a, inp_0a);
  input out_3a;
  input out_2a;
  input out_1a;
  input out_0a;
  output inp_0a;
  wire [1:0] internal_0d;
  wire vcc, gnd;
  LOGIC0 gnd_circuit (gnd);
  LOGIC1 vcc_circuit (vcc);
  c2 I0 (internal_0d[0], out_0a, out_1a);
  c2 I1 (internal_0d[1], out_2a, out_3a);
  c2 I2 (inp_0a, internal_0d[0], internal_0d[1]);
endmodule
```

4.2.5. Commercial CAD systems

The Balsa back-end generated gate level netlists to import into target CAD systems in order to produce circuit implementations. To date, three CAD systems have been targetted, each with its own netlist format, naming conventions and tool flow. Balsa currently has a number of named technology back-ends (given here in parentheses), each targeting one of the three CAD systems described below. Each of these technologies also specifies a netlist format and name mapping scheme (which characters are allowable and escaping mechanisms for these characters) to use with that netlist format.

Compass Design Automation tools from Avant! (armlr7)

Compass [18] is a complete IC design package with schematic entry, simulation, standard cell compilers and layout design packages. Balsa netlists are entered into Compass using Compass's native netlist format NLS. Schematics are generated from these netlists by the netlist import process and these schematics are used to produce layout using the standard cell place and route tool PathFinder. Compass was used with a cell library provided by ARM Ltd. to implement the AMULET3i described in chapter 6. The AMULET3i DMA controller was produced using Balsa synthesis with this technology. Synopsis's TimeMill [71] was used to simulate these implemented designs at both schematic (pre-place and route) level and after layout was generated.

Xilinx Alliance FPGA design tools (xc4000e, virtex)

The Xilinx Alliance tools [74] are used to generate programming streams for Xilinx XC4000E and Virtex FPGA devices. Innoveda's Powerview [39] CAD system is used to enter hand drawn top level schematics for FPGA pin connections. Balsa netlists are imported into Powerview and schematics for those netlist are generated by the Viewlogic tool ViewGen if so desired. EDIF 200 netlists are used to import into Powerview and to export netlists from Powerview into the Xilinx Alliance tools. The Xilinx tools are then used to map the Balsa design into FPGA look-up table entries and interconnect and to generate the bitstream used to program the target FPGA.

Cadence Design Framework II (ams035)

Cadence [15] is used to enter designs destined for silicon, standard cell implementations. Currently only the AMS 0.35µm 3LM process [1] is supported using Cadence's Silicon Ensemble place and route tools and Verilog [73] simulation. Verilog structural netlists are generated by balsa-netlist for importation into Cadence and schematics are generated from these netlists by the Cadence tool ihdl. Cadence is a more popular IC design tool than Compass and so the majority of work on future technologies will probably be conducted using Cadence tools.

The current back-end does not allow the technology description to contain any technology specific optimisations or compilation options for use with the Balsa to Breeze compiler balsa-c. Such compilation guidelines may be added in future revisions of the technology mapping portion of the back-end.

4.3. Component implementations

The handshake component implementations produced by this back-end from its template are very similar to the implementations of Tangram and Balsa handshake circuits described elsewhere. The synchronising components (Synch, Passivator …) consist of wire forks and C-element trees, transferrers are built using only wires and variables are built with transparent latches using the delay caused by driving the latch enable signal to provide a write port acknowledgement. The Call components have conventional merged requests and C-element gated acknowledgements (like those described in §2.3.1).

Currently, Balsa handshake circuits use the same handshaking protocols for all push channels and another single protocol for pull channels. Push and sync channels use 4-phase broad signalling. For push channels this choice makes data validity as long as possible (in order to simplify data processing components) and in sync channels to allow cheaper parallelism (by use of Fork instead of Concur) and use of the return-to-zero phase. Using broad push signalling with a cheap, wires-only transferrer requires that the pull components making up the expression sourcing that transferrer keep their data valid for some time after the end of their output handshakes.

Using 4-phase broad pull for these expression channels would be ideal in order to provide this long data validity. Unfortunately, broad pull requires that variable read ports keep their values between handshakes even where that variable is written to. For this reason, pull channels are implemented using 4-phase reduced broad signalling. Input data to pull components is allowed to change between one handshake and the next although it must remain valid for at least as long as the data-valid period of the push handshake on the output side of a transferrer to which the pull channel is connected. In practice, this constraint is met by the strict sequencing of writes and reads to variables enforced by the compilation process. A variable sourcing the pull channel of a transferrer on which a transfer is currently taking place will not be written to until the handshake controlling that transferrer (which completely encloses the output, push, handshake) has completed.

The pull data processing components (BinaryFunc, UnaryFunc, Split, Constant and CaseFetch) are implemented in such a way that their data processing portions operate correctly

(performs the correct logical function) even between handshakes. For components containing an adder structure, the carry chain of the adder is implemented using a pair of carry and carry-valid signals. Carry-valid can be used, when the component is activated, along with a completion detection tree to provide a matched path delay to correctly return an acknowledgement when data is ready. During the extended validity period between handshakes required by reduced broad signalling, the carry chain continues to function correctly (and so the adder continues to produce correct results) but the carry-valid signals are returned to zero.

4.4. Problems with this back-end

This back-end has a number of outstanding problems. Some of these are related to the way that this back-end does not optimise or perform signal drive strength analysis across component boundaries. Other problems are related to the lack of consistent timing validation in the current back-end.

4.4.1. Signal drive strengths

Managing delays caused by the load on signals due to signal fan-out is currently performed by compiling tables of 'required drive strengths' for all the signals within a handshake component while producing the parameterised version of that component. This tabulation is automated, the errant signals are reported to the user in order of their lack-of-drive. Any cells which are incapable of driving their output signals can then be replaced by more strongly driving cells or buffer cells.

Insertion of buffers to increase signal drive strength is currently performed by hand as it is difficult to get a good view of the problems caused by poorly driven signals until after simulation. When simulated, a design yields information about the delays caused by poor drive across signal boundaries and at the circuit peripheries.

The large number of Balsa handshake components which consist largely of feed-through connections from one port to another (e.g. Adapt, Fetch, Split, Combine) makes a hierarchical approach to managing signal buffering impractical as realistic, static values for the loads presented by the inputs to handshake components cannot be compiled. Flattening these port-

to-port connections by using the two component (`Brz` and `Brc` prefix component) expansion of handshake components can allow these feed-through connections to be removed but at the cost of producing more complicated, difficult to traverse schematics/netlists. Figure 4.3 shows the implementation of the 2-way DecisionWait handshake component. The acknowledgements for the output sync channels are fed directly back to the input sync channels as part of the enclosing, port-to-port connections, `Brz` wrapper.

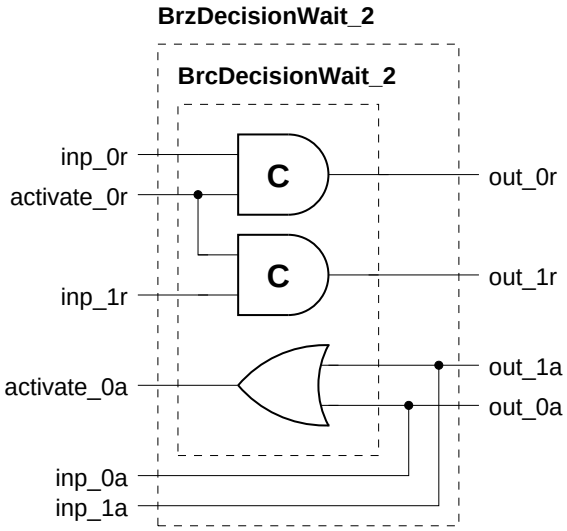


Figure 4.3. DecisionWait handshake component implementation

Two obvious solutions present themselves: the consistent use of timing analysis on produced circuits to choose points in the top level netlist to place buffers and the adoption of a set of handshake components which have fewer feed-through connections and so fewer signals to complicate partially flattened (flattened up to the `Brc` components) netlists. The components described in chapter 5 are intended, in part, to address this feed-through problem.

4.4.2. Timing validation

Timing analysis of Balsa synthesised circuits is currently undertaken by 'exhaustive' simulation of the final, implemented layout of those designs. Simulation trace files can be analysed to ensure that signal transitions are fast enough (addressing the signal drive strength problem) and that bundling constraints on channels and timing constraints internal to the handshake components are met.

Unfortunately, this form of analysis can not be guaranteed to expose all timing problems

as it may be impossible (or at least impractical) to exhaustively exercise a particular design. Therefore, static timing analysis will have to be added to future versions of the Balsa back-end which produce implementations which require internal timing constraints to be obeyed. It is also planned to target Balsa at the newer approaches to delay-insensitive circuit generation (such as n-of-m codes and NCL). These DI approaches will not require timing analysis other than to treat slow signal transitions due to poorly driven signals.

4.5. Other new design flow features

The new Balsa back-end isn't the only new part of the Balsa design flow. As mentioned previously, an improved version of the LARD simulation interface has been developed. A number of additions to balsa-c and the Balsa language have also been made since the work described in [6]. Two tools to help with design management, balsa-md and balsa-mgr have also been developed.

4.5.1. LARD simulation – breeze2lard

LARD [22][21] is essentially a flexible programming language with very fine grain threads, a run time environment with source level debugging and an acknowledgement of the importance of channel communication. Channel communication in LARD is implemented by careful use of shared memory, non preemptive threading and parameterised typing and although very flexible is very easy to break by accessing the shared memory directly or by failing to enforce mutually exclusive sender/receiver use of the two ends of a channel. LARD has, however, been of great use in producing initial designs for AMULET3i. The ability to write large, behavioural, channel connected blocks which simulate quickly allowing realistic amounts of code to be run on a modelled processor allows the user to try different processor organisations in a short space of time.

An example of the use of LARD as a simulation back-end for Balsa was given in [6]. This original tool mapped Breeze handshake circuits into LARD by instantiating models of each component in that handshake circuit as a separate LARD process producing a structural model of the Breeze. Structural translation leads to very slow simulations due to the heavy channel communication cost in LARD and also slow channel viewer startup due to the

potentially large number of channels involved. Balsa's data typing was simulated using arrays of boolean values to simplify bitfield extraction/concatenation.

Since then, a much improved tool, breeze2lard, has been developed to allow Balsa designs to be automatically translated into LARD to allow design validation and simulation. Breeze2lard creates LARD models by translating Breeze handshake circuit files either structurally into LARD (in the same way as the old Breeze to LARD translator) or by recovering behavioural descriptions from the handshake circuit control flow. Structural translation is used as a fall back position when performing behavioural description recovery for portions of the handshake circuit for which rules for reconstructing behaviour do not exist.

Unfortunately the existing versions of LARD posed a number of additional problems when used as a simulation engine for Balsa.

Changes to LARD

Many of the problems inherent in using LARD as a Balsa simulation environment are a result of its initial design and emphasis on higher level modelling. These include:

- Balsa data types are of fixed length but Balsa does support integers of lengths greater than the machine word. LARD models typically use the built in type `int` which represents a single machine word to move data around. LARD does not support arbitrary precision integers.
- LARD does not support bitwise record field positioning. A type such as `record a : 5 bits; b : 3 bits end` in Balsa is an 8 bit long type with two fields covering bitfields [4..0] and [7..5] respectively. LARD only supports named field extraction on whole words.
- LARD channels don't support pull channel behaviour. The standard LARD channel model provides only abstract, 2-phase like channel behaviour.
- LARD doesn't support source level debugging of anything but LARD. The user cannot 'renumber' the lines in a LARD file to make the use of a language translated to LARD transparent. In C this can usually be achieved by using the preprocessor cpp(1). cpp removes all preprocessor directives and expands macros but makes these changes transparent to the user by adding explicit filenames and line numbers in its output format

which the compiler uses to generate debugging information.

- The LARD channel viewer can't display enumerated types, multi-precision integers (however they are implemented) or record structures in any form other than simple formatted integers.
- The large number of channels present in handshake circuit descriptions lead to LARD models which compile and run very slowly.

The need for all of these features has been met by adding them either to the LARD language, its interpreter or by rewriting some portion of a LARD library.

Multi-precision integers were adopted by adding support for the GNU Multi-Precision arithmetic library (libgmp) [34] to LARD. Formatted printing and parsing were added by making C-like printf and scanf format string based printing and parsing functions available in the LARD language. These formatting operations were written into the code of the LARD interpreter to allow them to be used by the channel viewer for signal display. Bitfield record types are supported by bit extracting variable read operations on multi-precision integers. To support source level Balsa debugging, cpp style '# lineno filename' directives are recognised in input files by the LARD compiler (lcd). Pull channels are provided by a rewritten channel library which models channel request/acknowledge pairs as individual signals with 4-phase handshakes. All these new features were made available in LARD version 2.0.12.

The one remaining problem is the large number of channels present in translated designs which are structurally generated from Breeze. Parameterised models of the Balsa handshake component set were hand written in LARD which the translation tool would compose to generate a structural model of the Breeze handshake circuit. Translation could instead be performed from Balsa itself resulting in more behavioural models with fewer unnecessary channels. A large proportion of these channel are associated with the control tree rooted at the 'activation channel' (effectively the reset wire) of the circuit so the replacement of this tree by LARD code would significantly reduce the size of the simulation model. Unfortunately this would also make the translation tool dependent on the syntax stability of the Balsa language whereas the use of Breeze only requires new LARD component models to be written for new handshake components. Fortunately, due to the direct nature of Balsa compilation, we can perform the same control translation by examining the control handshake components

dangling off the activation channel.

Behavioural models

Behavioural descriptions are recovered from handshake circuits by noting that these circuits consist mainly of a control tree terminated in transferrer components. The transferrers are themselves connected to expressions or channels on their input sides and to variables or channels on their output sides. Their function is to transfer data from their inputs to their outputs under the control of their data-less sync control channels. Figure 4.4 shows this general 'coat hanger and bow ties' structure.

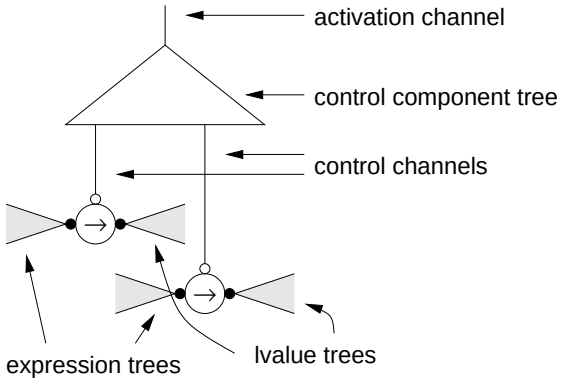


Figure 4.4. Control tree structure in handshake circuits

A diagram of this kind can be drawn for any handshake circuit as all control flows, ultimately, from the circuit activation. Consider this small buffer example:

```
procedure buffer (
  input i : byte;
  output o : byte ) is
variable x : byte
begin
  loop
    i -> x;
    o <- x;
    o <- (x + 1 as byte) -- add one
  end
end
```

The handshake circuit diagram and activation tree for the compiled version of this example is shown in figure 4.5 (with channel numbers indicated on both diagrams). This circuit acts a single-place buffer which sends out its received input value twice (once with its original value and a second time incremented) before accepting a second input. The activation tree structure

is simply a tree with three transferrers each connected to the variable `x` and respectively connected to ports `i`, `o` and `o`. The increment expression on `x` by the third communication is shown as part of the expression tree on that communication. The two connections to port `o` are shown without a multiplexer (The indicated component in the handshake circuit diagram) as any possible activation of the tree will never result in non-mutually exclusive writes to that port. The multiplexer is not shown on the activation tree diagram as connection components are not shown in this notation. The translator handles the activation tree by keeping a set

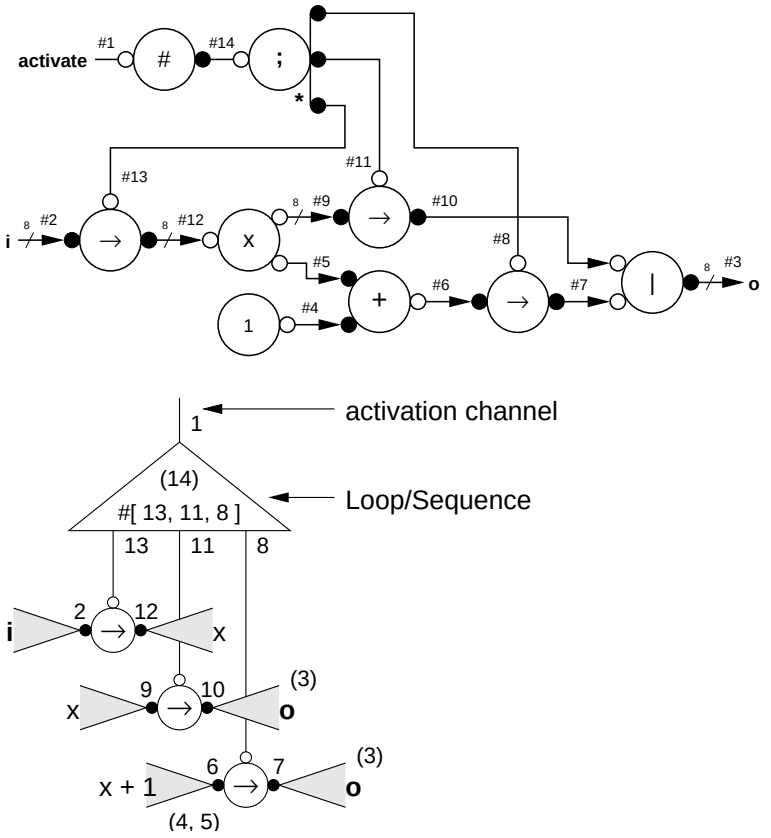


Figure 4.5. A simple buffer example: handshake circuit and activation tree

of root channels to control trees which need to be traversed. The majority of the control components (Loop, Sequence, While, Concur …) are known explicitly to the translator and are converted to a tree representing the LARD code to generate. Expressions and lvalues (the component tree connected to one or more variable write ports or output channels) are handled by two additional functions working like the activation tree crawler but along datapaths to and from variables and ports. The expression and lvalue functions are called when the activation tree crawler reaches a transferrer.

As LARD does not require explicit multiplexing or control demultiplexing for forks and joins in channels, the translator generates no code for these cases. If a control tree has a channel connected to a Call block then this is treated in the same way: an explicit channel communication is placed in the LARD code and the channel number of this communication is added to the working set of unexamined control trees.

Once all the control trees of a procedure have been examined, the LARD code is generated. The code for the buffer example is:

```
‘BP_buffer‘ (
  ‘P_activate‘ : var(SyncChan),
  ‘P_i‘ : var(PullChan(8)),
  ‘P_o‘ : var(PushChan(8))
) : expr(void) =
(
  ‘V0_x[0..7]‘ : var(BalsaVariable) .
  Init (‘V0_x[0..7]‘, V0_x[0..7]) ;

  forever (
    ‘P_activate‘ ? ( Body of select enclosed
      forever (
        ‘V0_x[0..7]‘ := (‘P_i‘ ? (?‘P_i‘)) ; input
        ‘P_o‘ ! Read (‘V0_x[0..7]‘) ; output
        ‘P_o‘ ! (Read (‘V0_x[0..7]‘) +
          s2mpint ("1"))[7 to 0]
  ) ) )
) .
```

The generated LARD code is almost exactly the same as the source Balsa code (in control structure if not in syntax). The major difference is the addition of the outer ‘`‘P_activate‘ ? (forever …)`’ loop which allows this procedure to be structurally composed and to respond to an explicit activation. If this were not present and the control structure followed the Balsa exactly, it would be difficult to accommodate the notion of shared blocks of code without more complicated analysis of the input code.

Consider the same example with the two output communications replaced by calls to a shared block of code (without the increment in the second communication). First in Balsa:

```
shared out is begin o <- x end
. . .
i -> x; out (); out ()
```

Resulting in the generated LARD code:

```
forever (
```

```
    'C10_@6:23' ? this is the shared block
    ( 'P_o' ! Read ('V0_x[0..7]') )
  ) |
  forever (
    'V0_x[0..7]' := ('P_i' ? (?'P_i')) ;
    sync ('C10_@6:23') /* call */ ;
    sync ('C10_@6:23') /* call */
  )
```

If the call to this procedure was used as an activation then the shared block code would never terminate. The shared blocks could be re-implemented with local LARD procedures but it was considered to be more flexible to make the activation explicit in this way.

Test harness generation

Test harnesses for LARD simulations can be created by breeze2lard along with LARD models of Balsa circuits. The generated test harnesses can provide values to inputs to the circuit under test (either constant or from a file) and capture output of the circuit to file. Other test behaviours (e.g. connecting a Balsa described microprocessor to a simulated memory system) can be created by adding to the LARD test harness by hand.

Greater support for using Balsa itself to describe test fixtures for circuits is planned in future versions of the behavioural simulation system.

4.5.2. New balsa-c features

Two new features have been added to Balsa as part of this work: parameterised procedures and output selection.

Parameterised procedures

Parameterised procedures have arguments other than port connections which are used to determine aspects of the procedures' expansion when instantiated. These parameters must be constants at compile time as they are typically used to control the size and number of ports and channels within the expanded procedure. As an example, consider this definition of an `n`-input multiplexer with *`w`* `bits` wide inputs and output (the type `cardinal` is used in configuration to indicate any natural number):

```
procedure Mux (
```

```
  parameter w : cardinal;
  parameter n : cardinal;
  array n of input inp : w bits;
  output out : w bits ) is
begin
  if n = 0 then print error,"Parameter n should not be
zero"
  |  n = 1 then
    loop
      select inp[0] -> inp then
            out <- inp
      end
    end
  |  n = 2 then
    loop
      select inp[0] -> inp then
        out <- inp
      | inp[1] -> inp then
        out <- inp
      end
    end
  else
    local
      channel out0, out1 : w bits
      constant mid = n / 2
    begin
      PMux over w, mid of inp[0..mid-1],out0 ||
          PMux over w, n-mid of inp[mid..n-1],out1 ||
          PMux over w, 2 of out0,out1,out
        end
  end
end
```

This definition constructs the required multiplexer by recursive decomposition into 2-way multiplexers. The `n` and `w` parameters can be applied to produce an expansion of this multiplexer (in this case a 4-way, 32b multiplexer) so:

```
procedure Mux_4_32 is Mux over 4, 32
```

This parameter-applied version of `Mux` is bound to the name `Mux_4_32` for later instantiation.

Parameterised Balsa procedures are expanded by re-examining the code of the procedure with the parameters replaced by constants. It is necessary for balsa-c to dump out the source code for parameterised procedures to compiled Breeze files in order to allow other Balsa files to expand their own versions of the procedures.

Output selection

Balsa descriptions can be written in which behaviour can be affected by the arrival of one of a number of input communications. The Balsa `select` and `arbitrate` statements are used to express this input choice. An example of the use of `select` can be seen in the parameterised procedure example given in the previous section.

Tangram `select` semantics have previously allowed selection on both inputs and outputs. This is easy to provide in Tangram as the communication on which the selection is based is sequenced with the command to execute as a consequence of that communication. Input selections are therefore terminated in assignments to variables and output communications are sourced by expressions, both of which occur before the body command of the Tangram `sel … les` command.

Balsa `select` commands, however, enclose the body command within the input communication in order to allow the value on the input channel to be used (unlatched) inside that command. To allow behaviour to be affected by the order of arrival of requests for outputs, it would be useful if the command attached to the `select` end of communication could be performed before the output communication. Unfortunately, due to the use of reduced broad signalling for pull (output selection) channels, the value of the output expression must be constant until some time after the end of the communication making it difficult to exploit interesting overlappings of output communication and command activation.

Output selection is, therefore, implemented as a strict sequencing of two operations: the activation of the chosen command and the output communication. A single term of a selection can consist of a number of channels, all of which must arrive before the command is activated. If some of these channels are inputs, the activated command can make use of their values to perform some operations. This operated-on value can then be assigned to a variable before being communicated outwards by the output channels of the selection.

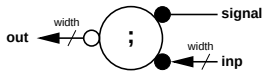
For example, an incrementer which receives requests from both its input and output, reads the input value and then returns an incremented copy of that value to the output after the end of the input communication could be described (where the output communication as part of `select` guard is expressed *channel* `<-` *expression*):

```
procedure AllPassiveInc (
  input i : byte;
  output o : byte ) is
variable x : byte
begin
  loop
    select i, o <- x then
      x := (i + 1 as byte)
    end
  end
end
```

This procedure is fairly useless as a buffer stage, however. The output and input communications are too closely interlocked. A more realistic use of output selection is to build monitors around shared resources. For example, this parameterised procedure could be used to provide a single read port, single write port arbitrated access shared variable:

```
procedure SharedVar (
  parameter t : type; -- parameter is a type
  input write : t;
  output read : t ) is
variable sharedVar : t
begin
  loop
    select write then
      sharedVar := write
    | read <- sharedVar then
      continue
    end
  end
end
```

The body of the read command is `continue` as no body command is required.



SequencePull

(**parameter** width : cardinal;
passive output out;
active sync signal;
active input inp : width **bits**)

#[out !° [signal ; inp ?• *inp*]]

Figure 4.6. SequencePull handshake component

Output selection is implemented in handshake components using the new SequencePull component. SequencePull acts like a Sequence component but has a pull activation port (`out`) on which a value from the second sequenced port (`inp`, which will be connected to the output

expression) can be connected. The body comand of the selection is connected (via the selection hardware used with input selection) to the first sequenced output of the SequencePull (port `signal`). SequencePull's symbol and behaviour are shown in figure 4.6.

4.5.3. Design management – balsa-md, balsa-mgr

Two tools have been written to help with design management: balsa-md and balsa-mgr. Balsa-md is a inter-file dependency analyser for Balsa files which can produce Makefiles to describe the building of Balsa projects and LARD simulation. Balsa-mgr is a graphical front-end for balsa-md which allows projects to be constructed and test harnesses for those projects to be described. Both tools can be extended to add new rules for driving the back-end tools described in §4.2.

Balsa-md is written in Scheme using the same codebase as the other back-end tools. Balsa-mgr is written in C using the GTK+ GUI toolkit for its graphical front-end.

4.6. Chapter summary

This chapter has described a programmable handshake component generator which can produce gate level implementations of all the handshake component described in chapter 3. A method for simulating Balsa designs by reconstructing behavioural models (written in LARD) from Breeze files. These models can then be executed to simulate the source design.

This back-end makes use of templates to perform its component expansions. It is hoped in future to adapt the notation described in §3.4 to allow many of these templates to be replaced by synthesis of behaviours described in that notation. In addition, future work will include the improvement of the source level debugging and simulation services offered by LARD.

Chapter 5. New Handshake Components

This chapter introduces some new handshake components. These components are intended to solve some of the problems with producing a direct, hierarchical implementations of handshake circuits generated from the current Balsa synthesis route. The aim is to produce handshake circuits which are composed of fewer, larger handshake components. Peephole optimisation at the handshake circuit level is used to replace clusters of existing handshake components with these new, more highly parameterised components which should help reduce the amount of gate level optimisation necessary to produce acceptably optimal implementations.

This chapter is composed of three sections:

1. A brief overview of the problems inherent in existing handshake circuit synthesis and some justification for introducing larger handshake components to help solve these problems.
2. Descriptions of the proposed new components and rules for their use.
3. A summary explaining the remaining problems which the introduction of these components do not solve.

Chapter 7 discusses the application of these components to the implementation of a substantial test design. Comparisons of this test design with and without the new components are quantified.

5.1. The trouble with Balsa handshake circuits

This section describes the problems with the handshake circuits approach when used as a general approach to design description. Limitations of the handshake circuits used with the 'old' Tangram system and the current Balsa system as well as some recent improvements to the Tangram system are discussed.

Handshake circuits provide an attractively simple way to implement asynchronous logic

circuits with channel based descriptions. The use of explicit request/acknowledge handshake signalling to implement communications between components mirrors well the structure of channel based HDL descriptions. This useful feature makes automated syntax-directed and hand-built implementations of channel based descriptions easier in handshake circuits than descriptions written in other paradigms (such as networks of state machines or RTL level descriptions). Unfortunately, not all designs find their most natural descriptions in the same paradigm.

Implementations composed from van Berkel's original handshake component set (the set used with the version of Tangram described in [9]) tend to consist of large numbers of small handshake components each with fixed (i.e. unparameterised) behaviours. The small number of available handshake component forms present in the provided handshake component set belies the fact that each component which a handshake circuit contains may be different, parameterised version of one of those components. This leads to circuits composed of large numbers of components selected from a library containing a large selection of parameterised components (which have previously been expanded from templates by a mechanism like that described in §4.2). Handshake circuit netlists therefore have a tendency to become an unwieldy mess of too many components selected from a library which is uncomfortably large and with too many interconnecting channels.

The Balsa handshake component set is largely the same as the original Tangram handshake component set. The same small, unparameterised control components are combined with the same connection components to produce similarly fine-grain handshake circuits. The Balsa components do, however, tend to have parameterised numbers of ports (e.g. *n*-port Sequencers rather than trees of 2-port SEQ components) to allow a number of like components to be combined in order to allow local, specific implementation optimisations to be applied to these components and for handshake circuits to be generated which are less cluttered by internal control channels.

The benefits of this principle of replacing a cluster of components with a single parameterised component whose implementation is locally optimised for area are clear. What is less obvious is the improvement in synthesis directness (the notion of directness was introduced in §2.2), especially when directness is applied to the path from handshake circuits to their

final implementations. Larger handshake components allow the netlists produced by the Balsa back-end to be more comprehensible and easier to understand by the designer. Larger components also make hierarchical place and route more practical which may help ease the problem of achieving timing closure.

Preserving hierarchy in handshake circuit implementations and keeping synthesis direct all the way to the gate/standard cell level is the easiest way to improve the link that the designer has between the input Balsa description and the structure of the final implementation of that description. Although there are direct implementations for each of the handshake circuits, and these direct implementations can easily be composed to form handshake circuits, these handshake circuits do not always reflect the best optimised gate level implementations of those circuits. A number of problems exist as either a consequence of direct circuit implementation or as outstanding problems within handshake circuits.

5.1.1. Signal drive strength management

Signal drive strength management is concerned with ensuring that each signal in an implementation is sufficiently well driven to ensure that it possible to drive the number of cell input loads to which it is connected (or, more usually in CMOS, the drive is large enough to produce an acceptably small delay and acceptably fast edge speed). This is usually only a problem with implementations intended to be fabricated as ICs where signal drives must be handled by the designer by either inserting buffering cells or by replacing existing cells with cells of greater drive strength.

The problem of inserting buffers into hierarchical handshake circuits was briefly discussed in §4.4.1. This was in relation to feed-through signals within handshake components and their implementations in the new Balsa back-end. Partial flattening of handshake circuit netlists was suggested as a solution with the additional drive provided by inserting components before the forks in signals exposed in the top level of the circuit by that flattening.

Signal drive strength problems also appear in Balsa handshake circuits around variable read ports. The signal forks which provide data to each of a Variable component's read ports are driven only by the latch cell which forms that Variable's storage element. The decision not to insert buffering (by default) in this situation was taken because the majority of read port

bits are immediately discarded by Mask or Adapt handshake components in order to make bitfield selections on variable contents. Drive strength problems also occur at the read ports of FalseVariables in the same way.

5.1.2. Control optimisation

The control in handshake circuits is composed of a small number of handshake components (Loop, Sequence, Concur, While …) composed to construct activation trees (as discussed in §3.5.1 and §4.5.1). These trees provide relatively conservative implementations of sequential and concurrent command composition with no overlap of handshakes in sequentially composed command and handshake independence on commands connected in parallel. Balsa does not currently perform any analysis on sequentially composed commands to determine if their activation handshakes could be overlapped (either in part or in whole) to allow more efficient hiding of the return-to-zero phases of activation handshakes. Also, parallel composed commands need not always have totally independent activation handshakes, a degree of synchronisation is often allowable.

For sequencing, Plana describes just such a system of overlapped sequencer circuits [61] based on the work of Martin [49], Josephs, Bailey [4], Kagotani and Nanya [42] and a system of data hazard avoidance methods when using these circuits. These circuits can be constructed from sub-components in the same way as existing Tangram sequencers are constructed from S-elements. Resynthesis techniques have already been discussed (in §2.4.2, §2.4.3) for building better control circuits to replace clusters of handshake components. These types of improvements to control components are not yet integrated into Balsa.

5.1.3. Channel construction

Many of the connection components available in Balsa exist only to provide synchronisations and merges in communications between sync channels and datapaths. These connection components are used to construct Balsa language level channels by connecting handshake channels from input/output commands and procedure instantiations. Communications on channels need to be directed to the input which is currently active in a way that preserves delay-insensitivity. Multi-cast communications also need to be routed to just those inputs

which are party to the multi-cast at this point in the description[1].

Van Berkel describes the Tangram compilation function for providing an active-input/output, passive interconnect solution which preserves delay-insensitivity in systems with simple, choice-free communications. The Tangram compilation function describes channel combination in compilation by using three-ported connection components which provide connections between two commands. The third port of each of these connection components continues the communication formed by that component on to other sequentially or concurrently composed commands. Peephole optimisation can reduce portions of these connection component trees into the more compact *n*-ported forms mentioned in the introduction to this section.

In single-rail implementations, the datapath fork and demultiplexing components are essentially identical to their dataless equivalents with just forked datapath wiring as an addition. The need to present channel-based interfaces to their environments makes the implementation of channel synchronisation in handshake circuits less elegant than in techniques such as micropipelines (§2.3.1) where datapath and control are separated and where bundling constraints need only be imposed at major interfaces.

Difficult synchronisation arrangements also come about where passive input operations on the same channel occur in several places within a body of code. With active input commands, the requests from the control tree are passed on to the connection components to be combined by C-elements into complete synchronisations. Sequenced inputs at different points in the circuit description can be combined using call elements. With passive inputs, however, Balsa currently implements channel communication/control synchronisation with a FalseVariable component followed by a network of DecisionWait and Synch components. In order to allow multiple inputs on the same channel, the request from the communication must be forked to all inputting FalseVariable components (or alternatively, a single FalseVariable could be used with forking of the 'signal' port requests), the acknowledgement to this request is sourced by only one input command with the request being withdrawn from all the inputs by the application of this acknowledgement. This arrangement is not delay-insensitive and as such is not compatible with a channel based implementation in Balsa. The CallActive component

[1]Multi-cast communications spread a single output communication onto many, parallel composed, inputs. e.g. `c <- 1 || c -> v1 || c -> v2` has the effect of assigning 1 to both of the variables `v1` and `v2`.

was provided to perform this type of forked request, single acknowledgement behaviour in non-DI channels. Figure 5.1 shows an example of the use of CallActive (shown as '|!') with a two point input on channel a.

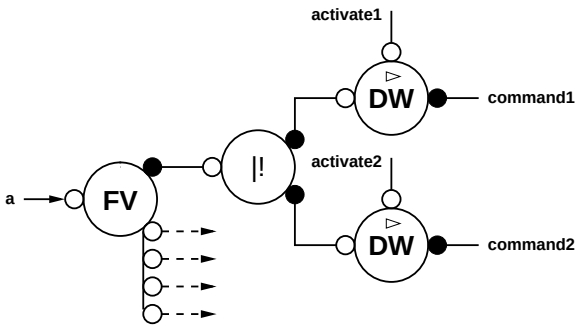


Figure 5.1. Sequenced passive inputs implemented with CallActive

All of these connection and conditional synchronisation combinations can be more elegantly implemented by the introduction of 'programmable' synchronisation/call components in which the synchronisation and call choices can be expressed as a specification string in a similar way to the Case component. Implementations can then be made by localising the isochronic wire-forks required to implement passive input structures and to simplify the construction of the Balsa channels by allowing the datapath and control to be separated within the new component.

5.1.4. Operations with individual signals

Where it is necessary to describe behaviour in terms of individual signal transitions a system based entirely on channel communications is insufficient. Signal level operations can be implemented in handshake circuits by using small interface components which have channel interfaces on one side and signal interfaces on the other. These components can perform operations such as: probing of signal levels, waiting on signal transitions and asserting levels on individual signals.

In order to make signal level descriptions possible, Tangram has adopted language constructs for handling the generation of transitions on individual signals and for waiting on input signal transitions. These language constructs have been adopted to make building interfaces between Tangram handshake circuits and their environments easier [57]. In particular, Tangram circuits

can be built to wait for clock edges to allow easier interfacing with synchronous environments [44].

Balsa does not include the ability to perform operations on individual signals. The signal level manipulation components are typically only used at the peripheries of circuits and so it is considered easy enough for the designer to make those components part of the circuit surrounding the Balsa synthesised components. Where individual signal manipulation is used inside a description, the signals can often be replaced by channels or by extra control.

5.1.5. Datapath operations

Directly compiled datapaths contain a rich source of potential gate level datapath optimisation opportunities. This is especially true where constants are used as inputs to datapath components. Consider this example of a Tangram/CSP style input selection operation between channels `c` and `d` which are read into variables `c_v` and `d_v` and for which the commands `C` and `D` are activated according to which input arrives first (the communication and body command are sequenced):

```
local
  variable s : bit -- store choice of input channel
begin
  loop
    select c then
      c_v := c || s := 0
    | d then
      d_v := d || s := 1
    end ;
    if s then C else D end
  end
end
```

The handshake circuit for this example is given in figure 5.2. Note that the assignments into `s` are made through a multiplexer attached to the constants 0 and 1 and activated by the two arms of the `select` command. As all data in Balsa is encoded in binary, a single bit variable composed of a single latch is used to store the value of `s`. This variable, the multiplexer (CallMux component), two transferrers and Constant components forming the two assignments to `s` could, therefore, be replaced by a single RS latch with a channel interface. Alternatively a single component could replace the push multiplexing of constants to allow the input to the variable to be encoded from the activity of the activation channels to the two, assignment

forming, transferrers. As the value assigned to `s` is, ultimately, decoded again by the `if` statement activating one of the commands `C` and `D`, it may also make sense to hold `s` in a one-hot encoded form. In this example, the true and complement outputs of the RS latch mentioned previously could be used. With more than two inputs to handle, more latch bits would be needed.

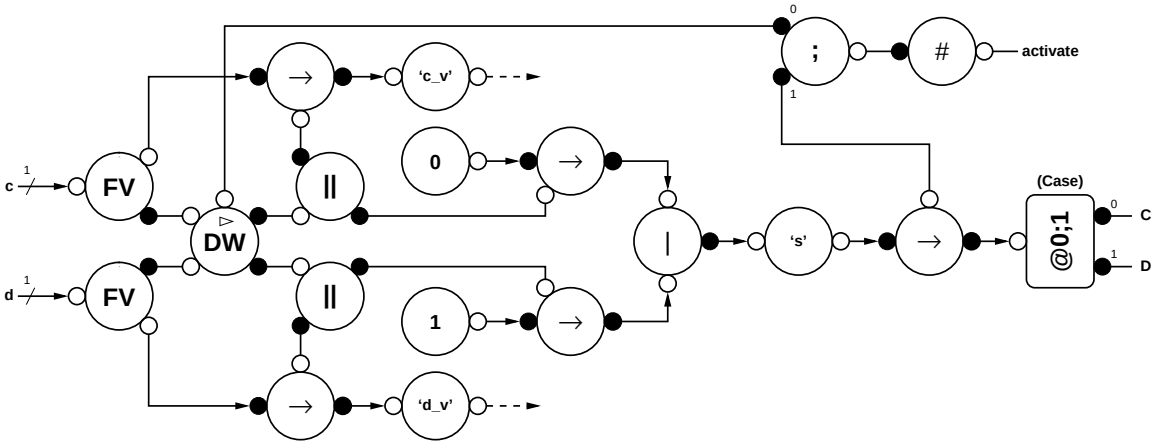


Figure 5.2. Tangram-style select handshake circuit

The current set of components present for implementing expressions make it expensive to build data operations which manipulate individual bits of a source word and construct a result word from these operations. Figure 5.3 shows the implementation of a bit reverse operation on a 4 b argument. This circuit involves gathering four arguments from the same source variable (`v`) (Mask components are used to select individual bits from the source variable) and then combining them into a single word using a tree of Combine components.

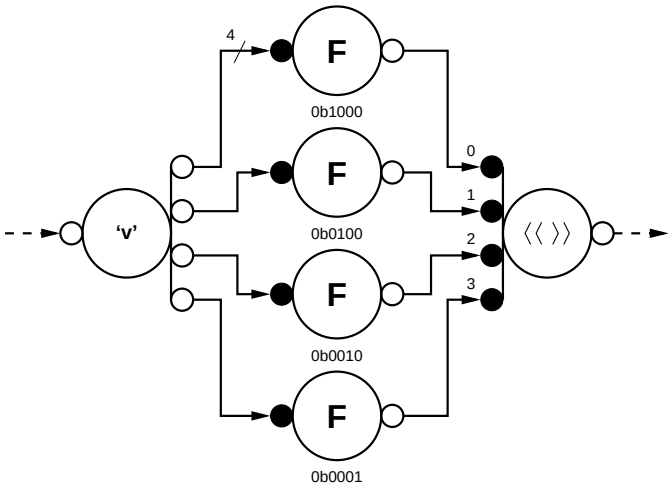


Figure 5.3. 4 b bit reverser handshake circuit

Unfortunately, each Combine component contains a C-element to synchronise the two incoming words. These C-elements are superfluous here as, when single rail bundled data is used, delay matching of the two variable reads can be combined and the C-element removed. Performing this optimisation on the technology mapped gate level netlist of the handshake circuit has been described by Peeters [58]. If gate level optimisation is to be avoided, other means of removing these superfluous datapath synchronisations must be employed.

This problem is exacerbated by the way that variables are split into Variable handshake components by the compilation process. This process breaks variables into small enough pieces that all variable writes can be satisfied by full-width writes on individual Variable components. For example, a 24b variable declared as: `variable v : array 3 of 8 bits` for which the assignments: `v[0..1] :=` ... and `v[1..2] :=` ... are performed will be split into three Variable components representing each of the three 8b portions of the variable. Those three portions represent the largest non-overlapping set of bitfields of `v` for which atomic writes (writing all bits of a bitfield) are possible. Writing to split variables requires the written data to similarly be split using a Split component. Split does, however, contain a C-element to synchronise acknowledgements returning on its output channels. This C-element is unnecessary for similar reasons to the redundancy of C-elements present in combining variable reads.

5.2. New handshake components

Compiled Balsa descriptions appear in Breeze netlist files as compositions of handshake components and nothing else. For this reason, it makes sense to attack the problems which Balsa suffers from by thinking of new handshake components. The components introduced in this section are distinguished from those described in chapter 3 by their use of specification string parameters (apart from Case and CaseFetch which are early examples of these component types). The specification strings are used to specify part or all of the behaviour of particular instances of these components.

The Balsa handshake component set can easily be extended. All that is needed is a description of the new components in both the template language described in chapter 4 and in LARD.

The specification string processing necessary to make template expanding decisions is provided by adding gate definitions and expression functions to the template language.

For example, the Case component is currently defined using a one-hot decoding generic gate called `decode`. A function called `complete-encoding?`[1] is also used. `Complete-encoding?` returns true when the Case specification string passed to it specifies an output for every input encoding. This information can be used to conditionally place an 'else' acknowledgement path in the Case component's expansion.

5.2.1. PatchVariable

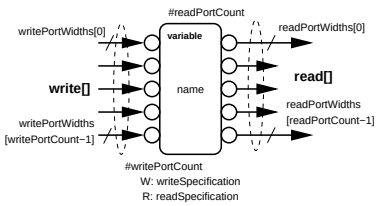
Variable components in Balsa have a single write port and as many read ports as are necessary to satisfy all of the points in a description which read from that variable. The problems of forks and unnecessary synchronisations in reads described in §5.1.5 stem from this organisation. Newer versions of Tangram [59] abandon variable read ports to remove these variable read synchronisations (a description of an alternative, gate level optimisation approach to this problem is given by Peeters [58]). This is an acceptable approach where data is bundled with control instead of having data-validity encoded with the data. Here the control path for the variable read ports is just a loop of wire from request to acknowledge with an option to insert a delay element or increase the drive on the data wires if timing constraints are not met. A Balsa Variable implemented this way looks much like the channel-loadable latch example shown in figure 1.4.

Future work on Balsa will include renewed interest in the use of DI codes to encode data (see §8.1. This being the case, read ports are an important and integral part of variable components (or at least the generation of a data encoded handshake by request of a transferrer or pull datapath component is required). It is also desirable to keep variable reads and writes together to allow the strict netlist form of Breeze descriptions to be preserved.

What is proposed is a more general Variable component with parameterised width write ports (with the multiplexing built into the variable) and parameterised width read ports. The read ports would, additionally, be parameterised to allow the permuting, concatenation

[1] A question mark at the end of a procedure name in Scheme identifies that procedure as a predicate (this is like the 'p' suffix used in other Lisps). The template language generally follows Scheme naming/typing conventions.

and sign-extension of read data (which may imply data recoding for DI codes) that would otherwise only be possible with Adapt, Mask, and Combine components. Such a generalised Variable component (named PatchVariable after its 'patchbox' read and write ports) is shown in figure 5.4.



PatchVariable

(**parameter** width : cardinal;
parameter writePortCount : cardinal;
parameter readPortCount : cardinal;
parameter writeSpecification : string;
parameter readSpecification : string;
parameter writePortWidths : **array** writePortCount **of** cardinal;
parameter readPortWidths : **array** readPortCount **of** cardinal;
array i in 0 .. (writePortCount - 1) **of passive input** write : i **bits**;
array i in 0 .. (readPortCount - 1) **of passive output** read : i **bits**)

Figure 5.4. Generalised variable handshake component – PatchVariable

The behaviour of this component is greatly affected by its parameters, as is its port structure (and so the behaviour is omitted from the description given). The `width` parameter specifies the total bitwise width of the set of latches within the variable, the `writePortCount` and `readPortCount` parameters specify the number of each of those port types. The most interesting parameters are the `writeSpecification` and `readSpecification`. The same form of encoding-specifying specification strings used by the Case component is used to describe the bitwise combinations of PatchVariable latch bits used to form read or write port connections. Repeated bits could be specified by a bit number and repetition count (to implement sign-extension, for instance).

In order to allow read and write ports with different widths, a new form of arrayed port needs to be defined whose individual port widths are determined by values in the specification strings. These new arrayed ports use the final two parameters `writePortWidths` and `readPortWidths` for their individual widths. This information is, strictly speaking, redundant as the two main specification strings provide the port widths implicitly. The `PortWidths` parameters are, however, decipherable in Balsa (they are arrays of cardinals) and so the heterogenous-width arrayed ports could be made consistent with the Balsa type system using the new syntax:

`array` *name* `in` *range* `of . . .`

The PatchVariable can be used to replace those Variable components broken up by bitfield assignments (e.g. assignments to elements of an `array` or `record` typed variable) or Variables whose read values are frequently combined with Combine in order to make their use more efficient. At the extreme, all the variables in a circuit could be replaced by a single PatchVariable although, in practice, only the common cases of broken and combined Variables need be combined. Consider this simple byte addressable register example:

```
procedure WordReg (
  input byteSelect : 2 bits; -- byte to read/write
  input byteWrite : byte;
  input wordWrite : 32 bits;
  output byteRead : byte;
  output wordRead : 32 bits
) is local
  variable reg : array 4 of byte
begin
  loop
    select byteSelect, byteWrite then
      byteRead <- reg[byteSelect]; -- read then write
      reg[byteSelect] := byteWrite
    | wordWrite then
      wordRead <- (reg as 32 bits); -- read then write
      reg := (wordWrite as array 4 of byte)
    end
  end
end
```

Without PatchVariable, this example would be implemented by four 8b Variables each with two read ports, one for the individual byte reads and the other to be combined into the single 32b read. Each variable has a multiplexer sourced write port taking data from the byte and word inputs. The old implementation of `WordReg` is shown in figure 5.5. Notice that the Case component used in this implementation already makes use of specification strings. A pair of (interim) components, SplitEqual and CombineEqual (with the same symbols as Split and Combine respectively), are also used to reduce the component count by not requiring individual $8b \times 8b \leftrightarrow 16b$, $16b \times 8b \leftrightarrow 24b$ ... Combine and Split components.

The synchronisations in the Split/Combine pair required to form the 32b wide ports could be saved by using PatchVariable. This saving would be more pronounced for a bit addressable register where 32 way Splits and Combines are required. Large amounts of the 'clutter' in handshake circuits is thereby removed by PatchVariable. Figure 5.6 shows the data portion

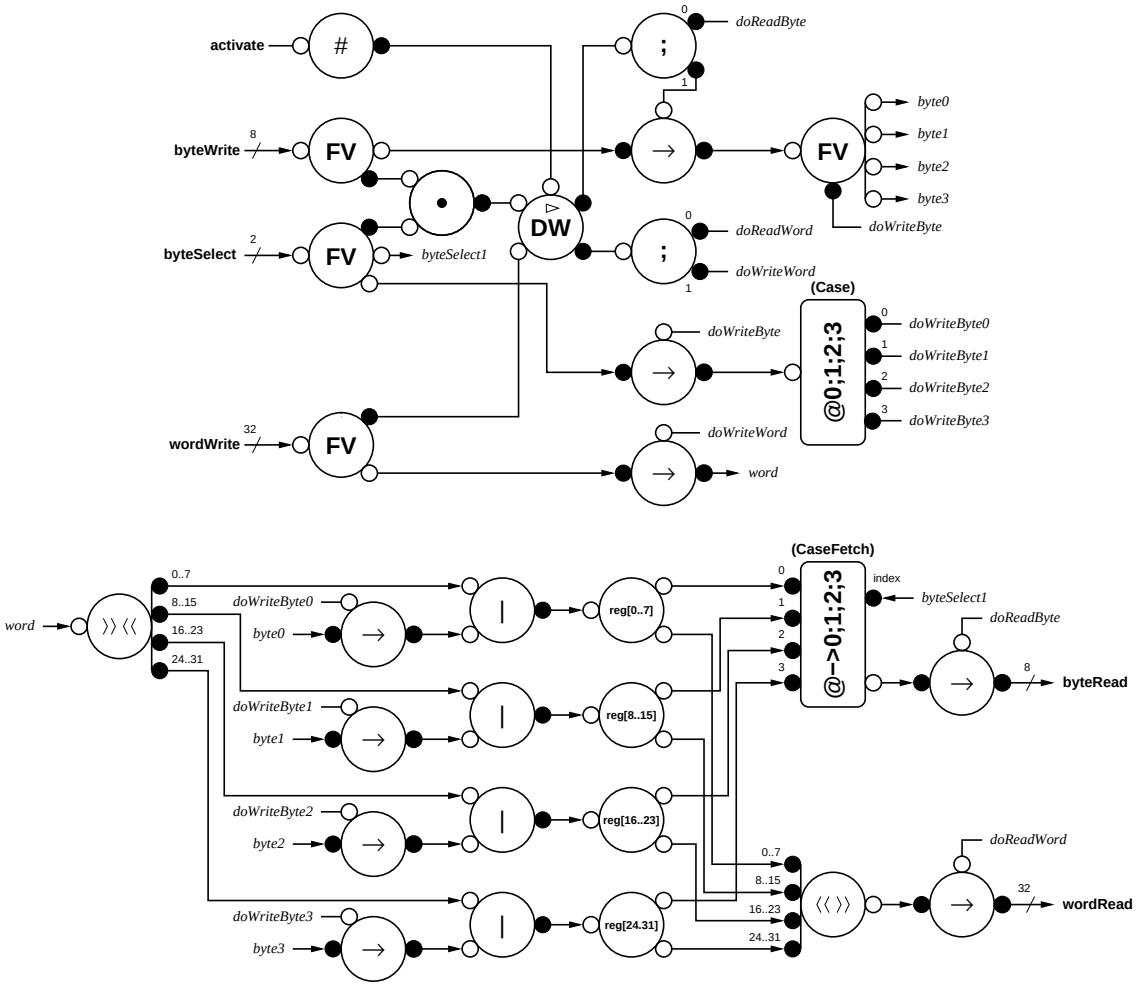


Figure 5.5. WordReg implementation using Variable components

of the same circuit (the control portion is the same as Figure 5.5) with a PatchVariable used to encapsulate much of the detail (components parameterised with specification strings are shown as curved boxes rather than as circles).

The elimination of the byte wide multiplexers in the new implementation reduced much of the write port complexity in this circuit. This is particularly useful as the multiplexing decisions for circuits with complicated variable writes are best made centrally.

PatchVariable is not the only component which could benefit from specification strings to select bits to multiplex or present to ports. PatchCallMux, PatchCallDemux and PatchFalseVariable components may all be potentially useful.

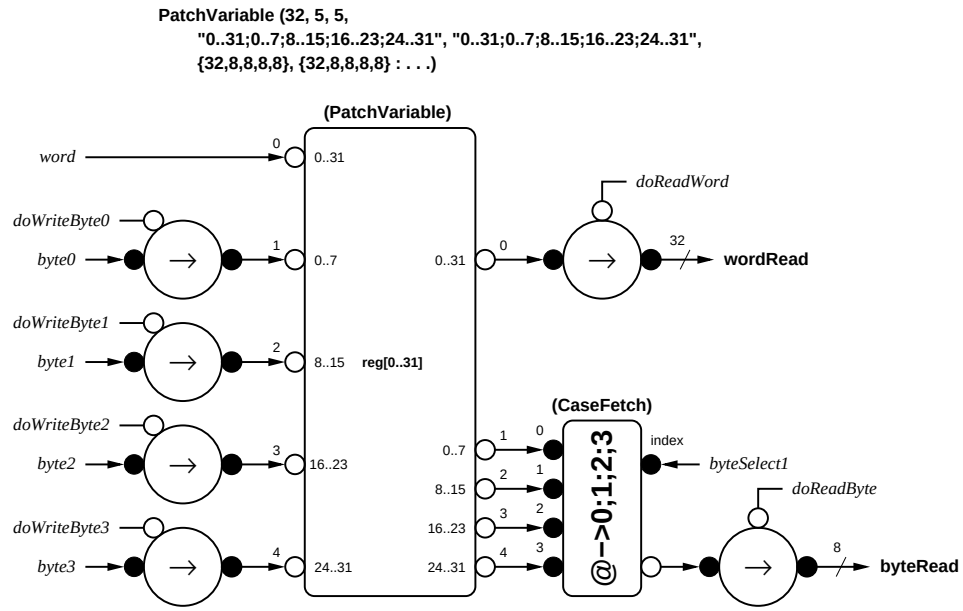
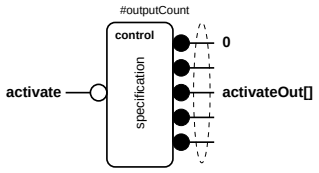


Figure 5.6. WordReg implementation using a PatchVariable component

5.2.2. ControlTree

Resynthesis of control trees in handshake circuits is a popular form of optimisation. The Loop, Sequence, Concur and Fork components can easily be combined in a single control component to make the circuit partitioning involved in this task easier (the port structure of such a component is simpler if While, Bar and Case components are omitted). A new component, ControlTree, with a specification string describing a subset of the handshake notation described in §3.4 could be used to parameterise the component. ControlTree's symbol and port structure are shown in figure 5.7.



ControlTree

(**parameter** outputCount : cardinal;
parameter specification : string;
passive sync activate;
array outputCount **of active sync** activateOut)

Figure 5.7. Generalised control tree handshake component – ControlTree

ControlTree is not only of use where resynthesis is used. Sequence and Concur components form the majority of control trees in Balsa. Both of these components are built from S-elements to allow the greatest independence for handshakes connected to their passive ports. It

is only strictly necessary to place the S-elements in a control tree at the tree's leaf channels. Control trees could, therefore, be replaced by 2-phase Concur and Sequence implementations (like the cheap Fork and Fetch components) with S-element handshake components on the leaves (or ControlTree could be used for more flexibility).

The Call combination of activation channels to access shared resources could also be incorporated into ControlTree. A Call consists of a decision-wait structure in order to handle acknowledgements (just like the call element described in §2.3.1). This decision-wait element could be removed where it sources acknowledgements only to S-elements (e.g. the output activations of Concur or Sequence components) or single inputs of C-elements (e.g. the output activations of Fork components) as both of these components are tolerant of spurious ackknowledgements whilst otherwise idle. Encapsulating this reduced-Call in ControlTree may allow the extent of the isochronic forks formed by the forked acknowledgment to be controlled.

5.2.3. PassiveConnect, Connect

Connection networks connecting only sync channel connections fall into two categories: those which connect only the active ports of other components (i.e. are all-passive) and those with both active and passive port connections. The Passivator is the simplest example of an all-passive interconnect component, but not the only one. Consider this Balsa description:

```
select c then
  continue
| d then
  continue
end
```

A compiled form of this description contains a DecisionWait to implement the `select` command, connected to channels `c` and `d` and sourcing requests to Continue components (which are nothing but loops of wire from request to acknowledgement). Despite the presence of the Continue components, this is a useful description. It finds applications, for example, where tokens are passed by dataless handshakes and a decision must be made between multiple sources of tokens.

Unfortunately, there is no current single handshake component to implement this `select` structure. Where `c` and `d` are sourced from more than one place (e.g. from a command: `sync c; sync d; sync c`) or are part of a multi-cast, other connection components

will be connected to the synchronising components forming this `select` command. In the same way as Call's within control trees may be optimised (see §5.2.2), amalgamations of synchronising/call components may be combined to better advantage. The centralised, all-passive, synchronisation/call component PassiveConnect is proposed. Its port structure and symbol are shown in figure 5.8. A similar component Connect (also shown in figure 5.8, not to be confused with the Tangram CON straight connector component) is defined for use with sync channel interconnect with sourcing (otherwise irreducible) passive inputs. These two components can replace all of the components: Synch, Passivator, DecisionWait, Fork and Call; either alone or in their various combinations.

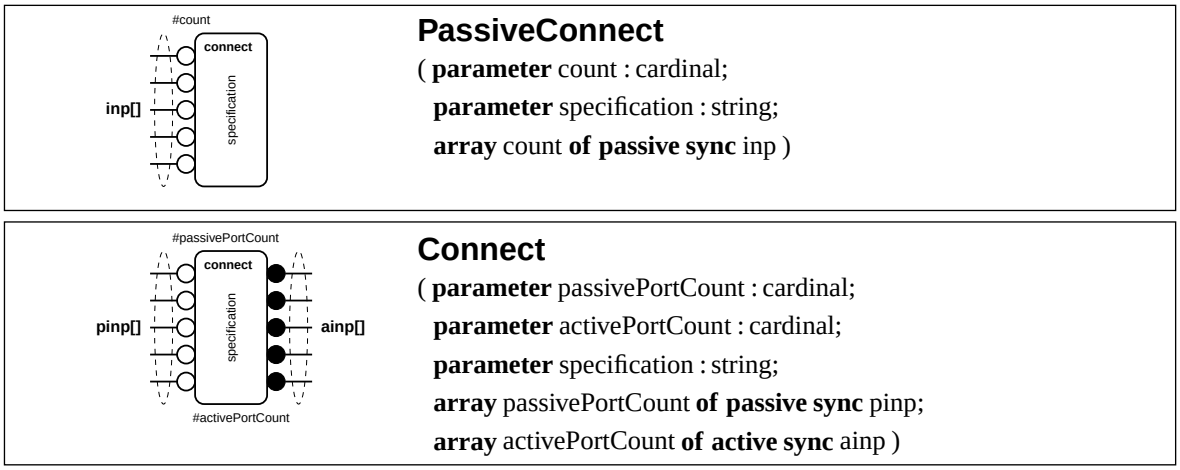


Figure 5.8. Generalised sync interconnect – PassiveConnect, Connect

Similar generalised connection components can be defined for data-bearing communications. Four groups of ports would be necessary on these components: passive inputs, active inputs, passive outputs and active outputs. Components with only active connections are not possible, neither are Balsa components with arrayed ports with no connections. Data-bearing components with either no inputs or no outputs are similarly useless (usually). The required combinations of components would include all nine combinations of {passive inputs, active inputs, both senses of inputs} $\times$ {passive outputs, active outputs, both senses of outputs}. These components would need the heterogenous arrayed port structures described for the PatchVariable component (§5.2.1). This complexity is probably suggestive of the inadvisability of adopting these components. Future work on datapath manipulation (see §8.1) will include investigations of the partial separation of data and control. This may include the simplification of the handshaking within data interconnect components by using the Connect component to

control dataflow.

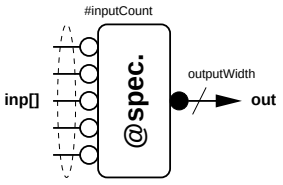
5.2.4. Encode

There are many examples of cases where the CallMux components leading to variables or output communications need to select between constants. In these cases, direct implementations will result in multiplexer cells with one or more inputs tied off to 0 or 1. For example, consider this (rather contrived) example of a 2b complement operation:

```
case v1 of -- v1, v2 are 2 b variables
  0 then v2 := 3
| 1 then v2 := 2
| 2 then v2 := 1
| 3 then v2 := 0
end
```

The handshake component implementation of this example is shown in figure 5.10. The four Constant components sourcing the CallMux (via four transferrers) are multiplexed between to form the write port to `v2`. The equivalent flattened multiplexer organisation for this implementation is also shown in figure 5.10. This arrangement is clearly a very inefficient way of implemented the desired value encoding where gate level optimisation across handshake components is undesirable.

The activation channels connected to the transferrers sourcing the CallMux effectively act as a set of one-hot encoded input selects for these multiplexers. More optimal implementations can, therefore, be produced by logic optimisation of an expression involving only these select inputs. The component Encode (figure 5.9) is intended to replace the Constants, Fetchs and CallMux in this arrangement with just such an optimised implementation.



Encode

(**parameter** outputWidth : cardinal;
parameter inputCount : cardinal;
parameter specification : string;
array inputCount **of passive sync** inp;
active output out : outputWidth **bits**)

Figure 5.9. Encode handshake component

Encode performs the reverse operation to the Case component. Case maps an incoming binary word into activity on one of a set of one-hot active outputs (encoded on sync channels).

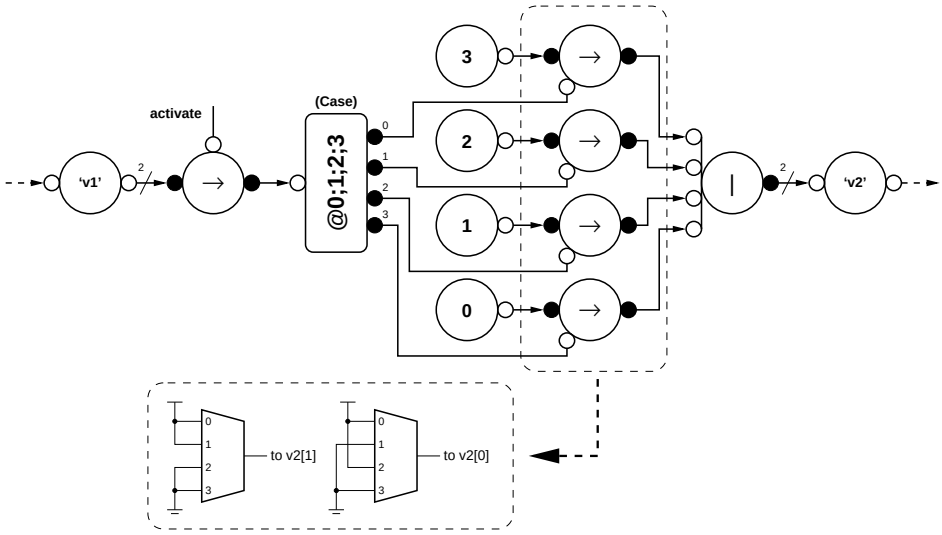


Figure 5.10. 2b complement operation

Encode is used to map one-hot encoded inputs back into binary. Using Encode, the `case` command example given above can be implemented with one Case component and one Encode component (as shown in figure 5.11. A similar arrangement can be used to implement any logical function albeit with a one-hot encoded intermediate.

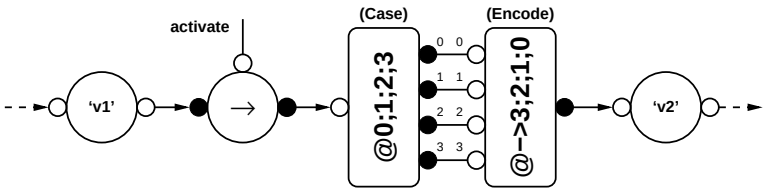


Figure 5.11. 2b complement operation – using Encode

Encode specification strings must specify the input values which map to each output. The gates for these encodings can be built using the same logic optimisation used to construct Case. An `encode` gate added to the template language is all that is needed to express this use of optimisation. Both `encode` and `decode` are handled by the Balsa back-end by running an external logic optimiser.

5.3. Chapter Summary

This chapter has introduced a number of new components for use with the Balsa design flow. Of these components, only Encode is currently generated by balsa-c. The peephole

optimisation patterns for the other components have not yet been integrated into balsa-c. The optimised implementation of the DMA controller described in the next chapter has, therefore, had these components applied by hand.

The optimisations described in this chapter do not address the removal of transferrer components or (for the most) part improvements in datapath synthesis. These two points are related as a potential form of optimisation of data operations in handshake circuits is the transformation of the 'pull expression' to 'push lvalue' structure formed by transferrers into entirely push pipelines. This is considered to be future work (§8.1).

Chapter 6. The AMULET3i DMA Controller

This chapter presents the design and implementation of a DMA controller using Balsa. This controller is a mixed asynchronous/synchronous design built for the AMULET3i macrocell [32]. This controller was implemented using a combination of Balsa, full-custom layout and hand designed standard cell logic. The Balsa portions were implemented without the new components detailed in the previous chapter and as such it represents a validation of the Balsa design flow using the new back-end with a substantial design example.

The DMA controller communicates with the CPU, memory and peripherals through MARBLE asynchronous macrocell bus interfaces to allow them to be integrated into different versions of the AMULET3i macrocell.

A second, fully asynchronous, DMA controller is present in the next chapter.

6.1. Suitability of DMA controllers as Balsa examples

A DMA controller has a number of properties which make it a suitable example design. These include:

Substantial size

A multi-channel DMA controller contains both a considerable number of configuration registers and also the mechanism required to select between these registers. In addition, the apparatus to deal with incoming requests from multiple DMA request sources is of significant size.

Autonomous control

Once programmed, the controller will perform autonomous transfer operations with little or no external stimulus. The speed of the controller can be measured by the period of these transfer loops and so the complexity of the test bench required to drive the controller can be kept to a minimum.

Register based operation

The transfer performing portion of the controller can be constructed as a sequential loop of register read; transfer; register update operations. Building the control this way using shared registers in preference to communicating sub-controllers gives greater scope for optimisations which improve register access efficiency and optimisations which resynthesise sequential control than a more highly concurrent design.

Arbitration issues

Access to the configuration registers is possible from both the CPU and the controller's transfer apparatus. Mutually exclusive, arbitrated accesses to the register bank must be provided by the controller. In both the AMULET3i and fully asynchronous control descriptions, situations are described where the arbitration for the DMA registers and arbitration for access to the bus can potentially conspire to cause deadlock. Solutions to these problems are described which involve the partition of the controller into (large) parallel threads of operation. These partitionings are good examples of the use of parallelism and channel communications to solve fundamental design issues.

6.2. The AMULET microprocessors

The AMULET group's main research effort to date has been the development of existence proofs that practical microprocessors can be constructed using asynchronous design techniques. Three processors have so far been developed. All three were designed to execute unmodified ARM binaries on architectures appropriate to the ARM architecture and asynchronous implementation. These processors are:

AMULET1 [56] – A test piece demonstrating feasibility of a full custom asynchronous processor implemented with Sutherland's 2-phase micropipelined methodology. AMULET1 was just a microprocessor core on a single die, presenting a 2-phase asynchronous interface to off-chip I/O. AMULET1 has been fabricated on two different silicon processes requiring only mask scaling to work correctly.

AMULET2e [27] – An embeddable microprocessor with on-board memory/cache and

peripherals presenting a conventional looking microprocessor bus interface to off chip RAM and peripherals. AMULET2e represents a move from a pure proof of approach to a usable device and also a move to 4-phase signalling. The AMULET2e die includes a processor core (the AMULET2), 4KB of memory which can be configured as either fast RAM or as a 64-way associative cache, an off-chip memory interface using an on-chip delay as a timing reference and an 8b off-chip I/O port. AMULET2e has been fabricated and a number of demonstration boards have been constructed using the device.

AMULET3i [30] – A processor, RAM, peripherals hard macrocell held together by an asynchronous macrocell interconnect bus which also presents a synchronous interface to on-chip synchronous peripherals. AMULET3i is integrated into the otherwise synchronous DRACO communications IC.

6.3. AMULET3i and DRACO

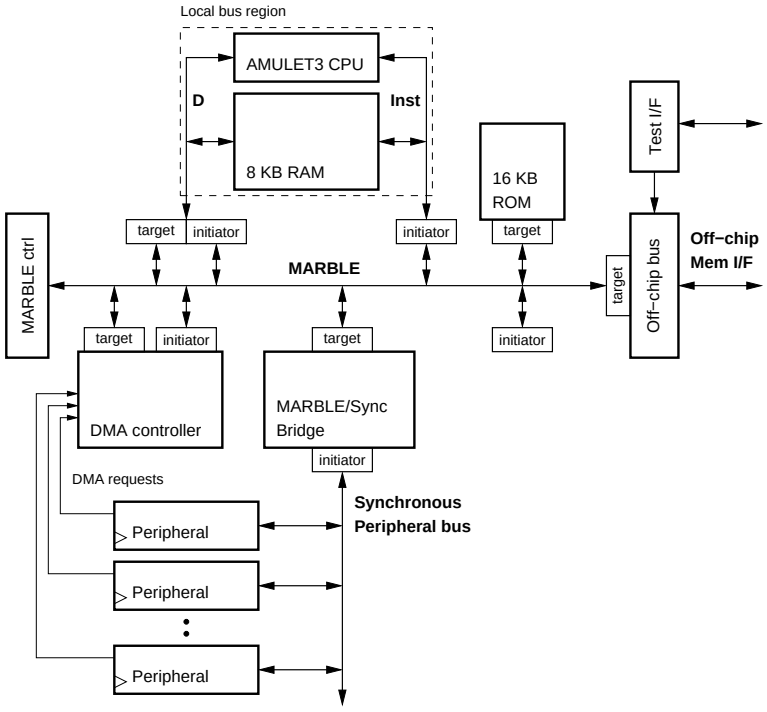


Figure 6.1. The AMULET3i macrocell

The AMULET3i macrocell forms the main current research effort of the AMULET group. The AMULET3i macrocell consists of a number of components built using a variety of tools and connected together by a macrocell bus and connected to off-chip and on-chip

synchronous bus interfaces. AMULET3i's first application will be as part of the DRACO communications IC (shown in figure 6.2). The components of AMULET3i are:

AMULET3 – AMULET3 is a 4-phase micropipelined implementation of a processor capable of executing the ARM v4t instruction set [3]. The processor consists of roughly 50% (by area) of custom designed datapath and 50% of standard cell logic. The standard cells used in this and other components of the whole macrocell were designed by ARM Ltd. to a proprietary set of portable design rules for 0.35 µm and 0.25 µm 3LM CMOS processes. In addition, a large number of peculiarly asynchronous cells were designed by group members to improve the performance of controllers present in the design. This group of cells consists mostly of symmetric, asymmetric and complex gate input C-elements and were designed to the same portable design rules as the stock 'synchronous' cell library.

The custom datapath was designed to handle single rail data with bundled control and delay matched bulk delays. Data dependent delays only exist where early completion of iterative instructions (e.g. multiplies) or forwarding of register data from the queue mechanism (a register forwarding mechanism novel to AMULET3 in the arena of asynchronous processors [33]) reduces the execution time of instructions. Connections between the standard cell control and the datapath provide the flow control through the datapath.

The controllers in the standard cell blocks were designed partly by hand and partly using petri-net synthesis using the tool Petrify [19]. Petrify produces equations describing the implementation of SI controllers which a skilled operator can then map onto the available standard cells. A large proportion of the complex cells in the locally built cell library are present only because they are required to optimise critical paths either synthesised or hand built controllers. The use of Petrify has aided the construction of correct controllers (which have, never the less, been hand optimised afterwards) and also served as a verification tools for hand designed controllers with difficult to understand behaviours.

The whole AMULET3 processor core macrocell consists of about 100 000 transistors and occupies approximately 4 mm^2 of silicon or about 20% of the whole AMULET3i macrocell area.

MARBLE – The Manchester Asynchronous Research Bus for Low Energy is a fully asyn-

chronous multi-initiator macrocell interconnect bus used to connect the AMULET3 macrocell to its memory and peripherals. MARBLE consists of single rail bus wiring and a centralised controller providing arbitrated access to the bus and a single point of address decoding.

8 KB RAM – The RAM is connected directly to the processor core by two 'local' buses: one for instructions, the other for data. These buses are in turn connected to MARBLE through a pair of bridges: one initiator bridge to allow the processor to access MARBLE and one target bridge to allow other bus initiators on MARBLE (notably the DMA controller) to access the RAM. The RAM is organised as 8 blocks of 1KB with independent pipelined access to each block. The RAM is constructed from full custom cells.

MARBLE/Synchronous Bus Bridge (MSB) – The MSB is a target interface connected to MARBLE which allows simple strobed synchronous peripherals to be connected to the AMULET3i macrocell. This interface is used to perform all transfers between the AMULET3 and synchronous communications peripherals in DRACO.

DMA Controller – A 32 channel DMA controller used primarily to transfer data between peripherals, synchronous RAM and the 8 KB of on-macrocell RAM. The DMA controller is connected to MARBLE through two interfaces: A target interface through which the controller is programmed and an initiator interface by which the controller performs its transfers. Transfers can either be performed free running (such as a block to block memory copy) or be initiated by a DMA request signal. The DMA requests in DRACO all come from the accompanying synchronous peripherals and so the part of the controller which maps incoming requests to DMA channels is a regular array of multiplexer cells feeding into synchronous state machines, one for each DMA channel. The remainder of the DMA controller is made up of a number of full custom register bank blocks containing the controller channel state and a large block of standard cell logic synthesised from a Balsa description with a few dozen extra cells of hand designed logic.

Test Interface – The test interface is an 8 b wide off-chip interface acting as a MARBLE initiator to allow test data to be fed into the processor for production testing and design validation.

16 KB ROM – The ROM is 8 b wide and is 'mask programmable'. It is used to bootstrap

the macrocell and may contain a program specific to the ICs intended application. The ROM consists of a MARBLE target interface and a few hand designed gates (actually the result of optimising Petrify output) in standard cell along with a compiled non-self-timed ROM block constructed by a ROM compiler shipped with the CMOS process.

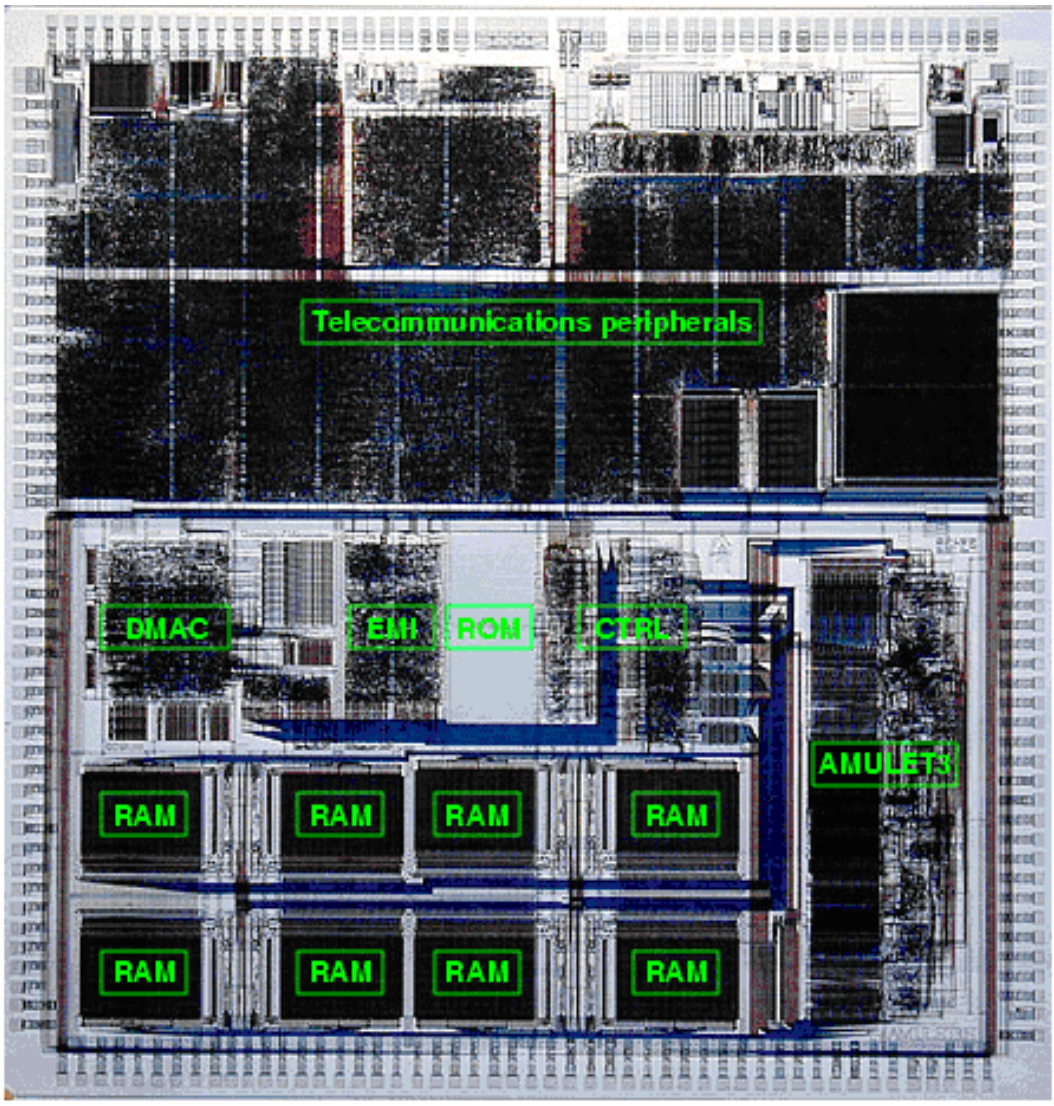


Figure 6.2. DRACO communications IC (AMULET3i is the lower half of this IC)

The AMULET3i macrocell was constructed using a mixture of design approaches and implementation styles with hand designed and synthesised standard cell, full custom datapath and memories and compiled regular blocks. When it is fabricated it will be one of the first asynchronous microprocessors implemented for commercial use (the Tangram-synthesised 80C51 [31] is arguably the first). AMULET3i also executes the same instruction set as the, already widespread, ARM microprocessors and could be used as a replacement for these processors in

system-on-chip applications where its potential low power and low EMI properties would be useful.

Both of the DMA controllers presented in the remainder of this chapter are suitable for use with AMULET3 and MARBLE. The hybrid asynchronous/synchronous controller is actually embedded within AMULET3i whereas the simpler, fully asynchronous, controller is provided for applications where an AMULET3i-like macrocell is used in a system with asynchronous peripherals (perhaps with a MARBLE to asynchronous peripheral bridge).

6.4. The AMULET3i DMA controller

The AMULET3i DMA controller's role is to transfer data between memories (both on the AMULET3i macrocell and on the synchronous peripheral bus) and the synchronous peripherals present in a complete SoC. These transfers involve processing synchronous requests from peripherals and interfacing to MARBLE to perform those transfers. The synchronous nature of the arriving requests and the peripherals would seem to suggest that the DMA controller should itself be synchronous and located on the synchronous bus. This was not possible, however, due to the simple, strobed nature of the synchronous bus which is only capable of supporting a single bus initiator. For both the AMULET3 and the DMA controller to perform accesses on the synchronous bus in such an arrangement, the MSB would be required to support accesses initiated on MARBLE from initiators on the synchronous bus. In order to reduce the complexity of the MSB, this behaviour is not supported so requiring the DMA controller to be placed directly on MARBLE and making an (at least partially) asynchronous implementation possible and desirable.

In order to perform transfers initiated by synchronous DMA requests, the controller is composed of a mixture of asynchronous and synchronous units bound together by asynchronous control synthesised from the language Balsa. The use of synthesis to implement the control portions of the design allowed the DMA controller's structure to be rapidly re-engineered in response to the changing requirements of the provider of the synchronous peripherals.

6.5. DMA controller requirements

The DMA controller was designed primarily to transfer data between peripherals and memory by initiating pairs of MARBLE read-then-write operations between pairs of units. There are many complicating factors:

- Only a certain number of transfers should be performed for each DMA 'run'. Count registers are required to keep track of the number of completed transfers.
- Memory accesses must be performed on sequential addresses. Source and destination address registers are required to hold the addresses of peripherals and memories. Incrementers are needed to update address registers after memory accesses.
- A *DMA client request* signalling mechanism is necessary to allow peripherals to signal their readiness to be transferred to/from.
- To allow many peripherals to have outstanding transfers, a number of entire sets of count, address and control registers must be kept, one for each DMA *channel*.
- A DMA request to channel request mapping table and associated mapping hardware is needed to map peripherals to DMA channels.
- To decide which of a number of outstanding requests to service first, a *priority ordering* of requests must be implemented. This ordering can be based on the order of channel numbers to which requests map.
- Each channel requires a number of control bits to specify such things as the DMA request number for this channel, whether to increment addresses and decrement the count registers for each transfer, whether the channel is enabled or not.
- Free running and memory to memory transfers without DMA request signalling must also be supported.
- In DRACO, the peripherals may each have a set of addresses from which data must be transferred on each request. To support this, a request 'chaining' mechanism is implemented requiring extra control bits for each channel.

The design of the DMA controller was influenced greatly by the structure of the peripherals outside the AMULET3i subsystem. The designers of these peripherals specified that the DMA controller would be required to support 16 DMA request signals and 32 channels in

order to accommodate the convoluted series of chained transfers required when a number of peripherals are in use. Ordinarily, a DMA controller would have a more modest number of channels (typically 4 or 8) and a larger number of requests than channels. The provision of as many as 32 channels results in a total of 3648b of state using 32b addresses and count values and 18b control (the need for 18b of control is explained later). These state bits would be held in the DMA controller's register file.

To reduce the size and number of registers required to hold the control, count and address information, the channels are partitioned into two types: *long* channels with full 32b addresses and count registers and *short* channels with 16b registers. In addition, to reduce the complexity of the request to channel mapping hardware, the short channels are further divided into *head* and *chain-only* channels. The head channels are capable of receiving DMA requests from peripherals and so require request handling and mapping hardware. The chain-only channels, on the other hand, can only receive requests as subsidiary transfers in a chain of transfers initiated by a head or long channel. The addition of these channel type distinctions does add to the complexity of the register bank control in the completed controller, however. After introducing the new channel types, only 2240b state bits are required.

6.6. The anatomy of a transfer

A transfer begins with a DMA request arriving on one of the DMA request signals shown at the bottom of figure 6.3. The request is presented to the synchronous peripheral interface (the SPI). The SPI filters incoming requests, maps client requests onto DMA channels and cleans up request signals before ferrying them on to the transfer engine.

The transfer engine is the power-house of the DMA controller. It receives requests from the SPI and initiates transfers. When initiating a transfer, the transfer engine will interrogate the register bank control for channel register values to pass on sequentially as read and write addresses to MARBLE using the initiator interface. The initiator interface is shown in figure 6.3 as an intermediate between the transfer engine and the MARBLE initiator interface. It acts to 'buffer' requests for transfers allowing requests for subsequent transfers to be handled by the SPI and transfer engine in parallel with the current transfer. The initiator interface's

operation and the mechanism used to update register bank registers are explained in §6.8.

Performing a transfer is a relatively simple operation. Unfortunately, there are problems (described in §6.8) inherent in having the register bank as a shared resource accessible from both the target interface on MARBLE and the transfer engine. These problems complicate the way in which transfers on MARBLE and requests for register bank access are interleaved.

6.7. Handling DMA requests – the SPI

In a completely asynchronous environment, arbiters are needed to select between unsynchronised incoming requests. In AMULET3i, however, the peripherals are synchronous and so provide clock synchronised DMA requests. Where requests arrive in such a synchronised manner, arbitration is not only unnecessary, it is unwise. The likelihood of an arbiter signal becoming metastable and so requiring a possible lengthy resolution is increased if all the input signals are presented within the same, short period of time. For this reason, it was decided to implement the SPI using synchronous techniques.

The SPI controls the mapping of 16 incoming synchronous peripheral requests onto DMA controller channels and the filtering out of requests for disabled channels. For this reason the channel enable and request number to channel number mappings for all channels are stored in the SPI. In addition, a global *fake request* register is included at the front of the SPI. The fake request register allows the SPI to be tested by introducing software generated requests at the front of the request to channel mapping block. The SPI's registers can be programmed from the register bank control (and so from the target interface) using the address/data bundles shown in figure 6.3.

Each incoming client request is processed by the SPI using a small state machine, one machine per DMA channel. These state machines allow incoming requests to be latched and for the register bank control to be able to set and reset requests. The register bank control resets transfer requests in the SPI at the end of each transfer. Requests can also be set by the register bank control to allow free running transfers to be enabled for that channel. The channel state machines act as modulo-3 saturating counters counting requests. Each new request increments the request counter and each request reset from the register bank control decrements the re-

quests count. Keeping count of outstanding requests allows a new DMA request to be issued by a peripheral as soon as a previous request has begun to be acted upon. In this way, a peripheral which is read by a DMA transfer need not wait for the whole transfer to be completed before requesting a second transfer. A peripheral sourcing data for a transfer can use the DMA initiated read on itself as an implicit acknowledgement for its DMA transfer request. The peripheral can then happily signal a second transfer to take place, the request for which the SPI duly counts. That second request will not be acknowledged (the DMA initiated read take place) until after the first transfer has been completed and so only a maximum of 2 outstanding requests need be counted on each channel, hence the modulo-3 counter behaviour. The use of a synchronous SPI implementation allows the request signal to be 'synchronously pulsed' so giving greater flexibility to the designer of peripherals in the way that transfer requests may be generated.

The processed requests from the channel request state machines are bundled together and presented to the transfer engine as a single word. The transfer engine selects a channel on which to perform a transfer by applying static prioritisation of channel requests with the channel priorities being set by the channel number; channel 0 has the highest priority and channel 21 has the lowest priority.

6.8. Accessing the registers

On receipt of a DMA request, the transfer engine requests a copy of the register contents for that request's channel from the register bank control. After performing the two initiator bus transactions of the transfer, updated copies of the channel register values are retired to the register bank. This control interaction can be achieved in a number of ways:

6.8.1. Single register bank access with locking

The transfer engine could lock the register bank while reading register values. The updated values could then be calculated by the register bank control and written back when the transfer engine frees the lock. In this way, the incrementers used in register value update can reside in the register bank control block and save on transferring the value back from the transfer

engine. The channel status bits could be updated while incrementing register values as the transfer would actually have been completed.

The register bank is locked during the entire period of the transfer. This means that the transfer engine needs no local registers to hold the channel register values as the register bank read ports can retain these values during the transfer. Unfortunately, locking the register bank prevents access to the register bank from the target interface while the transfer is taking place. If another initiating device has claimed the bus to access the DMA controller's target interface and the transfer engine has already locked the register bank, read register bank values and is preparing to read from the bus using its initiator interface, then neither the transfer engine nor the other initiator device can proceed and so the system becomes deadlocked.

This type of bus management hazard can occur on split transfer and atomic transfer buses. In atomic transfer buses it can be fixed by the transfer engine claiming the bus (without issuing an address) before locking the register bank. The two bus transactions which make up the DMA transfer could then be carried out with bus locking enabled to prevent another initiator attempting to read the locked DMA register bank. In a split transfer system the address and data portions of the bus must be claimed (arbitrated for) separately so it is possible for an initiator to have already issued an address to the DMA register bank control and then relinquished control of the address portion of the bus before the transfer engine starts its transfer. If the transfer engine then locks the register bank control and claims the address portion of the bus before the initiating device can claim the data portion of the bus and complete its transaction, the transfer engine may be prevented from then claiming the data portion of the bus for its transaction and once again the system is deadlocked.

6.8.2. Two sequential register bank accesses

The transfer engine could make two passes at the register bank and update register values itself performing the transfer between the two register bank accesses.

This approach solves the possibility of deadlock that the single register bank access approach suffers from by decoupling the transfer engine from the register bank during the actual data transfer operation. Unfortunately it also requires two, completely sequenced, operations on

the register bank for each transfer.

6.8.3. Two accesses with parallel write-back

The register bank could update register values and the transfer engine would make only a single pass at the register bank. The transfer itself can be performed after transferring values from the register bank into local registers. An indication that a run of transfers has been completed (to update the status bits/signal interrupts) could be sent to the register bank separately.

This approach allows the second of these register bank operations required by the split register bank access (the updated register write-back) to be performed in parallel with the DMA transfer. Unfortunately, this decoupling of write-backs and transfers requires a separate end-of-run signal to indicate that the final transfer of a run has been completed. End-of-run register status and interrupt signalling need to be triggered by this signal instead of the issuing of the final transfer so that interrupts (or polled responses to end-of-run) are not raised before the final transfer has actually been completed.

6.8.4. The Initiator Interface

In the complete DMA controller design, the transfer engine has a companion process: the initiator interface. The initiator interface actually performs the DMA transfer option with the MARBLE initiator interface and also signals end-of-transfer to the register bank which will ultimately signal a CPU interrupt if necessary. Figure 6.3 shows the connections between the transfer engine, the initiator interface and the register bank control.

The initiator interface allows transfers to be decoupled from request processing by the transfer engine allowing register update, the DMA transfer and the processing of requests can all be performed in parallel.

6.9. Structure and implementation

Balsa was used to describe and synthesise the non-regular parts of the controller. Using synthesis this way allowed the design to be re-engineered as the customer's demands changed

with the minimum of designer effort and time. During the production of the custom layed-out parts of the controller the specification for the DMA controller changed in major ways at least twice. Each change required large sections of the Balsa description to change without incurring long delays to rework circuits by hand.

The Balsa portions of the controller were synthesised into standard cell layout in four steps:

1. Balsa-c was used to generate handshake circuit netlists.
2. Handshake circuit netlists were translated into gate level netlists for Compass Design Automation tools using balsa-netlist.
3. A small amount of hand optimisation was performed on those netlists. This was mostly to make up for deficiencies in early versions of balsa-netlist.
4. Netlists were compiled into standard cell layout using Compass's PathFinder standard cell place and route software.

The completed DMA controller consists of four main parts: the MARBLE bus interfaces, the SPI, a large block of automatically placed and routed standard cells and a number of small register banks.

6.9.1. Controller structure

Figure 6.3 shows the structure of the DMA controller with each of the large boxes corresponding to a process in the Balsa description. The majority of the control complexity in the DMA controller is located in the *register bank control* unit. This unit controls access to the DMA registers (most of which are physically located in the *register bank blocks*), performs the address/count increment operations on DMA registers and handles interrupt signalling. The register bank control handles requests for register access from the CPU (via MARBLE) and the *transfer engine* and also handles the *end-of-run* indication proffered by the *initiator interface* at the end of a transfer run. Mutually exclusive access to the register bank control is provided by a 3-way arbitration coded as part of the unit's Balsa description.

The *synchronous peripheral interface*, transfer engine and initiator interface actually process client requests and perform transfers across MARBLE using the DMA controller's MARBLE initiator interface.

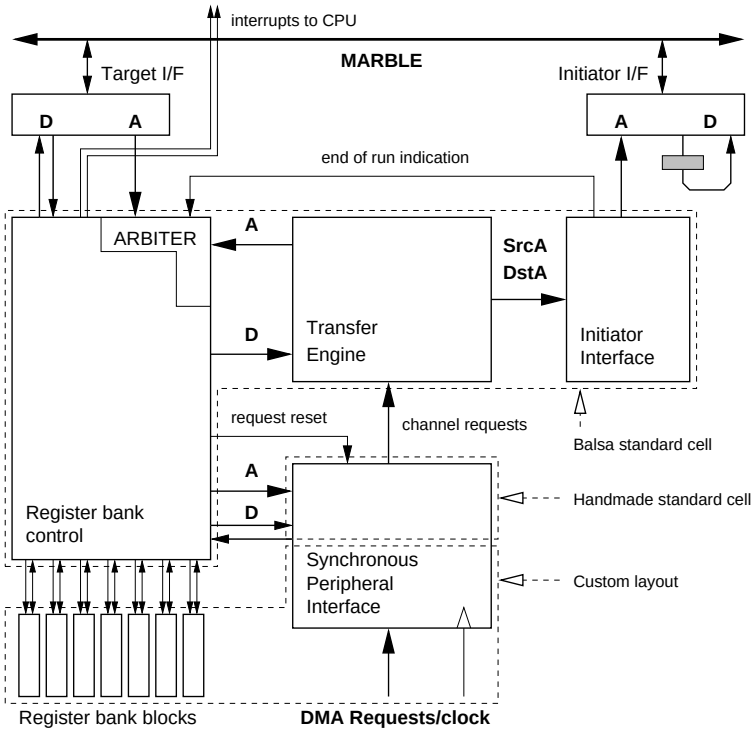


Figure 6.3. DMA controller structure

6.9.2. MARBLE bus interfaces

The MARBLE bus interfaces consist mostly of hand composed bus driver cells and wires. The control, address and data latches for the interfaces are built as compiled standard cell blocks. Each MARBLE interface consists of approximately 2000 transistors.

6.9.3. The regular SPI block

The SPI consists of a block of 22 stripes (one per request capable channel) of both custom and standard cells. Each stripe contains 16 custom made multiplexer cells making up that channels DMA request selection hardware and 5 custom made register cells to store the enable and DMA client request to channel mapping state. The remaining standard cells implement the address decoding for the enable, request number and request set/reset interface from the register bank control and the channel state machine itself.

The SPI occupies around 15% of the total area of the DMA controller.

6.9.4. Standard cell datapath and control

The standard cell block contains the decoding and control for the register banks, some of the MARBLE interfacing glue, the SPI top level control, the DMA request prioritisation hardware and all the other Balsa synthesised control and datapath shown in the register bank control, transfer engine and initiator interface blocks shown in figure 6.3. The register bank control and the transfer engine make up the lion's share of the standard cell block. This dominates the DMA controller, occupying nearly 50% of its total area.

6.9.5. Register blocks

The register bank blocks contain channel count and control data totalling around 2000 b (a few of the register control bits reside in the register bank control). These registers were constructed from custom made register and decoder cells in a similar manner to the register file in the processor core. Each register bank block corresponds to a particular channel register type and is indexed by channel number. Each block provides a single read/write interface to the standard cell block, allowing all the registers for a single channel to be read or written in one operation. DMA transfer operations require two accesses to a channel's registers: once to read addresses/count values for a channel and a second time to update the registers with incremented address and count values.

6.10. Balsa control description

Balsa was used to implement the register bank control, transfer engine and initiator interface. Each block was described by a single Balsa process communicating with its neighbours using handshake channels. For example, the main loop in the transfer engine is:

```
loop
  ChannelReq := {0, false} ||
  CountEqZero := false;

  -- Read DMA request vector
  select PRR then
    -- Priority encode, chan 0 has
    -- highest priority
    if PRR[0] then ChannelReq := {0, true}
    else if PRR[1] then ChannelReq := {1, true}
```

```
      …

      if ChannelReq.dotfr then
        PerformTransfer ();
        -- while we are asked to chain
        while RegReadData.usechain
              and RegReadData.genable then
          ChannelReq := { RegReadData.nextchan, true};
          PerformTransfer ()
        end
      end
    end
  end
```

The initialisations of the variables ‘ChannelReq’ and ‘CountEqZero’ can be seen in the above example. They are followed by a `select` statement within which values on the channel ‘PRR’ (which carries a vector of DMA requests from the SPI) are visible. The transfer is performed (or rather communicated to the initiator interface) by the sub-process `PerformTransfer`. The `while` loop is used to perform the tail transfers of a chain of channels. It is easy to see that chains are composed of ‘linked lists’ of channels from the way that they are processed.

The initiator interface is the simplest of the Balsa blocks and simply performs transfers on behalf of the transfer engine. The complete definition of the initiator interface is:

```
procedure DMA_II (
    output II_Addr  : MARBLEAddr;
    input DI        : IIData;
    -- Interrupt interface
    output EndOfRun : ChannelNo
) is local
  variable RegReadData : IIData
begin
  loop
    DI -> RegReadData;
    -- READ from Source Device
    II_Addr <- {RegReadData.src, Read,
        RegReadData.size};
    -- WRITE to Destination Device
    II_Addr <- {RegReadData.dst, Write,
        RegReadData.size};
    if RegReadData.endofrun then
      EndOfRun <- RegReadData.channelno
    end
  end
end
```

The channel interface that the process (`DMA_II`) presents to other processes can be clearly identified along with the locally defined variable RegReadData and the loop of four operations.

6.11. Controller performance

The DMA controller is required to perform transfers between peripherals and RAM on the synchronous side of the MARBLE/synchronous bus bridge. Most of the time, the DMA controller will be the only unit communicating with the synchronous peripherals. It must, therefore, be capable of issuing transfer requests fast enough to saturate the synchronous bus bridge.

Each synchronous bus transaction takes 3 bus cycles. The clock for this bridge is anticipated to run at between 13.824 MHz and 55.296 MHz. Each bus transaction will, as a consequence, take between 54.3 ns and 217 ns to complete. To perform a complete DMA transfer, two bus transactions are required and so the DMA controller must be capable of issuing a transfer every 108.6 ns.

Simulation (with Avant!'s TimeMill) on a capacitance-extracted view of the final controller has shown that transfers can be issued every 90 ns. The implemented controller therefore cycles at the required rate.

The DRACO chip has been fabricated and samples have been received (as of early September 2000) by AMULET group. Production tests show that all parts of the AMULET3i macrocell function correctly, including the DMA controller (with the small exception of a timing problem with the AMULET3 processor core's multiplier). Timing figures for the fabricated DMA controller are not yet available.

6.12. Chapter summary

This chapter has presented a significant Balsa example description, a DMA controller. The AMULET3i DMA controller is offered as an existence proof of the use of Balsa to implement

complex designs. A similar controller is described in the next chapter to evaluate the effectiveness of the changes made to the Balsa component set described in earlier chapters.

It should be pointed out that Jantaraprim [40] has described a number of ways to implement the control portions of asynchronous DMA controllers similar to the one described in this chapter. This is no coincidence, Balsa replaced his early hand-designed controllers in order to ensure that the product (AMULET3i) could be delivered on time. The design of the DMA controller described in this chapter remains wholly the work of the author of this thesis.

Chatchai Jantaraprim was, however, of great assistance in modelling early versions of the Balsa DMA controller in LARD.

Chapter 7. A Simplified DMA Controller

This chapter describes a simplified version of the DMA controller from the previous chapter. This controller is implemented using the 'old' version of Balsa and again with the changes described in chapter 5. As many of the advantages of the components described in that chapter affect the area and granularity (reducing the number of channels) of components, great increases in performance are not expected. Instead, reduced circuit complexity is to be expected by removing a portion of the small, 'unnecessary' components which tend to dominate handshake circuit implementations. A modest decrease in circuit area is also to be expected.

The full Balsa description for the simplified DMA controller can be found in appendix 2.

7.1. The simplified DMA controller

A simpler 4 channel DMA controller is presented as a more practical description to use for exploring the effect of the new handshake components on Balsa compilation. This DMA controller is written entirely in Balsa and so can be compiled for any of the technologies which the Balsa back-end supports.

The simplified controller provides:

- 4 full address range channels each with independent source, destination and count registers.
- 8 client DMA request inputs with matching acknowledgements.
- Peripheral to peripheral, memory to memory and peripheral to/from memory transfers. Each channel has both source and destination client requests so 'true' peripheral to peripheral transfers can be performed by waiting for requests from both parties.

Some features present in the AMULET3i DMA controller are omitted from this simpler controller. These include:

- Synchronous request handling with prioritisation. Requests are arbitrated in asynchronous fashion rather than being sampled.
- Byte and halfword (16 b) transfers. Only word wide transfers are supported, transfer count registers count the number of word transfers to perform.
- The three different channel types.
- The second interrupt request. The AMULET3i DMA controller supports signalling of either of the two interrupts which the ARM architecture supports (IRQ/FIQ). The simplified DMA controller only has a single interrupt line – IRQ.
- Chaining of channel requests. This feature was supported by the AMULET3i DMA controller specifically to support the dumb peripherals present in DRACO.
- The 'initiator interface' which allows request handling and transfer initiation to be overlapped by the transfer engine.

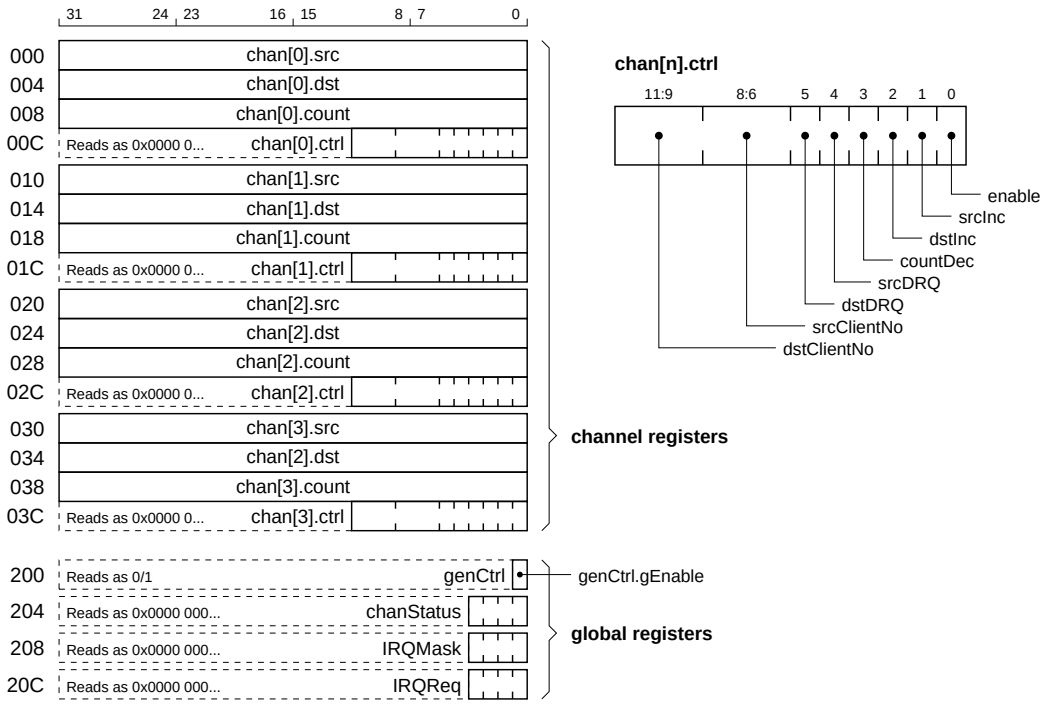


Figure 7.1. Simplified DMA controller programmer's model

Figure 7.1 shows the programmer's view of the controller's register memory map. The register bank is split into two parts: the channel registers and the global registers.

7.1.1. Global registers

The global registers contain control state pertaining to the state of currently active channels and of interrupts signalled by the termination of transfer runs. There are 4 global registers:

genCtrl: General control

In this controller, the general control register only contains one bit: the global enable – gEnable. The global enable is the only controller bit reset at power-up. All other controller state bits must be initialised before gEnable is set. Using a global enable bit in this way allows the initialisation part of the Balsa description to remain small and cheap.

chanStatus: Channel end-of-run status

The chanStatus register contains 4 bits, one per DMA channel. When set by the DMA controller, a bit in this register indicates that the corresponding channel has come to the end of its run of transfers.

IRQMask, IRQReq: Interrupt mask and status

The IRQMask register contains one bit per channel (like chanStatus) with set bits specifying that an interrupt should be raised at the end of a transfer run of that channel (when the corresponding chanStatus bit becomes set). IRQReq contains the current interrupt status for each channel.

The channel status, IRQ mask and IRQ mask bits are kept in global registers in order to reduce the number of DMA register reads which must be performed by the CPU after receiving an interrupt in order to determine which channel to service.

7.1.2. Channel registers

Each channel has 4 registers associated with it in the same way as the AMULET3i DMA controller. The two address registers (channel[*n*].src and channel[*n*].dst) specify the 32 b source and destination addresses for transfers. The count register (channel[*n*].count) is a 32 b count of remaining transfers to perform, transfer runs terminate when the count register is decremented to zero. The ctrl register (channel[*n*].ctrl) specifies the updates to be performed on the other three registers and the clients to which this channel is connected. Writing to the control register has the effect of clearing interrupts and end-of-run indication on that channel. The ctrl register contains 8 fields:

enable: Transfer enable

If the enable bit is set, this channel should be considered for transfers when a new DMA request arrives. Channel enables are not cleared on power-up. The genCtrl.gEnable bit can be used to prevent transfers from occurring whilst the channel enable bits are cleared during startup.

srcInc, dstInc, countDec: Increment/decrement control

These bits are used to enable source, destination and count register update after a transfer. Source and destination registers are incremented by 4 after transfers if srcInc and dstInc (respectively) are set. Note that the bottom 2 bits of these addresses are preserved. The count register is decremented by 1 after each transfer if countDec is set. Resetting either srcInc or dstInc results in the corresponding address remaining unchanged between transfers. This is useful for nominating peripheral (rather than memory) addresses. Resetting countDec results in 'free-running' transfers.

srcDRQ, dstDRQ: Initial DMA requests

Transfers can take place on a channel when a pair of DMA requests have been received, one for the source client and the other for the destination client (the *requests-pending* registers). The srcDRQ and dstDRQ bits specify the initial states for those two requests. Setting both of these bits indicates that the source and destination requests should be considered to have already arrived. Resetting one or both of the bits specifies that requests from the corresponding {src,dst}ClientNo numbered client should trigger a transfer (both client requests are required when both control bits are reset).

srcClientNo, dstClientNo: Client to channel mapping

These fields specify the client numbers from which this channel receives source and destination DMA requests. These fields are only of use when either srcDRQ or dstDRQ (or both) are reset.

7.1.3. DMA controller structure

The structure of the simplified DMA controller is shown in figure 7.2 The simplified DMA controller is composed of 5 units:

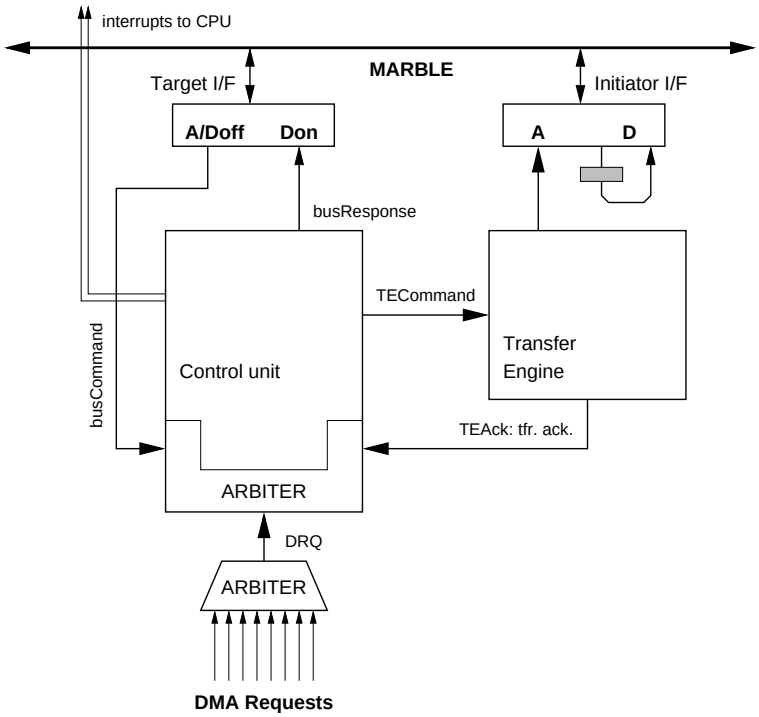


Figure 7.2. Simplified DMA controller structure

MARBLE target interface

The MARBLE target interface provides a connection to MARBLE through which the controller can be programmed. Accesses to the registers from this interface are arbitrated in with incoming DMA requests and *transfer acknowledgements* from the transfer engine. This arbitration and the decoupling of transfer engine from control unit allow this DMA controller to avoid the potential bus access deadlock situations described in §6.8.

The MARBLE interface used here carries an 8b address (8b word address, 10b byte address) like that of the AMULET3i DMA controller. This allows the same address mapping of channel registers and the possibility of having extended the number of channels to 32 without changing the global register addresses.

MARBLE initiator interface

The initiator interface is used by the DMA controller to perform its transfers. As with the full AMULET3i DMA controller, only the address and control bits to this interface are connected to the Balsa synthesised controller hardware. The data to and from the initiator interface is

handled by a latch (shown as the shaded box in figure 7.2). Only word-wide transfers are supported and so this latch is all that is needed to hold data values between the read and write bus transactions of a transfer.

Control unit

The control unit performs the operations of the register bank control and the SPI of the AMULET3i controller by receiving DMA requests as arbitrated control commands in the same manner as requests for register accesses from the CPU. Each DMA channel has a pair of register bits, the requests-pending bits, which recode the arrival of requests for that channel's source and destination clients. After marking-up an incoming request, the control unit examines the requests-pending registers of each channel in turn to find a channel on which to perform a transfer. If a transfer is to be performed, the register contents for that channel are forwarded to the transfer engine and the register contents are updated to reflect the incremented addresses and decremented count. DMA requests are acknowledged straight away when no transfer engine command is issued or just after the command is issued where a transfer command is issued to the transfer engine. The acknowledgement of DMA requests does not guarantee the timely completion of the related transfer, peripherals must observe bus accesses made to themselves for this purpose. The acknowledgement serves only to confirm the receipt of the DMA transfer request. A request must be removed after an acknowledgement is signalled so that other requests can be received through the request arbitration tree to mark-up potential transfers for other channels.

Transfer engine

The transfer engine receives its activating stimuli from the control unit. This differs from the AMULET3i controller where the transfer engine receives channel requests and itself initiates a request to the control unit for channel register values. This controller's transfer engine takes commands from the control unit when a DMA transfer is due to be performed and performs no DMA request mapping or filtering of its own. The only reason for having the transfer engine in this design is to prevent the potential bus deadlock situation if an access to the register bank is made across MARBLE while the DMA controller is trying to perform a transfer. In this situation control of the bus belongs to the initiator (usually the CPU) trying to access the DMA

controller. This initiator cannot proceed as the DMA controller is engaged in trying to gain the bus for itself. With a transfer engine, and the decoupling of DMA request/CPU access from transfer operations, the control unit is free to fulfil the initiator's register request while the transfer engine is waiting for the bus to become available.

After performing a transfer, the transfer engine will signal to the control unit to provide a new transfer command, it does this by a handshake on the transfer acknowledge channel (marked TEAck in the figure). This channel passes through the control unit's command arbitration hardware and serves to inform the control unit that the transfer engine is free and that the request-pending register can be polled to find the next suitable transfer candidate. The acknowledgement not only provides the self-looping activation required to perform memory to memory transfers but also allows the looping required to service requests for other types of transfer which are received during the period when the transfer engine was busy.

A flag register, `TEBusy`, held in the control unit is used to record the status of the transfer engine so that commands are not issued to it while a transfer is in progress. This flag is set each time a transfer command is issued to the transfer engine and cleared each time a transfer acknowledgement is received by the control unit. The request-pending registers are not re-examined (and a transfer command issued) if the `TEBusy` is set.

Arbiter tree

The DMA controller receives DMA requests on an array of 8 sync channels connected to the input of the ARBITER unit shown in figure 7.2. This arbiter unit is a tree of 2-way arbiter cells which combines these 8 inputs into a single 'DMA request number' which it provides to the control unit. DMA requests are acknowledged as soon as the control unit has recorded them. Only the successful transfer of data between peripherals should be used as an indication of the actual completion of a DMA operation. When a transfer is begun (i.e. passed from control unit to transfer engine), that transfer's channel registers and requests-pending registers are updated before another arbitrated access to the control unit is accepted. As a consequence, a new request on a channel can arrive (and be correctly observed) as soon as any transfer-related bus activity occurs for that transfer.

7.2. The Balsa description

The Balsa description of the DMA controller is composed of 3 parts: the arbiter tree, the control unit and the transfer engine. The two MARBLE interfaces sit outside the Balsa block and are controlled through the target{command,response} (`mta` and `mtd`) and initiator address/control (`mia`) ports attached to the Balsa description. The top level of the DMA controller is:

```
procedure DMAArb is ArbFunnel over NoOfClients

procedure dma (
  input mta : MARBLE8bACommand;
  output mtd : MARBLEResponse;
  output mia : MARBLECommandNoData;
  output irq : bit;
  array NoOfClients of sync drq
) is local
  channel DRQClientNo : ClientNo
  channel TECommand : array 2 of Word
  sync TEAck
begin
  DMAArb (drq, DRQClientNo) ||
  DMAControl (mta, mtd, DRQClientNo, TECommand, TEAck,
IRQ) ||
  DMATransferEngine (TECommand, TEAck, mia)
end
```

Interrupts are signalled by writing a 0 or 1 to the `irq` port. This interrupt value must then be caught by an external latch to generate a bare interrupt signal.

7.2.1. Arbiter tree

DMA requests from the client peripherals arrive on the sync channels `drq`, these channels connect to the request arbiter `DMAArb`. The procedure declaration for `DMAArb` is given in the top level as a parameterised version of the procedure `ArbFunnel`.

`ArbFunnel` is a paramaterisable tree composed of two elements: `ArbHead` and `ArbTree`. Pairs of incoming sync requests are arbitrated and combined into single bit decisions by `ArbHead` elements. These single bit channels are then arbitrated between by `ArbTree` elements. An `ArbTree` takes a number of decision bits from each of a number of inputs (on the `i` ports) and produces a rank of 2-input arbiters to reduce the problem to half as many

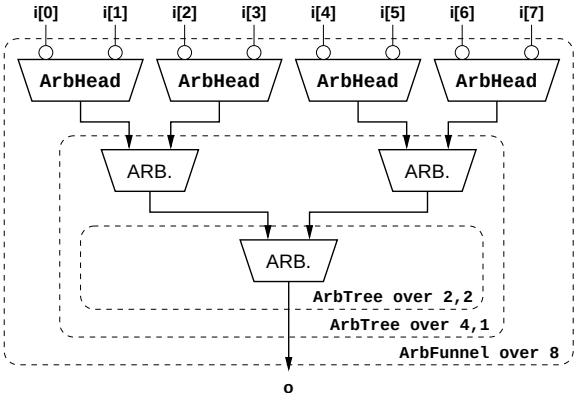


Figure 7.3. 8-input arbiter – ArbFunnel

inputs each with 1 extra decision bit. Recursive calls to `ArbTree` reduce the number of input channels to one (whose final decision value is returned on port `o`). Figure 7.3 shows this organisation for an 8-input `ArbFunnel`. The Balsa implementation of these three functions is largely uninteresting and the new components described in chapter 5 have no effect on their organisation.

7.2.2. Transfer engine

The transfer engine in this controller has relatively little to do. Its Balsa code is:

```
procedure DMATransferEngine (
  input command : array 2 of Word;
  sync ack;
  output busCommand : MARBLECommandNoData
) is local
  variable commandV : array 2 of Word
begin
  loop
    command -> commandV;
    busCommand <- {commandV[0],read,word};
    busCommand <- {commandV[1],write,word};
    sync ack
  end
end
```

The transfer engine is, like the arbiter unit, quite simple. It exists only as a buffer stage between the control unit and MARBLE initiator interface. This function is reflected in the sequencing in the Balsa description and the latches used to store the outgoing addresses.

7.2.3. Control unit

The bulk of the controller is contained in the control unit. Unlike the register bank control in the AMULET3i DMA controller, this control unit actually contains all the channel register latch bits and register access multiplexers/demultiplexers. The reduced number of channels and single channel type makes this arrangement practical. There are in total 445 b of programmer accessible state bits. The ports, local variables and local channels of the control unit's Balsa description are:

```
procedure DMAControl (
  input busCommand : MARBLE8bACommand;
  output busResponse : MARBLEResponse;

  input DRQ : ClientNo;

  output TECommand : array 2 of Word;
  sync TEAck;

  output IRQ : bit
) is local
  -- combined channel registers
  variable channelRegisters :
    array NoOfChannels of ChannelRegister
  variable channelR, channelW : ChannelRegister
    array over ChannelRegType of bit
  variable channelNo : ChannelNo
  variable clientNo : ClientNo

  variable TEBusy : bit

  variable gEnable : bit
  variable chanStatus : array NoOfChannels of bit
  variable IRQMask, IRQReq : array NoOfChannels of bit

  variable requestPending :
    array NoOfChannels of RequestPair

  channel commandSourceC : DMACommandSource
  channel busCommandC : MARBLE8bACommand
  channel DRQC : ClientNo
  variable commandSource : DMACommandSource
  . . .
```

The `ChannelRegister` is the combined source, destination, count and control registers for one channel. The variable `channelRegisters` is accessed by reading or writing these full 108b wide registers (32 + 32 + 32 + 12). The two registers, `channelR` and `channelW`,

are used as read and write buffers to the channel registers. This allows the partial writes required for CPU access to individual 32 b words to fragment only these two registers, not all of the channel registers. The variables `channelNo` and `clientNo` are used to hold channel and client numbers between operations. DMA request arrival and request mark-up can modify `clientNo` and channel register accesses and ready-to-transfer polling can modify `channelNo`.

The three channel declarations are used to communicate between a sub-procedure of `DMAControl`, `RequestHandler`, which arbitrates requests from the arbiter tree, MARBLE target interface and transfer engine acknowledge for service by the control unit. `RequestHandler`'s description is fairly uninteresting and so will not be discussed.

The body of the control unit, with the less interesting portions removed, is as follows:

```
begin
  Init ();
  -- RequestHandler is an ArbFunnel
  --   with accompanying data
  RequestHandler (busCommand, DRQ, TEAck, commandSourceC,
    busCommandC, DRQC) ||
  loop
    -- find source of service requests
    commandSourceC -> commandSource;
    case commandSource of
      DRQ then DRQC -> clientNo; MarkUpClientRequest ()
    | bus then
      select busCommandC then
        if (busCommandC.a as RegAddrType).globalNchannel
        then . . . -- global R/W from the CPU
        else -- channel regs
          channelNo :=
            (busCommandC.a as ChannelRegAddr).channelNo;
          ReadChannelRegisters ();
          case busCommandC.rNw of
            . . . -- most of CPU reg. access code omitted
            -- CPU ctrl register write
            | ctrl then channelW.ctrl :=
              (busCommandC.d as ControlRegister) ||
              requestsPending[channelNo] := {0,0} ||
              ClearChanStatus ()
            end;
            WriteChannelRegisters ()
          end
        end
      end
```

```
      else -- TEAck
        TEBusy := 0;
        if gEnable then AssessInterrupts () end
      end;
      if gEnable and not TEBusy then
        TryToIssueTransfer ()
      end
    end
  end
```

A number of procedure calls are made by the control unit body (e.g. `AssessInterrupts ()`). These procedure calls are to shared procedures whose definitions follow the local variables in `DMAControl`'s description. In Balsa, local procedures which are declared to be 'shared' are only instantiated in the containing procedure's handshake circuit in one place. (normal procedure calls place a new copy of that procedure's body for each call). Calls to shared procedures are combined using a Call component making their use cheaper than normal procedures for whom a new copy of the called procedure's body is placed at each call location.

DMA request handling – MarkUpClientRequest

Incoming DMA requests are marked up in the request pending registers as previously described. The procedure `MarkUpClientRequest` performs this operation by testing all the channels' srcClientNo and dstClientNo control bits with clientNo (the client ID of the incoming request) in parallel. `MarkUpClientRequests` description is:

```
  shared MarkUpClientRequest is
  begin
    for || i in 0..NoOfChannels-1 then
      if channelRegisters[i].ctrl.srcClientNo = clientNo
        then requestsPending[i].src := 1
      end ||
      if channelRegisters[i].ctrl.dstClientNo = clientNo
        then requestsPending[i].dst := 1
      end
    end
  end
```

The `for ||` loops in this description performs parallel structural instantiation of NoOfChannels copies of the body `if` statements.

Register access – ReadChannelRegisters, WriteChannelRegisters

The shared procedures used to access the channel registers are very short. They make the only variable-indexed accesses to the channel registers. The two procedures are:

```
shared ReadChannelRegisters is begin
  channelR := channelRegisters[channelNo]
end

shared WriteChannelRegisters is begin
  channelRegisters[channelNo] := channelW
end
```

Notice that no individual word write enables are present and so in order to modify a single word, a whole channel register must be read, modified, then written back. The `ReadChannelRegisters` followed by `channelW := channelR` in the description of the CPU's access to the channel registers uses this update method.

Channel status and interrupts – ClearChanStatus, AssessInterrupts

The interrupt output bit is asserted by `AssessInterrupts`. Interrupts are signalled when the IRQReq register is non-zero and are reassessed each time an action which could clear an interrupt occurs. `ClearChanStatus` is called during channel control register updates to clear interrupts and channel status (end-of-run) indications. Their descriptions are:

```
shared AssessInterrupts is begin
  IRQ <- (IRQReq as NoOfChannels bits) /= 0
end

shared ClearChanStatus is begin
  chanStatus[channelNo] := 0 ||
  IRQReq[channelNo] := 0;
  AssessInterrupts ()
end
```

Ready-to-transfer polling – TryToIssueTransfer, IssueTransfer

Whenever the DMA controller is stimulated by its command interfaces, it tries to perform a transfer. The request-pending, and ctrl.enable bits for each channel are examined in turn to determine if that channel is ready to transfer. Incrementing the channel number during this search is performed using a local channel to allow the incremented value to be accessed in parallel from two places. `TryToIssueTransfer`'s Balsa description is:

```
shared TryToIssueTransfer is local
```

```
    variable foundChannel : bit
    variable newChannelNo : ChannelNo
  begin
    foundChannel := 0 || channelNo := 0;

    while not foundChannel then
      -- source and destination requests arrived?
      if requestsPending[channelNo] = {1,1}
        and channelRegisters[channelNo].ctrl.enable then
        ReadChannelRegisters ();
        requestsPending[channelNo] :=
          channelR.ctrl.srcDRQ, channelR.ctrl.dstDRQ ||
        foundChannel := 1 ||
        IssueTransfer () ||
        UpdateRegistersAfterTransfer ()
      else
        local
          channel incChanNo : array ChannelNoLen + 1 of bit
        begin
          incChanNo <- (channelNo + 1 as
            array ChannelNoLen + 1 of bit) ||
          select incChanNo then
            foundChannel := incChanNo[ChannelNoLen] ||
            newChannelNo := (incChanNo[0..ChannelNoLen-1]
              as ChannelNo)
          end;
          channelNo := newChannelNo
        end
      end
    end
  end
```

Notice that if a transfer is taken, the `requestPending` bits for that channel are re-initialised from the ctrl.{srcDRQ,dstDRQ} control bits for that channel. The procedure `IssueTransfer` actually passes the transfer on to the transfer engine. Its definition is:

```
  shared IssueTransfer is begin
    TEBusy := 1 ||
    TECommand <- {channelR.src, channelR.dst}
  end
```

The interlock formed by checking `TEBusy` before attempting a transfer, and the setting/resetting of `TEBusY` by transfer initiation/completion ensures that no transfer is presented to the transfer engine (deadlocking the control unit) while it is occupied. The `TEAck` communication back to the control unit also provides stimuli for re-triggering the DMA controller to perform outstanding requests. This re-triggering, combined with the sequential polling of channels, allows outstanding requests (received while the transfer engine was busy) to be

correctly serviced. Notice that a static prioritisation on pre-arrived requests is enforced by sequential channel polling.

Register increment/decrement – UpdateRegistersAfterTransfer

After a transfer has been issued, the registers for that transfer's channel must be updated. This is performed by the procedure `UpdateRegistersAfterTransfer` whose description is:

```
shared UpdateRegistersAfterTransfer is begin
  channelW.ctrl := channelR.ctrl ||
  if channelR.ctrl.srcInc then
    channelW.src := (channelR.src + 1 as Word)
  end ||
  if channelR.ctrl.dstInc then
    channelW.dst := (channelR.dst + 1 as Word)
  end ||
  if channelR.ctrl.countDec then
    channelW.count := (channelR.count - 1 as Word)
  end;
  if channelW.count = 0 then
    chanStatus[channelNo] := 1 ||
    if IRQMask[channelNo] then
      IRQReq[channelNo] := 1
    end ||
    channelW.ctrl.enable := 0
  end;
  WriteChannelRegisters ()
end
```

This procedure uses two incrementers and a decrementer to modify the channel's source address, destination address and count respectively. If the channel's transfer count is decremented to zero, the `chanStatus` bit, interrupt status and channel enable are all updated to indicate an end-of-run.

7.3. An implementation

Presenting the DMA controller description to balsa-c produces a handshake circuit implementation. This handshake circuit consists of 485 handshake components, connected together with 688 handshake channels. The utility breeze-cost can be used to estimate the area occupied by each handshake circuit in a design[1]. The units returned by breeze-cost are linear μms of standard cells on an old 1μm, 2LM CMOS process. The total cost of this DMA controller

is 83030.25 units. The precision of this value is related to the exact cell sizes used by breeze-cost, not the accuracy of breeze-cost estimation. The cells in this old cell library have a 45μm pitch, adding 50% extra routing space (which is typically required in 2LM processes), the DMA controller's area would be approximately 5.5 mm^2 (or 0.67 mm^2 after linear scaling to 0.35μm).

The distribution of handshake component areas for this design is shown in figure 7.4. The graph shows the total areas of components in groups spanning 1000 breeze-cost units. This shows the distribution of small and large components within the design. Table 7.1 shows the frequencies of components within these groups. Notice that 469 (96% of the total) of the components lies in the area range [0, 999]. Of these small components, 291 (60% of the total number of components) are recorded as having 0 area (i.e. consist only of wires).

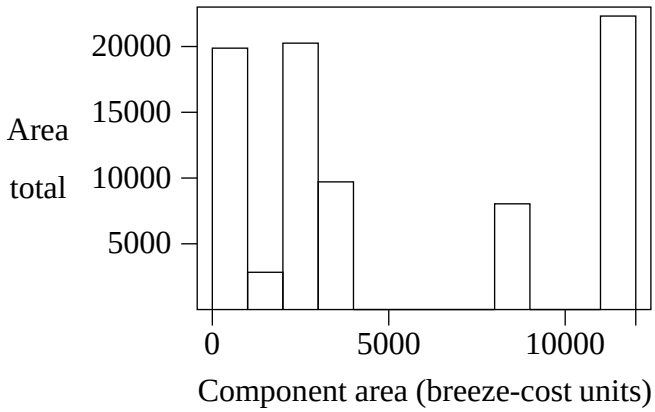


Figure 7.4. Simplified DMA controller component areas

Area range	Frequency
[0, 999]	469
[1000, 1999]	2
[2000, 2999]	8
[3000, 3999]	3
[8000, 8999]	1
[11000, 11999]	2

Table 7.1. Simplified DMA controller component frequencies

[1]The tool breeze-cost was not mentioned in chapter 4 as it was part of the original Balsa design flow described in [6].

It is obvious that the 0-area components (and to a lesser degree, the small components) add a considerable amount of complexity to this designs implementation (in terms of unnecessary netlist detail) without each making a significant contribution to the area of the circuit. The aim of the optimisations is to reduce this dominance and increase the size of components.

7.3.1. Optimisation opportunities

The register accesses and flag bit updates present in this DMA controller are amenable to optimisation using the new components from chapter 5. For example, the `channelW` register is written from a number of places:

```
1.  channelW.ctrl := channelR.ctrl ||
2.  channelW.src := (channelR.src + 1 as Word)
3.  channelW.dst := (channelR.dst + 1 as Word)
4.  channelW.count := (channelR.count - 1 as Word)
5.  channelW.ctrl.enable := 0
6.  channelW := channelR
7.  channelW.src := busCommandC.d
8.  channelW.dst := busCommandC.d
9.  channelW.count := busCommandC.d
10. channelW.ctrl := (busCommandC.d as ControlRegister) ||
```

These assignments come from different places and address different sized portions of `channelW`. The write port structure of `channelW` will, therefore consist of a number of multiplexers (CallMuxs) and broken-up variables like the example given in §5.2.1. Once reduced, the 5 CallMuxs, 8 Split/Combine components and 5 Variable components of which `channelW` was composed are reduced to a single PatchVariable of cost 10717 units. Such a large variable/multiplexer combination component could easily be realised as a small, regularly layed-out, register bank. Mask/Adapt/Combine components attached to FalseVariable read ports can, similarly, be absorbed into the FalseVariable to form PatchFalseVariables.

The flag variables (e.g. `TEBusy`) are updated by writing 0 or 1 into them. The Fetch, Constant, CallMux combination required for these assignments can be replaced by Encode components.

7.3.2. The optimised implementation

After replacing components in the controller implementation with a number of the new handshake components, the distribution of component sizes is as shown in figure 7.5.

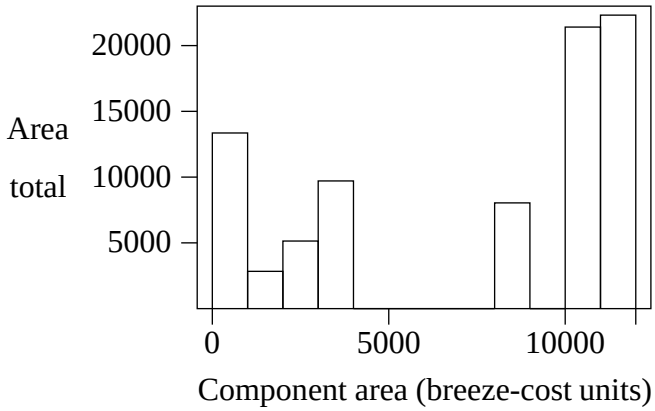


Figure 7.5. Simplified DMA controller optimised component areas

Area range	Frequency
[0, 999]	279
[1000, 1999]	2
[2000, 2999]	2
[3000, 3999]	3
[8000, 8999]	1
[10000, 10999]	2
[11000, 11999]	2

Table 7.2. Simplified DMA controller optimised component frequencies

The optimised circuit costs 82801 breeze-cost units. This is a very modest area decrease over the unoptimised implementation although the total component count has been reduced to 288.

Notice that the majority of the circuit area is now concentrated in the range [10000, 11999] cost units but that a majority of the components are still small. 160 of these components are still 0-sized. The majority of these 0-sized components (122) are transferrers. The new components are not intended to remove transferrers (although components like PatchVariable could have integrated read/write port transferrers). Transferrers are particularly innocuous,

however, as their implementations contain no ‘dangling’ wires, or wire forks. Flattening of transferrers within a netlist may then be a good approach to making handshake circuit netlists more comprehensible.

The remaining 0-sized components are mostly Constant components used to source decision bits in the `DMAArb` procedure. These components could be removed by optimising across the stages in `DMAArb` to centralise the decisions between incoming requests. This may not be possible as `DMAArb` contains arbiters between stages which are difficult to optimise across.

7.4. Chapter Summary

The new components described in chapter 5 have been applied to a substantially sized example design. They have been shown to be effective in reducing the number of very small components. Further new components may be necessary if it is desirable to reduce the number of transferrers which are the most frequent components in typical handshake circuit implementations.

Amalgamating small components from Balsa designs into larger, composite, components allows specific, component boundary crossing, optimisations to be exploited. This may include restricted forms of gate level optimisation or exploitation of timing-specific optimisations. Large, separately timing verified, components can be composed as pre-placed-and-routed macrocells (hard macrocells) to form whole circuits with only interconnect timing validation left to perform. This use of hierarchical place and route realises in silicon the boundaries between handshake components.

A large number of different, small components implies a large library of components from which they are drawn. For a particular design, an implementation composed of a small number of large, highly parameterised components will tend to form a library of highly specific components which may only be instantiated once. A similar circuit composed of many small components may make better use of reuse (i.e. more instances of each component type) but will almost certainly create a larger library of cells. Designs with large numbers of miscellaneous cells are harder to exchange between tools (problems with feed-through pin connections are particularly non-portable). The passage of a design from tool to tool typically

occurs a number of times in the Balsa CAD back-ends.

Chapter 8. Conclusions

The work described in this thesis has produced two substantial artifacts: the Balsa back-end tool set and the AMULET3i DMA controller. The Balsa back-end is capable of producing implementations of parameterised handshake components and presenting those implementations to a chosen CAD system. The AMULET3i DMA controller serves as substantial validation of this tool set.

The AMULET3i DMA controller is not just a paper design, the DRACO communications IC in which it is embedded has been fabricated. This IC was developed as a joint effort between the AMULET group and a commercial partner and is destined to be used in DECT mobile communications products. The DMA controller was built to a commercially-sourced specification which was drawn up without reference to Balsa. Realising a design based on this specification proves the flexibility of Balsa as a design language for general hardware design tasks.

The new handshake components described in chapter 5 are the result of working with the large netlists generated whilst building the AMULET3i DMA controller. The simpler, fully asynchronous, DMA controller design used to demonstrate the implementation-simplifying effects of these components reflects the control complexity of the full AMULET3i DMA controller. This simplified controller is, therefore, offered as substantial test design to other researchers (its specification is outlined in §7.1 and the complete Balsa description is given in appendix 2).

The Balsa system is available to download under the terms of the GNU General Public License version 2. A link can be found at:

http://www.cs.man.ac.uk/amulet/projects/balsa

This URL is guaranteed valid until at least March 2003.

8.1. Future work

Funding has been secured [23] to extend Balsa in two ways: improvements to datapath

synthesis of and additions to the behavioural simulation and design management systems. A number of other additions and improvement to the Balsa system are also envisaged.

8.1.1. Improved datapath synthesis

Automated tools exist to create custom layout of datapaths from a library of pitch-matched datapath cells. These datapath compilation tools produce compositions of cells which perform the desired function with control inputs (for multiplexers, adders, etc.) presented as ports.

The synthesis of Balsa datapath components could be adapted to make use of these tools to produce smaller, faster datapath implementations. Variants of the provided cell libraries could be produced to allow matched path control delays or delay-insensitive encodings of control (or carry chains) to be integrated into the datapath layout.

Datapath synthesis work on Balsa will include this form of custom datapath generation and also work on layout-aware placement for synthesis. This will include automated placement of standard cells (or functions in FPGA look-up tables) to produce regular layout with correctly placed (i.e. localised) interconnect.

The use of delay-insensitive codes to encode data for regular and 'cell compiler' placed standard cell layout will also be explored. A Balsa back-end technology using NCL is also planned.

8.1.2. Simulation and design management

The LARD-based simulation solution described in chapter 4 is a great improvement over the use of LARD described in [6]. Unfortunately, LARD was not created with the intention of using it an intermediate for modelling descriptions made in other source languages. The compiler and simulation environment do not provide a transparent enough view of the original description to make the experience of debugging Balsa using LARD seamless for the user.

Work will be carried out to rework the LARD simulation environment to better support Balsa. Integration of the nascent Balsa design manager (balsa-mgr) with this simulation

environment will allow the construction of an integrated Balsa design environment (an IDE). Support for test harness generation (and application), CAD interface integration and assessment and visualisation of designs (better than just breeze-cost) to allow users to make intelligent design choices based on simulation/synthesis results are all required.

8.1.3. Other work

Other future improvements to Balsa include:

- The full integration of the components described in chapter 5.
- Creation of new back-end technologies for newly developing IC design kits.
- Improvements in FPGA interconnect efficiency of implemented Balsa designs (interconnect for local synchronisation signals tends to be expensive in FPGAs).
- Exploring ways of using existing static-timing validation tools (for FPGA and ASIC designs) to validate Balsa designs without exhaustive simulation.

Control resynthesis work on Balsa handshake circuits is also currently being undertaken by T. Chelcea at Columbia University [16].

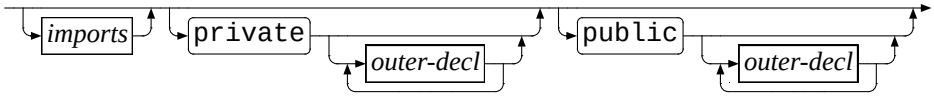
Appendix 1. Balsa Language Reference

1.1. Top level and declarations

balsa-description

Each Balsa file contains a single balsa-description. That description may import definitions from other files and make a number of definitions of its own. Definitions in the `private` part of the file are visible only to subsequent definitions in this file. Definitions in the `public` part are exported to the Breeze file by balsa-c. Names become bound at point of definition and are visible to subsequent definitions in the file. Balsa has separate namespaces for procedure/function/shared procedures names, variables and channel, and types.

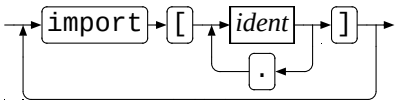
balsa-declaration



imports

Imports read in definitions from other files by parsing their (previously compiled) Breeze files. The dot separated paths represent file system paths from the root of the directories listed in the include search path defined by balsa-c. The first file to be found by matching against those path is the first to be included. Name conflicts between imported files are considered errors.

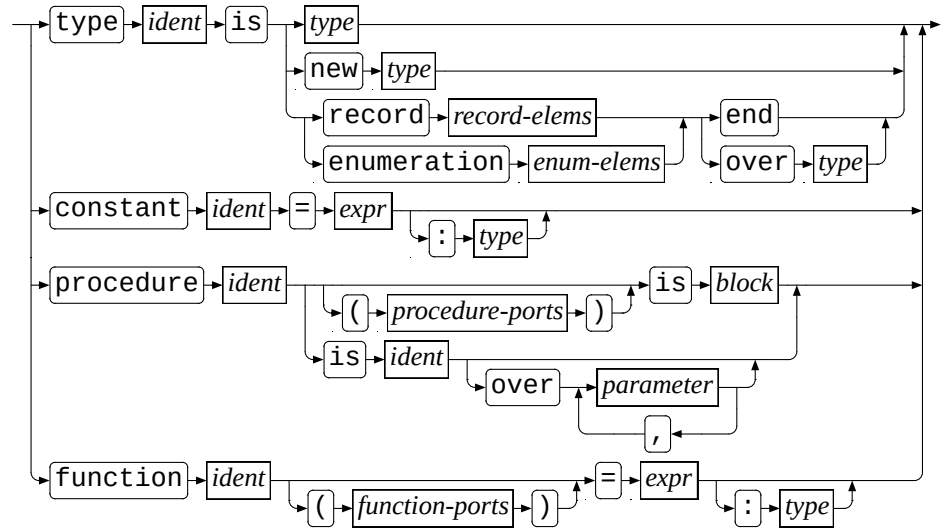
imports



outer-decl

At the top level of a balsa-description only the declaration/definition flavours given by outer-decl are allowed. The `over` sections in record and enumeration declarations allow a bounding type to be given which determines (and restricts) the size of those types.

outer-decl

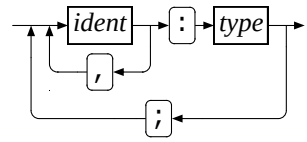


record-elems and enum-elems

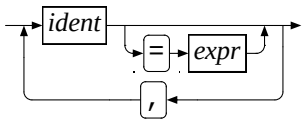
Record elements are packed in the given order into record typed values. The first element will occupy the least significant bits of a record typed value.

Enumeration elements get values starting from 0 and increasing by one from left to right. Elements with explicit values reset this counter to that value and so for a pair of elements: `elem1 = 5, elem2`; the value of element `elem2` is 6.

record-elems



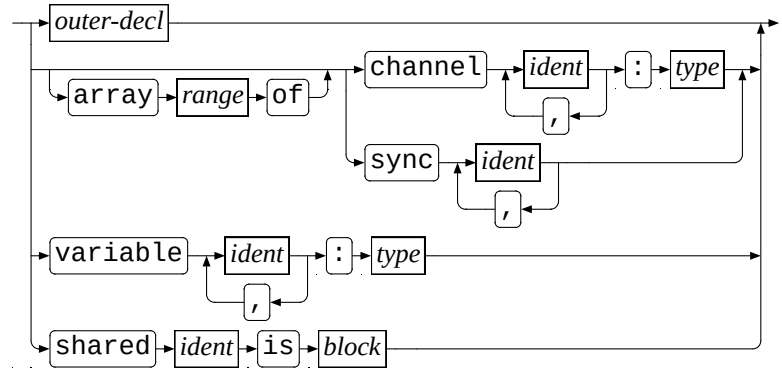
enum-elems



inner-decl

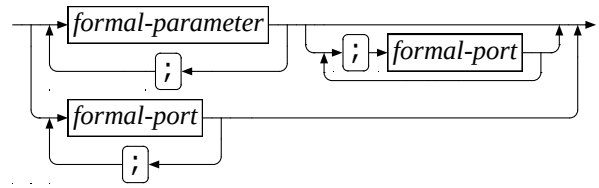
Channels, variables and shared procedures may only be declared inside procedures.

inner-decl

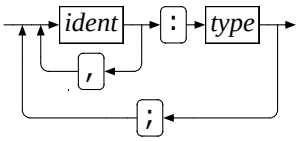


procedure-ports and function-ports

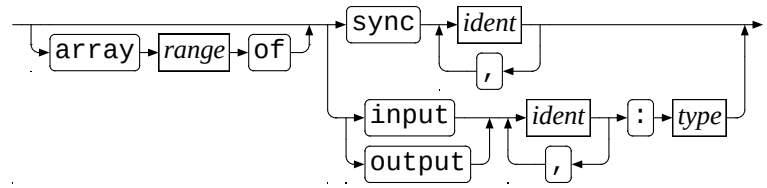
procedure-ports



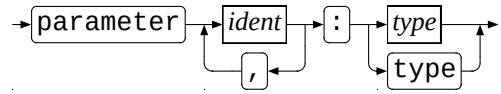
function-ports



formal-port



formal-parameter



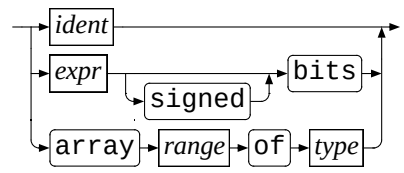
1.2. Expressions, types, ranges and lvalues

type, range and parameter

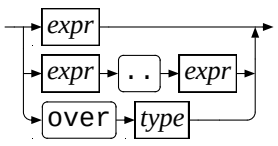
Balsa has two classes of anonymous types: numeric types (declared with the `bits` keyboard) and arrays of other types. Numeric types can be either signed or unsigned (without the `signed` keyboard). Signedness has an effect on expression operators and casting. Enumeration types and record types must be bound to names by a type declaration before use. Type equivalence in Balsa is usually by comparing points of declaration for named types, by size and signedness for numeric types and by size of range and base type for arrays. Only numeric types and arrays of other types may be used without first binding a name to those types.

Ranges are used for specifying the indices of elements of arrays, array slicing/element extraction, arrayed channels and in `case` command patterns and `for` loops. A single *expr* value is equivalent to $0 \ldots expr - 1$ in array and arrayed channels and to just the value of *expr* in slice/array extract operations, `case` patterns and `for` loops. Ranges over a type are equivalent to a range $0 \ldots 2^{sizeof\,type}$. For array elements and `for` loops ranges are always ascending even when specified with the greater value on the left hand side (the two values are swapped). Ranges may be applied over numeric types and enumerations. All slice operations must have constant indices.

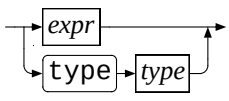
type



range



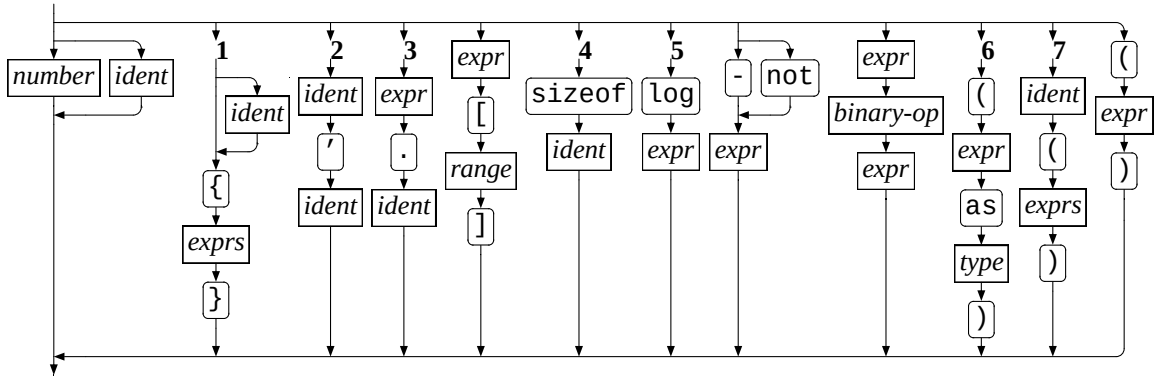
parameter



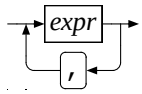
expr

1. Record/array construction. The optional ident specifies a type where this isn't obvious from context
2. Enumeration element choice: e.g. `Colours'Red`. Where the enumeration type is obvious from context, the type name can be omitted: e.g. `Red`
3. Record element extraction
4. Number of bits in type ident. Must be resolvable at compile time.
5. $\lceil \log_2 expr \rceil$. Must be resolvable at compile time.
6. Type cast operator, the parentheses are required. All value except signed numeric values are cast by truncation or zero padding. Signed numeric values cast into wider signed numeric types are sign extended. The only implicit type coercion in Balsa is the extension of numeric literals and constant expressions.
7. Function call expression.

expr



exprs



binary/unary-operators

Operator are shown in order of decreasing precedence.

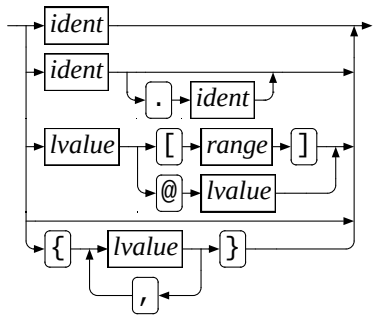
Symbol	Operation	Valid types	Notes
`[ ]`	array indexing	array	non-const index possible, can generate **lots** of hardware
`.`	record indexing	record	
`not`, `log`, `-` (unary)	unary ops.	numeric	log only works on constants, - makes returns a result one bit wider than the argument

`+`, `-`	add, subtract	numeric	results are between one and two bits longer than largest argument (two bits for signed +/- unsigned with a larger or equally sizes unsigned argument)
`*`, `/`, `%`	multiply, divide, remainder	numeric	only applicable to constants
`<`, `>`, `<=`, `>=`	inequalities	numeric, enumeration	
`=`, `/=`	equals, not equals	all	comparison is by sign extended value for signed numeric types
`and`	bitwise and	numeric	Balsa uses type `1 bits` for `if`/`while` guards so bitwise and logical operators are the same
`or`	bitwise or	numeric	

lvalue

The variable naming clause on the left hand side of assignments, the channel naming clauses on either side of channel I/O commands and the formal arguments to procedures are grouped together under the term lvalues. Lvalues name variables, channels, record elements of variables, slices/elements of variables of array types and the slices, construction and concatenation (with the `@` symbol) of arrayed channels used in procedure calls. The left hand array will occupy the low index portion of resulting array.

lvalue



1.3. Commands

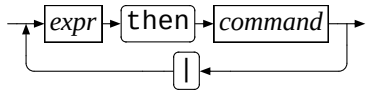
command

1. `continue` is the null command, `halt` causes deadlock.
2. Channel I/O and variable assignment. `->` is channel input, `<-` is channel output.
3. binds tighter than `;`, use *block* to override precedence.
4. Indefinate repetition.
5. `if` and `while` commands may have multiple guarded commands, only one of which is

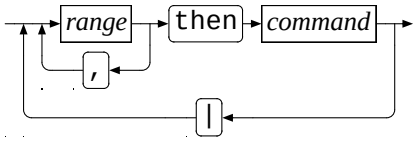
executed when its guard is met.

6. `for` performs structural iteration/composition similar to **`for generate`** in VHDL. The iteration range must be resolvable at compile time.
7. `select` and `arbitrate` perform passive input selection, `select` without arbitration, `arbitrate` with arbitration. A `select` may apply to any number of channel sets, `arbitrate` must be applied to exactly two sets. Selected channels are available as read-only variables within the guarded commands.
8. `print` can be used to give diagnostic messages during compilation. Any Balsa value can be passed to print but if the first element is one of the values of the built-in enumerated type `BalsaError`, this is used to specify what type of message is given. `BalsaError` is defined as `{fatal, error, warning, report}`.

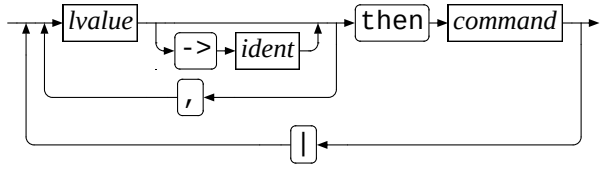
expr-guards



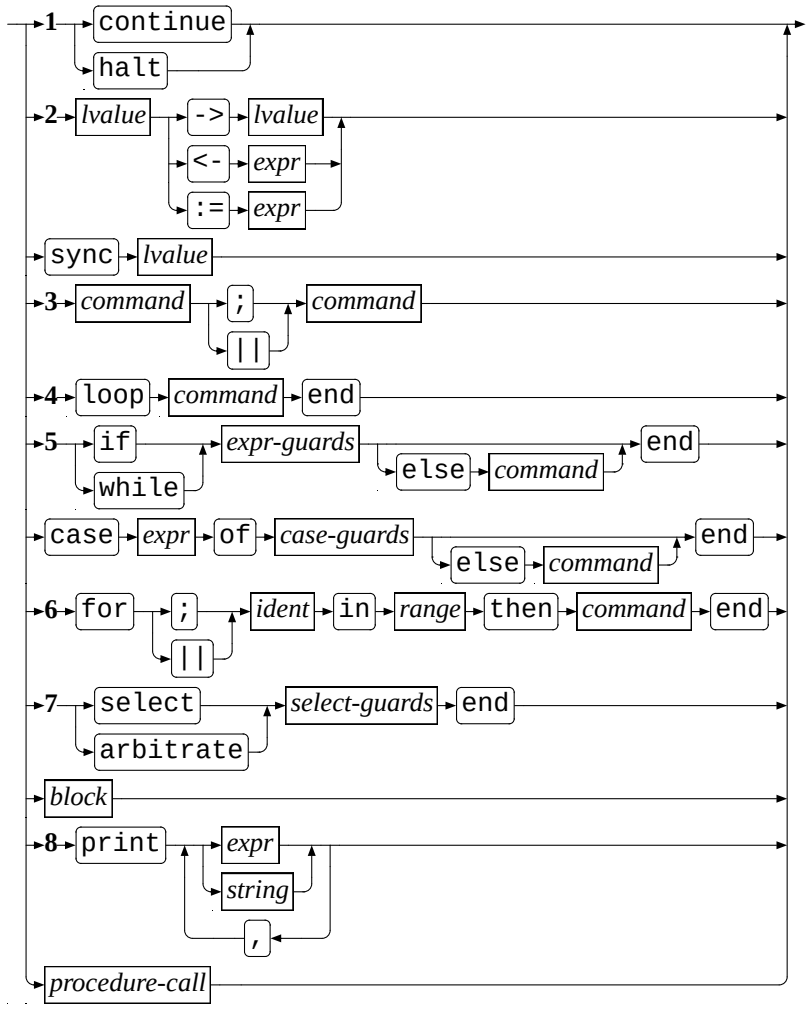
case-guards



select-guards



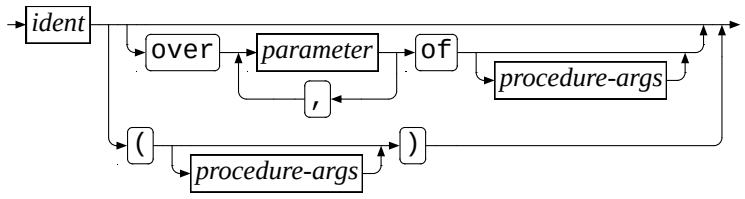
command



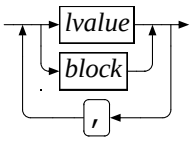
procedure-call

The `over` procedure call syntax is used to call parameterised procedures. Pre-parameterised versions of such procedures can be bound to names using a `procedure` declaration (see *outer-decl*). A block can be given in place of a sync channel name allowing the called procedure to activate that block with a handshake on that port.

procedure-call



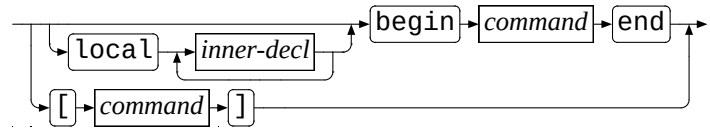
procedure-args



block

Blocks allow inclusion of local definitions around a command and the overriding of the precedence of command composition.

block

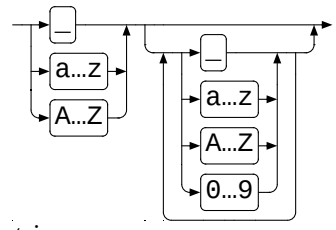


1.4. Terminals and comments

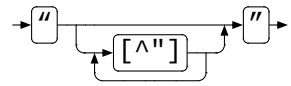
The terminals may have no extra whitespace inserted in them. Numbers can be given in decimal (starting with one of 1…9), hexadecimal (`0x` prefix), octal (`0` prefix) and binary (`0b` prefix). Identifiers follow rules similar to C, they are case sensitive and may not be identical to any of the predefined keywords.

There are two forms of comments: line comments begun by `--` and continuing to the end of the current line, and block comments enclosed the symbols `(--` and `--)`. Block comments are nestable.

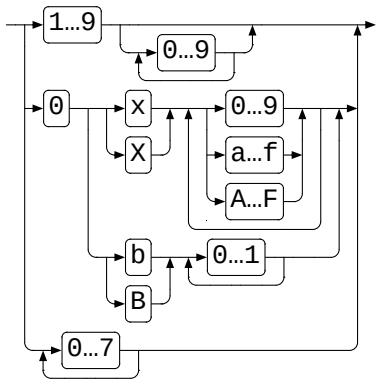
ident



string



number



Appendix 2. Simplified DMA Controller Source Code

2.1. ctrl.balsa

```
(-- 'ctrl.balsa': Register/DMAReq unit --)

import [types]
public

procedure RequestHandler (
  input busCommandIn : MARBLE8bACommand;
  input DRQIn : ClientNo;
  sync TEAck;
  output request : DMACommandSource;
  output busCommandOut : MARBLE8bACommand;
  output DRQOut : ClientNo
) is local
  channel drqAndBus : DMACommandSource
begin
  loop
    arbitrate DRQIn then
      drqAndBus <- DRQ || DRQOut <- DRQIn
    |  busCommandIn then
      drqAndBus <- bus || busCommandOut <- busCommandIn
    end
  end ||
  loop
    arbitrate drqAndBus then request <- drqAndBus
    |  TEAck then request <- TEAck
    end
  end
end

procedure DMAControl (
  input busCommand : MARBLE8bACommand;
  output busResponse : MARBLEResponse;

  input DRQ : ClientNo;

  output TECommand : array 2 of Word;
  sync TEAck;

  output IRQ : bit
) is local
  variable channelRegisters : array NoOfChannels of ChannelRegister
  variable channelR, channelW : ChannelRegister
  variable channelNo : ChannelNo
  variable clientNo : ClientNo

  variable TEBusy : bit

  variable gEnable : bit
  variable chanStatus : array NoOfChannels of bit
  variable IRQMask, IRQReq : array NoOfChannels of bit
```

```
  variable requestsPending : array NoOfChannels of RequestPair

  channel commandSourceC : DMACommandSource
  channel busCommandC : MARBLE8bACommand
  channel DRQC : ClientNo
  variable commandSource : DMACommandSource

  shared ReadChannelRegisters is begin channelR :=
channelRegisters[channelNo] end
  shared WriteChannelRegisters is begin channelRegisters[channelNo] :=
channelW end

  shared IssueTransfer is begin
    TEBusy := 1 || TECommand <- {channelR.src, channelR.dst}
  end

  shared UpdateRegistersAfterTransfer is begin
    channelW.ctrl := channelR.ctrl ||
    if channelR.ctrl.srcInc then
      channelW.src := (channelR.src + 1 as Word)
    end ||
    if channelR.ctrl.dstInc then
      channelW.dst := (channelR.dst + 1 as Word)
    end ||
    if channelR.ctrl.countDec then
      channelW.count := (channelR.count - 1 as Word)
    end;
    if channelW.count = 0 then
      chanStatus[channelNo] := 1 ||
      if IRQMask[channelNo] then IRQReq[channelNo] := 1 end ||
      channelW.ctrl.enable := 0
    end;
    WriteChannelRegisters ()
  end

  -- Take the client number in 'clientNo' and find channels which
  -- are triggered by that client request, set their reqeuest bits.
  shared MarkUpClientRequest is
  begin
    for || i in 0 .. NoOfChannels-1 then
      if channelRegisters[i].ctrl.srcClientNo = clientNo then
        requestsPending[i].src := 1
      end ||
      if channelRegisters[i].ctrl.dstClientNo = clientNo then
        requestsPending[i].dst := 1
      end
    end
  end

  -- Examine the transfer ready bits and
  -- issue a transfer to the TE if we can
  shared TryToIssueTransfer is local
    variable foundChannel : bit
    variable newChannelNo : ChannelNo
  begin
    foundChannel := 0 || channelNo := 0;

    while not foundChannel then
      if requestsPending[channelNo] = {1,1}
        and channelRegisters[channelNo].ctrl.enable then
        ReadChannelRegisters ();
```

```
          requestsPending[channelNo] :=
            {channelR.ctrl.srcDRQ, channelR.ctrl.dstDRQ} ||
          foundChannel := 1 ||
          IssueTransfer () ||
          UpdateRegistersAfterTransfer ()
        else
          local channel incChanNo : array ChannelNoLen + 1 of bit
          begin
            incChanNo <-
              (channelNo + 1 as array ChannelNoLen + 1 of bit) ||
            select incChanNo then
              foundChannel := incChanNo[ChannelNoLen] ||
              newChannelNo :=
                (incChanNo[0..ChannelNoLen-1] as ChannelNo)
            end;
            channelNo := newChannelNo
          end
        end
      end
    end

    -- Assess interrupt status after finishing a transfer
    shared AssessInterrupts is begin
      IRQ <- (IRQReq as NoOfChannels bits) /= 0
    end

    shared ClearChanStatus is begin
      chanStatus[channelNo] := 0 ||
      IRQReq[channelNo] := 0;
      AssessInterrupts ()
    end

    -- Initialise a few key variables
    shared Init is begin
      gEnable := 0 || TEBusy := 0 ||
      requestsPending := (0 as array NoOfChannels of RequestPair)
    end
  begin
    Init ();
    RequestHandler (busCommand, DRQ, TEAck, commandSourceC,
      busCommandC, DRQC) ||
    loop
      commandSourceC -> commandSource;
      case commandSource of
        DRQ then DRQC -> clientNo; MarkUpClientRequest ()
      |  bus then
        select busCommandC then
          if (busCommandC.a as RegAddrType).globalNchannel then
            case busCommandC.rNw of -- global regs
              read then
              case (busCommandC.a as GlobalRegAddr).regType of
                chanStatus then busResponse <- {(chanStatus as Word)}
              | IRQMask then busResponse <- {(IRQMask as Word)}
              | IRQReq then busResponse <- {(IRQReq as Word)}
              else (-- genCtrl --) busResponse <- {(gEnable as Word)}
              end
            |  write then
              case (busCommandC.a as GlobalRegAddr).regType of
                chanStatus then chanStatus :=
                  (busCommandC.d as array NoOfChannels of bit)
              | IRQMask then IRQMask :=
```

```
              (busCommandC.d as array NoOfChannels of bit)
            | IRQReq then IRQReq :=
              (busCommandC.d as array NoOfChannels of bit)
            else (-- genCtrl --) gEnable := (busCommandC.d as bit)
            end
          end
        else -- channel regs
          channelNo := (busCommandC.a as ChannelRegAddr).channelNo;
          ReadChannelRegisters ();
          case busCommandC.rNw of
            read then
            case (busCommandC.a as ChannelRegAddr).regType of
              src then busResponse <- {channelR.src}
            | dst then busResponse <- {channelR.dst}
            | count then busResponse <- {channelR.count}
            | ctrl then busResponse <- {(channelR.ctrl as Word)} ||
              ClearChanStatus ()
            end
          |  write then
            channelW := channelR; -- copy back other registers
            case (busCommandC.a as ChannelRegAddr).regType of
              src then channelW.src := busCommandC.d
            | dst then channelW.dst := busCommandC.d
            | count then channelW.count := busCommandC.d
            | ctrl then channelW.ctrl :=
                (busCommandC.d as ControlRegister) ||
              requestsPending[channelNo] := {0,0} ||
              ClearChanStatus ()
            end;
            WriteChannelRegisters ()
          end
        end
      end
    else (-- TEAck --)
      TEBusy := 0;
      if gEnable then AssessInterrupts () end
    end;
    if gEnable and not TEBusy then TryToIssueTransfer () end
  end
end
```

2.2. arb.balsa

```
(-- 'arb.balsa': Arbiter trees --)

import [balsa.types.basic]
public

-- ArbHead: 2 way arbcall with channel no. output
procedure ArbHead (
  sync i0, i1;
  output o : bit
) is begin loop
  arbitrate i0 then o <- 0
  | i1 then o <- 1
end end end

-- ArbTree: a tree arbcall which outputs a channel number
```

```
--  prepended onto the input channel's data. (invokes itself
--  recursively to make the tree)
procedure ArbTree (
  parameter inputCount : cardinal;
  parameter depth : cardinal; -- bits to carry from inputs
  array inputCount of input i : depth bits;
  output o : (log inputCount) + depth bits
) is local
  function AddTopBit (hd : bit; tl : depth bits) =
    ((tl as array depth of bit) @ {hd} as depth+1 bits)
  function AddTopBit2 (hd : bit; tl : depth+1 bits) =
    ((tl as array depth + 1 of bit) @ {hd} as depth+2 bits)
  function AddTop2Bits (hd0 : bit; hd1 : bit; tl : depth bits) =
    ((tl as array depth + 1 of bit) @ {hd0,hd1} as depth+2 bits)
begin
  case inputCount of
    0, 1 then print error,
      "can't build an ArbTree with fewer than 2 inputs"
  | 2 then loop
      arbitrate i[0] then o <- AddTopBit (0, i[0])
      | i[1] then o <- AddTopBit (1, i[1])
      end
    end
  |  3 then local channel lo : 1 + depth bits
    begin
      ArbTree over 2, depth of i[0..1], lo ||
      loop
        arbitrate lo then o <- AddTopBit2 (0, lo)
        | i[2] then o <- AddTop2Bits (1, 0, i[2])
        end
      end
    end
  else local
      constant halfCount = inputCount / 2
      constant halfBits = depth + log halfCount
      channel l, r : halfBits bits
    begin
      ArbTree over halfCount, depth of i[0..halfCount-1], l ||
      ArbTree over inputCount-halfCount, depth of
        i[halfCount..inputCount-1], r ||
      ArbTree over 2, halfBits of {l,r}, o
    end
  end
end

-- ArbFunnel: build a tree arbcall (balanced apart from the last
--  channel which is faster than the rest) which produces a channel
--  number from an array of sync inputs
procedure ArbFunnel (
  parameter inputCount : cardinal;
  array inputCount of sync i;
  output o : log inputCount bits
) is local
  constant halfCount = inputCount / 2
  constant oddInputCount = inputCount % 2
begin
  if inputCount < 2 then
    print error, can't build an ArbFunnel with fewer than 2 inputs
  | inputCount = 2 then
    ArbHead (i[0], i[1], o)
  | inputCount > 2 then
```

```
    local
      array halfCount+1 of channel li : bit
    begin
      for || j in 0 .. halfCount - 1 then
        ArbHead (i[j*2], i[j*2+1], li[j])
      end ||
      if oddInputCount then
        ArbTree over halfCount+1, 1 of li[0 .. halfCount], o ||
        loop select i[inputCount - 1] then li[halfCount] <- 0 end end
      else
        ArbTree over halfCount, 1 of li[0..halfCount-1], o
      end
    end
  end
end
```

2.3. dma.balsa

```
(-- 'dma.balsa': DMA controller --)

import [ctrl]
import [te]
import [arb]
public

procedure DMAArb is ArbFunnel over NoOfClients

procedure dma (
  input mta : MARBLE8bACommand;
  output mtd : MARBLEResponse;
  output mia : MARBLECommandNoData;
  output irq : bit;
  array NoOfClients of sync drq
) is local
  channel DRQClientNo : ClientNo
  channel TECommand : array 2 of Word
  sync TEAck
begin
  DMAControl (mta, mtd, DRQClientNo, TECommand, TEAck, irq) ||
  DMATransferEngine (TECommand, TEAck, mia) ||
  DMAArb (drq, DRQClientNo)
end
```

2.4. types.balsa

```
(-- 'types.balsa': Types and constants --)

import [marble]
public

-- Number of channels, clients
constant NoOfChannels = 4
constant NoOfClients = 8
constant ChannelNoLen = log NoOfChannels
constant ClientNoLen = log NoOfClients
```

```
type ChannelNo is ChannelNoLen bits
type ClientNo is ClientNoLen bits

--- DMA Commands

type DMACommandSource is enumeration
  DRQ, bus, TEAck
end

--- Channel registers

-- ChannelRegType: the 4 registers per DMA channel
type ChannelRegType is enumeration
  src, dst, count, ctrl
end

-- ControlRegister: control register LS bits
type ControlRegister is record
  enable : bit;
  srcInc : bit;
  dstInc : bit;
  countDec : bit;
  srcDRQ : bit;
  dstDRQ : bit;
  srcClientNo : ClientNo;
  dstClientNo : ClientNo
end

type ChannelRegister is record
  src, dst, count : Word;
  ctrl : ControlRegister
end

--- Global registers

-- GenCtrlRegister: General control register structure
type GenCtrlRegister is record
  gEnable : bit
over Word

-- GlobalRegType: the 6 global registers
type GlobalRegType is enumeration
  genCtrl, chanStatus, IRQMask, IRQReq
end

-- RegAddrType: union with RegAddr to tell global from channel
register
type RegAddrType is record
  padding : 7 bits;
  globalNchannel : bit
over 8 bits

-- Channel registers address format
type ChannelRegAddr is record
  regType : ChannelRegType;
  channelNo : ChannelNo;
  padding : 8 - (sizeof ChannelRegType + ChannelNoLen) bits
over 8 bits

-- Global registers address format
type GlobalRegAddr is record
```

```
  regType : GlobalRegType;
  padding : 7 - sizeof GlobalRegType bits;
  one : bit
over 8 bits

type RequestPair is record
  src : bit;
  dst : bit
end

--- Transfer engine
type TECommand is record
  srcAddr : Word;
  dstAddr : Word
end
```

2.5. marble.balsa

```
(-- 'marble.balsa': MARBLE types and parts --)

-- MARBLE Address/Data Interface
import [balsa.types.basic]
public

-- Basic types
type Word is 32 bits
type HalfWord is 16 bits

-- MARBLE types
type RNW is enumeration write, read over bit
type MSize is enumeration byte, halfWord, word over 2 bits

-- Command bundle, 32b and 8b addresses
type MARBLECommandNoData is record
  a : Word;
  rNw : RNW;
  size : MSize
end

type MARBLE8bACommand is record
  a : 8 bits;
  d : Word;
  rNw : RNW;
  size : MSize
end

-- Response bundle, no abort facility, not needed for data writes
type MARBLEResponse is record
  d : Word
end
```

2.6. te.balsa

```
(-- 'te.balsa': DMA Transfer engine --)
```

```
import [types]
public

procedure DMATransferEngine (
  input command : array 2 of Word;
  sync ack;
  output busCommand : MARBLECommandNoData
) is local
  variable commandV : array 2 of Word
begin
  loop
    command -> commandV;
    busCommand <- {commandV[0],read,word};
    busCommand <- {commandV[1],write,word};
    sync ack
  end
end
```

References

[1] Austria Mikro Systems International AG., Schloss Premstätten, A-8141 Unterpremstätten, AT. URL *http://www.amsint.com*.

[2] AMULET Low Power Cell Library. URL *http://www.cs.man.ac.uk/amulet/projects/exact.html#C*.

[3] D. Jagger. *Advanced RISC Machines Architecture Reference Manual*. ISBN 0-13-736299-4. Prentice Hall, 1996.

[4] A. Bailey, M. B. Josephs. Sequencer Circuits for VLSI Programming. In *2nd Working Conference Asynchronous Design Methodologies*. Dept. of Math. and Comp. Sci., Eindhoven Univ. of Technology, NL; Centre for Concurrent Systems and VLSI, CISM, South Bank Univ., London, May 1995.

[5] W. J. Bainbridge. *Asynchronous Systems on Chip Interconnect*. Ph.D. thesis, Department of Computer Science, The University of Manchester, 2000.

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

[7] J. Beister, G. Eckstein, R. Wollowski. CASCADE: A Tool Kernel Supporting a Comprehensive Design Method for Asynchronous Controllers. In *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, DK*. Lecture Notes in Computer Science. Springer. Dept. of Electrical Engineering, University of Kaiserslautern, DE; Infineon Technologies, Munich, DE.

[8] C. H. van Berkel. Beware the isochronic fork. Tech. Rep. UR 003/91 (1991), Philips Research Laboratories.

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

[10] K. van Berkel, F. Huberts, A. Peeters. Stretching Quasi Delay Insensitivity by Means of Extended Isochronic Forks. In *2nd Working Conference Asynchronous Design Methodologies*. Philips Research Laboratories; Twente Univ. of Technology, NL; Eindhoven Univ. of Technology, NL, May 1995.

[11] K. van Berkel, A. Bink. Single-Track Handshake Signaling with Application to Micropipelines and Handshake Circuits. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'96*. Philips Research Laboratories and Technische Universiteit Eindhoven, Eindhoven, NL, 1996.

[12] Edited by: W. Braeur, W. Reisig, G. Rozenberg. *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Lecture Notes in Computer Science, Advances in Petri Nets, Part II. Springer, September 1986.

[13] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, US, September 1991.

[14] S. M. Burns. Automated Compilation of Concurrent Programs into Self-timed Circuits. Master's thesis Caltech-CS-TR-88-2 (December 1987), Dept. of Computer Science, California Institute of Technology, CA, US.

[15] Cadence Design Systems Inc.. URL *http://www.cadence.com.*

[16] T. Chelcea. Report on Burst-Mode Oriented Back-End for the Balsa System. Unpublished report, August 2000.

[17] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* Ph.D. thesis, Massachusetts Institute of Technology, MA, US, June 1987.

[18] COMPASS from Avant! Corporation, 46871 Bayside Parkway, Fremont, CA, US. URL *http://www.avanticorp.com.*

[19] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *IEICE Transactions on Informations and Systems*, pages 315–325, 1997.

[20] D. A. Edwards. Balsa User Manual, Department of Computer Science, The University of Manchester, 1999.

[21] P. B. Endecott, S. B. Furber. Modelling and Simulation of Asynchronous Systems Using the LARD Hardware Description Language. In *12th European Simulation Multiconference.* Department of Computer Science, The University of Manchester.

[22] P. B. Endecott. LARD Documentation Home Page. URL *http://www.cs.man.ac.uk/amulet/projects/lard/index.html.*

[23] D. A. Edwards, S. B. Furber. A Datapath Compiler and Enhanced Simulation Environment for Balsa: an Asynchronous Silicon Synthesis System. Engineering and Physical Sciences Research Council Grant No. GR/N 19618/01.

[24] K. M. Fant, S. A. Brandt. NULL Convention Logic™. Tech. Rep. (1997), Theseus Logic Inc. 2177 Youngman Ave., St. Paul, MN, US.

[25] C. Farnsworth, D. A. Edwards, J. Liu, S. S. Sikand. A Hybrid Asynchronous System Design Environment. In *2nd Working Conference Asynchronous Design Methodologies*, May 1995. URL *http://www.cs.man.ac.uk/amulet/publications/papers/hybrid.html.*

[26] R. M. Fuhrer, N. K. Jha, S. N. Nowick, B. Lin, M. Theobald, L. Plana. MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines. Tech. Rep. CUCS-020-99 (July 1999), Department of Computer Science, Columbia University, NY, US.

[27] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'97*. Department of Computer Science, The University of Manchester; Cogency Technology Inc, UK, April 1997.

[28] S. B. Furber, A. Efthymiou, Montek Singh. A Power-Efficient Duplex Communications System. In *Asynchronous Interfaces: Tools, Techniques and Implementations, AINT'2000*. Department of Computer Science, The University of Manchester, UK and Department of Computer Science, Columbia University, NY, US, 2000.

[29] S. B. Furber, P. Day. Four-Phase Micropipeline Latch Control Circuits. ISSN 1063-8210. No publisher vol. 4, no. 2, 247–253 (June 1996).

[30] J. D. Garside, S. B. Furber, S.-H. Chung. AMULET3 Revealed. In *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'97*. Department of Computer Science, The University of Manchester, April 1999.

[31] H. van Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, G. Stegmann. An asynchronous low-power 80C51 microcontroller. In *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'98*. Technische Universiteit Eindhoven, Eindhoven, NL, March 1998.

[32] J. D. Garside, W. J. Bainbridge, A. Bardsley, D. M. Clark, D. A. Edwards, S. B. Furber, J. Liu, D. W. Lloyd, S. Mohammadi, J. S. Pepper, O. Petlin, S. Temple, J. V. Woods. AMULET3i – an Asynchronous System-on-Chip. In *Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC2000*. Department of Computer Science, The University of Manchester, April 2000.

[33] D. A. Gilbert. *Dependency and Exception Handling in an Asynchronous Microprocessor*. Ph.D. thesis, Department of Computer Science, The University of Manchester, 1997.

[34] T. Granlund, TMG Datakonsult. GNU MP - The GNU Multiple Precision Arithmetic Library. Tech. Rep. (1996), online 'info' documentation, Free Software Foundation.

[35] M. R. Greenstreet, T. Ono-Tesfaye. A Fast asP*, RGD Arbiter. In *Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'99*, April 1999.

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

[37] H. Hulgaard, P. H. Christensen. Automated Synthesis of Delay Insensitive Circuits. Master's thesis (1990), Dept. of Computer Science, Tech. Univ. of Denmark, Lyngby.

[38] INMOS Ltd.. *Occam 2 Programming Manual*. Series in Computer Science. Prentice-Hall International, 1989.

[39] Powerview from Innoveda Inc., 293 Boston Post Rd. West, Marlboro, MA, US. URL *http://www.innoveda.com*.

[40] C. Jantaraprim. An Asynchronous DMA Controller. Master's thesis (2000), Department of Computer Science, The University of Manchester.

[41] M. B. Josephs, J. T. Udding. Delay-insensitive circuits: an algebraic approach to their design. In *CONCUR '90*, June 1990.

[42] K. Kagotani, T. Nanya. Performance enhancement of two-phase quasi-delay-insensitive circuits. In *Systems and Computers in Japan*, pages 39–46, May 1996.

[43] Edited by: R. Kelsey, W. Clinger, J. Rees. Revised^5 Report on the Algorithmic Language Scheme. (R5RS). Tech. Rep. (February 1998).

[44] J. Kessels. Designing Asynchronous Standby Circuits for a Low-Power Pager. In *Third*

International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'97. Philips Research Laboratories, NL, April 1997.

[45] T. Kilks, S. Vercauteren, B. Lin. Control Resynthesis for Control-Dominated Asynchronous Designs. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'96*, 1996.

[46] P. Kudva. G. Gopalakrishnan, V. Akella. An Asynchronous High Level Synthesis System Targeted at Interacting Burst-Mode Controllers. In *International Conference on Hardware Description Languages (CHDL)*, 1995.

[47] J. Liu. *Arithmetic and Control Components for an Asynchronous System*. Ph.D. thesis, Department of Computer Science, The University of Manchester, 1998.

[48] T. Lord. Guile Scheme. Tech. Rep. (1996), online 'info' documentation, Free Software Foundation.

[49] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. UT Year of Programming Series. No publisher, 1–64 (1990).

[50] A. J. Martin. Synthesis of Asynchronous VLSI Circuits. Tech. Rep. Caltech-CS-TR-93-28 (1993), Dept. of Computer Science, California Institute of Technology, CA, US.

[51] A. J. Martin, A. Lines, R. Manohar, M. Nyström, U. Cummings, T. K. Less. The Design of an Asynchronous MIPS R3000 Microprocessor. In *17th Conference of Advanced Research in VLSI*, pages 164–181. IEEE Computer Society Press, 1997.

[52] P. A. Molina. *The Design of a Delay-Insensitive Bus Architecture using Handshake Circuits*. Ph.D. thesis, Imperial College of Science, Technology and Medicine, University of London, UK, 1997.

[53] D. Morris, R. N. Ibbett. *The MU5 Computer System*. ISBN 0 333 25750 2. Macmillan press, 1979.

[54] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. Ph.D. thesis, Stanford University, CA, US, 1993.

[55] P. Patra, S. Polonsky, D. S. Fussell. Delay Insensitive Logic for RSFQ Semiconductor Technology. In *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'97*. Intel Corp.; Physics Dept. SUNY Stony Brook, NY, US; Dept. of Computer Sciences, The University of Texas at Austin, TX, US, April 1997.

[56] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. Ph.D. thesis, Department of Computer Science, The University of Manchester, 1994.

[57] A. M. G. Peeters. Support for Interface Design in Tangram. In *Asynchronous Interfaces: Tools, Techniques and Implementations, AINT'2000*. Philips Research Laboratories, July 2000.

[58] A. M. G. Peeters. *Single-Rail Handshake Circuits*. Ph.D. thesis, Technische Universiteit Eindhoven, Eindhoven, NL, 1996.

[59] A. M. G. Peeters. Tangram Asynchronous Design Flow: Learning Curves. In *ACiD-WG*

Workshop, Newcastle upon Tyne. Philips Research Laboratories, January 1999.

[60] Philips Consumer Communications. URL *http://www.pcc.philips.com/journalist/010798c.html*.

[61] L. A. Plana. *Contributions to the Design of Asynchronous Macromodular Systems.* Ph.D. thesis, Department of Computer Science, Columbia University, NY, US, 1998.

[62] M. Renaudin, P. Vivet, F. Robin. ASPRO-216: a Standard-Cell Q.D.I. 16-Bit RISC Asynchronous Microprocessor. In *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'98*. E.N.S.T. de Bretagne; France Telecom – CNET Grenoble, FR, March 1998.

[63] P. A. Riocreux, M. J. G. Lewis, L. E. M. Brackenbury. Power reduction in self-timed circuits using early-open latch controllers. ISSN 0013-5194. No publisher vol. 36, 115–116 (January 2000).

[64] F. D. Schalij. Tangram Manual. Tech. Rep. UR 008/93 (1995), Philips Electronics N.V..

[65] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgal, A. Saldanha, H. Savoj, P. R. Stephen, R. K. Brayton, A. Sangolovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Tech. Rep. UCB/ERL. M92/41 (May 1992), Department of Electrical Engineering and Computer Science, University of California, Berkely, CA, US.

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

[67] I. E. Sutherland. Micropipelines. *Communications of the ACM* **32(6)**, Pages 720–738 (June 1989).

[68] A. Takamura, M. Kuwakao, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, T. Nanya. TITAC-2: A 32-bit Scalable-Delay Insensitive Microprocessor. In *ICCD '97*, pages 288–294, October 1997.

[69] Tanner Research Inc., 2650 East Foothill Boulevard, Pasadena, CA, US. URL *http://www.tanner.com*.

[70] Theseus Logic Inc. URL *http://www.theseus.com*.

[71] TimeMill from Synopsis Inc., 700 East Middlefield Rd., Mountain View, CA, US. URL *http://www.synopsis.com*.

[72] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Krieger Publishing, 1969.

[73] Open Verilog International, 15466 Los Gatos Boulevard, PMB 071, Suite 109, Los Gatos, CA, US. URL *http://www.ovi.org*.

[74] Xilinx Inc., 2100 Logic Drive, San Jose, CA, US. URL *http://www.xilinx.com*.

[75] K. Y. Yun, D. L. Dill, S. M. Nowick. Synthesis of 3D Asynchronous State Machines. In *IEEE International Conference on Computer Design (ICCD): VLSI in Computers and Processors, Cambridge, MA, US*, pages 346–350, October 1992.