Balsa: An Asynchronous Circuit Synthesis System

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

1998

Andrew Bardsley

Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

The Balsa system for asynchronous circuit synthesis is described. Balsa is a CSP-like language based on the Tangram VLSI programming language developed by Philips Electronics NV. and Eindhoven University of Technology. The Tangram language can be transparently compiled into an intermediate representation (between the language and gate level implementations) called handshake circuits.

This thesis describes those additional features that Balsa exhibits above Tangram, the Breeze intermediate file format for handshake circuits, the balsa-c compiler (from Balsa to Breeze), a simulation environment for Balsa and also suggests a framework for a 'back-end' compiler to transform Breeze into standard cell logic.

In particular the support for separate compilation and the use of a flexible communication enclosed input choice mechanism are claimed as useful additions to the expressiveness of Tangram. New handshake components (which are the constituent parts of handshake circuits) are proposed which are used to implement this choice mechanism as well as more generalised forms of the existing Tangram system components.

Declaration

A description of the Balsa language and a previous implementation of the balsa-c compiler for that language was previously submitted in support of an application for the degree of Bachelor of Science in the Department of Computer Science of The University of Manchester. No claim is made that the language and compiler description defined in the report for that previous work (as it appears in the departmental library of the Department of Computer Science) should be considered part of the work in support of the author's application for the degree of Master of Philosophy.

The base language described in §3.1 excluding those features described in §3.1.5 is of the same language as described by the report presented as part of the previous application for the degree of Bachelor of Science.

The extensions outlined in §3.1.5, the description of the balsa-c compiler in this thesis (which is of a different implementation) and all other original descriptions made in this thesis **are** claimed as part of the author's work towards the application for the degree of Master of Philosophy.

No portion of the work referred to in this thesis, other than that described above, 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

Dedication

This thesis is dedicated to the memory of Michael Jealous

Acknowledgements

Thanks go to:

My supervisor – Dr. Doug A. Edwards

still directed, still laissez faire

My parents

for being ever patient sounding boards

The members of the AMULET group

and all the other people who have offered inspiration and derision this year

My tolerant proof readers – Dr. David A. Gilbert, W. John Bainbridge

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 v3.11 on an Intel Pentium II based PC running the Linux operating system v2.1.62. Figures were prepared using xfig v3.2 and imported into Lout as Encapsulated Postscript.

Chapter 1. Introduction

1.1. Thesis overview

This thesis describes the Balsa asynchronous hardware synthesis system. The Balsa system is heavily based on the Handshake Circuit paradigm developed by Philips Electronics NV. and Eindhoven University of Technology and serves to extend work carried out by the AMULET Group at Manchester University under the EXACT project[1] into the benefits of the use of Micropipeline style single-rail implementations of handshake components.

The Balsa Compiler (balsa-c) forms the front-end of a design flow which previously included the Philips Tangram compiler and the EXACT handshake component library. balsa-c is based on an extended form of the Tangram compilation mechanism (as described by van Berkel).

As a language Balsa attempts to make VLSI programming more palatable to the engineer by resembling familiar programming languages and by including greater expressiveness with the enclosing select construct and more general case construct. Reuse of library modules is made possible by the use of the intermediate language / file format – Breeze.

Gate level implementations of the new components introduced are presented along with an outline for a 'back-end' handshake circuit to gate level netlist tool – breeze2gates. Simulation of Balsa designs is performed using the LARD tool [35] (also developed in the AMULET group) at a handshake circuit level.

1.2. Asynchronous design

The global clock forms the central tenet of today's dominant design paradigm, that of synchronous circuits. Synchronous circuits use one (or a small collection of closely related) signals to synchronise changes of state across a machine. Viewing changes of state as

[1] EXACT was the EXploitation of Asynchronous Circuit Technologies project (ESPRIT 6143) of which The University of Manchester was a partner.

occurring simultaneously allows circuits to be modelled at the so-called register transfer level (RTL) where the new output values of combinatorial circuit elements are all calculated on the basis of the current state and the values on the circuit inputs, the new state being a function of those values.

The global clock is therefore a powerfully attractive abstraction for simplifying the implementation of sequencing and synchronisation and for hiding from the designer those undesirable features of the underlying intrinsically analogue electronic technology, these include hazards and metastability.

Recently there has been renewed interest in designing without clocks. The very large, very fast modern designs and the demand for low-power portable computationally powerful devices (mobile phones, PDAs, digital cameras, portable CD players etc.) makes demands of clocked, synchronous, designs which are difficult to meet. The distribution and control of the energy dissipated by a free running clock are major development headaches for the modern synchronous designers.

Synchronous design does however rest on the assumption that the global clock signal can be distributed to all parts of the circuit without phase differences. In practice this is difficult to achieve and all synchronous implementations must be tolerant of the maximum clock skew. Fractal clock distribution networks can be used to match delays from the clock generating circuit to the registers of a design, but this design alleviates neither the large power consumption of such a network nor the requirement to tolerance for process variations.

If the clock can be eliminated and the problems of synchronisation, relative timing and control of hazards can be overcome, asynchronous designs may be of some use. It remains to be proven that claims for low-power operation are justified for asynchronous techniques and so for the remainder of this thesis only the claim that asynchronous signalling techniques offer greater *composability* than synchronous systems is considered. The use of composability of small units to create complete designs is demonstrated by Sutherland in his Micropipeline design style [22]. The composability offered by asynchronously communicating circuit primitives is also the basis for the handshake circuits design style on which the work contained in this thesis is based.

The composability of asynchronous circuits by means of asynchronous communications is in contrast with the difficulties associated with connecting synchronous systems driven from different clocks together. Such synchronous compositions may be control heavy (introducing wait states, requiring some form of synchronous acknowledgement), slow (each transfer cycle taking the greater of two multiples of the clock periods of both circuits) and intrinsically susceptible to metastable failure.

1.3. The contribution made by this work

The current Balsa language and the balsa-c compiler are based on a final year undergraduate project carried out by the author [1]. The re-implementation of the compiler, the introduction of new features not found in Tangram and the compiler's integration into a simulation and synthesis framework are claimed as the contributions made by the work here described.

By these additions it is hoped that the expressiveness of a Tangram-like language is increased and that new design areas are opened up (in particular the ability to choose between control and data driven descriptive styles). Handshake components are used as a target form not only to benefit from their qualities as abstractive tools (abstracting implementations away from particular signalling and data validity protocols and data encoding schemes) but also to allow separate development of the compiler front and back ends.

1.4. The structure of this thesis

Subsequent chapters are:

2. Handshake Circuits

   A short introduction to the structure of handshake components and circuits. The language Tangram and the compilation mechanism which translates Tangram into handshake circuits are also briefly covered.

3. The Balsa Language

A description of the Tangram derived language, Balsa. Features of the language are explained and compared with similar features in existing programming and hardware description languages. A new input selection scheme is discussed along with a mechanism for compiling descriptions using this scheme into handshake circuits.

4. Compiling Balsa to Handshake Circuits

The use of the Tangram compilation function in a practical compiler and the extensions made to the compilation function to include the added expressiveness of Balsa descriptions are detailed.

5. Handshake Component Implementations and Simulation

Aspects of the choices made to arrive at a usable set of gate-level handshake component implementations. A simulation environment and the effects of applying optimisations at different levels of abstraction are also described.

6. The test designs

Two example designs, the SSEM and STUMP, written in Balsa are given. The SSEM is a simple sequential processor written in the Tangram style, STUMP is described in a more push-orientated pipeline element style. Important design features of the test circuits are highlighted as well as the use of hand optimisation to improve upon a simple design.

7. Conclusions and Further Work

Important features of Balsa are summarised. The difficulty of making conclusions about ongoing work is reflected upon. Comparisons are made between Balsa / Tangram and other approaches to the problem of asynchronous circuit synthesis, specifically the works of Burns [36] and Brunvand [15]. Future work required to complete the Balsa system is outlined.

Chapter 2. Handshake Circuits

The intention of this chapter is to familiarise the reader with the notation and terminology used in the remainder of this thesis.

This will consist of a basic introduction to the structure of the hardware description language *Tangram* and to the *Handshake Circuit* intermediate representation which the Tangram compiler targets.

A notation is defined to allow the description of the behaviour of the Handshake Component primitives which are the basis of Handshake Circuits. A mapping is also given between the handshake component set used by the Balsa compiler balsa-c (the compiler is described in §4.2) and the compiler that is part of the Tangram system. This is provided as a guide for readers familiar with Tangram and the Tangram names for components.

The Breeze language is also introduced. Breeze is the target format for the Balsa compiler and is simply a netlist format for handshake circuits. Breeze component port structures are compared with those of the Tangram netlist format HCL to illustrate the use of parameterisable *arrayed ports* to define the port structures of Balsa handshake components. A more detailed description of Breeze and the used of arrayed ports / channels can be found in §5.2 and §3.1.5.2 respectively.

This chapter should not be considered to be a detailed or complete account to the handshake circuit methodology. The author directs the interested reader to the Ph.D. thesis of Kees van Berkel [28] for a fuller description of the derivation and formal basis of handshake circuits.

2.1. Handshake circuits

Handshake circuits, described by van Berkel [28] are used as an intermediate representation for asynchronous circuits compiled from the hardware description language Tangram. They allow the construction of circuits without knowledge of the details of the underlying handshake protocols or the implementation technology. Such circuits are compositions of *handshake*

components connected by handshake communicating channels. Each Tangram language construct has a direct compilation mechanism into a handshake circuit fragment by means of a front-end compiler. This direct approach allows the designer to choose the degree of complexity of the implementation and its position in the power / area / speed … trade off *n*-dimensional solution space. Direct compilation also allows the front-end compiler to have a lower complexity and greater speed than synthesisers based on state-space exploration techniques such as Burst Mode machines (for example, the tool 3D [29]) or Petri-nets (the tools ASSASSIN [10], FORCAGE [33] and Petrify [23]).

2.2. The handshake components

The handshake component set used by the Tangram and Balsa systems consists of approximately thirty, parameterisable components. Parameter types include:

- the datapath width supported by the component on its various ports.
- the number of ports of the same type with similar behaviour (these are known as *arrayed ports* and are used extensively in the Balsa component set).
- specification strings which affect component behaviour (such as the *value* parameter of a Constant handshake which is intended to provide a constant value on a single output port for use in compiled expressions).

Each handshake component also has a number of ports. Each port has three attributed features:

Type

This is associated with data bearing ports and determines the number of data wires bundled.

Direction

This is one of input, output or nonput. The direction indicates the **data** direction on that port.

Sense

This is either active or passive. A port may either be the initiator of a communication (active, the source of the request signal) or its target (passive, receiving the request signal). A port's sense is separate from its data direction and determines the direction of the request and acknowledge lines connected to that port.

The Balsa system uses the language Breeze as a target format for the compilation of Balsa. Breeze allows structural compositions of handshake components (ie. handshake circuits) to be described using a syntax not unlike Balsa itself.

For each component in the Balsa component set there is an associated Breeze declaration which describes the port and parameter structure of that component. For example:

```
part $BrzVariable (
  parameter width         : cardinal;
  parameter readPortCount : cardinal;
  parameter name          : string |
  passive input write     : width bits;
  array readPortCount of passive output read
    : width bits
)
```

describes the Variable component which has three parameters (`width`, `readPortCount` and `name`) and two groups of ports (`read` and `write`). The type `cardinal` is used to indicate a numeric parameter in the range $[0, \infty]$ and is usually used to indicate the widths and number of ports in components with datapath or numbered port structures. The `string` type used in the name parameter is supported to allow annotation of components by the compiler and to allow specification strings.

All the ports of a Breeze component must have a type of the form: *n* bits (the use of this form of type, the numeric type, is detailed in §3.1.2) in order to simplify the typing of Breeze compositions. Breeze is explained more fully in §5.2.

The read ports of the Variable component are grouped together in an *arrayed port* where each of the ports `read[0]` to `read[readPortCount - 1]` has the same type, direction and

sense. The use of arrayed ports and arrayed channels in Balsa is described in §3.1.5.2, they are largely the result of the desire to be able to parameterise back-end components in a way which is consistent with the Balsa procedure port structure and type system but without introducing port structures which have a variable length 'tail' of ports (like the varargs style unconstrained argument lists of the C language). An instance of the Variable component with two read ports connected to wire numbers 2 and 3, a write port connected to wire 4 and a width of 12 bits would look something like:

```
$BrzVariable (12,2,"Some name" | #4,{#2,#3})
```

The two read ports are gathered together in a braced list so giving a port structure which is a two element list of write and read ports. The channels appearing in a Breeze file are all numbered instead of named in order to simplify name clashes and aid numeric indexing of arrays of channels.

2.2.1. Composing handshake circuits

In the composition of a handshake circuit, a number of handshake components are connected together by channels. Each channel connects exactly one passive and one active port of a pair of handshake components. The ports of the composed handshake circuit are formed by channels connecting ports within the circuit to the environment. All other ports must be internally connected to other components' ports.

As any particular active or passive port could be either an output or an input, two forms of communication are possible: push and pull.

Push communications

Push communications take place on channels which connect passive inputs to active outputs. This is the Micropipeline [22] style communication where the request flows in the same direction as the data. Data validity is signalled by request and released by acknowledge.

Pull communications

Pull communications take place on channels which connect active inputs to passive outputs. In this form of communication, data flows in the same direction as the acknowledgement. A communication is initiated by the recipient of the data and so data validity is signalled by the acknowledge line and released by the request (either the $r \downarrow$ of this communication or the $r \uparrow$ of the next).

The communication type is a property of the channel. The compilation mechanism for Tangram (which is explained in §2.3) results in circuit fragments with active input and active output ports (as a result of the active nature of the transferrer's data ports). Circuits constructed from compositions of compiled circuit fragments must have their interconnecting ports connected by Passivator components.

The Passivator introduces synchronisation of requests from input and output ports and mediates the overlapping of the two handshakes (one push, one pull) so that the data valid phases of the two data-validity protocols involved coincide. Channel communications in Tangram also include the concept of the *broadcast channel*, here one output channel can synchronously communicate a value to several inputs by means of a tree of components which serve a conceptually similar role to the Passivator.

The combinations of active / passive, input / output / nonput ports and their relationships to push and pull channels is shown in [tab. 2.1]. In accordance with established convention handshake components are drawn as large circles with smaller circles as ports, the sense and direction of the ports being indicated by the open or filled port circle and the direction of the arrow on the channel's arc respectively. An example of this representation is given in [fig. 2.1]. Nonput ports have no associated data and so no data direction or sense of push / pull on their connected channels. Sync channels are indicated by arcs without arrows.

The channel connections to ports in [tab. 2.1] make the the data, request and acknowledgement signals of the ports explicit for the purposes of emphasising their directions. This does not, however, indicate that a bundled data implementation is necessarily the only means of implementing handshake components although this form of implementation will be the main focus for this thesis.

Ordinarily handshake circuit diagrams show each channel as a single line. The circuit in [fig. 2.1] shows a pipeline of three *buf* single place buffer elements (the Balsa code for which is shown in the figure) connected together with channels and utilising the Passivator component (the component with two passive ports and a single dot within the circular component symbol) to make their active ports compatible.

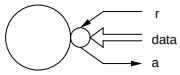
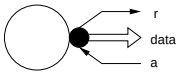
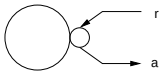
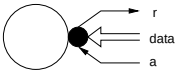
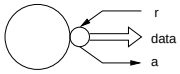
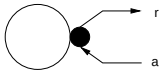
	Input	Output		Nonput
Push	passive input 	active output 	**Sync**	passive sync 
Pull	active input 	passive output 	**Sync**	active sync 

Table 2.1. Push and pull channels, active / passive to input / output mapping

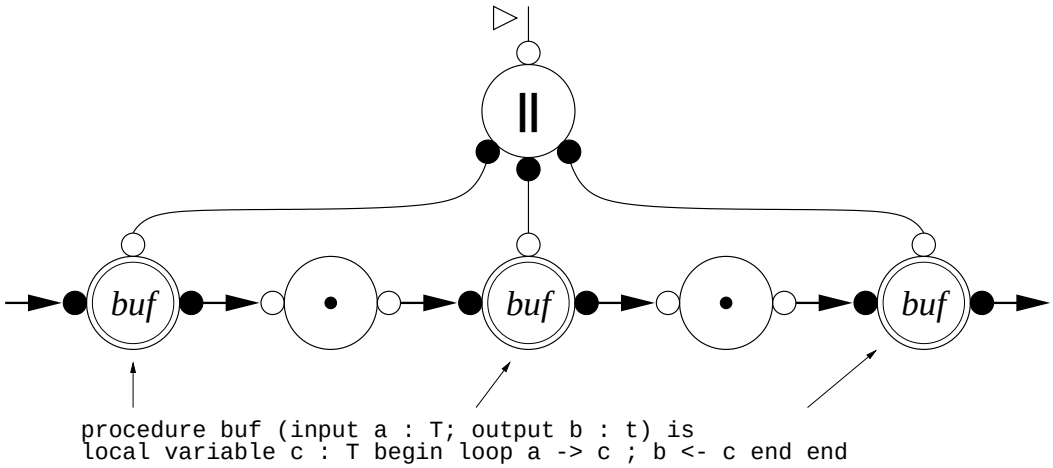


Figure 2.1. A Passivator-connected pipeline of buffer elements

This three place buffer circuit has three ports: the activation (on the top, indicated by the symbol ‘ ▷ ’), an input a and output b. The passive, nonput activation channel is present in all compiled Tangram and Balsa commands and serves to *enclose* the behaviour of the circuit. Enclosure is an important concept in the description of handshake component behaviours and allows us to succinctly describe a single (passive) port handshake request initiating a behaviour and an acknowledgement being asserted when that behaviour has terminated. For example, if an activation handshake encloses a circuit behaviour *A* this can be written:

$$\#[\triangleright : A]$$

where the colon indicates the enclosure of the right hand term by the left hand handshake and the #[ ] indicates (possible) repetition of the bracket enclosed behaviour. In a two phase context this is equivalent to:

$$\#[\triangleright_r; A; \triangleright_a]$$

where sequencing is introduced by the semicolon and the terms $\triangleright_r$ and $\triangleright_a$ refer to transitions on the request and acknowledges of the activation channel. Several interpretations are possible for four phase signalling depending on the usual data validity protocol for the channel. This is the enclosure refinement for four phase early (sometimes known as narrow):

$$\#[\triangleright_r \uparrow ; A; \triangleright_a \uparrow ; \triangleright_r \downarrow ; \triangleright_a \downarrow]$$

the up and down arrows indicate the direction of the signal transition. This enclosure could also be written:

$$\#[\triangleright_r \uparrow ; A; \triangleright_a \uparrow ; \triangleright \downarrow]$$

where $\triangleright \downarrow$ is equivalent to sequenced down transitions on request and then acknowledge.

Three forms of choice terms are part of the notation also, these are:

2.2.1.1. Boolean guarded choice – a multiple guard 'if'

The boolean guarded choice operator pairs up boolean expressions with action terms. The choice operator will evaluate all its guards before choosing a single true guard and executing the term associated with that guard. The general form of this term type is:

$$[G_1 \rightarrow T_1 \mid G_2 \rightarrow T_2 \mid \ldots \mid G_n \rightarrow T_n]$$

The term will terminate after the chosen guard's term is executed (or immediately if no guard is found to be true). Note that an 'else' clause can be specified by the negated disjunction of all the other guards, a simple if G then A else B clause would appear as:

$$[G \rightarrow A \mid \neg G \rightarrow B]$$

2.2.1.2. Boolean guarded repeated choice – a multiple guarded 'while'

This behaves in a similar way to the boolean guarded choice except that after the execution of the term corresponding to a chosen guard the guards will be re-evaluated and a second term may be invoked. This will continue until all guards become false. A boolean guarded repeated choice looks like:

$$\#[G_1 \rightarrow T_1 \mid G_2 \rightarrow T_2 \mid \ldots \mid G_n \rightarrow T_n]$$

2.2.1.3. Signal / communication guarded choice – choice of behaviour based on signalling order

Each guard in this form of choice is a term which begins with the receipt of an input signal. This guard is usually of either of the forms $C; T$ or $C : T$ where C is a communication on a passive port and T is a term to execute. The use of an enclosing communication is particularly useful in the description of passive input handshake components which must enclose an output communication within a choice of input communication (a Balsa CallMux which is the Tangram push MIX component is a good example). The communication guarded choice has the form:

$$[T_1 \mid T_2 \mid \ldots \mid T_n]$$

where each term T_1 to T_n is a communication guarded term as described above. As it is not possible to describe an 'else' clause in which no communication is received (as we can place no timeout on the arrival of the next signal) a repeating form of this term could never terminate (as it would not be possible for all the guards to 'fail') and so no special repeating version of this type of choice is necessary.

With the addition of the concurrent behaviour operator ' || ' this completes the notation. $A \parallel B$ specifies that A and B should be executed concurrently with the composite term terminating when both A and B have terminated.

This notation will be used as the common-currency notation in the remainder of this thesis to

make informal and semi-formal component behaviour descriptions. For the most part this is the same notation as van Berkel's *handshake process calculus* [28] although its informal use is not strictly correct[1].

2.2.2. The Balsa handshake component set

The handshake component set used in the Balsa system is for the most part the same as that used by Tangram. Component and port names, port structures and exact behaviours may differ, however, as the Balsa components tend to be parameterisable with respect to their numbers of ports. For example, in the Tangram target language HCL a mixer component would appear:

```
MIX [PUSH 17] (b_111_,b_18_,b_112_)
```

This is a two input input-request-driven multiplexer with two passive input ports (here connected to channels `b_111_` and `b_18_`) and a single output port (connected to `b_112_`). The port list of the component is specified in the parenthesised term. The square-bracketed term specifies the parameters of that component. Here `[PUSH 17]` implies that this is a push mixer which is 17 bits wide. The behaviour of this component is (informally):

$$MIX(a, b, c).\ \#[[a : c \mid b : c]]$$

Where an output communication on `c` is enclosed by a communication on either `a` or `b` (the movement of data is not shown in this specification).

In Breeze (with the Balsa component set) this `MIX` component is a CallMux component, a similar instance to that above, in Breeze, would be:

```
$BrzCallMux (17,2 | {#111,#18},#112)
```

[1]The notation used in [28] is slightly different. The sense of each channel is indicated in each term involving that channel. For example: $a\uparrow$ in the notation used here would become $a^{\circ}\uparrow$ (for a passive a). This is done simply in the service of brevity. Also in the cause of brevity, the declaration of variables and internal channels will not be shown, it should be assumed that unbound upper case names refer to variable and lower case names to ports or internal channels (port names will be introduced by the enclosing notation *processName*(*ports*). *expression.*

The component name is prefixed with '`$Brz`' to place it out of the namespace of common Balsa identifiers (which may not contain the character '$'), the '`Brz`' prefix identifies this as a Breeze component. The port structure for the CallMux consists of an array of passive inputs and a single active output. The parameters (those arguments before the '|') specify a 17 bit data path and two passive inputs. This component is by this mechanism parameterisable in its number of passive inputs. Trees of such components can all be reduced into a single component to aid optimisation (even if that optimisation just involves expanding this multi-way component into a more balanced tree of two way components).

Balsa names will be used for the remaining components as and when they are necessary. These new names are meant to distinguish the Balsa parameterised components from those of the Tangram system. Tangram names for equivalent Balsa components will be given where appropriate to aid the reader familiar with handshake circuits. As a guide, the 36 components used by Balsa are listed in [tab. 2.2], [tab. 2.3], [tab. 2.4] and [tab. 2.5].

Balsa name	**Tangram name**	**Balsa additions / notes**
Call	MIX [~]	output-count parameter
CallMux	MIX [PUSH]	output-count parameter
CallDemux	MIX [PULL]	input-count parameter
Fork	FORK [~]	output-count parameter
ForkPush	FORK [PUSH]	output-count parameter
Synch	JOIN [~]	port-count parameter
SynchPush	JOIN [PUSH]	passive-output-count parameter
SynchPull	JOIN [PULL]	output-count parameter
Passivator	PAS [~]	port-count parameter
PassivatorPush	PAS []	output-count parameter
Continue	RUN [~]	
ContinuePush	RUN []	
Halt	STOP [~]	
HaltPush	STOP []	

Table 2.2. Balsa and Tangram 'channel pipework' components

Balsa name	Tangram name	Balsa additions / notes
Constant	CST	
UnaryFunc	UN	functions: Negate, Invert
BinaryFunc	BIN	functions: +, -, =, /=, <, >, <=, >=, and, or
Mask	FILTER	
Adapt	ADAPT	
Split	SPLIT	only two output ports
Combine	COMBINE	only two input ports
Variable	VAR	

Table 2.3. Balsa and Tangram datapath components

Balsa name	Tangram name	Balsa additions / notes
Concur	PAR	active-nonput-count parameter
Sequence	SEQ	active-nonput-count parameter
Loop	REP	
Repeat	COUNT	
While	DO	returns ack. on guard failure
WhileElse	DOELSE	never returns acknowledgement
Bar	BAR	guard/command-count parameter
Arbiter	ARB	
Fetch	TFR	
Case	CASE	Balsa uses a ‘specification string’, §5.1.1.4

Table 2.4. Balsa and Tangram control components

Balsa name	Notes
CaseFetch	implements case choice in exprs.
DecisionWait	unarbitrated choice
FalseVariable	allows handshake enclosed reads on channel
NullAdapt	adapts from an input to a nonput channel

Table 2.5. Components new to Balsa

2.3. Tangram

Core Tangram is a hardware description language used by van Berkel [28] to illustrate the semantics and compilation mechanism of Tangram. In this section a superset of Core Tangram is outlined (which remains a subset of the full Tangram language as described in [17]) to illustrate those features which both full Tangram and Balsa share. The language is a derivative of CSP [7] and so describes communication between static concurrent processes by explicit input and output commands. Control flow is expressed by the sequenced and concurrent composition of the language's basic commands.

A Tangram program consists of a single file which describes the whole of a circuit. A the top level a circuit has a number of external ports connecting it to its environment. All of the type, variable and sub-structure definitions used by the program are contained within a block within the scope of these ports.

Within the body of the program, the user can declare procedures which can be instantiated to form part or whole of the description of the circuit. At the end of the list of declarations is a single command which will form the top level of the circuit. This command may be a primitive command, a procedure invocation, or combine other commands using sequencing, concursing, choice or looping constructs.

2.3.1. Tangram basic commands

2.3.1.1. Input, output and synchronisation commands

The communication primitives are used to move data between concurrent modules in a program. Input and output are expressed: `c?v` and `c!e` respectively (where `c` is a channel on which to communicate, `v` is a variable of the same type as that channel and `e` is an expression whose value is to be written across the channel). Synchronisation of two concurrent processes can be effected by a nonput channel connecting those processes, the command `c~` forms one process's end of a synchronisation on nonput channel `c`. Tangram communications have a *broadcast* nature, for all those parallel processes which perform communications on a shared channel, that channel should connect a single output command

with any number of input commands (one in each of the other processes). The one-to-many nature of the broadcast communication when combined with the synchronising command forms a multi-way synchronisation. As no data direction is associated with a nonput communication the synchronising action is symmetric with respect to each synchronising command. Communication primitives can be used to communicate along locally declared channels between concurrent commands, between the ports of concurrently invoked procedures and within the body of a procedure (or the top level) to the ports of that procedure. The top level of a circuit will typically be composed of a number of concurrently invoked procedures connected to the environment by the top level ports and to each other by locally declared channels.

2.3.1.2. Assignment

An assignment (for example `v := e`, where `v` is a variable and `e` is an expression of the same type) has the effect of changing the value present in a variable. Tangram is essentially an imperative language and so assignment and the maintenance of state in variables is the basis of its design. We can also see the assignment as a form of non distributed communication where:

```
                    v := e
is equivalent to  _v?v || _v!e
```

where `_v` is a local channel used nowhere other than in this assignment. Both assignment and the output command define those points where expressions are combined into commands.

Assignment completes the command set of Tangram. The construction of procedures from composite commands, binding of variables and channels to names, port structures for procedures and their inclusion as commands within other procedures are not considered here in the context of Tangram.

2.3.1.3. `skip` and `stop` commands

The `skip` command does nothing. More precisely it can be used wherever a command is required which has no effect and **does** terminate. This command does find a use in real

Tangram descriptions as the Tangram `case` and `if` commands have, by default, behaviours where their process will deadlock if all their guard conditions fail. The `stop` command, by contrast, never terminates. A `stop` command is used to deadlock a process and so finds few practical uses in all but top-level descriptions (where the ability to halt a circuit in such a way that the power must be removed before reactivation may be useful). In combination with other commands the following transformations are semantic preserving (`A` is any command):

```
skip ; A    ⇔  A
A ; skip    ⇔  A
A || skip   ⇔  A
stop ; A    ⇔  stop
```

These definitions are very useful in optimising sequenced or concurrent commands in which one or more commands has been reduced to either the `skip` or `stop` commands by a previous optimising step.

2.3.1.4. Sequencing and concurrency

The `skip` and `stop` definitions above use both sequencing (indicated by `A ; B`, process `A` is executed before process `B`) and concurrent composition (indicated by `A || B`, `A` is executed in parallel with `B`)[1]. Sequenced and concursed commands are themselves commands, giving us a means of composing descriptions from the basic commands (with the aid of repetition and choice composite commands). The definition of Tangram (and also Balsa) makes provision for the static checking of concurrent accesses to shared channels and variables, in particular the two rules:

1. No two concurrent processes may write to the same channel

2. No two concurrent processes may access the same variable except where both processes

[1] It is unfortunate that English has no satisfying verb for the action of making processes concurrent. Concurrent composition and concursing are used here to express that meaning. It should also be obvious that where it is indicated that a command is 'executed' an analogy is being drawn between the execution of instructions on a machine and the enactment of a command's behaviour by a synthesised circuit.

only **read** that variable.

These static checks are necessary for the correct composition of channel interconnections and the correct behaviour of variables in handshake circuit implementation, although they can limit the expressiveness of Tangram and Balsa.

2.3.2. Conditional and repetition commands

To the set of basic commands, Tangram adds a number of repetition and conditional execution composite commands. These include:

Unbounded repetition

```
forever do A od
```

Constant number of repetitions

```
for CST do A od
```

Multiple guarded 'while' loop

```
do G1 ; C1 or G2 ; C2 … od
```

Multiple guarded 'if' command

```
if G1 ; C1 or G2 ; C2 … fi
```

'case' command

```
case E is CST1 then C1 or CST2 then C2 esac
```

The use of multiple concurrently evaluated guards in the `while` and `if` commands follows the use of similar guarded structures in CSP. In the event of multiple guards becoming simultaneously true, the choice of which single guarded command to execute is non-deterministic.

2.4. The Tangram compilation function

The Tangram compilation function defines the *syntax directed* mapping from Tangram

commands into their handshake circuit implementation. Such a mapping preserves the form of the original Tangram description in the implementation and so places the burden of higher level optimisation on the designer. This reliance on transparent compilation to parameterised macromodular components and the design – compile – evaluate loop for hand optimisation (where an automated route may have an iterating compiler or back-end optimiser) is distinctive to Tangram in the field of hardware description languages.

The function described here is that outlined by van Berkel in chapter 7 of [28]. This function has an 'active ported' nature and employs both push and pull channels connected by passivating Passivator, Synch and Call trees where composition of commands demands. The transformation rules fall into three rough categories:

Compilation of the basic commands.

The basic commands connect channels, variables and expressions together to form single commands which have an activation and external ports similar to the components described in §2.2.

Compilation of the compositional commands

The compositional commands have the task of combining commands to form a single command. Accesses to shared channels, variables and ports must be resolved and combined.

Compilation of expressions

Expressions are composed in a similar manner to compositional commands except that their activation must carry a result value back to the command in which they appear, also expressions make no accesses to channels. Tangram expressions have a pull nature and as such they always have the form of a tree of datapath components connecting read ports of variables (each variable being a concrete component, a Variable) to a single, pull, activation port which returns the result of the expression.

In addition to these categories there is the compilation of a block enclosing channel, variable and port declarations with a command in that context. This entails the connection of dangling channels connected to points within a command which access variables or channels to the

components created by the declarations of those variables or channels. This results in a set of components with fewer dangling channels and unresolved accesses.

2.4.1. The basic commands

Input, output and assignment are implemented in a similar manner. A Fetch component (a transferrer) transfers the result of an expression to a channel or variable, that channel value is picked up by another transferrer at the far end of the communication (or transferred directly into a variable in the case of assignment). The use of an active ported transferrer is what leads us to the push / pull active ported implementation of Tangram. Much optimisation effort can be applied to reducing compositions of transferrers and connecting components to a form more like an assignment (the equivalence of assignment and a point-to-point communication was mentioned in §2.3.1.2). Nonput communication can be implemented by connecting the command's activation to the channel on which the communication is to take place.

The commands `stop` and `skip` are implemented by the Halt and Continue handshake components respectively. Halt is the empty component where the incoming request is unconnected and the acknowledge is tied off. Continue, on the other hand, loops back request to acknowledge and so fulfils its role as NO-OP in the component set.

Sequencing and concurrency introduce the notions of combining the accesses made by two commands (and their activations) into a single command. The activations of the composed commands present no problems, Sequence and Concur components fan a single activation (that of the composite command) into the multiple activations of the contained commands. The accesses those commands make to shared variables and channels are resolved at this point by the connection of channel accesses with Synch, Call and Passivator components (it is not the intention to detail this process here as it is essentially the same one used by Balsa). The functions *Mix* and *Join* (which are detailed in §7 of [28]) specify the manner in which accesses are combined for sequential and concurrent command compositions.

2.4.2. The compositional commands

The remaining compositional commands (other than sequencing and concurrency) are implemented in a similar manner to the sequence and concurrent commands. Special

components exist for the implementation of each command (Loop for unbounded repetition, While for the `do` loop …) where commands and expressions are combined to give the uniform: passive activation, external instance reference, active ported compositions. A point of interest here is the way that the activation of a composite command encloses the activations of the commands of which it is composed. This allows repetition by repeated activation of the command. The conditional commands are implemented by effectively sequentially combining the accesses made by the commands of each choice thus reflecting the sense that they are mutually exclusive in a similar manner to the disjoint activation of sequenced commands.

2.4.3. Expressions

Expressions are kept relatively simple in Tangram. There are no conditional expression forms or state holding elements. Indeed it is impossible to implement the data-driven nature of other descriptive paradigms without placing explicit communication commands. The pull nature of Tangram expressions does allow us a single transferrer to implement assignment. This single transferrer makes it easier to sequence operations by the control of the single activation of that transferrer rather than having to monitor all of the incoming communications to the assignment's expression as would be necessary in a data driven system. Results are, as a consequence of this assignment implementation, only generated when requested. Any data driven or parallel computation must be explicitly coded for by the designer. The trade off between a pull / transferring implementation and the optimisation possibilities inherent in mapping familiar patterns of such implementations into push (data driven) components such as those used in the Micropipeline design style remains an interesting field of research.

The lack of conditionals in expressions can be particularly annoying. Consider an adder which can conditionally complement one of its inputs[1]:

```
case invert
is true then result := lhs + rhs
or false then result := lhs + complement (rhs)
```

[1]The function `complement` is used here to invert the bits of the word rhs. The use and declaration of functions and the lack of a bitwise complement operator are not relevant here.

```
esac
```

This will yield a functionally correct implementation although two adders will be placed instead of one as the '+' symbol appears in the description twice. Alternative implementations with a single adder require an extra variable to hold the intermediate result of complementing (or not) `rhs` before performing the addition. With a conditional expression operator (similar to the ternary `? :` operator in C) we could rewrite this example:

```
result := lhs +
  if invert then complement (rhs) else rhs
```

Tangram does not have this expression form and Balsa does not add it. Adding conditional expressions to the language would restrict the ability to add communication primitives to the language of expressions and for the meaning of those expressions to be both clear and simple to implement. Consider the case of a conditional expression, where one branch performs a communication but the other branch does not. Should the conditional expression evaluate all its guarded expressions in parallel with the guard then make a choice, leading to a possibly unintended communication? Alternatively, should only the chosen expression be evaluated so sequencing the evaluation of guard and guarded expression? When the guard itself contains a communication the sequenced / concursed behaviour of the guard and guarded communications must be made clear.

The Balsa enclosing input selection mechanism (which is described in §3.2) does allow one futher form for this example:

```
local channel processedRhs : someType
begin
  if invert then processedRhs <- not rhs
  else processedRhs <- rhs ||
  select processedRhs then
    result := lhs + processedRhs
  end
end
```

Where the channel `processedRhs` is used to carry the right hand side value to the addition where no variable is required to hold its value due to the use of the `select` mechanism.

2.5. Chapter summary

This chapter is very much a whistle-stop tour of the concepts and terminology surrounding handshake components. Special note should be taken of:

- The change of handshake component names from Tangram.
- The simplified form of the notation of handshake process calculus described. This is the only behavioural notation used in the remainder of this thesis.
- The terms type, direction and sense for port structures.
- The terms push and pull for channel communications.

Every effort is made to use both the notation and terminology of this chapter consistently throughout the body of the thesis.

Chapter 3. The Balsa Language

The Balsa language was developed as a response to a need for a language to replace Tangram in an existing Tangram application, viz. the single-rail implementation of Tangram handshake circuits carried out by The University of Manchester under the EXACT project. As such Balsa strongly and shamelessly resembles Tangram and can be considered to be an extension of that language. This chapter describes the syntax and semantics of Balsa by comparison with Tangram.

The main features which Balsa brings to Tangram are outlined along with their implementations in handshake components. These features include an input communication selection mechanism similar to the Ada rendezvous, arrayed channels and support for better parameterisation and separate compilation. These additions address specific practical requirements from the language. In the case of input selection a number of examples are given to illustrate the use of the new feature.

3.1. Balsa as an extension of Tangram

The Balsa language can be viewed as offering an extension to the functionality of Tangram. Balsa shares the same channel communications and 'activation orientated' control flow as Tangram along with many of the same control composition commands. For compilation to handshake circuits the same active ported compilation technique (as outlined in §2) is used. At the top-level Balsa has no concept of the program. A design may consist of many files containing procedure / type / constant declarations which come together in a top-level procedure which composes described the structure of the design. This top-level procedure would typically be at the end of a file which **`import`**s all the other relevant design files. This importing feature forms a simple but effective way of allowing component reuse and maps simply onto the notion of the imported procedures being either pre-compiled handshake circuits or existing (possibly hand crafted) library components. This approach is similar to the use of header files in the C language or DEFINITION modules in Modula 2. Declarations have a syntactically defined order (left to right, top to bottom) with each declaration having its

scope defined from the point of declaration to the end of the current (or importing) file. This gives us the same simple 'declare before use' rule of C and Modula although Balsa has no facility for 'prototypes' which are used in C to facilitate the definition of mutually recursive functions and datatypes. 'Declare before use' is more restrictive but easier to implement than the simultaneous declarations of Java and VHDL.

Each Balsa design file has this form[1]:

⟨file⟩ ::= ⟨imports⟩ ⟨privates⟩ ⟨publics⟩

⟨imports⟩ ::= (**`import [`** ⟨path⟩ **`]`**)*

⟨privates⟩ ::= ε
| **`private`** ⟨declarations⟩

⟨publics⟩ ::= ε
| **`public`** ⟨declarations⟩

The import list ⟨imports⟩ for a file precedes any declarations contained in that file, simplifying the handling of name-space conflicts which may occur between imported and local declarations. The file then contains an optional set of private declarations which are visible only to subsequent declarations in the remainder of this file and a set of public declarations which are ⟨import⟩able by other files once this file has been compiled. The ⟨path⟩ given in an `import` statement is a dot separated directory path to the file required. For example, the statement `import [breeze.types.synthesis]` appears in all Breeze output files to import the parameter types necessary for the Breeze handshake components. The file examined by this statement will be '…/breeze/types/synthesis.breeze' where the '…' is the first directory in a given search path (given as an argument to the compiler or by way of an environment variable) for which the completed path refers to an existing Breeze file. This dotted path search method is similar to that of Java except that Balsa does not implement Java's notion of multi-file packages. It was felt that a simple, file orientated structuring of designs was an improvement over both the packaging mechanism of VHDL (for which there

[1]An extended form of BNF is used to describe valid syntax in this chapter. A term (*a*) ∗ denotes zero or more repetitions of the term *a* and (*a*)? indicates that the term *a* is optional.

is no standard mapping to the notion of the file) and the C preprocessor like mechanism of Verilog.

3.1.1. Declarations

⟨declarations⟩ ::= (⟨declaration⟩)*

⟨declaration⟩ ::= **type** ⟨identifier⟩ **is** ⟨type_decl_body⟩
| **constant** ⟨identifier⟩ **=** ⟨expression⟩ (**:** ⟨type⟩)?
| **procedure** ⟨identifier⟩ (**(** ⟨formal_ports⟩ **)**)? **is** ⟨block⟩

Declarations introduce new type, constant or procedure names into the global namespaces from this point until the end of the enclosing block (or file in the case of top-level declarations). There are three disjoint namespaces: one for types, one for procedures and a third for all other declarations (just constants for top-level declarations but for procedure local declarations this includes variables and channels). The syntactic distinctions made by the language as to where the use of type, variable and procedure names may be used makes the use of multiple namespaces possible. Where a declaration within an enclosed / inner block has the same name as one previously made in an outer / enclosing context the local declaration will hide the outer declaration for the remainder of that inner block.

Constant declarations bind values (which may be of any type) to names. The value of a constant is taken from a compile time constant expression. The type of such a name when used in an expression will be the same as the ⟨expression⟩ whose value was bound to that name, the inclusion of the ⟨type⟩ term of a constant declaration is usually used to exploit compile time type checking to suggest or enforce a type for the constant. Constants declared in the public body of a file are exported to the breeze output file. Constant declarations use the symbol '**=**' to signify that a binding of a value to a name is made. Types and procedures are not first class 'valued' objects in Balsa and so the separator '**is**' is used in their declarations.

⟨type_decl_body⟩ ::= ⟨type⟩
| **record** ⟨record_elements⟩ (**end** | **over** ⟨type⟩)
| **enumeration** ⟨enum_elements⟩ (**end** | **over** ⟨type⟩)

⟨type⟩ ::= ⟨identifier⟩
| ⟨expression⟩ (**signed**)? **bits**
| **array** ⟨range⟩ **of** ⟨type⟩

The **over** ⟨type⟩ terminated form of record and enumeration declarations allows the user to specify a bounding type for that record or enumerated type. The bounding type is so named because it provides a maximum, bounding, limit to the size of the declared type. This is useful to both allow the creation of types which are padded to a given width and to prevent record and enumeration types from exceeding a maximum width by allowing the user to specify that width, for example: `type a is record b,c : byte over 17 bits` declares a type `a` which has 16 useful bits in two elements `b` and `c`. The bounding type is `17 bits` and so any value of type `a` has width 17, the most significant bit being a padding bit. If a record type is subsequently defined which has elements of type `a`, those elements will each have a width of 17 bits so introducing padding bits into the body of the newly declared type. For record type declarations these padding bits can be explicitly included as a redundant element of the type, this will however affect the syntax of record construction expressions and also force the compiler to test for matching of read and write accesses to those elements. Enumeration types with padding can also be declared by including a redundant element of the required width in the enumeration declaration, the `over` syntax is thereby just syntactic sugar. The width of the bounding type must be greater or equal to the width of the record or enumerated type it bounds.

The term ⟨type⟩ is used where a pre-declared or aliased (numeric / array) type is required. Record and enumerated types cannot be aliased and so a record or enumeration declaration is only valid within the type declaration body.

3.1.2. Typing

3.1.2.1. Numeric types

Numeric types incorporate numbers over either the range $[0, 2^n - 1]$ or $[-2^{n-1}, 2^{n-1} - 1]$ ($n \in [1, INT\_MAX]$, on a 32b machine $n \in [1, 2^{32} - 1]$). Named numeric types are (for the purposes of type equivalence) just aliases of unnamed types of the same range. Examples of numeric types are:

`8 bits` gives the range $[0, 2^8 - 1]$

or `8 signed bits` gives the range $[-2^7, 2^7 - 1]$

Numeric types are the most commonly used class of types in Balsa descriptions. They allow the user to simply declare a bit granular general purpose numeric range. The intention here is to reflect the concrete nature of the implementation by not allowing numeric ranges which do not reflect the range granularity inherent in a binary implementation. Numeric types serve a role similar to the simple bit orientated typing used by Verilog and to some extent the bit vectors of VHDL.

3.1.2.2. Enumerated types

Enumerated types consist of named numeric values. Enumerated types must be given names and are therefore checked for equivalence by name. The named values of an enumeration are given values starting at zero and incrementing by one from left to right in the same way as for the language C. Names can be individually bound to specific numbers at the point of declaration and many names can be given to the same value. Each enumerated type has an associated numeric type which is the smallest numeric type which will hold all of the enumeration values. The identifiers associated with the elements of an enumeration only take their meaning as part of the enumeration where a value of that enumerated type is expected. For instance, if `a` has type `Colour` which was declared:

```
type Colour is enumeration
        Black, Brown, Red, Orange, Yellow, Green,
        Blue, Violet, Purple=Violet, Grey, Gray=Grey,
        White
end
```

The command `a := Grey` is valid and assigns the value of the `Grey` element of `Colour` (viz. 8) to `a`. The command `a := Colour'Grey` is also valid as it specifically qualifies the identifier `Grey` as being a member of `Colour`, this is useful in the context of constant declarations where a specific type is indicated:

```
                constant NkMMultiplier = Colour'Red
```
being equivalent to
```
                constant NkMMultiplier = Red : Colour
```

Note that in the second case the '`: Colour`' is part of the syntax of constant declarations rather than a type qualifier for the right hand side expression.

The implementation of enumerated types preserves the directness of translation into hardware. The user can assign specific values to specific elements of the enumeration. The user may also define the number of bits used by the enumeration by either specifying elements within a certain range (for which the size of the 'smallest' numeric type which can contain all those values is used) or by explicitly providing a bounding type. Enumerations do not, however, exhibit the 'special type' / numeric type duality that ANSI C **`enum`** types enjoy. Enumerations may only be used in numeric expressions by explicit type casting. The identification of an enumeration element's type by context and the sharing of element names between enumeration types is a feature taken from VHDL.

3.1.2.3. Record types

Records are bitwise compositions of named elements of possibly different pre-declared types e.g.:

```
type Resistor is record
        FirstBand, SecondBand, Multiplier : Colour;
        Tolerance : ToleranceColour
end
```

`Resistor` has four elements: `FirstBand`, `SecondBand`, `Multiplier` of type `Colour` and `Tolerance` of type `ToleranceColour`. `FirstBand` is the first

element and so represents the least significant portion of the bitwise value of a value of type `Resistor`, `R15.SecondBand` will give the `SecondBand` value from value `R15` of type `Resistor`.

Records are similar to C **`struct`**s with bitfields except that a defined order and packing for the elements of the type is defined. The Balsa record structure is intended to be similar to the tupled type construction used in Tangram (albeit with named fields).

3.1.2.4. Array types

Arrays are numerically indexed compositions of same-typed values. Arrays can be created anonymously as with numeric types; `array 0..7 of SomeType` is an array type of 8 elements indexed across the range [0, 7], each element has type `SomeType`. Arrays are typically used as the basis of register banks (or other addressable register system) and with array slicing to allow arbitrary bitfield extraction, for example:

```
((instruction as array 0..31 of bit)[31..28]
    as InstructionType)
```

performs a type cast on the value `instruction` into a bitwise array from which a 4 bit slice is extracted and cast into the type `InstructionType`. Note that the order of the indices in a slice operation is irrelevant, `[0..31]` is equivalent to `[31..0]`. In general Balsa packs all composite typed structures in a least significant to most significant, left to right manner.

Array types allow the creation of indexable variables in Balsa but also allow bit extraction from words in a similar way to the simple bitwise types of Verilog and the identical use of arrays (with indexing and slicing) in VHDL. A Tangram description requiring bit slicing would typically require the declaration of a new (tupled) type and a cast into that type followed by an element extraction. The addition of array slicing makes the declaration of a new type unnecessary.

3.1.3. Procedures

Procedures form the bulk of Balsa description. Each procedure has a name, a set of ports

and an accompanying behavioural description in the same way as Tangram. Procedure declarations follow this pattern:

⟨declaration⟩ ::= …
| **procedure** ⟨identifier⟩ (**(** ⟨formal_ports⟩ **)**)? **is** ⟨block⟩

⟨formal_ports⟩ ::= ⟨port⟩ (**;** ⟨port⟩)*

⟨port⟩ ::= (**input** | **output**) ⟨identifiers⟩ **:** ⟨type⟩
| **array** ⟨range⟩ **of** (**input** | **output**) ⟨identifiers⟩ **:** ⟨type⟩
| **sync** ⟨identifiers⟩
| **array** ⟨range⟩ **of sync** ⟨identifiers⟩

⟨identifiers⟩ ::= ⟨identifier⟩ (**,** ⟨identifier⟩)*

⟨block⟩ ::= (**local** ⟨local_declarations⟩)? **begin** ⟨command⟩ **end**
| **(** ⟨command⟩ **)**

⟨local_declarations⟩ ::= (⟨local_declaration⟩)*

⟨local_declaration⟩ ::= ⟨declaration⟩
| **variable** ⟨identifiers⟩ **:** ⟨type⟩
| **channel** ⟨identifiers⟩ **:** ⟨type⟩
| **array** ⟨range⟩ **of channel** ⟨identifiers⟩ **:** ⟨type⟩
| **sync** ⟨identifiers⟩
| **shared** ⟨identifiers⟩ **is** ⟨block⟩

The appearance of procedure declarations is superficially similar to procedure declarations in VHDL. Each procedure may have a number of ports (each of which can be connected to a channel), the `sync` keyword introduces nonput channels in Balsa. Both nonput and data-bearing channels can be members of ‘arrayed channels’ (those declarations in the above BNF which include the `array` keyword). Arrayed channels allow numeric / enumerated indexing of otherwise functionally separate channels, arrayed channels are explained further in §3.1.5.2.

Procedures may also bear a list of local declarations which may include other procedures,

type and constant declarations which will never appear in an output breeze file or exist outside the scope of that procedure as well as the variable, shared block, channel and arrayed channel declarations local to the ⟨command⟩. Local channels may be arrayed in the same way as ports.

The ubiquitous Tangram single place buffer looks like this in Balsa:

```
type word is 8 bits

-- This is a comment: Single place buffer
procedure buffer (input i : word; output o : word) is
local variable x : word
begin
  loop
    i -> x;  -- Input communication
    o <- x   -- Output communication
  end
end
```

With the exception of syntactic changes this description is essentially the same as the original Tangram program, which has a similar form to either a CSP or OCCAM description of the same behaviour. For the majority of straightforward input – process – output descriptions the Balsa description will be virtually identical in form to the Tangram description. The comparable commands of the two languages are listed in [tab. 3.1].

3.1.4. Tangram features missing from Balsa

Balsa lacks a number of features found in the full version of Tangram. In most cases uses of such features can be replaced by use of either an alternate style of description or by the use of constructs present in Balsa which are not present in Tangram.

Notable examples are:

3.1.4.1. Variable procedure arguments

Variables cannot be passed to procedures as arguments, only channels can be connected to procedure ports. For a procedure which accesses the same variable multiple times (especially

Tangram	Balsa	Notes
a?x	a->x	x must be a variable.
b!y	b<-y	y is an ⟨expression⟩
c~	sync c	
stop	halt	
skip	continue	
z := e	z := e	No implicit type cast is performed in Balsa.
A ; B	A ; B	
A || B	A || B	|| has higher precedence than ;
forever do ... od	loop ... end	
if ... then ... or ... then ... else ... fi	if ... then ... also ... then ... else ... end	without an else clause Tangram defaults to else stop, Balsa defaults to else continue
do ... then ... or ... then ... else ... od	while ... then ... also ... then ... else ... end	
case ... of ... then ... or ... then ... else ... esac	case ... of ... then ... also ... then ... else ... end	Balsa allows guards of the form 1,2,3..6

Table 3.1. Commands in Balsa and Tangram

in parallel) more than one port must be present for each access to the same variable. This makes the port structure of a procedure dependent on the internal variable access behaviour which can be considered to be detrimental to the aims of 'separate compilation'.

3.1.4.2. Tuples and simultaneous assignment

The Tangram statement <<a,b>> := <<c,d>> will assign the current value of expression c to variable a and likewise d to b. This form of assignment depends on the construction of an anonymous tuple typed expression from the expressions c and d as well as the construction of an assignable expression (an lvalue or location) from variables a and b. Balsa has neither

anonymous tuples nor composite typed lvalues. Parallel assignment must be achieved by the statement: `a := c || b := d`. The slippery subject of structural type compliance in a type system with a single bit type size granularity is thereby side-stepped.

3.1.4.3. Implicit type-casting

The only form of implicit type-casting present in Balsa is the promotion of numeric literals under an expectation of a wider numeric type, for example: `a := 32` where `a` has type `32 bits` is valid because the right hand expression is expected to have the same type, is a literal numeric constant and has an inherent type (number 32 has type `6 bits`) which is narrower than 32 bits. This hyper-strict typing together with the trinity of numeric, bitwise array and record typing typical of many design descriptions can make for unwieldy sequences of explicit casts. Consider an instruction register in a microprocessor which is loaded from a numeric type channel and from which fields are extracted by using 'mask' record types or by bitwise addressing using an array type. Each 'masking' operation performed on this register will require one or more casts to achieve. A simpler case is the very simple assignment statement `x := x + 1`. This is not valid Balsa as the type of an addition of a 32 bit (the `x`) and a 1 bit type (the literal `1`) is `33 bits`. `x := (x + 1 as 32 bits)` is the correct form of such an assignment. This strict approach to numeric typing takes much of the guess work out of operations which have different return types from their arguments. In this particular example it forces the user to remove (with a truncating type-cast) the extra carry out bit from the addition.

The 33rd bit of a `32 bits` × `32 bits` addition is the carry out of the most significant bit of that addition. This carry out bit can be used by casting the addition result into a record type:

```
type AddResult is record Result : 32 bits; Carry : bit
end
variable r : AddResult
…
r := (a + b as AddResult)
```

The expression `r.Carry` will give the required carry bit, `r.Result` yields the 32 bit addition result. A similar kind of record type could be used to extract the most significant bit

of `r.Result` to produce a negative flag. By the use of record and array types, bit extraction can be made considerably less painful than if explicit shifts and masks were necessary.

3.1.4.4. CSP like communication selection mechanism

Balsa replaces the communication guarded selection of Tangram / CSP with a similar construct which forces the guarding communication to enclose the body action. This construct is examined further in §3.2.

3.1.4.5. `ram` and `reg` distinctions for arrayed memory access

Tangram provides different declaration forms for arrayed memories with different access patterns. `ram` provides a single read, single write port memory (single simultaneous read or write) possibly implemented with an SRAM macrocell. `reg` provides for multiple read port memories of the type used in register banks. Balsa allows variable declarations to have array types. The array type can be used to implement both of these forms of memory (although the implementation is with variable handshake components and so it is better to provide support for SRAM through an externally defined procedure) and also to allow bitwise manipulation of variables.

For the declaration:

```
variable z : array 0 .. 7 of 4 bits
```

`z` is a single bit-vector 32 bits in length consisting of nibble sized chunks numbered 0 to 7 from least to most significant bit places. `z` can be cast into a 32 bit type without loss of bitwise representation and can be sliced and cast into other sized types, e.g. `z[1..2]` returns a slice of `z` of type `array 0 .. 1 of 4 bits` (array slices always return array values which are based at index 0. Type compliance for array types only considers the number of elements and the element size not the exact range of indices). The way in which array type accesses are implemented using slices of variables is outlined in §4.3.1.3.

3.1.5. Balsa features not present in Tangram

Balsa includes a number of features not present in Tangram, these enhancements include mechanisms for supporting better code reuse, parameterisation and the interfacing of Balsa with existing 'handshake component interface wrapped' parts and raw handshake component netlists into Balsa descriptions.

3.1.5.1. Separate compilation

The breeze intermediate file format is both the target and module format for the balsa-c compiler. Pre-compiled Balsa modules as well as cells already in the back-end CAD framework's database which have handshake circuit wrappers, can be picked up by other Balsa descriptions for reuse. Reuse can either imply the inclusion of the handshake component network from the included breeze file producing a flat HC netlist or by reference to the imported breeze file's declarations. Breeze has an intentionally simple format to allow parsing by simple scripts without knowledge of the original Balsa port typing other than port widths.

3.1.5.2. Arrayed channels

As previously stated, procedure port and procedurally local channels may be *arrayed*. That is they may consist of several distinct channels which can be referred to by a numeric or enumerated index. This is similar to the way in which variables can have an array type but in the case of arrayed channels each channel is distinct for the purposes of handshaking, each indexed channel has no relationship to the other channels in the arrayed port / local channel other than the single name they share. The syntax for arrayed channels is different to that of array typed variable declarations, this makes it easier to disambiguate arrays (which are types) from arrayed channels (where the arraying is a feature of the declaration). As an example:

```
array 0 .. 3 of channel ArrayedChannel
  : array 0 .. 3 of nibble
```

declares four channels `ArrayedChannel[0]` to `ArrayedChannel[3]` each of the

16b wide type `array 0 .. 3 of nibble`.

3.1.5.3. Type system features

The enumerated and record type classes described in §3.1.2 are not present in Tangram. The Tangram type system is based on anonymous tuples with positionally numbered members. The array and arrayed channel features are also not present in Tangram although the `ram` and `reg` keywords do allow for arrays of variables (see §3.1.4.5).

3.1.5.4. The `for` command

The `for` command is similar to the `for` … `generate` … statement of VHDL [20] [21] or to a lesser extent the `forpar` and `forseq` expressions of LARD [35]. `for` is a structural command unlike `repeat` and finds its majority of applications in allowing the parameterisation of pipelines. The command has the following form:

⟨for_command⟩ ::= **`for`** (**`||`** | **`;`**)? ⟨identifier⟩ **`in`** ⟨range⟩ **`then`** ⟨command⟩ **`end`**

The generated body commands can be either sequentially or concurrently composed depending on the keyword following the `for` keyword in a `for` command (concurrently for '`||`', sequentially for '`;`' or no keyword).

An *n* place buffer ($n \geq 2$) using the `buffer` declaration from §3.1.3, arrayed channels and the `for` command looks something like this:

```
constant n = 5

-- buffer_n: n place buffer using procedure buffer
procedure buffer_n ( input i : word; output o : word ) is
local array 1 .. n-1 of channel ichan : word
begin
  buffer (i, ichan[1]) ||   -- First buffer
  buffer (ichan[n-1], o) || -- Last buffer
  for || i in 1 .. n-2 then -- i th buffer
    buffer (ichan[i], ichan[i+1])
  end
end
```

The `for` command allows the repeated placement of a number of instances of a body ⟨command⟩ with the activations of the separate instances connected by either a sequencer or a concursor (dependent on the choice of either `for ;` or `for ||` forms of the `for` command). The given `identifier` is bound as a constant within each instance of the body. The ⟨range⟩ dictates how many occurrences of the body are placed and also the value of the bound constant in each instance, for sequentially placed commands the body commands will be executed with the bound constant taking ascending values across the range of the ⟨range⟩ term. The ⟨range⟩ term may range over numeric or enumerated types.

3.2. Input choice with `select`

The Balsa `select` construct is similar to the selection mechanism present in the Micropipeline language Yellow [18] (and by further elaboration that of task entry selection in Ada [24]). This form of selection was chosen due to its flexibility (see §3.2.2) and for the low cost of its implementation (see §4.3.3 and §3.2.3). The `select` command has the following form:

⟨select_command⟩ ::= (**`select`** | **`arbitrate`**) ⟨select_terms⟩ **`end`**

⟨select_terms⟩ ::= ⟨select_term⟩ (**`also`** ⟨select_term⟩)*

⟨select_term⟩ ::= ⟨channel_guards⟩ **`then`** ⟨command⟩

⟨channel_guards⟩ ::= ⟨channel_name⟩ (, ⟨channel_name⟩)*

⟨channel_name⟩ ::= ⟨identifier⟩
| ⟨identifier⟩ [⟨expr⟩]

As an example: (where lower case letters are channels and upper case letters are commands.)

```
select a,b then A
also   c   then B
also   d   then C
end
```

When encountered, the `select` command will wait for one of its guard conditions to become true. This implies that all the channels listed in one of the commands ⟨channel_guards⟩ terms must become capable of providing an input. The channels listed in each ⟨select_term⟩ must be disjoint from all other channel sets in the same `select` command and each channel must be capable of an input communication. After the guard becomes true, the command associated with that guard will be executed and once this command has completed control returns to the command sequentially following the `select` command.

3.2.1. Differences from Tangram select

This behaviour is different from the Tangram select construct in several ways.

3.2.1.1. No output selection

Only input selection is permitted. Output selection can be simulated with a pair of channels, one a request and the other a response with a ⟨select_term⟩ of the form: *request* `then` *response* `<-` *expression*. Output selection would require either the output command to be the last action of the guarded command or for the output value to be stored pending completion of the guard communications (see the following note on enclosure). The first option would require potentially complicated analysis of the command structure to ensure the final action is actually the guarding communication and the second the addition of extra hardware and control to the `select` implementation.

3.2.1.2. Multiple guard channels

Each guard may consist of multiple input channels. This is particularly useful where no choice is required but the passive input semantics of guard borne input channels is desired.

3.2.1.3. Guard enclosure

The input communications on the guard channels for a given command **enclose** the command for that ⟨select_term⟩. This behaviour allows the process at the sourcing end of the communication to be stalled by the action of the command. The values present on the input

channels can be made accessible to the command as read-only variables without imposing the cost of a variable or multiple communications of the same value across the input channels.

3.2.1.4. Lack of arbitration

The choice offered by `select` is unarbitrated. Substituting the keyword `arbitrate` for `select` does allow a two way arbitrated choice to satisfy the fundamental need for arbitration in some forms of description. For the unarbitrated `select`, behaviour in the case where two or more guards become true simultaneously is undefined. The case of a second guard becoming true during the execution of a command associated with a different guard is similarly undefined. Both of these cases may occur in a description in which two inputs are unsequenced by the environment. It is, therefore, important that an environmental constraint guaranteeing mutual exclusion of communications on the channel guard sets be present to ensure correct behaviour of the circuit.

The lack of arbitration allows a very low overhead implementation of selection in those cases where the environment does imply mutual exclusion of guard communications yet punches a sizable hole in the simple, compositional resource access interference checking performed on parallel processes. Guarded commands are considered to be strictly mutually exclusive by the compiler and so may interfere with one another if mutual exclusion is not enforced. In the absence of a mechanism for describing higher-level constraints on circuit behaviour the lack of arbitration allows designers to apply their knowledge of such constraints to produce more compact and efficient implementations.

3.2.2. Uses of selection

The `select` construct allows the description of Micropipeline-like components with passive inputs and active outputs: this increases the expressiveness of Balsa descriptions by both allowing selection of flow of control by input arrival order and by allowing input communications without a variable. Compare:

```
      a -> v ; A
with  select a then A end
```

The first command performs the communication and operation A in series allowing the component at the far end of a to proceed whilst A is being executed, the second command forces the component at the far end of a to stall whilst executing A. This feature allows the omission of the variable v and the use of shared resources between the far end component and A to be shown to be mutually exclusive, that is to say that during the execution of this communication the sequential process of which the write end of the communication is a part performs no variable or channel accesses other than those present in the expression whose value is communicated along a.

Consider this example:

```
-- top: shared resource failure
procedure top is
local
  variable a : word
  channel b : word

  procedure procA (output b : word) is
  begin
    <A command manipulating a> ;
    b <- <expr> ;
    <A command manipulating a> ;
  end

  procedure procB (input b : word) is
  begin
    select b then
    <A command manipulating a> ;
    end
  end
begin
  procA(b) || procB(b)
end
```

Here processes procA and procB both access the shared variable a. This example would not however be considered valid by balsa-c as the two processes both manipulate a and so (using the simple compositional access interference rules outlined §2.3.1.4) their parallel composition is not interference free. If the possible orderings of accesses to a are considered, it is obvious that procB can only access a during a communication with procA on channel b. During such a communication no other parallel process within procA (and also no other

thread present in the body of process `top`) makes use of `a`. The accesses to `a` can, therefore, be seen to be interference free.

The automated analysis of such communication enforced mutual exclusion (and also similar access-pattern enforced mutual exclusion enforced by the algorithm implemented) of accesses pattern would be a useful addition to to the language but is not a current feature of balsa-c.

The `select` construct can be used for both input choice and choiceless passive inputs. Micropipeline-like unbuffered input pipeline elements are made possible. Some examples of these uses are:

3.2.2.1. Input selection – the unbuffered multiplexer

The set of handshake components used in the Tangram system includes a push multiplexer (The Push Mix Component). This is also part of the component set for the Balsa compiler (The CallMux component – Balsa back-end components have Micropipeline-like names where functionality is similar to a Micropipelined component). This multiplexer implements a simple unarbitrated choice between two input ports. The chosen input channel's value is then output on the multiplexer's third, active, output port whilst blocking the process at the active end of the selected passive input. This behaviour requires that the communication on the input guard channel encloses the output communication and so suggests a form of select language construct in which the guard is not a communication but simply a probe on an input channel which if true initiates a single communication enclosing the guarded command. In handshake process calculus this may be written: $\#[[x : A \mid y : B]]$ where within A the value on the channel x is available and likewise with command / guard pair B and y.

In Balsa the (byte wide) unbuffered multiplexer is described:

```
-- mux: Unbuffered multiplexer
procedure mux (input a,b : byte; output c : byte) is
begin
  loop
    select a then c <- a -- a and b behave like variables
    also b then c <- b   -- within their guarded commands
    end
  end
end
```

The resultant handshake circuit is shown in [fig. 3.1]. This implementation introduces two new handshake components: the FalseVariable and the DecisionWait. The symbols for these components and a description of their behaviours can be found in §3.2.3.

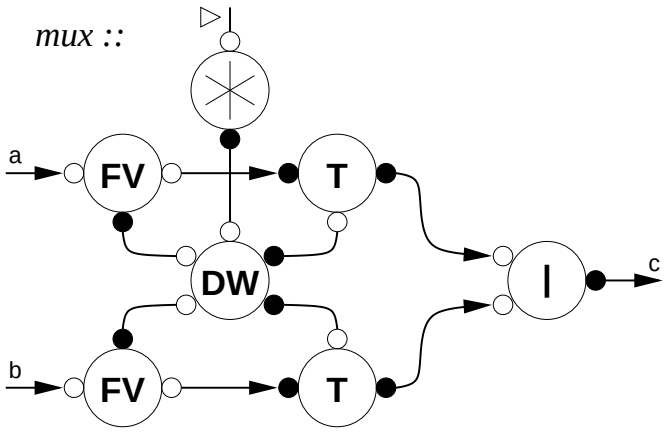


Figure 3.1. *mux* implementation in handshake components

3.2.2.2. Unbuffered choiceless pipeline elements – tupling element

In practice, push pipeline components can often be constructed without input choice. A Balsa description of a push combine (input tupling) handshake circuit could be something like:

```
-- combine: Unbuffered input tupling procedure
procedure combine (input a : typeA; input b : typeB;
        output c : typeC) is
begin
  loop
    select a, b then -- Wait for a and b
      c <- typeC {a, b}
    end
  end
end
```

The handshake circuit implementation of *combine* is shown in [fig. 3.2]. Allowing select commands which involve no input selection allows the user to write descriptions which incorporate the enclosing-communication behaviour. In the *combine* example, no variables are required to receive the values on ports `a` and `b` and no sequencing of the {`a`, `b`} and `c` communications is necessary. The removal of this sequencing (at least one Sequence and

one Concur component) and variables associated with the input communications (also the potential for `select` specific optimisations) allows more area efficient implementation of such data driven descriptions.

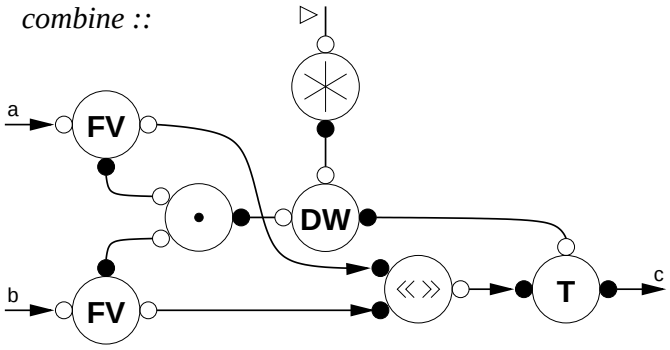


Figure 3.2. *combine* implementation in handshake components

3.2.2.3. Constructing Micropipeline-like pipelines

An abstract pipeline is shown in [fig. 3.3]. Input data on **inp** is steered through one of the two functional units (f_1 or f_2) based on the value of a single bit from the incoming bundle. The results from the functional units are caught by the multiplexer at the back of the pipeline and presented on the port **out**. The pipeline has no internal latches and so the multiplexer need not arbitrate between functional units as only one can be active at any given time. A possible Micropipeline implementation for this pipeline is shown in [fig. 3.4]. The treatment of the request and acknowledge as separate signals and the use of Select, Call and Merge elements makes this a typical example of a conditionally steered and joined pipeline in the Micropipeline style.

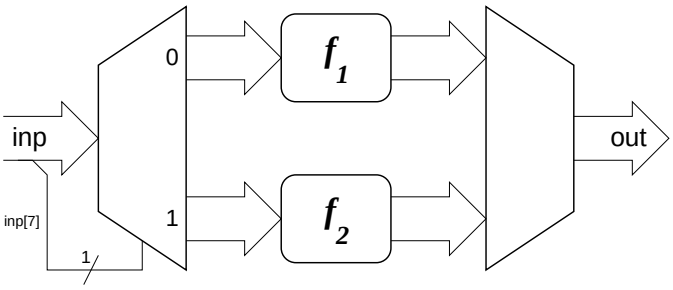


Figure 3.3. An example unbuffered pipeline

The handshake circuit for this example is shown in [fig. 3.5]. The *mux* procedure from §3.2.2.1

is used along with three other procedures each using the `select` command to produce modules with passive inputs (none of these three `select`s contains an input choice, however). The *dmx* demultiplexer procedure could be described like this:

```
-- dmx: Steering demultiplexer
procedure dmx (
  input I : byte;
  output O1, O2 : byte) is
begin
  loop
    select I then
      -- Use top bit of I to choose output
      if (I as array 7..0 of bit)[7]
      then O2 <- I
      else O1 <- I
      end
    end
  end
end
```

The body of the `select` command encloses the **I** handshake. The **I** value is steered towards the appropriate output by the output communications in the `if` command. The `if` command is implemented with a Case handshake component which closely resembles the Select Micropipeline component. After optimisation the *dmx* procedure is reduced to just this Case component with its output requests appropriately connected to the outgoing requests of **O1** and **O2**. The *mux* procedure is similarly reduced during optimisation to a CallMux component which resembles the Call / multiplexer combination shown in the right-hand dashed box of [fig. 3.4].

This shows that we can produce push orientated pipeline stages which have comparable implementations to manually generated designs using the Micropipeline component set. The option of including arbitrarily complicated operations within the `select` guarded commands (including non select-enclosed communications and control orientated descriptions) allows us to sympathetically describe solutions natural to both data driven and control driven design styles.

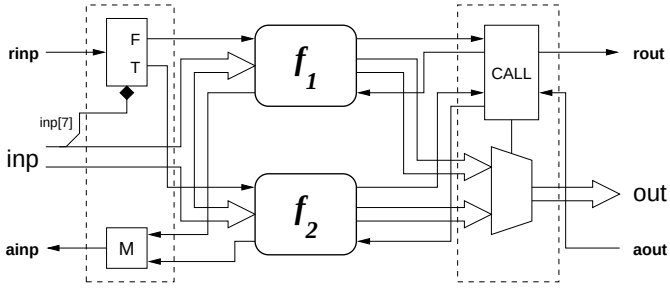


Figure 3.4. Micropipeline implementation of the example pipeline

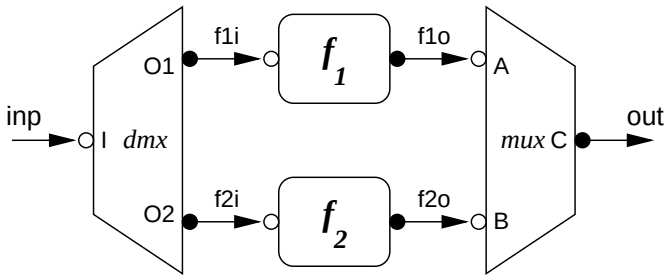


Figure 3.5. Handshake circuit implementation of the example pipeline

3.2.2.4. Tangram-like selection

The Tangram input selection scheme can also be constructed in Balsa using the Balsa `select` command:

```
-- t_select: Tangram-like selection
procedure t_select (input a,b : word) is
local variable c : bit
begin
  select
       a then c := 0
  also b then c := 1
  end ; -- release a/b, sequentially activate A/B
  case c of
       0 then A
  also 1 then B
  end
end
```

A gate level implementation of `t_select` involves the use of two 1 bit data latches: one in the multiplexer for the assignments `c := 0` and `c := 1` and the second for the variable `c`.

It is difficult to perform peep-hole optimisations at the HC level to improve on this circuit as the inefficiency (the presence of the second unnecessary latch) is associated with the ignorance of the stateful nature of the implementation of the multiplexer (CallMux) HC. The latch within the multiplexer holds a reference to the input port on which the most recent communication took place, this reference value drives the select inputs on the data multiplexing gates. As the inputs to the multiplexer HC are constant the latch contents could also be used to represent the relevant input constant. This is especially easy to implement in this case as the constants are 0 and 1 which match the latch output values 0 and 1 which drive a 2-1 multiplexer. With the multiplexer latch representing the constant the variable handshake component becomes unnecessary. Dual rail signalling may offer a simple solution to this optimisation problem. For a multiplexer with constant inputs the cost of the dual rail implementation is lower than that of the single rail equivalent. No data multiplexer steering latch is required and one of each of the two signalling wires and consequently half of the gates of each datapath bit can be optimised away.

3.2.3. Implementing selection

Selection is implemented with two new handshake components: The DecisionWait and the FalseVariable. The symbols for these two components (which are used in context in the implementations of the *mux* and *combine* examples in §3.2.2.1 and §3.2.2.2) are shown in [fig. 3.6].

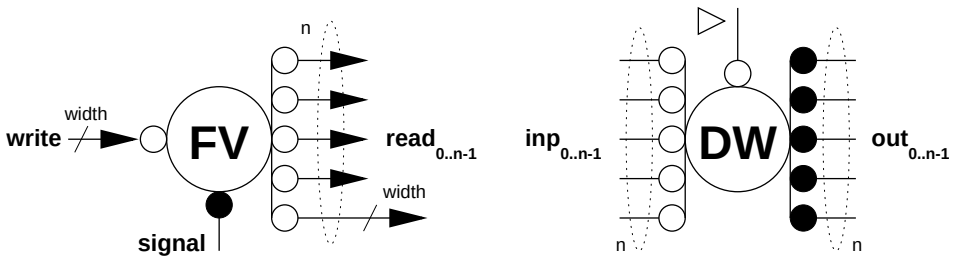


Figure 3.6. The FalseVariable and DecisionWait component symbols

3.2.3.1. The DecisionWait handshake component

The DecisionWait component has an activation port, *n* passive handshake ports and *n* active handshake ports. On receipt of the activation and a request on **one** of the passive ports, a

request is raised of the active port corresponding to that passive port. The acknowledge and synchronised return to zero on the active, passive and activation ports continues in sequence. DecisionWait has the behaviour:

$$\#[\ \triangleright\ :[inp_0 : out_0\ \mid$$
$$inp_1 : out_1\ \mid\ \ldots\ \mid$$
$$inp_{n-1} : out_{n-1}$$
$$]\ ]$$

A DecisionWait with a single passive *inp* and a single active *out* port is equivalent to a Synch component (a JOIN in the Tangram component set). With more than one passive/active port the DecisionWait performs an unarbitrated choice between the requests on one of the passive ports leading to a request on the corresponding active port.

The implementation of DecisionWait need not enforce mutual exclusion on its input *inp* ports. This mutual exclusion must be enforced by the environment. Also note that the expression $\triangleright : inp_m : out_m$ specifies the enclosure of a passive communication (inp_m) within another passive communication ($\triangleright$). Such an enclosure is subject to input reordering (where the order of a sequence of inputs cannot be distinguished because of the assumption of delay insensitivity) and so what may seem to suggest the sequencing of activation and *inp* handshake does, in fact, allow either of the behaviours: $\triangleright_r ; inp_{m_r}$ or $inp_m ; \triangleright_r$.

In implementation a DecisionWait consists of a one dimensional decision wait [32] and a merge. The decision wait restricts this component to at best a QDI implementation.

3.2.3.2. The FalseVariable handshake component

The FalseVariable resembles a normal Variable with one passive write port and a number of passive read ports, it differs however in the presence of an active 'probe' port: signal. The behaviour of the FalseVariable is:

$$\#[write : (signal \parallel command)]$$

Signal indicates arrival of a write, this signal should activate the circuit which has access to the read ports which will therefore only attempt reads during the period when data is valid

on write. This restriction breaks one of the cardinal rules in the construction of handshake components, that of port independence. This can be seen to be an acceptable sacrifice as the use of the FalseVariable by the compiler ensures the enclosure of all reads within the command connected to the signal port.

A communication on the signal port of the FalseVariable will ultimately form part of the activation of the guarded command for which the FalseVariable's write port communication is a guard. This communication on signal will then enclose the activation of the command. This enclosure ensures that the read ports of the FalseVariable are only accessed during the data valid period of the write port communication (as: $write : signal : \triangleright_{command}$ with accesses to the read ports only within the command body this implies $write : signal : \#?[read], \#?$ here loosely means an indeterminate number of repetitions).

This input-signal-encloses-action behaviour allows descriptions of Micropipeline like (or more generally push, latch free behaviour) circuits without explicit input latches to be written in Balsa for compilation into handshake circuits. The latch free implementation of the FalseVariable is a consequence of handshake enclosure. In those cases where the data validity periods of the write, read and signal may be overlapped purely by the interconnection of signalling wires, the FalseVariable's implementation is simply those wires.

3.2.3.3. Arbitration

The two way arbiter forms the third component used solely in the implementation of the arbitrated form of `select`: `arbitrate`. This component is just a conventional four phase arbiter with mutual exclusion guaranteed across the whole input handshake. Two way arbitration is achieved by use of the `arbitrate` … `end` construct in place of `select` … `end`. The arbiter is placed between the FalseVariables or nonput inputs of the guard channels and the passive ports of the two way DecisionWait. `arbitrate` could possibly be extended to allow more than two guards but as the implementation technology currently only includes two way arbiters a restriction to two guards was felt to be reasonable. Also for the majority of guard counts greater than two the arbitration tree which would have to be built would of its nature be unbalanced, it may be better, therefore, to force the designer to nest two arbitrations at the language level in order to achieve the desired arbiter balancing.

Multi-way (esp. 3 and 4 way) monolithic arbiters can be constructed [13] which have *n* stable states. There is little practical experience with such devices although in theory and simulation they appear promising.

3.2.3.4. Compiling `select`

The `select` command is compiled into a circuit fragment which has its activation connected to a DecisionWait component's activation. This DecisionWait serves as the mask preventing the selection from being fired whilst the command is inactive. Each of the passive inputs is connected to a Synch-joined nonput signal formed from the channels of a single guard. For input communications which carry data this nonput signal is the signal port output of a FalseVariable connected to that input channel. Each of the active outputs of the DW is connected to the activation of the command associated with the corresponding passive port guard input, the body of this command may access the value on each of its guard channels via the read ports on the respective FalseVariable.

The general compilation approach for Balsa `select` statements can be seen in the dual, two-channel-guarded example [fig. 3.7]. Here the channel sets {a, b} and {c, d} guard the two commands A and B respectively. On receipt of the requests for **all** the channels in one or other guard set, the appropriate command is activated.

```
select a,b then A
also   c,d then B
end
```

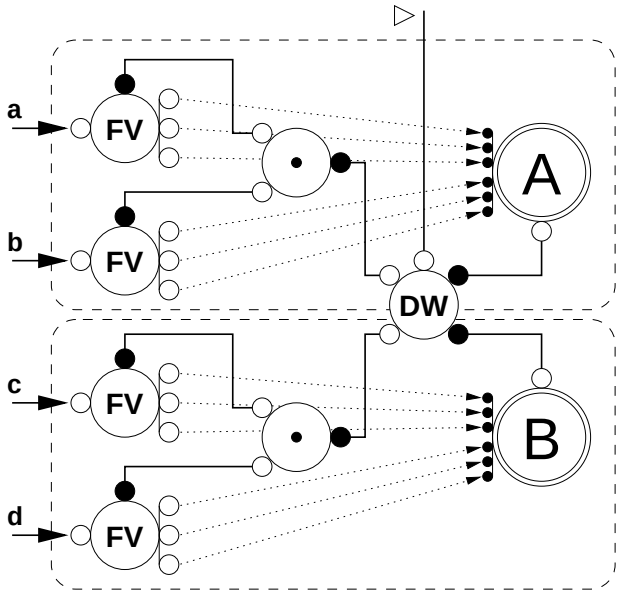


Figure 3.7. The form of a compiled **`select`** command

The FalseVariable components allow the incoming data on the guard channel to be read by the guarded command an indefinite number of times. This does not imply latching of the data but simply the enclosure of the guarded commands activation in the handshake on the write port of the component. This is achieved by using the active signal output of the FalseVariable to signal the validity of data on the write port. The channel connected to the signal port is combined with other signal channels and filtered by the DecisionWait to produce the guarded command's activation. In the same manner that the communications made by the guarded command are enclosed in that command's activation, the accesses made by the command to the FalseVariable's read ports are enclosed in the signal handshake and so within the write handshake. The implementation of the FalseVariable is described in §5.1.1.3.

The PushSynch components[1] after the FalseVariables combine the guard channel signals to produce a single activation signal for the guarded command. The compiler will disallow attempts to use guard channel sets which are not disjoint (e.g. {a, b}, {b, c} in [fig. 3.7]) as the operation of such selection would not be delay-insensitive under all input orderings. For the previous example: channel request orderings of b ; c or b ; a would imply the initiation of a handshake on the JOIN component combining the {a, b} signals and also the JOIN for the {b, c} guard set. Once initiated one or other of these handshakes could not be retracted without removing the relevant request and so breaking the signalling protocol on that channel. Enforcing disjoint guard sets is therefore a language-level requirement.

In the case where the `select` command does not require selective activation, most notably in the form `loop select … end end` where the select is indefinitely repeated and that repeater's activation is connected (by means of a direct or Fork'ed connection) to the global activation (ie. the circuit reset), the DecisionWait component can be omitted. This removes the Repeater, the DecisionWait and also allows FalseVariable / Fetch (transferrer) combinations to be optimised away (this is explained in §5.4). In this fashion the designer can make use of the push handshake components such as the CallMux (Push / multiplexing mixer) directly. This optimisation and a number of others possible for special cases of the `select` implementation are outlined in §5.4.

[1] these are JOIN components in the Tangram HC set

3.3. Chapter summary

The addition of communication enclosed input selection is the major contribution that Balsa has to make to the descriptive abilities of Tangram. An implementation has been detailed for such a selection mechanism along with descriptions of the new handshake components required to achieve this implementation. Gate level implementations of these components are given in §5.3.

Enclosed selection allows the description of data driven circuits without the need for input communications to terminate in a variable. The many ported, non synchronised reads possible during an enclosed input make this form of communication appear just like a variable read to the enclosed command. In this way the simple 'variable terminated' nature of Tangram expressions is preserved and if so desired the control paths of entire expressions can be removed and replaced by a matched delay. The FalseVariable component gets its name from this ability to mimic a variable. The DecisionWait is introduced to allow the re-inclusion of the activation channel so allowing `select` commands to be used in the same contexts as other Balsa commands.

The Balsa `case` command also differs substantially from its Tangram counterpart. A single guard may match more than one value of the guard expression. For example:

```
case a of 0,2 then A
also 1,3 then B
end
```

If `a` is a variable of type `2 bits` then this command will execute command `A` when `a` is either 0 or 2. The command `B` will be executed for guard expression values of 1 or 3. A similar Tangram command would look like this:

```
case a is 0 then A
or 1 then B
or 2 then A
or 3 then B
end
```

Here the commands A and B are each repeated implying that two copies of each will be placed during compilation. Even if A and B are simply calls to shared procedures the cost of a Call component is incurred for each call made. An implementation of this four way `case` command will therefore generate at least three Tangram CASE components and two Call components (Tangram MIX components).

The Balsa example given above is not so expensive. Only a single Case component and no Call components are required. The choice of shared procedure A and B is made in the decoding logic within the Case command. In this case the Case command's 'specification' parameter will code for an XOR gate. A gate level implementation for the Case command is given in §5.1.1.4.

Chapter 4. Compiling Balsa to Handshake Circuits

This chapter describes practical considerations in the implementation of a the Balsa compiler balsa-c. These include a description of some of the internal forms used within balsa-c during the production of a Breeze. Van Berkel's Tangram compilation function (as described in §2.4) is fleshed out to support the addition of passive input ports, composite typed data (records and arrays) and the use of shared procedures.

The place of the balsa-c compiler in a synthesis and simulation system and the tools used to produce balsa-c and associated tools are described

4.1. Balsa design flow

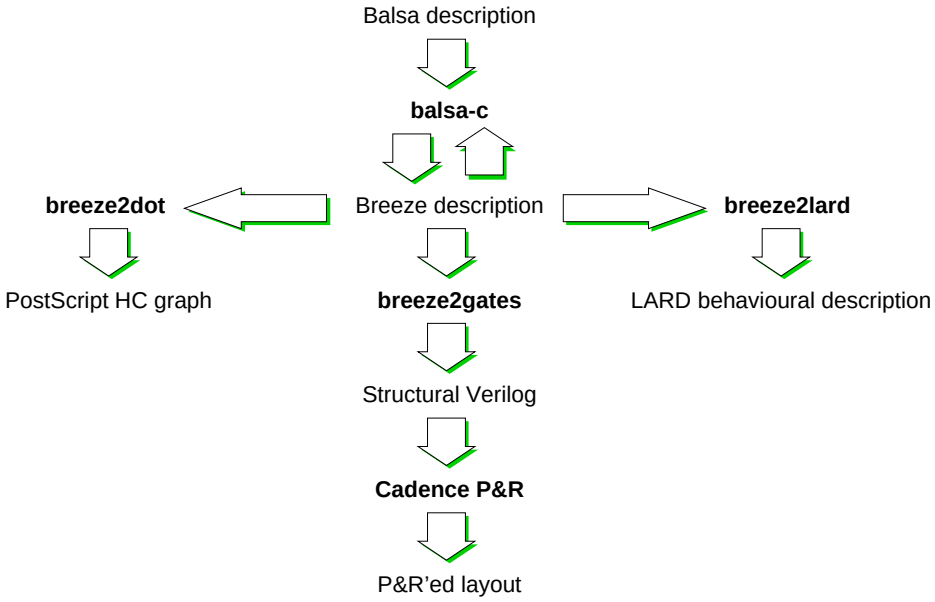


Figure 4.1. Balsa design flow

The Balsa system consists of a number of tools which fit together in the manner shown in [fig. 4.1] (names of tools are given in bold text). The compilation of a Balsa description into a gate level description is shown in the central column of the figure, the compilation consists of three steps:

1. **Balsa → Breeze compilation with balsa-c**

   The balsa-c compiler produces handshake circuits from the input Balsa descriptions. The circuits are expressed in the Breeze intermediate language (Breeze was introduced in §2.2 as a Balsa-syntax-compatible means of describing parameterisable handshake component port structures, the language is further explained in §5.2).

2. **Breeze → gate level netlist mapping with breeze2gates**

   The handshake circuit described in an input Breeze file is transformed into a gate level netlist (specifically a structural-Verilog netlist of standard cells). breeze2gates serves two functions:

   1. Producing specific implementations of parameterised handshake components from a set of production rules by breeze2gates. This involves interaction with the target CAD system to determine which of these components already exists in that CAD systems design database.

   2. Composition of a netlist of those parameterised cells expanding out the channel interfaces into individual request, acknowledge and data signals.

   The tool is implemented in Perl [30] in order to allow rapid development of new production rules for new handshake components. The implementation of the current (prototype) breeze2gates is outlined in §5.3.

3. **Gate level netlist → finished silicon with Cadence**

   The Cadence Design Framework II is used as the target CAD platform for the netlist produced by breeze2gates. Gate level simulation and final silicon production is carried out within Cadence.

Two other tools make up the remainder of the Balsa system: breeze2dot and breeze2lard. breeze2dot produces graphical printouts of Breeze handshake circuits and is useful in debugging handshake circuit production and optimisation whilst modifying balsa-c.

breeze2lard transforms Breeze netlists into the LARD modelling language to allow behavioural simulation using handshake component behavioural models expressed in a LARD plug-in library. breeze2lard is examined in more detail in §5.5.

4.2. The balsa-c compiler – internal structure and function

The balsa-c tool is a single pass compiler written in C using the Karlsruhe Compiler Toolkit [25] and the GNU Multi-precision Arithmetic library [37]. The lexical analyser and parser are automatically generated by the Karlsruhe tools *rex* and *lalr* from an input description in a similar manner to the UNIX tools *lex* and *yacc*. The toolkit was used in the production of balsa-c in preference to the more common UNIX tools because of its generated parsers greater speed, automatic parser error recovery and integrated tools for the production of attribute grammar evaluators. The GNU Multi-precision Arithmetic library (libgmp) is used within the compiler to support the arbitrary precision constants (and the constant folding optimisation) required by Balsa.

The compiler takes valid Balsa descriptions as input, generates an internal parse tree and performs a single directed walk of that tree to generate handshake circuits directly by performing an extended form of the Tangram compilation function.

The Karlsruhe abstract syntax tree generating tool *ast* [27] and the attribute grammar evaluator generating tool *ag* [26] are used to produce an ordered attribute grammar evaluator for Balsa which mirrors the structure of the compilation function. The compilation of a ⟨command⟩ is a good example of the use of this evaluator. The attribute grammar evaluation for each command combines the sub-commands' (and expressions') attributes (e.g. the sequential command A ; B combines the attributes for A and B) to produce attributes which are passed back up the parse tree. Some of the attributes relevant to command (and also expression) compilation are:

`context`

The (ordered) package of bound identifiers at this point in the description. The context is passed down the parse tree towards the primitive commands / expressions at the leaves. The context includes variable, internal channel, external port, constant, procedure and type declaration information.

`attributes`

The package of attributes passed back up the parse tree to be combined in enclosing (closer to the tree root) nodes. The attributes contain:

`accesses`

The list of access patterns on variables and channels which exist at this point in the program. Channel accesses for commands are combined whilst descending the parse tree, variable accesses are pooled until the top of the enclosing command / expression block is met on the journey back up the tree. In the terminology of the Tangram compilation function the accesses are the alphabet structure for a given expression or command. Each access contains a reference to a `Wire` (the internal representation of a connecting handshake channel) which acts as a read / write port onto that resource.

`components`

A list of handshake components to be placed for this command / expression.

`wires`

The combined list of all `Wire`s used by components and accesses. Each wire in this list will have exactly two references made to it in one of the combinations: 1 reference in a component and 1 in an access or 2 references in a component or 1 reference in a component and 1 in the activation. Where two references within the component list are made, this `Wire` is internal to the command.

`value` (expressions only)

The Type and numeric value (iff this expression is manifestly constant) of the expression. A Type referenced here must be an element of the current `context`.

`activation` (commands only)

A reference to the activation `Wire` for this command.

`permanent` (commands only)

A boolean, true iff the command never returns an acknowledgement on its activation. Knowledge of the permanence of commands allows selective optimisation of the components used to implement activation combining operations.

The compilation of a command transforms `context` and command attributes into a return `attributes` set. The same transformation occurs for expressions; these are combined with commands (such as output communication or assignment commands) to form a single return command `attributes` set. The compilation of a procedure is broken into five operations (which are shown in their relevant abstract syntactic positions in [fig. 4.2]):

1. Gathering of port attributes into an incoming `context` (which is combined with the `context` from the scope level above the procedure).
2. Processing of local declarations. These combine with the `context` created by processing the ports to form an input `context` for the procedure command.
3. Processing of the local command, producing a resultant return `attributes` set.
4. Creation of components to connect `Wire`s local to the command to the variables / channels in the local declarations.
5. Connection of dangling port Wires to port instances, checking and creation of the Procedure object for insertion into the subsequent declaration's incoming `context`.

This sequence of operations forms an ordered evaluation mechanism for the attribute grammar which models the operation of the Tangram compilation function in an *imperative* fashion (as opposed to the declarative form in which the mechanism is presented [28]).

4.3. Extensions to the Tangram compilation function

The compilation function implemented in balsa-c is based on the Tangram compilation function. Additional attributes (notably typing) have been added to support those features

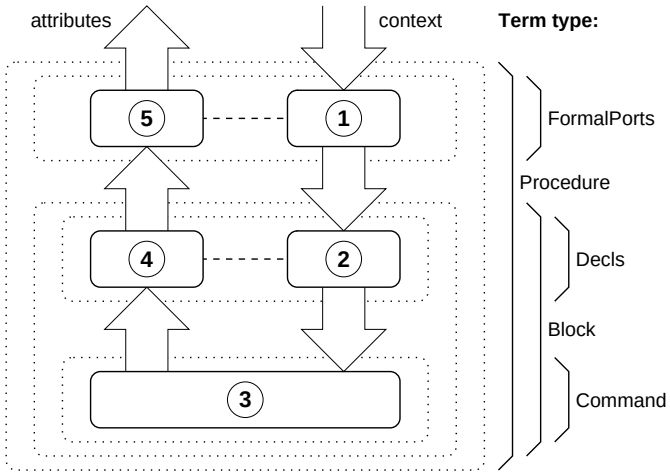


Figure 4.2. Procedure compilation in the balsa-c attribute grammar evaluator

present in Balsa which are not present in Tangram as described in §2.3.

The compilation of the basic commands (input, output, assignment …) are basically the same as Tangram but with the addition of Balsa specific type information. Balsa has additional types to Tangram and also supports no implicit type-casts. For an output command `a <- b`, channel `a` must have the same type as the expression `b` for this command to be valid.

The iterative and choice commands of Tangram are all present in Balsa albeit with different names and syntaxes ([tab. 3.1] compares the common commands of the two languages. The syntax of Balsa is influenced most by Modula and VHDL and as such the names and syntaxes of Balsa commands tend to resemble similar constructs in those languages). Balsa adds to these: a more general `case` statement, the `select` input selection mechanism and the `for` multiple-placement iterative directive. In the compilation of ⟨expression⟩s, Balsa adds shift operations and vector types which allow bit slicing / extraction and vector construction in a similar manner to record construction. For channel operations, vectored channels allow numerically indexed arrays of independent channels to be exploited. Procedures may be shared between sequential threads where the handshake components for that procedure are placed once but the procedure is called from several points in the program. Combination of the many activations produced by shared use of procedures is simple but some rules on the use of local channels must be enforced to restrict sharing to ‘safe’ cases.

The compilation mechanisms for these new behaviours are outlined in the following sections:

4.3.1. Composite typing – variable placement, arrays and record type compilation

Each access to a variable made within a command or expression body is recorded in the `accesses` attribute of the `attributes` packages generated by that command or expression. The `accesses` attribute consists of a list of cells referring to variables, channels and ports which are in scope for this command or expression. These instances[1] must be present in the `context` of this command. Each element of the `accesses` list refers to a single instance and has a different format for each instance type.

4.3.1.1. Variable accesses

Variable accesses name the variable involved, the direction (read / pull or write / push) of each access to that variable, the `Wire` connected to the variable in each access, the width of the accessing `Wire`[2] and the offset within the variable of the zeroth bit of the `Wire` for each access (the width and offset attributes actually belong to the `Wire` rather than the access). This is coded as a four element structure containing: a list of read accesses and a list of write accesses, a bit mask marking those bits of the variable which are referred to by each of the read and write access lists. The read and write bit masks are used to determine whether the entire width of this variable has been covered by the accesses in this `accesses` list element.

4.3.1.2. Channel accesses

A channel access contains the `Wire` to which the access to this channel is connected. The `Wire` contains the push / pull indication which tells whether this is an input or output communication on that channel. Each channel access only contains a number of flags for indicating where a channel is involved in a select construct, whether the channel is *complete*

[1]The type `Instance` is used in the code to refer to this class of objects (plus a few other cases), the term *instance* will be loaded with this meaning for the remainder of this chapter

[2]`Wire`s were previously defined in §4.2 as objects used in the balsa-c attribute grammar. Each `Wire` encodes the details of a Breeze channel connecting a pair of components in the target handshake circuit.

(i.e. has exactly one source and at least one sink), the read-only variable instance to which the channel is connected during a select operation and so on.

4.3.1.3. Accesses to other types of instances

Other instances include: ports (which are treated in a similar manner to channels), shared call instances (which are tokens which identify the activation ports of local shared procedures) and constant instances (these never appear in `accesses` lists, only in `contexts`.)

4.3.1.4. Compiling partial width accesses to variables

The `accesses` list is sorted by the pointer value of the instance referred to (which makes a good instance wise unique key). The combination of `accesses` lists of sub-commands whilst compiling a composite command (e.g. sequence command: A ; B, concurrent command A || B) is the implementation of the Tangram compilation mechanism Mix and Join functions.

The combination of channels under the Mix and Join operators is similar to the Tangram compilation function except for the typing of channels and the handling of passive input (`select`ed) channels. Variable accesses, however, are not combined until the `attributes` have reached the point of variable declaration in their trip back up the parse tree. Variable accesses are accrued within the `accesses` list elements and are processed in batch when the Variable component is placed.

Access to a variable need not necessarily involve the whole width of that variable. Consider the code fragment:

```
type dual is record a, b : 16 bits end
variable c, d : dual
…
c.a := d.b
```

Here the `b` element of `d` is assigned into the `a` element of `c`. Both elements match in type (`16 bits`) and so the assignment is type conformant although only half of each of `c` and

d is actually accessed. An access[1] to the variable c for this command would record a 16 bit wide push wire at offset 0 connected to variable c with a write access bit mask of $0000FFFF_{16}$. The d access would refer to a 16 bit wide pull wire at offset 16 with a read bit mask of $FFFF0000_{16}$.

On Mix or Join combination, the bit masks of the accesses to common variables in the accesses lists of the two commands involved would be bit-ORed together. Thus, for a composite command for either of the variables c or d in the above example, a complete set of accesses (where all bits of the variable are both written and read) would have a (read, write) bit mask pair of ($FFFFFFFF_{16}$, $FFFFFFFF_{16}$).

When Variable components are placed for each variable in the local declarations of a ⟨block⟩, the write-access list within the access for that variable is examined in order to identify the largest bit slices of that variable for which only atomic writes are made (e.g. for a 9 bit variable which is written as two 6 bit chunks offset at 0 and 3 bits respectively, we can see that the largest atomically written bit slices are 0..2, 3..5 and 6..8 as the two write accesses overlap by three bits). These write accesses are connected (by a series of CallMuxs where multiple writes to the same slice occur) to individual Variable components. The read accesses to those components are connected to the (possibly many) read ports of the Variables with the required splits and combines to produce the required width of offset wires.

Using this simple mechanism for handling partial variable reads and writes we can implement the following features:

- Partial assignment, individual array or record element assignment (with a constant index in the case of array element assignment).
- Constant indexed array element / slice extraction expressions (e.g. `ArrayVariable[10]` and `ArrayVariable[5..10]`)
- Record element extraction expressions (e.g. `c.a`)
- Shift operations on values read directly from variables.

[1] `accesses` elements are typed as `Access`es in the code, *access* here denoted this type of object.

4.3.1.5. Compiling array accesses with non-constant indices

We cannot, however, implement array accesses (assignments or element extractions) with non-constant indices with this slice extraction approach as we need to be able to make a choice of array element based on the index expression. In the case of array element assignment an assignment is of the form:

```
A[B] := C
```

where `A` is an array typed variable, `B` is a non-constant expression and `C` is an expression. This command can always be rewritten in the form of a `case` command ranging over the index type of `A`. If the index type of `A` (and therefore the type of `B` in the above command) is `2 bits` then we have the possible implementation:

```
case B of
     0 then A[0] := C
also 1 then A[1] := C
also 2 then A[2] := C
also 3 then A[3] := C
end
```

This expression does, however, contain the right hand expression, `C`, four times. Given that `C` will be evaluated from exactly one of the `case` terms in this command we can place this expression just once and use a CallDemux (pull demultiplexer) to expand out the result to each of the four assignment commands. Balsa-c implements array element assignment in just this manner. A given implementation will consist of one Case component, *n* Fetch (transferrer) components (one per assignment), an *n* way CallDemux and a single placement of the assigned expression. In explicit Balsa this can be thought of as being similar to either of the commands:

```
-- Array element assignment without select
local channel _C : typeOfC
```

```
begin
  _C <- C ||
  case B of
       0 then _C -> A[0]
  also 1 then _C -> A[1]
  also 2 then _C -> A[2]
  also 3 then _C -> A[3]
  end
end

-- Array element assignment with select
local channel _C : typeOfC
begin
  _C <- C ||
  select _C then
    case B of
         0 then A[0] := _C
    also 1 then A[1] := _C
    also 2 then A[2] := _C
    also 3 then A[3] := _C
  end
end
```

Neither of these two commands exactly models what is produced by balsa-c (in that neither will immediately result in the same circuit) but both are reasonable functional equivalents.

Array element extraction is more difficult, however. In order to extract an array element under the pull control scheme used by Balsa expressions, we must sequence the evaluation of the index expression and the fetching of the appropriate array element. The pull nature of the activation / result channel on an expression limits the ways in which we can implement an expression borne extraction. If a command can be substituted for the expression (say by treating the expression and the command of which it is a part as a composite command), one could use a similar form of assignment as used in the above commands (except that the constant indexed array element extractions would form the right hand values of the `case` element assignments). The simplest solution to this problem was the introduction of a special component for this operation. The CaseFetch (indicated by the characters `@T` in the handshake component symbol) component is hereby introduced. CaseFetch has a single pull activation / result output channel on which the chosen element is presented, a single pull input channel from which the index expression result is read and *n* pull input channels each of which is connected to an access wire to the appropriate element of the array. The behaviour

of CaseFetch is:

$$\#[\triangleright \uparrow : (index?I; read[I]); \triangleright \downarrow]$$

Where the *read*[*I*] expression is a single read on one of the array element pull input channels using the value read from *index* to select (which in an implementation derived from this behaviour must have a latch, for variable `I`, to hold the index value) the appropriate port.

4.3.2. Procedure sharing – activation and access handling

If we consider a simple microprocessor description:

```
-- processor: A simple fetch then execute processor
procedure processor is
begin
  loop
    fetch();
    case instructionType of
         ALU then executeALU(); incPC()
    also LOAD, STORE then executeMem(); incPC()
    also BRANCH then executeBranch()
    end
  end
end
```

The instruction is fetched (`fetch()`), decoded (the `case`), then executed (in the body of the `case` elements). The procedure `incPC` is invoked twice in the description (once for each of the ALU and LOAD / STORE operations), this will lead to an implementation with the handshake components making up `incPC` being placed twice even though the calls to `incPC` are mutually exclusive (by virtue of the sequential mixing of `case` element commands). If a single instance of `incPC` is placed to which two activations are connected by the inclusion of a Call component in the activation channel, we can call `incPC` from multiple, sequential, points in the program and save the area cost of one of the `incPC` procedures. This can be achieved in Balsa by use of a shared procedure, e.g.:

```
local shared incPC is begin … end
begin
```

```
    incPC();
    …
    incPC()
end
```

This would declare `incPC` as such a shared procedure. Just combining the activations sourced from several points in the program is fairly straightforward. A SharedCall instance is added to the accesses list for a shared procedure call command, this is then resolved at the point of declaration of the shared procedure on the journey back up the parse tree. The local channel and variable accesses made by the shared procedure itself present us with much more of a problem. Consider this case:

```
local
  channel a : word
  variable b : word
  shared sh is begin a -> b end
begin
  a <- 5 || sh() ;
  a <- 10 || sh()
end
```

Here the shared procedure `sh` is called from two points which are sequential (as required) yet the meaning of the channel `a` is different in each calling context. Local channels can be used in multiple sequential points within their scope as effectively different channels. The above example is a case in point, to correctly place `sh` we must be capable of associating the use of `a` within `sh` for each calling context for `sh`. In general this is difficult to achieve and so the use of shared procedures is restricted by the following rules:

- Shared procedures must not use local channels in their command. This restricts the calling context to including only variables (the context of whose accesses are invariant across their scope) and external channels (i.e. ports, which are similarly constrained).
- Shared procedures have no arguments. This removes the need to connect shared procedures by local channels to their environments.

It is believed that these rules do not impose a serious restriction on the usefulness of shared

procedures but do allow us to statically resolve access contexts. The mixing of accesses made by shared procedure calls can now be achieved by tagging a copy of the procedure's accesses list with an 'unplaceable' flag. This flag causes all the required checks to be made on channel / variable access compliance but does not result in components being placed to combine accesses with real, placeable accesses. The access connections for a shared procedure are made in a similar fashion to the accesses on variables, that is to say, at the point of declaration (which now shares a common (in fact a restricted) context with the block command) whilst working back up the parse tree.

Whilst Tangram incorporates the idea of shared procedures it is not clear from the literature what usage rules apply to prevent local channel references to 'shift context'. Van Berkel does not describe the treatment which shared procedures receive in his compilation scheme [28].

4.3.3. The choice commands – `select` and `case`

The `select` command and its handshake component implementation was examined in §3.2. Two new handshake components were introduced (the FalseVariable and the DecisionWait). Here a fourth new 'choice' component is introduced (the third being the CaseFetch) thus completing the set of new Balsa components. The new component is the Case component, this differs from the Tangram CASE component which performs a choice between two output activation channels on the basis of an input value (very much in the manner of the Micropipeline Select component). The Balsa Case, however, has a specification string parameter and any number of output activation channels. The specification string allows us to specify a set of ranges across the input value (the port structure of Case is similar to the Tangram CASE excepting the number of output activation channels) for which a particular output activation should be invoked. In this way we can describe `case` commands which have a number of guard constants (the `LOAD, STORE` example in the previous section is a good example) because we can describe many different input patterns being associated with a single output activation (which will be the `case` guarded command activation).

The Case component is used to implement both the `if` and `case` commands in Balsa. Where the guard expression(s) are not accesswise disjoint from the guarded commands, a buffer assembly is introduced (in the same way that a buffer is introduced for an auto-assignment

e.g. `a := a or b`) to decouple the guard value evaluation from the command invocation.

4.4. Tools other than balsa-c – breeze2dot, breeze2lard, breeze2gates

A detailed description of the structure and function of breeze2gates is given in §5.3. This tool is used to produce Verilog netlists from the given Breeze handshake component netlists.

The remaining two tools are used in simulation and evaluation of Breeze netlists. Breeze2dot allows us to produce (reasonably readable) handshake circuit printouts. The tool *dot* [5][6] is used as the basis of breeze2dot as a graph layout engine. The Breeze input file is preprocessed into dot format by a Perl script and then piped through a modified version of dot (which places the open and closed circuit port symbols on the graph) producing a postscript output.

Breeze2lard is used to produce LARD descriptions of Balsa circuits. The use and structure of breeze2lard is discussed in §5.5.

4.5. Chapter summary

The tools and methods for implementing balsa-c have been described. The method involves an extended form of the Tangram compilation function to support the additions made to Tangram by Balsa. Of particular note is the simple manner in which assignment to and reads from variables of composite types are expressed by the elements of the `accesses` attribute grammar attribute.

The Mix and Join operators of the Tangram compilation function are defined to combine variable reads and writes which span arbitrary bitfields of local variables. In this way record element extraction, array indexing and arbitrary type casts can be accommodated.

This approach does have its disadvantages. Consider the expression:

```
{c, d, e, f}[b]
```

Here an array is constructed from the contents of the variables `c`, `d`, `e` and `f`. This array is then indexed by the contents of variable `b`. If the array had been constructed from parts of the same variable then the indexing could be implemented by slicing that variable in the way described in this chapter for the treatment of array and record element extraction. This example cannot be treated in the same simple manner as the array spans multiple variables. To perform the element extraction the array typed value composed of the four variables must be made (by combining variable read ports) and then sliced into element sized chunks. This is necessary as the left hand side of the array indexing expression may be a cast expression, for example:

```
({a, g} as array 8 of bit)[b]
```

In the same way as arbitrary casts are allowed for expressions which involve single variables, array construction expressions (with possibly many variables) may be cast into another type. Extending the array element extraction mechanism to allow one-to-one mappings of variable reads to indices would be sufficient to handle the four variable example given above. This extension would not work where the left hand (array typed) expression may be the result of a type cast. Using a one-to-one mapping of variables of the previous four variable example to optimise for an array construction left hand side will not work with a cast left hand side.

As a result of the complexity of multi-variable left hand sides, Balsa restricts the use of array element extraction to constant and single variable left hand sides.

Chapter 5. Handshake Component Implementations and Simulation

This chapter will discuss the process of transforming a handshake circuit description (in Breeze) into standard-cell logic along with the associated problems of speed / area optimisation and simulation. Many of the topics discussed here are not currently features of the Balsa system and in part this chapter serves as a view of future directions (the threads of which are brought together in §7.4) and justification of approach. The chapter breaks down into:

1. Gate level implementations of handshake components. Implementations of the new components FalseVariable, DecisionWait, Case and CaseFetch are given along with commentary on the specification and implementation of other, slightly modified, components.

2. The prototype 'back-end', breeze2gates. Written in Perl, this script is the current basis for the test back-end of Balsa. The breeze2gates program serves as both a handshake circuit netlist creator (netlist in the CAD native format that is) and a parameterising engine for generating handshake component implementations.

3. Optimisations at handshake circuit and gate levels, concentrating on optimisations of the `select` command.

4. Simulation with LARD. LARD is an asynchronous circuit modelling language developed within the AMULET research group to fulfill a need for a behavioural modelling platform to replace the proprietary tool *asim* (which was used in the development of AMULET2). LARD has been successfully employed in the modelling of AMULET3. Here it serves as a mechanism for performing handshake circuit level simulations of Balsa synthesised designs, transformed into LARD by the tool breeze2lard.

5.1. Implementations of handshake components

The target technology for handshake component implementation in this thesis will be the gate level netlist. The set of gates available includes:

- Conventional combinational logic gates: AND, OR, invert, XOR and also complex combinational gates such as AND-OR-inverts.
- Conventional latch and flip-flop elements: transparent latches, edge triggered latches, RS flip-flops.
- Specific 'asynchronous' gates: symmetric / asymmetric C-elements, mutual exclusion elements.
- Composites, common gate combinations: S-elements, multiplexers, demultiplexers. These may be either transistor level, single cells or combinations of gates.

A gate level netlist can be submitted to a commercial CAD system for generation of the final (placed and routed) design. Here the Cadence CAD framework is used to produce standard cell implementations using a 1μm cell library developed as part of the EXACT project. This route was that employed by the work of EXACT at The University of Manchester in the single-rail handshake component Hybrid Design Environment [8]. In the placement of cells for single-rail handshake components, timing relationships between signals were not considered as part of the hybrid back-end. Gate delays ensured correct behaviour of control circuits (which could, therefore, be considered to be of a *speed-independent* (SI) nature) with data bundling depending on the insertion of delays in the control path (or datapath generated completion signalling) to meet control – data timing constraints. This approach is not altogether satisfactory; a solution involving the explicit checking of back-annotated post-layout timing values against bundling and fork constraints would be more robust. After back annotation of timing estimates gained from extraction and simulation steps, some manual reworking of a placed circuit may be necessary to ensure correct behaviour. The implementations given in the remainder of this section follow the model used by the hybrid environment.

5.1.1. Control components

The Balsa component set contains 12 control components, those borrowed from Tangram shown in [tab. 2.4] and the FalseVariable and DecisionWait components new to Balsa whose use in the compilation of `select` is explained in §3.2. Of these, three components are identical to their Tangram equivalents: Loop (which is a Tangram REP component, the repeater) , Repeat (which activates the circuit connected to its active nonput port a fixed number of times, specified by one of the component's parameters, for each activation of the component, Repeat is the Tangram component COUNT used in the implementation of Tangram `for do … od` loops) and Fetch (TFR, the transferrer). The remaining components differ in the following ways:

5.1.1.1. Multiway components – Concur and Sequence

Concur and Sequence are the Tangram components SEQ (the sequencer) and PAR (the concursor). Within the Tangram component set these components have a single passive nonput activation port and two active nonput (outgoing activation) ports implementing two way concursing and sequencing. The extension of this two way behaviour to create n way (where $n \geq 2$) sequencers and concursors can be implemented either by the combination of small (2 or even 3 or 4 way) components or by synthesis of a custom controller for the required number of output activations. In combining smaller components balanced trees can be built to replace the unbalanced binary trees created by simplistic compilation, this and the advantage of the improved clarity of the Breeze description of a given control structure make multiway components advantageous.

The synthesis of specific controllers for n way control components has been previously explored [2]. The combination of sequencing and concursing into a single control tree (which may also include other choice-less control such as that provided by Loop and Repeat components) can also be synthesised to produce a single monolithic controller [31] [38]. The area / speed advantages of such a (signal level) synthesised solution should be contrasted with the increased timing difficulties introduced by the (typically speed-independent) gate level implementations sourced by such compilers (especially where the internal handshakes, present in a compositional handshake component control tree implementation, are replaced

by control sequencing). Within a compositional approach, speed-independent behaviours (and also QDI, bundling constraint and more time dependent timing model based behaviours) can be localised by the hand design and analysis of handshake component implementations. The potential to control the degree of gate level optimisation is also an advantage of a compositional approach over a synthesis approach. A simplistic approach to gate level optimisation can destroy timing constraints required of synthesis approaches.

For the purposes of the prototype back-end, these multiway components are implemented by balanced trees of S-elements for sequencing and Fork and S-element'ed connectors (early enclosing wide protocol adapter) for concursors.

5.1.1.2. The `while` command – While, WhileElse and Bar components

While and WhileElse are the top level control components associated with the `while` command. They are similar to the Tangram DO component and have identical port structures except for the addition of a passive nonput port in the case of WhileElse, for the connection of an 'else' term command. The While component gathers a guard value from an active input port connected to either a guard command or expanded by the Bar component, executes the guarded command by signalling on an active nonput port activateOut if the guard value was 1 (true). This behaviour is enclosed by an activation handshake and will repeat, within the activation handshake, for as long as the guard holds (remains one). On the failure of the guard the While component will return an acknowledgement on the component's activation, the WhileElse component will execute the command connected to its elseActivate port then repeat (never returning an acknowledgement). Symbolically:

$$\begin{aligned}
&\mathit{While}(\triangleright, \mathit{guard}, \mathit{activateOut}).\\
&\quad \#[\,\triangleright\ : \mathit{guard}?G; \#[G \rightarrow \mathit{activateOut}; \mathit{guard}?G\ \mid\ \neg G \rightarrow \mathit{skip}]\,]^{1}\\
&\mathit{WhileElse}(\triangleright, \mathit{guard}, \mathit{thenActivate}, \mathit{elseActivate}).\\
&\quad \triangleright\ : \#[\mathit{guard}?G; [G \rightarrow \mathit{thenActivate}\ \mid\ \neg G \rightarrow \mathit{elseActivate}]]
\end{aligned}$$

The behaviour of the WhileElse component differs from the While (or DO) component in that the activation acknowledgement is never sent, WhileElse does not terminate and so no outer

[1] G in the While behaviour refers to a variable which is assigned from the communication $\mathit{guard}?G$.

unbounded repetition loop is required.

The Bar component is similar to the Tangram BAR component. It serves to expand the single guard capability of While and WhileElse to a number of guard / command choices. The Tangram two way BAR command must be combined into a tree to implement `do` loops of two or more guards. The Balsa components can combine all guards in a single component so reducing or at worst keeping the same area overhead of the BAR tree but with less reliance on gate level optimisation. This is achieved by localising the guard-choice priority encoder into a single component allowing three and four input gates (and regular structures) to be better exploited. The non-determinism of the choice of command to execute where multiple guards are found to be true is purely an artifact of the formal semantics of guarded choice, in implementation the choice is made in a prioritised way, one of the two commands connected to the BAR being favoured where both guards become true.

The port structure of Bar takes *n* active 1 bit guard inputs (each associated with an active nonput command activation) and maps to a single passive guard output and a passive nonput, incoming command activation. The behaviour of the component is similar to the Tangram part. The *n* guards are gathered into a single guard by a large OR tree, that guard is returned to the environment through the single passive guard port. The environment will choose whether or not to activate the associated command – if it chooses to activate the command, the Bar component will receive a nonput communication on $\triangleright$. This communication leads the Bar component to activate one of the commands connected to a port within the nonput arrayed port *activateOut* for which the respective guard was previously found to be true.

The Bar functionality could easily be combined into the While and WhileElse components. This is not done as the majority of `while` loops have only a single guard and so the added complexity of a multi-guarded While would rarely be used. The multiway Bar component also finds applications with the Case component, which is used to implement the `if` command in Balsa. Providing Bar functionality within While would also require this functionality to be added to an If component. As no If component is necessary with separate Case and Bar components this guard tree functionality is best kept separate. Bar's behaviour is:

$$Bar(\triangleright, guard, activateOut_{1..n}, guardInput_{1..n}).$$

$$
\begin{array}{l}
\#[\ [\\
\quad \triangleright : [G = 0 \rightarrow activateOut_1 \mid \ldots \mid G = n-1 \rightarrow activateOut_n] \mid \\
\quad guard : [[guardInput_1 : \ldots : guardInput_n : \\
\qquad [G := choose(guardInput_{1..n}) \parallel \\
\qquad guard := \vee\{guardInput_1, \ldots, guardInput_n\}]]] \\
]\ ]
\end{array}
$$

Guard collection and the execution of a guarded command are sequenced by the arbitrating action of the communication choice [$\triangleright$ | *guard*]. The function *choose* deciding which of the commands should be executed in the event of the activation being received. This *guard* ; *command* sequencing is not sufficient to cope with the Case implemented `if` statement, however. Consider [fig. 5.1], a pair of guards G_1 and G_2 which are compatible in their accesses to commands C_1 and C_2 are combined by a Bar component and transferrer onto the input port of a Case component. The Case component will choose which of the two commands to execute based on that guard (where the guard is 0, the Continue component is activated and so the `if` command terminates, returning the acknowledgement through the Fetch component).

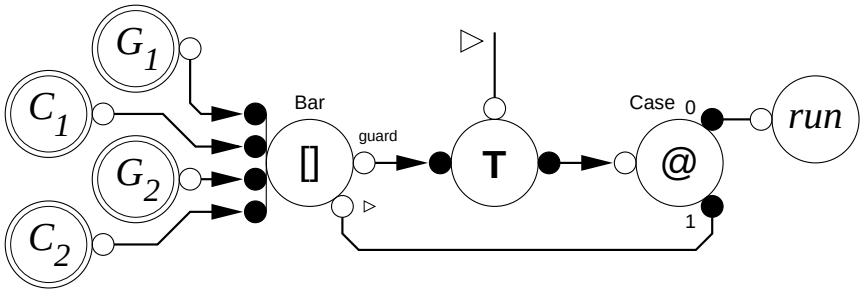


Figure 5.1. An `if` command implemented using Case

The input to a Case component must enclose its active nonput communications (in this case the commands) and so it can be seen that the guard communication of the Bar component will not be complete before the possible arrival of a communication on the activate port. Guard expression communication and command activation must be allowed to overlap. The mutual exclusion of guard gathering and command activation formed by the communication choice can be seen to act in the manner:

$$\#[guard; \triangleright\ ]$$

if it is assumed that the guard **is** true and so activation **is** signalled. The parallelism of this expression can be increased by either making the two communications fully parallel or by allowing one to partially overlap the other.

Parallel actions: $\#[guard \parallel \triangleright]$

Partial overlapping: $\#[guard\uparrow ; [\triangleright \parallel guard\downarrow]]$

Both of these behaviours still allow the complete environmental sequencing of guard and activation. The partial overlapping of guard and activation has the advantage that the guard choice variable (G) is guaranteed to have been written (as part of the action which the communication $guard\uparrow$ here represents) before the activation can be acted upon. In practice the environment will always enforce this constraint itself (as the guard value must reach the environment before a choice as to whether the command is activated is made) and so the fully parallel description shows similar merit. When combined in the complete description, the partial overlapping behaviour is difficult to express as the communication guard for activation must run in parallel with the guard return to zero. This could be expressed:

$$
\begin{array}{l}
Bar(\triangleright, guard, activateOut_{1..n}, guardInput_{1..n}). \\
\quad \#[\,[\\
\qquad [[\triangleright : [G = 0 \rightarrow activateOut_1 \mid \ldots \mid G = n-1 \rightarrow activateOut_n]] \\
\qquad\quad ; [guardInput\downarrow : guardInput_1\downarrow : \ldots : guardInput_n\downarrow\,] \mid \\
\qquad guardInput\uparrow : [[guardInput_1\uparrow : \ldots : guardInput_n\uparrow\,]; \\
\qquad\quad [G := choose(guardInput_{1..n}) \parallel \\
\qquad\quad guardInput := \vee\{guardInput_1, \ldots, guardInput_n\}]] \\
\quad]\,]
\end{array}
$$

The fully parallel description can be expressed as two separate threads communicating via a shared variable G. Disjunction of read and write accesses on G are enforced by the environmental sequencing of $guardInput\uparrow ; \triangleright$.

$$
\begin{array}{l}
Bar(\triangleright, guardInput, activateOut_{1..n}, guardInput_{1..n}). \\
\quad \#[\triangleright : [G = 0 \rightarrow activateOut_1 \mid \ldots \mid G = n-1 \rightarrow activateOut_n]] \parallel \\
\quad \#[guardInput : [[guardInput_1 : \ldots : guardInput_n]; [G := choose(guardInput_{1..n}) \parallel
\end{array}
$$

$guardInput := \vee\{guardInput_1, \ldots, guardInput_n\}]]]$

5.1.1.3. New components – FalseVariable and DecisionWait

The biggest differences between control structures from Tangram to Balsa are the addition of the enclosing `select` and the extended `case` commands. The uses and handshake circuit implementations of both commands were outlined in §3.2 and §4.3.3 introducing two new components for `select` implementation (FalseVariable and DecisionWait) and a modified Case component for implementing Balsa `case` commands.

The one-dimensional decision wait component which forms the heart of the handshake component DecisionWait is that described by Josephs [32]. Decision wait acts as an event filter connecting *n* input signals to *n* output signals by way of a rank of two input C-elements each with one input connected to an input signal and the other connected to a common activation signal. A two way (or 2×1 way where the component is considered to be two-dimensional) decision wait is shown in [fig. 5.2] with a symbol and gate level implementation. The forking of the common activation symbol should be isochronic in order to allow each of the inputs of the C-elements to be 'cancelled' when the activation is removed. On assertion of one of the *n* inputs and the activation (in either order), the corresponding output will become active. Removal of both input and activation will lower the output signal. This behaviour allows the introduction of an activation synchronised filter in the paths of a number of disjoint incoming signals.

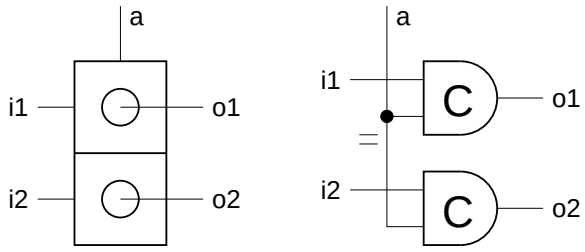


Figure 5.2. A two way decision wait

The DecisionWait consists of an *n* way decision wait and an *n* way OR gate. The requests from input passive ports are directed towards the output active port requests through the

decision wait. Each acknowledgement from the output ports is forked and sent separately to the acknowledgement of the passive port and to one of the inputs of the OR gate which in turn leads to the acknowledgement on the activation port. The activation request is connected to the decision wait activation. This implementation is shown in [fig. 5.3]. Only those forks within the decision wait have a timing requirement, the acknowledgement forks are untimed as the active port request will only change after both acknowledgement and passive port requests have been removed.

The FalseVariable component is introduced to allow a number of reads from an input channel to be enclosed within a single communication on that channel. This creates the read-only variable behaviour of channels within `select` guarded communication commands. Where the data valid period of the incoming channel's communication is the same or is longer than that of the control and variable-read channels within the guarded command (such as a broad input channel, broad control and reduced broad variable-reads) the FalseVariable can be implemented with only delay matching elements. No input latch or control interaction between variable reads and the input channel is necessary. The enclosure of reads within the input communication is enforced by the environment and the compilation scheme's derivation of the guarded command's activation from the output **signal** port on the FalseVariable component. Such a zero cost FalseVariable is shown in [fig. 5.3]. Where protocols differ, the enclosure can usually be created by the insertion of a protocol adapter (such as an S-element) in the activation path as shown in [fig. 5.3].

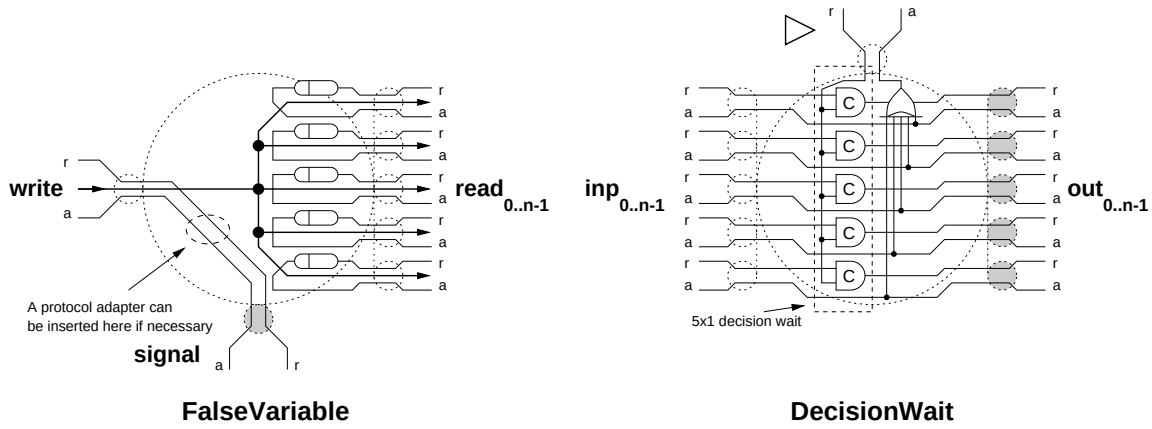


Figure 5.3. Gate level implementations of FalseVariable and DecisionWait

5.1.1.4. The modified case component – Case

The Tangram CASE component allows an input communication to initiate a communication on one of two output ports. The choice of output port is made by the most significant bit of the incoming communication, the remainder of that communication's value is passed along the output port.

Trees of CASE components can be constructed to implement multiway `case` commands by presenting the value of the discriminating expression onto the input port of the root component of the tree, such an arrangement (for the four way case) is shown in [fig. 5.4]. The output ports at the leaves of the CASE tree are, in fact, nonput and connect the leaf components to the activations of the commands associated with each `case` choice. This arrangement is very simple for `case` statements in which all 2^n outputs are connected to commands. Where a number of cases are not covered (or are covered by a default `else` case) Halt or Continue components (STOP and RUN for Tangram) can be used to 'cap' a dangling output port. An example of this capping is shown in [fig. 5.1] where the `else` clause of an `if` command is capped with a Continue, the `if` command is implemented in the form of a two way case command. Where both outputs of a CASE component are capped with the same type of component, that CASE component can be pruned to leave just a Continue or Halt in its place.

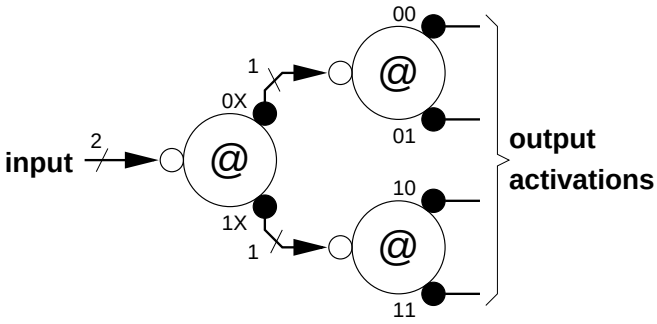


Figure 5.4. A three component CASE tree

It may also be useful (indeed necessary for the implementation of Balsa `case`) to combine a number of leaf activations into a single activation where a single command should be activated for a number of different values of discriminating expression. A CASE tree implementation would implement such an extension with the addition of Call components combining leaf activations, [fig. 5.5] shows such a Call linked CASE tree. This example is that of the case

command given in §4.3.2, the enumeration elements ALU, LOAD, STORE and BRANCH having the values 0, 1, 2 and 3 respectively. Notice that as the LOAD, STORE case spans two CASE components we cannot prune the tree to remove the Call component (which is possible where the Call combines both outputs of a single CASE). All forms of the more flexible Balsa case command can be implemented with CASE trees. Pruning and Call activation combination may produce a resultant control graph (consisting of a CASE tree drawn together by a number of Call trees) which is fairly complicated.

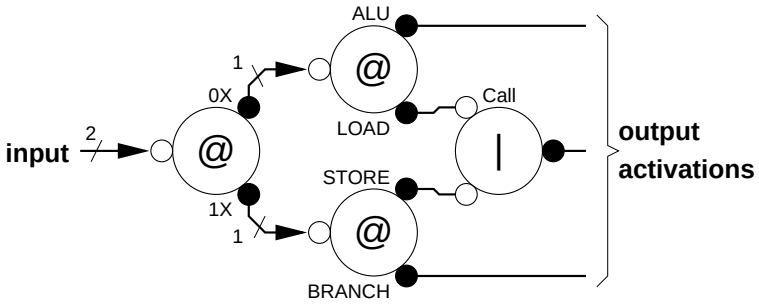


Figure 5.5. A CASE tree with Call linked outputs

An alternative approach is taken by the Balsa Case component. A single input's value will determine which, if any, of a number of nonput (output) activation channels is activated. The Case component therefore has three parameters:

1. The width of the input data bundle.

2. The number of active nonput channels.

3. A specification string which determines the behaviour of the component by means of a value set to nonput channel mapping.

The symbol for the Balsa Case component is the same as that of the Tangram CASE with the exception that the Balsa part has only nonput output activation ports (as opposed to the bit-stripping output ports of CASE) and a parameterisable number of such ports.

A single Case component can replace an entire CASE tree. Multiple input values can be associated with a single output activation and the Continue-capping functionality is handled internally. This generality of behaviour is specified by the specification string, which must be of the form:

⟨case_spec⟩ ::= ⟨case_spec_term⟩ (, ⟨case_spec_term⟩)* (, _)?

⟨case_spec_term⟩ ::= ⟨range⟩ (; ⟨range⟩)*

⟨range⟩ ::= ⟨decimal_integer⟩
| ⟨decimal_integer⟩ .. ⟨decimal_integer⟩

The ⟨decimal_integer⟩ term should be a non-negative integer literal which is within the range $[0, 2^w - 1]$ where w is the width of the input port to the Case component.

A semi-colon separated list of ⟨range⟩s makes up a ⟨case_spec_term⟩. Each ⟨case_spec_term⟩ specifies those input values which should invoke the command which is in the same position in the list of active nonput channels as the ⟨case_spec_term⟩ is in the ⟨case_spec⟩ list. The Case component which implements the `case` command of the processor example of §4.3.2 is expressed in Breeze as:

```
$BrzCase ( 2,3,"0,1..2,3" | #1,{#2,#3,#4} )
```

The channel numbers here match those shown in [fig. 5.5] for the activations to the command for each case. Instead of being a ⟨case_spec_term⟩ the last port specification in the ⟨case_spec⟩ may be '_'. This indicates that the associated port should be activated for any value of the input channel which falls outside those values covered by the other ⟨case_spec_term⟩s, this is the specification for an `else` clause. Without an `else` connection the Case component defaults to the behaviour of a Continue component when an input value falls outside its specified ranges.

Implementing Case is fairly straightforward. As we can be sure that the specified ranges are disjoint, a piece of automatically optimised sum-of-products combinational logic can be used to map input values to output activations. The arrangement shown in [fig. 5.6] shows such a piece of logic, in the form of a PAL, with a rank of AND and single inverted input AND gates generating the true and complement inputs to the PAL (constituting dual-rail encoded versions of the input signals, the isochronic forks present in this dual-rail expansion can be encapsulated within a demultiplexer cell). The outputs of the PAL are connected to

the outgoing nonput ports' requests. Returning acknowledgements are combined by an OR gate to form the acknowledgement of the input channel. As previously mentioned, the use of an automated optimisation of the decoding logic is not a potential source of hazards, this is because the 4-phase handshaking protocol requires that the request be removed from the selected output port before its acknowledgement is removed. The output request is generated from a piece of combinational logic for which both the true and inverse inputs to the AND gates are gated by the incoming request and so the outputs of the AND gates which make up the first level of the combinatorial logic can never go high unless the correct input coding is present (and once high they are not lowered until the output handshake is completed and the input request is removed). The implementation is free of both static one hazards (due to the stable nature of the inputs after the arrival of an input request) and static zero hazards (due to the SOP implementation and gating of both true and complement data inputs).

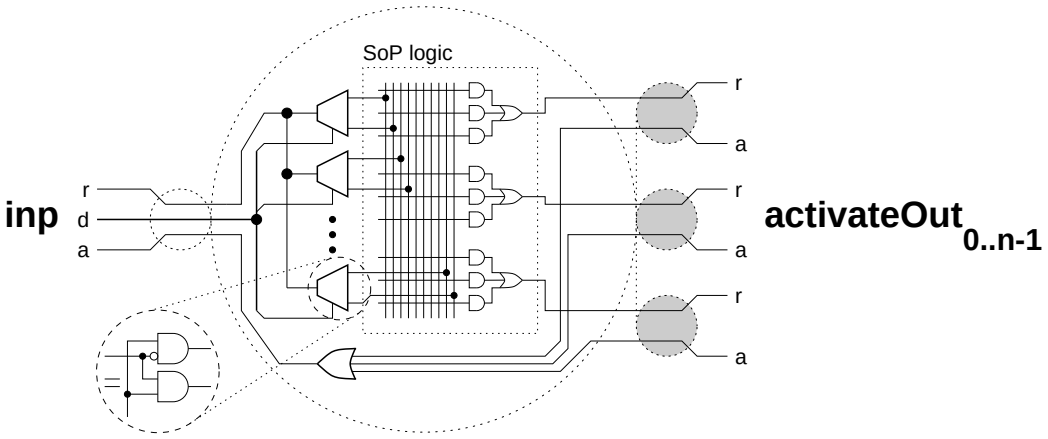


Figure 5.6. The Balsa Case component's implementation

5.1.1.5. Arbitration – Arbiter

The Arbiter component used by Balsa has the same required behaviour as that of the Tangram ARB component. It is used to guarantee the mutual exclusion of two passive nonput channels' communications by passing a single communication at a time onto one of two active nonput channels. The behaviour is:

$$Arbiter(inpA, inpB, outA, outB).\ \#[[inpA : outA \mid inpB : outB]]$$

Here the arbitrating nature of the communication driven selection notation is important.

It is important to ensure that the mutual exclusion is enforced across the entire duration of the incoming communication. This is doubly important to the use of Arbiter in Balsa since the DecisionWait component to which it is usually connected does not ensure a safe behaviour where a request on one input arrives before the falling request of another input. The required functionality can be achieved by the inclusion of a few extra gates around a mutual exclusion element (which is a cell library primitive), [fig. 5.7] shows such a construction. Note the timed forks which are necessary to ensure that the hand over of the mutual exclusion from the incoming requests to the generated acknowledgements cannot glitch due to a fast response of the environment by removing the incoming request (sequenced to occur after the acknowledgement reaches the environment beyond the fork) before the acknowledgement is observed by the OR gate.

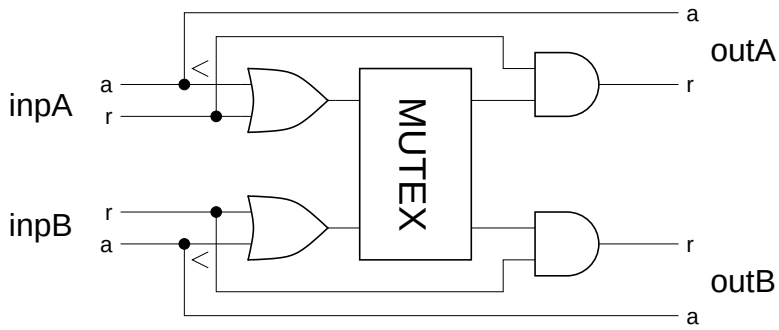


Figure 5.7. An Arbiter implementation around the mutual exclusion element

5.1.2. Channel construction components

The meaning of a channel within the Balsa and Tangram languages is not the same as a channel at the handshake circuit level. A handshake circuit channel will connect a single passive port to a single active port in a point-to-point manner. Language level channels can be used for the same purpose but can also communicate in a broadcast manner where a single output is connected to multiple inputs for the purpose of synchronised communication. Language level channels can also be used sequentially in a way that can result in that channel being split in two such as in the case:

```
local channel a : word
```

```
begin
  a -> x || a <- 5 ;
  a -> y || a <- 10
end
```

The two (sequential, multiplexed) communications on a could easily be optimised into a pair of single communications on two language level channels. In the mapping of language level to handshake circuit channels a set of channel constructing components are used to give us the sequenced multiplexing, broadcast and passivating (the connecting of many active ports together) behaviours. The channel construction components and their uses are:

5.1.2.1. Call and CallMux (MIX and MIX[PUSH])

Call and CallMux implement channel multiplexing. Call is the same component as a micropipeline call, CallMux incorporates a data multiplexer steering one of a number of input channels towards a single output channel.

5.1.2.2. CallDemux (MIX[PULL])

CallDemux has the same control behaviour as Call and CallMux but has pull rather than push ports. On receipt of a single request on one of its output ports, the CallDemux will request, receive acknowledgement from and convey a value from a single input port to the requesting output port.

5.1.2.3. Fork and ForkPush (FORK)

The Forks distribute activations (and data for the ForkPush) from a single passive input to all their active outputs. In a technology where return to zero synchronisation is allowable (such as a 2-phase component set or cases of parallelism where independence of RTZ is not necessary), the Fork can be substituted for a Concursor.

5.1.2.4. Synch, SynchPush and SynchPull (JOIN)

The Synch components perform synchronised communication. Synch connects a number of passive nonput ports into a single active nonput port on which one communication will

occur after receipt of requests on each of the passive ports. SynchPull has a similar behaviour (with pull ports) but also conveys the value read from the common port to each of the others. SynchPush has the set of pull ports of SynchPull but receives its input value along a passive (push) input, the value is sent to each push output and also along an active 'extension' port (which can be connected to the passive input of a similar SynchPush to form a larger SynchPush component). The push JOIN component is the primary means of connecting active channels used by the Tangram compilation mechanism, the component SynchPush finds a similar role in Balsa.

5.1.2.5. Passivator and PassivatorPush (PAS)

The Passivator and PassivatorPush components are very similar to Synch and SynchPush except that all ports are passive and the PassivatorPush component has no extension port (this is in fact the optimised form of SynchPush where the extension port is connected to a Continue).

5.1.2.6. Continue, ContinuePush, Halt and HaltPush (RUN, STOP)

Continue and ContinuePush are handshake sinks, they immediately return an acknowledgement to any request. This 'null' behaviour can be useful for tying off unused ports on other components. Halt and HaltPush are similarly simple components, they **never** return an acknowledgement and are useful for introducing deadlock failure into descriptions.

The implementations of all these components are identical to their Tangram counterparts excepting that all the Balsa components (other than the 'channel terminators' Continue, ContinuePush, Halt and HaltPush) have a set of arrayed ports. The behaviour of the multi-ported components is that of 'tree' combinations of their two-ported Tangram equivalents, this difference is introduced to allow more compact handshake circuit descriptions by gathering together glue components (and also allowing balancing and amalgamation of components within the C-element, OR gate trees and wire forks of which these components are composed).

5.1.3. Datapath components

The datapath components are also very similar to Tangram. The only notable addition is the CaseFetch component which is used to implement non-constant indexed array element extraction expressions. A specification for CaseFetch was given in §4.3.1, a further description will not be given here as CaseFetch represents work in progress in the enhancement of the Balsa language.

5.2. The Breeze intermediate language

The Breeze language has two functions within Balsa. Firstly it serves as the handshake circuit target format for the balsa-c compiler, containing the handshake component netlist for the compiled circuit. Secondly it also acts as a library format for balsa-c as a Breeze description can be read back into balsa-c and used as the basis for a new description. In this way Breeze allows us to achieve some level of reusability and separate compilation within Balsa.

5.2.1. Breeze syntax

Syntactically Breeze is very similar to Balsa, a single design file will contain a number of declarations for constants, types and parts. All declarations within a Breeze file are exported from it and so only a single top level declaration list is necessary. The single place buffer of §3.1.3, passed through balsa-c, becomes (a few balsa-c generated comments have been removed):

```
-- Breeze intermediate file
-- Created: Fri Nov 28 13:20:33 1997
-- By: bardslea@amu9 (SunOS)
-- With: balsa-c version 970612

type word is 8 bits

part buffer (
    passive sync activate;
    active input i : 8 bits;
    active output o : 8 bits ) is
attributes ( isProcedure,isPermanent,noOfChannels=8 )
local
```

```
        sync #1
        pull channel #2 : 8 bits
        push channel #3 : 8 bits
        pull channel #4 : 8 bits
        sync #5
        push channel #6 : 8 bits
        sync #7,#8
    begin
        $BrzVariable ( 8,1,"x" : #6,{#4} )
        $BrzLoop ( #1,#8 )
        $BrzSequence ( 2 : #8,{#7,#5} )
        $BrzFetch ( 8 : #7,#2,#6 )
        $BrzFetch ( 8 : #5,#4,#3 )
    end
```

The file begins with a compile-run information comment block, type declarations follow (for `word` in this case), then followed by `part` declarations. A part is the handshake component implementation of a procedure and will have the same port structure as the procedure from which it was compiled with the addition of a passive sync activation input. After the ports, a list of `attributes` is given, these may be either just names or name – value pairs. Here there are three of them:

`isProcedure`

This part was compiled from a Balsa procedure and so can be reconstituted into a procedure when read back into balsa-c.

`isPermanent`

This part returns no acknowledgement to an activation (as there is an unbounded loop within connected to that activation). Attributes are a useful means of conveying such procedural behaviour properties to back-end tools.

`noOfChannels=8`

The `noOfChannels` attribute has a numeric argument. In this case this indicates that this part has eight (handshake circuit) channels in its definition. Knowing the number of channels within a part declaration may allow a script reading this Breeze file to allocate a correctly sized array before the channel information is read from the file.

There then follows a list of all channels, this list is sequentially numbered and includes those

channels connected to the part's ports (which are numbered `#1` for the activation, `#2` for input `i` and so on). The typing of these channels is reduced to a numeric type of the correct width to match the real type of the channels as expressed in Balsa. This reduction reduces the required complexity of typing handshake component ports and also allows 'dumb' scripts to work on Breeze files.

The final section of a part's definition is the list of handshake components itself. Connections between components and to the ports of the part are made to the numbered channels. No unresolved connections are allowed as no nesting of part definitions is allowable under Breeze.

The similarity of Breeze to Balsa is an intentional feature to allow Breeze parts and Balsa procedures to co-exist in the same file (thus allowing the `import` feature to be implemented using simple file inclusion).

5.2.2. Breeze within the Balsa system

The position of the Breeze description within the Balsa design flow is shown in [fig. 4.1]. As well as its use as an intermediate between balsa-c and breeze2gates, Breeze is the source format for the breeze2lard simulation and breeze2dot handshake circuit visualiser.

What is not shown is Breeze's role as a method for including hand designed components and circuits within an automatically generated design. Hand-crafted part descriptions can be included in a Breeze file in the same form as those automatically generated if the user so desires. Parts can also be made to refer to hand-crafted gate or transistor level circuits by setting the appropriate attribute and omitting the part's component list (the channel list must still remain to allow scripts to width-wise type the part's ports).

The inclusion of Breeze parts into Balsa descriptions allows the user to explicitly place handshake components. This feature further expands the descriptive possibilities of Balsa by allowing the user to circumvent the compilation process.

5.3. The breeze2gates handshake component to gate mapper

The breeze2gates tool described here is currently only a prototype for a complete gate level mapping tool. The functions and internal structures for such a tool are outlined here along with the current support provided by breeze2gates. Such a tool has two main functions:

1. To interact with the target CAD system to determine which parameterised components required of a design are already present in that system's database. The remaining, non existent, components would be generated by breeze2gates and entered into the CAD system.

2. To generate a native format netlist of the handshake circuit's structure for entry into the target CAD system.

The completed breeze2gates will replace the single-rail back-end developed for the Tangram HCL to structural Verilog mapping by the EXACT project at Manchester. That tool was written in the Cadence design framework's internal scripting language SKILL (which has aspects of both Lisp and C) and interfaced directly with Cadence by performing internal database lookup and symbol manipulation through SKILL's database programming interface. In order to allow a degree of portability between CAD systems (and also, possibly primarily, to allow development without having to interface to a large, running CAD system) breeze2gates is currently implemented in Perl.

5.3.1. Perl – the practical extraction and report language

Perl[1] [30] is a very similar language to SKILL in many ways. Perl was developed by Larry Wall as a practical scripting language for system administration tasks. It incorporates the imperative, iterative style of C (with support for formatted output, pattern matching and persistent objects) with the native support for list datatypes, garbage collection and manipulations on lists of Lisp. Perl also provides *hashes* (associative arrays or dictionaries), list / array duality (both indexing and link-following for lists) and a stand-alone, scripting

[1] also known as the Pathologically Eclectic Rubbish Lister, this name is actually in the Perl UNIX manual page!

nature which allows breeze2gates to be implemented as a separate program from balsa-c and the target CAD system. A final implementation may however favour a different language as Perl has a reputation for obscure syntax and context dependent evaluation of expressions (e.g. evaluating a list in scalar context will return the number of items in that list, not report an error) which make debugging a chore. The embeddable Scheme [40] interpreter, Guile (GNU Ubiquitous Intelligent Library Extension, [39]) may offer a better solution to both the generation of back-end scripts and the internal debugging of balsa-c. Guile allows C and Scheme to share data structures in such a way that a program can be constructed from components from both languages.

5.3.2. The Breeze parser and the netlist format

Breeze2gates receives as its input a Breeze file describing the design to be mapped. Only flattened handshake component descriptions are considered here but a complete tool would include provision for a tree of Breeze files to be included providing different, possibly nested, parts of a design. The Breeze file is processed by a parser and transformed into an internal netlist format, type and constant information is discarded, breeze2gates only considers the widths of channels to be significant. The five component buffer Breeze example given in §5.2.1 would in the internal format become:

```
[ ['BrzVariable', [8,1],
    [['gp_6',0,[['b_6',8,0]]],
      ['gp_4',1,[['b_4',8,1]]]]],
  ['BrzLoop', [],
    [['gp_activate',0,[['c_activate',0,0]]],
      ['ga_8',0,[['b_8',0,0]]]]],
  ['BrzSequence', [2],
    [['gp_8',0,[['b_8',0,0]]],
      ['ga_7_5',2,[['b_7',0,0],['b_5',0,0]]]]],
  ['BrzFetch', [],
    [['gp_7',0,[['b_7',0,0]]], ['ga_i',0,[['c_i',8,1]]],
      ['ga_6',0,[['b_6',8,0]]]]],
  ['BrzFetch', [],
    [['gp_5',0,[['b_5',0,0]]], ['ga_6',0,[['b_6',8,1]]],
      ['ga_o',0,[['c_o',8,0]]]]]
]
```

This is a very regular but difficult (for humans) to read format. In Perl, square brackets delimit lists of values. This is, therefore, a list of components. Each component is a triple (or rather: a list of three elements) of (name, parameter list, port list). Each port is also a triple, this time of (name, number of subports, subport list). A subport is a component port of an arrayed port, the second port of the Sequence component in the above example is a good example of an arrayed port (the port named `ga_7_5` which is a two element array of channels #7 and #5). Non-arrayed ports have a subport count of zero and only a single subport in their subport list. Each subport is a triple of (name, width, is pull) where is pull is 1 for a pull channel and 0 for a push or sync channel.

The naming of ports, and subports is consistent with either their channel number or port name (in the case of external ports). External ports are named `c_externalName`, internal channels are `b_channelNumber` and groups are `gp_subportNameList` or `ga_subportNameList` depending on the sense of that port. This explanation illustrates the verbosity (and overbracketted!) nature of a consistent simple notation for representing netlists in a list form. A similar form of netlist representation is used for gate level descriptions (where the port – subport relationship becomes a form of bus construction – ripping) with each component explicitly placed by breeze2gates requiring the long form notation to be used for its instantiation.

5.3.3. Scripting to specify parameterisable components

Each parameterisable component has three associated Perl functions: `ComponentName`, `ComponentPorts` and `ComponentInstances` which return the name, port specification and component lists for `Component`. The `ComponentInstances` function is the most important of these as it actually performs the duty of parameterising the components gate level implementation. As an example, here is the `SynchPullInstances` function:

```
# SynchPullInstances – make a SynchPull with
#   outputCount width sized ports
sub SynchPullInstances # args: width outputCount
{
  my ($width, $outputCount) = @_;
```

```
    my $i;
    # Join input ack. to pout acks. through a fork
    my @instances = (ForkTree (0, ['inp_a',0,1], 'ack',
      SingletonSlicesAcrossNameRange
        ('pout','_a',0,0,$outputCount)),
    # Combine pout reqs. to inp req. via a C-element tree
      CElementTree (['inp_r',0,1], 'req',
      SingletonSlicesAcrossNameRange
        ('pout','_r',0,0,$outputCount)));
    # Handle the data connections (wire forks)
    for $i (0 .. ($width-1))
    {
      push @instances,(ForkTree (0, ['inp',$i,1], 'data$i',
        SingletonSlicesAcrossNameRange
          ('pout',",$i,0,$outputCount)));
    }
    return \@instances;
  }
```

The functions `CElementTree` and `ForkTree` create arbitrary width trees of C-elements and wire forks (with buffering) respectively. Together with port wire slicing functions like `SingletonSliceAcrossNameRange` (which extracts a list of each *i*th signal from the given port array) and `InvTree` which creates a buffered inverter tree we can construct any of the channel interconnecting components in only a few lines of code. The advantage of writing such a parameterising function in a language such as Perl rather than relying on the parameterisation expressive functionality of hardware description languages / file formats such as VHDL and EDIF is the flexibility which a programming language gives to allow the parameterisation to be expanded to encompass technology dependent features such as timing analysis (a parameterising function could potentially calculate worst case delays) and capacitive load balancing (such as is employed in the `InvTree` and `ForkTree` functions).

The gate level descriptions of handshake components are printed from the internal form into a file in the structural Verilog format. This is then read into Cadence using VerilogIn which creates a basic symbol and (usually fairly poorly laid out) schematic. The handshake component netlist is treated in a similar fashion to create the final, hierarchical, design in Cadence. As the internal form of the netlist is independent of the target CAD system's netlist format, formats other than Verilog (structural VHDL, EDIF or BLIF for example)

may be generated by changing the choice of functions used to map the internal form into the file-borne format.

5.4. Peephole and gate-level optimisations

Optimisations to handshake circuits can take place at several distinct stages:

- Language level optimisations. These are typically performed during attribute grammar evaluation and include constant folding and unreachable code removal.
- Handshake component peephole optimisation. Post-compilation optimisations on the handshake circuit netlist. A typical example is the removal of Tangram CON connector components when compiling using the Tangram compilation function (the Balsa compiler generates no connectors).
- Gate level peephole optimisations. Gate level optimisations are typically performed as a final stage after flattening the handshake components of the circuit into individual gates. Timing related gate substitution such as the removal of Fork – Delay – C-element structures around variable read ports in expressions ([4] outlines many such optimisations) may also come under this heading.

Many optimisations are possible and the balsa-c compiler and breeze2gates back-end implement some of the more straightforward language level and handshake component level cases. This section will be restricted to a discussion of the optimisation surrounding `select` implementation as this is the major new feature which Balsa introduces over Tangram.

In the compilation of `select`, handshake component and gate level optimisation may be employed to produce significantly more area efficient implementations. Two notable handshake component optimisations exist:

5.4.1. DecisionWait activation removal

In each of the examples *combine* and *mux* (introduced in §3.2.2.2 and §3.2.2.1 respectively) there is an unbounded repetition of the select behaviour. In practice the activations of these procedures would often appear at the top level of a circuit description (connected to a Fork

component activated by the activation for the whole circuit). The activation on the repeater is therefore acting as a circuit reset. The action of the DecisionWait element in allowing the activation of the guarded command to be stalled until the arrival of the circuit activation is made unnecessary. The DecisionWait component and the repeat can therefore be removed. The activation signals for the guarded commands would then be sourced directly from the incoming requests on the guarding channels. If these requests are held low during circuit initialisation, the command activations will be similarly low and so the self-initialising feature of handshake circuit descriptions is preserved.

5.4.2. FalseVariable – Transferrer removal

In the *mux* example, once the DecisionWait component is removed due to the previous optimisation, the activations of the guarded commands are each connected directly to the activation port of one of two transferrers. The input ports of these transferrers are each connected to the read port of the same FalseVariable that provides their activation. The FalseVariable / transferrer pair can be seen to act as a connector from incoming (selected) port to the output port of the transferrer and so both components can be discarded.

For the *mux* example this leads to the interesting result of the final, post-optimised implementation being a single CallMux component from input ports (a, b) to output port (c). Balsa's enclosing select construct allows the user to describe (and so allows the user to explicitly describe) a push handshake component which would otherwise only be available to the user through channel mixing or shared variable write ports.

The gate level implementation of the *combine* example given in §3.2.2.2 (whose handshake circuit is shown in [fig. 3.2]) is shown in [fig. 5.8]. This shows the typical implementations of FalseVariable and DecisionWait components for cases where the input and nonput channel protocols are the same (assuming broad input channels, broad nonput channels and a reduced broad protocol on the read side of FalseVariables).

The enclosure of FalseVariable read port accesses within the 'signal' based activation allows a very simple read port implementation in most protocol combinations. Broad read ports would incur the cost of a latch, with differing read and write port protocols the mismatch could be

solved by a protocol adapter in the write – probe path inside the FalseVariable.

The DecisionWait implementation shows only a single passive channel (other than the activation), this component is equivalent in function to the SynchPush component with the activation and passive input as the SynchPush's two passive inputs. For DecisionWaits with two or more inputs (let us say *n* inputs) the C-element and fork become *n* C-elements, *n* (self timed) forks, an *n*-way isochronic fork supplying the activation request to each of the C-elements and an *n*-way OR gate (or similar merge element) connecting the forked acknowledgements of the active ports to the activation acknowledgement. The untimed forks in the acknowledgement paths imply that the complete enclosure of the guarding communication acknowledgement within the activation communication cannot be guaranteed. This is not an issue however as this behaviour is common to passivator based communication structures (where an acknowledgement is forked towards a number of passive ports) and expresses the general result that (under trace reordering in a delay-insensitive setting) the enclosure of a passive ported communication by another passive ported communication has no fixed acknowledgement ordering.

Examples of possible gate level optimisations on the handshake circuit of the *combine* example are:

- Removal of the delay matching and C-element rendezvous of the FalseVariable read ports and Combine components. The C-element in question could be replaced by a single delay or the inclusion of this delay in the activation path of the transferrer (or indeed the DecisionWait, SynchPush or both FalseVariable's signal ports). This is a timing related optimisation.

- Amalgamation of the two input C-elements in the SynchPush and DecisionWait into a single C-element.

- Removal of the SynchPush component's C-element in the case where inputs a and b are sourced from variables and so have only delay elements in their request – acknowledgement paths.

After removal of the DecisionWait and the C-element within the Combine component, the implementation of the *combine* example becomes the circuit shown in [fig. 5.9]. This circuit bears a close resemblance to the original **pull** combine component except that the operation is now data driven.

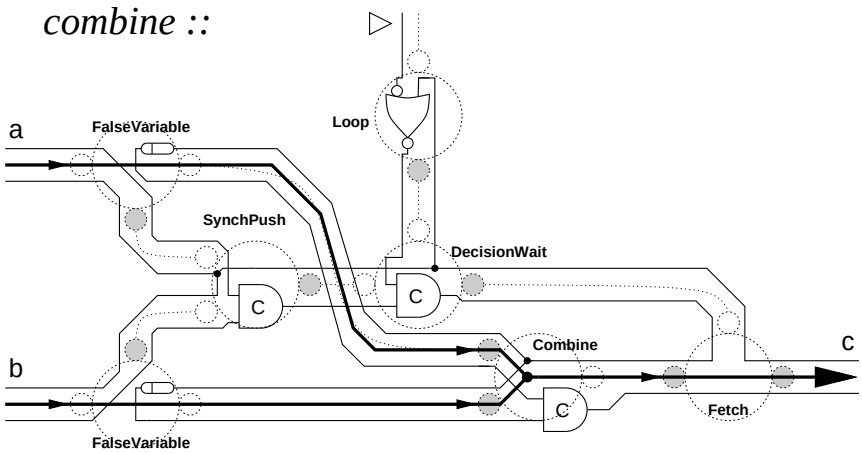


Figure 5.8. Gate level implementation of *combine* – before gate level optimisation

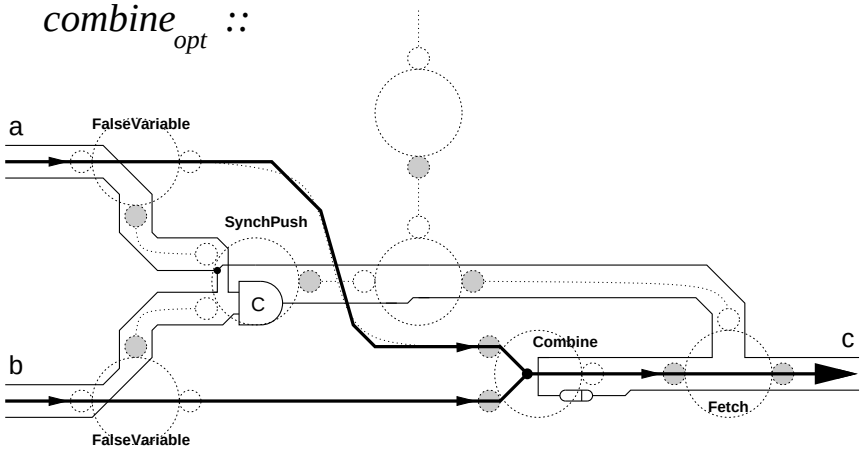


Figure 5.9. Gate level implementation of *combine* – after gate level optimisation

5.5. Simulation using LARD

5.5.1. Choosing the 'level' of simulation

Simulation of a Balsa description can be performed at several distinct levels of abstraction (in much the same way that optimisations can). These include:

Language level behavioural simulation

A simulator would be constructed which accepts Balsa as an input format and is capable of executing Balsa within a debugging environment. Alternatively Balsa could be transformed into the input language of an existing simulation system.

Handshake component simulation

The behaviours of the individual handshake components of the compiled Balsa description are simulated as a channel connected composition. Direct relationships between the components and channels here and those in the source Balsa are still evident (largely due to the transparent nature of the compilation scheme), estimated timing may also be introduced.

Gate level simulation

Either the flattened or hierarchical (the hierarchy being defined by the composition of handshake components) gate level netlist could be used as the basis for simulation. Realistic gate timings and estimated interconnect timings can be introduced.

Switch and analogue extracted layout based simulation

Post layout capacitance / resistance extraction with transistor or cell level switch or analogue models can be used to perform more timing accurate simulation. This is the lowest level of simulation usually performed and typically consumes more simulation runtime than any other form.

As the aim of simulation at the current time is to judge the correct function of descriptions and the reliability of the compilation mechanism, realistic timing is not of great importance.

The possibility of further change to the Balsa language and the difficulties this may cause in re-implementation of any language level simulator would tend to favour handshake component level simulation. The requirements for such a simulation environment are:

- The simulation language must be able to specify the handshake components at a signal transition level and allow the inclusion of back-annotated timing if necessary.
- An environment must exist for the graphical display of simulation results and the debugging of developing descriptions.

- The environment must be portable to a similar degree as the Balsa tool set and allow for the easy (free?) distribution of both simulation models and possibly the environment itself.

Fortunately the chosen solution presented itself in the form of LARD [35], the Language for Asynchronous Research and Development, created by Phil Endecott as part of a project to aid the modelling and refinement of the AMULET3 microprocessor within the AMULET group.

5.5.2. LARD simulation

LARD provides CSP-like mixed sequential and parallel commands with channel communication as a basic language feature. LARD for the most part resembles a concurrent programming language, being built around a shared memory, time-slice multi-threaded virtual machine. Communication primitives are implemented on top of this machine and function by signalling data validity through an element of a shared memory structure. On top of LARD, an expanded channel library is used to provide pull channels and pull channel handshake enclosure primitives to the user (this library was written by John Bainbridge and expands LARD's single-track-like communication primitives to support 2 and 4-phase early, late, broad and tri-state'able channels).

On top of these, a library of handshake component behavioural descriptions forms the functional core of each simulation. LARD provides type-polymorphic features which make the description of parameterisable handshake components simple, there is no need to create specific instances of the required components for each differing set of parameters. The LARD type `int` which is the usual type for numeric information in LARD models does present a problem where the simulation of Balsa descriptions is concerned. A Balsa type may have any positive number of bits limited only by the native integer representation of the target machine, this is a limit on the **number of bits** in each type not that type's numeric range.

The LARD `int` type only allows 32b numbers to be represented and so to implement the multi-precision arithmetic of Balsa a different type must be used for simulated channels. In the current version of the simulation generator each channel's value is represented by an array of boolean elements, one for each bit of that value. This is a fairly poor encoding (32b required

for each bit represented) and may be improved upon in future versions.

The transformation from Breeze to LARD is performed by the Perl script breeze2lard. The output of breeze2lard is fed into the LARD compiler and then onto the LARD simulation environment. The LARD environment's *channel viewer* allows each handshake channel in the design to be monitored and for the behaviour of the design to be observed. Test harnesses can be written in either Balsa or LARD and compiled into the simulation using breeze2lard and the LARD compiler. An example of the environment in action is given in §6.1.

5.6. Chapter summary

This chapter has described gate level implementations of the new handshake components introduced in previous chapters. Short descriptions are given of all the components of the Balsa handshake component set have been given together with a means for simulating handshake circuits using the asynchronous modelling language LARD.

Peephole optimisations of common uses of the DecisionWait and FalseVariable components are suggested. A description of many such optimisations possible in the core of components which Balsa shares with Tangram can be found in Ad Peeter's Ph.D. thesis [4].

Significant optimisations to the input selection mechanism are proposed for cases where the activation of the selection command is 'directly' connected to the top level circuit activation through a repeater. Initially the requirement for an activation appears to be a disadvantage of handshake circuits when compared to other descriptions where only a circuit reset is required. This is misleading as in the majority of cases where the activation is not used (i.e. connected through a repeater to the circuit activation) the removal of acknowledgement path gates leads to circuits similar to those created by design styles which include an explicit reset signal.

Chapter 6. The example designs, the SSEM and STUMP

Two example designs are given here: a communication-coupled-block model of the STUMP microprocessor and a simpler, sequential description of the SSEM processor. These were chosen as test designs due to their limited size and the desire to give examples of both the Tangram / CSP descriptive style with the SSEM implementation and of the use of enclosing selection in the description of a push communication block modelled processor, STUMP.

6.1. The small scale experimental machine, SSEM

The small scale experimental machine (SSEM or the Baby) was the prototype implementation of the Manchester Mark I digital electronic computer. In June of 1948 the SSEM ran its first program and became the world's first stored program digital computer. A replica of this original valve and Williams tube based machine has been constructed by the Computer Conservation Society at Manchester Computing with assistance from The University of Manchester and ICL PLC. The commissioned replica should execute that same first program in June of 1998 to celebrate the 50^{th} anniversary of computing in Manchester and the birth of modern digital computing. It seems fitting to use this very simple computer as an example to demonstrate the capabilities of Balsa as a circuit synthesis system and to contrast the advancement of computing technology from racks of valves and CRT's in a bit serial, multiple cascaded clock computer with just 32 words of store to a VLSI implementation of the same design with a bit parallel datapath and asynchronous control.

6.1.1. The behaviour of the SSEM

The behaviour described here is that outlined in the documentation to M1SIM, an SSEM emulator written by Andrew Molyneux which runs under DOS on IBM compatible PCs. Equivalent behaviour with respect to the rebuilt SSEM is dependent on the accuracy of this documentation.

The SSEM is a 32b processor with a single (fixed length) instruction format specifying an instruction type and an absolute address for that instruction's operand. This is a single address instruction machine with a single accumulator register. ALU operations are of the form $ACC \coloneqq ACC \odot M[absolute]$. Only the relative jump instruction JRP treats the address field as anything other than an operand address. JRP adds the address field value to the CI (Control Instruction, the program counter in modern terminology) register. The absolute jump instruction JMP uses the address field to fetch the jump target address from the store so making JMP an absolute **indirect** jump instruction. TEST is the third and final form of jump. TEST skips the next instruction if $ACC \leq 0$ and is the only instruction which has a conditional outcome.

The single instruction format is shown in [fig. 6.1], the address field being split between high order (CRT selection) and low order (CRT line select) bits as each CRT can hold 32 lines of 32b each. Here we consider only a single CRT design and so only the 5b low order address field is relevant.

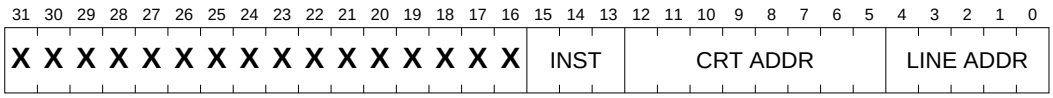


Figure 6.1. SSEM instruction format

Execution of an instruction proceeds in the following order:

1. $CI \coloneqq CI + 1$ – Increment CI
2. $PI \coloneqq M[CI]$ – Fetch instruction (PI is the present instruction register)
3. Decode and execute instruction, with memory operand fetch if necessary

This sequence repeats until a STOP instruction halts the machine. Notice how incrementing CI occurs before instruction fetch and so the first instruction to be executed will be from address 1.

6.1.2. The Balsa top level

The instruction format and this fetch – execute loop behaviour give the basis for the Balsa

description:

```
-- Balsa SSEM description, first attempt

type Word is 32 bits

type SSEMFunc is enumeration
  JMP, JRP,     -- Abs. and rel. jumps
  LDN, STO,     -- Load negative and store
  SUB, SUB_alt, -- Two encodings for subtract
  TEST, STOP    -- Skip and stop ;)
end

type SSEMInst is record
  LineNo : 5 bits;
  CRTNo : 8 bits;
  Func : SSEMFunc
over Word -- Pad 16b to 32b

-- First attempt at SSEM top level
procedure SSEM ( … ; sync Halted ) is
local
  variable CI, ACC : Word
  variable PI : Word -- cast to SSEMInst where needed
  variable Stopped : bit
  … -- other declarations
begin
  ACC := 0 || CI := 0 ||
  Stopped := 0; -- reset initialisation
  while not Stopped then
    IncrementCI();
    FetchInstruction();
    DecodeAndExecuteInstruction()
  end;
  halt; sync Halted -- STOP instruction effect
end
```

The `Stopped` variable allows us to define a behaviour for the STOP instruction which is not part of the main fetch – execute loop (which also signals completion along the `Halted` port). The three procedures `IncrementCI`, `FetchInstruction` and `DecodeAndExecuteInstruction` will be filled in to provide the required functions. For example:

```
procedure IncrementCI is
```

```
begin CI := (CI + 1 as Word) end
```

Is a serviceable `IncrementCI` procedure but does introduce an extra, temporarily used, variable in which to store the result of the expression `(CI + 1 as Word)` before the assignment to `CI`. This variable is placed by the compiler to prevent the read from `CI` from overlapping the write back into `CI` (which would, with a transparent latch implementation or a transferrer specification which allows overlapping of handshakes on read and write ports, cause a race condition between the completion of the write and the propagation of the new value back to the inputs of the adder). We can share this temporary variable between other `CI` modifying assignments by making it explicit:

```
variable CI_slave : Word

procedure IncrementCI is
begin CI_slave := (CI + 1 as Word) end
```

The contents of `CI_slave` then need to be assigned into `CI` at the end of instruction execution.

6.1.3. Memory

Instruction fetch is just a memory read with assignment into the PI. `SSEM` must gain ports to access memory, say:

`MemRNW` — Direction select. Zero for write to memory, one for read.

`MemA` — Memory address, 32b wide: only the five least significant bits are significant.

`MemR` — Memory data read port. A read operation will have the form: `MemA <- address || MemRNW <- 1 || MemR -> MDR` creating a new register `MDR` to hold incoming data.

`MemW`

Memory data write port. A write operation will have a similar form: `MemA <- address || MemRNW <- 0 || MemW <- data`.

Three shared procedures will be used to access memory, namely:

`MemoryWrite`

Write ACC value to the address given in the current instruction.

`MemoryRead`

Read `ACC_slave` value (using a slave for ACC in much the same way as `CI` and `CI_slave`) from the address given in the current instruction into `MDR`.

`InstructionFetch`

Read new `PI` value from the address held in `CI_slave`.

The `FetchInstruction` procedure can now become just a call to `InstructionFetch`. The read instruction is now ready to be executed.

6.1.4. Decode and execute

Decoding and execution of the fetched instruction is done within a `case` statement. Each of the seven instructions matches a single case and invokes a command which will contain calls to the shared procedures. The procedures themselves will only be instantiated once but may be called by each of the case commands as only one command may be being executed at any given time. The `case` command looks like:

```
case (PI as SSEMInst).Func of
  -- Absolute jump
     JMP then MemoryRead(); CI_slave := MDR
also JRP then
  -- Relative jump
  MemoryRead(); CI_slave := (CI + MDR + 1 as Word)
also LDN then
  -- Load negative
  MemoryRead(); ACC_slave := ( - MDR as Word)
```

```
also STO then MemoryWrite()
  -- Subtract memory from ACC
also SUB .. SUB_alt then
  MemoryRead(); ACC_slave := (ACC - MDR as Word)
  -- Skip on negative ACC
also TEST then if (ACC as array 32 of bit)[31] -- -ve?
  then CI_slave := (CI + 2 as Word) end
  -- Stop dead
also STOP then Stopped := 1
end ;
-- Copy slaves into master variables
ACC := ACC_slave || CI := CI_slave
```

The updating assignments of `ACC` from `ACC_slave` and `CI` from `CI_slave` are shown sequential to the `case` command.

Notice that each of the SUB and JRP instructions makes use of a dyadic operator (subtract and add in these cases) and that the LDN instruction negates `MDR` before assigning it to `ACC_slave`. The addition `CI + MDR + 1` used by the JRP instruction could be optimised from two addition BinaryFunc components to just one adder with a carry-in of 1. All of these operations could be replaced with a shared subtract procedure. For example the the expression $CI + MDR + 1$ used in the JRP instruction can be rewritten as $CI + (\sim (\sim MDR)) + 1$[1] which is equivalent to $CI - (\sim MDR)$ which can be performed by two passes through a single subtracter ($MDR' := FFFF_{16} - MDR$ followed by $CI - MDR'$). Such a substitution would make an excellent area optimisation, removing at least one 32b adder structure from the original design.

6.1.5. Compilation and simulation

This completes the description of the SSEM with a number of language level optimisations (the use of shared procedures and removal of *auxiliary variables* by the use of explicit 'slave' variables). This design can be compiled by balsa-c into handshake components and then by breeze2lard to LARD and so simulated with LARD. The handshake circuit implementation of this example (in Breeze) and the full Balsa source are given in appendix A. This could

[1]The prefix operator ' ~ ' is used here to indicate the bitwise complement of the argument. This is the ' ~ ' operator from the C language.

be further improved upon by sharing a subtractor for all the instruction types. The compiled circuit can now be fed through breeze2lard and be simulated with LARD.

6.1.5.1. Compilation times

Balsa-c compilation of this example takes 20ms (on a Pentium II-266, times are not dissimilar for the Sun Workstation binary), execution of breeze2lard to produce a LARD design takes a further 360ms and the compilation of that LARD into a byte-code file takes 1.4s (this being the most complicated step). For a complete compilation from the Balsa file and a LARD test harness and the LARD linking of design and test harness into a single byte-code file takes 6.3s. These very short compilation times make the use of LARD as part of a compile – simulate – debug development method a practical proposition.

6.1.5.2. The test harness

For the SSEM the test harness consists of a LARD model of a simple 32 word memory module with similar ports to the SSEM description. The memory is loaded from a string array before invocation of the SSEM procedure. The main LARD expression of the test harness is:

```
(
  breeze_SSEM (c_activate, c_mem_addr, c_mem_rNw,
    c_mem_R, c_mem_W, c_halted) |
  Memory (c_mem_addr, c_mem_rNw, c_mem_R, c_mem_W,
    c_halted) |
  ( c_activate ! null )
)
```

This joins memory and processor together and also provides the activation for the processor by way of the output command with a `null` argument. Within the LARD environment we can see writes to and reads from memory as channel transactions.

6.1.5.3. Simulation

A small SSEM program is used to demonstrate this model. The program consists of eight instructions and three constant words and performs the complicated task of counting from 1 to 4. In its entirety the program is:

```
        org 0
loop:   data 0    ; Target address for JMP loop

        org 1
        LDN sum   ; ACC := -M[sum]
        SUB one   ; ACC := (-M[sum]) - 1
        STO sum   ; M[sum]' := (-M[sum]) - 1
        LDN sum   ; ACC := -M[sum]'
        STO sum   ; M[sum]" := -M[sum]'
                  ; => M[sum]" := M[sum] + 1
        SUB four  ; ACC := M[sum]" - 4
        TEST      ; skip if ACC < 0
        STOP      ; stop if ACC >= 0
        JMP loop  ; CI := M[loop]

        org 16
one:    data 1    ; constant 1
sum:    data 0    ; start sum at 0
four:   data 4    ; iteration count
```

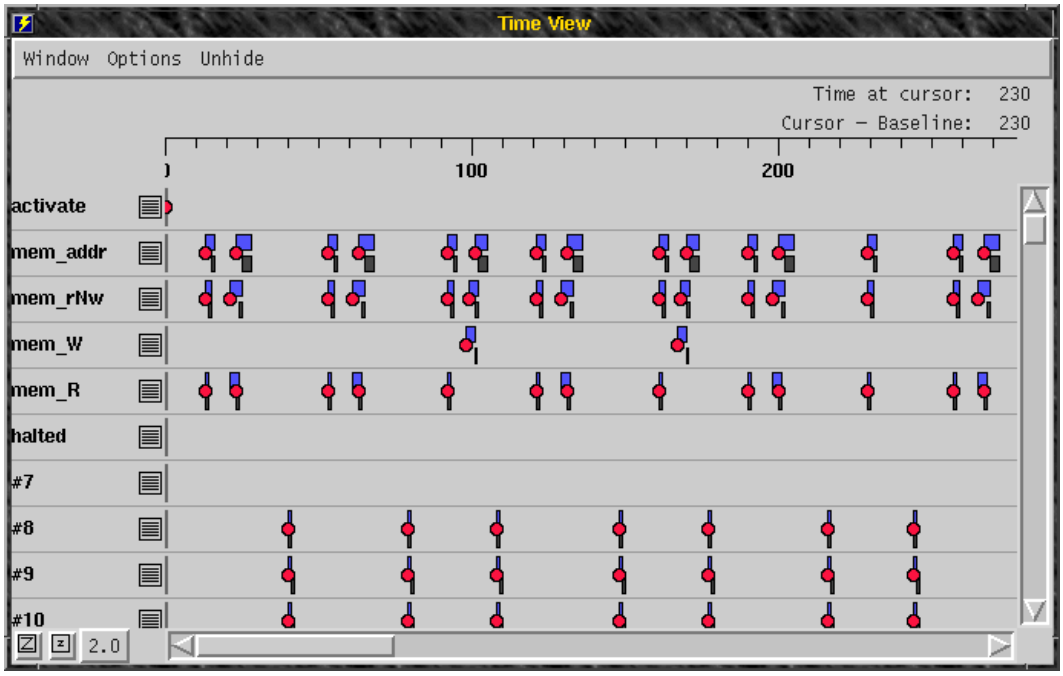


Figure 6.2. SSEM test program running in LARD

The processor has no direct add instruction or un-negated load and so this convoluted method must be used to ensure that the result is correct before the termination condition ($ACC = 4$) is

evaluated. The running program (in the LARD channel viewer) is shown in [fig. 6.2]. Only one loop iteration is shown, the paired memory writes associated with the two STO instructions per loop iteration can be clearly seen (on channel mem_W) as can the paired instruction / operand memory reads (on mem_R).

The speed of the running simulation remains a problem. This small example takes 32s to run on the same system on which it was previously compiled. This may not be an unacceptable time to wait for this simulation but at only one instruction per second for such a small design this is an unacceptably slow simulation environment. This slowness is in the main due to the inefficient data encoding presently used in Balsa to LARD simulations (the boolean arrays used for each channel's bundled data lead to UnaryFunc and BinaryFunc components being implemented in bit serial fashion from loops iterating over those arrays). The number of channels generated by a handshake circuit description also slow down simulation, especially when all channels are traced as is the case here.

6.2. A simple 16b RISC, STUMP

STUMP[1] is a very simple 16b RISC microprocessor used to teach the principles of microprocessor design and implementation as well as the hazards of non interlocking pipelined processors. The initial design for STUMP was devised by the author as part of an undergraduate project and augmented by Doug Edwards in its transformation from toy design to teaching tool [9]. The synchronous STUMP has:

A three stage pipeline

Fetch, execute and result write back

Three 16b fixed length instruction formats

$R_{dst} := R_{src1} \odot R_{src2}$, $R_{dst} := R_{src1} \odot$ *Immediate* and the branch instruction format

[1]The name STUMP is meant to be a pun on the name ARM (a STUMP being a cut down ARM). The name is misleading as STUMP is not based on the ARM design.

Nine instructions

ADD, ADC, SUB and SBC (addition and subtraction both without and with carry input), AND and OR (logical operations), LD and ST (memory load and store) and B (PC relative branch with 8b offset).

Six general purpose 16b registers

R1..R6, R0 is fixed to the constant 0, R7 is the program counter available as part of the register bank)

A single 4b condition code

N, C, V and Zs flag in the CC register which is conditionally loadable by ALU instructions.

A single ALU

Inline, one place, shift and rotate operations for register to register operations.

Single cycle instruction pipelined execution

A CPI of 1 for all instructions except LD and ST which take 2 cycles.

The three instruction formats are shown in [fig. 6.3]. The INST field codes for the 6 ALU instructions (ADD .. AND), memory operations (LD and ST have the same instruction number, LD being distinguished from ST by the value of the LDCC flag) and the branch instruction. A suitable Balsa enumeration type would be:

```
-- InstructionType: STUMP instruction encodings
type InstructionType is enumeration
  ADD, ADC, SUB, SBC, OR, AND, LD, ST=LD, B
over 3 bits
```

The LDCC bit is set for an ALU operation where that instruction should update the condition code based on its result. ALU operations either combine two registers to produce a result or a register and a 5b signed immediate. For non-immediate instructions the remaining two bits (5b immediate field less the 3b register field, the field marked ROT in the instruction encoding) specify an inline rotate operation for the second register before the ALU.

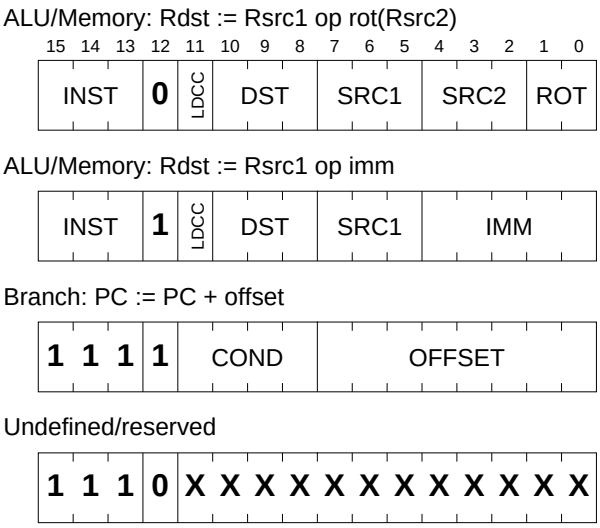


Figure 6.3. STUMP instruction formats

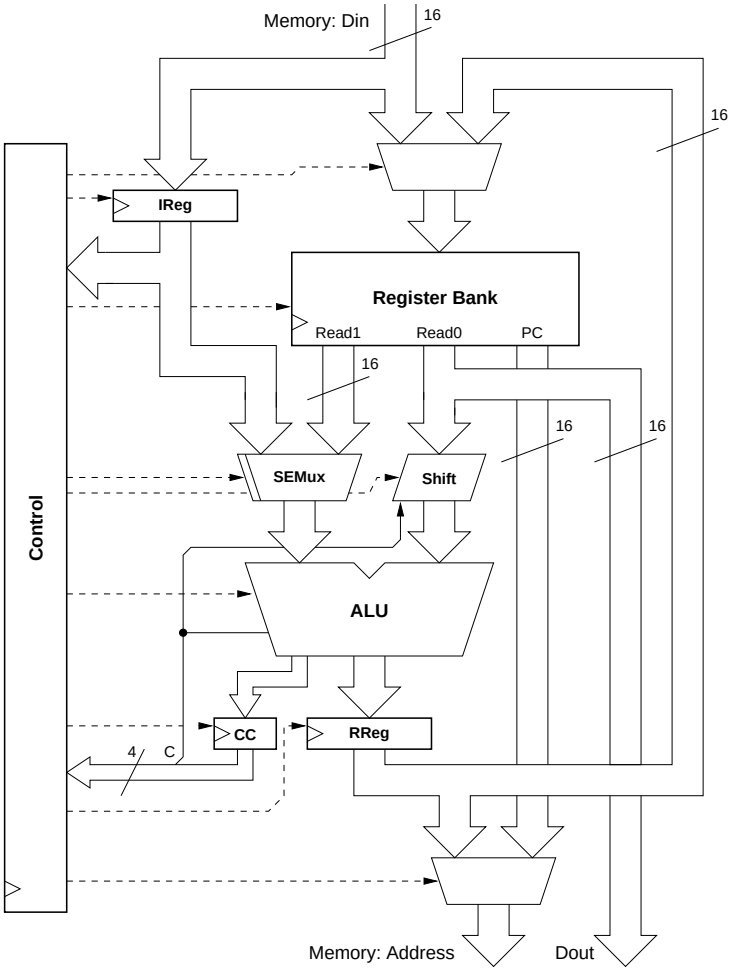


Figure 6.4. Synchronous STUMP organisation

The top level arrangement of the synchronous STUMP is shown in [fig. 6.4], units which have state holding (latch) behaviours are indicated by an edge sensitive clock input. The functions of the units named in the figure are:

Control

The control unit provides control signals to all other units and administers the overall state of the machine.

Register Bank

The register bank provides the persistent storage of the general purpose registers R1..6 and the program counter PC / R7. The program counter incrementer also lives in the register bank. Each STUMP instruction may write to at most one register and read from two others. For this reason the register bank has a single write port and two read ports in order to allow an instruction to complete within one cycle. During the execution of an instruction a subsequent instruction may be fetched from memory by using the dedicated PC read port (marked PC on the figure) as an address source for an external memory read.

IReg

The I register, IReg, holds the current instruction being executed. To be exact, this is the instruction for which an ALU operation is in progress during any particular cycle. Those portions of the IReg register's instruction which need to be carried over to the result write back phase of instruction execution are kept in the control unit.

SEMux

SEMux is the sign extending multiplexer. Its task is to multiplex between read port 1 of the register bank and the immediate field from the current instruction held in IReg. This multiplexing represents the choice between register and immediate operands for the two general instruction formats. The sign extending nature of the multiplexer is necessary as the 5b immediate field in register – immediate instructions is signed, the multiplexer also handles the introduction of branch offsets (which are 8b wide and signed) for branch instructions. Branch instructions make use of the ALU to calculate the new PC value by adding the branch offset to the current PC.

Shift

The shifter in line with the ALU allows register – register instructions to incorporate an optional one place shift or rotate. The shifter options are: No shift, one place right shift, one place right rotate, one place rotate right through carry. Left shifts can be effected by adding a register value to itself.

ALU

The ALU performs the general arithmetic and logical operations specified by the current instruction. Four condition bits are produced by the ALU's operation.

CC

The Condition Code register, CC, holds the condition bits output by the previous ALU operation for which storage of the condition was requested (by setting the LDCC flag in the instruction).

RReg

The Result Register, RReg, holds the result of the previous ALU operation prior to result write back to the register bank. The main role of this register in STUMP is to (slightly) speed up the register operand fetch – ALU operation – result write back path by splitting it over two cycles. The register's other role is to highlight the problems of instruction dependency handling in a pipelined STUMP implementation without register forwarding (this role being part of STUMP's duty as a teaching tool).

An asynchronous STUMP has a similar pipeline structure but allows the designer more freedom in the positioning of storage elements. The Balsa implementations of the register bank, ALU and sign-extend multiplexer units will be described here.

6.2.1. The register bank

The STUMP register bank contains six general purpose registers and the program counter. Its functions are to allow read and write access to all of the registers and also to optionally increment the program counter. This can be performed by the Balsa description:

```
type Word is 16 signed bits
```

```
type Register is 3 bits -- Register specifier

-- RegisterControl: control word for register bank
type RegisterControl is record
  Read0 : Register;
  Read1 : Register;
  Write : Register;
  DoRead0 : bit;    -- if true then ReadPort0 <- R[Read0]
  DoRead1 : bit;    -- if true then ReadPort1 <- R[Read1]
  DoWrite : bit;    -- if true then WritePort -> R[Write]
  DoIncPC : bit     -- if true PC := PC + 1
end

procedure STUMP_RegisterBank (
  input control : RegisterControl;   -- Single word
  input WritePort : Word;            -- Register write &
  output ReadPort0, ReadPort1 : Word -- read ports
) is local
  variable R : array 1..7 of Word
begin
  loop
    select control then
      local
        shared DoRead0 is begin
          if control.Read0 = 0
          then ReadPort0 <- 0
          else ReadPort0 <- R[control.Read0]
          end
        end

        shared DoRead1 is begin
          if control.Read1 = 0
          then ReadPort1 <- 0
          else ReadPort1 <- R[control.Read1]
          end
        end

        shared DoWrite is begin
          select WritePort then
            if control.Write /= 0
            then R[control.Write] := WritePort
            end
          end
        end
      begin
        if control.DoRead0 then DoRead0() end ||
        if control.DoRead1 then DoRead1() end ;
        if control.DoIncPC then
          R[7] := (R[7] + 1 as Word) end ;
        if control.DoWrite then DoWrite() end
      end
```

```
      end
    end
  end
```

This describes a behaviour where a single control word is read, two conditional register bank reads are performed followed by a conditional PC increment followed by a conditional register write. The sequencing of reads and writes allows us to combine these operations in the same 'cycle' in the same way as the synchronous implementation does. The PC increment function precedes a write operation to allow the `IncPC` field of `control` to be strapped to a 1 and still perform correct writes to the register bank's `R[7]` (the PC, these commands must be sequenced to prevent interference when writing `R[7]`).

Further points of note are the use of `select` to enclose the whole read, write, PC update action within the `control` communication. One side-effect of this is the requirement that the `DoRead0`, `DoRead1` and `DoWrite` be local to that `select`'s body command in order to be able to read the `control` word's fields.

The `WritePort` port also makes use of `select` although for a different purpose. Instructions which write their results to R0 in STUMP do so to discard that result (as R0 permanently reads the value 0), a communication on `WritePort` must still take place for this discarding action to take place and so the easiest way to accomplish this is to enclose the condition for the register write within a `select` guard communication. The check for a 0 index into the register bank is now enclosed in the `WritePort` read and so a value will be read from that port and either discarded (where the `control.Write /= 0` test fails) or written into the register bank (with the assignment `R[control.Write] := WritePort`).

The complexity added by the special treatment of R0 breaks the orthogonality of the array read and assignment commands. This can be remedied by the inclusion of array construction expressions in a future version of Balsa (this is part of the author's current work), the shared procedures `DoRead0`, `DoRead1` and `DoWrite` then become:

```
shared DoRead0 is begin
  -- Build an array including the constant 0
```

```
  ReadPort0 <- {0, R[1..7]}[control.Read0]
end

shared DoRead1 is begin
  ReadPort1 <- {0, R[1..7]}[control.Read1]
end

shared DoWrite is begin
  WritePort -> {_, R[1..7]}[control.Write]
end
```

The '_' in the array construction for `DoWrite` would be a special symbol for a 'null' write terminator (in implementation, a ContinuePush component). The inclusion of array construction will save the inclusion of Case components for each of the conditions in the shared procedures and also allow `WritePort` to be implemented using an active, pull port.

6.2.2. The ALU

The ALU has both the simplest handshake behaviour and the most complicated internal function of the units of STUMP. The structure of the ALU is:

```
-- ALU control word
type ALUControl is record
  Operation : InstructionType;
  WriteCC   : bit; -- Output a CC value
  UpdateCC  : bit -- Update internal CC
end

-- CC register bits
type CCType is record
  C, V, Z, N : bit
end

procedure STUMP_ALU (input control : ALUControl;
  input SrcA, SrcB : Word; -- Arguments
  output Result : Word;
  output CC : CCType       -- Result and flags
) is local
  variable result : Word
  variable cc : CCType
begin
  loop
    select control, SrcA, SrcB then
      -- Perform ALU op into result and cc
```

```
        Result <- result ||
        if control.WriteCC then CC <- cc end
      end
    end
  end
```

The local variable `result` is used to simplify manipulations of the ALU result required to generate the flags. This could be replaced by a number of output channel communications on `Result`. The CC register is local to the ALU in this example (as this is an easier arrangement), the shifter must receive its carry input directly from the control unit which is connected to the CC port of the ALU.

The ALU can perform four basic operations:

Addition Used for ADD and ADC instructions as well as branch target address calculations and LD / ST source / target address calculation. A 16b adder with a carry input will perform this operation.

Subtraction Used for SUB and SBC instructions, the same adder as is used for add operations may be used with the addition of a rank of inverts on the right hand inputs.

AND and OR The logical operation must be carried out with separate hardware from the arithmetic operations.

Using internal channels to communicate we can describe these four operations so:

```
-- Adder types
type Adder_1_16 is record
  f0 : bit; f1 : 16 signed bits end
type Adder_1_16_1 is record
  f0 : bit; f1 : 16 signed bits; f2 : bit end
type Adder_16_1 is record
  f0 : 16 signed bits; f1 : bit end
…
-- Retype result variable
variable result : Adder_16_1
variable tempV : bit -- Temporary overflow
variable tempC : bit -- Temporary carry
…
```

```
case control.Operation of
     AND then result := {(SrcA and SrcB as Word), 0}
  || tempV := 0
also OR  then result := {(SrcA or  SrcB as Word), 0}
  || tempV := 0
else -- ADD..SBC,LD/ST,B
  local
    channel AddArgs : Adder_1_16
  begin
    case control.Operation of
         ADC then AddArgs <- {cc.C, SrcB}
    also SUB then AddArgs <- {1, not SrcB}
    also SBC then AddArgs <- {not cc.C, not SrcB}
    else          AddArgs <- {0, SrcB}
    end
  ||
    select AddArgs then
      result := (((Adder_1_16{AddArgs.f0, SrcA}
        as 17 bits) +
        (AddArgs as 17 bits) as array 0..17 of bit)
        [1..17] as Adder_16_1) ;
      tempV := -- Omitted (complicated)
    end
  end
end ; -- Now set the flags
if control.UpdateCC then cc := {0, 0, 0, 0} end ;
-- Write result
Result <- result.f0
```

The four separate `AddArgs` communications and the inversion of `SrcB` for the two subtraction operations can be combined in post-optimisation. Notice that the structure of `case` commands can be rearranged to produce different variable, sequencing and internal channel combinations. It is possible that a different organisation is faster than that given, rearranging the description manually to find that combination is the basis of the transparent compilation, describe – simulate – refine development loop.

Notice that it is necessary to use a fair number of types in this complicated bit manipulating operation. The creation of a carry in to a simple 16b adder by the inclusion of a 17^{th} least significant bit should be reduced in gate level optimisation from a single bit adder with both inputs connected together to a carry in connection to the next adder in the addition chain.

The flags cannot be set from the `result` register (with the exception of the overflow flag which can be derived from the adder inputs and result). The flag update is omitted here as it

is particularly complex. The descriptive style of Balsa is not well suited to building units such as the ALU where the required composition of gates is known but the description is hampered by the difficulties of describing this with handshake components.

6.2.3. The sign extending multiplexer – SEMux

The sign extending multiplexer is one of the simplest units of this design. A choice between register bank and instruction register value is made by a `select` command. The instruction register's value must, therefore, be *pushed* into the SEMux. This can be achieved by making the IReg live in the control block as a Balsa variable, a write of this value to the SEMux input port can be made where either a branch or a register – immediate instruction is executed. The least significant eight bits of the instruction along with a single bit tag informing the multiplexer whether this is a 5 or 8 bit immediate is also supplied. The Balsa description would be:

```
-- Instruction immediate field type
type ImmediateField is 5 signed bits

-- Immediate control word
type ImmediateControl is record
  Imm : 8 signed bits;
  BranchNImm : bit -- 0: 5b immediate, 1: 8b offset
end

procedure STUMP_SEMux (input Imm : ImmediateControl;
  input Reg : Word;
  output Out : Word
) is begin
  loop
    select Imm then
      if Imm.BranchNImm then Out <- (Imm.Imm as Word)
      else Out <- (((Imm.Imm as array 0..7 of bit)[0..4]
        as ImmediateField) as Word)
      end
    also Reg then
      Out <- Reg
    end
  end
end
```

This is an excellent example of the use of type casting to achieve sign extension. Consider the case of the 5b immediate field where the `Imm.Imm` 8b field is sliced down into the 5 least significant bits before being cast twice, first into a 5b **signed** numeric type and again into a full **signed** word. A cast from a signed numeric type to another (wider) signed numeric type has the effect of sign extending that narrower value into the width of the wider, target, type.

This is also in many ways a rather contrived example as the same behaviour could be achieved by drawing the register bank read value through the control block and making a decision about the inputs to the ALU there. This would require a greater degree of control sequencing for those instructions with register – register operands (with the SEMux the register read value on read port 1 would flow straight through to the ALU without interaction with the control block).

6.3. Chapter summary

Example designs of two microprocessors with differing design styles have been given. The SSEM description has been compiled into Breeze, transformed into a structural handshake component model in LARD by the script breeze2lard and simulated under the LARD environment. This is as close to a proof of principle as can be provided at the current time. The completion of the synthesis route into gate level VLSI using breeze2gates is currently work in progress.

The choice of two microprocessors as test designs is deliberate. Philips have until recently only demonstrated the applicability of Tangram to ASIC applications, most notably the implementation of error correcting decoding units for CD and DCC applications. In implementing a microprocessor it is the intention of the author to explore the applicability of the handshake circuit approach to the design of programmable controllers. Handshake circuits are currently aimed at the design of low power, low to medium speed circuits. It may be the case that small custom microcontrollers are a possible application area of interest. Tangram has recently been used to implement a self-timed version of the Intel 8051 microcontroller [19]. This design has a very simple control driven sequential nature similar to that used in the SSEM description covered in this chapter. The SSEM also demonstrates the description

of an accumulator based microprocessor. A description of a specialised processor such as a programmable DSP would have a similar sequential, accumulator based nature. The STUMP description demonstrates simple uses for the enclosing input selection mechanism in structural, data driven designs.

Chapter 7. Conclusions and Further Work

7.1. This thesis

Four major topics been discussed in this thesis:

- The description of the Balsa language and its additions to Tangram.
- The implementation of tools to compile Balsa to handshake components using an extended Tangram compilation function.
- The implementation of new handshake components introduced by the extensions of Tangram and by the modification of the target handshake component set.
- The demonstration of a first-cut simulation model generation tool, breeze2lard.

7.1.1. The Balsa language

As a design description language, Balsa has the majority of the features of Tangram with the addition of the parameterisation, separate compilation and input selection mechanisms described in §3.

The additions made by Balsa to Tangram serve two main purposes:

7.1.1.1. 'Normalising' Tangram

Tangram should be made more similar to other HDLs to increase its acceptability to the current users of those HDLs. Balsa has a more conventional type system for an HDL. Emphasis is placed on the representation of bits of information rather than ranges of integers. Numeric types cannot be described which do not include all of the 2^n possible values of their n bit realisations. Arrays and records are defined by a sequence of bits making up a word. Extraction from arrays and records is simply bitfield extraction from that word. Type casting is

the zero-padding or truncation of that word. No structural type casting operator exists in Balsa (cf. structural `cast` and bitwise `fit` casts of Tangram); the responsibility for structural type manipulation lies with the designer. Other relevant features are the sugary (somewhat VHDL like) syntax of Balsa and the inclusion of facilities to allow separate compilation. Separate compilation allows the designer to reuse parts of old designs in new projects. In practice this allows the designer to not only reuse Balsa source but also to make use of hand-designed or optimised units in new designs. The description of port structures of foreign components used in Balsa descriptions seems to be the most useful application of this feature.

7.1.1.2. Adding to the expressiveness of Tangram

Tangram is based on the CSP notion of atomic communication. Without considering signalling protocols of hardware handshake implementations the designer can reason about the communication behaviour of a circuit modelling communication as single point synchronisations. The use of activation channel communications to enclose actions in the definitions of handshake components and handshake circuits relies on a model of communication which includes enclosure between **two** events. Enclosure can be modelled in hardware by the exchange of events on a pair of handshake signals with the enclosed action performed between these two events. The data valid phase of many signalling protocols used in bundled data implementations can be made to partially or fully enclose the period of the handshake in which the behaviour associated with an enclosure occurs. The extension of the concept of enclosure to the source description language will thereby allow the designer to exploit the stability of communicated data and the control simplification of communication enclosed behaviours.

In short, allowing the enclosing `select` in Balsa allows the designer to describe procedures with enclosing and unbuffered behaviours which are not possible in Tangram. This in part makes up for the lack of data driven control in Tangram but does tend to lead to descriptions which are more structural in nature than the designer might have intended as explicit communications and channels tend to replace the implicit channels of a Tangram expression. An enclosing `select` command does also make the implementation of the Tangram / CSP sequenced (guard communication ; guarded command) selection scheme more difficult. The example given in §3.2.2.4 shows a two way Tangram like selection built using an enclosing

selection and with explicit sequencing. The description of the gate level optimisations possible for this circuit given in that section yields a circuit which is only slightly larger than the implementation of the similar description in Tangram. The increased expressiveness that enclosure allows and its ability to describe the Tangram selection mechanism make it the mechanism of choice of Balsa.

Other notable features added to Tangram by Balsa are:

Arrayed channels (§3.1.5.2)

The ability to describe a port structure which consists of several separate ports indexable in the same way as an array.

Separate compilation (§3.1.5.1)

A simple file-based module system allowing separate compilation of separate parts of a design. Closer integration with a CAD system may also be achieved by linking the Balsa code organisation with that of the CAD system's database.

Structural iteration commands (§3.1.5.4)

The `for ||` and `for ;` commands allow the user to describe the repeated placement of hardware in a similar way to the `for` … `generate` … command available in VHDL. Future additions to Balsa may add parameters to Balsa procedures and make this structural iteration facility available to `if` … `else` structures and `select` commands.

7.1.2. Compiling Balsa

Balsa shares the same core compilation mechanism as Tangram. The use of handshake circuits as an intermediate form allows the separation of the compilation process into two stages:

Front end – User visible language to HCs

The source language (Tangram or Balsa) is translated into a network of channel communicating handshake components. This compilation phase is, for the most part, independent of the details of the final implementation technology such as the signalling mechanism and data encoding used to realise channels. Optimisations may be considered at both the language level (such as constant folding or dead code removal) and at the handshake component level (typically peephole optimisation of control structures). The balsa-c compiler is an example of a front end HC compiler.

Back end – HCs to target technology

A VLSI implementation of a handshake circuit is produced. Specific details of the implementation technology are considered and optimisation can be carried out in the light of these details. A simple back end may treat gate level descriptions of the handshake components may be treated as macros for the creation of whole handshake circuits. A more complicated tool may attempt to re-synthesise parts of the design (typically the control path) for better area, speed or power performance. Such a back end compiler is planned as part of the final Balsa system.

The primary difference between the syntax directed approach that Balsa shares with Tangram and that of more conventional, state-space exploration (and particularly synchronous) synthesis techniques is the 'connectedness' the designer feels with the design refinement process. Explicit user intervention in a rapid design – compile – simulation – re-engineer cycle allows the user to make their own trade off decisions. Less direct methods and in particular the dominant VHDL / Verilog functional synchronous synthesis route emphasise the automated nature of synthesis and the production of a number of implementations at a number of points in the speed / area trade off solution space. In an ideal world, such tools would respect the designer's descriptive style and produce a selection of well optimised designs in a timely manner. The reality is of slow tools producing good implementations for designs made in their favoured descriptive style. Design thereby becomes a design – synthesise – simulate (or assess the performance of the design in some other manner) – re-engineer design to fit in with the synthesiser's preferred style cycle.

Syntax directed compilation may at first seem like an abandonment of automation,

an admission of defeat. It should be seen that this is not so and that a direct approach simply emphasises the role of the designer in the transformation of a description into an implementation. In learning the 'tricks' of a non-direct synthesiser the designer is as much seeing a direct mechanism as if a syntax directed method was used all along.

7.1.3. The Balsa handshake component set

A number of additions are made to the component set used by Tangram. The DecisionWait and FalseVariable components (whose uses are detailed in §3.2.3 and gate level implementation are given in §5.1.1.3) are introduced to allow the user to describe unbuffered input selection. The power of the Case component is greatly increased and its implementation (detailed in §5.1.1.4) is generalised to reduce the need to re-synthesise CASE tree structures during the back end phase of compilation.

7.1.4. Simulating Balsa with LARD

Simulation of Balsa designs is by translation of intermediate handshake circuits into the modelling language LARD. The tool breeze2lard was described in §6 as a means for producing valid LARD from Breeze netlists and the simulation of compiled Balsa designs using a library of parameterisable handshake component descriptions written in LARD. LARD is the simulation engine of choice due to its wider use within the AMULET3 project as a high level modelling and validation language. For future projects a version of breeze2lard is in development which will translate a handshake circuit netlist from Breeze into a less structural LARD description. This will be achieved by traversing the activation tree of the given handshake circuit and placing native LARD commands and control compositions in place of handshake component instances.

The lack of a gate level targeting back end for Balsa limits the ability to make direct comparison between Tangram and Balsa. The use of the Tangram compilation function should result in very similar handshake circuit descriptions from the two tools. The use of the enclosing `select` in Balsa descriptions will hopefully yield smaller (due to the removal of latches) and faster (as enclosure reduces the need to sequence communications with actions and also removes the passivation required when connecting active ports) circuits than those

described in the Tangram style.

It is hoped that this lack of concrete results will be rectified in due course. Two major design exercises are planned with which to compare Balsa with other approaches. These designs are:

A 16b Digital Signal Processor – stDSP

The stDSP is a self-timed DSP core designed by Cogency Technology Ltd. [34]. The architecture of stDSP is almost identical to that of an existing synchronous DSP core (for which stDSP is a drop-in replacement) designed for low-power and high electro-magnetic compatibility applications. A Balsa synthesised version of this core could, therefore, be directly compared with both hand-designed asynchronous and synchronous implementations.

A Direct Memory Access controller for the AMULET3

The AMULET3 is the third asynchronous implementation of the ARM architecture to be undertaken by the AMULET group of The University of Manchester. AMULET3 will take the form of a macrocell to be included in a single-chip solution by a commercial partner. This chip will include a number of synchronous peripherals, the AMULET3 macrocell and a DMA controller. The DMA controller will handle the majority of transactions made between the synchronous peripheral and the asynchronous macrocell (communicating across an asynchronous on-chip bus: MARBLE [41]). In order to reduce design time and increase flexibility (to accommodate an evolving specification) it has been decided to undertake this design using Balsa.

It is hoped that the implementation of these designs will yield convincing quantitative (in the case of the DSP, speed / area / power comparisons) and qualitative (ease of use / suitability of the descriptive style to practical designs) measures of the success of Balsa as an HDL.

7.2. Related Work at Manchester

Balsa is a synthesis orientated language, the key tools under consideration being those required to realise a standard cell implementation of Balsa designs (specifically balsa-c and breeze2gates). This work should be contrasted with two other asynchronous circuit

description / modelling projects being carried out within the Department of Computer Science at The University of Manchester. These are LARD [35] and Rainbow [16][18][3].

7.2.1. LARD

LARD was described in §5.5 as a simulation environment for handshake circuits. LARD's original purpose remains the high level modelling of asynchronous designs, specifically the architecture of AMULET3 [11][12]. High level here refers to models in which the individual inter-module communications are revealed but internal function (along with sequencing) is modelled in a 'programming language' style. Programming language features such as composite typing, type polymorphism and user defined operators are all available.

LARD allows descriptions of circuits in which Micropipeline-like communications are used (push only communications with the possibility of enclosure of a behaviour within an input communication). The communication operators present in LARD are not primitive to the language and are, in fact, built on the use of shared data between threads being scheduled by a time-sharing virtual machine. The pull handshake behaviour required of the handshake circuit simulations described in §5.5 were implemented using an add-on channel communication library built on the same principles. This flexibility allows the rapid development of designs by refinement from high level model (which is unsynthesisable) to final design (which may closely resemble the final implementation) without changing language or tools. A further benefit of this flexibility is the ability to write high level test harnesses for LARD models **in** LARD.

Compiled LARD is represented by a byte code intermediate form being executed by a virtual machine implemented in software (in much the same way as the languages Smalltalk and Java are byte code interpreted). Emphasis is placed on the speed of simulation of designs involving small numbers of significant communications (as the reliance on polling of shared data makes the fine-grain use of channels inefficient).

It is unlikely that a synthesiser for LARD will be written. Defining a small enough subset of the language for which correct synthesis can be ensured would be a not insignificant task.

7.2.2. Rainbow

The Rainbow project is an attempt to create an inter-working set of languages for the description and reasoning about Micropipelined asynchronous designs. This work is being carried out by the Asynchronous Hardware Verification Group at Manchester. The principal languages currently in development are Green and Yellow.

Green [14] is a dataflow-style, pipeline construction language and as such has both visual and textual forms. Green is typically used to describe that part of design which is simply the composition of push pipeline elements and pipeline buffering stages. The storage of persistent state and the control sequencing of activities is difficult in Green which concerns itself mostly with the manipulation and direction of data.

Yellow on the other hand is a CSP-like language describing control flow in an explicit manner with communication primitives being important for synchronisation and data movement. As with Green, only push behaviour is considered within Yellow (although the push / pull nature of channels and the signalling and data validity protocols operating on those channels remain hidden to the user as with Balsa) although Yellow is capable of describing sequenced behaviour.

Yellow is in many ways very similar to Balsa with Green acting as a convenient form for representing data driven pipeline behaviours. Although no compilation mechanism currently exists for Yellow (compilation of Green being straight forward) the language may lend itself to a handshake circuit implementation.

Rainbow's mandate has been and remains the formal reasoning about the properties of asynchronous digital circuits. The Rainbow languages are a means to this end and as such each language has carefully defined semantics. Any description in any of the Rainbow languages (Green and Yellow as well as the yet-to-appear languages Red and Blue) can be reduced to a process algebra description from which reasoning about the properties of that description can be attempted.

7.2.3. Justifying three projects

The concurrent development of Balsa, Rainbow and LARD should not be considered to be poor organisation on the parts of the research groups involved. Each project represents a different approach to the problem of constructing asynchronous systems and although each project includes a means for capturing a description (its language(s)) the goals of the three projects are sufficiently different to make each a viable proposition.

The emphasis of future work on the development of Balsa will remain with synthesis.

7.3. Other approaches to CSP synthesis

A number of other approaches have been documented for the automated synthesis of asynchronous systems from CSP like descriptions, notably those of Burns [36] and Brunvand [15]. The main differences between these approaches and that of Tangram and Balsa are related to:

- Target technologies – gate level vs. modular.
- Preferred port senses for each of input, output and sync ports and the means for composing port connected groups of modules.
- Treatments of input selection and arbitration.
- Choices of and abstraction away from signalling protocols / data encoding schemes.
- Where optimisation or refinement of the description is performed – manipulation of the description vs. gate / module level optimisation.

7.3.1. Burns: Automated Compilation of Concurrent Programs into Self-timed Circuits

Burns describes an automated method for compiling descriptions in a form similar to Martin's CHP to gate level circuit descriptions. A number of predefined modules are used to simplify the target descriptions. These include the Q-element which is similar to the

S-element described by van Berkel, the ‘S’ component which is a one-hot data encoded version of the micropipeline select element and the ‘me’ component which is a level sensitive mutual exclusion element. Each of these modules (with the possible exception of the mutual exclusion element) can be implemented with just the conventional combinational logic gates, the C-element and the use of isochronic forks.

7.3.1.1. Channel primitives: Bidirection channels and the Probe

The signalling mechanism used is 4 phase and involves the use of one-hot encoded data giving rise to $2^n + 1$ wires for each n bit wide unidirectional channel. The communication notation used allows for bidirectional communications by the inclusion of multiple ‘acknowledge’ wires. The common request / acknowledge terminology is not used, a bidirectional channel is simply a $2^n + 2^m$ wire channel with n bits flowing in one direction and m bits flowing the opposite direction. A 4 bit passive input port may be represented:

passivechannelName(16, 1)

This denotes a passive port with 16 input signals and 1 output signal (the acknowledgement). The distinction between active and passive ports is made at the language level allowing the flexibility to choose between push and pull directed design styles. Tangram and Balsa descriptions do not allow such distinctions as the Tangram compilation mechanism favours an ‘all active’ approach to the implementation of normal (non selecting) communications. The distinction is made in the handshake components. Handshake component push and pull unidirectional channels can be explicitly described by pairs of active and passive ports of degrees (n, 1) and (1, n) for output and input ports respectively.

Within the body of a process, bidirectional communication operations are considered to be atomic, with the exchange of data (or data for acknowledgement) occurring without the provision for the inclusion of an enclosed command. This fits in well with the atomic communication semantics of CSP but does not allow the description of handshake enclosed behaviours without the addition of a notable feature of both Burns’ notation and Martin’s CHP: the probe. The probe allows the description of enclosing and communication selecting behaviours by allowing the readiness of a channel to perform a communication to be the subject of a guard expression. Take the example from Burns which uses a similar notation

to that of the handshake process calculus described in §2 except that $\overline{D}$ signifies a probe on channel D, D alone signifies a communication on D and *[ ] is the unbounded repetition operator:

$$*\ [\ [\overline{D} \rightarrow S_0'; S_1'; \ldots; S_{n-1}'; D]\ ]$$

Here a probe on channel D forms a guard expression for the action $S_0'; S_1'; \ldots; S_{n-1}'; D$. A probe expression is not a predicate in the usual sense as the probe's function is to delay the satisfaction of the guard until the channel D is ready to communicate. In this example the probe on D guards the sequenced execution of S_0' to S_{n-1}' followed by a communication on channel D. If D is connected to a passive sync port in this process (of the form *passiveD*(1,1)) then this expression can be seen to be equivalent to this expression in the handshake process calculus:

$$\#[\ D : (S_0'; S_1'; \ldots; S_{n-1}')\ ]$$

This is of course the behaviour of the Sequence component in the Balsa handshake component set.

By the use of the guard we can express communication selection by allowing more than one guard in a guarded choice expression to include a channel probe e.g:

$$*\ [\ [\overline{A} \rightarrow C; A\ \mid\ \overline{B} \rightarrow C; B]\ ]$$

This describes a Call component enclosing a communication on C by either an initiating communication on A or B. Again the use of the probe in a guard expression changes the meaning of the whole guarded expression. If A becomes ready to communicate so leading to the satisfaction of the guard $\overline{A}$ and the execution of the expression $C; A$ the guard expression $\overline{B}$ effectively becomes false. The change of semantics which results in the use of the probe in guard expressions provides a good enough reason to disallow its use elsewhere (i.e. in general boolean expressions). In conjunction with normal predicates, guard expressions involving probes can be a powerful means of expressing input selection behaviour in a manner similar to the predicate guarded communication guards of CSP but whilst allowing the description of enclosing behaviours. Although a more powerful construct than enclosure, the probe can be used to express behaviours where the nesting of an expression within an enclosing

communication is not possible, for example:

$$*\,[\,A;[\overline{A} \rightarrow expr_1];expr_2\,]$$

describes the enclosure of $expr_1; expr_2$ within a communication on A where the initial action of the circuit is to perform a communication on A. In the handshake process calculus this could be expressed:

$$\#[\,A;A : (expr_1; expr_2)\,]$$

In this form the enclosure of the sequence $expr1; expr2$ is made explicit and is not allowed to straddle the end of an iteration of the enclosing unbounded repetition. The probe expression could equally be expressed as:

$$*\,[\,A;[\,\overline{A} \rightarrow skip\,];expr_1;expr_2\,]$$

or $$A;\ *\,[\,[\,\overline{A} \rightarrow skip\,];expr_1;expr_2;A\,]$$

This gives us several ways to express this simple enclosure, two of which are not obvious on first parsing. Restricting the use of the probe, or the requirement for a compiler to handle all its possible uses, may contribute to the complexity of such a compilation scheme. Burns considers probe to be essential to the compilation mechanism which includes handshake expansion and handshake re-shuffling as compilation phases expressing the act of waiting on an input signal using a probe. In the use of handshake components as a target technology the choice of signalling protocols and data encoding is deferred until the final stage of compilation (mapping into logic gates) making the signal level manipulation of circuit behaviour by the input description difficult if not impossible. Enclosure can be seen in this way to be an extension of the properties of the channel communication rather than an extension of the signal level descriptive abilities of the target technology into the input description as with the probe.

7.3.1.2. Input selection

The use of unary encoded data and the ability to predicate guard expressions on the value of incoming channel communications leads to an implementation of ‘channel value guards’

which is identical to that of selection of a number of incoming sync communications. Burns gives an implementation of such a passive input structure using *n* C-elements, an *n* way isochronic fork and an *n* input OR gate which is almost identical to the DecisionWait described in §5.1.1.3 as an addition to Tangram made by Balsa. The Balsa implementation differs however in its timing of the acknowledgement made to the initiating channel (the activation) and exploits the enclosure of the guarded command in the guard communication to allow the free reading of the value of the incoming channel communication (using a FalseVariable component) without incurring additional synchronisation, the cost of placing a Variable to receive the incoming channel's value or the cost of accounting for a possibly variable number of references made to the channel's value during the execution of the guarded command. This 'low cost read' property is not possible using the probe primitive without the inclusion of an explicit variable even where enclosure of the guarded command in the communication is possible.

7.3.1.3. Arbitration

CHP allows the designer to explicitly distinguish between instances of communication selection which involve arbitration and those in which the environment ensures the mutual exclusion of competing communications. This is a useful feature in a directly compiled language and so is carried over into Balsa in the form of the `select` and `arbitrate` flavours of input selection.

7.3.1.4. The Compilation Approach

The compilation mechanism implemented by Burns differs from the Tangram approach by its emphasis on a fixed target signalling protocol / data encoding and on its use of manipulation of terms in the input notation to refine towards an implementation. The direct compilation method used by Tangram defines simple mappings from input expressions to output handshake circuits and favours the use of handshake circuit level optimisations (typically peephole optimisation) to improve the quality of the generated circuit. In choosing to perform handshake expansion and handshake re-shuffling along with the general probe primitive Burns imposes a great responsibility on the compiler to perform correct manipulation of the input description. The considerably simpler Tangram compiler may be more easily

validated as the transformation from Tangram to handshake components is more transparent and relies heavily on the channel bearing nature of the handshake circuit descriptive form to preserve the notion of the abstract communication primitives of the source language into the target representation.

7.3.2. Brunvand: Translating Concurrent Communicating Programs into Asynchronous Circuits

A lisp-like form of the concurrent programming language OCCAM is the input language of Brunvand's compiler. OCCAM is for the most part a concrete syntax form of CSP with practical data typing and support for the composition of processes which communicate by channel connected ports. Brunvand targets 2 phase bundled-data implementations using a component set similar to that of the micropipeline design style. This includes the 2 phase micropipeline select, the transition arbiter, the call block, C-element, merge (XOR gate), the toggle, the Q-select (a modified form a select), a register element and the 'enable' module which allows the gating of an incoming channel onto a bus. The modules are implemented in standard cell logic gates with the bundling constraint being met by the use of long control paths and a good dose of luck in the layout phase.

7.3.2.1. Channel primitives: Unidirectional channels

The input notation does not allow the explicit designation of passive and active ports. In this sense port descriptions resemble Tangram. Synchronising communication ports are active in nature with synchronisation being formed by the use of a C-element joining requests from both ports in a communication to the acknowledgements of each port. This is identical to the use of Synch components to connect sync channels in Balsa, although Balsa channels may be extended to connect more than two ports by the placement of a multiple input Sync component. OCCAM considers a data-bearing channel to connect a single output to a single input and so Brunvand implements active outputs and passive inputs. Composition of port connected processes can be achieved simply by connecting port pairs together.

The use of passive input ports does not imply that such circuits are data / incoming request driven. The synchronising C-element present in the sync communication is hidden in the input

port structure of a data bearing channel. In this way the symmetry of the sync communication is preserved and the structure of connections of data bearing channels simplified. Tangram requires an additional external component (one, or a combination of: SynchPush, SynchPull and ForkPush) to form this synchronisation as the composition of processes connected by broadcast channels may involve more complicated synchronisations than are possible with the simple passive input port approach. To combine sequential writes to a single channel, Tangram makes use of the CallPush and CallPull components to multiplex data onto a common channel whilst steering control events. Brunvand separates this multiplexing into tristate bus driving Enable components in the output port structures themselves. Post compilation optimisation is used to remove these components where no multiplexing is required.

7.3.2.2. Input selection

The semantics of channel communication guarded selection is similar to CSP. The guard communication is sequenced with the command it guards. The statement:

(*Alt*
  ((*True*(?*chan*1))
    (*body*1))
  ((*True*(?*chan*2))
    (*body*2)))

is the basic two input selection. Brunvand recognises the difficulty of implementing selection where the selection command must be activated as part of a larger circuit. This problem is solved in Balsa by the use of the DecisionWait component (which is also similar to Burns' unary data encoded choice) to introduce an activation channel into the paths of the guard channels. Brunvand explores several possible solutions to this problem using a round-robin polling ring to select the first input channel to become available. This allows the implementation of an Alt (selection) command which chooses from the input channels in some predefined order. A fair form of Alt is also presented whose implementation has internal state which allows prevention of the case where one guard channel is serviced in preference to the others due to its position in the source description.

The description of enclosing behaviours is not possible in this notation. The use of repeated synchronisation on a sync channel is used by Brunvand in examples to give a similar effect. No distinction is made between arbitrated and unarbitrated forms of Alt.

7.3.2.3. The Compilation Approach

Emphasis is placed on the use of post-compilation optimisation to improve upon a generated circuit. This is the same approach used by both the Tangram and Balsa compilers. Handshake circuit netlists differ in their use of channels rather than signals to connect components. A handshake circuit optimiser must therefore be careful to ensure that no channel remains unconnected to a component or an external port. The use of Continue and Halt components makes this possible. The pre-optimisation circuits generated by Brunvand's compiler may have dangling signals which imply the need to prune components.

7.4. Future work

Future directions for work to complete the Balsa system include:

- Completion of breeze2gates, production of standard cell (and possibly FPGA) implementations of Balsa designs.
- Improvement of the simulation environment to increase speed and allow for 'source level debugging' by tagging simulation channel names with code positions.
- Work on the timing aspects of standard cell place & route. The timing constraints within an asynchronous circuit make it impossible to trust the untimed layout of gates by a conventional synchronous place & route tool. Either new layout tools will be needed or a systematic approach to post-layout timing verification must be developed.
- Introduction of parameterisable Balsa procedures in the same manner that Breeze components may be parameterised. Best use of the `for` command and arrayed channels can be made where procedures are allowed to take parameters and where those procedures can be written in an unparameterised form into an output Breeze file for later inclusion in other Balsa designs.

- improved array construction expressions and element extractions operators.
- a Breeze-like sublanguage for describing gate level structural circuits. This may be useful for hand-crafting designs such as ALUs to get around the lack of expressiveness in Balsa expressions without resorting to the excessive use of explicit local channels.
- debugging commands: VHDL like assertions and possibly I/O operations which are ignored for synthesis but useful in simulation.
- Creation of convincing example implementations allowing automatically generated Balsa designs to be compared like-for-like with hand designed asynchronous designs. Comparisons with synchronous implementations of similar designs may also be useful.
- The formal specification of the semantics of Balsa. This may allow Balsa descriptions to be transformed into a form suitable for ‘model checking’ with developing Rainbow tools.

References

[1] A. Bardsley. An Asynchronous Logic Synthesiser. Undergraduate Project Report, 1996.

[2] A. Bailey, M. B. Josephs. Sequencer Circuits for VLSI Programming. In *Second working conference on Asynchronous Design Methodologies*, May 1995.

[3] Rainbow project web pages, Department of Computer Science, The University of Manchester. URL `http://www.cs.man.ac.uk/fmethods/projects/AHV-PROJECT/ahv-project.html`.

[4] A. M. G. Peeters. *Single-Rail Handshake Circuits.* Ph.D. thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1996.

[5] Edited by B. Krishnamurthy. *Practical Reusable UNIX Software.* ISBN 0-471-05807-6. John Wiley & Sons. AT&T Research, 1995.

[6] AT&T Research. Practical Reusable UNIX Software. URL `http://www.research.att.com/sw/tools/reuse`.

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

[8] C. Farnsworth, D. A. Edwards, J. Liu, S. S. Sikand. A Hybrid Asynchronous System Design Environment. In *Asynchronous Design Methodologies*, May 1995. URL `http://www.cs.man.ac.uk/amulet/publications/papers/hybrid.html`.

[9] M. E. Clarke, D. A. Edwards, P. Webster, A. Bardsley. CS2262 Laboratory Manual, Chapter 5. The STUMP Processor Chip Project, Department of Computer Science, University of Manchester, January 1996.

[10] Ch. Ykman-Couvreur, B. Lin, H. De Man. ASSASSIN: An Asynchronous I/O Interface Synthesis System. In *IMEC documentation manual*, September 1994.

[11] D. A. Gilbert. *Dependency and Exception Handling in an Asynchronous Microprocessor*. Ph.D. thesis, Department of Computer Science, The University of Manchester, December 1997.

[12] D. A. Gilbert, J. D. Garside. A Result Forwarding Mechanism for Asynchronous Pipelined Systems. In *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'97*. Department of Computer Science, The University of Manchester, April 1997.

[13] D. J. Kinniment, A. V. Yakovlev, B. Gao. Metastable Behaviour and System Performance. In *Proceedings of the Second UK Asynchronous Forum*. University of Newcastle upon Tyne, UK, July 1997.

[14] D. K. Fellows. The Green Component of the Rainbow Hardware Description System. Master's thesis (1996), Department of Computer Science, The University of Manchester.

[15] E. Brunvard. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, September 1991.

[16] H. Barringer, D. Fellows, G.D. Gough, P. Jinks, B. Marsden, A. Williams. Design and Simulation in Rainbow: A framework for Asynchronous Micropipeline Circuits. In *European Simulation Symposium (ESS'96)*, Genoa, Italy.

[17] F. D. Schalij. Tangram Manual. Tech. Rep. UR 008/93 (1995), Philips Electronics N.V..

[18] H. Barringer, D. K. Fellows, G. D. Gough, A. Williams. Abstract Modelling of Asynchronous Micropipeline Systems using Rainbow. In *Hardware Description Languages and their Applications*, pages 285–302. Edited by: C. D. Kloos and E. Cerny, 1997.

[19] 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, The Netherlands, March 1998.

[20] IEEE Standard VHDL Language Reference Manual. ANSI/IEEE Std. 1076-1987, IEEE, 1988.

[21] IEEE Standard VHDL Language Reference Manual. ANSI/IEEE Std. 1076-1993, IEEE, 1993.

[22] I. E. Sutherland. Micropipelines. *Communications of the ACM* **32(6)**, Pages 720–738 (June 1989).

[23] 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.

[24] J. D. Ichbiah et al. *Reference Manual for the ADA Programming Language.* ANSI/MIL-STD-1815 A-1983, 1983.

[25] J. Grosch. A Toolbox for Compiler Construction. Karlsruhe Toolbox for Compiler Construction. Tech. Rep. 20 (1990), Gesellschaft für Mathematik und Datenverarbeitung MBH, Forschunsstelle für Programmstrukturen an der Universität Karlsruhe.

[26] J. Grosch. Ag - An Attribute Evaluator Generator. Karlsruhe Toolkit for Compiler Construction. Tech. Rep. 16 (1992), Gesellschaft für Mathematik und Datenverarbeitung MBH, Forschunsstelle für Programmstrukturen an der Universität Karlsruhe.

[27] J. Grosch. Ast - A Generator for Abstract Syntax Trees. Karlsruhe Toolkit for Compiler Construction. Tech. Rep. 15 (1992), Gesellschaft für Mathematik und Datenverarbeitung MBH, Forschunsstelle für Programmstrukturen an der Universität Karlsruhe.

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

[29] K. Y. Yun, D. L. Dill, S. M. Nowick. Synthesis of 3D Asynchronous State Machines. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 346–350, October 1992.

[30] L. Wall, T. Christiansen, R. L. Schwartz. *Programming Perl*. ISBN 1-56592-149-6. O'Reilly & Associates Inc.. Second Edition, 1996.

[31] M. A. Peña, J. Cortadello. Combining Process Algebras and Petri Nets for the Specification and Synthesis of Asynchronous Circuits. In *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'96*. Department of Computer Architecture, Universitat Politècnica de Catalunya, Spain.

[32] M. B. Josephs, J. T. Udding. Delay-insensitive circuits: an algebraic approach to their design. In *CONCUR '90*, June 1990.

[33] M. Kishinevsky, A. Kondratyev, A. Taubin, V. Varshavsky. *Concurrent Hardware – The Theory and Practice of Self-Timed Circuits.* Series in Parallel Computing. Wiley, 1994.

[34] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, J. Liu. A low-power, low-noise configurable self-timed DSP. In *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'98.* Cogency Technology U.K. Ltd., March 1998.

[35] P. B. Endecott. LARD Documentation Home Page. URL `http://www.cs.man.ac.uk/amulet/projects/lard/index.html`.

[36] S. M. Burns. Automated Compilation of Concurrent Programs into Self-timed Circuits. Master's thesis Caltech-CS-TR-88-2 (December 1987), California Institute of Technology, Pasadena, California, USA.

[37] T. Granlund. GNU MP - The GNU Multiple Precision Arithmetic Library. Tech. Rep. (1993), online 'info' documentation, Free Software Foundation.

[38] 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*.

[39] T. Lord. Guile Scheme. Tech. Rep. (1996), online 'info' documentation, Free Software Foundation.

[40] Edited by: W. Clinger, J. Rees. Revised^4 Report on the Algorithmic Language Scheme. (R4RS). Tech. Rep. (November 1991).

[41] W. J. Bainbridge, S. B. Furber. A Result Forwarding Mechanism for Asynchronous Pipelined Systems. In *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC'98*. Department of Computer Science, The University of Manchester, March 1998.

Appendix A. SSEM example, Balsa source and Breeze output

A.1. Balsa source

```
--  SSEM model in Balsa
--
--  1/12/97 Andrew Bardsley

public

type Word is 32 bits

--  SSEM function types
type SSEMFunc is enumeration
  JMP, JRP,     --  Abs. and rel. jumps
  LDN, STO,     --  Load negative and store
  SUB, SUB_alt, --  Two encodings for subtract
  TEST, STOP    --  Skip and stop ;)
end

--  Complete instruction encoding
type SSEMInst is record
  LineNo : 5 bits;
  CRTNo : 8 bits;
  Func : SSEMFunc
over Word

--  SSEM: Top level
procedure SSEM (output MemA : Word;
  output MemRNW : bit;
  input MemR : Word; output MemW : Word;
  sync Halted) is
local
  variable CI, ACC : Word
  variable PI : Word
  variable Stopped : bit
  variable ACC_slave, CI_slave : Word
  variable MDR : Word

  --  Memory operations, shared procedures
```

```
  shared MemoryWrite is
  begin
    MemRNW <- 0 ||
    MemA <- ((PI as SSEMInst).LineNo as Word) ||
    MemW <- ACC_slave
  end

  shared MemoryRead is
  begin
    MemRNW <- 1 ||
    MemA <- ((PI as SSEMInst).LineNo as Word) ||
    MemR -> MDR
  end

  --  Local procedures
  procedure IncrementCI is
  begin CI_slave := (CI + 1 as Word) end

  procedure InstructionFetch is
  begin MemRNW <- 1 || MemA <- CI_slave || MemR -> PI end

  procedure DecodeAndExecuteInstruction is
  begin
    case (PI as SSEMInst).Func of
         JMP then MemoryRead(); CI_slave := MDR
    also JRP then
      MemoryRead(); CI_slave := (CI + MDR + 1 as Word)
    also LDN then
      MemoryRead(); ACC_slave := ( - MDR as Word)
    also STO then MemoryWrite()
    also SUB .. SUB_alt then
      MemoryRead(); ACC_slave := (ACC - MDR as Word)
    also TEST then if (ACC as array 32 of bit)[31]
      --  skip if ACC is negative
      then CI_slave := (CI + 2 as Word) end
    also STOP then Stopped := 1
    end ;
    ACC := ACC_slave || CI := CI_slave
  end
begin
  ACC := 0 || CI := 0 ||
  Stopped := 0; --  reset initialisation
  while not Stopped then
    IncrementCI();
        InstructionFetch();
    DecodeAndExecuteInstruction()
  end;
  sync halted; halt --  STOP instruction effect
end
```

A.2. Breeze output

```
--  Breeze intermediate file
--  Created: Tue Dec  2 01:45:50 1997
--  By: bardslea@Fascination (Linux)
--  With: balsa-c version 971201

--  Imports

--  Types
type Word is 32 bits
type SSEMFunc is enumeration JMP=0,JRP=1,LDN=2,STO=3,
  SUB=4,SUB_alt=5,TEST=6,STOP=7 over 3 bits
type SSEMInst is record LineNo : 5 bits;CRTNo : 8 bits;
  Func : SSEMFunc over 32 bits
--  Constants

--  Parts

part SSEM (
  passive sync activate;
  active output MemA : 32 bits;
  active output MemRNW : 1 bits;
  active input MemR : 32 bits;
  active output MemW : 32 bits;
  active sync halted ) is
attributes ( isProcedure,noOfChannels=119 )
local
  sync #1
  push channel #2 : 32 bits
  push channel #3 : 1 bits
  pull channel #4 : 32 bits
  push channel #5 : 32 bits
  sync #6,#7
  pull channel #8 : 32 bits
  push channel #9 : 32 bits
  sync #10
  pull channel #11 : 32 bits
  push channel #12 : 32 bits
  sync #13,#14
  pull channel #15 : 1 bits
  push channel #16 : 1 bits
  sync #17
  pull channel #18 : 2 bits
  pull channel #19 : 32 bits
  pull channel #20 : 33 bits
  pull channel #21 : 32 bits
  push channel #22 : 32 bits
```

```
sync #23
pull channel #24 : 1 bits
sync #25
push channel #26 : 1 bits
pull channel #27,#28 : 32 bits
pull channel #29 : 33 bits
pull channel #30 : 32 bits
push channel #31 : 32 bits
sync #32,#33,#34
pull channel #35 : 32 bits
pull channel #36 : 33 bits
pull channel #37 : 32 bits
push channel #38 : 32 bits
sync #39,#40,#41
pull channel #42 : 1 bits
pull channel #43,#44 : 32 bits
pull channel #45 : 33 bits
pull channel #46 : 34 bits
pull channel #47 : 32 bits
push channel #48 : 32 bits
sync #49,#50,#51
pull channel #52 : 32 bits
push channel #53 : 32 bits
sync #54,#55,#56
pull channel #57 : 3 bits
sync #58
push channel #59 : 3 bits
sync #60
push channel #61 : 32 bits
pull channel #62 : 32 bits
sync #63
pull channel #64 : 32 bits
push channel #65 : 32 bits
sync #66
pull channel #67 : 1 bits
push channel #68 : 1 bits
sync #69,#70
pull channel #71 : 1 bits
pull channel #72 : 32 bits
pull channel #73 : 33 bits
pull channel #74 : 32 bits
push channel #75 : 32 bits
sync #76
pull channel #77,#78 : 1 bits
sync #79
pull channel #80 : 1 bits
push channel #81 : 1 bits
sync #82
pull channel #83 : 32 bits
push channel #84 : 32 bits
sync #85
```

```
    pull channel #86 : 32 bits
    push channel #87 : 32 bits
    sync #88,#89
    push channel #90 : 32 bits
    pull channel #91 : 32 bits
    sync #92
    pull channel #93 : 5 bits
    pull channel #94 : 32 bits
    push channel #95 : 32 bits
    sync #96
    pull channel #97 : 1 bits
    push channel #98 : 1 bits
    sync #99,#100
    pull channel #101 : 32 bits
    sync #102
    pull channel #103 : 5 bits
    pull channel #104 : 32 bits
    push channel #105 : 32 bits
    sync #106
    pull channel #107 : 1 bits
    push channel #108 : 1 bits
    sync #109,#110
    push channel #111,#112 : 32 bits
    push channel #113 : 1 bits
    pull channel #114,#115,#116 : 32 bits
    push channel #117 : 32 bits
    pull channel #118 : 32 bits
    push channel #119 : 32 bits
  begin
    $BrzVariable ( 32,3,"CI" : #119,{#72,#44,#19} )
    $BrzCallMux ( 32,2 : {#9,#84},#119 )
    $BrzVariable ( 32,2,"ACC" : #117,{#28,#118} )
    $BrzMask ( 1,32,2147483648 : #24,#118 )
    $BrzCallMux ( 32,2 : {#12,#87},#117 )
    $BrzVariable ( 32,3,"PI" : #61,{#114,#115,#116} )
    $BrzMask ( 5,32,31 : #103,#116 )
    $BrzMask ( 5,32,31 : #93,#115 )
    $BrzMask ( 3,32,57344 : #57,#114 )
    $BrzVariable ( 1,1,"Stopped" : #113,{#77} )
    $BrzCallMux ( 1,2 : {#16,#81},#113 )
    $BrzVariable ( 32,2,"ACC_slave" : #112,{#11,#101} )
    $BrzCallMux ( 32,2 : {#31,#38},#112 )
    $BrzVariable ( 32,2,"CI_slave" : #111,{#64,#8} )
    $BrzCallMux ( 32,4 : {#22,#48,#53,#75},#111 )
    $BrzVariable ( 32,4,"MDR" : #90,{#52,#43,#35,#27} )
    $BrzConcur ( 3 : #110,{#109,#106,#102} )
    $BrzFetch ( 1 : #109,#107,#108 )
    $BrzConstant ( 1,0 : #107 )
    $BrzFetch ( 32 : #106,#104,#105 )
    $BrzAdapt ( 32,5,false,false : #104,#103 )
    $BrzFetch ( 32 : #102,#101,#5 )
```

```
$BrzCallMux ( 1,3 : {#68,#98,#108},#3 )
$BrzCallMux ( 32,3 : {#65,#95,#105},#2 )
$BrzConcur ( 3 : #100,{#99,#96,#92} )
$BrzFetch ( 1 : #99,#97,#98 )
$BrzConstant ( 1,1 : #97 )
$BrzFetch ( 32 : #96,#94,#95 )
$BrzAdapt ( 32,5,false,false : #94,#93 )
$BrzFetch ( 32 : #92,#91,#90 )
$BrzCallDemux ( 32,2 : {#62,#91},#4 )
$BrzSequence ( 4 : #1,{#89,#79,#6,#7} )
$BrzConcur ( 3 : #89,{#88,#85,#82} )
$BrzFetch ( 32 : #88,#86,#87 )
$BrzConstant ( 32,0 : #86 )
$BrzFetch ( 32 : #85,#83,#84 )
$BrzConstant ( 32,0 : #83 )
$BrzFetch ( 1 : #82,#80,#81 )
$BrzConstant ( 1,0 : #80 )
$BrzWhile ( #79,#78,#76 )
$BrzUnaryFunc ( 1,1,Invert,false : #78,#77 )
$BrzSequence ( 4 : #76,{#70,#60,#58,#14} )
$BrzFetch ( 32 : #70,#74,#75 )
$BrzAdapt ( 32,33,false,false : #74,#73 )
$BrzBinaryFunc ( 33,32,1,Add,false,false,false :
  #73,#72,#71 )
$BrzConstant ( 1,1 : #71 )
$BrzConcur ( 3 : #60,{#69,#66,#63} )
$BrzFetch ( 1 : #69,#67,#68 )
$BrzConstant ( 1,1 : #67 )
$BrzFetch ( 32 : #66,#64,#65 )
$BrzFetch ( 32 : #63,#62,#61 )
$BrzCase ( 3,7,"7;6;4..5;3;2;1;0" :
  #59,{#17,#25,#34,#110,#41,#51,#56} )
$BrzFetch ( 3 : #58,#57,#59 )
$BrzCall ( 4 : {#55,#50,#40,#33},#100 )
$BrzSequence ( 2 : #56,{#55,#54} )
$BrzFetch ( 32 : #54,#52,#53 )
$BrzSequence ( 2 : #51,{#50,#49} )
$BrzFetch ( 32 : #49,#47,#48 )
$BrzAdapt ( 32,34,false,false : #47,#46 )
$BrzBinaryFunc ( 34,33,1,Add,false,false,false :
  #46,#45,#42 )
$BrzBinaryFunc ( 33,32,32,Add,false,false,false :
  #45,#44,#43 )
$BrzConstant ( 1,1 : #42 )
$BrzSequence ( 2 : #41,{#40,#39} )
$BrzFetch ( 32 : #39,#37,#38 )
$BrzAdapt ( 32,33,false,true : #37,#36 )
$BrzUnaryFunc ( 33,32,Negate,false : #36,#35 )
$BrzSequence ( 2 : #34,{#33,#32} )
$BrzFetch ( 32 : #32,#30,#31 )
$BrzAdapt ( 32,33,false,false : #30,#29 )
```

```
  $BrzBinaryFunc ( 33,32,32,Subtract,false,false,false :
    #29,#28,#27 )
  $BrzCase ( 1,1,"1" : #26,{#23} )
  $BrzFetch ( 1 : #25,#24,#26 )
  $BrzFetch ( 32 : #23,#21,#22 )
  $BrzAdapt ( 32,33,false,false : #21,#20 )
  $BrzBinaryFunc ( 33,32,2,Add,false,false,false :
    #20,#19,#18 )
  $BrzConstant ( 2,2 : #18 )
  $BrzFetch ( 1 : #17,#15,#16 )
  $BrzConstant ( 1,1 : #15 )
  $BrzConcur ( 2 : #14,{#13,#10} )
  $BrzFetch ( 32 : #13,#11,#12 )
  $BrzFetch ( 32 : #10,#8,#9 )
  $BrzHalt ( #7 )
end

-- EOF
```