Balsa:
A Tutorial Guide.

Doug Edwards, Andrew Bardsley, Lilian Janin, Luis Plana & Will Toms
## Contents

### 1 Introduction

1.1. Introducing Balsa

- What is Balsa? ......................................................... 1
- Basic concepts ....................................................... 1

1.2. Tool set and design flow ............................................. 3

1.3. Changes in releases

- Performance Enhancement ........................................... 5
- Experimental features .............................................. 5
- Bug Fixes .................................................................. 6
- Other Changes ............................................................ 6
- Deprecated or eliminated constructs/files .................... 7
- New constructs .......................................................... 7
- Changed behaviour .................................................... 7
- Balsa-mgr ................................................................. 8
- Cost Estimator .......................................................... 8
- Simulation environment ............................................. 8
- Channel viewer .......................................................... 8
- Back-end technologies ............................................... 8
- The Manual .............................................................. 8

### 2 Getting Started

2.1. A single-place buffer

- Description ............................................................... 9
- Commentary on the code ............................................ 9
- Reserved words ........................................................ 10
- Compiling the circuit ............................................... 10
- The synthesised circuit ........................................... 10

2.2. Two-place buffers

- 1st design ............................................................... 11
- 2nd design ............................................................... 12
## 3 The Balsa Language

### 3.1. Data Types
- Numeric types
- Enumerated types
- Record types
- Array types
- Constants
- Arrayed channels

### 3.2. Data Typing Issues
- Casts
- Bit ordering and padding in arrays
- Auto-assignment

### 3.3. Control Flow and Commands
- Sync
- Channel assignment
- Variable assignment
- Sequence operator
- Parallel composition
- Continue and Halt
- Looping constructs
- Structural iteration
- Conditional execution

### 3.4. Binary/Unary Operators

### 3.5. Description Structure
- File structure
- Declarations
- Procedures
- Shared procedures
- Functions
- Conditional ports and declarations
- Conditional ports
- Variable ports
3.6. Examples................................................................................................................. 42
  Modulo-16 counter .................................................................................................... 42
  Removing auto-assignment ....................................................................................... 42
  Modulo-10 counter .................................................................................................... 43
  A loadable up/down counter ..................................................................................... 43
  Sharing hardware ...................................................................................................... 44
  A “while” loop description ....................................................................................... 45
  Pitfalls in loop terminations ..................................................................................... 46
  The danger of “for” loops ......................................................................................... 46
  Selecting channels .................................................................................................... 47

4  Parameterised & Recursively Defined Circuits ........ 49

  4.1. Summary .............................................................................................................. 49

  4.2. Parameterised descriptions ................................................................................. 49
  A variable width buffer definition .......................................................................... 49
  Pipelines of variable width and depth ..................................................................... 50

  4.3. Recursive definitions ......................................................................................... 51
  An n-way multiplexer .............................................................................................. 51
  Commentary on the code ......................................................................................... 52
  A balsa test harness ................................................................................................ 52
  Handshake multiplier ............................................................................................... 52

  4.4. Pitfalls with Parameterised Procedures ............................................................. 53

5  Handshake Enclosure .................................................. 55

  5.1. Summary .............................................................................................................. 55

  5.2. Systolic counters ................................................................................................. 55
  A systolic modulo-11 counter ................................................................................. 57
  All even cells ............................................................................................................. 57
  All odd cells ............................................................................................................. 58
  A decoupled all even cell ......................................................................................... 59
  Parameterised version ............................................................................................. 60

  5.3. Active enclosure ................................................................................................. 60

  5.4. Use of enclosed channels.................................................................................... 61

6  Balsa Design Examples ............................................ 65

  6.1. Summary .............................................................................................................. 65

  6.2. A Population Counter ......................................................................................... 65
  Commentary on the code ......................................................................................... 67
  Enclosed Selection ................................................................................................... 67
  Avoiding deadlock: ................................................................................................. 67

  6.3. A Balsa shifter ..................................................................................................... 67
  Testing the shifter ..................................................................................................... 69
6.4. An Arbiter Tree ..................................................................................................... 69
6.5. A Stack Description............................................................................................. 71
   Commentary on the code .......................................................................................... 72
6.6. A Simple Processor – The Manchester SSEM (The Baby) ......................... 72
   SSEM types ............................................................................................................. 73
   Channel and Variable Declarations ..................................................................... 74
   Useful functions and shared procedures ............................................................... 74
   Decode and execute procedure ............................................................................. 75
   main body ............................................................................................................... 75
   Simulation .............................................................................................................. 75

7 Building test harnesses with Balsa ................................................................. 77

   7.1. Overview ......................................................................................................... 77
   Builtin types .......................................................................................................... 77
   Builtin Functions ................................................................................................... 78
   Strings ...................................................................................................................... 78

   7.2. Summary of Library Functions ..................................................................... 79
   types.builtin .......................................................................................................... 79
   sim.string ............................................................................................................... 79
   sim.fileio ............................................................................................................... 80
   sim.memory ............................................................................................................ 81
   sim.portio .............................................................................................................. 82
   sim.sim ................................................................................................................... 84

   7.3. Writing your own builtin functions ............................................................... 84
   The Balsa and C code ............................................................................................ 84
   Registering the function ......................................................................................... 84
   Compiling HelloWorld .......................................................................................... 85
   Invoking HelloWorld ............................................................................................ 86
   HelloWorld in Verilog ........................................................................................... 86
   Using balsa-mgr ..................................................................................................... 86

   7.4. Builtin functions with arguments ................................................................. 87
   Builtin typed arguments ......................................................................................... 89
   Return values ......................................................................................................... 90
   Functions with parameterised arguments ............................................................. 90

   7.5. Object Reference Counting .......................................................................... 91
   Variable assignment .............................................................................................. 91
   Function objects array .......................................................................................... 91

   7.6. Predefined types ............................................................................................ 92
   BalsaString .......................................................................................................... 92
   BalsaFile .............................................................................................................. 93

   7.7. Example Custom Test Harnesses ................................................................. 93
   Data Formatting ..................................................................................................... 93
   FileIO ...................................................................................................................... 95
   Memory models ..................................................................................................... 96
   A Processor Test Harness .................................................................................... 98
8 Implementations ................................................................. 99

8.1. Introduction ............................................................................. 99
Technologies .................................................................................. 99
Styles ............................................................................................. 99
Options .......................................................................................... 100

8.2. Creating an implementation .................................................. 101

9 Adding Technologies to Balsa ................................................. 105

9.1. The Balsa backend ................................................................. 105
Technologies and Styles ............................................................... 105
Directory structure ................................................................. 106

9.2. The technology configuration file ......................................... 107

9.3. Handshake component declarations ..................................... 109

9.4. Handshake component implementation descriptions ............ 110

9.5. Adding a new technology ...................................................... 111

9.6. The abs language ................................................................. 112
Bundles ......................................................................................... 112
Channels ....................................................................................... 112
Slices ............................................................................................. 113
Gate Operators ........................................................................... 113
Example ...................................................................................... 115

9.7. Netlists ..................................................................................... 117
ports: ............................................................................................. 118
nets ................................................................................................. 118
instances ....................................................................................... 118
attributes ..................................................................................... 119

9.8. The BALSATECH environment variable ................................ 119

9.9. The ABS Grammar ............................................................... 120
Components .................................................................................. 120
Styles ............................................................................................. 120
Gates .............................................................................................. 121
Slices ............................................................................................. 121
Include ......................................................................................... 122
Types .............................................................................................. 122
Expressions .................................................................................. 123

9.10. Netlist Format ....................................................................... 124
Netlist ........................................................................................... 124

10 Balsa Reference ................................................................. 127

10.1. Summary ............................................................................... 127

10.2. Balsa programs ................................................................. 127
11 The Balsa Language Definition ................................. 131
11.1. Summary .......................................................... 131
11.2. Reserved words .................................................. 131
11.3. Balsa Language Definition ................................. 131

12 The Breeze Language Definition .............................. 137
12.1. Summary .......................................................... 137
12.2. Breeze Language Definition ................................. 137
   Lexical ................................................................ 137
   Main Breeze language ........................................... 138
   Breeze extensions ................................................ 139

13 The Breeze Components ......................................... 141
13.1. Summary .......................................................... 141
13.2. Activation driven control components .................. 141
13.3. Channel termination components ....................... 141
13.4. Control to datapath interface components .......... 141
13.5. Pull datapath components .................................. 142
13.6. Connection components .................................... 142
13.7. Non-delay-insensitive components ...................... 142
13.8. Simulation-only components .............................. 142
13.9. Breeze components ordered by name ................. 142

14 Reporting Bugs ..................................................... 155
   Check your Balsa version ....................................... 155
   Searching for Bugs ............................................... 156
   Create a Bugzilla account ..................................... 156
   File the bug! ....................................................... 156
   What happens next? ............................................. 156

15 References .......................................................... 157
1 Introduction

1.1. Introducing Balsa

This document describes version 3.5 of the Balsa system and introduces significant performance enhancements. All users should recompile their Balsa source code descriptions after upgrading to this release. See “Changes in releases” on page 5 for further details. An online bug reporting mechanism has been introduced: see “Reporting Bugs” on page 155.

The tools described here can be run on any POSIX environment with X11 and at least 32bit integers (Linux, FreeBSD, MacOS X, Solaris). However, in order to produce a concrete implementation in either silicon or FPGA form, vendor specific tools are required: for example Xilinx design software, or the Cadence design framework with an appropriate cell library technology.

What is Balsa?

Balsa is the name of both the framework for synthesising asynchronous (clockless) hardware systems and the language for describing such systems. The approach adopted is that of syntax-directed compilation into communicating Handshaking Components and closely follows the Tangram [1] system of Philips. The advantage of this approach is that the compilation is transparent: there is a one-to-one mapping between the language constructs in the specification and the intermediate handshake circuits that are produced. It is relatively easy for an experienced user to envisage the architecture of the circuit that results from the original description. Incremental changes made at the language level result in predictable changes at the circuit implementation level. This is important if optimisations and design-tradeoffs are to be made easily at the source level and contrasts with a VHDL description in which small changes in the specification may make radical alterations to the resulting circuit.

Basic concepts

A circuit described in Balsa is compiled into a communicating network composed from a small set of Handshake components (~45 components, listed in Section “The Breeze Components” on page 141). The components are connected by channels over which communications or handshakes take place. Channels may have datapaths associated with them (in which case a handshake involves the transfer of data), or may be purely control (in which case the handshake acts as a synchronisation or rendez-vous point).

Each channel connects exactly one passive port of a handshake component to to one active port of another handshake component. An active port is a port which initiates a communication. A passive port responds (when it is ready) to the request from the active port by an acknowledge signal.
Data channels may be push channels or pull channels. In a push channel, the direction of the data flow is from the active port to the passive port, corresponding to a micropipeline style of communication. Data validity is signalled by request and released on acknowledge. In a pull channel, the direction of data flow is from the passive port to the active port. The active port requests a transfer, data validity is signalled by an acknowledge from the passive port. An example of a circuit composed from handshake components is shown in Fig. 1.1.

Here a Fetch component, also known as a Transferrer, (denoted by “→”) and a Case component (denoted by “@”) are connected by an internal data-bearing channel. Circuit action is activated by a request to the Fetch component which in turn issues a request to its environment (on the left of the diagram). The environment supplies the demanded data, indicating its validity by the acknowledgement signal. The Fetch component presents a handshake requests and data to the Case component using an active port (shown as a filled circle) which the Case component receives on its passive port (shown as an unfilled circle). Depending on the data value, the Case component issues a handshake to its environment on either the top right or bottom right port. Finally, when the acknowledgement is received by the case component, an acknowledgement is returned along the original channel and terminating this handshake. The circuit is ready to operate once more.

Data follows the direction of the request in this example and the acknowledgement to that request flows in the opposite direction. In this figure, individual physical request, acknowledgement and data wires are explicitly shown. Data is carried on separate wires from the signalling (it is “bundled” with the control although this is not necessary with other data/signalling encoding schemes. The bundled data scheme illustrated in Fig. 1.1 is not the only implementation possible. Methodologies exist (DI codes, dual rail encoding, NULL Convention Logic [2]) to implement channel connections with delay-insensitive signalling where timing relationships between individual wires of an implemented channel do not affect the functionality of the circuit. Handshake circuits can be implemented using these methodologies which are robust to naive realisations, process variations and interconnect delay properties. Version 3.5 of Balsa supports bundled data, and DI dual rail and 1-of-4 back-ends.

Normally, handshake circuits diagrams are not shown at the level of detail of Fig. 1.1, a channel being shown as a single arc with the direction of data being denoted by an arrow head on the arc and control only channels, comprising only request/acknowledge wires, being indicated by an arc without an arrowhead.

The circuit complexity of handshake circuits is often low: for example, a Fetch component may be implemented using only wires. An example of a handshake circuit for a modulo-10 counter [see “Removing auto-assignment” on page 42] is shown in Fig. 1.2. The corresponding gate level implementation is shown in Fig. 1.3.

Figure 1.1: Two connected handshake components
arranges the overlapping of the two handshakes (one push, one pull) such that the data-valid phases of the two data-validity protocols overlap.

1.2. Tool set and design flow

Balsa comprises a collection of tools, some of the more important are listed below.

- **balsa-c**: the compiler for the Balsa language. The output of the compiler is an intermediate language **breeze**.
- **balsa-netlist**: produces a netlist appropriate to the target technology/CAD framework from a Breeze description.
- **breeze2ps**: a tool which produces a postscript file of the handshake circuit graph.
- **breeze-cost**: a tool which gives an area cost estimate of the circuit.
- **balsa-md**: a tool for generating makefiles

---

**Figure 1.2**: Handshake circuit of a modulo-10 counter

**Figure 1.3**: Gate level circuit of a modulo-10 counter
1.2. Tool set and design flow

- **balsa-mgr**: a graphical front-end to *balsa-md* with project management facilities.
- **balsa-make-test**: automatically generates test harness for a Balsa description.
- **breeze-sim**: the preferred simulator working at the handshake component level
- **breeze-sim-control**: a graphical front-end to the simulation and visualisation environment

Obtainable in separate packages are:

- **gtkwave**: a waveform viewer
- **balsa-verilog-sim**: a package which makes Verilog simulation of Balsa descriptions easier by providing wrapper scripts for common simulators and by supporting user-written builtin functions which can be called from Balsa

A balsa-mode is also available for xemacs providing automatic syntax-based indentation of Balsa descriptions

An overview of the Balsa design flow is shown in Fig. 1.4

A Balsa description of a circuit is compiled using *balsa-c* to an intermediate *breeze* description. Most of the Balsa tools are concerned with manipulating the breeze handshake intermediate files. Breeze files can be used by back-end tools implementations for Balsa descriptions, but also contain procedure and type definitions passed on from Balsa source files allowing breeze to be used as the package description format for Balsa.
Behavioural simulation is provided by *breeze-sim*. This simulator allows source level debugging, visualisation of the channel activity at the handshake circuit level as well as producing conventional waveform traces that can be viewed using the waveform viewer *gtkwave*. The target CAD system may also be used to perform more accurate simulations and to validate the design. *breeze-sim* is still under active development: the facilities and user interface provided may be differ in detail from that described in this manual.

### 1.3. Changes in releases

#### Version 3.5

This release adds several changes aimed at achieving substantial performance improvements by the introduction of new handshake components. A number of experimental features have been made visible to the to the general user – use at your own peril – and there have been a number of minor bug fixes. There *may* be timing issues in the Xilinx back-end with certain combinations of handshake components. A future release will address these problems.

**Performance Enhancement.**

- New FalseVariable, Concur and SequenceOptimised Handshake Component implementations. These were described in a paper at ICCD05. These new implementations overlap the RTZ phase of the command, resulting in better performance with no area penalty. These are now the default implementations. The user can get the old, “conventional” implementations using the style options FV=conv, PAR=conv and SEQ=conv respectively when setting the BALSATECH environment variable or from balsa-mgr.

  In the single-rail (bundled-data) backend, the new FalseVariable implementations are only allowed if the new PassivatorPush implementation is used. If the PP=conv style option is used, the FV style option is ignored. This is required for correct implementation of the single-rail circuits.

- Variable and FalseVariable Handshake Components are now implemented with variable-sized read ports. This results in better area and performance by eliminating the need to split Variable and FalseVariable components (using the balsa-c option var-read-split) introducing additional split and combine components. This makes the var-read-split balsa-c option unnecessary.

- New average-case adder in DR. Previous releases only had a worst-case (time-balanced) adder in DR. This release introduces an average-case DR adder, with a quick return-to-zero phase. This results in better performance. The user can still get the balanced adder using the style option LOGIC=balanced when setting the BALSATECH environment variable or from balsa-mgr.

**Experimental features**

Several experimental features are included in this release. As the name implies, they may not be supported in future releases. They are believed to be correct at this time.

- `||!` this parallel operator instructs the compiler not to check for concurrent read and write accesses to variables by the composed commands. The user should be careful when using this operator: the circuit will fail if these access do exist. This operator should only be used when the user knows that the environment will NOT produce the concurrent read and write accesses.

- `->! select!` These input operators modify the semantics of the input commands. The normal operators (`->` and `select`) wait for the input to arrive before activating the enclosed command. The new operators will activate the command as soon as the control flow reaches the input operation, without waiting for the data to arrive. This gives the control a head start. The
command will not operate on the wrong data since the data will identify its validity (with the req/ack signals) but any operation that does not depend on the data may occur before the data arrives. This may result in the wrong operation. The new operators still guarantee that the command will not complete until data has also completed. For example:

```plaintext
a, b ->! then
  o1 <- a + b
||
  o2 <- 1
end
```

In this case, \(o1\) always gets the value of \(a+b\) after \(a\) and \(b\) have arrived. However, \(o2\) will get the value 1 as soon as the control flow reaches the input command, without waiting for \(a\) and \(b\). Consider:

```plaintext
a, b ->! then
  o <- 1
end
```

In this case, \(o\) will get a 1 as soon as the control flow reaches the input command without waiting for \(a\) and \(b\). It should be noted that the input command will not complete until the inputs have arrived thus maintaining the overall flow of control.

Clearly, the `select!` operator should only be used for passive inputs without choice. Otherwise, all the alternative commands will be activated without waiting for the data to make the choice. These will result in incorrect operation.

### Bug Fixes
  A new Fetch component is introduced. As an alternative, a new PassivatorPush with data storage is introduced which also solves the problem resulting in better performance albeit at the cost of larger area.

  **Explanation:** The Fetch component has a reduced_broad input (pull channel) and a broad output (push channel). This cannot be done with an "all-wires" implementation of the Fetch. Most of the time we get away with it because the inputs come from Variable or FalseVariable components that make those inputs broad. The problem is with active input commands like \(i \rightarrow o\), which get the input from a PassivatorPush. In this case, the new Fetch or the new PassivatorPush is needed.

  The new PassivatorPush component is the default, to target better performance. The user can still get the old, "conventional" implementation of the PassivatorPush using the style option `PP=conv` when setting the BALSA TECH environment variable. This will automatically introduce the new Fetch component implementation (resulting in better area and reduced performance).

- Added handling of global signals by balsa-make-impl-test
- Dual-rail implementation of CaseFetch was not guaranteed QDI for certain specifications
- Insertion of suggest-buffers in Variable, FalseVariable and PassivatorPush components is done in a better way in all design styles.

### Other Changes
- Support for builtin functions in verilog was added for the 1-of-4 style.
- Balsa-make-helpers has been updated. It supports all design styles (single-rail, dual-rail, 1-of-4). It generates a helpers-cells file and a gate-mappings file which include the new, generated cells only. **The user must concatenate these files with the original ones.**

- The reference sections of the manual have been updated to correct errors and to reflect the latest changes to the system.
1.3. Changes in releases

Version 3.4

This release adds “builtin” types [see “Built-in types” on page 77] – file I/O, string handling and memory models are included adding significant changes in the capabilities of the simulation tools. Interfaces to a number of Verilog simulators have been included.

Version 3.3

The changes listed here are the major changes since the first version of the Balsa manual. Some of these changes have however appeared in various snapshots that were published on the Balsa website and some were described in a text book [3] produced to promote the European Low-Power Initiative for Electronic System Design.

Deprecated or eliminated constructs/files

- public and private keywords have been eliminated
- else clauses of while statement are no longer supported
- the keyword local is not required for declarations which immediately follow procedure declarations.
- .sbreeze files are no longer generated as part of the compilation process. A modified .breeze format now replaces both .breeze and .sbreeze files.

New constructs

- Ports to procedures can now be connected to variables to allow communications on the procedure’s ports to perform reads and writes to the variable [see “Variable ports” on page 41].
- a multicast keyword has been added to prefix channel/sync declarations to suppress warning about multicast channels. The “-c warn-multicast” option to balsa-c now does nothing – it is enabled by default.
- implicits and don’t care values may be used more widely in expressions; see “implicants” on page 32, and “case statements” on page 37.
- ports, local and global declarations may be conditional [see “Conditional ports” on page 41].
- new loop constructs have been added [see “Looping constructs” on page 36].
- case statements may be parameterised [see “Conditional execution” on page 37]
- simulation time printing is now supported by the print command [see “Control Flow and Commands” on page 34]
- a bit-array-cast operator, #, has been added as syntactic sugar to simplify array slicing and casting.
- active input enclosure commands have been added
- the syntax of parameterised procedure calls has changed

Changed behaviour

- the syntax of the while command has been changed. Existing programs may no longer compile [see “Looping constructs” on page 36] for more details.
- should multiple guards be true (in if and while) commands, the earliest command in the guard list is executed – previously the command chosen was undefined.
- if commands, ports and declarations now correctly fail to evaluate subsequent commands if an earlier guard is true.
- breeze files must be regenerated – they are no compatible with the latest version, sbreeze files are obsolete.
1.3. Changes in releases

**Balsa-mgr**
The GUI to the Balsa system, balsa-mgr, is now stable and is the recommended way of driving the tool set.

**Cost Estimator**
The cost estimator now handles hierarchical circuits correctly.

**Simulation environment**
LARD is no longer the recommended functional simulation route. A new simulation engine operating on the breeze description of circuits simulates directly and gives a speed improvement of 25,000 times. Co-simulation with existing lard test harnesses is still possible, but with reduced performance. Lard support for Balsa is no longer part of the main distribution, but is available as a separate package, balsa-lard.

**Channel viewer**
The LARD channel viewer is no longer used for a graphical representation of channel activity. Although impressive for small demonstration purposes, it was very slow, it was difficult to restrict the view to “important” channels, snapshots could not be saved and restarted etc. The new simulation viewer is based on a conventional waveform viewer derived from GTKWave.

**Back-end technologies**
A wide range of backend technologies and styles are supported and easily controlled via balsa-mgr. Users can select between single rail (bundled data), dual rail, 1-of-4 and NCL styles each with different latch implementations. A Xilinx technology and a generic Verilog netlist are distributed. For users with appropriate licensing arrangements, a number of silicon technologies, e.g. AMS 0.35µm and ST 01.8µm are available.

**The Manual**
The format of the manual has changed. A more complete definition of the language is included. There is now a section on how to create different back-end technologies and styles. The example descriptions have been extended. The emphasis of the manual has changed: the previous version over-emphasised passive enclosed selection. Many users were misled into believing that this descriptive style was good practice. It is hoped that this version separates the issue of passive versus active ports from that of enclosed handshakes and encourages a more natural style of description.
2

Getting Started

Summary
In this chapter, simple buffer circuits are described in Balsa introducing the basic elements of a Balsa description. The GUI to the Balsa system, *balsa-mgr*, is used to hide the complexity of the underlying command line tools. All the examples illustrated here can be found in the Examples directory of this documentation.

To install the Basla system, either from a binary distribution, or from a build of the source distribution, follow the instructions in the INSTALL file.

2.1. A single-place buffer

Description
A Balsa description, in buffer1a.balsa, of a byte-wide, single place buffer is:

```
buffer1a.balsa

(--- Balsa program defining an 8 bit wide single place buffer
   This is an example of a multi-line (--- nested ---) comment
---)

--- Single line comments are also allowed
import [balsa.types.basic]

procedure buffer1 (input i : byte; output o : byte) is
variable x : byte
begin
loop
   i -> x        -- Input communication
   ;
   o <- x        -- Output communication
end
end
```

Commentary on the code
This Balsa description builds a single-place buffer, 8 bits wide. The circuit requests a byte from the environment which, when ready, transfers the data to the register. The circuit signals to the environment on its output channel that data is available and the environment reads it when it chooses. The description introduces:

**comments:** Balsa supports both multi-line and single line comments; both types may be nested.
2.1. A single-place buffer

**modular compilation:** Balsa supports modular compilation. The `import` statement in this example includes the definition of some standard data types such as `byte`, `nibble`, etc. A full list of the current definitions is given in `<BalsaInstallDir>/share/balsa/types/basic.breeze`. The search path given in the `import` statement is a dot separated directory path similar to that of Java except multi-file packages are not implemented. The `import` statement may be used to include other pre-compiled balsa programs thereby acting as a library mechanism. The `import` statements must precede other declarations in the files. The `import` statement is included in this example for completeness only. None of the types defined in `basic.breeze` are actually used this example so the `import` statement could have been omitted.

**procedures:** The `procedure` declaration introduces an object that looks similar to a procedure definition in a conventional programming language. In Balsa, a procedure is compiled to handshake circuit comprising a network of handshake components. The parameters of the procedure define the interface to the environment outside of the circuit block. In this case, the module has an 8-bit input datapath and an 8-bit output datapath. The body of the procedure definition defines an algorithmic behaviour for the circuit; it also implies a structural implementation. In this example, a variable `x` (of type `byte` and therefore 8 bits wide) is declared implying that a 8-bit wide storage element will be appear in the synthesised circuit.

The behaviour of the circuit is obvious from the code: 8-bit values are transferred from the environment to the storage variable, `x`, and then sequential output from the variable to the environment. This sequence of events is continually repeated (`loop ... end`).

**channel assignment:** the operators `->` and `<-` are channel input and output assignments and imply a communication or handshake over the channel. Because of the sequencing explicit in the description, the variable `x` will only accept a new value when it is ready; the value will only be passed out to the environment when requested. Note that the channel is always on the left-hand side of the expression and the corresponding variable on the right-hand side.

**sequencing:** The `;` symbol separating the two assignments is not merely a syntactic statement separator, it explicitly denotes sequentiality. The program has been formatted somewhat artificially to emphasise the point. The contents of `x` are transferred to the output port after the input transfer has completed. Because a `;` connects two sequenced statements or blocks, it is an error to place a `;` after the last statement in a block.

**Reserved words**

Care must be take to avoid using Balsa’s keywords as variable or procedure names. Usually, this is not a difficult restriction to remember, but a common mistake, especially for beginners experimenting with the language, is to name an input channel `in`. Unfortunately, `in` is a reserved word and will generate a Balsa compile error.

**Compiling the circuit**

`balsa-c buffer1a`

The description in `buffer1a` is compiled producing an output file `buffer1a.breeze`. This is a file in an intermediate format which can be imported back into other balsa source files (thereby providing a simple library mechanism). The file extension `.balsa` of the source filename is optional and contains no special significance to the compilation system. However, if a different file extension is used, the file name including the extension must be given as the argument to the `balsa-c` command.

The file extension `.breeze` is of significance to the compilation system

Breeze is a textual format file designed for ease of parsing and therefore somewhat opaque. A primitive graphical representation of the compiled circuit in terms of handshake components can be produced (in `buffer1a.ps`) by:

`breeze2ps buffer1a`

The resulting handshake circuit is shown in Figure 2.1. Note that this is not actually taken from the output of `breeze2ps`, but has been redrawn to make the diagram more readable. Although it is not necessary to understand the exact operation of the compiled circuit, a knowledge of the structure is

1. there is, of course, no predefined type `word`
2.2. Two-place buffers

Helpful for an understanding of how to describe circuits which can be efficiently synthesised using Balsa. A brief description of the operation of the circuit is given below. The circuit has been annotated with the names of the various handshake elements.

The port at the top of the Loop ("#") component is an activation port which encloses (see “Handshake Enclosure” on page 55) the behaviour of the circuit. It can be thought of as a reset signal which, when released, initiates the operation of the circuit. All compiled Balsa programs contain an activation port.

The activation port starts the operation of the Loop which initiates a handshake with the Sequencer (";"). This component first issues a handshake to the left-hand Fetch component "\( \rightarrow \)" causing data to be moved to the storage element in the Variable element (marked "x" to match the variable name). The Sequencer then handshakes with the right-hand Fetch component causing data to be read from the Variable element. When these operations are complete, the Sequencer completes its handshake with the repeater which start the cycle again.

2.2. Two-place buffers

1st design

Having built a single place buffer, an obvious goal is a pipeline of single buffer stages. Initially consider a two-place buffer; there are a number of ways we might describe this. An obvious way is to define a circuit with two storage elements:

```
-- buffer2a: Sequential 2-place buffer with assignment between variables
import [balsa.types.basic]
procedure buffer2 (input i : byte; output o : byte) is
  variable x1, x2 : byte
begin
  loop
    i -> x1; -- input communication
    x2 := x1; -- implied communication
    o <- x2 -- output communication
  end
end
```

In this example in we explicitly introduce two storage elements, \( x_1 \) and \( x_2 \). The contents of the variable \( x_1 \) are caused to be transferred to the variable \( x_2 \) by means of the assignment operator ":=".
2.3. Parallel composition and module reuse

However, transfer is still effected by means of a handshaking communication channel. This assignment operator is merely a way of concealing the channel for convenience.

2nd design

The implicit channel can be made explicit as shown in buffer2b.balsa:

```
-- buffer2b: Sequential version with an explicit internal channel
import [balsa.types.basic]
procedure buffer2 (input i : byte; output o : byte) is
  variable x1, x2 : byte
  channel chan : byte
begin
  loop
    i -> x1;         -- input communication
    chan <- x1 || chan -> x2; -- transfer x1 to x2
    o <- x2          -- output communication
  end
end
```

The channel, which was in the previous example, concealed behind the use of the ":=" assignment operator has been made explicit. The handshake circuit produced (after some simple optimisations) is identical to buffer2a. The "||" operator is explained in the next example.

It is important to understand the significance the operation of the circuits produced by buffer2a and buffer2b. Remember the ";" is more than a syntactic separator: it is an operator denoting sequence. Thus, first the input, i, is transferred to x1. When this operation is complete, x1 is transferred to x2 and finally the contents of x2 are written to the environment. Only after this sequence of operations is complete can new data from the environment be read into x1 again.

2.3. Parallel composition and module reuse

The operation above is unnecessarily constrained: there is no reason why the circuit cannot be reading a new value into x1 at the same time that x2 is writing out its data to the environment. The program in buffer2c achieves this optimisation.

```
-- buffer2c: a 2-place buffer using parallel composition
import [balsa.types.basic]
import [buffer1a]
procedure buffer2 (input i : byte; output o : byte) is
  channel c : byte
begin
  buffer1 (i, c) ||
  buffer1 (c, o)
end
```

Commentary on the code

In the description above, a 2-place buffer is composed from 2 single-place buffers. The output of the first buffer is connected to the input of the second buffer by their respective output and input ports. However, apart from communications across the common channel, the operation of the two buffers is independent.

The deceptively simple program above illustrates a number of new features of the balsa language:

- **modular compilation**: The import mechanism is used to include the buffer1a circuit described earlier.
- **connectivity by naming**: The output of the first buffer is connected to the input of the second buffer because of the common channel name (c) in the parameter list in the instantiation of the buffers.
parallel composition: The “||” operator specifies that the two units which it connects should operate in parallel. This does not mean that the two units may operate totally independently: in this example the output of one buffer writes to the input of the other buffer creating a point of synchronisation. Note also that the parallelism referred to is temporal parallelism. The two buffers are physically connected in series.

2.4. Placing multiple structures

If we wish to extend the number of places in the buffer, the previous technique of explicitly enumerating every buffer becomes tedious. What is required is a means of parameterising the buffer length (although in any real hardware implementation the number of buffers cannot be variable and must be known beforehand). The for construct together with compile-time constants may be used.

```balsa
buffer_n.balsa
    -- buffer_n: an n-place parameterised buffer
    import [balsa.types.basic]
    import [buffer1a]
    constant n = 8

    procedure buffer_n (input i : byte; output o : byte) is
        array 1 .. n-1 of channel c : byte
    begin
        buffer1 (i, c[1]) || -- first buffer
        buffer1 (c[n-1], o) || -- last buffer
        for || i in 1 .. n-2 then -- buffer i
            buffer1 (c[i], c[i+1])
        end
    end
```

Commentary on the code

constants: the value of an expression (of any type) may be bound to a name. The value of the expression is evaluated at compile time and the type of the name when used will be the same as the original expression in the constant declaration. Numbers can be given in decimal (starting with one of 1…9), hexadecimal (0x prefix), octal (0 prefix) and binary (0b prefix).

arrayed channels: procedure ports and locally declared channels may be arrayed. Each channel can be referred to by a numeric or enumerated index [see “Arrayed channels” on page 32], but from the point of view of handshaking, each channel is distinct and no indexed channel has any relationship with any other such channel other than the name they share.

for loops: a for loop allows iteration over the instantiation of a subcircuit. The composition of the circuits may either be parallel composition – as in the example above – or sequential. In the latter case, “;” should be substituted for “||” in the loop specifier. The iteration range of the loop must be resolvable at compile time.

A more flexible approach uses parameterised procedures and is discussed later [see “Parameterised descriptions” on page 49].

2.5. Using balsa-mgr

Balsa-mgr is project manager environment which acts as a front-end to the Balsa commands such as balsa-c and breeze2ps. It hides much of the complexity of the various command-line options that more complicated compilation and simulation scenarios demand. The use of the project manager is best illustrated by using it to rerun the compilation of the single place buffer described in buffer1a [“buffer1a.balsa” on page 9]

Creating a new project

The command balsa-mgr invokes the project manager. Select “Project ⇒ New” from the pull-down menu as shown in Figure 2.2 to display the dialogue box shown in Figure 2.3. A default name for the project is generated; this may be over-ridden to something more meaningful. The “Project
2.5. Using balsa-mgr

Directory” text box specifies the root directory: a file named “Project” is created here containing information about the project. The button to the right of the text box activates a file browser for graphically selecting the required directory. The “File Import Path” text box allows the directory in which the source Balsa files reside to be specified. By default, this is the current directory (relative to the root of the project) but may be changed either by directly typing in the text box or by using the file browser activated by clicking on the button to the right of the text box. More directory import paths can be added by means of the “New” button.

Only one project is allowed per directory but each project may have several compilation targets. The options in the other tabbed panes, “Compilation options” and “Definitions” are described later.

The source files to be compiled must be specified. Either select “Files ⇒ Add Files into Project” from the pull-down menu, or the keyboard accelerator Ctrl-A, or click on the icon as shown in Figure 2.4. Pick *buffer1a.balsa* and click “OK”. The filename should appear in the left-hand pane of

---

**Figure 2.2:** Creating a new project.

**Figure 2.3:** The New Project Dialogue Box.
2.5. Using balsa-mgr

the project manager together with the name of any procedures listed in that file. Clicking on the file
or procedure name will cause the contents of the file to be listed in the right hand edit pane as shown
in Figure 2.5. The file may be edited in-situ in the pane or by an external editor (defined in the
environment options pane assessable by the “Project ⇒ New” pull-down menu) which can be
invoked by clicking on the edit icon above the edit pane.

Compiling a description

Files that have not been compiled will have a warning symbol against them. Users should be aware
that until the file has been compiled, the list of procedures displayed under any filename is the result
of a simplistic parsing of the source file and may occasionally be misleading: for example
procedures that have been commented out and parameterised procedure definitions (see
“Parameterised descriptions” on page 49) will be shown erroneously. Further, conditionally
declared procedures (see “Conditional ports and declarations” on page 40) are missing. Upon a
successful compilation, the procedures will be correctly displayed.

In order to compile the circuit, either middle click on the file name or click on the Makefile tab in
the left-hand pane. The new view, Figure 2.6, reveals the actions available. Click on the Compile
button to compile the description. If the project has been changed since the user last saved it, a save-
project dialogue box appears. A new window, the execution window, is spawned which records
various stages in the compilation process.

Behind the scenes, balsa-mgr analyses the dependencies in the sources files in the project, creates
a Makefile that reflects these dependencies and generates rules in the Makefile to invoke the various
Balsa commands. If the initial Balsa description is syntactically incorrect in such a way as to make
impossible the determination of dependencies, the Makefile will not be correctly generated.

Running Balsa from the command line allows more flexibility than from within balsa-mgr, however
balsa-mgr is much more convenient for the majority of tasks. Since describing a GUI is
exceptionally tedious, users are encouraged to browse the various icons and pull-down menus
themselves. Note that right-clicking in the various panes brings up various context sensitive menus.
2.5. Using balsa-mgr

Compilation errors

If errors are found during the compilation of a circuit, the errors, together with the line number and character position of the error, are reported in the output pane of the execution window. Clicking on the displayed error message causes the offending code to be highlighted in red in the edit pane window.

1. Start balsa-mgr and select the project defined previously for the circuit buffer1a
2. Add the file *buffer2b* to the project
3. Compile the circuit by clicking on the **Compile** button for buffer2b.breeze. The circuit should compile OK
4. Change the parallel composition operator “||” to the sequential operator “;”.
5. Save the file and recompile buffer2b.breeze
6. An compile time error should now be reported in the execution window:

   buffer2b.balsa:11:16: unbalanced channel usage; can't perform <write> ; <read> on channel 'chan'

7. Click on the message. The offending code should be highlighted in the edit-pane window on line 11 starting at character position 16. If tabs are used in the source file, the tab size must be known in order for the character position to be correctly reported. VI users may set the value in their ~/.exrc file which is consulted by balsa-c. Alternative, -t <tabsize> may be passed as an option to balsa-c from the Compilation Options pane from the Project Options menu.

   Quite apart from illustrating the mechanics of error reporting within the balsa-mgr framework, this example demonstrates why designing asynchronous circuits requires a deeper understanding of the design process than does the design of synchronous circuits. It is important to realise why the compiler objects to the circuit description. Line 11 contains two statements. In the first, data is output from variable *x1* to the internal channel *chan*. The next statement, because of the sequence operator “;” cannot start until the previous statement has completed which requires data to be taken on the channel to be acknowledged. It is this second statement transferring data from the channel *chan* to the variable *x2* which would cause the data transfer on to the channel to be acknowledged. In other words, the first statement is waiting for the second statement, but the second statement can not start until the first has terminated.

   In this particular case, the compiler can spot the problem. However, conceptually similar deadlock situations can arise at higher levels of system specification. In such cases, the circuit will compile satisfactorily, but will deadlock in operation.

8. Correct the error before proceeding further.

### Handshake circuit graph

Click on the **View** button opposite the label “buffer1a.ps”. If necessary, the circuit will be compiled and a PostScript viewer will appear displaying the handshake circuit graph just as it did when the viewer was invoked via the command line.

### Circuit cost

The area cost of a circuit may be found by determined by clicking on the **Run** button opposite the **cost** label. Doing so will cause the execution window (Figure 2.7) to display the area cost of the circuit. This cost is only a guideline figure assuming a particular back-end implementation. Nevertheless, the cost figure is useful for gaining quick feedback on how changing the description of a circuit affects its size. The output from breeze-cost needs some interpretation: each handshake circuit is listed together with its cost, name, data width, and the internal channel identifiers to which the component is connected. Note that the cost of the Fetch component is zero. This is because in the back-end assumed for the cost function, a Fetch component is a wire only element.
2.6. Simulation.

Apart from the various simulation possibilities available once the design has been converted to a silicon layout, there are three strategies for evaluating/simulating the design from Balsa.

1. Default test harness.
   A default test harness can be generated. The default test harness exercises the target Balsa block by repeatedly handshaking on all external channels; input data channels receive a user defined value on each handshake, although it is possible to associate an input channel with a data file. Data sent to output channels appears on the output pane of the execution window. Note that if the interface to procedure under test is changed, a new test-harness must be generated. By default, the Makefile cannot check this dependency: the test-harness file must either be removed manually or by running `make clean`.

2. Balsa test harness.
   If a more sophisticated test sequence is required, Balsa is a sufficiently flexible language in its own right to be able specify most test sequences. A default test harness can then be generated to exercise the Balsa test harness: see “Building test harnesses with Balsa” on page 77.

Saving Window Contents

The outputs logged in the StdErr or StdOut panes can be edited or saved to a file by right-clicking in the pane. When editing, either the internal editor in the Balsa-mgr edit pane or an external editor as defined in the environment options can be nominated.
3. Custom LARD test harness.
   For some applications, it may be necessary to write a custom test harness in a language such as LARD. However, LARD is no longer supported as part of the Basla system.

**Adding a test fixture**

To simulate a circuit description using Balsa’s simulation facilities, a test fixture has to be added to the design framework. The easiest way is to automatically generate a default test harness.

1. Start balsa-mgr and select the project defined previously for the circuit `buffer1a`
2. Delete the file `buffer2b` from the project by selecting it in the file pane and right clicking to choose “Delete” from the pop-up menu. This step isn’t actually necessary but `buffer2b` is not used again
3. Add the file `buffer2c` to the project.
4. In the Makefile tab, you can notice that a set of compilation actions for `buffer2c` has been added to those for `buffer1a` (see Figure 2.8).

![Figure 2.8: Buffer2c actions added to Makefile pane](image)

5. Back in the Files pane, make sure that `buffer2` is selected.
6. Pick “Selected Item ⇒ Add Test Fixture” from the pull-down menu or right-click on “buffer2” in the Files pane. A window for creating a text fixture is spawned (Figure 2.9).
7. A Warning button warns you that the Breeze file has not been generated, meaning that the Balsa file `buffer2c.balsa` has not been compiled. Compiling is necessary for the GUI to get a knowledge of the procedure’s input and output ports. Click on the warning button to compile `buffer2c.balsa`. The “Test Options” dialog should change and show the procedure’s port `i` and `o`.
8. Change the name of the test fixture from the default (test1) to something more meaningful, e.g. `buff2c`
9. Select the Port Name “`i`”
10. Change the active radio button in the “Component Type” pane from “input value” to “input from file” (see Figure 2.10).
2.6. Simulation.

Some test values (in a variety of representations: decimal, hex and binary) have been provided in the file `th1.dat` in the directory containing the example. Set the value of the filename in the “Port Value/Filename” text box to `th1.dat` either by typing directly into the text box or by clicking on the file browser button immediately to the right of the text box.

Note that data values can be specified in various notations (binary, octal, hexadecimal, decimal). The format of data files is line based: Only one data item is allowed per line and com-

![Figure 2.9: The test options dialog before Balsa compilation](image)

![Figure 2.10: The test option dialog after Balsa compilation](image)
2.6. Simulation.

plex data types values should not be split across lines. Anything after a data item is treated as a comment and is passed to the simulation.

12. Validate your test fixture definition by clicking OK.

In order to run the test, click on the Makefile tab. The Makefile view shown in Figure 2.11 now shows two actions added under the Tests section. Clicking on the Run button for sim-buff2c generates the simulation output in the execution window (Figure 2.12). The numbers reported on the left hand side of each channel activity are simulation times – either the time at which data is presented at an input channel from the external environment or the time at which data is presented on an output channel to the external environment. Note however that at the breeze level the simulator has a very simplistic timing model, so these values should be treated with caution.
2.6. Simulation.

**Capturing output**

The contents of the output window of the execution window can be captured by right-clicking in the output pane; alternatively the output can be directed to a file when defining or editing the test harness.

**Graphical Simulation Tools**

In the previous examples, the output of the simulation is textual appearing in the output pane of the execution window. The simulation may also be viewed in a conventional style waveform viewer or the channel activity can be viewed directly on a representation of the handshake circuit graph. To activate the viewers, switch to the Makefile pane of Balsa-mgr and click on Run button for sim-win-buff2c. This will generate any intermediate files required and bring up a window breeze-sim-ctrl (Figure 2.13) which controls the simulations and animations.

![Figure 2.12: The output from a text-only simulation](image)

![Figure 2.13: The Simulator Controller](image)

The controller allows:

- a new simulation file for a design to be produced and displayed in a waveform viewer (GTKWave).
2.6. Simulation.

- an existing simulation file to be viewed in the GTKWave viewer.
- the speed of the simulation to be varied.
- the handshake circuit graph to be displayed, animated and analysed.
- the source code to be displayed.
- the associations between handshake circuit channels and source code constructs to be displayed.

**Breeze-sim-controller icons**

- If an existing simulation file is detected when the controller is started, a warning triangle icon is displayed to alert the user to the possibility that this file could be overwritten (the file has the extension of .hhh).
- An existing simulation file may be viewed without it being regenerated by clicking on the waveform viewer icon at the bottom of the controller window. If the waveform viewer is active, clicking on the icon kills the viewer.
- The coloured button at the bottom left of the controller window indicates the status of the simulation: red means the simulation is stopped, green that the simulation is running and blue that the simulation is paused.
- A new simulation trace file can be generated and displayed by clicking on the simulation run/pause button at top-left of the controller window. The simulation can be terminated by means of the simulation stop button to the right of the run/pause button. The simulation is displayed in the GTKWave viewer as the simulation file is produced. The speed of the simulation can be slowed down by means of the speed slider control.
- The two icons at the bottom right of the window reveal further functionality: the left icon reveals a graph of the handshake circuit and the right button opens a window onto the source code.

**Generating the simulation trace**

Although breeze-sim-ctrl can be used to view the static handshake circuit (in order, for example, to analyse the associations between the handshake elements and the Balsa description), its aim is to graphically control the simulation process and display the simulation events in various ways. Before any visualisation, it is necessary to generate a simulation trace. The presence of a simulation trace is indicated by the **Duration** indicator, showing the total length of the actual simulation. You can generate a new simulation (.hhh) trace file by running the simulation with the **Play** button. The simulation is generating events very quickly, and the trace file can quickly become very large. If your simulation is too long, you may want to keep the simulation trace short by slowing the simulation speed down with the slider control and by stopping the simulation with the **Stop** button when it reaches the desired size (the **Duration** indicator of the simulation is updated in real time).

Which simulated events are saved in the trace file can be chosen from the Trace options menu, in the Breeze-sim options section (Figure 2.14). The choice is between tracing all the channels or tracing only the procedure ports. Tracing all the channels results in a large trace file containing all the necessary information for any kind of visualisation or post-analysis. Tracing only the procedure ports is useful for keeping the trace file small, while still being able to view in GTKWave the events happening at the interface of your procedures. This is often enough for checking that high-level communications are behaving as expected without going into the details of the implementation.
2.6. Simulation.

GTKWave, the Waveform Viewer

Clicking on the waveform viewer icon or the simulation run button will start the GTKWave viewer (this automatic launch of GTKWave when a simulation is runned is the default behaviour, and can be overriden by placing an empty file named nogtkwave in the project directory). A list of channels is displayed in the right-hand pane as shown in Figure 2.15. Request signals are shown in red and acknowledge signals are shown in green. Data bearing channels have the data value displayed under the request/acknowledge signals. Clicking in this pane will display a vertical timeline cursor in the window.

Which channels are viewed at the launch of GTKWave can be chosen from the Trace options menu, in the GTKWave options section (Figure 2.14). The four possible choices are View all traced channels, View procedure ports only, View named ports only and View none. They are self-explanatory, except perhaps the third one: Named ports correspond to all the procedure ports but the activation signals associated to every procedure (these signals do not have any name in the breeze file).
2.6. Simulation.

The left-hand pane shows the channel names and the state of the request/acknowledge signals and data values at the cursor point. It is necessary to click in the waveform display pane to get the channel names to display correctly in the first instance. GTKWave is highly configurable; a detailed description of its operation is not given here, rather a summary of its capabilities is provided below.

- The display of the traces passed to the viewer from the simulation controller can be configurable by use of the add or add all buttons. The former allows signals to be chosen from a pick-list or by a regular expression description together with range specifiers – useful for specifying buses.
- Traces can be removed or repositioned by means of the cut and paste buttons.
- Traces can be sorted in a number of different ways.
- The traces can be zoomed in or out at the mid point of the display window by means of the zoom buttons.
- Specific areas of the display can be zoomed by right-click, drag in the display window.
- The display can be stepped by a fixed number of nsecs at a time or by the width of the display window.
- Data may be displayed in a number of formats.
- Markers can be added to the display.
- The various menubars and toolbars can be hidden by means of the icons at the bottom right of the window.
- The various menubars and toolbars are detachable. Click and drag on the gripper at the left-hand end of the bar to detach it. To return it to the window, drag it back to its correct place in the window or, more simply, double click on the gripper.

Viewing and animating the handshake circuit graph.

Note: the features described in this section are experimental and are likely to change in future releases. Not all buttons/controls are described – in the main this is because they are for internal developer use only.

If the handshake circuit graph icon (at the bottom-right corner of the controller window) is clicked, the controller window changes to that shown in Figure 2.16. It shows a graph representation of the handshake circuit compiled, and is intended to display the activity (events) happening on the various channels during the simulation.

First, you might want to change the layout style, especially if your graph does not appear nicely when using the default layout. This is done by selecting the check box entitled “Layout uses control flow”, near the bottom left corner of the window. When this check box is selected, the layout handshake circuit graph is laid out with the control flows going from top to bottom. When the check box is unselected, the graph is laid out based on data flows, with data flowing from top to bottom. The default style (data flow-based) gives a good visualisation of large circuit, especially when associated with the “Control: Gray” button (located above the check box). However, for small circuits, organising the data flow vertically does not always result in a nice layout.

Then, you might want to customise the appearance of the graph. For this, you can:

- drag&drop components or groups.
- resize groups with Shift+drag&drop.
- pan the display by dragging the background.
- zoom in/out by using the zoom icons.
- toggle the channel names and their values by using the next toggle icons.
- reduce/develop groups to show their components and sub-groups with middle-click and right-click (Right-clicking on a group reduces/develops its sub-groups; Middle-clicking on a group hides/shows its handshake components. You typically need to use right-click to fully develop groups and middle click to fully reduce them).
2.6. Simulation.

- make a group become the main viewed group by using Ctrl+Left click. Ctrl+Shift+click sets the parent of the selected group to be the main viewed group, and you can use successive Ctrl+Shift+clicks on successive parents to go back to a higher level view of the circuit.

On the left, a group of controls offer you to gray some parts of the circuit out, in order to visualise more effectively either the control flow or the data flow. The Max Dev button recursively develops every group in the circuit.

Pressing the empty icon button under the graph view develops a new button bar for accessing developpers' functionalities. The first button, “Screen Shot”, may be useful to you, as it generates a screenshot.ps file in the current directory, containing a postscript version of the viewed graph (however, the graph is usually not centered on the page and needs post-processing).

The circuit is then animated by clicking on the Animate button. The speed of the animation can be modified by means of the slider control next to the button. The animation may be stepped by means of the up/down arrows next to the current time value. As a short-cut, right-clicking on the arrows will take the simulation to the start/end of the animation. This feature is useful for rerunning the animation. The two buttons next to the time-controlling up/down arrows are stepping the animation to the next/previous viewable event.

Clicking on a channel selects it for a list of action available in the left-hand pane entitled “Selection”:

- “Delete” unselects a channel(s).
- “Dilate” expands the selection to the surrounding channels.
- “Run’til” runs the simulation until some activity appears on the selected channel(s).
- “Follow” runs until some activity on the channel and then looks for activity on the surrounding channels and expand the selection to those newly activated surrounding channels.

Figure 2.16: Channel tree and handshake circuit graph revealed.
2.6. Simulation.

When “Capture” is pressed, hovering the mouse over a handshake channel automatically selects it. Associated with the source code viewer, one can quickly see what source code corresponds to each channel.

Finally, the text box next to the Capture button is a search box that selects/unselects all the channels whose name contain the entered string (the search is run when the user presses the Return key). This search box is useful, for example, when dealing with Verilog files generated from the Balsa description: The channel numbers being the same in Verilog and in Breeze, specific Verilog channel numbers can be searched and viewed on the handshake circuit graph (or in the source code viewer, in order to link Verilog back to the original Balsa description).

<table>
<thead>
<tr>
<th>Debugging a deadlock</th>
<th>Source Code Viewer</th>
</tr>
</thead>
<tbody>
<tr>
<td>If your simulation ends up in an unexpected deadlock, you can try our “deadlock debugging helper”, currently in development (i.e. if it does not work for you, it is kind of normal). Select the channel corresponding to the latest event that happened during your simulation and run the deadlock analysis by selecting “Highlight Deadlock” in the Debug menu. It should highlight (and add in the Selection box) a list of channels which are thought to be related to the deadlock. The most useful channels for debugging the deadlock are those where a change of channel activity happens, for example when the string of highlighted channels passes from a channel where no event happened to a channel blocked with a “Request Up” event pending. You can follow this string of highlighted channels and use the source code view to locate the position of some of them, as explained below.</td>
<td></td>
</tr>
<tr>
<td>Clicking the source code viewer icon brings up a separate (initially empty) window. It is advised to click on “Show All Channel Positions”, in order to load every file and colorise keywords according to the handshake channels that are referred to in the compiled circuit. Once the source code window is open, any channel selection from the main window is reported at the bottom of the source code window. A subsequent click on the “Goto Source” button highlights the source code that correspond to the selected channel. In the other way around, it is possible to select channels that correspond to a keyword from the source code by right-clicking after the first letter of the desired keyword and choosing “Select Channels” in the contextual menu (the algorithm searches backwards from the selected character until it finds a matching channel). When the checkbox next to the “Goto Source” button is selected, any newly selected channel will be automatically reported inside the source code, as if the user pressed the “Goto Source” button after selecting the channel. Unselecting “Notebook style” displays every opened file simultaneously, next to each other.</td>
<td></td>
</tr>
</tbody>
</table>
2.6. Simulation.
3 The Balsa Language

Summary

The previous chapter introduced Balsa, but was mostly concerned with the auxiliary tools that support the Balsa environment. The language itself is small and in this section most of its major features and constructs are introduced. Later chapters discuss more advanced topics such as parameterisation and recursively defined structures (“Parameterised & Recursively Defined Circuits” on page 49) and the enclosed semantics of the choice operator (“Handshake Enclosure” on page 55). A more formal and complete, BNF style, language description can be found in Section 11.3. on page 131.

3.1. Data Types

Balsa is strongly typed with data types based on bit vectors. Results of expressions must be guaranteed to fit within the range of the underlying bit vector representation. There are two classes of anonymous types: numeric types which are declared with the `bits` keyword and arrays of other types. Numeric types can be either signed or unsigned. Signedness has an effect on expression operators and casting. Only numeric types and arrays of other types may be used without first binding a name to those types. Balsa has three separate namespaces: one for procedure and function names, a second for variable and channel names and a third for type declarations.

**Numeric types**

Numeric types incorporate numbers over the range \[0, 2^n - 1\] or \[-2^{n-1}, 2^{n-1} - 1\] depending on whether they represent either unsigned or signed and where \(n \in [1, \text{INT_MAX}]\); on a 32-bit machine \(n \in [1, 2^{32} - 1]\). Named numeric types are just aliases of the same range. An example of a numeric type declaration is:

```balsa
type word is 16 bits
```

This defines a new type `word` which is unsigned (there is no `unsigned` keyword) covering the range \([0, 2^{16} - 1]\). Alternatively, a signed type could have been declared as:

```balsa
type sword is 16 signed bits
```

which defines a new type `sword` covering the range \([-2^{15}, 2^{15} - 1]\).

Some predefined types are available in `<BalsaInstallDir>/share/balsa/types/basic.balsa`, including `byte`, `nibble`, `boolean` and `cardinal` as well as the constants `true` and `false`. Other predefined
3.1. Data Types

types may be added from time to time. Users are advised to consult the contents of the file in their particular release of the Balsa system.

Enumerated types

Enumerated types consist of named numeric values. The named values are given values starting at zero and incrementing by one from left to right. Elements with explicit values reset the counter and many names can be given to the same value, for example:

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

The value of the Violet element of Colour is 7, as is Purple. Both Grey and Gray have value 8. The total number of elements is 12. An enumeration can be padded to a fixed size by use of the over keyword:

```plaintext
type SillyExample is enumeration
    e1=1, e2
over 4 bits
```

Here 2 bits are sufficient to specify the 3 possible values of the enumeration (0 is not bound to a name, e1 has the value 1 and e2 has the value 2). The over keyword ensures that the representation of the enumerated type is actually 4 bits.

Occasionally, it is necessary when referring to an element of an enumeration to indicate the type to which that element belongs. The notation Colour’Purple specifically indicates the identifier Purple as being a member of Colour. Most users will never need this notation; about the only time it is required is when using elements of enumerations within casts and even in that case there are more transparent ways of achieving the same effect.

Enumeration types must be bound to names by a type declaration before use.

Record types

Records are bitwise compositions of named elements of possibly different (pre-declared) types with the first element occupying the least significant bit positions, e.g.:

```plaintext
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 (both types must have been previously declared). FirstBand is the first element and so represents the least significant portion of the bitwise value of a type Resistor. Selection of elements within the record structure is accomplished with the usual dot notation. Thus if R15 is a variable of type Resistor, the value of its SecondBand can extracted by R15.SecondBand.

A record can be constructed by listing its fields as a list within braces. Thus if R4K7 is a record variable of type Resistor, its value may be set:

```plaintext
R4K7 := {Yellow, Violet, Red, Gold}
```

As with enumerations, record types can be padded:

```plaintext
type Flags is record
    carry, overflow, zero, negative, int_en : bit
over byte
```

The 5-bit record is padded to 8 bits by use of the over keyword. Even in those cases where padding is not required such as in the example below, specification of the data-type required is useful because the compiler will enforce error checking to ensure that the structure is in fact what it is believed to be.

```plaintext
type Flags is record
### 3.1. Data Types

**Array types**

Arrays are numerically indexed compositions of same-typed values. An example of the declaration of an array type is:

```plaintext
type RegBank_t : array 0..7 of byte
```

This introduces a new type `RegBank_t` which is an array type of 8 elements indexed across the range [0, 7], each element being of type `byte`. The ordering of the range specifier is irrelevant; `array 0..7` is equivalent to `array 7..0`. In general, a single expression, `expr`, can be used to specify the array size: this is equivalent to a range of `0..expr-1`. Anonymous array types are allowed in Balsa, so that variables can be declared as an array without first defining the array type:

```plaintext
variable RegBank : array 0..7 of byte
```

Arbitrary bit-fields within an array can be accessed by an array slicing mechanism e.g. `a[5..7]` extracts elements `a5`, `a6`, and `a7`. As with all range specifiers, the ordering of the range is irrelevant. In general, Balsa packs all composite typed structures in a least significant to most significant, left to right manner. Array slices always return values which are based at index 0.

Arrays can be constructed by means of a list constructor or by concatenation of other arrays of the same base type:

```plaintext
variable a, b, c, d, e, f : byte
variable z2 : array 2 of byte
variable z4 : array 4 of byte
variable z6 : array 6 of byte

z4 := {a, b, c, d}                      -- array construction
z6 := z4 @ {e, f}                              -- array concatenation
z2 := (z4 @ {e, f}) [3..4]                  -- element extraction by array slicing
```

In the last example, the first element of `z2` is set to `d` and the second element is set to `e`. The parentheses are necessary to satisfy the precedence rules. Note that array slices always return values which are based at index 0. Thus in the following rather bizarre example, the first element of `z2` is assigned to `c` and the second element to `d`:

```plaintext
z2 := (({a, b, c, d} @ {e, f}) [1..4]) [1..2]  -- returns (c, d)
```

Array slicing is useful to allow arbitrary bitfields to be extracted from other datatypes. In general, the original datatype has to be cast into an array first before bitfield extract and then cast back again into the correct datatype. See “Casts” on page 32 for concrete examples.

**Constants**

Constant values can be defined in terms of an expression resolvable at compile time. Constants may be declared in terms of a predefined type otherwise they default to a numeric type. However, string constants are not allowed. Valid examples are:

```plaintext
constant minx = 5
constant maxx = minx + 10
constant hue = Red : Colour
constant colour = Colour'Green
```

Complex data type (array and record) constants may be defined:

```plaintext
constant InitArray = {1, 2, 3, 4} : MyArrayType
constant R4K7 = {Yellow, Violet, Red, Gold} : Resistor
```

The two examples above may also be written:

```plaintext
constant InitArray = MyArrayType (1, 2, 3, 4)
constant R4K7 = Resistor (Yellow, Violet, Red, Gold)
```
3.2. Data Typing Issues

Integer constants may be specified in decimal (e.g. 42), binary (e.g. 0b00101010) octal (e.g. 052) or hexadecimal (e.g. 0x2a). Note that leading zero signifies an octal constant. The underscore character “_” is allowed within numbers to improve readability (e.g. 0b_0010_1010).

Implicants – values containing don’t cares– are allowed as normal expression types and be used to define both simple numeric constants and complex data type constants. The symbol “x” denotes a single don’t care digit, and the value “?” yields an implicant matching all values of the expected type. Not all operators may be used with such implicants, working operators include as, array and record construction and #. Examples of the use of implicants are:

```
constant OddNum = 0bx1
constant DataProcInst = {?, 0b00x, ?, ?} : InstructionFormat
```

The latter could be used in decoding an instruction formatted into four fields in which it is known that data-processing type instructions are uniquely identified by the value 000 or 001 in the second field.

The main use of implicants is in matching case guards [see "case statements" on page 37].

Arrayed channels

Channels may 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 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 and each indexed channel has no relationship to the other channels in the array other than the single name they share. The syntax for arrayed channels is different to that of array typed variables making it easier to disambiguate arrays from arrayed channels. As an example:

```
array 4 of channel XYZ : array 4 of byte
```

declares 4 channels, XYZ[0] to XYZ[3], each channel is a 32-bit wide type array 0..3 of byte. An example of the use of arrayed channels was shown previously when discussing the placement of multiple structures [see “Placing multiple structures” on page 13].

3.2. Data Typing Issues

As stated previously, Balsa is strongly typed: both left-hand and right side of assignments are expected to have the same type. The only form of implicit type-casting is the promotion of numeric literals and constants to a wider numeric type. In particular care must be taken to ensure that the result of an arithmetic operation will always be compatible with the declared result type. Consider the assignment statement

```
x := x + 1
```

This is not a a valid Balsa statement because potentially the result is one bit wider than the width of the variable x. If the potential carry-out from the addition is to be ignored, the user must explicitly force the truncation by means of a cast.

```
x := (x + 1 as 32 bits)
```

The keyword as indicates the cast operation. The parentheses are a necessary part of the syntax to make the precedence of as more obvious. If the carry out of the addition of two 32-bit numbers is required, a record type can be used to hold the composite result:

```
type AddResult is record
    Result : 32 bits;
    Carry : bit;
end

variable r : AddResult

r := (a + b as AddResult)
```

The expression r.Carry accesses the required carry bit, r.Result yields the 32-bit addition result. Casts are required when extracting bit fields. Here is an example from the instruction decoder of a
3.2. Data Typing Issues

simple microprocessor. The bottom 5 bits of 16-bit instruction word contain an 5-bit signed immediate. It is required to extract the immediate field and sign-extend it to 16 bits:

```
type Word is 16 signed bits
type Imm5 is 5 signed bits

variable Instr : 16 bits—bottom 5 bits contain an immediate
variable Imm16 : Word

Imm16 := (((Instr as array 16 of bit) [0..4] as Imm5) as Word)
```

First, the instruction word, Instr, is cast into an array of bits from which an arbitrary subrange can be extracted:

```
(Instr as array 16 of bit)
```

Next the bottom (least significant) 5 bits must be extracted:

```
(Instr as array 16 of bit) [0..4]
```

The extracted 5 bits must now be cast back into a 5-bit signed number:

```
((Instr as array 16 of bit) [0..4] as Imm5)
```

The 5-bit signed number is then signed extended to the 16-bit immediate value:

```
(((Instr as array 16 of bit) [0..4] as Imm5) as Word)
```

The double cast is required because a straight forward cast from 5 bits to the variable Imm16 of type Word would have merely zero filled the topmost bit positions even though Word is a signed type. However, a cast from a signed numeric type to another (wider) signed numeric type will sign extend the narrower value into the width of the wider target type.

Extracting bits from a field is a fairly common operation in many hardware designs. In general, the original datatype has to be cast into an array of bits first before bitfield extraction. The smash operator # provides a convenient shorthand for casting an object into an array of bits. Thus the sign extension example above is more simply written

```
((#Instr [0..4] as Imm5) as Word)
```

Whilst anonymous array types are allowed, it is not always possible for the compiler to be able to deduce the appropriate type of an array constructor during a cast operation:

```
type Word32 is 32 bits
variable a, b, c, d : byte
variable Imm32: Word32

Imm32 := ({a, b, c, d} as Word32)←can’t determine type of array
```

The compiler has to be given a hint by specifying the type of the array constructor:

```
type A4_t is array 4 of byte
Imm32 := (A4_t {a, b, c, d} as Word32)
```

Here, A4_t indicates the type of the array constructor. Note that a previously declared type must be used: the following statement results in (many) compile time errors:

```
Imm32 := (array 4 of byte {a, b, c, d} as Word32) -- error
```

The following snippets illustrate the relationship between the bit ordering in array constructors and their numeric values represented by those arrays:

```
constant x = (2 as 4 bits)
print "x is: ", x, " ", #x ;
```

x is defined a being a 4 bit value; printing it as an array of bits (using the # operator) gives:

```
x is: 2 (0,1,0,0)
```
The most-significant bit is the rightmost bit element of the array – note this is contrary to the normal representation of bits in a binary number where binary 0110 would represent decimal 4. Concatenating \( x \) with another array of bits

\[
y := (#x @ (0,1) \text{ as } 8 \text{ bits});
\]

\[
\text{print } "y \text{ is: }", y, ",", \#y;
\]
gives:

\[
y \text{ is: } 34 \ (0,1,0,0,0,1,0,0)
\]

**Auto-assignment**

Statements of the form

\[
x := f(x)
\]

are allowed in Balsa. However, the implementation generates a temporary variable which is then assigned back to the variable visible to the programmer – the variable is enclosed within a single handshake and cannot be read from and written to simultaneously. Since auto-assignment generates twice as many variables as might be suspected, it is probably better practice to avoid the auto-assignment, explicitly introduce the extra variable and then rewrite the program to hide the sequential update thereby avoiding any time penalty. An example of this approach is given in “Removing auto-assignment” on page 42.

### 3.3. Control Flow and Commands

Balsa’s sparse command set is listed in Table 3.1. A more formal definition of the command syntax is given in Section 11.3. on page 131.

<table>
<thead>
<tr>
<th>command</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sync</strong></td>
<td>Control only (dataless) handshake</td>
</tr>
<tr>
<td>( &lt;- )</td>
<td>handshake data transfer from an expression to an output port</td>
</tr>
<tr>
<td>( -&gt; )</td>
<td>handshake data transfer to a variable from an input port</td>
</tr>
<tr>
<td>( := )</td>
<td>assigns a value to a variable</td>
</tr>
<tr>
<td>;</td>
<td>sequence operator</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td><strong>continue</strong></td>
<td>a null command</td>
</tr>
<tr>
<td><strong>halt</strong></td>
<td>causes deadlock</td>
</tr>
<tr>
<td><strong>loop ... end</strong></td>
<td>repeat forever</td>
</tr>
<tr>
<td><strong>loop ... while ... then ... also ... end</strong></td>
<td>conditional loop with optional initial command.</td>
</tr>
<tr>
<td><strong>for ... in ... then ... end</strong></td>
<td>structural (not temporal) iteration</td>
</tr>
<tr>
<td><strong>if ... then ... else ... end</strong></td>
<td>conditional execution, may have multiple guarded commands</td>
</tr>
<tr>
<td><strong>case ... of ... end</strong></td>
<td>conditional execution based on constant expressions</td>
</tr>
<tr>
<td><strong>select ... end</strong></td>
<td>non-arbitrated choice operator</td>
</tr>
</tbody>
</table>

*Table 3.1: Balsa Commands*
### 3.3. Control Flow and Commands

<table>
<thead>
<tr>
<th>command</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrate ... end</td>
<td>arbitrated choice operator</td>
</tr>
<tr>
<td>print &lt;args&gt;</td>
<td>if 1st arg is one of fatal, error, warning, report, print subsequent args at compile time at the appropriate error level. If 1st arg is runtime (the default) evaluate and print args during a simulation</td>
</tr>
<tr>
<td>&lt;block&gt;</td>
<td>allows inclusion of local definitions around a command and the overriding of the precedence of command composition. See Section 11.3. on page 131.</td>
</tr>
</tbody>
</table>

| Sync | sync <channel> awaits a handshake on the named channel. Circuit action does not proceed until the handshake is completed. |
| Channel assignment | <channel_out> <- <expression> The result of the expression (commonly, the value of a variable) is transferred to the named output channel. |
| | <channel_in> -> <variable> Data from the named input channel is transferred to a variable. |
| | <channel_in> -> <channel_out> Data from the named input channel is transferred to the named output channel. |
| | <channel_in> -> then <command> end The handshake on the named input channel encloses the command block. Thus the data remains valid until the command block terminates. Data on the input channel can be read more than once or assigned to multiple channels. |

| Variable assignment | <variable> := <expression> transfers the result of an expression into a variable. The result type of the expression and that of the variable must agree. |

| Sequence operator | ";" separating two commands is not merely a syntactic operator, it explicitly denotes sequentiality. Because a semicolon connects two sequenced statements of a block, it is an error to place a semicolon after the last statement in a block. Doing so is a common beginner’s error and may result the error message: expected one of tokens ‘ident [ { sync local begin continue halt loop while if case for select arbitrate print ’ |

| Parallel composition | “||” composes two commands such that they operate concurrently and independently. Both commands must complete before the circuit action proceeds. Beware of inadvertently introducing dependencies between the two commands so that neither can proceed until the other has completed. The “||” operator binds tighter than “;”. If that is not what is intended, then commands may be grouped in blocks as shown below |
| | [ CmdSeq1 ; CmdSeq2 ] || CmdPar1 |

Note the use of square brackets to group commands rather than parentheses. Alternatively, the keywords begin ... end may be used.

| Continue and Halt | continue is effectively a null command. It has no effect, but may be required for syntactic correctness in some instances. The command halt causes a process thread to deadlock. |

---

Table 3.1: Balsa Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;channel_out&gt; &lt;- &lt;expression&gt;</td>
<td>The result of the expression (commonly, the value of a variable) is transferred to the named output channel.</td>
</tr>
<tr>
<td>&lt;channel_in&gt; -&gt; &lt;variable&gt;</td>
<td>Data from the named input channel is transferred to a variable.</td>
</tr>
<tr>
<td>&lt;channel_in&gt; -&gt; &lt;channel_out&gt;</td>
<td>Data from the named input channel is transferred to the named output channel.</td>
</tr>
<tr>
<td>&lt;channel_in&gt; -&gt; then &lt;command&gt; end</td>
<td>The handshake on the named input channel encloses the command block. Thus the data remains valid until the command block terminates. Data on the input channel can be read more than once or assigned to multiple channels.</td>
</tr>
</tbody>
</table>

---

Balsa: A Tutorial Guide. V3.5 - Printed: 19/5/06
3.3. Control Flow and Commands

Looping constructs

The loop command causes an infinite repetition of a block of code. An example, summarised below, was given in the description “A single-place buffer” on page 9.

```
loop i -> x ; o <- x end
```

Finite loops may be constructed using the loop while construct\(^1\). An example of its use with a single guard is:

```
loop while x < 10 then
  x := (x+1 as byte)
end
```

Multiple guards are allowed in as shown below:

```
loop while
  x < 10 then x := (x + 1 as byte)
  | x >= 10 then x := 0
end
```

A variation on the while construct uses the also keyword to allow a final command which is executed at the end of each loop iteration if any of the guards is satisfied:

```
loop while
  x < 10 then x := (x+1 as byte)
  | x >= 10 then x := 0
  also print "Value of x is ", x
end -- loop
```

Loops with an initial command before the guard test – similar to a do...while loop found in other languages – are supported. The example below illustrates such a repetitive loop using both multiple guards and the also statement. Both are optional as in the previous while loops

```
loop
  i -> x
  while
    x < 10 then print x, " is less than 10"
    | x < 100 then print x, " is > 10 and < 100"
  also print "about to read another value"
end;
  print "exiting loop - value of x is: ", x
```

The example above also illustrates the ordering in the evaluation of the guards. For values of x less than 10, both guards are satisfied, however the language guarantees that only the command associated with the first in the list of guards will be executed. Note that the loop exits when a value greater or equal to 100 is read from the input channel i.

The equivalent of a repeat...until or a do...while loop can be specified as a simpler form of the construct above, thus:

```
loop
  print "value of x is: ", x;
  x := (x + 1 as 4 bits)
  while x <= 10
end
```

Structural iteration

Balsa has a for loop construct. Beware, in many programming languages it is a matter of convenience or style as to whether a loop is written in terms of a for loop or a while loop. This is not so in Balsa. The for loop is similar to VHDL’s for ... generate command and is used for iteratively laying out repetitive structures. An example of its use was given earlier [see “Placing multiple structures” on page 13]. An illustration of the inappropriate use of the for command is

```
for
  print "value of x is: ", x;
  x := (x + 1 as 4 bits)
  while x <= 10
end
```

1. Note that previous releases of Balsa used a different syntax for the while command; descriptions that used while loops will no longer compile correctly
3.3. Control Flow and Commands

given in “The danger of “for” loops” on page 46. Structures may be iteratively instantiated to operate either sequentially or concurrently with one another.

Conditional execution

Balsa has if and case constructs to achieve conditional execution. The if ... then ... else statement allows conditional execution based on the evaluation of expressions at run-time. Its syntax is somewhat similar to that of the while loop.

if statements

```
if condition1 then command
| condition2 then command
| condition3 then command
else CmdD
end
```

If more than guard (condition) is satisfied, then just as for a while loop, the command associated with the first mentioned guard is the one chosen. The else clause is optional.

The case statement is a multi-way decision maker that tests whether an expression matches one or more possible values.

case statements

Balsa’s case statement is similar to that in a conventional programming language. A single guard may match more than one value of the guard expression.

```
case x+y of
  1 .. 4 then o <- x
| 5 .. 10 then o <- y
  else o <- z
end
```

Case guards may be generated by means of a for statement case guard generator.

```
case s of
  for j in 1 .. 3 then
    o[j] <- i
  | 0 then
    print "Handling port 0 specially" ||
    o[0] <- i-1
end
```

The code above is equivalent to:

```
case s of
  1 then o[1] <- i
| 2 then o[2] <- i
| 3 then o[3] <- i
| 0 then
  print "Handling port 0 specially" ||
  o[0] <- i-1
end
```

The case matches in the for loop can be any general expressions resolvable at compile time. Only one for iteration variable is allowed per guard and the case matches must be disjoint from one another.

The form of case expansion illustrated in the example above is not particularly useful. It finds more application in defining the behaviour of parameterised components.

Implicants (or don’t care conditions) [see “Constants” on page 31 ] may be used in case statements:

```
procedure impl1 is
  begin
    for ; i in 1 .. 15 then
      case i of
        0bx1x then print "don’t care guard: ", i
      else
        print "covering case: ", i
```
### 3.4. Binary/Unary Operators

Balsa’s binary operators are shown in order of decreasing preference in Table 3.2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Valid types</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>record indexing</td>
<td>record</td>
<td>takes value from any type and reduces it to an array of bits</td>
</tr>
<tr>
<td>#</td>
<td>smash</td>
<td>any</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>array indexing</td>
<td>array</td>
<td>non-const index possible, can generate lots of hardware</td>
</tr>
<tr>
<td>^</td>
<td>exponentiation</td>
<td>numeric</td>
<td>only constants</td>
</tr>
<tr>
<td>not, log, – (unary)</td>
<td>unary operators</td>
<td>numeric</td>
<td>log only works on constants, returns the ceiling: e.g. log 15 returns 4 – returns a result 1 bit wider than the argument</td>
</tr>
<tr>
<td>*, /, %</td>
<td>multiply, divide, remainder</td>
<td>numeric</td>
<td>only applicable to constants</td>
</tr>
<tr>
<td>+, –</td>
<td>add, subtract</td>
<td>numeric</td>
<td>results are one or 2 bits longer than the largest argument</td>
</tr>
<tr>
<td>@</td>
<td>concatenation</td>
<td>arrays</td>
<td></td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>inequalities</td>
<td>numeric</td>
<td></td>
</tr>
<tr>
<td>=, /=</td>
<td>equals, not equals</td>
<td>all</td>
<td>comparison is by sign extended value for signed numeric types</td>
</tr>
<tr>
<td>and</td>
<td>bitwise and</td>
<td>numeric</td>
<td>Balsa uses type 1 bits for if/while guards so bitwise and logical operators are the same.</td>
</tr>
<tr>
<td>or, xor</td>
<td>bitwise or</td>
<td>numeric</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Balsa binary/unary operators

### 3.5. Description Structure

**File structure**

A typical design will consist of several files containing procedure/type/constant declarations which come together in a top-level procedure that composes the overall design. This top-level procedure would typically be at the end of a file which imports 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. 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. Thus Balsa has the same simple “declare before use” rule of C and Modula, though without any facility for prototypes. Each Balsa design file has the following simplified structure of Table 3.3\(^1\). A complete syntax for the Balsa language is given in Section 11.3. on page 131.

| (file) | ::= | (import [〈dotted-path〉])** (〈outer-declarations〉) |
| (dotted-path) | ::= | (〈identifier〉 ( . 〈identifier〉))** |
| (〈outer-declarations〉) | ::= | (〈outer-declaration〉)** |
| (〈outer-declaration〉) | ::= | 〈type〉 〈identifier〉 is (〈type-declaration〉) |
| | | | constante 〈identifier〉 = 〈expression〉 ( : 〈type〉)? |
| | | | procedure 〈identifier〉 is 〈identifier〉 ( {〈procedure-formals〉} )? |
| | | | procedure 〈identifier〉 ( {〈procedure-formals〉} )? is | localized? (〈inner-declarations〉) begin (command) end |
| | | | function 〈identifier〉 ( {〈function-formals〉} )? = 〈expression〉 ( : 〈type〉)? |
| | | | if 〈expression〉 then (〈outer-declarations〉) |
| | | | ( | 〈expression〉 then (〈outer-declarations〉) )* |
| | | | else (〈outer-declarations〉) |
| | | | end |

Table 3.3: Balsa File Structure

**Declarations**

Declarations, shown in Table 3.3, introduce new type, constant or procedure names into the global namespaces from the point of declaration 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. At the top level, only constants are this last category, however, variables and channels are included in procedure local declarations. 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.

Procedure names may be aliased. This feature is useful when instantiating particular instances of parameterised procedure definitions [see “A variable width buffer definition” on page 49].

**Procedures**

Procedures form the bulk of the a Balsa description. Each procedure has a name, a set of ports and an accompanying behavioural description. Procedure declarations follow the pattern of Table 3.4 (a

| (〈procedure-formals〉) | ::= | (〈formal-parameters〉) |
| | | | (〈formal-ports〉) |
| | | | (〈formal-parameters〉); (〈formal-ports〉) |
| (〈formal-parameters〉) | ::= | parameter 〈identifiers〉:〈type〉 |
| | | | ( ; parameter 〈identifiers〉 :〈type〉)* |
| (〈formal-ports〉) | ::= | (〈formal-port〉);(〈formal-port〉)* |

Table 3.4: Procedure Port Declarations

1. An extended form of BNF is used to describe the syntax. A terms \((a)^*\) denotes zero or more repetitions of the term \(a\) and \((a)^?\) indicates that the term \(a\) is optional
3.5. Description Structure

The complete syntax for the Balsa language is given in Section 11.3. on page 131. Each procedure may have a number of ports each of which can be connected to a channel. The sync keyword introduces nonput (dataless) channels. Both nonput and data bearing channels can be members of “arrayed channels”. Arrayed channels allow numeric/enumerated indexing of otherwise functionally separate channels. Examples of their use can be found in “Pipelines of variable width and depth” on page 50.

Procedures can also carry a list of local declarations which may include other procedures, type and constants. The keyword local is optional for declarations which immediately follow the procedure declaration since the semantics of the Balsa language ensure that they must be local to the procedure in question.

Shared procedures

Normally each call to a procedure generates separate hardware to instantiate that procedure. A procedure may be shared, in which case calls to that procedure access common hardware thereby avoiding duplication of the circuit at the cost of some multiplexing to allow sharing to occur. An example of the use of a shared procedure is given in “Sharing hardware” on page 44.

Functions

In many programming languages, functions can be thought of as procedures without side affects returning a result. However, in Balsa there is a fundamental difference between functions and procedures. Parameters to a procedure define handshaking channels that interface to the circuit block defined by the procedure. Function parameters on the other hand are merely typed identifiers. Balsa’s functions return results in a manner similar to functions in other programming languages.

Conditional ports and declarations

Declarations, including procedure and port declarations may be conditional. Examples are shown below.

```plaintext
constant debug = true

if debug then
```

| (formal-port) ::= (array (range) of)? (input | output) (identifiers) : (type) |
| (array (range) of) ? sync (identifiers) |
| if (expression) then (formal-ports) |
| ( | (expression) then (formal-ports) )* |
| else (formal-ports) |
| end |

| (range) ::= (expression) |
| (expression) .. (expression) |
| over (type) |

| (inner-declarations) ::= (inner-declaration)* |

| (inner-declaration) ::= (outer-declaration) |
| variable (identifiers) : (type) |
| (chan-opts) (array (range) of)? channel (identifiers) : (type) |
| (chan-opts) (array (range) of) ? sync (identifiers) |
| shared (identifier) is (local) ? (inner-declarations) |
| begin (command) end |
| if (expression) then (inner-declarations) |
| ( | (expression) then (inner-declarations) )* |
| else (inner-declarations) |
| end |

Table 3.4: Procedure Port Declarations
procedure p1 is
begin
print "this is the debug version of procedure p1"
end
else
procedure p1 is
begin
print "this is the production version of procedure p1"
end
end

Two definitions of \texttt{p1} are provided: the actual definition used depends on the value of the constant \texttt{debug}.

**Conditional ports**

Port declarations and variables may also be conditional. The next example is a 2-place buffer described in “Parallel composition and module reuse” on page 12. Internal channels such as that connecting the two 1-place buffers are not visible. It is occasionally necessary for debugging purposes to make an internal channel visible. It must therefore be included as a port in the procedure declaration. A conditional port declaration allows a single procedure definition to be used for both debugging and production purposes.

\begin{verbatim}
constant debug = true

procedure buf1 (input i : byte ; output o : byte) is
    variable x : byte
begin
    loop
        i -> x ; o <- x
    end -- loop
end -- procedure buf1

procedure buf2 (input i : byte; if debug then output c : byte end; output o : byte)
    if not debug then channel c : byte end
begin
    buf1(i,c) || buf1(c,o)
end -- procedure buf2
\end{verbatim}

The guard expressions must in all cases be constant at compile time/parameterised procedure expansion time.

**Variable ports**

Ports to procedures can be connected directly to variables to allow communications on the procedure’s ports to perform reads and writes to the variable.

\begin{verbatim}
procedure write_zero( output o : byte) is
begin o <- 0 end

variable v : byte
write_zero( -> v)
\end{verbatim}

In this example, zero is written into the variable \texttt{v}. Variable read/writes can be used as an abbreviated way of passing expressions to a procedure. For example:

\begin{verbatim}
c1 <- expr1 ||
c2 <- expr2 ||
c3 -> var1 ||
proc1(c1, c2, c3)
\end{verbatim}

can be replaced by
### 3.6. Examples

In this section various designs of counter are described in Balsa. In flavour, they resemble the specifications of conventional synchronous counters, since these designs are more familiar to newcomers to asynchronous systems. More sophisticated systolic counters, better suited to an asynchronous approach are described in “Systolic counters” on page 55. In this example below, the role of the clock which updates the state of the counter is taken by a dataless sync channel, named $aclk$. The counter issues a handshake request over the sync channel, the environment responds with an ACK (acknowledge completing the handshake) and the counter state is updated.

#### Modulo-16 counter

```balsa
-- count16a.balsa: modulo 16 counter
import [balsa.types.basic]

procedure count16 (sync aclk; output count : nibble) is
variable count_reg : nibble
begin
loop
    sync aclk;
    count <- count_reg;
    count_reg := (count_reg + 1 as nibble)
end
end
```

This counter interfaces to its environment by means of two channels: the dataless sync channel and the channel $count$ which outputs the current value of the counter. The internal register implied by the variable $count_reg$ and the output channel are of type $nibble$ (4 bits) which is predefined in $balsa.types.basic$. After count_reg is incremented, the result must be cast back to type nibble. Note that issues of initialisation/reset have been ignored. The Balsa simulator gives a warning when uninitialised variables are accessed.

#### Removing auto-assignment

The auto-assignment statement in the example above, although concise and expressive, hides the fact that in most back-ends, a temporary variable is created so that the update can be carried out in a race-free manner. By making this temporary variable explicit, advantage may be taken of its visibility to overlap its update with other activity as shown in the example below.

```balsa
-- count16b.balsa: write-back overlaps output assignment
import [balsa.types.basic]

procedure count16 (sync aclk; output count : nibble) is
variable count_reg, tmp : nibble
begin
    loop
```

One advantage of this form of port connection is the ability for the value of the expression to be read an arbitrary (including zero number of times) number of times. For example:

```balsa
c <- expr || proc(c)
```

If proc attempts to read $c$ more than once, deadlock will occur (because of course the write to channel $c$ from expr will only occur once). A way round the problem is the description:

```balsa
loop c <- expr end || proc(c)
```

However the resulting composition is permanent even if proc itself is non-permanent. A permanent circuit is one that never returns – the consequence being that sequential compositions of such circuits are liable to deadlock, thus the following form may be preferred:

```balsa
proc( <- expr)
```

This form of description is more efficient because of pull-style of Balsa implementations.
3.6. Examples

In this example, the transfer of the count register to the output channel is overlapped with the incrementing of the temporary shadow register. There is some slight area overhead involved in parallelisation and any potential speed-up may be minimal in this case, but the principal of making trade-offs at the level of the source code is illustrated.

### Modulo-10 counter

The basic counter description above can be easily modified to produce a modulo-10 counter. A simple test is required to detect when the internal register reaches its maximum value and then to reset it to zero.

```plaintext
-- count10a.balsa: an asynchronous decade counter
import [balsa.types.basic]

type C_size is nibble
constant max_count = 9

procedure count10(sync aclk; output count: C_size) is
  variable count_reg : C_size
  variable tmp : C_size
begin
  loop
    sync aclk;
    if count_reg /= max_count then
      tmp := (count_reg + 1 as C_size)
    else
      tmp := 0
    end
    count := count_reg;
    count_reg := tmp
  end
end
```

### A loadable up/down counter

This example describes a loadable up/down decade counter. It introduces many of the language features discussed earlier in the chapter. The counter requires 2 control bits, one to determine the direction of count, and the other to determine whether the counter should load or inc(dec)rement on the next operation. There are several valid design options; in this example, `count10b` below, the control bits and the data to be loaded are bundled together in a single channel, `in_sigs`.

```plaintext
-- count10b.balsa: an asynchronous up/down decade counter
import [balsa.types.basic]

type C_size is nibble
constant max_count = 9

type dir is enumeration down, up end

type mode is enumeration load, count end

type In_bundle is record
  data : C_size;
  mode : mode;
  dir : dir
end

procedure updown10 (input in_sigs: In_bundle; output count: C_size) is
  variable count_reg : C_size
  variable tmp : In_bundle
```
3.6. Examples

begin
loop
    in_sigs -> tmp; -- read control+data bundle
    if tmp.mode = count then
        case tmp.dir of
            down then -- counting down
                if count_reg /= 0 then
                    tmp.data := (count_reg - 1 as C_size)
                else
                    tmp.data := max_count
                end
            | up then -- counting up
                if count_reg /= max_count then
                    tmp.data := (count_reg + 1 as C_size)
                else
                    tmp.data := 0
                end
            end -- case tmp.dir
        end;
        count <- tmp.data || count_reg:= tmp.data
    end
end

The example above illustrates the use of if ... then ... else and case control constructs as well the use of record structures and enumerated types. The use of symbolic values within enumerated types makes the code more readable. Test harnesses which can be automatically generated by the Balsa system [see “Simulation.” on page 18] can also read the symbolic enumerated values. For example, here is a test file which initialises the counter to 8, counts up, testing that the counter wraps round to zero, counts down checking that the counter correctly wraps to 9.

{8, load, up} load counter with 8
{0, count, up} count to 9
{0, count, up} count & wrap to 0
{0, count, up} count to 1
{0, count, down} count down to 0
{0, count, down} count down to 9
{0, count, down} count down to 9
{1, load, down} load counter with 1
{0, count, down} count down to 0
{0, count, down} count down & wrap to 9

Sharing hardware

In Balsa, every statement instantiates hardware in the resulting circuit. It is therefore worth examining descriptions to see if there any repeated constructs that could either be moved to a common point in the code or replaced by shared procedures. In count10b above, the description instantiates two adders: one used for incrementing and the other for decrementing. Since these two units are not used concurrently, area can be saved by sharing a single adder (which adds either +1 or -1 depending in the direction of count) described by a shared procedure. The code below illustrates how count10b can be rewritten to use a shared procedure. The shared procedure add_sub computes the next count value by adding the current count value to a variable, inc, which can take values of +1 or -1. Note that to accommodate these values, inc must be declared as 2 signed bits.

The area advantage of the approach is shown by running breeze-cost: count10b has a cost of 2141 units, whereas the shared procedure version has a cost of only 1760. The relative advantage becomes more pronounced as the size of the counter increases.

--- count10c.balsa: introducing shared procedures
import [balsa.types.basic]

type C_size is nibble
constant max_count = 9
type dir is enumeration down, up end
type mode is enumeration load, count end
type inc is 2 signed bits

type In_bundle is record
  data : C_size;
  mode : mode;
  dir : dir
end

procedure updown10 (input in_sigs: In_bundle; output count: C_size) is
  variable count_reg : C_size
  variable tmp : In_bundle
  variable inc : inc

  shared add_sub is
  begin
    tmp.data := (count_reg + inc as C_size)
  end -- begin

  begin
    loop
      in_sigs -> tmp; -- read control+data bundle
      if tmp.mode = count then
        case tmp.dir of
          down then -- counting down
            if count_reg /= 0 then
              inc := -1;
              add_sub()
            else
              tmp.data := max_count
            end -- if
          | up then -- counting up
            if count_reg /= max_count then
              inc := +1;
              add_sub()
            else
              tmp.data := 0
            end -- if
        end -- case tmp.dir
      end; -- if
      count <- tmp.data || count_reg := tmp.data
    end -- loop
  end -- begin

In order to guarantee the correctness of implementations, there are a number of minor restrictions on
the use of shared procedures

- shared procedures can not have any arguments
- shared procedures can not use local channels
- if a shared procedure uses elements of the channel referenced by a select statement [see
  “Handshake Enclosure” on page 55], the procedure must be declared as local within the
  body of that select block.

A “while” loop description

An alternative description of the basic modulo-10 counter employs the while construct:

-- count10d.balsa: modulo 10 counter alternative implementation
import [balsa.types.basic]

type C_size is nibble
constant max_count = 10
procedure count10(sync aclk; output count: C_size) is
    variable count_reg : C_size
begin
    loop
        loop while count_reg < max_count then
            sync aclk;
            count := count_reg;
            count_reg := (count_reg + 1 as C_size)
        end; -- loop while
        count_reg := 0
    end -- loop
end -- begin

Pitfalls in loop terminations

Users should be be vigilant in specifying loop termination conditions correctly. The finite bit length of variables inherent in Balsa descriptions can cause problems for the unwary. Consider the following code that iterates around the loop 10 times with \( x \) taking values from 0 … 9.

\[
\text{variable } x: 4 \text{ bits} \\
\text{begin} \\
\text{loop while } x \leq 9 \text{ then} \\
\text{print } \text{"value of } x \text{ is: ", x;} \\
\text{x := (x + 1 as 4 bits)} \\
\text{end} \\
\text{end}
\]

Suppose it is now required to loop round all values of \( x \), i.e. from 0 … 15. Simply changing the comparison constant causes the code never to terminate:

\[
\text{variable } x: 4 \text{ bits} \\
\text{begin} \\
\text{loop while } x \leq 15 \text{ then -- never terminates} \\
\text{print } \text{"value of } x \text{ is: ", x;} \\
\text{x := (x + 1 as 4 bits)} \\
\text{end} \\
\text{end}
\]

The condition is always satisfied because \( x \) can only be in the range 0 … 15 wrapping round back to 0. There are two solutions:

\[
\text{variable } x: 4 \text{ bits} \\
\text{begin} \\
\text{loop} \\
\text{print } \text{"value of } x \text{ is: ", x} \\
\text{while } x < 15 \text{ then continue} \\
\text{also } x := (x + 1 as 4 bits) \\
\text{end} \\
\text{end}
\]

A more elegant solution that relies on recognizing and exploiting the wrapping back to 0 is:

\[
\text{variable } x: 4 \text{ bits} \\
\text{begin} \\
\text{loop} \\
\text{print } \text{"value of } x \text{ is: ", x} \\
\text{x := (x + 1 as 4 bits)} \\
\text{while } x /= 0 \\
\text{end} \\
\text{end}
\]

The danger of "for" loops

In many programming languages, while loops and for loops can be used interchangeably. This is not the case in Balsa: a for loop implements structural iteration, in other words, separate hardware is instantiated for each pass through the loop. The following description, which superficially appears
very similar to the while loop example of count10d previously, appears to be correct: it compiles without problems and simulation appears to give the correct behaviour. However, breeze-cost reveals an area cost of 11577, a factor 10 increase. It is important to understand why this is the case.

The for loop is unrolled at compile time and 10 instances of the circuit to increment the counter are created. Each instance of the loop is activated sequentially. The handshake circuit graph that be produced is rather unreadable; setting max_count to 3 will be produce a more readable plot.

```balsa
-- count10e.balsa: beware the "for" construct
import [balsa.types.basic]

type C_size is nibble
constant max_count = 10

procedure count10(sync aclk; output count: C_size) is
variable count_reg : C_size
begin
    loop
        for ; i in 1 .. max_count then
            sync aclk;
            count <= count_reg;
            count_reg:= (count_reg + 1 as C_size)
        end; -- for ; i
        count_reg:= 0
    end -- loop
end -- begin
```

If, instead of using the sequential for construct, the parallel for construct (for || ...) is used, the compiler will give error message complaining about read/write conflicts from parallel threads. In this case, all instances of the counter circuits would attempt to update the counter register at the same time leading to possible conflicts. If you understand the resulting potential handshake circuit, then you are well on the way to a good understanding of the methodology.

Selecting channels

The asynchronous circuit described below merges two input channels into a single output channel, it may be thought of a self selecting multiplexer. The select statement chooses between the two input channels a and b by waiting for data on either channel to arrive. When a handshake on either a or b commences, data is held valid on the input and the handshake not completed until the end of the select ... end block. This is an example of handshake enclosure and avoids the need for an internal latch to be created to store the data from the input channel; a possible disadvantage is that because of the delayed completion of the handshake, the input is not released immediately to continue processing independently. In this example, data is transferred to the output channel and the input handshake will complete as soon as data has been removed from the output channel. An example of a more extended enclosure can be found in the code for the population counter [see “A Population Counter” on page 65].

```balsa
-- merge1.balsa: unbuffered Merge
import [balsa.types.basic]

procedure merge (input a, b : byte; output c : byte) is
begin
    loop
        select a then c <- a   -- channel behaves like a variable
        | b then c <- b   -- ditto
    end -- select a
end -- loop
end -- procedure merge
```

The system designer must ensure that inputs a and b never arrive simultaneously. In many cases, this is not a difficult obligation to satisfy. However, if a and b are truly independent, the possibility of metastability failure arises just as in a synchronous system. In this case, select can be replaced by arbitrate which allows an arbitrated choice to be made. In this case, in contrast to a
3.6. Examples

synchronous implementation, there is no possibility of failure, the delay-insensitive handshake

circuit paradigm ensures that no matter how long the arbiter takes to resolve, the circuit will still
operate correctly. Arbiters are relatively expensive both in area and speed and may not be possible
in some gate array technologies and so should not employed unnecessarily.

```plaintext
-- merge2.balsa: unbuffered arbitrated MUX.
import [balsa.types.basic]
procedure merge2 (input a, b :byte; output c :byte) is
begin
  loop
    arbitrate a then c <- a -- channel behaves like a variable
      | b then c <- b -- ditto
    end -- arbitrate
  end -- loop
end -- begin
```
4 Parameterised & Recursively Defined Circuits

4.1. Summary

Parameterised procedures allow designers to develop a library of commonly used components and then to instantiate those structures later with varying parameters. A simple example is the specification of a buffer as a library part without knowing the width of the buffer. Similarly, a pipeline of buffers can be defined in the library without requiring any knowledge of the depth of the pipeline when it is instantiated.

4.2. Parameterised descriptions

The example `pbuffer1.balsa` below defines a single place buffer with a parameterised width:

```
-- pbuffer1.balsa - parameterised buffer example
import [balsa.types.basic]

-- single-place, parameterised-width buffer definition
procedure Buffer (parameter X : type ;
                 input i : X;
                 output o : X
) is
  variable x : X
begin
  loop
    i -> x ;
    o <- x
  end -- loop
end -- procedure Buffer

-- now define a byte-wide buffer
procedure Buffer8 is Buffer(byte)
```

A variable width buffer definition
4.2. Parameterised descriptions

The definition of the single place buffer given previously [see “A single-place buffer” on page 9] is modified by the addition of the parameter declaration which defines \( X \) to be of type \( \text{type} \). In other words \( X \) is identified as being a type to be refined later. Once an abstract parameter type has been declared, it can be used in later declarations and statements: for example, input channel \( i \) is defined as being of type \( X \). No hardware is generated for the parameterised procedure definition itself.

Having defined the procedure, it can be used in other procedure definitions. \( \text{Buffer8} \) defines a byte wide buffer that can be instantiated as required as shown, for example, in procedure \( \text{test1} \). Alternatively, a concrete realisation of the parameterised procedure can be used directly as shown in procedure \( \text{test2} \). Note that a test harness can be attached directly to the definition \( \text{Buffer8} \) with implied ports \( i \) and \( o \).

Pipelines of variable width and depth

The next example illustrates how multiple parameters to a procedure may be specified. The parameterised buffer element is included in a pipeline whose depth is also parameterised.

```
-- buffer2.balsa - parameterised pipeline example
import [balsa.types.basic]
import [pbuffer1]

-- BufferN: a n-place parameterised, variable width buffer
procedure BufferN (parameter n : cardinal ;
 parameter X : type ;
 input i : X ;
 output o : X)

 procedure buffer is Buffer(X)
 begin
  if n = 1 then  -- single place pipeline
   buffer(i, o)
  | n >= 2 then  -- parallel evaluation
   local array 1 .. n-1 of channel c : X
   begin
    buffer(i, c[1])  || -- first buffer
    buffer(c[n-1], o) || -- last buffer
    for || i in 1 .. n-2 then
     buffer(c[i], c[i+1])
    end
   end
  else print error, "zero length pipeline specified"
 end

-- Now define a 4 deep, byte wide pipeline.
procedure Buffer4 is BufferN (4, byte)
```

\( \text{Buffer} \) is the single place parameterised width buffer of the previous example and this is reused by means of the library statement \( \text{import[pbuffer1]} \). In this code, \( \text{BufferN} \) is defined which in a very
similar manner to the example described in “Placing multiple structures” on page 13, except that the
number of stages in the pipeline, \( n \), is not a constant but is a parameter to the definition of type
cardinal. Note that this definition includes some error checking. If an attempt is made to build a zero
length pipeline during a definition, an error message is printed.

4.3. Recursive definitions

Balsa allows a form of recursion in definitions (as long as the resulting structures can be statically
determined at compile time). Many structures can be elegantly described using this technique which
forms a natural extension to the powerful parameterisation mechanism. The remainder of this
chapter illustrates recursive parameterisation, “Balsa Design Examples” on page 65 gives other
interesting examples.

An \( n \)-way multiplexer

An \( n \)-way multiplexer can be constructed from a tree of 2-way multiplexers. A recursive definition
suggests itself as the natural specification technique: an \( n \)-way multiplexer can be split into two \( n/2 \)-
way multiplexers connected by internal channels to a 2-way multiplexer.

--- _Pmux1.balsa_: A recursive parameterised MUX definition

```balsa
import [balsa.types.basic]

procedure PMux (parameter X : type;
        parameter n : cardinal;
        array n of input inp : X;
        output out : X ) is
begin
    if n = 0 then print error,"Parameter n should not be zero"
    n = 1 then
        loop
            select inp[0] then
                out <- inp[0]
            end -- select
        end -- loop
```

Figure 4.1: Decomposition of an \( n \)-way Multiplexer
4.3. Recursive definitions

```plaintext
| n = 2 then
loop
  select inp[0] then
  out <- inp[0]
  | inp[1] then
  out <- inp[1]
end -- select
end -- loop
else
local
  channel out0, out1 : X
  constant mid = n/2
begin
  PMux (X, mid, inp[0..mid-1], out0) ||
  PMux (X, n-mid, inp[mid..n-1], out1) ||
  PMux (X, 2, {out0,out1}, out)
end -- begin
end -- if
end -- begin
```

Commentary on the code

The multiplexer is parameterised in terms of the type of the inputs and the number of channels \( n \). The code is straightforward. A multiplexer of size greater than 2 is decomposed into two multiplexers half the size connected by internal channels to a 2-1 multiplexer. Notice how the arrayed channels, \( \text{out0} \) and \( \text{out1} \) are specified as a tuple. The recursive decomposition stops when the number of inputs is 2 or 1 (specification of a multiplexer with zero inputs generates an error).

A balsa test harness

The code below illustrates how a simple Balsa program can be used as a test harness to generate test values for the multiplexer. The test program is actually rather naive.

```plaintext
-- test_pmux.balsa - A test-harness for Pmux1
import [balsa.types.basic]
import [pmux1]

procedure test (output out : byte) is
  type ttype is sizeof byte + 1 bits
  array 5 of channel inp : byte
  variable i : ttype
begin
  begin
    i:= 1;
    loop while i <= 0x80 then
      inp[0] <= (i as byte);
      inp[1] <= (i+1 as byte);
      inp[2] <= (i+2 as byte);
      inp[3] <= (i+3 as byte);
      inp[4] <= (i+4 as byte);
      i:= (i + i as ttype)
    end
    PMux5Byte(inp, out)
  end
```

Handshake multiplier

Consider a procedure that for each handshake on an input port generates \( n \) handshakes on an output port. A simple solution would use the for construct, but a more elegant (and less expensive) approach is to use the recursive approach.

If \( n \) is even, the repeater can be composed from two \( n/2 \) repeaters. If \( n \) is odd, the repeater can be composed from two \( n/2 \) repeaters together with an additional extra handshake.
-- GenHS. A recursive procedure generating n Handshakes for each call
import [balsa.types.basic]

procedure repeat (parameter n : cardinal; sync o ) is
begin
    if n = 0 then
        print error, "Repeat n must not be 0"
    | n = 1 then
        sync o
    else
        local
            shared doNext is begin repeat(n/2, o) end
            begin
                if (n as bit) then -- n is odd
                    sync o
                end ;
            doNext () ; doNext ()
        end
    end
end

procedure Gen11 is repeat(11)

procedure test (sync i, o) is
begin
    loop
        sync i;
        Gen11(o) -- Generate 11 Handshakes
    end -- loop
end

A shared procedure doNext is responsible for the recursive call of repeat with half the repetition count. Note that doNext is local to the main repeat procedure.

4.4.  Pitfalls with Parameterised Procedures.

A parameterised procedure often contains a choice in its body to instantiate one of several options depending on parameters that are defined in its call. It is possible that compile time errors in the parameterised procedures will not be revealed until particular parts of the code are required. Thus, in the following example, if pproc is compiled as library component no error is reported; further if it is instantiated with n=1, the code is also compiled without error. However, if the procedure is called with n=2 as in procedure p2, a compile error will be reported. The code is a precis of code that existed in an example in previous editions of the Balsa Manual. The point is that errors in the descriptions of parameterised procedures may not reveal themselves immediately.

procedure pproc(
    parameter n : cardinal ;
    parameter w: cardinal ;
    output o : w bits
) is
begin
    if n = 1 then
        o <- (1 as w bits)
    else
        o <- (2 as w ) -- Note this should give a compile time error
    end
end

-- procedure p1 is pproc(1, 8) -- this will compile
-- procedure p2 is pproc(2, 8) -- this will not compile
4.4. Pitfalls with Parameterised Procedures.
Handshake Enclosure

5

5.1. Summary

Normally handshakes are points of synchronisation for assignments between channels or assignments between channels and variables. A transfer is requested and when all parties to the transaction are ready, the transfer completes. After completion of the handshake, the data provider is free to remove the data. If the data on a channel is required more than once, it must be stored in a variable. Balsa has two language constructs that allow the handshake on a channel to be held open whilst a sequence of actions completes. The handshake is said to enclose the other commands.

There are several implications of handshake enclosure:

- since data is not removed until the end of the handshake enclosure, intermediate storage of the data is not required
- data does not have to be read once and only one: it may be read many times or indeed never at all without causing deadlock.
- the enclosing handshake does not complete until all its enclosed commands complete: this has performance implications since the tree of handshakes connected to the enclosing handshake cannot themselves complete.

Handshake enclosure can be achieved by use of the select command or by assigning channels into a command using the syntax: <channels> -> then command end. An example of the use of select was illustrated in the description of a merge circuit in “Selecting channels” on page 47. In this example, the fact that the handshake on the chosen input channel is held open allows a buffer-free description to be used – a more natural description of the mux-like structure than one which includes a storage element. One side effect of the select command is that a subcircuit with passive ports is generated – Balsa normally generates active ported circuits.

5.2. Systolic counters

A more complex example illustrating handshake enclosure is a description of systolic counters originally described by Kees van Berkel [1]. These elegant counters possess the properties of constant response time and a constant upper bound on power consumption regardless of the length
of the counter. The basic idea is to recursively divide a modulo-n counter into a head counter and a tail n/2 counter as shown in Figure 5.1.

counter n

\[
\begin{array}{c}
\text{head:} \\
\text{co/ce} \\
\end{array}
\]

\[
\begin{array}{c}
a_{\text{left}} \\
b_{\text{left}} \\
\end{array}
\quad
\begin{array}{c}
a_{\text{right}} \\
b_{\text{right}} \\
\end{array}
\quad
\begin{array}{c}
\text{counter n/2} \\
\end{array}
\]

\textbf{Figure 5.1:} Counter Decomposition

The derivation of the cells is given in van Berkel. The head cell is either a \textit{Count-Odd} cell (CO) for odd n or a \textit{Count-Even} cell (CE) for even n. For CE cells, the head cell effectively doubles each \(a_{\text{right}}\) communication of the n/2 counter over its left-hand \(a_{\text{left}}\) channel then passing \(b_{\text{right}}\) over \(b_{\text{left}}\) after the n communications along \(a_{\text{left}}\). A Count-Odd cell issues an extra handshake to its left prior to handshake from \(b_{\text{right}}\) to \(b_{\text{left}}\). A special base case count-1 cell initiates a handshake on its \(a_{\text{left}}\) port followed by a handshake on its \(b_{\text{left}}\) port.

Note that ports \(a_{\text{left}}\) and \(b_{\text{left}}\) are active ports whereas \(a_{\text{right}}\) and \(b_{\text{right}}\) are passive ports. The counter is "primed" by handshakes flowing from right to left from the count-1 cell. The head cell chooses between handshakes arriving on \(a_{\text{right}}\) and \(b_{\text{right}}\). The sequencing implicit in the description guarantees mutually exclusive use of the channels so that a non-arbitrated select construct may be used to implement the choice. The architecture of the counter is somewhat similar to that described in Section, “Handshake multiplier,” on page 52.

The descriptions of the basic cells are:

\begin{verbatim}
-- count-even cell
procedure ce(sync a_left, a_right, b_left, b_right) is
begin
  loop
    select a_right then
      sync a_left ; sync a_left
      b_right then
      sync b_left
    end
  end
end

-- count-odd cell
procedure co(sync a_left, a_right, b_left, b_right) is
begin
  loop
    select a_right then
      sync a_left ; sync a_left
      b_right then
      sync a_left ; sync b_left
    end
  end
end

-- count-1 cell
procedure c1(sync a, b) is
end
\end{verbatim}
5.2. Systolic counters

Consider the case of a modulo-11 counter. It can be decomposed as:

\[ 11 = 1 + 2 \times 5 = 1 + 2 \times (1 + 2 \times 2) = 1 + 2 \times (1 + 2 \times (2 \times 1)) \]

The composition of the basic cells is shown in Figure 5.2. The description of the counter is simple:

```
begin
  loop
    sync a; sync b
  end
end
```

![Figure 5.2: Modulo-11 Systolic Counter](image)

The behaviour of the circuit is shown in the trace of Figure 5.3. An intriguing feature of this description is that there appears to be no state-holding variables defining the current state of the counter. The answer to this paradox is that the state of the counter is distributed over the control logic defined by the circuit description.

![Figure 5.3: Behaviour of a Modulo-11 Systolic Counter](image)

All even cells

The enclosed, non-buffered, semantics of the Balsa select statement may leads to interesting patterns of behaviour. This is not obvious from the previous modulo-11 counter example. However, it is

1. The traces have been rearranged vertically to make the behaviour clearer.
exposed by a modulo-8 counter composed entirely from count-even stages (plus a count-1 stage). Each CE module awaits a handshake on its right-hand \(a\) port. Upon initiation of this handshake, the module issues a handshake to its left-hand \(a\) port. However, this handshake cannot immediately complete because the left-hand receiving port handshake encloses a command to issue a handshake to its left. Thus the operation proceeds from the count-1 cell at the extreme right issuing a handshake which ripples through to the interface \(a\) port on the extreme left. The acknowledgement ripples back to the count-1 cell whereupon the handshake on the \(b\) channel ripples from right to left. As can be seen in Figure 5.4, the result is a highly sequential mode of operation.

![Figure 5.4: Behaviour of a Modulo-8 Systolic Counter](image)

**All odd cells** A modulo-15 counter composed entirely from count-odd stages exhibits similar behaviour as shown in Figure 5.5. However, it is possible to rewrite the description of the count-odd stage to introduce extra concurrency:

```
procedure coDec(sync a_left, a_right, b_left, b_right) is
begin
  loop
    sync a_left;
    select a_right then
      sync a_left
    b_right then
```

![Figure 5.5: Behaviour of a Modulo-15 Systolic Counter](image)
5.2. Systolic counters

Here the extra handshake to the left has been taken outside of the select command. All stages can simultaneously issue a handshake to their left and then await the incoming handshake which just been initiated on its right. As can be seen from Figure 5.6, there is a significant change in the pattern of behaviour. Whether or not this translates to a change in performance depends on the relative speeds of the handshake components in the synthesised circuits.

![Figure 5.6: Behaviour of a Modulo-15 Systolic Counter](image)

The effects of the enclosed behaviour of the select command may be mitigated by decoupling the reading of the selected channel from subsequent actions. It is necessary to record which of the two channels \(a\) or \(b\) the handshake actually arrived on. This may be done by identifying the channel in a single bit register as shown below:

```plaintext
procedure ceVar(sync a_left, a_right, b_left, b_right) is
  variable x : bit
  begin
    loop
      select a_right then
        x := 0
      |      b_right then
        x := 1
      end ;
      case x of
        0 then sync a_left ; sync a_left
        | 1 then sync b_left
      end
    end
  end

Substituting this new version of the count-even cell in the modulo-8 counter results in the behaviour of Figure 5.7. As can be seen, the channel activity has an entirely different characteristic from the counter of Figure 5.4.

A decoupled all even cell
The previous examples explicitly enumerated the constituent modules to emphasise how the counters were composed. A more generic approach is to define a parameterised counter. The example below also uses a conditional declaration to choose between a count-even module with enclosed behaviour and one with a decoupled behaviour. It also offers a choice between decoupled and non-decoupled implementations.

```vhd
-- parameterised systolic counter with choice of decoupled modules.
procedure countN (  
    parameter isDecoupled : bit ;  
    parameter n : cardinal ;  
    sync a, b  
) is  
  sync a_int, b_int  
begnin  
  if n = 0 then print error, “Parameter n should not be zero”  
  |  n = 1 then c1(a, b)  
  else  
  if (n as bit) then -- odd  
  if isDecoupled then  
  coDec(a, a_int, b, b_int)  
else  
  co(a, a_int, b, b_int)  
  end -- if isDecoupled  
else  
  if isDecoupled then  
  ceDec(a, a_int, b, b_int)  
else  
  ce(a, a_int, b, b_int)  
  end -- if isDecoupled  
end -- if n = 0  
end -- procedure countN  

procedure Count11PND is countN(true,11)  
procedure Count11PD is countN(false,11)
```

5.3. **Active enclosure**

The `select` command provides a means of choosing between a number of input channels. It also has two significant side effects.
5.4. Use of enclosed channels.

Enclosed channels act rather like variables; there are pitfalls in their use: they may be assigned to other channels.

```plaintext
procedure ex2 (
  input i : byte ;
  output o1 : byte ;
  output o2 : byte
)

type Flags is record
  V, C, Z, N : bit
end

procedure SetConditions (
  input flags : Flags;
  output conditions : Conditions
) is
begin
loop
  flags -> then
    conditions <- {
      flags.N or flags.Z,
      flags.C,
      flags.Z,
      flags.V,
      not flags.N,
      (not flags.N and flags.V) or (flags.N or not flags.V)
    }
end
end
end
```

- an input port attached to a `select` command is a passive rather than an active port. The ability to coerce a port to be passive (rather than active) should normally be of little concern to users except when interfacing to external circuits.

- the handshake behaviour, as discussed earlier, has enclosing semantics bringing the advantages of unbuffered channel access and the ability to read a channel multiple times as well as the disadvantages illustrated in the previous examples.

Since selection can be applied to any number channels (including a single channel), users trying to exploit the advantages of enclosed selection may be tempted to use `select` promiscuously. Resist the temptation, there are some disadvantages: constructs such as:

```plaintext
select a then cmd1 ; select a then cmd2
```

results in non delay-insensitive behaviour. Furthermore there are inefficiencies associated with the use of passive-ported structures within the generally pull-driven circuits generated by Balsa. Better is to use active enclosure and to reserve the use of `select` for those occasions when choice is genuinely required.

Active enclosure – so called because it generates an active-ported structure – is of the form:

```plaintext
<channels> -> then <command> end
```

As example, consider a channel bearing the flags from the ALU of a processor. The conditions corresponding to various conditional branches can be computed as shown below.

```plaintext
type Flags is record
  V, C, Z, N : bit
end

type Conditions is record
  LowerOrSame, CarrySet, Zero, Overflow, Plus, LessThan : bit
end

procedure SetConditions (
  input flags : Flags;
  output conditions : Conditions
) is
begin
loop
  flags -> then
    conditions <- {
      flags.N or flags.Z,
      flags.C,
      flags.Z,
      flags.V,
      not flags.N,
      (not flags.N and flags.V) or (flags.N or not flags.V)
    }
end
end
end
```
5.4. Use of enclosed channels.

) is
  variable x1, x2 : byte
begin
  loop
    select i then
      o1 <- i;
      o2 <- i
    end
  end
end

When copying the value on an enclosed channel to a variable, an assignment operator must be used:

procedure ex3 (input i : byte)
) is
  variable x1, x2 : byte
begin
  loop
    select i then
      x1 := i;
      x2 := i;
      print "vars are: ", x1, " ", x2
    end
  end
end

Because enclosed channels act like variables, the following description is not correct:

-- this example illustrates incorrect of channels
-- variables can't read them in the normal way
-- see example ex3.balsa for the correct method.
procedure ex4 (input i : byte)
) is
  variable x1, x2 : byte
begin
  loop
    select i then
      i -> x1;  -- incorrect
      i -> x2;  -- incorrect
      print "vars are: ", x1, " ", x2
    end
  end
end

Channels within an active enclosed block also act like variables:

procedure ex6 (input i : byte)
) is
  variable x1, x2 : byte
begin
  loop
    i -> then
      x1 := i;
      x2 := i;
      print "vars are: ", x1, " ", x2
    end
  end
end
5.4. Use of enclosed channels.
5.4. Use of enclosed channels.
6.1. Summary

In this chapter, several moderate size examples are presented that illustrate many of the language features that have been discussed previously. Many of these descriptions are taken from larger examples that have been fabricated.

6.2. A Population Counter

This design counts the number of bits set in a word. It comes from the requirement in an AMULET processor to know the number of registers to be restored/saved during LDM/STM (Load/Store Multiple) instructions.

The approach taken is to partition the problem into two parts as shown in Figure 6.1. Initially, adjacent bits are added together to form an array of 2-bit channels representing the numbers of bits that are set in each of the adjacent pairs. The array of 2-bit numbers are then added in a recursively defined tree of adders.

```balsa
-- popcount: count the number of bits set in a word
import [balsa.types.basic]

procedure AddTree {
  parameter inputCount : cardinal;
  parameter inputSize : cardinal;
  parameter outputSize : cardinal;
  array inputCount of input i : inputSize bits;
  output o : outputSize bits
} is
begin
  if inputCount = 1 then
    i[0] -> then o <- (i[0] as outputSize bits) end
  -- or one of the following (since i & o channels are the same width)
  -- i[0] -> then o <- i[0] end
  -- i[0] -> o
  | inputCount = 2 then
  i[0], i[1] -> then
```

---

Balsa: A Tutorial Guide. V3.5 - Printed: 19/5/06
6.2. A Population Counter

\[
o \leftarrow (i[0] + i[1]) \text{ as outputSize bits}
\]
end
else
local
    constant lowHalfInputCount = inputCount / 2
    constant highHalfInputCount = inputCount - lowHalfInputCount
channel lowO, highO : outputSize - 1 bits
begin
    AddTree (lowHalfInputCount, inputSize, outputSize - 1, 
        i[0..lowHalfInputCount-1], lowO) ||
    AddTree (highHalfInputCount, inputSize, outputSize - 1,
        i[lowHalfInputCount..inputCount-1], highO) ||
    AddTree (2, outputSize - 1, outputSize, {lowO, highO}, o)
end
end

procedure PopulationCount ( 
   parameter n : cardinal; 
   input i : n bits; 
   output o : log (n+1) bits 
) is 
begin
    if n % 2 = 1 then
        print error, "number of bits must be even"
    end; -- if
    loop
        i -> then
            if n = 1 then
                o <- 1 
            | n = 2 then
                o <- (#i[0] + #i[1]) \text{ -- add bits 0 and 1}
            else
\]

Figure 6.1: Structure of a bit-population counter
6.3. A Balsa shifter

General shifters are an essential element of all microprocessors including the AMULET processors. The following description forms the basis of such a shifter. It implements only a rotate right function, but it is easily extensible to other shift functions. The main work of the shifter is local procedure \texttt{rorBody} which recursively creates sub-shifters capable of optionally rotating 1, 2, 4, 8 etc bits. The structure of the shifter is shown in

\begin{verbatim}
import [balsa.types.basic]

procedure ror (parameter X : type;
  input d : sizeof X bits;
  input i : X;
  output o : X)
  is
begin
  loop
local
  constant pairCount = n - (n / 2)
array pairCount of channel addedPairs : 2 bits
begin
  for \( || c \) in 0..pairCount-1 then
    -- add bits \( c*2 \) and \( c*2 +1 \)
    addedPairs[c] <= (#i[c*2] + #i[(c*2)+1])
end ||
  AddTree (pairCount, 2, log (n+1), addedPairs, o)
end -- begin
end -- if
end -- select
end -- loop
end -- begin

procedure PopCount16 is PopulationCount (16)
procedure PopCount2 is PopulationCount (2)
procedure PopCount14 is PopulationCount (14)
-- procedure PopCount3 is PopulationCount (3)
\end{verbatim}

Commentary on the code

Procedures \texttt{AddTree} and \texttt{PopulationCount} are parameterised. \texttt{PopulationCount} can used to count the number of bits set in any sized word. \texttt{AddTree} is parameterised to allow a recursively defined adder of any number of arbitrary width vectors.

Enclosed Selection

The semantics of the enclosed handshake of \texttt{select} allow the contents of the input \( i \) to be referred to several times in the body of the \texttt{select} block without the need for an internal latch. An in-depth discussion of the implications of enclosed selection is given in “Handshake Enclosure” on page 55.

Avoiding deadlock:

Note that the formation of the sum of adjacent bits is specified by a parallel \texttt{for} loop.

\begin{verbatim}
for \( || c \) in 0..pairCount-1 then
  addedPairs[c] <= (#i[c*2] + #i[(c*2)+1])
\end{verbatim}

It might be thought that a serial \texttt{for ;} loop could be used at, perhaps, the expense of speed. This is not the case: the system will deadlock illustrating why designing asynchronous circuits requires some real understanding of the methodology. In this case the adder to which the array of \texttt{addPairs} is connected requires pairs of inputs to be ready before it can complete the addition and release its inputs. However, if the sum of adjacent bits is computed serially, the next pair will not be computed until the handshake for the previous pair has been completed -- which is not possible because \texttt{AddTree} is awaiting all pairs to become valid: result deadlock!
select d then
  local
    constant typeWidth = sizeof X

  procedure rorBody (parameter distance : cardinal;
    input i : X;
  ) is
    output o : X
  ) is
  ) is
  procedure rorStage (output o : X)
  ) is
  begin
    select i then
      if #d[log distance] then
        o <- (#i[typeWidth-1..distance] @
          #i[distance-1..0] as X)
      else
        o <- i
      end -- if
    end -- select
  end -- begin
  channel c : X
  begin
    if distance > 1 then
      rorStage (c) ||
      rorBody (distance / 2, c, o)
    else
      rorStage (o)
    end -- if
  end -- begin
  begin
    rorBody (typeWidth / 2, i, o)
  end -- begin
  end -- select

Figure 6.2:
### 6.4. An Arbiter Tree

This example builds a parameterised arbiter. This circuit forms part of a simple DMA controller described by Bardsley [5]. The architecture of an 8-input arbiter is shown. *ArbFunnel* is a parameterisable tree composed of two elements: *ArbHead* and *ArbTree*. Pairs of incoming sync requests are arbitrated and combined into single bit decisions by *ArbHead* elements. These single bit channels are then arbitrated between by *ArbTree* elements. An *ArbTree* takes a number of decision bits from each of a number of inputs (on the i ports) and produces a rank of 2-input arbiters to reduce the problem to half as many inputs each with 1 extra decision bit. Recursive calls to *ArbTree* reduce the number of input channels to one (whose final value is returned on port o).

--- **ArbHead**: 2 way arbcall: with channel id. output

```v vhdl
procedure ArbHead (
  sync i0, i1;
  output o : bit
) is begin loop
  arbitrate i0 then o <- 0
  | i1 then o <- 1
end end
```

--- **ArbTree**: a tree arbcall which outputs a channel number --- prepended onto the input channel's data. (invokes itself recursively to make the tree)

```v vhdl
procedure ArbTree (
  parameter inputCount : cardinal;
  parameter depth : cardinal; -- bits to carry from inputs
  array inputCount of input i : depth bits;
  output o : (log inputCount) + depth bits
) is begin
  case inputCount of
    0, 1 then print error, "can't build an ArbTree with fewer than 2 inputs"
```
6.4. An Arbiter Tree

| 2 then loop
|   arbitrate i[0] then o <- ( #i[0]) @ #0 as depth + 1 bits
|   i[1] then o <- ( #i[1]) @ #1 as depth + 1 bits
| end
| end
else local
constant halfCount = inputCount / 2
constant halfBits = depth + log halfCount
channel l, r : halfBits bits
begin
ArbTree (halfCount, depth, i[0 .. halfCount-1], l) ||
ArbTree (inputCount - halfCount, depth,
   i[halfCount .. inputCount-1], r) ||
ArbTree (2, halfBits, {l,r}, o)
end -- local
end -- case inputCount
end -- procedure ArbTree

-- ArbFunnel: build a tree arbcall
procedure ArbFunnel (parameter inputCount : cardinal;
array inputCount of sync i;
output o : log inputCount bits)
is
  constant halfCount = inputCount / 2
begin
  if ( 2 ^ log(inputCount)) /= inputCount then
    print fatal, "No of Inputs (",  inputCount, ") must be a power of 2"
  end; -- if ( log ( inputCount ) 2 ) /= inputcount
  if inputCount < 2 then
    print error, "can't build an ArbFunnel with fewer than 2 inputs"
  | inputCount = 2 then
    ArbHead (i[0], i[1], o)
  | inputCount > 2 then
    local

Figure 6.3: 8-input arbiter
6.5. A Stack Description

An n-place stack can be decomposed into a single place buffer at the head of the stack together with a n-1 stack as shown Figure 6.4.

![Diagram of a recursively defined stack]

**Figure 6.4:** A Recursively Defined Stack

Operations on the stack consist of either pushing data on channel i or popping data on channel o. The operations are assumed to be sequenced, so no arbitration is required between a push and a pop. A first sight, it appears as if a select command choosing between requests on the push channel, i, and the pop channel, o, is what is needed. Unfortunately, Balsa does not support output selection, that is the ability to choose between output channels. It is therefore necessary to supply an extra sync channel to indicate that a pop is required. The stack therefore waits for either a push request implicit in the pushData channel, i, or a pop request on the sync channel “pop”. In the latter case, data is transferred to the popData channel, o, from the top of stack buffer and the pop request is propagated down the stack.

```balsa
import [balsa.types.basic]

-- The stack description
procedure stack (parameter depth : cardinal ;
                input i : byte ;
                output o : byte ;
```
A single-place stack is just a simple buffer and this case is tested first, otherwise the stack is decomposed into the parallel composition of a single buffer and a stack of depth n-1. The decomposition stops when a single-place stack is reached. The top of stack buffer and the internal stack are connected by local channels nextI, nextO and nextPop. Notice that in the case of a pop request, the request is forwarded to the internal stack (sync nextPop) in parallel with reading the output of that internal stack (nextO -> x).

Commentary
on the code

6.6. A Simple Processor – The Manchester SSEM (The Baby)

This example describes a simple processor – the SSEM.

The Small-Scale Experimental Machine, known as SSEM, or the "Baby", was designed and built at the University of Manchester, and made its first successful run of a program on June 21st 1948. It was the first machine that had all the components now classically regarded as characteristic of the basic computer. Most importantly it was the first computer that could store not only data but any (short!) user program in electronic memory and process it at electronic speed. (Also, the electronic memory was a true Random Access Memory (RAM). A photograph of a reconstruction of the original machine is shown in Figure 6.5. More details of the history of the machine can be found in <www.computer50.org>.

The machine is a 32 bit processor with 2’s complement number representation allowing up to 256 banks of a 32 word memory. Each memory bank was in the form of a CRT, there being only one bank in the original implementation. The machine possessed a single register accumulator, a program counter (referred to in the original design as CI, although the description below uses the more usual of name of PC) and an instruction register IR which went under the name of PI in the original design.
The original machine had only 7 instructions:

- **JMP**: \( PC := M[\text{Addr}] \)  
  indirect jump
- **JRP**: \( PC := PC + M[\text{Addr}] \)  
  relative jump
- **LDN**: \( ACC := -M[\text{Addr}] \)  
  load negative
- **STO**: \( M[\text{Addr}] := ACC \)  
  store result
- **SUB**: \( ACC := ACC - M[\text{Addr}] \)  
  subtract
- **TEST**:  
  if ACC<0 then \( PC := PC +1 \);  
  skip
- **STOP**:  
  halt

The format of the instruction word is shown in Figure 6.6:

![SSEM instruction format](image)

The CRT address refers to the CRT bank and is always 0 in this description. The line address is the memory address. The operation of the machine is as follows:

- \( PC := PC +1 \)
- \( IR := IR[PC] \)
- Decode and execute instruction
  - memory operand fetch if required
- Repeat until STOP instruction

Note that the first instruction is at address 1.
type LineAddress is 5 bits
type CRTAddress is 8 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 : LineAddress;
  CRTNo : CRTAddress;
  Func : SSEMFunc
over word

procedure SSEM (  
  -- Memory interface, MemA,MemRNW,MemR,MemW
  output MemA : LineAddress;
  output MemRNW : bit;
  input MemR : word;
  output MemW : word ;
  -- Signal halt state
  sync halted
) is

  variable ACC, ACC_slave : word
  variable IR : word
  variable PC, PC_step : LineAddress
  variable MDR : word
  variable Stopped : bit

  -- Extract an address from a word
  function ExtractAddress (wordVal : word) =
    (wordVal as SSEMInst).LineNo

  shared WriteExtractedAddress is begin
    MemA <- ExtractAddress (IR) end

  shared MemoryWrite is
    begin MemRNW <- 0 || WriteExtractedAddress ()
    || MemW <- ACC_slave end

  shared MemoryRead is
    begin MemRNW <- 1 || WriteExtractedAddress ()
    || MemR -> MDR end

  -- Fetch an instruction IR := M[PC]
  procedure InstructionFetch is
    begin MemRNW <- 1 || MemA <- PC || MemR -> IR end

  shared ZeroACC is begin ACC := 0 end
  shared ZeroPC is begin PC := 0 end
  shared SUB is begin
    MemoryRead (); ACC_slave := (ACC - MDR as word)
  end

  -- Modify the programme counter PC
6.6.  A Simple Processor – The Manchester SSEM (The Baby)

shared IncrementPC is begin
  PC := (PC + PC_step as LineAddress) end
shared AddMDRToPC is begin
  PC_step := ExtractAddress (MDR); IncrementPC () end

Decode and execute procedure

--missing instruction aliased to sub procedure DecodeAndExecuteInstruction is
begin
  case (IR as SSEMInst).Func of
    JMP then MemoryRead (); ZeroPC (); AddMDRToPC ()
    JRP then MemoryRead (); AddMDRToPC ()
    LDN then ZeroACC (); SUB ()
    STO then MemoryWrite ()
    SUB .. SUB_alt then SUB ()
    TEST then
      if #ACC [31] == -ve? then IncrementPC () end  -- PC_step should already be 1
    STOP then Stopped := 1
  end;
  ACC := ACC_slave
end

main body
begin
  ZeroACC () || ZeroPC () ||
  Stopped := 0; -- reset initialisation
  loop while not Stopped then
    PC_step := 1;
    IncrementPC ();
    InstructionFetch ();
    DecodeAndExecuteInstruction ();
  end; -- loop
  sync halted
  -- halt -- STOP instruction effect
end

Simulation
The processor has to be coupled to a memory model containing a program for it to be simulated. Section, “Memory models,” on page 96 explains how this may be done and contains a test harness for running the gcd program that was the first program to be executed on the original SSEM.
6.6. *A Simple Processor – The Manchester SSEM (The Baby)*
7

Building test harnesses with Balsa

7.1. Overview

When simulating Balsa descriptions, a test harness is usually necessary to provide input stimuli and to display output results. In previous versions of Balsa, these test harnesses have usually either been written in LARD (with the old LARD based simulation system) or been described in a test description file for breeze-sim. Neither of these solutions has offered a seamless route for simulating Balsa together with a realistically complicated test harness. To address this problem, additions have been made to the Balsa language to allow test harnesses to be constructed entirely using Balsa. The simple test harness construction capabilities present in balsa-mgr have similarly been changed to generate Balsa test harnesses (rather than LARD or .testdesc based test harnesses) using a new utility: balsa-make-test.

In order to allow Balsa to be used to capture the kind of complicated test harnesses which were only previously possible with LARD, two major additions have been made to the Balsa language and simulation systems: builtin types and builtin functions.

Builtin types

A new class of types known as builtin has been introduced to represent simulation objects such as files and strings. For example, the declaration:

type File is builtin

can be found in the new library file [balsa.sim.fileio]. This declaration introduces a new type File which represents a file access object in a similar way that the type FILE * represents a file in C. Builtin functions can be declared which generate values of builtin types. These values can then be passed around the Handshake Circuit generated by balsa-c as 64bit pointer values which, in simulation, are pointers to a BalsaObject structure (described later). Builtin-typed values are reference counted by the simulation system and so need not be explicitly deallocated by the user. In most respects, builtin types and their values can be handled just like any other type or value in Balsa, they can be used as parameters, as types of parameters, ports, and variables and also as the return type for functions. There are a few restrictions on the use of values of builtin types, however. Such values can never be cast to another type or have any arithmetic operation performed on them. These restrictions allow builtin values to never be interchanged with non-builtin values. Such an interchange could have disastrous results for a simulation.
7.1. Overview

**Built-in Functions**

In order to manipulate built-in typed values, a new form of function declaration has been introduced to allow Balsa language functions to have underlying C language implementations for the purposes of simulation. The mechanism for calling these C functions allows the same compiled C description to be used by both breeze-sim and by netlist simulation tools which support compiled plugin modules. So far, interfaces (using the new Balsa tool `balsa-sim-verilog`) to the Verilog simulators Icarus Verilog, Verilog-XL, NC Verilog, Modelsim and Synopsys VCS have been implemented.

Each built-in function must have a declaration in Balsa as well as a definition in C. In Balsa, a typical built-in function declaration looks like this:

```balsa
function FileOpen (fileName : String; mode : FileMode) is builtin : File
```

This function is provided by the `balsa.sim.fileio` and is the sole function responsible for creating File type objects. A typical use of File and FileOpen might be:

```balsa
variable f : File
... begin ...
f := FileOpen ("my_file", read)
```

Notice that there is now a “true” string type in Balsa, and that a value of this type is used as the fileName argument to FileOpen. Redefinition of strings as built-in typed-values allows them to be much more useful in Balsa than their previous role of literal arguments to the “print” statement. The type FileMode used for the argument mode is just a simple Balsa enumeration type, showing that both built-in and simple bitwise types values can be passed into built-in functions.

In the Balsa distribution, the file `share/balsa/sim/fileio.c` provides the implementation for FileOpen (and the other file manipulation functions). The HelloWorld example later in this section will explain the structure of such a C file. Balsa-mgr can be used to produce Makefiles which can compile both the Balsa and the C, this is demonstrated in a later section.

Built-in functions can also have parameters in the same way as parameterised procedures to allow the typing of their ports to be varied between instances of the function. The simulation system handles these parameters by passing C language representations of Balsa values and types to the simulation C code. In this way it is possible to define built-in functions which can process arbitrarily complicated aggregate types. This feature is used by the function `ToString` provided by `balsa.types.builtin`.

**Strings**

The String type is unusual in that the user can insert literal strings into a Balsa description without explicitly calling a function. For example:

```balsa
variable s : String
... begin ...
s := "AA"
```

must create a string containing the text “AA” and then assign that String-typed object into the variable s. To create the string, a call to a built-in function is necessary as the simulation system must create an object to hold the string. To allow this close coupling of the String type with the compiler, String is defined in the library `balsa.types.builtin` which is implicitly imported into all Balsa descriptions. String typed values are created by a call to the “String” function (notice that this name is distinct from the type String as Balsa has separate name-spaces for types and function names). The print statement in Balsa has also been modified to make use of built-in functions rather than specialised simulation handshake components. A statement such as:

```balsa
print "Hello", v
```

is now implemented as (not showing the calls to String, the “sink” keyword is explained elsewhere):

```balsa
sink WriteMessage (StringAppend ("Hello", ToString (vs_type, v)))
```

The functions “WriteMessage”, “StringAppend” and “ToString” are all defined in `balsa.types.builtin` and can be also be explicitly called by the user. Other String functions, which `balsa-c` does not rely on, are defined in `balsa.sim.string`.
7.2. Summary of Library Functions.

A number of libraries are supplied in the standard Balsa library set to help with test harness construction. These (listed by their import path declaration) include:

[balsa.types.builtin]: Functions and type necessary for balsa-c functionality.
[balsa.types.type]: Type comprehension functions.
[balsa.sim.string]: Other String handling functions.
[balsa.sim.fileio]: File I/O.
[balsa.sim.memory]: Functions and types to implement memory models.
[balsa.sim.portio]: Port file/console I/O used by balsa-make-test.
[balsa.sim.sim]: Simulator specific operations such as time and command line argument access.

Guidance for using these libraries can be found in the comments in the appropriate .balsa files in the Balsa source distribution share/balsa/types and share/balsa/sim directories. A summary of some of those library functions that are most useful to users are given below.

**types.builtin**

-- create a string object from a string
function String (parameter string : String) is builtin : String

-- append str2 to str1 returning a string object
function StringAppend (str1, str2 : String) is builtin : String

-- Convert a value of (nearly) any type to a default formatted string
-- used by the compiler to implement runtime printing
function ToString (parameter X : type; value : X) is builtin : String

-- write a runtime printing message string, returning 1
function WriteMessage (str : String) is builtin : bit

**sim.string**

-- StringLength : returns the length of the given string
-- (0 for an empty or uninitialised string)
function StringLength (string : String) is builtin : cardinal

-- SubString : returns a sub-string of the given string between
-- character indices `index' and `index + length - 1'
-- If length = 0 or index >= StringLength (string) then
-- returns an empty string,
-- If `index + length' > StringLength (string) then returns a
-- sub-string of `string' between indices `index' and StringLenth (string) - 1
function SubString (string : String;
                   index : cardinal;
                   length : cardinal)
               is builtin : String

-- StringEqual : returns 1 if two strings or equal.
function StringEqual (str1, str2 : String) is builtin : bit

-- FromString : parse a value of the given type (in the default formatting)
-- from the given `source' string and return the remainder of the string in
-- `remainder'. Note that the most common way of calling this function will be
-- with the same string as source and remainder. To discard the remainder,
-- just pass a constant (or unused) string as remainder.
function FromString (parameter X : type;
7.2. Summary of Library Functions.

source : String;
remainder : String
) is builtin : X

-- RepeatString : make a string with `n' occurences of source string `str'
function RepeatString (str : String; n : cardinal) is builtin : String

-- FitStringToWidth : pad or clip a given string to create a string which is
-- exactly `width' characters long. `justification' chooses whether strings
-- shorter than `width' should be packed at the start (left) or end (right) of
-- the result string
type StringJustification is enumeration left, right end
function FitStringToWidth (str : String;
width : cardinal;
justification : StringJustification
) is builtin : String

-- NumberFromString : parse a number of the given radix (assuming there will
-- be no radix prefixes) from the given string. Radix is an element of [2,36]
function NumberFromString (
parameter X : type;
source : String;
radix : 6 bits
) is builtin : X

-- NumberToString: make a string representation of the given number in the
-- given radix. Insert underscores at the specified distance apart (except
-- where underscoreSpacing is 0)
function NumberToString (parameter X : type;
value : X; radix : 6 bits;
underscoreSpacing : 8 bits;
showLeadingZeroes : bit
) is builtin : String

-- TokenFromString : parse a whitespace delimited string token from the start
-- of `string' and return that token as the return value and the remains of the
-- string in `remainder'. Note that this is not the same as FromString
-- (String, ...) as that would require quotes around the string to be parsed.
function TokenFromString (string : String;
remainder : String
) is builtin : String

-- Chr : convert the given 8b value into a single character string
function Chr (value : byte) is builtin : String

-- Ord : returns the character value of the first character in the given
-- string. If the string is empty, returns 0
function Ord (char : String) is builtin : byte

sim.fileio

type File is builtin
type FileMode is enumeration
read, write,
writeUnbuffered, -- unbuffered file writing
writeLineBuffered -- flushes after each line
over 3 bits

-- FileOpen : open a file in the appropriate mode
function FileOpen (fileName : String; mode : FileMode) is builtin : File

-- FileReadLine : read upto an end of line and return a string without that trailing NL
function FileReadLine (file : File) is builtin : String

-- FileWriter : write a string to a file, returns the file object
function FileWriter (file : File; string : String) is builtin : File

-- FileEOF : returns 1 if file is at the end of a file
function FileEOF (file : File) is builtin : bit

-- FileClose : close the file stream
function FileClose (file : File) is builtin : File

sim.memory
type BalsaMemory is builtin

-- BalsaMemoryParams : parameters bundle, can add others
type BalsaMemoryParams is record
  addressWidth, dataWidth : cardinal
end

-- BalsaMemoryNew : make a new memory object, this is separate from the procedure BalsaMemory so we can, for example have a dump-memory routine
-- external to that procedure. You could can BalsaMemory with:
-- BalsaMemory (16, 32, <- BalsaMemoryNew (), ...)
function BalsaMemoryNew is builtin : BalsaMemory

-- BalsaMemory(Read,Write) : simple access functions
function BalsaMemoryRead (parameter params : BalsaMemoryParams; memory : BalsaMemory; address : params.addressWidth bits) is builtin : params.dataWidth bits
function BalsaMemoryWrite (parameter params : BalsaMemoryParams; memory : BalsaMemory; address : params.addressWidth bits; data : params.dataWidth bits) is builtin : BalsaMemory

-- BalsaMemory : a single read port memory component, reads a BalsaMemory object as it is initialised and then waits for an incoming address and rNw indication
procedure BalsaMemory (parameter params : BalsaMemoryParams; input memory : BalsaMemory; input address : params.addressWidth bits; input rNw : bit; input write : params.dataWidth bits; output read : params.dataWidth bits) is
  variable memory_v : BalsaMemory
begin
memory -> memory_v;
loop
  address, rNw -> then
  if rNw then -- read
    read <= BalsaMemoryRead (params, memory_v, address)
  else -- write
    write -> then
      sink BalsaMemoryWrite (params, memory_v, address, write)
7.2. Summary of Library Functions.

```vhd
procedure B1632 is BalsaMemory ((16, 32))

-- BalsaPrintSyncPortActivity : " "
procedure BalsaPrintSyncPortActivity (parameter portName : String; sync s) is
begin
loop
  sync s;
  print BalsaSimulationTime (), " : sync `", portName, 
end
end

-- BalsaWriteLogLine : write a log line for some channel activity
procedure BalsaWriteLogLine (parameter portName, activity : String; input message : String) is
begin
  message -> then
    print BalsaSimulationTime (), " : chan `", portName, ", " , activity, ", ", message
end
end

-- BalsaOutputPortToLog : print activity on the output port of some component in the default format
procedure BalsaOutputPortToLog (parameter X : type; parameter portName : String; input i : X) is
begin
loop
  i -> then
    BalsaWriteLogLine (portName, "reading", <-ToString (X, i))
end
end

-- BalsaOutputPortToLogWithFormat : print activity on the output port of some component in the specified format
procedure BalsaOutputPortToLogWithFormat (parameter X : type; parameter portName : String; parameter radix : 6 bits; parameter underscoreSpacing : 8 bits; parameter showLeadingZeroes : bit; input i : X) is
begin
loop
```
7.2. Summary of Library Functions.

i \rightarrow \text{then}
\text{BalsaWriteLogLine} (\text{portName}, "reading",
\text{<- NumberToString} (X, i, \text{radix, underscoreSpacing, showLeadingZeroes})
\text{end}
\text{end}

\text{-- BalsaOutputPortToFile : print activity on the output port of some component in the default format}
\text{procedure BalsaOutputPortToFile (}
\text{parameter X : type;}
\text{parameter portName : String;}
\text{input file : File;}
\text{input i : X}
\text{)} \text{is}
\text{variable line : String}
\text{begin}
\text{file \rightarrow then}
\text{loop}
i \rightarrow \text{then}
\text{line := ToString} (X, i);
\text{sink FileWrite} (\text{file, line});
\text{sink FileWrite} (\text{file, "\n");}
\text{BalsaWriteLogLine} (\text{portName}, "reading", \text{<- line})
\text{end}
\text{end}
\text{end}

\text{-- BalsaPrintInputPortFromValue : supply the given value to the port "o" each time an input happens on that port}
\text{procedure BalsaPrintInputPortFromValue (}
\text{parameter X : type;}
\text{parameter portName : String;}
\text{input value : X;}
\text{output o : X}
\text{)} \text{is}
\text{begin}
\text{value \rightarrow then}
\text{loop}
o \leftarrow \text{value};
\text{BalsaWriteLogLine} (\text{portName}, "writing", \text{<- ToString} (X, \text{value}))
\text{end}
\text{end}

\text{-- BalsaInputPortFromFile : source values for port o from the given file}
\text{procedure BalsaInputPortFromFile (}
\text{parameter X : type;}
\text{parameter portName : String;}
\text{input file : File;}
\text{output o : X}
\text{)} \text{is}
\text{variable line : String}
\text{variable value : X}
\text{begin}
\text{file \rightarrow then}
\text{loop while not FileEOF} (\text{file} \text{then}
\text{line := FileReadLine} (\text{file});
\text{value := FromString} (X, \text{line, line});
7.3. Writing your own builtin functions

To show the stages necessary to use a user-written builtin function, we will present a small example function, HelloWorld, written in a block hello.balsa with a C implementation in hello.c. The code for this example can be found in examples/simulation directory of the distribution. The example below is described stage by stage in order to highlight the Balsa tools used, but the process is greatly simplified by using balsa-mgr described in “Using balsa-mgr” on page 86.

The Balsa and C code

Every builtin function must have both a Balsa declaration and a C language definition. In writing your own builtin functions it is best to write the Balsa declaration first. For example:

```balsa
function HelloWorld is builtin : bit
```

declared a builtin function with no arguments and a single bit return value. Note that functions must have return values (to operate correctly in the Balsa type system) even when the implementation of the function may be considered to have a “void” return type. It is usual to use the return type bit and return the value 1 when the return value is not important. In Balsa, the sink keyword can then be used to call such a function and discard the return value. In some functions, one of the arguments could make a useful return value to allow function calls to be enclosed within each other. The Write function in [balsa.sim.fileio] is an example of such a function, it returns the File object passed to it to allow chains of Writes to be formed as a single expression.

Each function must have a C implementation of the form (continuing the HelloWorld example with a very simple body):

```c
static void HelloWorld (BuiltinFunction *function,
   BuiltinFunctionInstanceData *instance)
{
   fprintf (stderr, "Hello, Balsa user\n");
}
```

The two arguments, “function” and “instance”, pass to the builtin function information about the port structure, instance parameter values and per-call argument values of the Balsa function. “function” contains information common to all instances of the function and “instance” contains instance specific data. Note that as builtin functions can have parameters, and that port typing can be influenced by typing, port structure information should be read from the “instance” argument rather than the “function” argument.

Registering the function

To register a builtin function with the simulation system, a call to BalsaSim_RegisterBuiltinFunction is necessary. Each shared library which contains C implementations of builtin functions should declare a function with the name

```c
static void HelloWorld (BuiltinFunction *function,
   BuiltinFunctionInstanceData *instance)
```
The header file \texttt{balsasim/builtin.h} provides the definitions of the types used in the file and the prototype for BalsaSim_RegisterBuiltinFunction. This file can be found in the \texttt{src/libs/balsasim} directory of the Balsa distribution and \texttt{include/balsasim} of a Balsa installation. It also includes the files \texttt{balsasim/object.h}, \texttt{balsasim/parameter.h} and (through \texttt{parameter.h}) \texttt{balsasim/types.h}. These three files provide declarations for types and functions for manipulating BalsaObject objects, C descriptions of Balsa parameters and C descriptions of Balsa types respectively.

This example only registers one function using \texttt{BalsaSim_RegisterBuiltinFunction}: namely HelloWorld. The seven arguments passed to cause that registration are (in order):

- **name**: Balsa name of the function being registered ("HelloWorld").
- **parameterCount**: number of parameters taken by the Balsa function (in this example, 0).
- **arity**: argument count of the Balsa function (again, 0).
- **function**: pointer to the C function containing the top level of the implementation.
- **resultWidth**: number of bits in the result value of the function, or 0 if the width varies by instance (see Section, “Return values,” on page 90).
- **argumentWidths**: an array of “arity” unsigned ints, one per argument in order, which specify the widths in bits of their respective arguments. Each of these can be 0, as with resultWidth, to indicate that the widths are resolved on an instance-by-instance basis. This can be set to NULL (as in this example) when there are no arguments to the function.
- **objectCount**: number of BalsaObject objects created by a call to this function. (see Section, “Object Reference Counting,” on page 91).

With the C implementation in file hello.c and the Balsa declaration in file hello.balsa. The C implementation can be compiled with:

```bash
balsa-make-builtin-lib hello hello.c
```
This should create \texttt{hello.la}, \texttt{hello.o}, \texttt{hello.a} and either \texttt{hello.so}... or \texttt{hello.dylib}... files depending on your machine architecture. The Balsa declaration file can be compiled with:

```bash
balsa-c hello
```
Note that the C and Balsa descriptions are not checked against each other when being compiled. For this reason it is important that the parameters passed to BalsaSim_RegisterBuiltinFunction are correct to ensure correct operation of the builtin functions in simulation.

With both the shared library and the Breeze file for the HelloWorld function, that function is ready to be used.
7.3. Writing your own builtin functions

Invoking HelloWorld

A short Balsa description such as:

```balsa
import [hello]
procedure try is
    sink HelloWorld ()
end
```

can be used to test HelloWorld. The description can be compiled, a Balsa top-level test harness generated, and the resulting test harness run. If the test description listed above is found in the file `SimDemo.balsa`, the following commands generate a default test harness.

```bash
balsa-c SimDemo
balsa-make-test -d SimDemo try
```

The last command generates a Balsa test harness, `test-SimDemo_try.balsa` with a top level procedure name of `balsa`. Although not strictly necessary for this example, it is a good habit to get into to always generate such a test harness. The next two commands compile the balsa test harness file and then run the simulation.

```bash
balsa-c test-SimDemo_try
breeze-sim test-SimDemo_try
```

Breeze-sim will pick up the shared library for the block [hello] by noting that the file `hello.la` was in the same directory as the Breeze file `hello.breeze`. Files with the extension .la are GNU libtool library information files. They contain the path of the shared library which bears the same name as the .la file.

HelloWorld in Verilog

It is possible to use a Verilog simulator as shown below: A

```bash
BALSATECH=example
export BALSATECH
# Use "setenv BALSATECH example" in csh/tcsh
balsa-netlist -s -d -f -i helper test-SimDemo_try
balsa-make-impl-test -o Vtest test-SimDemo_try balsa
balsa-sim-impl -B test-SimDemo_try Vtest
```

The BALSATECH environment variable specifies a Verilog target implementation. Particular implementation styles, as well as the Verilog simulator to be used, can be specified.

If the balsa test harness file has not been generated, the command `balsa-c test-SimDemo_try` must be run first.

```bash
balsa-netlist produces a Verilog netlist for the test harness: test-SimDemo_try.v.
balsa-make-impl-test produces a top-level Verilog file Vtest.v.
balsa-sim-impl runs the Verilog simulation.
```

Using balsa-mgr

Balsa-mgr can be used to perform all the steps of the HelloWorld example and considerably simplifies the process. In the description that follows, it is assumed that the files `hello.balsa` (containing the builtin balsa declaration), `hello.c` (containing the builtin C language definition) and `SimDemo.balsa` (containing the Balsa test example) already exist.

1. Add the .balsa files to the project as shown in Figure 7.1.
2. In the file pane of balsa-mgr, right-click on the `hello.balsa` filename and select the Add Builtin Library option. In the resulting popup dialogue shown in Figure 7.2, add `hello.c` to the list of source files using the new button. The library should then be visible in the file pane as shown in Figure 7.3.
3. In the file pane of balsa-mgr, right-click on the `try` procedure in `SimDemo.balsa` and select the the “Add Test Fixture” option. Accept the defaults in the resulting dialogue box.
4. Click on the Makefile tab to switch to the Makefile pane and clicking on the run button for sim-test1 in the Tests subpane will build the library and run the simulation as shown in Figure 7.4.

Verilog simulation can be achieved within the framework of basla-mgr. To do this, an implementation has to be attached to the test harness (rather than to the procedure itself).

5. In the file pane of basla-mgr, right-click on the test1 test fixture name attached to the try procedure and select “Add Implementation”. The Verilog implementation is added to the test1 test harness.

6. Click on the Makefile tab in the left-hand pane in the basla-mgr window. A new test action has been added to test1 in the Tests subpane.

7. Click on the Run button for sim-test1-impl: the test harness will be run as a Verilog simulation.

### 7.4 Builtin functions with arguments

Builtin functions, like other Balsa functions, are passed per-call arguments. These arguments can be of builtin types or normal Balsa bitwise data values. In both cases, values are passed into C as multi-precision integer values packed into FormatData structures. The FormatData type contains two elements:
7.4. Builtin functions with arguments

**wordCount**: the length of the value in multiples of the size of the type `unsigned int` in C

**words**: an array of unsigned ints containing the value, with the least significant word of the value in words[0].

Signed bitwise values are passed as though they were unsigned values with the same bitwise representation as the original signed value and are **not** sign extended to the end of a word or to the end of the bitwise length of the value. Result values from builtin functions are passed back to Balsa from C in a FormatData structure also. A simple function to add 15 to a 16b number looks like this in C:
7.4. Builtin functions with arguments

```c
static void Add15 (BuiltinFunction *function,
    BuiltinFunctionInstanceData *instance)
{
    FormatData *i = instance->arguments[0]->words[0];
    instance->result->words[0] = i + 15;
}
```

for a function with a Balsa description of:

```plaintext
function Add15 (i : 16 bits) is builtin : 16 bits
```

and is equivalent to the “pure” Balsa function:

```plaintext
function Add15 (i : 16 bits) = (i + 15 as 16 bits) : 16 bits
```

As can be seen in this example, the result and argument FormatData structures can be accessed as elements “result” and “arguments” of the BuiltinFunctionInstanceData passed to the C function. The arguments element is an array of length `function->arity` (which will be the same value as passed to the BalsaSim_RegisterBuiltinFunction function as the “arity” argument), with the first argument at index 0. The FormatData structures for arguments and results values are pre-allocated by the simulation system and so should only ever be read or modified, never replaced by a different FormatData. Note that the above example passed its return value back and processed its arguments by directly accessing the first word of the `instance->result` and `instance->arguments[0]` FormatData structures. This is a perfectly valid way of approaching FormatData handling. The definition of the type FormatData and a library of functions to act on that type can be found in the header file `src/libs/format/data.h` in the Balsa distribution and `include/format/data.h` in a Balsa installation.

**Builtin typed arguments**

Builtin types can have any simulator-internal representation that the author of builtin functions which process that type desires. For example, the File type defined in the header file `balsasim/bfile.h` and used by block `[balsa.sim.fileio]` is defined as:

```c
typedef struct
{
    FILE *file;
    char *filename;
    BalsaFileMode mode;
} File;
```
BalsaFile;

Values of builtin types are passed around in Balsa, and to and from C, as pointers to BalsaObject structures. Using BalsaObject to encapsulate a pointer to a real value allows all builtin typed values to be handled consistently with respect to memory allocation management.

Builtin typed values are packed into argument and result FormatData structures as a 64 bit pointer value to the BalsaObject structure which encapsulates the pointer to that value’s real data. The BalsaObject pointer can be extracted from a FormatData structure using the function FormatDataGetBalsaObject. The pointer to that value’s real data can then be extracted as the “data” element of that BalsaObject. For example, the FileEOF function in [balsa.sim.fileio] extracts a File object from index 0 of its first argument and places the pointer to the BalsaFile structure into “file” (“BALSA_FILE” is just a C preprocessor macro for a cast to type BalsaFile; this macro is defined in balsasim/bfile.h

```c
static void Fileio_FileEOF (BuiltinFunction *function,
   BuiltinFunctionInstanceData *instance)
{
   BalsaObject *fileObject =
   FormatDataGetBalsaObject (instance->arguments[0], 0);
   BalsaFile *file = BALSA_FILE (fileObject->data);
   ...
}
```

**Return values**  
Builtin typed values can similarly be returned by packing the pointer to a BalsaObject into a FormatData using FormatDataSetBalsaObject. FileOpen in [balsa.sim.fileio] does this like so:

```c
FormatDataSetBalsaObject (instance->result, instance->objects[0], 0);
```

Notice that the object packed into `instance->result` is also an element of the instance structure. This is necessary because Balsa must track the location of builtin typed values at all times in order for the reference counting system used to deallocate unused objects to work correctly. To make the reference counting work effectively, the user must only use the BalsaObject structures contained in the `instance->objects` array (whose size is selected by the objectCount argument to BalsaSim_RegisterBuiltinFunction) and never any BalsaObject which is manually allocated. The simulation system monitors the reference counts of each object in `instance->objects` for each call of each builtin function, and will handle the deallocation/reassignment of objects without the user having to worry about explicit reference counting. As the BalsaObject structure only contains a pointer to the “real” data associated with a builtin typed value, calls to FormatDataSetBalsaObject are usually preceeded with a function call to pack that real data pointer into the BalsaObject and to nominate a function to be used to deallocate that data if the object ceases to be useful. In FileOpen, this call looks like:

```c
SetBalsaObject (instance->objects[0], balsaFile,
   (BalsaDestructor) DeleteBalsaFile);
```

On deallocation of the object in `instance->objects[0]`, DeleteBalsaFile will be called on the pointer ‘balsaFile’ (which will become stored in `instance->objects[0]`), in order to deallocate it. If NULL is passed to SetBalsaObject as destructor for this object, deallocation of that object will result in a call to free(3) on the real data pointer.

**Functions with parameterised arguments**  
Parameters passed to a builtin function can be used to parameterise the types of arguments passed to calls of those functions. For example, the ToString function, used to render string representations of Balsa values of any type, has as a parameter the expected type of the argument to the function. ToString’s declaration in Balsa (which can be found in [balsa.type.builtin]) is:

```c
function ToString (parameter X : type; value : X) is builtin : String
```

As previously explained, the `instance->parameters` array can be used to comprehend the parameter passed to a builtin function in C. For builtin functions with arguments which are not fixed in the Balsa declaration, this array must be used to determine the correct argument and result widths. To allow this to happen, simulation systems using the Balsa builtin function system must make an
7.5. Object Reference Counting

Allocation of BalsaObjects in simulation is done by counting the number of times an object becomes assigned to either a Balsa variable or an element of the instance->objects array in a function. The reference counting scheme used to implement this assignment counting is very conservative and only deallocates an object when that object’s place in a variable or instance->objects array must be overwritten. For a variable, this occurs on each assignment and for a function’s objects array this occurs each time the function is called.

Variable assignment

Objects are always held in a special variable handshake component, BuiltinVariable, inside a Breeze description. This special component is similar to a normal Variable handshake component but includes simulation mechanisms to handle the reference counting of stored and incoming data. Each time an assignment occurs on a BuiltinVariable, two events occur: Firstly, the object already residing in the variable (if any) is to be discarded and so has its reference count decremented and the object (and its “->data” payload) is deallocated if the reference count reaches 0. Secondly, the pointer to the new object being assigned is loaded into the variable’s latches and its reference count is incremented to indicate that it has been successfully stored.

Function objects array

The objects array in each function’s “instance” data is used to store objects which have not yet been assigned to variables or which will never end up in a variable (such as intermediate Strings in a chained StringAppend operation, for example). Each object is initialised with a reference count of 1 indicating that it is stored in exactly one place. As objects are passed out of the function as return values, those objects may have their reference counts increased to indicate that they have been stored elsewhere. On the next call to the function, each of the objects previously allocated must be
7.6. Predefined types

As previously stated, the “data” element of a BalsaObject can be used as a pointer to any value which the user wishes to use as the basis of a builtin-typed value in Balsa simulation. The builtin libraries which are provided with Balsa for string and file manipulation make use of the C BalsaString and BalsaFile types to represent those values. It is very likely that user-defined builtin functions will need to work with those predefined functions, and so it is important to understand the mode of operation of those types.

**BalsaString**

Strings are represented in Balsa simulation as char arrays encapsulated in dynamically allocated instances of the BalsaString struct defined in `balsasim/bstring.h`. Each instance of a String in Balsa is represented by a unique BalsaString in C. It is, however, possible for different String values to share their underlying char arrays in order to make sub-string operations (which are common when parsing files) more efficient. The BalsaString structure contains 4 elements:

- **char *allocatedString:** a pointer to the first element of the allocated char array which represents this string. Note that this pointer refers to the allocated array for the string, which may be shared with other BalsaStrings, and may not point to the first character of this particular string. BalsaStrings sharing a common char array *must* have the same value of allocatedString.

- **char *string:** a pointer to the character in allocatedString which corresponds to the first character of this BalsaString’s “real” string. For example, when tokenising the line “Hello, world” from a file, a BalsaString may be created which is a sub-string of the whole line and so has its allocatedString element pointing to the “H” in “Hello” and its string element pointing to the “w” in “world” indicating that that BalsaString represents part of the string starting with the “w”.

- **unsigned length:** the number of significant characters (between string[0] and string[length-1]) which comprise the string being represented. BalsaString strings are not required to be NUL terminated (although for safety it is good practice to make allocatedString one char longer and place a NUL in the final character) and so when passing the ->string element of a BalsaString to a C function, it is advisable to make a temporary copy of the string. int *refCount: a (pointer to the) count of the number of BalsaStrings which share the same allocatedString as this one. When deallocating a BalsaString, care must be taken to avoid mistakenly deallocating the allocatedString when other BalsaStrings may depend on it. The refCount is a single malloced int, initially set to 1 indicating a single BalsaString owns this allocatedString, which can be incremented for each sub-string creation and decremented for each sub-string deallocation. The functions BalsaStringRef and BalsaStringUnref are used to maintain this count and handle the deallocation of BalsaStrings.

Besides BalsaStringRef and BalsaStringUnref, the balsasim/bstring.h package only contains two other functions, both used to create new BalsaString objects:

- **NewBalsaString:** creates a BalsaString from an existing char array by copying “length” character from the source string into a newly allocated allocatedString. NewBalsaString can be called with a NULL strings, which causes it to allocate only the BalsaString object rather than the underlying char array. This can be useful when the required array is to be constructed by hand rather than copied. Note that after calling NewBalsaString this way, both allocatedString and string elements of the resulting BalsaString must be correctly initialised by the user. Passing -1 as the `length` argument to
NewBalsaString results in the creation of a BalsaString containing all of the source C string up to the first NUL character in that string.

NewBalsaSubString: creates a BalsaString which shares its allocatedString with the given BalsaString between start[0] and start[length-1]. The mechanism for sharing sub-strings is described above.

Better understanding of the common uses of the BalsaString type and its associated functions can be gained by reading the builtin function code in the [balsa.types.builtin] and [balsa.sim.string] libraries.

**Balsa**

File access is performed in Balsa using the File type. The is defined in the library [balsa.sim.fileio] using the underlying C type BalsaFile defined in balsasim/bfile.h. BalsaFile is a simple wrapper for the C standard library type FILE * and has 3 elements: FILE *file: the open file handle or NULL indicating that the file is not open. char *filename: a copy of the filename used to open the file. This is used for error reporting. BalsaFileMode mode: an enumeration indicating how the file was opened. Currently four options exist for this element: read, write, writeUnbuffered and writeLineBuffered. The options read and write correspond to the fopen file modes “r” and “w”. The buffered write options correspond to mode “w” with a subsequent call to setvbuf to select the appropriate file buffering mode.

The BalsaFileMode type is defined in Balsa (as type FileMode) and C as it is used as the argument to the FileOpen function. The C header file balsasim/bfile.h defines only two functions of interest to users wanting to create their own file handling functions: BalsaFileReadable and BalsaFileWritable. These functions can be used to check if a BalsaFile corresponds to an open file and if that file is readable/writable through the ->file element of that BalsaFile. Examples of the use of these functions can be found in the C implementation of the [balsa.sim.fileio] library.

### 7.7. Example Custom Test Harnesses

**Data Formatting**

Actually this example is now obsolete as the user can set the format of displayed data when configuring the test harness in balsa-mgr. However, since the example illustrates use of some of the builtin functions, the description is still included in the manual.

By default, numbers are written in decimal. The example in Simulation/Format illustrates the use of the builtin functions. The example is actually the shifter example “A Balsa shifter,” on page 67. The test procedure test_ror.balsa shifts a bit pattern of 3 consecutive ‘1’s around a 32 bit word The default output produced is:

```
230: chan 'o' reading 14
727: chan 'o' reading 7
1245: chan 'o' reading 2147483651
1749: chan 'o' reading 3221225473
2267: chan 'o' reading 3758096384
2785: chan 'o' reading 1879048192
3326: chan 'o' reading 939524096
3839: chan 'o' reading 469762048
4357: chan 'o' reading 234881024
4875: chan 'o' reading 11744064
5416: chan 'o' reading 58720256
5943: chan 'o' reading 29360128
6484: chan 'o' reading 14680064
7025: chan 'o' reading 7340032
7589: chan 'o' reading 3670016
8111: chan 'o' reading 1835008
8629: chan 'o' reading 917504
9147: chan 'o' reading 458752
9688: chan 'o' reading 229376
10215: chan 'o' reading 114688
```
It is not easy to spot that this is the correct behaviour. The procedure in the test harness produced by balsa-mgr that writes the output is:

```plaintext
procedure balsa
is
  channel o : 32 bits
begin
  test_ror32 (o) ||
  BalsaOutputPortToLog (32 bits, "o", o)
end
```

The output can be produced in binary format by rewriting the builtin procedure responsible for displaying the output: `BalsaOutputPortToLog`. The procedure has as to be renamed to prevent a name clash.

```plaintext
procedure BalsaOutputPortToLogX (parameter X : type;
  parameter portName : String; input i : X) is
begin
  loop
    i -> then
    -- original line in BalsaOutputPortToLog
    -- BalsaWriteLogLine (portName, "reading", <- ToString (X, i))
    BalsaWriteLogLine (portName, "reading",
      <- NumberToString (X, i, 2, 4, 1))
  end
end -- procedure BalsaOutputPortToLogX
```

This produces the output in binary with leading zeros with each 4 bit field separated by an underscore.
7.7. Example Custom Test Harnesses

The rotation of the 3 '1's can now be clearly seen. If it is desired to produce the output in hexadecimal with no leading zeros and no underscore separator, call BalsaWriteLogLine as:

```
BalsaWriteLogLine (portName, "reading", <- NumberToString (X, i, 16, 0, 0))
```

Further examples of conversions to and from strings can be found in “Memory models” on page 96.

**FileIO**

The examples in `simulation/FileIO` illustrate some basic use of the File I/O routines. These procedures read the contents of a file whose name is a parameter of the procedure. Note that it is not possible to test for the readability or existence of a file to open: if access is not allowed, the FileOpen procedure will fail internally producing an error message.

```
import [balsa.types.basic]
import [balsa.sim.fileio]

procedure rd_file1 (parameter fname : String) is
  variable file : File
begin
  file := FileOpen(fname, read);
  print "Opened file: ", fname;
  loop while not FileEOF(file) then
    print "content is: ", FileReadLine(file)
  end
end

The rd_file1 procedure is instantiated with the name of the file to be opened thus:

```
procedure rf1 is rd_file1("data")
```

where “data” is the name of the file to be opened. A disadvantage of this approach is that what is being generated is an instance of a parameterised procedure. Everytime the name of the file is changed, a new instance has to be compiled. Another approach is shown below:

```
procedure rd_file2 (input fname : String) is
  variable file: File
begin
  fname -> then
    file := FileOpen(fname, read);
    print "Opened file: ", fname;
    loop while not FileEOF(file) then
      print "content is: ", FileReadLine(file)
    end
  end
end
```

The file name is passed to rd_file2 from a top-level procedure using a variable port.
procedure rf2 is  
begin  
rd_file2(<- "data")  
end

Memory models

The example in Simulation/Memory/mem1.basa illustrates interfacing to Balsa’s memory model:

Simple memory manipulation

import [balsa.sim.memory]  
import [balsa.sim.string]

procedure ex1 is  
channel addr : 4 bits  
channel read, write : 8 bits  
channel rNw : bit  
variable addrCount : 4 bits  
begin  
-- Read the BalsaMemory description in /share/balsa/sim/memory.balsa  
-- for details. BalsaMemory is the name of a type that represents  
-- simulation memories and also a procedure encapsulating a memory  
-- model built from BalsaMemoryRead and BalsaMemoryWrite built-in function  
-- calls. You can either use this module or make your own use of the  
-- built-in functions directly.  
BalsaMemory (  
{4, -- address width  
8}, -- data width  
<- BalsaMemoryNew (), -- direct expression to port connection  
addr, rNw, write, read) ||  
begin  
addrCount := 0;  
print "Write inverse address as data" ;  
loop  
addr <- addrCount || rNw <- 0 ||  
write <- ( not addrCount as 8 bits);  
addrCount := (addrCount + 1 as 4 bits)  
while addrCount /= 0  
end;  

-- Now dump the memory,  
-- there really ought to be built-in functions for this  
addrCount := 0;  
loop  
addr <- addrCount || rNw <- 1 ||  
read -> then  
print "Address: ", addrCount, " Data: ",  
NumberToString (8 bits, read, 16, 4, 1)  
end;  
addrCount := (addrCount + 1 as 4 bits)  
while addrCount /= 0  
end  
end

More Complex memory composition

This example uses separate procedure to load the memory and dump its contents. These procedures are composed with a simple process that writes and read a few arbitrary locations. Notice the use of the string to number conversions (and vice versa). If the numeric values in the data file are in the default format (i.e. decimal values carry no prefix, hexadecimal numbers are prefixed with 0x etc.), the appropriate conversion routine to use is FromString. However, if the numbers are in a particular format (say hexadecimal) and are not prefixed, then NumberFromString must be employed with the appropriate radix passed in the function call.
import [balsa.sim.memory]
import [balsa.sim.string]
import [balsa.sim.fileio]

constant addr_width = 5
constant data_width = 8
constant MemSize = 2 ^ addr_width

type AddrWidth is addr_width bits
type DataWidth is data_width bits

procedure load_mem (
    input fname : String ;
    output addr_bus : AddrWidth ;
    output data_bus : DataWidth ;
    output rNw : bit ) is

    variable file : File
    variable num : DataWidth
    variable addr : AddrWidth

    begin
        fname -> then
            print "loading memory from: ", fname;
            file := FileOpen(fname, read);
            print "Opened file: ";
            loop while FileEOF(file) /= 1 then
                -- if data has no radix prefix use this conversion
                -- see the effect with supplied data file which has prefix.
                num := NumberFromString(DataWidth, FileReadLine(file),16);
                -- if data has radix prefix use this form
                -- this is probably what is required for supplied data
                -- num := FromString(DataWidth, FileReadLine(file), "");
                print num ;
                addr_bus <- addr || data_bus <- num || rNw <- 0;
                addr:= (addr + 1 as AddrWidth)
            end loop
        end
    end

procedure proc (  
    output addr_bus : AddrWidth;  
    output write_bus : DataWidth;  
    input read_bus : DataWidth ;  
    output rNw : bit ) is

    variable x : DataWidth

    begin
        -- poke the memory to show we can
        addr_bus <- 0 || write_bus <- 0xff || rNw <- 0;
        addr_bus <- 1 || write_bus <- 0xfe || rNw <- 0;
        -- read the memory to show we can
        addr_bus <- 1 || read_bus -> x || rNw <- 1;
        print "Value from address 1 is: ", x
    end

procedure dump_mem (  
    output addr_bus : AddrWidth ;  
    input data_bus : DataWidth ;  
    output rNw : bit ) is
A Processor Test Harness

This example in Simulation/Processor ties together many of the previous examples of using the built-in Balsa functions to create custom Balsa test harnesses. The ssem processor described previously (see “A Simple Processor – The Manchester SSEM (The Baby)” on page 72) is connected to a memory model which is loaded with the code corresponding to a program for computing the gcd of two numbers. The source code can be found in gcd.s. The two numbers are specified in locations 0x11 and 0x12 with the result, on termination, found in location 0x11. A description of processor can be found in ssem.pdf. An assembler sasm is provided for users who wish to write other programs.
8 Implementations

8.1. Introduction

Balsa provides means of describing and modelling asynchronous systems together with a means of functionally simulating these systems. However, Balsa is primarily a synthesis system and in this chapter the various implementation routes and options are described. In order to produce real silicon or a real gate-array implementations, access to the design-kits of the silicon or gate-array vendor is required – Balsa merely produces a netlist in format appropriate to a CAD system that supports the technology.

When creating an implementation, users may choose a particular technology, different “styles” within a technology and for each style a variety of options may be available.

Technologies

Currently Balsa supports the following technologies. Each technology has its own cell libraries, gate fan-in restrictions, instance naming and pin mapping conventions. Different technologies may also use different netlist formats. The technologies must be downloaded and installed as separate packages. Only the

**balsa-tech-ams:** This technology supports the AMS 350nm design kit and produces a Verilog netlist.

**balsa-tech-amulet:** This technology contains a set of custom cells designed within the Balsa group based on the SGS-ST 180nm library and produces a Verilog netlist.

**balsa-tech-sths018:** This technology contains only standard cells from the SGS-ST 180nm library.

**balsa-tech-example:** This technology produces a Verilog description based on example cells and is intended as template for users who wish to create their own back-ends.

**balsa-tech-xilinx:** This technology produces a EDIF netlist suitable for Xilinx gate-arrays

Styles

Currently the Balsa release supports the following back-end protocols for use with each technology:

**four_b_rb:** a bundled-data scheme using a four-phase-broad/reduced-broad signalling protocol.

**dual_b:** a delay-insensitive dual-rail encoding.

**one_of_2_4:** a delay-insensitive one-of-four encoding.
The bundled-data back end should be faster and smaller, but needs more careful post-layout timing validation. The two delay-insensitive schemes are larger and slower but should be more robust to layout variations.

## Options

Each option is set/unset or takes a value as shown in Table 8.1 on page 100.

<table>
<thead>
<tr>
<th>Option</th>
<th>Values</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>suggest-buffer</td>
<td>set/unset</td>
<td>Handshake circuit descriptions allow for nodes in the circuit to be identified as being points at which buffers may be inserted because the node may be heavily loaded. Setting this option will cause the buffers to be instantiated.</td>
</tr>
<tr>
<td>cad</td>
<td>cadence</td>
<td>This option only makes sense for Xilinx technology. Makes balsa-netlist produce verilog netlists compatible with Cadence implementations of Xilinx libraries.</td>
</tr>
<tr>
<td></td>
<td>ise</td>
<td>This option only makes sense for Xilinx technology. Makes balsa-netlist produce verilog netlists compatible with Xilinx ISE implementations of Xilinx libraries.</td>
</tr>
<tr>
<td>logic</td>
<td>DIMS</td>
<td>Implements “helper” cells – those cells composed from the basic cell library – in a DIMS style</td>
</tr>
<tr>
<td></td>
<td>NCL</td>
<td>Implements “helper” cells – those cells composed from the basic cell library – using NCL style gates. In many circumstances smaller implementations result.</td>
</tr>
<tr>
<td></td>
<td>Balanced</td>
<td>Creates balanced circuits where the notional path delays through the DIMS circuits are matched in an attempt to defeat Differential Power Analysis attacks in security applications such as smartcards</td>
</tr>
<tr>
<td>variable</td>
<td>SR</td>
<td>Variables stored in “standard” SR latches</td>
</tr>
<tr>
<td></td>
<td>Spacer</td>
<td>Each variable latch is reset to a NULL state before a write operation in an attempt to defeat Differential Power Analysis attacks in security applications such as smartcards</td>
</tr>
<tr>
<td></td>
<td>NCL</td>
<td>Variables are stored in pipeline style latches. More efficient for 1-of-4 codes.</td>
</tr>
<tr>
<td>n-of-m mapping</td>
<td>set/unset</td>
<td>Enables general n-of-m mapping strategy for dual rail (dual_b) styles. General users should accept the default option – the option is included for historical reasons.</td>
</tr>
<tr>
<td>sim</td>
<td>icarus</td>
<td>These option are only available for the example technology. They are various Verilog simulators known to work with the Balsa system. Note that balsa-sim-verilog must be configured to locate a particular simulator (see the installation notes). icarus and cver are publicly available simulators.</td>
</tr>
<tr>
<td></td>
<td>vxl</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ncv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vcs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>modelsim</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cver</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Style options
8.2. Creating an implementation

Many of the options offered are for use within specialist research projects; others depend on the exact tool flow used when targeting particular silicon technologies and design kits.

<table>
<thead>
<tr>
<th></th>
<th>conv</th>
<th>Use conventional implementation of the FalseVariable component with full enclosure of activity on the read channels within the handshake upon the write channel.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ovlp</td>
<td>Introduce concurrency within the passive enclosure of FalseVariable components by overlapping the return-to-zero phases of the “write” port and the “signal” port which triggers activity on the readports. This is the default option.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>conv</th>
<th>Use conventional implementation of the Concur component with full enclosure of activity on the output channels within the handshake of the activate channels.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ovlp</td>
<td>Introduce concurrency between the return-to-zero phases of the activation and output channels of Concur components. This is the default option.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>conv</th>
<th>Use conventional implementation of the Sequence component, a full handshake is completed on each output channel before initiating activity upon the next channel in the sequence.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ovlp</td>
<td>Introduce concurrency, where safe, between the return-to-zero phase of the an output channel and the processing phase of the next channel in the sequence. The balsa-c compiler determines within which components this optimisation can be safely implemented. This is the default option.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>conv</th>
<th>Use the conventional implementation of PassivatorPush Components. To maintain the data-validity protocols across the broad/reduced-broad interface in single-rail implementations, this option implements enclosure of output channel activity within input channel activity in Fetch components.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>broad</td>
<td>Use a “broad” implementation of single-rail PassivatorPush component. The broad implementation latches the data within PassivatorPush components, solving the data-validity problems caused by broad/reduced-broad protocol interfaces in Fetch components. This is the default option.</td>
</tr>
</tbody>
</table>

Table 8.1: Style options

Many of the options offered are for use within specialist research projects; others depend on the exact tool flow used when targeting particular silicon technologies and design kits.

8.2. Creating an implementation

In balsa-mgr, select the top-level procedure. Right-click and choose “Add Implementations” (as shown in Figure 8.1) causing a dialogue box to be spawned. The user can change the name of the implementation and the default balsa-netlist options (see Section, “Balsa Reference,” on page 127). Clicking on the technology tab reveals the technology and style options. Shown in Figure 8.2

Choose the technology desired, the implementation style and the style options. An icon for the implementation should be displayed in the File pane under the chosen procedure. Changing to the Makefile pane should reveal the new rule listed under the implementations subpane. Clicking on the Make button will generate the appropriate netlist for the technology.
8.2. Creating an implementation

All that remains is for the netlist to be imported into the CAD framework for the chosen technology! Future versions of this manual will give advice on how to do this.
8.2. Creating an implementation

Figure 8.3: Making the implementation.
Adding Technologies to Balsa

9.1. The Balsa backend

This chapter documents how multiple technologies and implementation styles (described in “Implementations,” on page 99) are handled. It describes how to add technologies and implementation styles to the Balsa system.

A Balsa description of a circuit is initially compiled to an intermediated breeze format containing references to generic, parameterised handshake components. To create a concrete implementation, `balsa-netlist` creates instances of expanded handshake components, in a .net format netlist, from the parameterised breeze specifications by applying the parameters to a description of the component.

The description used to generate the handshake component is composed from abstract gate operators and customised cells and is dependant on the implementation style and, in a small number of cases, the technology. The implementations are described in a special language `abs` (see “The abs language” on page 112). The .net file is then mapped, according to specifications defined by the technology, into the target netlist format which involves mapping the .net instance names into the names of the technology cells and decomposing large gates that are incompatible with the technology into smaller gates.

A knowledge (and love!) of the lisp-like language `scheme` is helpful for understanding how to construct a new backend.

The Balsa backend system allows for implementations in different technologies and different asynchronous styles. The technologies correspond to different cell libraries (either custom built or vendor-supplied standard cell libraries) for silicon foundaries or libraries for programmable gate-arrays such as Xilinx. Although each technology has its own cell libraries, gate fan-in restrictions, instance naming and pin mapping conventions and netlist formats, most handshake component descriptions are common across all technologies.

Balsa supports several different asynchronous implementation styles; the present release supports:

- a bundled data scheme using a four-phase-broad/reduced-broad signalling protocol
- a dual-rail delay insensitive scheme
- a one-of-four encodings delay-insensitive scheme.
Each implementation style may have several style options such as variations in the types of latches or the style of logic used. In contrast to technologies, styles need different component descriptions for each type of implementation.

There is much similarity between the requirements of different netlist formats which is reflected in the directory structure. Information specific to a particular technology is held in a directory corresponding to the technology’s name. Other information which is common to all technologies is held in the directory common. As control components are generally determined by the signalling protocol rather than by the data encoding, the descriptions for the implementation directories are split up into various control and datapath implementations to reduce the number of directories. A extract of the directory structure (rooted at balsa/share/tech) is shown in Figure 9.1.
9.2. The technology configuration file

The *common* directory contains the following files and directories.

- **common**: this is an empty configuration file for the technology – not used.
- **components**: this directory contains each component in a separate file. For each implementation style there is a link to the directory and file of the relevant description. Also in the directory are several definition files: *definitions.abs*, *dual-rail-definitions.abs*, *one-of-four-definitions.abs*; these files contain functions (in the abs language) used by many of the component descriptions.
- **components.abs**: this file includes all the components in the component directory.
- **helper-cells.abs**: this file is a set of descriptions of all the current helper-cells in abs – it allows helper-cells to be generated in any technology by the program balsa-make-helpers.
- **helper-mappings**: this is a 3-way mapping file format similar to gate-mappings to map from a helper-cell-abs description to a cell name in balsa-cells.net and an entry in gate-mappings. e.g.

  \[
  ("c-element3" "c-element3" "c3")
  \]

  Here the first argument is the name of the cell in helper-cells.abs the second the name of the cell the abs HC component descriptions and the gate-mappings file, and the third the name of the cell in the balsa-cell.net file to be generated.
- **template-balsa-mgr.cfg**: a template for adding technologies, styles and style options to balsa-mgr.

In each technology directory, `<tech>`, the following files are found:

- `<technology>`: essentially a configuration file for defining various files and component names used by the technology. The file format is described in “The technology configuration file,” on page 107.
- `<technology>-cells.net`: a file in .net format (see “Netlist,” on page 124) containing lists of all the cells in the library, together with their pin orderings and directions. The name of this file can be changed in the technology configuration file.
- **balsa-cells.net**: a file in .net format containing all the “helper” cells required by the various Balsa descriptions, such as adders, s-elements etc which are not resident as cells in the target technology library. The name of this file can be changed in the technology configuration file.
- **balsa-mgr.cfg**: this file is necessary so that the technology and its options are known about by balsa-mgr. A template for constructing the file can be found in `common/template-balsa-mgr.cfg`
- **components.abs**: contains descriptions of handshake components which are specific to that technology; typically the last line of the file will import common descriptions from `common/components.abs` indirectly via `common/components.abs`.
- **drive-table**: not currently used.
- **gate-mappings**: used to map between the abstract gate names and pin orderings of the .net output and that required by the technology. This file also contains information about different cells to use when driving large loads. The information required to “drive-up” signals where necessary is contained in the drive-table file. – however, at present, this feature is not available in this Balsa release. For more details see “Netlists,” on page 117.

9.2. The technology configuration file

Each technology is controlled by a configuration file, named the same as the technology:

- **net-signature-for-netlist-format** determines the netlist format to use for the technology, either verilog, edif\(^1\). The second argument, if true, sets the format as the default for that

\(^1\) “compass” is also allowed for historical reasons, producing a netlist in that design system’s proprietry format.
9.2. The technology configuration file

The net-signature-for-netlist-format procedure is also used by balsa-netlist with the -n option when producing new-netlists to allow different netlists to be produced other than the default - if a default netlist signature was not set, then balsa-netlist would produce an error as no netlist could be produced. It is possible to have two different netlist-signatures but their use is controlled by style-options.

breeze-gates-net-files is a list of the cell description files used by the technology.

breeze-primitives-file is the path to the component.abs file of the technology, breeze-tech-dir is a global variable in the scheme code that defines the path to the technology directory.

breeze-gates-mapping-file is the path to the gate-mappings file.

breeze-gates-drive-table is the path to the drive table, containing information about the loading and drive strengths of each gate.

tech-gate-max-fan-in sets the maximum fan-in for the logic gates (AND, OR, etc) in the library.

tech-c-element-max-fan-in sets the maximum fan-in for c-elements in the library.

tech-map-cell-name sets the mapping function for the handshake component names, at present only net-simple-cell-name-mapping function is available, which is a simple cropping procedure taking a boolean argument stating whether uppercase or lowercase letters are preferred by the technology. The length at which names are cropped is controlled by the tech-cell-name-max-length variable. The id function is used when no mapping is required, the function preserves the original balsa names.

tech-netlist-test-includes is a list of HDL models of the cell library, used by balsa-mgr when simulating implementations with CAD simulators.

The last four declarations set the power and ground net and component names for the technology. balsa-netlist instantiates power and ground components for connections between gates and the rails. The name of the net used to connect to these components is determined by the tech-gnd-net-name and tech-vcc-net-name variables. If no power and ground component names are supplied when specifying a verilog netlist, these nets are instantiated as supply0 and supply1 types respectively. and no power or ground components are instantiated.

There are three variables not shown in Figure 9.2:

Figure 9.2: A typical configuration file.

```
(net-signature-for-netlist-format 'verilog #t)
(set! breeze-gates-net-files ('"example-cells" "balsa-cells")
(set! breeze-primitives-file (string-append breeze-tech-dir
"components.abs"))
(set! breeze-gates-mapping-file (string-append breeze-tech-dir "gate-
mappings"))
(set! breeze-gates-drive-file (string-append breeze-tech-dir "cadence-
drive-table"))

;;; max. no. of inputs for and/or/nand/nor gates and c-elements
(set! tech-gate-max-fan-in 3)
(set! tech-c-element-max-fan-in 3)
(set! tech-map-cell-name id) ;; No mapping
(set! tech-netlist-test-includes ('"example-cells.v")
(set! tech-gnd-net-name "!gnd")
(set! tech-vcc-net-name "!vcc")
(set! tech-gnd-component-name "LOGIC0")
(set! tech-vcc-component-name "LOGIC1")
```
9.3. Handshake component declarations

For convenience, the descriptions of common handshake components are separated into implementation independent declarations and technology specific implementation descriptions. Each HC declaration (found in common/components/) consists of four parts as shown in Figure 9.3:

```
(primitive-part "Bar"
  (parameters
    ("guardCount" (named-type "cardinal"))
  )
  (ports
    (port "guard" passive output (numeric-type #f 1))
    (sync-port "activate" passive)
    (arrayed-port "guardInput" active input (numeric-type #f 1) 0
     (param "guardCount"))
    (arrayed-sync-port "activateOut" active 0 (param "guardCount"))
  )
  (symbol
    (centre-string "[]")
  )
  (implementation
    (style "four_b_rb" (include tech "common" "data-single-broad/Bar"))
    (style "dual_b" (include tech "common" "data-dual/Bar"))
    (style "one_of_2_4" (include tech "common" "data-1of4/Bar"))
  )
)
```

**Figure 9.3:** Example of a component abs file

| parameters  | Variable expressions used to customise the component. |
| ports       | Declaration of the ports of the component. There are four kinds of port: |
|             | sync-port |
|             | arrayed sync-port |
|             | data-port |
|             | arrayed data-port |
| symbol      | The symbol of the component as it would appear in an HC graph. |
| implementation | The implementation descriptions of the component for each implementation style – usually a link to a description in the appropriate style directory although descriptions may also be inlined. The format of these descriptions, their operators and operands is discussed below. |
9.4. Handshake component implementation descriptions

Handshake component instances are generated from these descriptions according to the parameters in the intermediate breeze file. The descriptions are a recipe written in the abs language which has operators to create gates or arrays of gates, as well as operators to construct and destruct wire vectors used by the component. Each HC implementation description consists of four separate sections:

defines

Specifies an optional list of expressions, defined by the parameters of the component. Definitions are of the form:

\[(\text{id} \ \text{expression})\]

The complete grammar for expressions is given in “The ABS Grammar,” on page 120 and includes operators such as *, /, and, if etc. It also contains several built-in functions:

- pop-count ;; the number of bits set in a linear representation
- find-set-bit ;; the first set bit of a binary number
- find-clear-bit ;; the first clear bit of a binary number
- style-option ;; returns true if a particular style option is in the BALSATECH variable
- bit-length ;; the number of bit required to implement a binary number
- bit-set ;; the boolean value of a given bit of a binary number
- bit-xor ;; result of a boolean xor operation on two binary numbers
- .. ;; create an integer list between a pair of values
- print ;; print a list of expression to current-port
- note ;; print a list of expressions to error-port

The full range of scheme’s built-in functions are also available.

The expression language contains the facility to support user-defined lambdas (anonymous functions). The lambdas are similar in style to lambdas in the scheme language. Lambdas are declared like any other expression and take the form:

\[(\text{lambda} (\text{params}*))\]
\[(\text{let-expression}?)]
\[(\text{body-expression})\]

Where params is a list of identifiers. The let expression is a list of local definitions, taking the form:

\[(\text{let})\]
\[(\text{identifier expression})*]\n\[(\text{let-expression}?)]
\[(\text{body-expression})\]

The body expression can be any valid expression in the expression language. Multiple expressions can be executed by enclosing them in a begin expression. Lambdas can be called from within the defines section or throughout the other sections by providing values for its parameters:

\[(\text{identifier params}*)\]

The expression language also includes several control lambdas, defined in definitions.abs for operating across lists etc:

- map(func args) ;; Applies func to the list args, and return the resulting list
- fold(func res args) ;; Applies func to the list args, accumulating the result in res.
- for-each(func args) ;; As map but does not return the resulting list - used for side effects

nodes

Defines a list of all the internal nodes in the circuit. Definitions are of the form:

\[(\text{name width lowbitIndex cardinality})\]

where width, lowbitIndex and cardinality are valid abs language expressions.

gates

Contains the implementation of the component written in the abs language.
Connections

Lists the port to port connections of the component.

```lisp
(defines
  (guard-count (param "guardCount"))
)
(nodes
  (*bypass* (+ 1 guard-count) 0 1)
)
(gates
  (c-element (ack "guard") (ack (each "guardInput")))
  (or (data "guard") (data (each "guardInput")))
  (or (ack "activate") (ack (each "activateOut")))
  (slice guard-count i (node "bypass"))
  (connect (req "activate") (slice 0 1 (node "bypass")))
  (demux2
    (slice 0 guard-count (node "bypass"))
    (slice 1 guard-count (node "bypass"))
    (combine (req (each "activateOut")))
    (combine (data (each "guardInput")))
  )
)

communications

Figure 9.4: Example of component implementation description – the single rail “bar”

9.5. Adding a new technology

Adding new technologies is straigh-forward. The whole process takes about an hour.

1. Create a `<technology>-cells.net` file. Add entries for each cell in the new target cell library. This step may be automated.

2. Create a `gate-mappings` file for the abs-gate operators that are available in the library. Mappings must be provided for anything that is used by the abs descriptions or any logic required to generate helpers by `balsa-make-helpers` such as `and`, `or`, `nor`, `nand`, `nor`, `inverter`, `xor`, `buffer`, `2-1-mux`, `1-2-demux`, `transparent-latch`, `tristate-inverter`, `tristate-buffer`, and `keeper inverters`. If the cell library contains asynchronous cells such as `c-elements`, `s-elements` or `mutexs` they should be be put here as well, otherwise less efficient versions will be generated by `balsa-make-helpers` out of standard cells.

3. Create a `component.abs` file including a link to the common components.abs. If `keeper-inverters` are not available add single rail implementations of `Variable`, `CallMux` and `CaseFetch` components.

4. Create a configuration file.

5. Install the skeleton implementation (or copy it to `share/tech`)

6. Set the environment variable `BALSATECH` to the new technology. Run `balsa-make-helpers` to produce two files: a `balsa-cells.net` file and a `gate-mappings` file. These files contain the descriptions and the mappings of the new cells that have been created by `balsa-make-helpers` and must be concatenated to the original files in the technologies.

7. Create a new `balsa-mgr.cfg` file using the template, in the common directory.

Adding more implementation styles is not easy: New descriptions must be made for each component, and the backend `scheme` code must be updated to inform the system about the structure.
of channels within the new style. A brief specification of the handshake components can be found in `<balsa-home>/doc/components.txt`. An example description is given in “Example,” on page 115

### 9.6. The abs language

Each implementation consists of a list of gate operators or helper cells operating on individual wires or wire vectors, these vectors are obtained by partitioning the port expressions into their constituent components. The following is a discussion of the datatypes and operators present in the abs language.

#### Bundles

A bundle is an expression of the data/signalling wire bundle which can be connected to the port of a handshake component. A bundle therefore may be take any of the forms of the port descriptions (arrayed/sync/data), and several operators allow their manipulation:

- "name" - the complete bundle, refers to the named port expression.
- `bundle “name” index` - used to extract a single channel from an arrayed port expression.
- `bundles “name” index count` - used to extract a range of channels from a bundled array.
- `each “name”` - used to apply an operation on each of the channels in a bundled array.

#### Channels

Channels are the individual communications primitives that constitute the bundles of a component. Channels are composed from several logical groupings of wires called portions. The structure of a channel and its portions depend on the implementation style, sense and direction of the channel.

<table>
<thead>
<tr>
<th>Implementation Style</th>
<th>Channel Sense</th>
<th>Port Sense</th>
<th>Portions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Four Phase Broad/</td>
<td>Push</td>
<td>Passive Input</td>
<td>req</td>
</tr>
<tr>
<td>Reduced Broad</td>
<td></td>
<td></td>
<td>ack</td>
</tr>
<tr>
<td></td>
<td>Pull</td>
<td>Active Output</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Active Input</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Passive Output</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>req</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ack</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data</td>
</tr>
<tr>
<td>Dual Rail</td>
<td>Push</td>
<td>Passive Input</td>
<td>req0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Active Output</td>
<td>req1</td>
</tr>
<tr>
<td></td>
<td>Pull</td>
<td>Active Input</td>
<td>ack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Passive Output</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>req0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ack0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ack1</td>
</tr>
<tr>
<td>One of Four</td>
<td>Push</td>
<td>Passive Input</td>
<td>req0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Active Output</td>
<td>req1</td>
</tr>
<tr>
<td></td>
<td>Pull</td>
<td>Active Input</td>
<td>ack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Passive Output</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>req0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ack0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ack1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ack2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ack3</td>
</tr>
</tbody>
</table>

These portions can be accessed from channels/bundles by the portioning operators:

- `(req bundle)`
- `(ack bundle)`
- `(data bundle)`
9.6. The abs language

(req0 bundle) (req1 bundle) (ack bundle)...

Where bundle, can be a single bundle expression or several bundled arguments:

(req "inp" "out") -> (req "inp") (req "out")
(req (each "inp") -> (req "inp" 0) (req "inp" 1)

These portioning operators return the relevant portion in the form of a slice (see next section), which are needed to be used as input to the gate operators. In order to extract slices from internal nodes with no logical grouping another portioning operator is used:

(node bundle)

where bundle takes the form of one or several internal node names.

Slices

Slices are the basic groups of wires that are manipulated by the gate operators of the system. Slices are a means of constructing a single dimensional wire vector, so must consist of wires of the same direction. Single slices are created by the portioning operators (req, ack, node etc), but slices can also be formed from other slices by using the combine operator:

(combine a b c d)

which combines a, b, c and d into one slice.

N.B. While most abstract gate operators will accept combinatorial slice as operands, some operators (e.g. slice and filter) will only accept single slice operands generated by the portioning operators.

The slice operator is used to extract bit fields from a single slice:

(slice low-bit-index cardinality single-slice)

returns a slice of the input single slice consisting of cardinality wires starting at low-bit-index.

The filter operator can be used to extract arbitrary bit patterns from a single slice:

(filter mask single-slice)

where mask is a decimal representation of the mask required to extract the desired bit pattern.

Slices can be reduced into a list of singleton slices by the smash operator:

(smash single-slice)

this is useful for applying operations across the elements of a slice:

(and (node "out") (smash (data "in")))

would AND all the elements of “in” together.

Slices can also be duplicated with the dup and dup-each operators.

(dup count slice)
(dup-each count slice)

where slice can be an individual slice or a list of several slices.

The dup operator replicates the slice count times, and can be used with the combine operator to make a combinatorial slice allowing the slice to be applied several times across another slice, eg:

(and (node "out") (data "inp") (combine (dup width (req "inp")))))

The dup-each operator replicates each element of a slice or slice list, allowing different arrays of individual wires to be applied across another slice.

(and (node "out") (data (each "inp")) (combine (dup-each width (req (each "inp")))))

Gate Operators

The abs language has several gate operators for performing operations on slices. The gate operators can operate on slices of any size as long as the cardinality of all the slices is equal, and are expanded
to produce several operators acting on single width slices. For example, the \texttt{and} gate operator operating on two nodes, \texttt{a} and \texttt{b}, of width 2 producing a result on node \texttt{c}, also two bits wide:

\begin{verbatim}
(and (node c) (node a) (node b))
\end{verbatim}

would produce two \texttt{and} gate operators:

\begin{verbatim}
(and (slice 0 1 (node c)) (slice 0 1 (node a)) (slice 0 1 (node b))
(and (slice 1 1 (node c)) (slice 1 1 (node a)) (slice 1 1 (node b))
\end{verbatim}

which would be mapped into .net gates as:

\begin{verbatim}
(and2 ("c_0n" 0) ("a_0n" 0) ("b_0n" 0))
(and2 ("c_0n" 1) ("a_0n" 1) ("b_0n" 1))
\end{verbatim}

and into a verilog netlist as:

\begin{verbatim}
(and2x1 c_0n[0],a_0n[0],b_0n[0])
(and2x1 c_0n[1],a_0n[1],b_0n[1])
\end{verbatim}

There are two types of gate operators, fixed or stretchable. Fixed input gates have a fixed number of arguments. Stretchable gates are a small set of basic gates which can take unlimited numbers of input arguments allowing tree-like structures to be created out of these simple gates. If the input and output slices are of plural cardinality, several of these tree structures will be created for each wire in the slices. The size of the gates used to create these trees is determined by the maximum fan-in for gates of the target technology. The abs gate operators are presented in “The ABS Grammar,” on page 120.

The abs language also provides the facility to add customisable cells as illustrated below:

\begin{verbatim}
(cell “cell-name” singleton-slice …)
\end{verbatim}

The cell can have any name, and is mapped to a CAD-specific cell in the technology’s gate-mappings file. The cell descriptions are stored in the technology’s helper cell description file, and must use technology specific instances. The only restrictions on these cells are that they must take singleton slices as inputs.

There are several other gate operators for use in constructing component descriptions:

\begin{verbatim}
(constant value output-slice)
\end{verbatim}

creates a constant from the decimal value, by tying wires to VCC or GND circuits.

\begin{verbatim}
(print args)
\end{verbatim}

prints debugging messages that can be viewed when instances of the component are being generated, takes unlimited number of arguments.

\begin{verbatim}
(macro identifier args)
\end{verbatim}

The macro operator is a method for creating complex abs expressions dependent on some parameters. A macro is lambda defined, as described, in the \texttt{defines} section. The lambda is called using the \texttt{macro} operator. The macro returns a gate operator which is then evaluated – after the execution of the original lambda. The macro is executed, and the resulting expression is evaluated as an abs expression. The arguments to the macro are also not evaluated, allowing abs expressions or snippets to be passed in without resulting in syntax errors.

The abs system also includes control structure to allow more complex customisable components to be created. The format of these structures is similar to the Scheme programming language:

\begin{verbatim}
(if expr
  gate
  gate
)
\end{verbatim}

where, \texttt{gate} is a valid gate operator of the abs language.
Either the consequent or the alternative is selected, depending on the value of `expr`. The syntax of `expr` is given in the “The ABS Grammar,” on page 120.

```
(cond
  (condition-expr gate ...)
  (condition-expr gate ...)
  ...
  (else gate ...)
)
```

Each condition-expr is evaluated in turn, if it evaluates to true then the gates section is executed, and the cond statement is exited. If no expression evaluates to true, the else statement is executed.

```
(case expr
  ((test-value ...) gate ...)
  ((test-value ...) gate ...)
  ...
  (else gate ...)
)
```

This is similar to cond statement except the expression is evaluated and then compared against the constant test values of each statement.

```
(gate gate ...)
```

allows several gates to be substituted into a single gate expression.

Several handshake components encode or decode binary values from/to one-hot wires described using decode/encode gate operators. Because the format of these encodings varies greatly, these components use a specification string to determine the encodings. The actual format of the string is given in the appendix. Each string has at most $n$ terms, one for each of the $n$ one-hot-wires and associated with each term is a value or set of values.

In the encoding string, this value represents the binary value to be output on receipt of activity on the relevant input wire.

In the decoding string, this value represents the input value or range of values that will activate the relevant output wire.

The decode/encode gate operators that are used to provide this logic are:

```
(encode option input-slices output slices)
```

```
(decode option input-slices output slices)
```

For both single and dual-rail implementations dual-rail QDI logic is employed. In single-rail this simplifies the delay-assumptions for components and makes timing-enclosure simpler. The option argument specifies whether to implement the logic in traditional a and/or realisation used for bundled data implementations, or a c-element/or realisation for return-to-zero delay insensitive implementations, or whether to use a m-of-n-mapping allow more complex codes such a 1-of-4 to be handled. For complex codes a mapping-function is passed as an argument to the encode gate that transforms binary implicants into the relevant encoding.

**Example**

This example in Figure 9.5 illustrates the description of the FalseVariable handshake component in a dual-rail implementation style using the abs language.

The FalseVariable component is used to implement passive input enclosure, it allows values of passive inputs to be read in several places without the need for explicit latching. The FalseVariable has three ports:

- "write" - the passive input dataport,
- "signal" - sync port to enclose the "read" port activity within a handshake on the "write" port.
The abs language

9.6. The abs language

- “read” - an arrayed set of passive ports to allow the reading of data in multiple sources.

The component has two parameters:
- “width” - width of the read and write datapaths
- “readPortCount” - number of read-ports.

The passive read and write ports form pull and push channels respectively.

The behaviour of the FalseVariable is as follows. Once a request is received on the write port (in single rail this is signalled by the request line; in dual-rail this requires completion detection to

```
0 (primitive-part "FalseVariable"
1   (parameters
2       ("width" (named-type "cardinal"))
3       ("readPortCount" (named-type "cardinal"))
4   )
5   (ports
6     (port "write" passive input (numeric-type #f (param "width")))
7     (sync-port "signal" active)
8     (arrayed-port "read" passive output
9       (numeric-type #f (param "width") 0 (param "readPortCount"))
10   )
11   (symbol
12     (centre-string "FV")
13   )
14   (implementation
15     (style "four_b_rb" (include tech "common"
16       "data-single-broad/FalseVariable")
17     (style "dual_b"
18       (nodes
19         ("writeSig" 1 0 1)
20         ("writeSigPart" (param "width") 0 1)
21       )
22       (gates
23         (or (node "writeSigPart") (req0 "write") (req1 "write"))
24         (c-element (node "writeSig") (smash (node "writeSigPart")))
25         (s-element (node "writeSig") (ack "write")
26           (req "signal")
27             (ack "signal")
28             ; data read ports
29             (and (combine (ack1 (each "read")))
30             (combine (dup-each (param "width") (req (each "read"))))
31             (combine (dup (param "readPortCount") (req0 "write"))
32             )
33             (and (combine (ack0 (each "read")))
34             (combine (dup-each (param "width") (req (each "read"))))
35             (combine (dup (param "readPortCount") (req1 "write")))
36             )
37         )
38     )
39   )
40 )

Figure 9.5: Description of dual rail FalseVariable
```

- "read" - an arrayed set of passive ports to allow the reading of data in multiple sources.
9.7. Netlists

detect the arrival of valid data) a handshake is initiated on the signal port. This handshake will enclose all of the reads to the set of read-ports. The read-ports are connected to pull-channels and so upon receiving a request they acknowledge with valid-data.

In the dual-rail implementation shown above each channel has a different set of portions:

**Push Channels:** req0, req1, ack. The request on dual-rail push-channels is encoded within the data. The req0 and req1 portions are each the width of the datapath (param "width") and respectively contain the zero and one wires of each dual-rail code group; ack is a single wire used to acknowledge receipt of data on the channel.

**Pull Channels:** req, ack0, ack1. The data of pull-channels enclose the acknowledgement. ack0 and ack1 are each the width of the datapath and contain the zero and one wires of each code group. req is a single wire used to request data.

**Sync Channels:** req, ack have single request and acknowledge wires.

The implementation comprises 3 parts:

1. Completion Detection (lines 21 22). The slice arguments of the \( \text{or} \) gate-operator of line 21 are each the width of the data-path (param "width") This has the affect of placing a single 2-input or-gate for each binary bit of the datapath, detecting the arrival of data in each dual-rail code group of the datapath. These signals are combined to a single signal using the \( \text{c} \)-element of line 22, the \( \text{smash} \) slice operator breaks down (node "writeSigPart") into individual single-wire slices, and so this operator instantiates a tree of \( \text{c} \)-elements the width of the datapath.

2. An \( \text{s} \)-element is used to enclose the "signal" handshake between the completion-detection signal and the acknowledgement of the "write" channel. As each slice is a single wire only one \( \text{s} \)-element is instantiated.

3. The read-ports to the read-channels are instantiated with the two \( \text{and} \) operators (lines 24-31), an operator for each ack0 ack1 portion of the channels. Each operator results in the instantiation of "readPortCount" arrays of AND-gates each of width "width", each slice argument to the and-gates is ("readPortCount" * "width") wide. The bundling operator \( \text{each} \) on line 24 creates a slice for each channel in the read-port array. This is combined to a single-slice with the \( \text{combine} \) command, The input arguments to the and-gate highlight the difference between the \( \text{dup} \) and \( \text{dup-each} \) commands. The \( \text{dup} \) command is used to duplicate the "write" request portions for each read-port, each wire of the write port is duplicated in turn, so each read-port receives all the wires of the write port. The \( \text{dup-each} \) command is used to ensure that each read-port only receives the request wires specific to that read port. The each operator of line 29 expands to "readPortCount" slices of request wires for the read-ports. These slices are then duplicated in turn so as to produce:

\[
(\text{slice } 0 \ 1 \ (\text{req} \ (\text{bundle } \"\text{read}\)\ 0))) \ ... \ (\text{slice width } 1 \ (\text{req} \ (\text{bundle } \"\text{read}\)\ 0))) \ (\text{slice } 0 \ 1 \ (\text{req} \ (\text{bundle } \"\text{read}\)\ 1)))
\]

rather than:

\[
(\text{slice } 0 \ 1 \ (\text{req} \ (\text{bundle } \"\text{read}\)\ 0))) \ ... \ (\text{slice } 0 \ 1 \ (\text{req} \ (\text{bundle } \"\text{read}\)\ \text{readPortCount}))) \ (\text{slice } 1 \ 1 \ (\text{req} \ (\text{bundle } \"\text{read}\)\ 0)))
\]

which would be produced by the \( \text{dup} \) command.

9.7. Netlists

balsa-netlist processes the breeze file by applying the specified parameters to the abs cell descriptions: The gate operators are expanded into instances of abstract gates containing single slice arguments. The stretchable gate operators are expanded into trees of gates of a size determined by the maximum gate fan in of the technology. The channels are expanded into their constituent vector components. The names are mapped to the target gate names, and their arguments re-ordered as necessary. Balsa-netlist then produces a .net netlist which is an internal netlist format, technology dependant, but independent of all CAD system netlist formats. Each technology has several files to control this stage:
9.7. Netlists

gate-mappings.net - This file contains the library cells to use in place of abstract gates and helper cells. Each entry contains the abstract gate name, the technology cell name for each available drive strength of the gate, and the pin mapping that takes place between the abstract gate pin ordering and the actual gate ordering. Eg:

(“and2” (“AND2” 1 2 0) (2 “AND22”) (3 “AND23”) (4 “AND24”))

Here an abstract 2-input and gate maps to the cell AND2, where the first pin (pin 0) of the abstract gate, in this case the output, maps to the last pin of the actual gate, the second (pin 1) to the first pin etc. The customisable gate operators, helper cells, must also be declared in here to allow the same helper cell to have different definitions in the various technologies.

<title>cells.net - This file contains a list of all the cells in the library and their arguments.

balsa-cells.net - This file contains a list of all the helper cells and balsa primitives not supported by the technology, e.g. c-elements, s-elements, arbiters etc.

The expansion process produces an intermediate netlist, based on the constraints of the target technology, but independent of any established netlist format, allowing each technology to produce netlists in various forms. The same format is used to declare the technology and helper cells in the files mentioned above. A circuit declaration The format of a circuit declaration has 4 fields: ports, nets, instances, attributes.

ports:
contains the channels of the input expanded into their constituent vectors. Each vector description is of the form:

(name direction width)

The naming scheme for channel portions is:

<channelname>_<channum><portionid>

<channum> : the channel index. If the channel is unarrayed, this number is always zero.

<portionid> : the portion identifier. Each portion has a different identifier shown below:

r - request wire
a - acknowledge wire
d - data vector
r0d, r1d, r2d, r3d - req data vectors (dual-rail/one-of-four)
a0d, a1d, a2d, a3d - ack data vectors (dual-rail/one-of-four)

nets
contains all the internal nodes of the circuit with arrayed nodes expanded into their individual. A net declaration takes the form:

(netname width)

the naming scheme for nodes is:

<nodename>_<nodenum><nodeid>:

<nodenum> : the node index, zero if unarrayed node.

<nodeid> : “n”.

The above naming schemes apply to generated circuits only: Technology and user defined helper cells are not restricted to this scheme, but must conform to target technology naming schemes.

instances
lists all the instances comprising the circuit. The format of instance declaration is:

(instance instancename (*connection* "connection" ...))
9.8. The BALSATECH environment variable

The BALSATECH environment variable is used to determine the technology, implementation style and any options available for the implementation style.

**Example:**

```
export BALSATECH=example/dualb/variable=spacer:logic=balanced
```

This sets the technology to the `example` technology, using the Dual-Rail backend. The last section sets the style options. Each implementation style has its own style options, these options can be used to change the resultant implementation from the default standard. Examples of style options include changing the cell library or the `variable` option which determines the cell to use for storage inside the Balsa Variable components. The `logic` option determines the style of logic to be used in the Binary-Function components. The format of the options is shown above with options being colon separated. Values can be assigned to options that may take multiple values, Boolean options just need to be set to `true`.

Current stylenames (corresponding to implemetation styles) are:

- `four_b_rb` – bundled data four-phase, broad, reduced broad protocol
- `dual_b` – dual rail delay insensitive encoding with return to zero signalling
9.9. The ABS Grammar

Components

component description:

\[
\begin{align*}
\text{primitive-part} & \ (\text{partname}) \\
& \ (\text{parameter-expr}) \\
& \ (\text{port-expr}) \\
& \ (\text{symbol-expr}) \\
& \ (\text{implementation-expr}) \\
\end{align*}
\]

\[(\text{parameter-expr}) \ (\text{parameters}) (\text{port-expr}) (\text{symbol-expr}) (\text{implementation-expr}) \]

\[(\text{port-type}) \ (\text{passive}) | \ (\text{active}) \]

\[(\text{port-sense}) \ (\text{input}) | \ (\text{output}) \]

\[(\text{symbol-expr}) \ (\text{symbol}) (\text{centre-string}) \]

\[(\text{implementation-expr}) \ (\text{style}) \ (\text{include-expr}) \]

Styles

Descriptions of implementation styles

\[(\text{style-descr}) \ (\text{define-expr}) \ (\text{node-expr}) \ (\text{gate-expr}) \]

\[(\text{define-expr}) \ (\text{defines}) (\text{bound-expr}) \]

\[(\text{node-expr}) \ (\text{nodes}) (\text{width}) (\text{low-bit-index}) (\text{cardinality}) \]

\[(\text{gate-expr}) \ (\text{gates}) (\text{gate-operator}) \]

\[(\text{connection-expr}) \ (\text{connections}) (\text{connect}) (\text{input-slice}) (\text{output-slice}) \]
9.9. The ABS Grammar

Gates

<table>
<thead>
<tr>
<th>Gate Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Fixed Gate)</td>
<td>constant (value) (output-slice)</td>
</tr>
<tr>
<td></td>
<td>stretchable (request-in-slice) (ack-in-slice) (ack-out-slice)</td>
</tr>
<tr>
<td></td>
<td>xor2 (output-slice) (input-slice0) (input-slice1)</td>
</tr>
<tr>
<td></td>
<td>mux2 (output-slice) (input-slice0) (input-slice1) (select-slice)</td>
</tr>
<tr>
<td></td>
<td>demux2 (input-slice) (output-slice0) (output-slice1) (select-slice)</td>
</tr>
<tr>
<td></td>
<td>inv (output-slice) (input-slice)</td>
</tr>
<tr>
<td></td>
<td>keeper-inv (output-slice) (input-slice)</td>
</tr>
<tr>
<td></td>
<td>latch (enable) (input-slice) (output-slice)</td>
</tr>
<tr>
<td></td>
<td>latch-n-enable (enable-slice) (input-slice) (output-slice)</td>
</tr>
<tr>
<td></td>
<td>tri-buffer (enable-slice) (input-slice) (output-slice)</td>
</tr>
<tr>
<td></td>
<td>tri-inv (enable-slice) (input-slice) (output-slice)</td>
</tr>
<tr>
<td></td>
<td>mutex (input-sliceA) (output-slice0) (output-slice1) (select-slice)</td>
</tr>
<tr>
<td></td>
<td>connect (input-slice) (output-slices)+</td>
</tr>
<tr>
<td></td>
<td>and (output-slice) (input-slices)+</td>
</tr>
<tr>
<td></td>
<td>or (output-slice) (input-slices)+</td>
</tr>
<tr>
<td></td>
<td>nor (output-slice) (input-slices)+</td>
</tr>
<tr>
<td></td>
<td>(c-element output-slice) (input-slices)+</td>
</tr>
<tr>
<td>(Stretchable Gate)</td>
<td>gnd (output-slices)+</td>
</tr>
<tr>
<td></td>
<td>vcc (output-slices)+</td>
</tr>
<tr>
<td>(Control Gate)</td>
<td>if-gate-operator</td>
</tr>
<tr>
<td></td>
<td>cond-gate-operator</td>
</tr>
<tr>
<td></td>
<td>case-gate-operator</td>
</tr>
<tr>
<td></td>
<td>gate-gate-operator</td>
</tr>
<tr>
<td>(if-gate-operator)</td>
<td>if (expr)</td>
</tr>
<tr>
<td></td>
<td>(gate-operator)</td>
</tr>
<tr>
<td></td>
<td>(gate-operator)</td>
</tr>
<tr>
<td>(cond-gate-operator)</td>
<td>cond</td>
</tr>
<tr>
<td></td>
<td>if (condition-expr) (gate-operator) +</td>
</tr>
<tr>
<td></td>
<td>(else (gate-operator))?</td>
</tr>
<tr>
<td>(case-gate-operator)</td>
<td>case (expr)</td>
</tr>
<tr>
<td></td>
<td>if (test-value)+ (gate-operator) +</td>
</tr>
<tr>
<td></td>
<td>(else (gate-operator))?</td>
</tr>
<tr>
<td>(gates-gate-operator)</td>
<td>gates (gate-operator)+</td>
</tr>
<tr>
<td>(Other Gate)</td>
<td>constant (value) (output-slice)</td>
</tr>
<tr>
<td></td>
<td>print (arg)+</td>
</tr>
<tr>
<td></td>
<td>macro (macro-name) (macro-args)+</td>
</tr>
<tr>
<td></td>
<td>encode (option) (input-slices)+ (output-slice)</td>
</tr>
<tr>
<td></td>
<td>decode (option) (input-slice) (output-slices)+</td>
</tr>
<tr>
<td></td>
<td>cell &quot;cell-name&quot; (singleton-slice)</td>
</tr>
<tr>
<td>(macro-args)</td>
<td>(identifier)+</td>
</tr>
</tbody>
</table>
9.9. The ABS Grammar

(option) 
| and-or
| c-or
| m-of-n-mapping (mapping-function)

Slices
Slices and slice operators

(slice) 
| ((slice-operator)* (partition-operator)? bundle-expr)

(slice-operator) 
| (combine (slice)+)
| (dup (slice))
| (dup-each (slice))
| (smash (single-slice))
| (filter (single-slice))
| (slice (low-bit-index) (cardinality) (single-slice))

The last 3 operators take (single-slice) arguments, these arguments must be the result of a partitioning operator only and cannot be proceeded by any other slice operator.

(partition-operator) 
| (req bundle-expr)
| (req0 bundle-expr)
| (req1 bundle-expr)
| (req2 bundle-expr)
| (req3 bundle-expr)
| (ack bundle-expr)
| (ack0 bundle-expr)
| (ack1 bundle-expr)
| (ack2 bundle-expr)
| (ack3 bundle-expr)
| (data bundle-expr)
| (node bundle-expr)

(bundle-expr) 
| "(name)"
| bundle "(name)" (index)
| bundles "(name)" (index) (count)
| each "(name)"

Include
description of include statements

(include-stmt) 
| (include (technology-desc) | "(subdirectory)")? "(filename)"

(technology-desc) 
| tech "(tech-name)"

The include statement allows the contents of other .abs files to be inserted into this file. Included .abs files must be present in the components directory (or any sub-directory) of one of the valid Balsa technologies. For example:

(include "ctrl-broad/Sequence")

will include the contents of the file components/ctrl-broad/Sequence.abs in the current technology.

(include tech "common" "ctrl-broad/Sequence")

will include the same file but from the ctrl-broad sub-directory of the components directory of the common technology.

Types
type definitions

(type-expr) 
| (named-type-expr)
| (numeric-type-expr)
| (alias-type-expr)
| (array-type-expr)
| (enumeration-type-expr)
| (record-type-expr)
The type identified by named-type are useful predeclared types (in balsa/types/basic and balsa/types/synthesis) such as cardinality or boolean.

This type is only used a parameters to a select few gates which take a specification string

Expressions

expression types

λ-dec-expr

λ-call-expr

if-expr

arith-expr

fn-expr

encoding-expr

param-expr

bind-name

primitive-expr

λ-dec-expr

λ-call-expr

if-expr

arith-expr

fn-expr

pop-count (expr)

find-set-bit (expr)

find-clear-bit (expr)

style-option (expr)

bit-length (expr)

bit-set? (expr) (expr)
9.10. Netlist Format

Netlist format of .net netlists

\[
\begin{align*}
\text{Netlist} & \quad \text{format of .net netlists} \\
\text{(bit-xor} & \quad (\text{expr} < \text{expr}) \\
\text{(..} & \quad (\text{expr} < \text{expr}) \\
\text{(print} & \quad ((\text{expr})*) \\
\text{(note} & \quad ((\text{expr})*) \\
\text{⟨scheme-expr} & \quad (\text{expt} (\text{exponent-expr} < \text{expr})) \\
\text{⟨encoding-expr} & \quad (\text{strings}) \\
\text{⟨param-expr} & \quad (\text{param}^* (\text{param-name})^*) \\
\text{⟨primitive-expr} & \quad \#t \\
\text{⟨param-list} & \quad (\text{identifiers})^* \\
\text{⟨let-expr} & \quad (\text{bound-exprs})^+ \\
\text{⟨bound-expr} & \quad (\text{identifiers} < \text{expr}) \\
\text{⟨param-list} & \quad (\text{expr})^* \\
\text{⟨arith-op} & \quad (+ | - | * | / | = | /= | > | < | >= | <= 
\end{align*}
\]
9.10. Netlist Format

\[
\text{netlist} \quad \begin{cases} \text{(net-circuit-decl)} \end{cases}^* \\
\text{net-circuit-decl} \quad \begin{cases} \text{circuit} \quad \text{name} \quad \text{net-ports} \quad \text{net-nets} \quad \text{net-instances} \quad \text{net-optional} \end{cases} \\
\text{net-ports} \quad \begin{cases} \text{ports} \quad \text{net-port} \end{cases}^* \\
\text{net-port} \quad \begin{cases} \text{portname} \quad \text{net-direction} \quad \text{cardinality} \end{cases} \\
\text{portname} \quad \begin{cases} \text{channelname} \quad \text{cardinality} \quad \text{portid} \end{cases} \\
\text{portid} \quad \begin{cases} \text{r} \quad \text{a} \quad \text{d} \\
\text{r0d} \quad \text{r1d} \quad \text{r2d} \quad \text{r3d} \\
\text{a0d} \quad \text{a1d} \quad \text{a2d} \quad \text{a3d} \end{cases} \\
\text{net-direction} \quad \begin{cases} \text{input} \\
\text{output} \\
\text{inout} \\
\text{hiz} \end{cases} \\
\text{net-nets} \quad \begin{cases} \text{nets} \quad \text{net-net} \end{cases}^* \\
\text{net-net} \quad \begin{cases} \text{name} \quad \text{cardinality} \end{cases} \\
\text{net-instances} \quad \begin{cases} \text{instances} \quad \text{net-instance} \end{cases}^* \\
\text{net-instance} \quad \begin{cases} \text{instance} \quad \text{name} \quad \text{net-instance-connections} \end{cases} \\
\text{net-instance-connections} \quad \begin{cases} \text{(name) \quad \text{index}} \\
\text{(name) \quad \text{index} \quad \text{cardinality}} \\
\text{name} \\
\text{vector} \\
\text{unconnected} \end{cases} \\
\text{net-vector} \quad \begin{cases} \text{vector} \quad \text{net-instance-connection} \end{cases}^* \\
\text{net-optional} \quad \begin{cases} \text{attributes} \quad \text{net-attribute} \end{cases}^* \\
\text{net-attribute} \quad \begin{cases} \text{name} \quad \text{value} \end{cases} \\
\text{attributes} \quad \begin{cases} \text{cell-type} \quad \text{cell-name} \\
\text{global-ports} \quad \text{portname} \\
\text{feedthrough} \quad \text{port-indices} \\
\text{simulation-reset} \quad \text{simulation-signal} \end{cases} \\
\text{port-indices} \quad \begin{cases} [0-9]^+ \begin{cases} [0-9]^+ \end{cases}^* \\
\text{simulation-signal} \quad \begin{cases} \text{(net-instance-connection) \quad \text{simulation-value}} \end{cases}^+ \\
\text{simulation-value} \quad \begin{cases} 0 \quad 1 \quad x \quad z \end{cases} \end{cases}
\]
10 Balsa Reference

10.1. Summary

This chapter documents the command line interface to some of the more important components of the Balsa system. Balsa-mgr is a GUI to these programs, but the expert user may wish to build their own design flow by calling these programs directly.

10.2. Balsa programs

\texttt{balsa-c \{<switch>\}* <block/file-name>}

The switches are:

- \texttt{-l <path>} append \texttt{<path>} to import file path (\texttt{--import})
- \texttt{-e} discard import path (\texttt{--discard-import})
- \texttt{-o <directory>} directory for output intermediate files (\texttt{--output})
- \texttt{-O} DON'T optimise generated HC's (\texttt{--no-optimise})
- \texttt{-b} inhibit banner (\texttt{--no-banner})
  tabs indent by \texttt{<distance>} places (\texttt{--tab})
- \texttt{-t <distance>} Used for identifying correctly column numbers in the source code when error reporting.
- \texttt{-v} be verbose (\texttt{--verbose})
  compilation option (\texttt{--compile-option})
  \texttt{<option>} can be:
  \texttt{allow-sequential-selection} -- allow the generation of the non-delay-insensitive \texttt{BrzCallDemuxPush} and \texttt{BrzActive} components
- \texttt{-c <option>} \textbf{Deprecated} code generation features:
  \texttt{var-read-split} -- split variables on read bitfields as well as writes
  \texttt{no-wire-forks} -- don't use the WireFork component as a replacement for permanent Forks
  \texttt{use-masks} -- use Mask components instead of slice
- \texttt{-} accept no more switches
- \texttt{-B} don't generate a Breeze file (\texttt{--no-breeze})
10.2. Balsa programs

balsa-netlist {<switch>}* <block/file-name>

The switches are:

-F generate a flat Breeze file (--flatten-files)
-i suppress import [balsa.types.synthesis] line in output
(-no-imports-in-output)
-j suppress ‘import [balsa.types.synthesis]’ line in output
(-no-imports-in-output)
-p decorate error/warning messages with balsa-c source position
(-error-positions)
-M report a list of imported blocks on which this file depends (-
(-depends)
used by balsa-md for its dependency analysis
-p <type> Type can be: (runtime | report | error | warning | fatal)
(--print-type)

-h, -? Display this message (--help)
-b Don't print the balsa-netlist banner (--no-banner)
-v Be verbose, print cell names as they are produced. (--verbose)
-c Don't try to make a CAD system native netlist (--no-cad-netlist)
-m Don't read in old cell name mappings from the .map file (--no-old-cell-
names)
-d Dump a netlist in the given format (edif, verilog, compass ...)
any other scheduled netlist writes, several -n can be used (--make-other-
netlist).
-n <format> NB. Name mapping/mangling occurs when the internal netlist is
generated, all of these additional netlists will contain names mapped to
work with the default format.
-d Don't print prototypes for undefined cells (where appropriate).
(-no-prototypes)
-p Do print prototypes for undefined cells (where appropriate)
(-prototypes)
-i <type> Add cell type <type> to the list of cell types to netlist. If no additional cell
types are given, then only the netlist definitions for Balsa cells are emitted
(--include-cell-type)
-x <cellname> Exclude the cell <cellname> from the generated netlist. No definition or
prototype will be emitted (--exclude-cell)
-l <directory> Add named directory to the Breeze import path (--import)
-t <component> <args> create test component (--test-component)
-l <filename> Make a list of generated files in file <filename> (--file-list)
-a Emit definitions for all parts found even if the top level block doesn't need
them (--all-parts)
-s Insert simulation initialisation code in netlist formats which support this
option (--simulation-initialise)
-L <filename> write a log of balsa-netlist messages to file <filename> (--log)
-f Replace feedthrough cells with netlist appropriate aliases
(--replace-feedthroughs)
-g Propagate global ports on cells (--propagate-globals)
-e <procedureName> Produce encounter compatible netlist. This option buffers the ports of procedure <procedureName> and propagates power and ground connections through sub-procedures to top level (\textit{--encounter})
10.2. Balsa programs
The Balsa Language Definition

11.1. Summary

The syntax of the balsa language is given in this chapter. An extended form of BNF is used to describe the syntax. A term \( a^n \) denotes zero or more repetitions of the term \( a \), the term \( a^+ \) denotes one or more repetitions of \( a \) and \( a^? \) indicates that the term \( a \) is optional (i.e. zero or one repetitions of the term \( a \)). Terminal symbols are shown in **bold face**, non terminal symbols are enclosed by angle brackets \( \langle \rangle \).

11.2. Reserved words

The following are reserved words. Most (but not all) correspond to current keywords in the Balsa language, others are reserved for future releases of the Balsa system.

active, also, and, arbitrate, array, as, begin, bits, case, channel, constant, continue, else, end, enumeration, for function, halt, if, import, in, input, is, let, local, log, loop, multicast, new, not, of, or, output, over, parameter, passive, print, procedure, pull, push, record, select, shared, signed, sizeof, sync, then, type, val, variable, while, xor.

11.3. Balsa Language Definition

\[
\begin{align*}
\langle \text{bin-digit} \rangle & ::= \ (0 | 1) \\
\langle \text{oct-digit} \rangle & ::= \ (0 \ldots 7) \\
\langle \text{dec-digit} \rangle & ::= \ (0 \ldots 9) \\
\langle \text{hex-digit} \rangle & ::= \ (0 \ldots 9 | \text{a} \ldots \text{f} | \text{A} \ldots \text{F}) \\
\langle \text{letter} \rangle & ::= \ (\text{a} \ldots \text{z} | \text{A} \ldots \text{Z})
\end{align*}
\]
11.3. Balsa Language Definition

\[ \langle \text{identifier} \rangle ::= (\langle \text{letter} \rangle | \_)(\langle \text{letter} \rangle | \langle \text{dec-digit} \rangle | \_)* \]

\[ \langle \text{literal} \rangle ::= (1 \ldots 9)(\langle \text{dec-digit} \rangle | \_)* \]

\[ \langle \text{string} \rangle ::= (\langle \text{letter} \rangle | \langle \text{dec-digit} \rangle | \! | \# | \$ | \% | \& | \( | \| | \* | + | , | . | / | : | ; | < | = | > | \_ | ? | @ | \[ | \| | ^ | \_ | ^ | \{ | \} | ~ ) \]

\[ \langle \text{string-char} \rangle ::= (\langle \text{letter} \rangle | \langle \text{dec-digit} \rangle | \! | \# | \$ | \% | \& | \( | \| | \* | + | , | . | / | : | ; | < | = | > | ? | @ | \[ | \| | ^ | \_ | ^ | \{ | \} | ~ ) \]

\[ \langle \text{file} \rangle ::= (\text{import} [\langle \text{dotted-path} \rangle ])*\langle \text{outer-declarations} \rangle \]

\[ \langle \text{dotted-path} \rangle ::= \langle \text{identifier} \rangle (\langle \_ \langle \text{identifier} \rangle \rangle )*\]

\[ \langle \text{outer-declarations} \rangle ::= (\langle \text{outer-declaration} \rangle )* \]

\[ \langle \text{outer-declaration} \rangle ::= \text{type} \langle \text{identifier} \rangle \text{is} \langle \text{type-declaration} \rangle \]

\[ \langle \text{type-declaration} \rangle ::= \langle \text{type} \rangle \]

\[ \langle \text{identifiers} \rangle ::= \langle \text{identifier} \rangle (\langle \_ \langle \text{identifier} \rangle \rangle )* \]

\[ \langle \text{type} \rangle ::= \langle \text{identifier} \rangle | \langle \text{expression} \rangle (\text{signed})? \text{\_bits} \]

\[ \langle \text{expression} \rangle \text{\_array} \langle \text{range} \rangle \text{\_of} \langle \text{type} \rangle \]
11.3. Balsa Language Definition

\[
\langle \text{function-formals} \rangle ::= \langle \text{identifiers} \rangle : \langle \text{type} \rangle ( \langle \text{function-formals} \rangle \rangle ^* \\
\langle \text{builtin-function-formals} \rangle ::= \langle \text{formal-parameters} \rangle \\
| \langle \text{function-formals} \rangle \\
| \langle \text{function-formals} \rangle ; \langle \text{function-formals} \rangle \\
\langle \text{procedure-formals} \rangle ::= \langle \text{formal-parameters} \rangle \\
| \langle \text{formal-ports} \rangle \\
| \langle \text{formal-parameters} \rangle ; \langle \text{formal-ports} \rangle \\
\langle \text{formal-parameters} \rangle ::= \text{parameter} \langle \text{identifiers} \rangle : \langle \text{type} \rangle \\
| ( \langle \text{function-formals} \rangle ) \\
\langle \text{formal-ports} \rangle ::= \langle \text{formal-port} \rangle ( \langle \text{formal-port} \rangle ) ^* \\
\langle \text{formal-port} \rangle ::= ( \text{array} (\text{range}) \text{of} )? ( \text{input} | \text{output} ) \langle \text{identifiers} \rangle : \langle \text{type} \rangle \\
| ( \text{array} (\text{range}) \text{of} )? \text{sync} \langle \text{identifiers} \rangle \\
| \text{if} \langle \text{expression} \rangle \text{then} \langle \text{formal-ports} \rangle \\
| ( \text{if} \langle \text{expression} \rangle \text{then} \langle \text{formal-ports} \rangle )^* \\
| \text{else} \langle \text{formal-ports} \rangle \end \\
\langle \text{range} \rangle ::= \langle \text{expression} \rangle \\
| \langle \text{expression} \rangle .. \langle \text{expression} \rangle \\
| \text{over} \langle \text{type} \rangle \\
\langle \text{inner-declarations} \rangle ::= ( \langle \text{inner-declaration} \rangle )^* \\
\langle \text{inner-declaration} \rangle ::= \langle \text{outer-declaration} \rangle \\
| \text{variable} \langle \text{identifiers} \rangle : \langle \text{type} \rangle \\
| \langle \text{channel-options} \rangle ( \text{array} (\text{range}) \text{of} )? \text{channel} \langle \text{identifiers} \rangle : \langle \text{type} \rangle \\
| \langle \text{channel-options} \rangle ( \text{array} (\text{range}) \text{of} )? \text{sync} \langle \text{identifiers} \rangle \\
| \text{shared} \langle \text{identifier} \rangle \text{is} ( \text{local} )? \langle \text{inner-declarations} \rangle \\
| \text{begin} \langle \text{command} \rangle \text{end} \\
| \text{if} \langle \text{expression} \rangle \text{then} \langle \text{inner-declarations} \rangle \\
| ( \text{if} \langle \text{expression} \rangle \text{then} \langle \text{inner-declarations} \rangle )^* \\
| \text{else} \langle \text{inner-declarations} \rangle \end \\
\langle \text{channel-options} \rangle ::= ( \text{multicast} )? \\
\langle \text{expression} \rangle ::= \langle \text{identifier} \rangle \\
| \langle \text{literal} \rangle \\
| \langle \text{string} \rangle \\
| ( \langle \text{identifier} \rangle )? \{ \langle \text{expressions} \rangle \} \\
| \langle \text{identifier} \rangle \ ' \langle \text{identifier} \rangle \\
| \langle \text{unary-operator} \rangle \langle \text{expression} \rangle
11.3. Balsa Language Definition

```plaintext
sizeof (identifier)
| (expression) (binary-operator) (expression)
| (expression) . (identifier)
| (expression) [ (range) ]
| ( (expression) as (type) )
| # (expression)
| ( (expression) )
| (identifier) ( (expressions) )

expressions ::= (expression) ( , (expression) )*

unary-operator ::= ( - | + | not | log | # )

binary-operator ::= ( + | - | * | / | % | ^ | = | /= | < | > | <= | >= | and | or | xor | @ )

command ::= continue
| halt
| (channel) -> (lvalue)
| (channel) -> (channel)
| (channels) -> then (command) end
| (channels) ->! then (command) end
| (channel) <- (expression)
| sync (channel)
| (lvalue) := (expression)
| (block)
| (command) ; (command)
| (command) || (command)
| (command) || ! (command)
| loop (command) end
| loop (command) while (expression) end
| loop ((command)) ? while (guards) ( also (command) )? end
| if (guards) ( else (command) )? end
| case (expression) of (case-guard)
| ( | (case-guard) )* (else (command))?
| end
| for ( | | ; ) (identifier) in (range) then (command) end
| select (channel-guard) ( | (channel-guard) )* end
| select! (channel-guard) ( | (channel-guard) )* end
| arbitrate (channel-guard) | (channel-guard) end
```
11.3. Balsa Language Definition

\[
\begin{align*}
\langle \text{channels} \rangle & ::= \langle \text{channel} \rangle ( , \langle \text{channel} \rangle )^* \\
\langle \text{channel} \rangle & ::= \langle \text{identifier} \rangle \\
\langle \text{lvalue} \rangle & ::= \langle \text{identifier} \rangle \\
& \quad \langle \text{lvalue} \rangle . \langle \text{identifier} \rangle \\
& \quad \langle \text{lvalue} \rangle \ [ \langle \text{expression} \rangle \ ] \\
\langle \text{block} \rangle & ::= ( \text{local} \langle \text{inner-declarations} \rangle )? \begin{array}{l} \text{begin} \langle \text{command} \rangle \end{array} \end{array} \\
& \quad \begin{array}{l} \text{end} \end{array} \\
\langle \text{guards} \rangle & ::= \langle \text{expression} \rangle \ \text{then} \langle \text{command} \rangle \\
& \quad ( | \langle \text{expression} \rangle \ \text{then} \langle \text{command} \rangle )^* \\
\langle \text{case-guard} \rangle & ::= \langle \text{case-matches} \rangle \ \text{then} \langle \text{command} \rangle \\
& \quad \text{for} \langle \text{identifier} \rangle \ \text{in} \langle \text{case-matches} \rangle \ \text{then} \langle \text{command} \rangle \\
\langle \text{case-match} \rangle & ::= \langle \text{expression} \rangle \\
& \quad \langle \text{expression} \rangle \ldots \langle \text{expression} \rangle \\
\langle \text{case-matches} \rangle & ::= \langle \text{case-match} \rangle ( , \langle \text{case-match} \rangle )^* \\
\langle \text{channel-guard} \rangle & ::= \langle \text{channels} \rangle \ \text{then} \langle \text{command} \rangle \\
& \quad ( | \langle \text{channels} \rangle \ \text{then} \langle \text{command} \rangle )^* \\
\langle \text{procedure-actuals} \rangle & ::= \langle \text{actual-parameters} \rangle \\
& \quad \langle \text{actual-channels} \rangle \\
& \quad \langle \text{actual-parameters} \rangle , \langle \text{actual-channels} \rangle \\
\langle \text{actual-parameters} \rangle & ::= \langle \text{actual-parameter} \rangle ( , \langle \text{actual-parameter} \rangle )^* \\
\langle \text{actual-parameter} \rangle & ::= \langle \text{expression} \rangle \\
& \quad ( \text{type} )? \langle \text{type} \rangle \\
\langle \text{actual-channels} \rangle & ::= \langle \text{actual-channel} \rangle ( , \langle \text{actual-channel} \rangle )^* \\
\langle \text{actual-channel} \rangle & ::= \langle \text{identifier} \rangle \\
& \quad \langle \text{actual-channel} \rangle \ [ \langle \text{range} \rangle \ ] \\
& \quad \langle \text{lvalue} \rangle \\
& \quad \langle \text{block} \rangle \\
& \quad \{ \langle \text{actual-channel} \rangle ( , \langle \text{actual-channel} \rangle )^* \} \\
& \quad \langle \text{actual-channel} \rangle @ \langle \text{actual-channel} \rangle
\end{align*}
\]
12 The Breeze Language Definition

12.1. Summary

Breeze is the intermediate language used for compiled Balsa programs. It serves as a repository for libraries and is the level at which all tools in the Balsa system interact. Users who wish to use components described outside of Balsa need to provide a Breeze wrapper for those components so that they may be used within the Balsa system.

12.2. Breeze Language Definition

Lexical

\[\begin{align*}
\langle \text{dec-dig} \rangle & ::= \ (0 \ldots 9) \\
\langle \text{lc-letter} \rangle & ::= \ (a \ldots z) \\
\langle \text{letter} \rangle & ::= \ (a \ldots z \mid A \ldots Z) \\
\langle \text{positive} \rangle & ::= \ (1 \ldots 9) (\langle \text{dec-dig} \rangle)^* \\
\langle \text{natural} \rangle & ::= \ (0 \mid \langle \text{positive} \rangle) \\
\langle \text{integer} \rangle & ::= \ (\langle \text{natural} \rangle \mid -\langle \text{positive} \rangle) \\
\langle \text{identifier} \rangle & ::= \ "(\langle \text{letter} \rangle \mid _) (\langle \text{letter} \rangle \mid \langle \text{dec-dig} \rangle \mid _)^* " \\
\langle \text{dotted-identifier} \rangle & ::= \ "(\langle \text{letter} \rangle \mid _.) (\langle \text{letter} \rangle \mid \langle \text{dec-dig} \rangle \mid _.\)"^* " \\
\langle \text{boolean} \rangle & ::= \ (\#t \mid \#f) \\
\langle \text{symbol} \rangle & ::= \ \langle \text{lc-letter} \rangle \ (\langle \text{lc-letter} \rangle \mid \langle \text{dec-dig} \rangle \mid _) \ : \ )^* \\
\langle \text{string-char} \rangle & ::= \ (\langle \text{letter} \rangle \mid \langle \text{dec-dig} \rangle \mid ! \mid \# \mid $ \mid \% \mid & \mid ( \mid ) \mid * \mid + \mid , \mid - \mid . \mid / \mid : \mid ; \mid < \mid = \mid > \mid ? \mid @ \mid [ \mid ] \mid ^ \mid _ \mid \} \mid \{ \mid \{ \mid | \mid \sim \mid \$ \mid \" \mid ) \\
\langle \text{quoted-symbol} \rangle & ::= \ "\langle \text{symbol} \rangle "
\end{align*}\]
Main Breeze Language

\[
\begin{align*}
\langle \text{breeze-file} \rangle & ::= (\langle \text{import} \rangle)^* (\langle \text{definition} \rangle)^* \\
\langle \text{import} \rangle & ::= (\text{import} \langle \text{dotted-identifier} \rangle) \\
\langle \text{definition} \rangle & ::= \langle \text{constant-defn} \rangle \\
& \quad | \langle \text{type-defn} \rangle \\
& \quad | \langle \text{part-defn} \rangle \\
& \quad | \langle \text{balsa-defn} \rangle \\
& \quad | \langle \text{netlist-defn} \rangle \\
& \quad | \langle \text{composition-defn} \rangle \\
\langle \text{constant-defn} \rangle & ::= (\text{constant} \langle \text{identifier} \rangle \langle \text{integer} \rangle \langle \text{type} \rangle) \\
\langle \text{type-defn} \rangle & ::= (\text{type} \langle \text{identifier} \rangle \langle \text{type} \rangle) \\
\langle \text{type} \rangle & ::= (\text{numeric-type} \langle \text{boolean} \rangle \langle \text{positive} \rangle) \\
& \quad | (\text{enumeration-type} \langle \text{boolean} \rangle \langle \text{positive} \rangle ( (\langle \text{identifier} \rangle \langle \text{integer} \rangle) )^+ ) \\
& \quad | (\text{record-type} \langle \text{positive} \rangle ( (\langle \text{identifier} \rangle \langle \text{type} \rangle )^+ ) \\
& \quad | (\text{array-type} \langle \text{type} \rangle \langle \text{integer} \rangle \langle \text{positive} \rangle \langle \text{type} \rangle) \\
\langle \text{part-defn} \rangle & ::= (\text{breeze-part} \langle \text{identifier} \rangle \\
& \quad \quad (\text{ports} (\langle \text{part-port} \rangle)^* ) \\
& \quad \quad (\text{attributes} (\langle \text{part-attribute} \rangle)^* ) \\
& \quad \quad (\text{channels} (\langle \text{channel} \rangle)^* ) \\
& \quad \quad (\text{components} (\langle \text{component} \rangle)^* ) ) \\
\langle \text{balsa-defn} \rangle & ::= ... \\
\langle \text{part-port} \rangle & ::= (\text{sync-port} \langle \text{identifier} \rangle \langle \text{port-sense} \rangle \langle \text{position} \rangle (\langle \text{option} \rangle)^* ) \\
& \quad | (\text{port} \langle \text{identifier} \rangle \langle \text{port-sense} \rangle \langle \text{port-direction} \rangle \langle \text{type} \rangle \langle \text{position} \rangle (\langle \text{option} \rangle)^* ) \\
& \quad | (\text{arrayed-port} \langle \text{identifier} \rangle \langle \text{port-sense} \rangle \langle \text{port-direction} \rangle \langle \text{type} \rangle \langle \text{integer} \rangle \langle \text{positive} \rangle \langle \text{type} \rangle \langle \text{position} \rangle (\langle \text{option} \rangle)^* ) \\
& \quad | (\text{arrayed-sync-port} \langle \text{identifier} \rangle \langle \text{port-sense} \rangle \langle \text{integer} \rangle \langle \text{positive} \rangle \langle \text{type} \rangle \langle \text{position} \rangle (\langle \text{option} \rangle)^* ) \\
\langle \text{port-sense} \rangle & ::= (\text{active} | \text{passive} ) \\
\langle \text{port-direction} \rangle & ::= (\text{input} | \text{output} ) \\
\langle \text{part-attribute} \rangle & ::= (\text{is-procedure} ) \\
& \quad | (\text{is-function} ) \\
& \quad | (\text{is-permanent} ) \\
& \quad | (\text{view-attribute} ) \\
& \quad | (\text{position} )
\end{align*}
\]
12.2. Breeze Language Definition

```
Breeze extensions

| ⟨option⟩ | := (at ⟨natural⟩ ⟨natural⟩ ⟨dotted-identifier⟩ ⟨natural⟩ )
| ⟨position⟩ | := ⟨symbol⟩ (⟨value⟩ )+ )
| ⟨option⟩ | := ⟨integer⟩
| ⟨value⟩ | := (⟨identifier⟩
| ⟨value⟩ | := ⟨boolean⟩
| ⟨value⟩ | := ⟨symbol⟩
| ⟨value⟩ | := ⟨integer⟩)
| ⟨option⟩ | := ⟨identifier⟩
| ⟨channel⟩ | := (sync ⟨position⟩ (⟨option⟩ )+ )
| ⟨channel⟩ | := ⟨channel-sense⟩ ⟨positive⟩ ⟨position⟩ (⟨option⟩ )+ )
| ⟨channel-sense⟩ | := (push | pull )
| ⟨component⟩ | := (component ⟨identifier⟩
| ⟨parameter⟩ | := (⟨integer⟩ | ⟨identifier⟩ )
| ⟨channel-no⟩ | := (⟨positive⟩
| ⟨channel-no⟩ | := (⟨positive⟩ )+ )

⟨view-attribute⟩ | := (view ⟨identifier⟩ ⟨view-spec⟩ ⟨options⟩ )
| ⟨view-spec⟩ | := (lines-spec ⟨string⟩+ )
| ⟨view-spec⟩ | := (list-spec ⟨quoted-symbol⟩ ⟨value⟩ )
| ⟨netlist-defn⟩ | := (breeze-netlist ⟨identifier⟩
| ⟨net-port⟩ | := (⟨identifier⟩ ⟨net-direction⟩ ⟨positive⟩ )
| ⟨net-direction⟩ | := (input | output | inout | hiz )
| ⟨net-net⟩ | := (⟨identifier⟩ ⟨positive⟩ )
| ⟨net-instance⟩ | := (instance ⟨identifier⟩
| ⟨net-instance⟩ | := (instance ⟨identifier⟩
| ⟨net-instance-connection⟩ | := ⟨net-single-instance-connection⟩
| ⟨net-instance-connection⟩ | := ⟨net-single-instance-connection⟩
| ⟨net-instance-connection⟩ | := (vector ⟨net-single-instance-connection⟩ + )
| ⟨net-single-instance-connection⟩ | := ⟨identifier⟩
```

Balsa: A Tutorial Guide. V3.5 - Printed: 19/5/06

139
12.2. Breeze Language Definition

\[
\begin{align*}
\langle\text{breeze-net-attribute}\rangle & \ ::= \ \langle\text{view-attribute}\rangle \\
& \quad \mid \ \langle\text{option}\rangle \\
\langle\text{composition-defn}\rangle & \ ::= \ (\text{breeze-composition}\langle\text{identifier}\rangle) \\
& \quad \ (\text{ports}\ (\text{part-port})^* ) \\
& \quad \ (\text{attributes}\ (\text{comp-attribute})^* ) \\
& \quad \ (\text{nets}\ (\text{net-net})^* ) \\
& \quad \ (\text{instances}\ (\text{comp-instance})^* ) \\
\langle\text{comp-instance}\rangle & \ ::= \ (\text{instance}\langle\text{identifier}\rangle) \\
& \quad \ ((\langle\text{comp-instance-connection}\rangle)^* ) \\
& \quad \ (\langle\text{option}\rangle)^* ) \\
\langle\text{comp-instance-connection}\rangle & \ ::= \ (\text{net-single-instance-connection}) \\
& \quad \mid \ (\text{comp-single-instance-connection}) \\
& \quad \mid \ (\text{vector}\ (\langle\text{comp-instance-connection}\rangle)^+ ) \\
\langle\text{comp-single-instance-connection}\rangle & \ ::= \ (\langle\text{comp-portion-connection}\rangle\langle\text{natural}\rangle) \\
& \quad \mid \ (\langle\text{comp-portion-connection}\rangle\langle\text{natural}\rangle\langle\text{positive}\rangle) \\
& \quad \mid \ (\langle\text{comp-portion-connection}\rangle) \\
\langle\text{comp-portion-connection}\rangle & \ ::= \ (\text{identifier}) \\
& \quad \mid \ (\text{portion}\langle\text{identifier}\rangle) \\
& \quad \mid \ (\text{portion}\langle\text{identifier}\rangle\langle\text{natural}\rangle) \\
\langle\text{comp-attribute}\rangle & \ ::= \ (\langle\text{view-attribute}\rangle) \\
& \quad \mid \ (\text{position}) \\
& \quad \mid \ (\langle\text{option}\rangle) \\
\langle\text{portion}\rangle & \ ::= \ (\text{portion-name}) \\
& \quad \mid \ (\text{portion-name}\langle\text{natural}\rangle) \\
\langle\text{portion-name}\rangle & \ ::= \ (\text{req}|\text{ack}|\text{data})
\end{align*}
\]
13 The Breeze Components

13.1. Summary

Each of the handshake components described in this section is accompanied by a description of that component’s behaviour. This behaviour is expressed in a notation invented by Bardsley [5], which is a modified form of van Berkel’s *handshake circuit calculus* [1].

13.2. Activation driven control components

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

13.3. Channel termination components

These components are: Continue, ContinuePush, Halt and HaltPush.

13.4. Control to datapath interface components

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

### 13.5. Pull datapath components

Compiled data operations (+, -, ...) in Balsa consist of a sync channel meeting a transferrer causing a result to be requested from a tree of pull datapath components implementing the required function and pushing that result onto an output channel or into a variable (variables are the components which implement HDL level variables as latches). The pull datapath components form an activation driven tree in the same way as control components but with variables or input channels at the leaves. The activations of these components are pull ports with the incoming request flowing (and forking) towards the leaves of the tree with the result flowing (and joining) back to the root forming the result acknowledgement. The datapath components are: Adapt, Slice, Constant, Combine, CombineEqual, CaseFetch, UnaryFunc, BinaryFunc and BinaryFuncConstR.

### 13.6. Connection components

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

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

### 13.7. Non-delay-insensitive components

These are unsafe components whose behaviour can break due to race conditions. They are generated by the Balsa compiler when sequenced select/arbitrate statements on the same channel are used. The activation of their input leads to the activation of all their outputs, but only one output acknowledgement is expected in return. Other outputs will be Returned-To-Zero (if 4-phase protocol) even without a proper acknowledgement. These components are: CallActive and CallDemuxPush.

### 13.8. Simulation-only components

These components are only used for simulation. They cannot be synthesised. Currently, only one component fits in this category: BuiltinVariable.

### 13.9. Breeze components ordered by name
13.9. Breeze components ordered by name

**ActiveEagerFalseVariable**

( `parameter` width : cardinal;
  `parameter` readPortCount : cardinal;
  `parameter` specification : string;
  `passive` sync activate;
  `active` input write : width bits;
  `active` sync signal;
  `array` readPortCount of `passive` output read )

#[ activate: write ?* signal : v := write ] || #[ read0 !* v ] || ... || #[ read readPortCount-1 !* v ]

**Adapt**

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

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

**Arbiter**

( `sync` inpA, inpB, outA, outB )

#[ [ inpA : inpB | inpB : outB ] ]

**Bar**

( `parameter` guardCount : cardinal;
  `passive` output guard : 1 bits;
  `passive` sync activate;
  `array` guardCount of `active` input guardInput : 1 bits;
  `array` guardCount of `active` sync activateOut )

#[ guard !* ( c := [ guardInput0 ?* ... ?* guardInput guardCount-1 ?* choose(guardInput0, ..., guardInput guardCount-1) ] != -1 ) ] ||
#[ activate : activateOut ]
BinaryFunc

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

BinaryFuncConstR

( parameter outputWidth : cardinal;
  parameter inputWidth : cardinal;
  parameter constWidth : cardinal;
  parameter op : BinaryOperator;
  parameter outputIsSigned : boolean;
  parameter inputIsSigned : boolean;
  parameter constIsSigned : boolean;
  parameter constant : constWidth bits;
  passive output out : outputWidth bits;
  active input inp : inputWidth bits )

BuiltinVariable

( parameter readPortCount : cardinal;
  parameter name : string;
  passive input write : 64 bits;
  array readPortCount of output read : 64 bits )

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

# [ out !° inp ?° op(outputWidth, outputIsSigned, inputIsSigned, constIsSigned, op, constant, inp) ]

# [ write !° v := write ] ||
# [ read0 ! v ] || ... || # [ read(readPortCount-1) !° v ]
13.9. Breeze components ordered by name

**Call**

(\texttt{parameter} \texttt{inputCount} : \texttt{cardinal};
\texttt{array} \texttt{inputCount} of \texttt{passive sync} \texttt{inp};
\texttt{active sync} \texttt{out})

\[ \texttt{[ [ \texttt{inp}_0 : \texttt{out} | ... | \texttt{inp}_{\texttt{inputCount}-1} : \texttt{out} ]} \]

**CallMux**

(\texttt{parameter} \texttt{width} : \texttt{cardinal};
\texttt{parameter} \texttt{inputCount} : \texttt{cardinal};
\texttt{array} \texttt{inputCount} of \texttt{passive input} \texttt{inp} : \texttt{width bits};
\texttt{active output} \texttt{out} : \texttt{width bits})

\[ \texttt{[ [ \texttt{out} ! \cdot \texttt{inp}_0 | ... | \texttt{out} ! \cdot \texttt{inp}_{\texttt{inputCount}-1} ]} \]

**CallDemux**

(\texttt{parameter} \texttt{width} : \texttt{cardinal};
\texttt{parameter} \texttt{outputCount} : \texttt{cardinal};
\texttt{array} \texttt{outputCount} of \texttt{passive output} \texttt{out} : \texttt{width bits};
\texttt{active input} \texttt{inp} : \texttt{width bits})

\[ \texttt{[ [ \texttt{out}_0 ! \cdot \texttt{inp} ? \cdot \texttt{inp} | ... | \texttt{out}_{\texttt{outputCount}-1} ! \cdot \texttt{inp} ? \cdot \texttt{inp} ]} \]

**CallActive**

(\texttt{parameter} \texttt{outputCount} : \texttt{cardinal};
\texttt{passive sync} \texttt{inp};
\texttt{array} \texttt{outputCount} of \texttt{active sync} \texttt{out})

\[ \texttt{[ \texttt{inp} : \texttt{out}_0, ... , \texttt{out}_{\texttt{outputCount}-1} ]} \texttt{but non-DI: RTZ to all outputs when first ack received.} \]

**CallDemuxPush**

(\texttt{parameter} \texttt{width}, \texttt{outputCount} : \texttt{cardinal};
\texttt{passive input} \texttt{inp} : \texttt{width bits};
\texttt{array} \texttt{outputCount} of \texttt{active output} \texttt{out} : \texttt{width bits})

\[ \texttt{[ \texttt{inp} : \texttt{out}_0, ... , \texttt{out}_{\texttt{outputCount}-1} ]} \texttt{but non-DI: RTZ to all outputs when first ack received.} \]
13.9. Breeze components ordered by name

**Case**

( parameter inputWidth : cardinal;
  parameter outputCount : cardinal;
  parameter specification : string;
  passive input inp : inputWidth bits;
  array outputCount of active sync activateOut )

# [ inp ?\[ decode(outputCount, specification, inp) ! = - 1 ->
    activateOut[decode(outputCount, specification, inp) ! = - 1 ] ]

**CaseFetch**

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

# [ out !° index ?° inp_index ?° inp_index ]

**Combine**

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

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

**CombineEqual**

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

# [ out !° inp_0 ?° ... ?° inp_{inputCount-1} ?° combineEqual(inp_0, ..., inp_{inputCount-1}) ]
13.9.  Breeze components ordered by name

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

# [ activate : [ activateOut_0 || ... || activateOut_\text{outputCount-1} ] ]

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

# [ out ! of value ]

**Continue**
( passive sync inp  
)

# [ inp ]

**ContinuePush**
( parameter width cardinal;  
  passive input inp : width bits  
)

# [ inp ]

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

# [ activate : [ inp_0 : out_0 | ... | inp_{\text{portCount}-1} : out_{\text{portCount}-1} ] ]
13.9. Breeze components ordered by name

**Encode**

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

# [ [ inp0 : out !* encode(outputWidth, inputCount, specification, 0) ] ... ]
```

**FalseVariable**

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

# [ write ?° [ v := write ; signal ] ] || # [ read0 !° v ] || ... || # [ read(readPortCount-1) !° v ]
```

**Fetch**

```
Fetch
( parameter width : cardinal;
  parameter outBroad : boolean;
  passive sync activate;
  active input inp : width bits;
  active output out : width bits )

# [ activate : out !* inp ?* inp ]
```

**Fork**

```
Fork
( parameter outputCount : cardinal;
  passive sync inp;
  array outputCount of active sync out )

# [ activate : [ out0 , ... , out(outputCount-1) ] ]
```
13.9. Breeze components ordered by name

**ForkPush**

\[
\text{ForkPush ( parameter width, outputCount : cardinal;}
\text{ passive input inp : width bits;}
\text{ array outputCount of active output out : width bits )}
\]

\[
\#[ \text{inp} \text{ ? } \text{out0} ! \text{inp} , \ldots , \text{out}\text{outputCount-1} ! \text{inp} ]
\]

**Halt**

\[
\text{Halt ( passive sync inp )}
\]

**HaltPush**

\[
\text{HaltPush ( parameter width cardinal;}
\text{ passive input inp : width bits )}
\]

\[
\text{stop}
\]

**InitVariable**

\[
\text{InitVariable ( parameter width, readPortCount : cardinal;}
\text{ parameter name : string;}
\text{ parameter initValue: width bits;}
\text{ passive input write : width bits;}
\text{ array readPortCount of output read : width bits )}
\]

\[
v := \text{initValue} ;
\]

\[
\#[ \text{write} ?_0 v := \text{write }] ||
\]

\[
\#[ \text{read}_0 ! v ] \ldots || \#[ \text{read}_{\text{readPortCount}-1} ! v ]
\]

**Loop**

\[
\text{Loop ( passive sync activate;}
\text{ active sync activateOut )}
\]

\[
\text{activate : } \#[ \text{activateOut }]
\]
13.9. Breeze components ordered by name

NullAdapt
( parameter inputWidth : cardinal;
   passive sync out;
   active input inp : inputWidth bits )

# [ inp : out ]

Passivator
( parameter count : cardinal;
   array count of passive sync inp )

# [ inp0 : ... : inp_{count-1} ]

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

# [ out0 !^ ... !^ outputCount-1 !^ inp ?^ inp ]

PassiveEagerFalseVariable
( parameter width : cardinal;
   parameter readPortCount : cardinal;
   parameter specification : string;
   passive sync activate;
   passive input write : width bits;
   active sync signal;
   array readPortCount of passive output read )

# [ activate: write ?^ [ signal : v := write ] ] || # [ read0 !^ v ] || ... || # [ read_{readPortCount-1} !^ v ]

PassiveSyncEagerFalseVariable
passive sync activate;
passive sync write;
active sync activateOut;

# [ activate: (activateOut ; write ) ]
13.9. Breeze components ordered by name

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

# [ activate : [ activateOut[0] ; ... ; activateOut[outputCount-1] ] ]

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

#[ out !° inp ?? slice(outputWidth, lowIndex, inp) ]

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

#[ inp ?? [ LSOut !° bitfield(0, LSOutputWidth-1, inp) ]
  MSOut !° bitfield(LSOutputWidth, inputWidth-1, inp) ] ]

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

#[ inp ?? [ out[0] !° bitfield(0, outputWidth-1, inp) ] ... ]
  [ out[outputCount-1] !° bitfield(outputWidth-outputWidth, inputWidth-1, inp) ] ]
13.9. Breeze components ordered by name

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

#[ inp0 : ... : inp_{inputCount-1} : out ]

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

#[ pout0 !° ... !° pout_{outputCount-1} !° inp ?° inp ]

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

# [ pout0 !° ... !° pout_{outputCount-1} !° inp ?° aout !° inp ]

UnaryFunc
( parameter outputWidth : cardinal;
parameter inputWidth : cardinal;
parameter op : UnaryOperator;
passive output out : outputWidth bits;
active input inp : inputWidth bits )

type UnaryOperator is enumeration (op symbol between brackets)
  Negate (~), Invert (-)
end

# [ out !° inp ?° op(outputWidth, inputIsSigned, op, inp) ]
13.9. Breeze components ordered by name

Variable

( parameter width, readPortCount : cardinal;
  parameter name : string;
  parameter specification : string;
  passive input write : width bits;
  array readPortCount of output read : width bits )

# [ write ° v := write ] ||
# [ read_0 ! v ] ... || # [ read_readPortCount-1 ° v ]

While

( passive sync activate;
  active input guard : 1 bits;
  active sync activateOut )

# [ activate : [ guard ° g : [ g -> activateOut ] ] ]

WireFork

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

# [ activate : [ out_0 ... , out_outputCount-1 ] ]
14 Reporting Bugs

We are currently using Bugzilla as a bug tracking system.

The current URL is http://bluu.cs.man.ac.uk. Should it change, you will always find an updated link on the main Balsa web page at http://www.cs.manchester.ac.uk/apt/projects/tools/balsa/.

![Figure 14.1: Balsa's bug tracking system](image)

You have found a bug? Here are 5 steps to getting it fixed!

**Check your Balsa version**

Before you begin, make sure you are using the latest version of Balsa. Balsa might have already been fixed!
Searching for Bugs

Check to see if the bug has already been reported. This step is very important! If you find that someone has filed your bug already, please go to the next step anyway, but instead of filing a new bug, vote for or comment on the one you’ve found. If you can’t find your bug in Bugzilla, go to the next step.

Create a Bugzilla account

You will need to create a Bugzilla account to be able to report bugs (and to vote or comment on them). Once you have an account you can report bugs on any product. Balsa is declared as an individual product in Bugzilla, and all of our bugs can be classified in various components under the Balsa product. If you have registered, proceed to the next step.

Note: We are currently experimenting with a guest account avoiding the need to create an account for reporting bugs. However, we might suppress it if people abuse it (it is possible for guests to edit other guests’ bug reports).

File the bug!

Now you are ready to file your bug. Balsa is divided into 11 components:

- balsa-c: for bugs in the Balsa compiler
- breeze-sim: for bugs in the Breeze simulator
- breeze-sim-ctrl: for bugs in the GUI for breeze-sim
- balsa-mgr: for bugs in Balsa Manager, the graphical frontend
- balsa-tech-amulet: for bugs in the AmuST technology backend
- balsa-tech-example: for bugs in the Example technology backend
- balsa-tech-xilinx: for bugs in the Xilinx technology backend
- Balsa - all technologies: for bugs involving more than one balsa technology
- Balsa - all: for bugs involving multiple components of the Balsa system
- balsa-sim-verilog: for bugs in the wrapper scripts for Verilog simulators
- Balsa manual: for reporting errors found this manual

Click on “Enter a new bug report”, select which component is affected, and write a summary (bug’s title) and description of your bug. Try to give enough information for us to be able to reproduce your bug. You can even attach files if needed.

What happens next?

Once your bug is filed, you will receive an email when it is updated at each stage in the bug life cycle. After the bug is considered fixed, you might want to ask us for the updated Balsa distribution.
References


