Balsa:
A Tutorial Guide.

Doug Edwards, Andrew Bardsley,
Lilian Janin & Will Toms

Version V3.4.2 – 2/1/05

Contents

1 Introduction

1.1. Introducing Balsa

This document describes version 3.4 of the Balsa system. This release adds significant changes in the capabilities of the simulation tools. File I/O, string handling and memory models are included.

Version 3.3 of the Balsa system was a major upgrade from previous versions although some of the extra functionality had been available in the various snapshots that have been downloadable from the Balsa website. Significant changes were introduced in all aspects of the system: language, simulation environment, back-end target technologies and not least in the documentation itself. Existing users of Balsa shoulf be aware of the changes which are summarised in the section “Changes in releases” on page 4. Whilst most existing Balsa descriptions should compile without problems, changes to the syntax of the `while` construct may cause existing descriptions not to compile. Most users should not be affected by the changes, since while loops, although available in earlier releases, were not described in the previous version of the manual.

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 a 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 teh 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 (~45) set of Handshake components. 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.

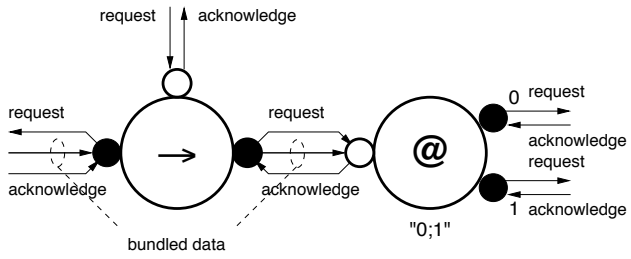

Figure 1.1: Two connected handshake components

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 isues 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 3.4 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 circcuit for a modulo-10 counter [see

"Removing auto-assignment" on page 42] is shown in Fig. 1.2. The corresponding gate level

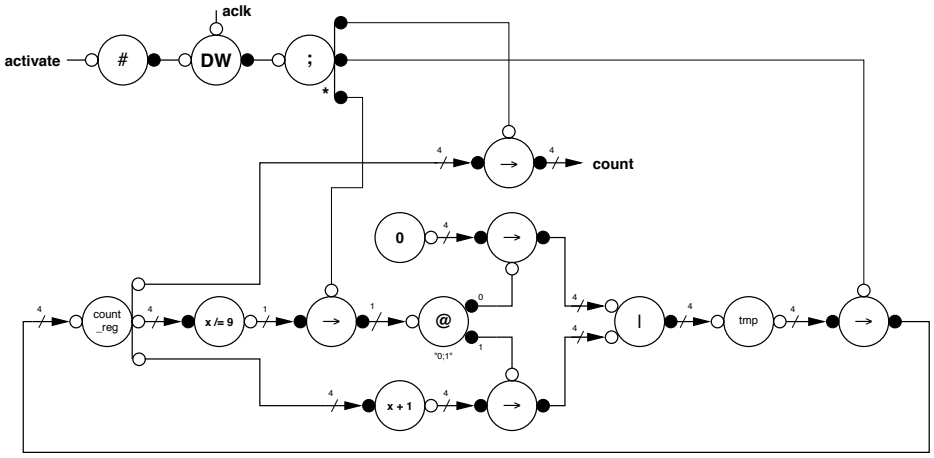

Figure 1.2: Handshake circuit of a modulo-10 counter

implementation is shown in Fig. 1.3.

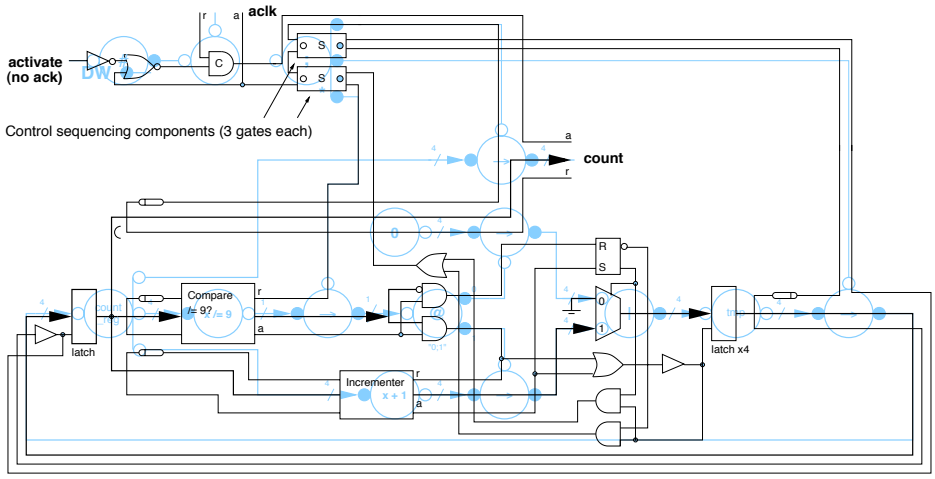

Figure 1.3: Gate level circuit of a modulo-10 counter

Note that the compilation function results in circuit fragments in which both input and output ports are active. Since passive ports can only connect to active ports and vice-versa, circuits constructed from compositions of compiled circuit fragments must have their interconnecting ports connected by *passivator* components. A passivator synchronises requests from input and output ports and 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 ouput 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
- *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 simualtor 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 manipulatng 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 facilites and user interface provided may be differ in detail from that described in this manual.

1.3. Changes in releases

Version 3.4

This release adds “builtin” types [see “Builtin 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 a text book [3] produced to promote the European Low-Power Initiative for Electronic System Design.

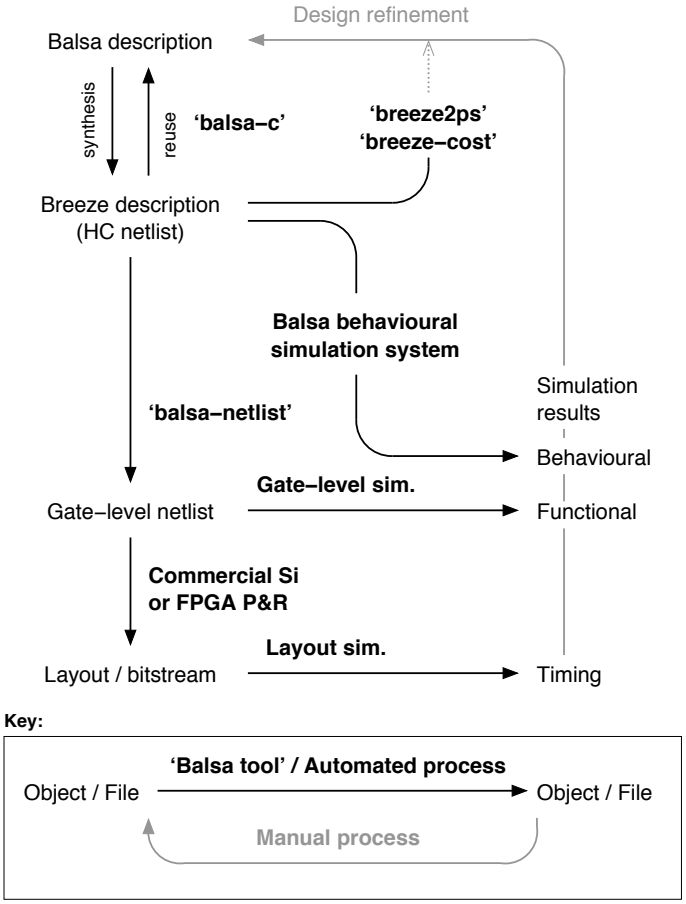

Figure 1.4: Design Flow

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 supress warning about multicast channels. The "-c warn-multicast" option to balsa-c now does nothing – it is enabled by default.
- implicants 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.

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 now 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 distibuted. 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.

If the Balsa system has been compiled from source, it should only be necessary to include the binary directory in the user's search path. Users using binary only distributions, should source the *apttools.sh* script located in the distribution package.

Previous users of the system should note that since LARD is no longer the preferred simulation route,

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
    ;             -- Sequence operator
    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.

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.[1]. 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
```

1. there is, of course, no predefined type `word`

The synthesised circuit.

The resulting handshake circuit is shown in Figure 2.1. Note that this is not actually taken from the

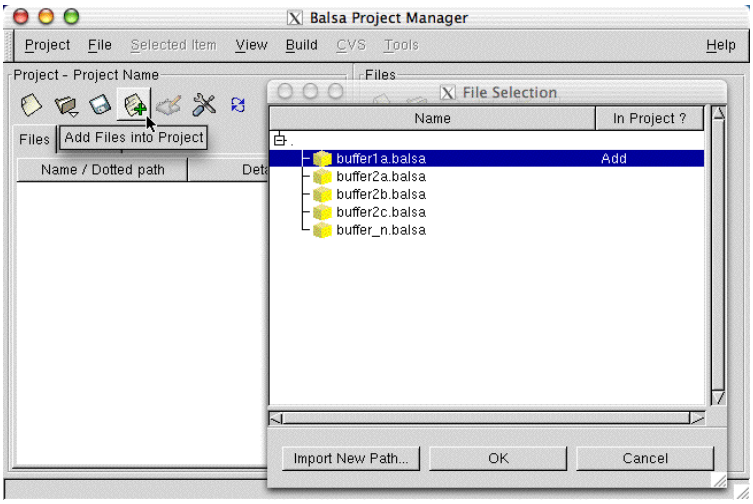

Figure 2.1: Handshake Circuit for a single place buffer

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 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 “→” 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.balsa

```
-- 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, `x1` and `x2`. The contents of the variable `x1` are caused to be transferred to the variable `x2` by means of the assignment operator "`:=`". 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.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 before-hand). The `for` construct together with compile-time constants may be used.

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 7]

Creating a new project

The command `balsa-mgr` invokes the project manager. Select "Project ⇒ New" from the pull-

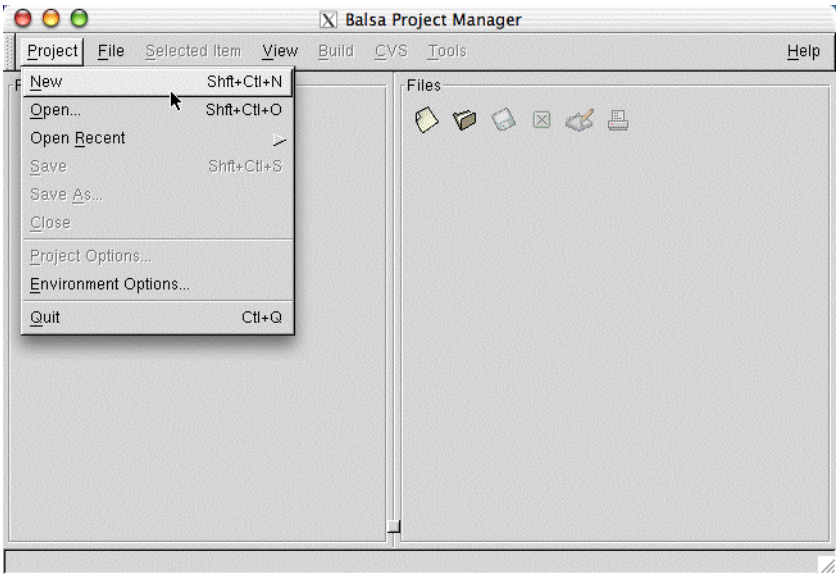

Figure 2.2: Creating a new project.

down menu as shown in Figure 2.2 to display the dialogue box shown in Figure 2.3. A default name

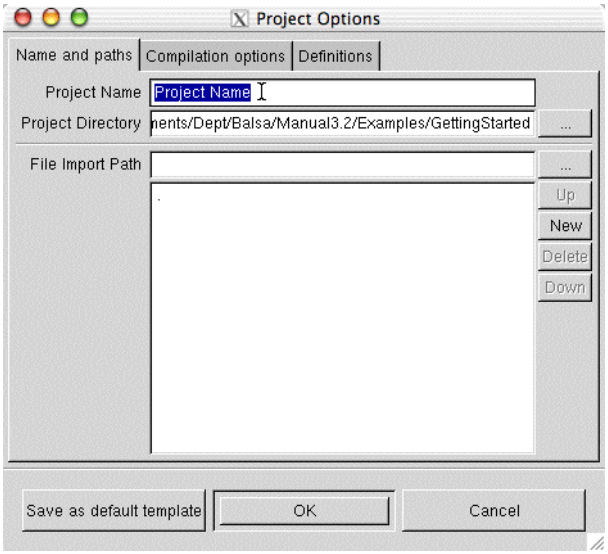

Figure 2.3: The New Project Dialogue Box.

for the project is generated; this may be over-ridden to something more meaningful. The "Project 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

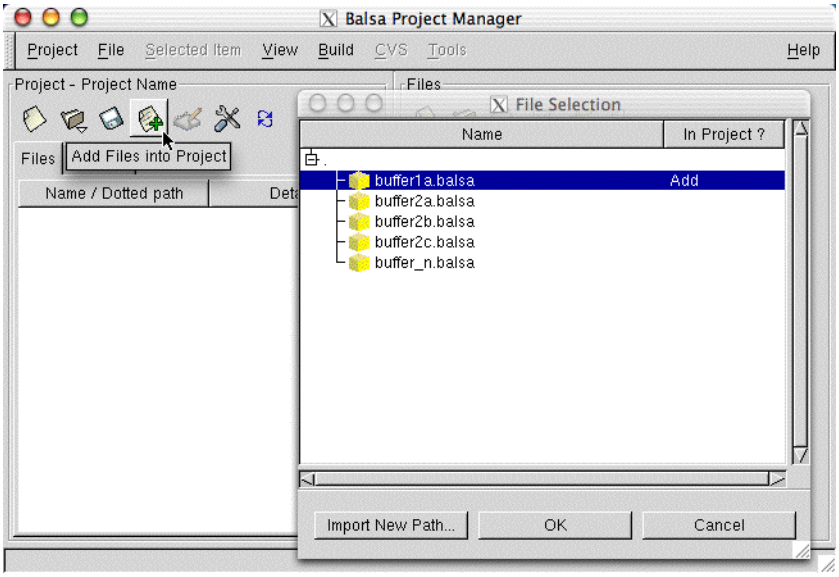

Figure 2.4: Adding Files to the Project Manager.

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 uder any filename is the result of a simplistic parsing of the source file and may *occassionally* 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.

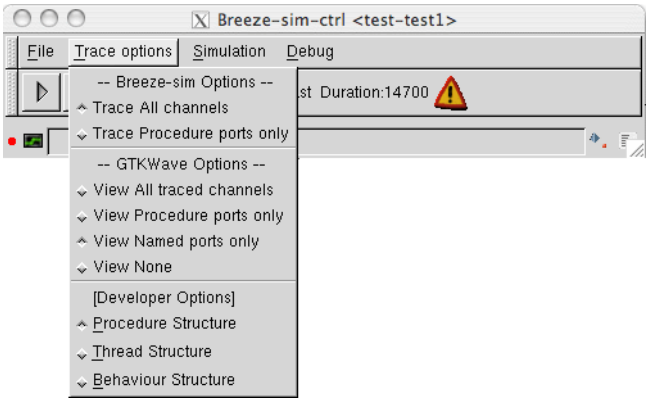

Figure 2.5: Displaying a file in the Edit Pane.

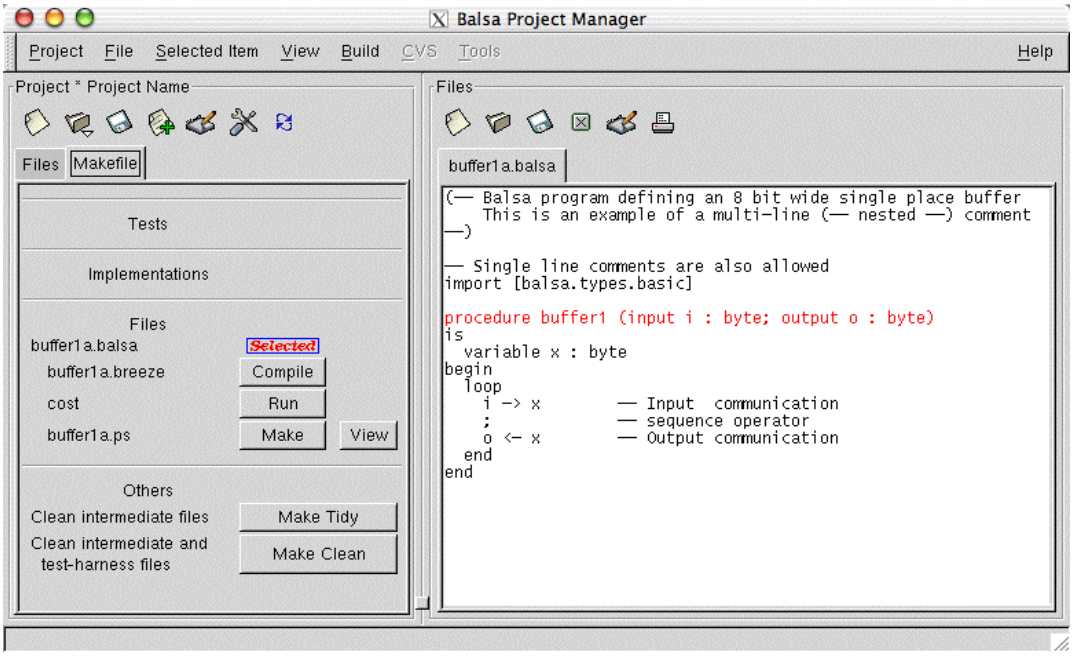

Figure 2.6: The Makefile Pane.

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.

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 displayed in red 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 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.

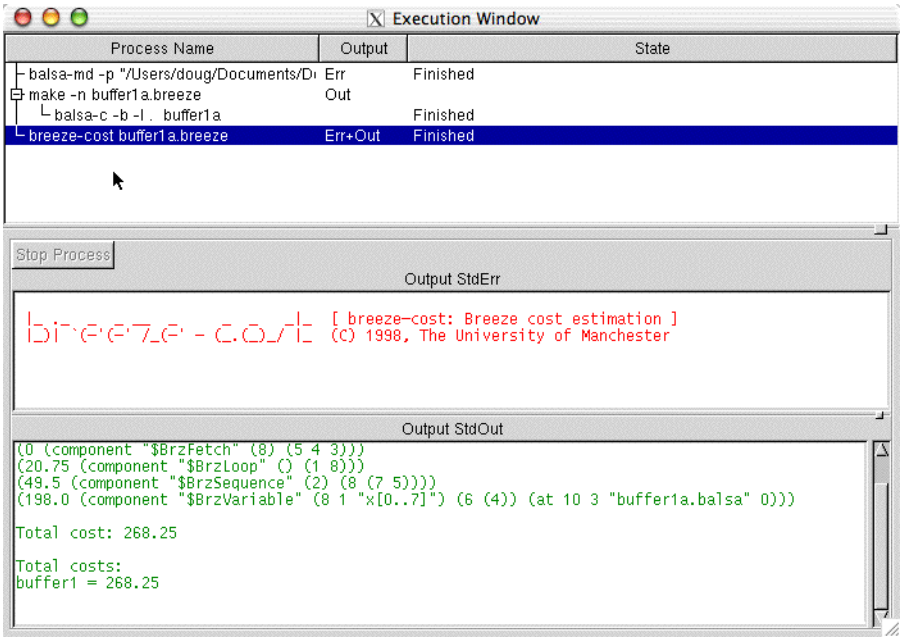

Figure 2.7: Execution window, showing the cost of buffer1a

Saving Window Contents

The ouputs 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.

Flattened vs non-flattened view

When a circuit is composed hierarchically, there is a choice of whether the resulting circuit is generated in a hierarchical or flattened manner. Balsa-mgr allows either representation.

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. Click on the Makefile tab in the left-hand pane of the balsa-mgr window. A set of compilation actions for *buffer2c* (see Figure 2.8) has been added to those for *buffer1a*.
5. View the handshake circuit by clicking on the **View** button for buffer2c.ps. A hierarchical view of the composed circuit is shown in Figure 2.9.
6. Determine the cost of the circuit: a cost of 558.75[1] should be reported. The cost is reported for the total of the hierarchical circuit and for the individual components.

Figure 2.9 shows the extra components required to compose the two instances of the single place buffer.. The Fork component activates the two single place buffers in parallel. The Passivator component. The Passivator component connects two active ports: the output port of the first buffer and the input port of the second buffer.

The description of buffer2c can be flattened during compilation by passing the appropriate flag through to the command line of the compilation tool. Balsa-mgr allows this to be done without detailed knowledge of the command line tools.

1. Click on the “Project Options” icon or select “Project ⇒ Options” from the pull-down menu.

1. The exact cost may vary between different releases of Balsa.

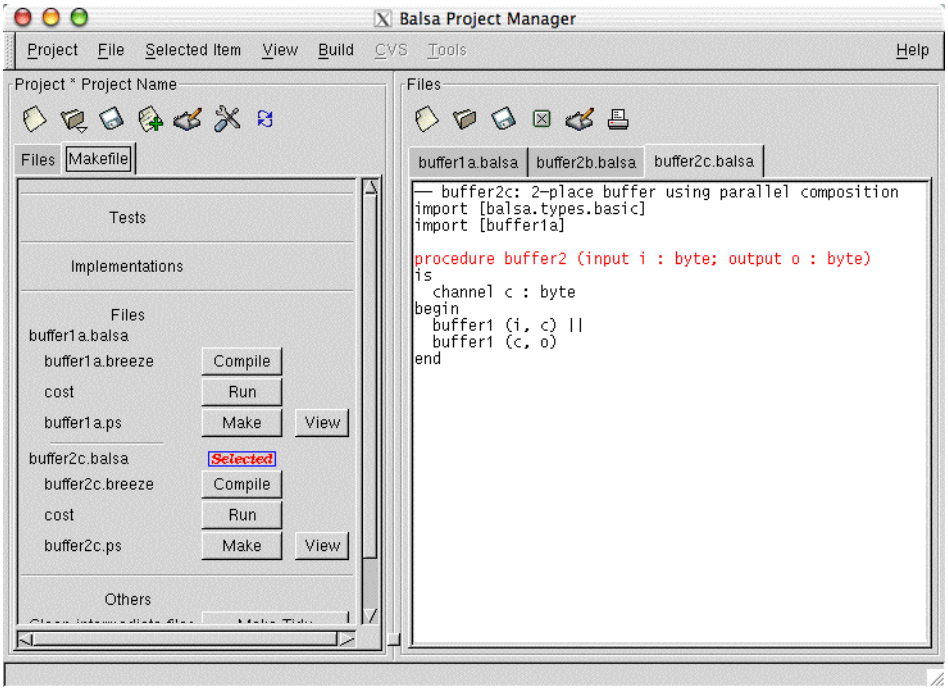

Figure 2.8: Buffer2c actions added to Makefile pane

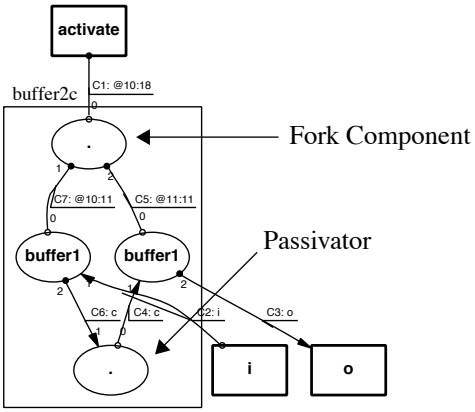

Figure 2.9: Hierarchical view of buffer2c handshake circuit graph.

2. Click on the “Compilation Option” tab on the project dialogue window that appears (see Figure 2.3)
3. Check the check-box marked Flattened Compilation in the compilation option pane (see Figure 2.10). The exact layout of this window and the set of options available may be different from that shown because of developments in the Balsa system.
4. Rerun the view of the of the handshake circuit graph. A flattened view should be obtained as shown in Figure 2.11.
5. Restore the compilation settings to the default “Hierarchical Compilation”

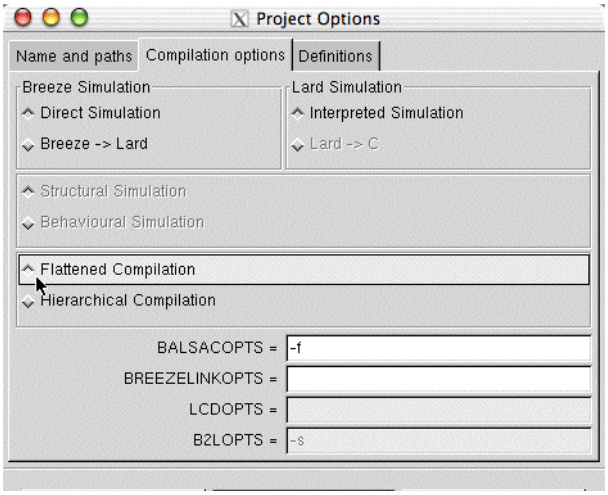

Figure 2.10: The Compilation Options pane

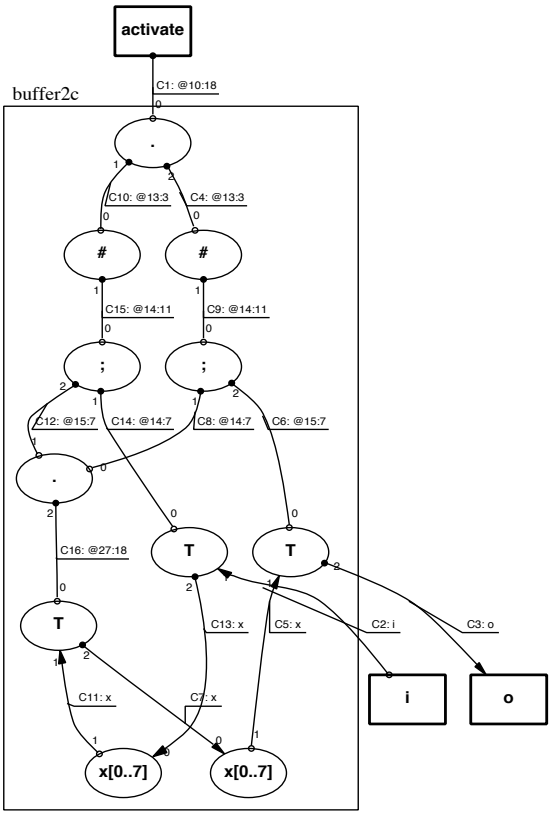

Figure 2.11: Flattened view of buffer2c

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 auser 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 can not 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.

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. Make sure that buffer2 is selected in the Files pane.
2. Pick “Select Item ⇒ Add Test Fixture” from the pull-down menu or right-click in the left-hand “File” pane. A window for creating a text fixture is spawned [Figure 2.12]

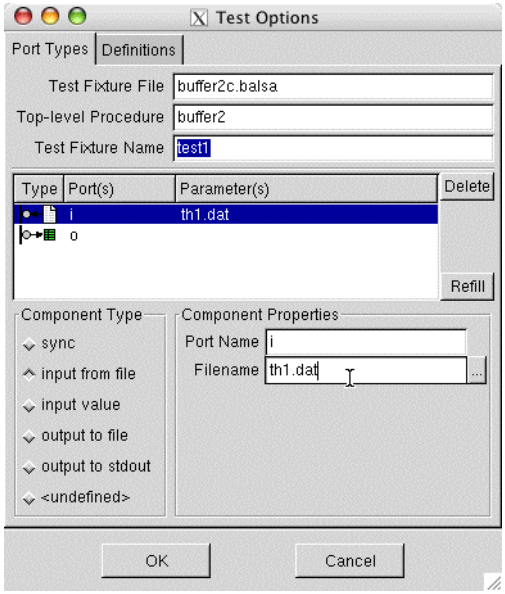

Figure 2.12: The test fixtures pane

3. Change the name of the test fixture from the default (test1) to something more meaningful, e.g. buff2c
4. Select the Port Name “`i`”
5. Change the active radio button in the “Component Type” pane from “input value” to “input from file”
6. 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 items is allowed per line and complex 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.

7. Dismiss the window by clicking OK.

Text-only simulation

In order to run the test, click on the Makefile tab. The Makefile view shown in Figure 2.15 now

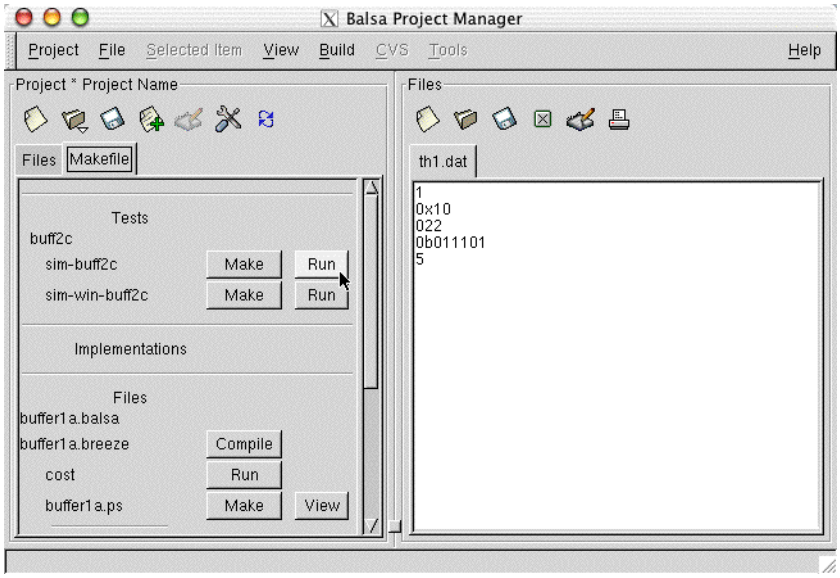

Figure 2.13: Test Harness added to Makefile pane

shows two actions added under the Tests" subpane. Clicking on the **Run** button for sim-buff2c generates the following output in the execution window. 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 the simulator has a very simplistic timing model, so these values should be treated with caution.

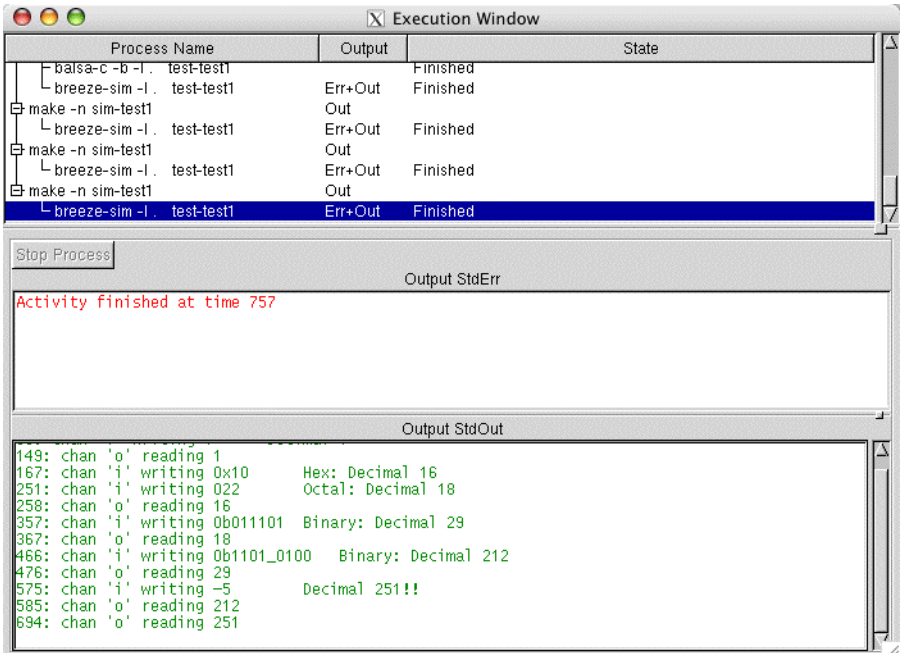

Figure 2.14: The ouput from a text-only simulation

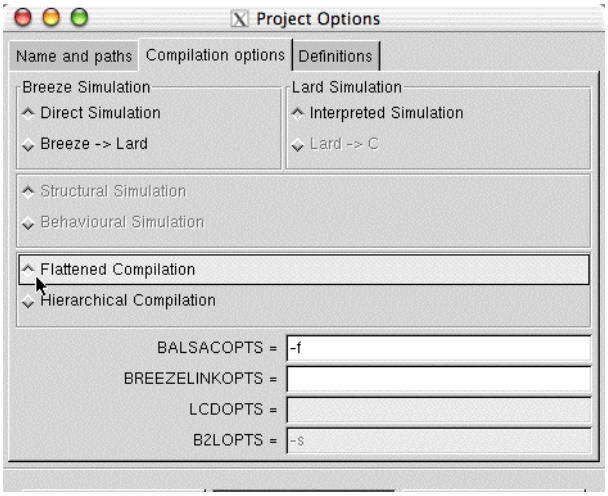

Figure 2.15: Test Harness added to Makefile pane

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 2.16) which controls the simulations and animations.

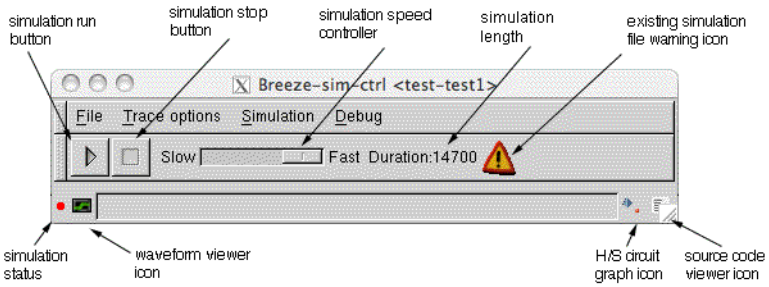

Figure 2.16: The Simulator Controller

The controller allows:

- a new simulation file for a design to be produced and displayed in a waveform viewer (GTKWave).
- 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 2.17). The choice is between tracing all the channels or tracing

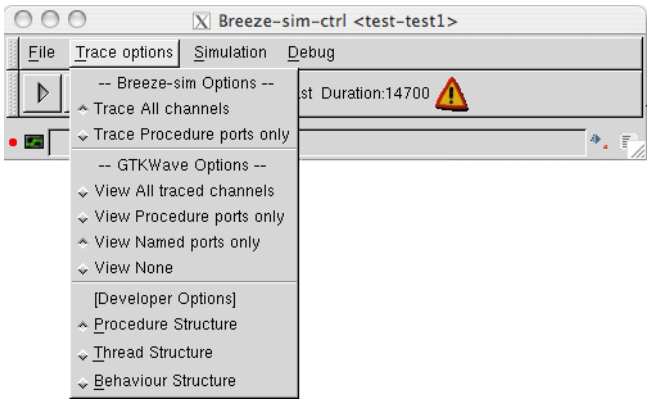

Figure 2.17: Trace options menu

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.

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 2.18. Request signals are shown in red and

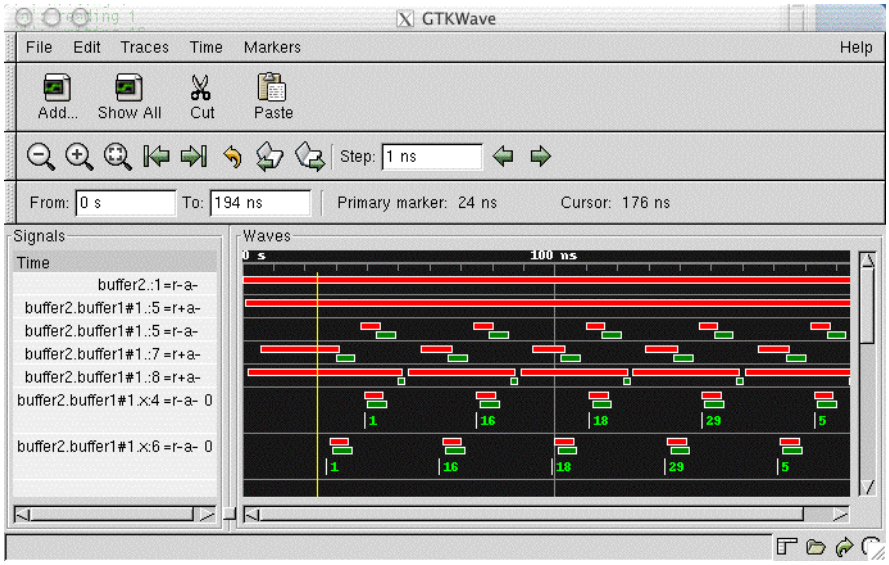

Figure 2.18: Channel viewer window.

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 2.17). 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).

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 tool bars 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 2.19. 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 hanshake 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.

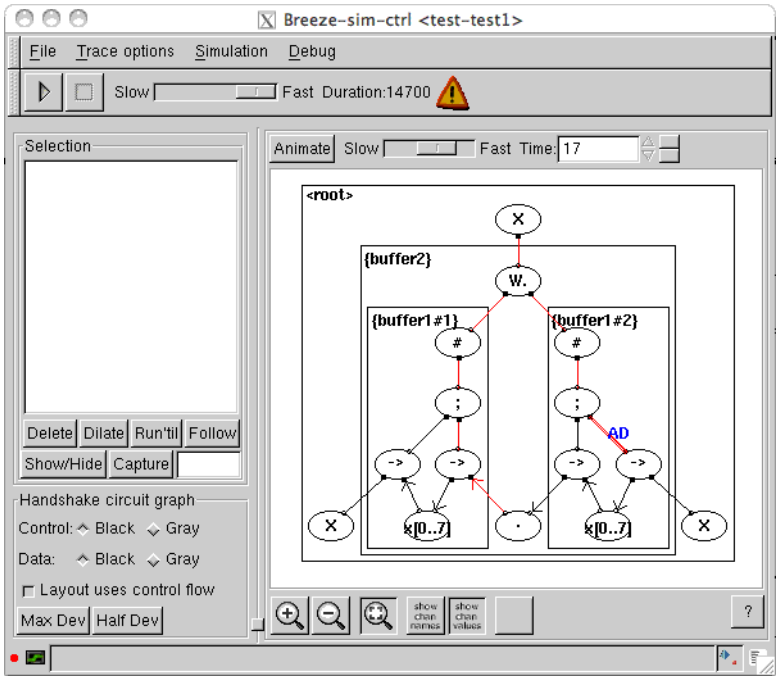

Figure 2.19: Channel tree and handshake circuit graph revealed.

- 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).
- 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.

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).

Debugging a deadlock

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.

Source Code Viewer

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.

2.7. Compilation and Simulation Options

These options are reached by clicking on the "Project Options" icon or selecting "Project ⇒ Options" from the pull-down menu.s and selecting the compilation pane.

Flattened vs hierarchical compilation

As discussed in earlier (see "Flattened vs non-flattened view" on page 16), this option allows a choice of flattening the design during the compilation process or maintaining the hierarchy. A flattened design allows a slightly greater degree of peephole optimisation to take place, but at the expense of exposing the internal channels and losing the hierarchy at later stages of the design process. The hierarchical view is more appropriate for most users and this is the default setting in the balsa-mgr.

Direct Simulation vs Breeze ⇒ Lard

Users are strongly recommended to use only the direct simulation route. This uses the new breeze simulation engine and GTKWave channel viewer environment which has been described earlier. Checking the Breeze ⇒ Lard box will enable the old deprecated LARD simulation environment.

Lard simulation options

If the Breeze ⇒ Lard route is (against all advice) chosen, users have two choices:

Interpreted Simulation: This is the default option and works with the existing LARD channel viewer. It is however very slow.

Lard ⇒ C: This option considerably speeds simulation times (by up to two orders of magnitude compared to an interpretedsimulation), but can not be used with the LARD channel viewer, nor can it currently be used under MacOS X. The speed is still significantly slower than the new Breeze simulation engine.

Structural vs behavioural simulation

This option only applies to users who are still persisting with the deprecated LARD simulation route. It will be removed in subsequent releases of the Balsa system.

Users may choose whether to use behavioural or structural simulation. Behavioural simulation (the default) is much faster. However the timings are not necessarily representative of relative channel activity. Structural simulation allows the interaction between the handshake components to be explored more accurately and is useful as a teaching aid to understanding the behaviour of fine grained asynchronous circuits. However, as the overall timings are still very approximate, users only interested in functional simulation are recommended to use the behavioural option.

Users may wish to explore the significance of the various options by simulating *buffer2c* with various combinations of switches.

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 Table 11.1 on page 127.

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 $n \in [1, INT\_MAX]$; 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:

```
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:

```
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

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:

```
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:

```
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.:

```
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:

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

As with enumerations, record types can be padded:

```
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.

```
type Flags is record
```

```
  carry, overflow, zero, negative : bit
over 4 bits
```

Array types

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

```
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 of 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:

```
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:

```
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`:

```
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, sting constants are not allowed. Valid examples are:

```
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:

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

The two examples above may also be written:

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

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

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 11].

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 he 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.

Casts

If the variable x was declared as 32 bits, the correct form of the assignment above is:

```
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

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
```

Bit ordering and padding in arrays

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-signifcant bit is the rightmost bit element of the array – note this is contary 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}  as 8 bits);
print "y is: ", y , " ", #y;
```

gives:

```
y 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 Table 11.2. on page 127.

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

Table 3.1: Balsa Commands

command	Notes
arbitrate … end	arbitrated choice operator
print <args>	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
<block>	allows inclusion of local definitions around a command and the overriding of the precedence of command composition. See Table 11.1 on page 127.

Table 3.1: Balsa Commands

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.

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 7.

```
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 11]. An illustration of the inappropriate use of the `for` command is

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

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 imp1 is
begin
  for ; i in 1 .. 15 then
    case i of
      0bx1x then print “don’t care guard: “, i
    else
      print “covering case: “, i
```

```
        end
      end
    end
```

3.4. Binary/Unary Operators

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

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

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 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 Table 11.1 on page 127.

⟨file⟩	::=	(**import [** ⟨dotted-path⟩ **]**)* ⟨outer-declarations⟩
⟨dotted-path⟩	::=	⟨identifier⟩ (**.** ⟨identifier⟩)*
⟨outer-declarations⟩	::=	(⟨outer-declaration⟩)*
⟨outer-declaration⟩	::=	**type** ⟨identifier⟩ **is** ⟨type-declaration⟩
	|	**constant** ⟨identifier⟩ **=** ⟨expression⟩ (**:** ⟨type⟩)?
	|	**procedure** ⟨identifier⟩ **is** ⟨identifier⟩ (**(** ⟨procedure-formals⟩ **)**)?
	|	**procedure** ⟨identifier⟩ (**(** ⟨procedure-formals⟩ **)**)? **is** (**local**)? ⟨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

⟨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

complete syntax for the Balsa language is given in Table 11.1 on page 127). 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.

```
constant debug = true

if debug then
```

```
  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 `p1` are provided: the actual definition used depends on the value of the constant debug.

Conditional ports

Port declarations and variables may also may be conditional. The next example is a the 2-place buffer described in “Parallel composition and module reuse” on page 10. 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.

```
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
) is
  if not debug then channel c : byte end
begin
  buf1(i,c) || buf1(c,o)
end -- procedure buf2
```

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.

```
procedure write_zero( output o : byte) is
begin o <- 0 end

variable v : byte
write_zero( -> v)
```

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

```
c1 <- expr1 ||
c2 <- expr2 ||
c3 -> var1  ||
proc1(c1, c2, c3)
```

can be replaced by

```
proc1( <- expr1, <- expr2, -> var )
```

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:

```
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:

```
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:

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

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

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 acknowledge completing the handshake and the counter state is updated.

Modulo-16 counter

```
-- 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.

```
-- 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
```

```
    sync aclk;
    tmp := (count_reg + 1 as nibble)||
    count <- count_reg;
    count_reg := tmp
  end
end
```

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.

```
-- 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 -- loop
end -- begin
```

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. The 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*.

```
-- 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
```

```
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.

```
variable x: 4 bits
begin
  loop while x <= 9 then
    print "value of x is: ", x;
    x := (x + 1 as 4 bits)
  end
end
```

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

```
variable x: 4 bits
begin
  loop while x <= 15 then -- never terminates
    print "value of x is: ", x;
    x := (x + 1 as 4 bits)
  end
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:

```
variable x: 4 bits
begin
  loop
    print "value of x is: ", x
  while x < 15 then continue
  also x := (x + 1 as 4 bits)
  end
end
```

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

```
variable x: 4 bits
begin
  loop
    print "value of x is: ", x;
    x := (x + 1 as 4 bits)
  while x /= 0
  end
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.

```
-- 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].

```
-- 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

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.

```
-- 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

A variable width buffer definition

The example *pbuffer1below* 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)
```

```
-- now use the definition
procedure test1(input a : byte; output b : byte) is
begin
  Buffer8(a,b)
end -- procedure test1

-- alternatively
procedure test2(input a : byte; output b : byte) is
begin
  Buffer(byte, a,b)
end -- procedure test2
```

The definition of the single place buffer given previously [see "A single-place buffer" on page 7] is modified by the addition of the parameter declaration which defines `X` to be of type `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. `Buffer8` defines a byte wide buffer that can be instantiated as required as shown, for example, in procedure `test1`. Alternatively, a concrete realisation of the parameterised procedure can be used directly as shown in procedure `test2`. Note that a test harness can be attached directly to the definition `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.

```
-- pbuffer2.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
) is

  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
end

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

Buffer is the single place parameterised width buffer of the previous example and this is reused by means of the library statement `import[pbuffer1]`. In this code, *BufferN* is defined which in a very

similar manner to the example described in “Placing multiple structures” on page 11, 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.

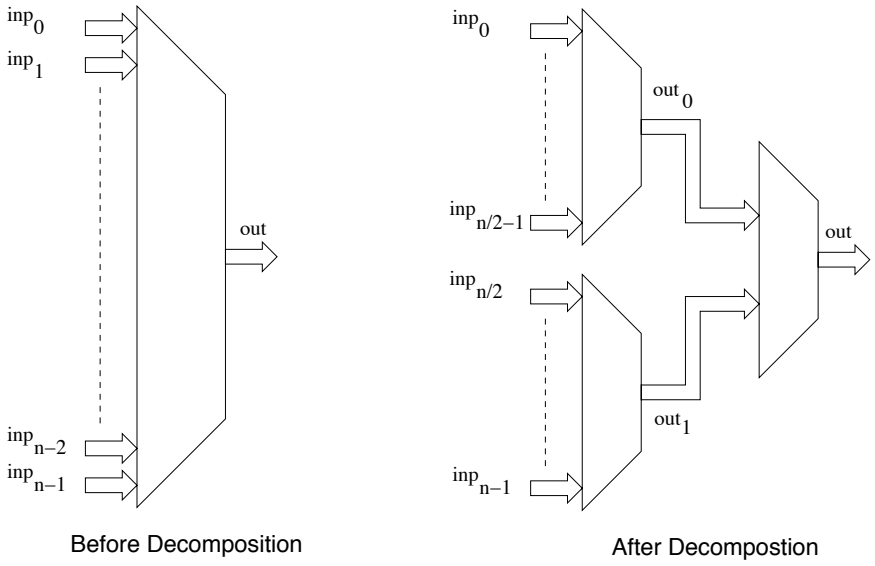

Figure 4.1: Decompostion of an n-way Multiplexer

```
--- Pmux1.balsa: A recursive parameterised MUX definition
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
```

```
  |  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

-- Here is a 5-way multiplexer
procedure PMux5Byte is PMux(byte, 5)
```

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, *out0* and *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.

```
-- 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
  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 repetiton 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
```

5 Handshake Enclosure

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.

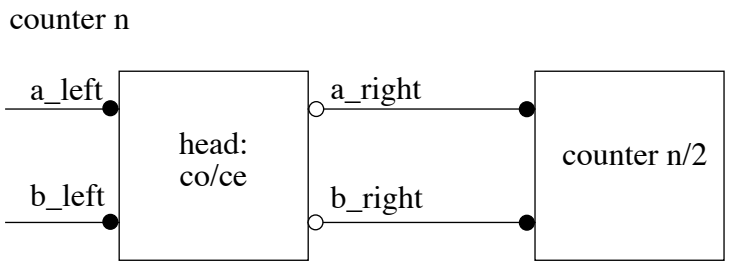

Figure 5.1: Counter Decomposition

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

Note that ports *a_left* and *b_left* are active ports whereas *a_right* and *b_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_right* and *b_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:

```
-- 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
```

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

A systolic modulo-11 counter

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

```
11 = 1 + 2*5 = 1 + 2*(1 + 2*2) = 1 + 2*(1 + 2*(2*1))
```

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

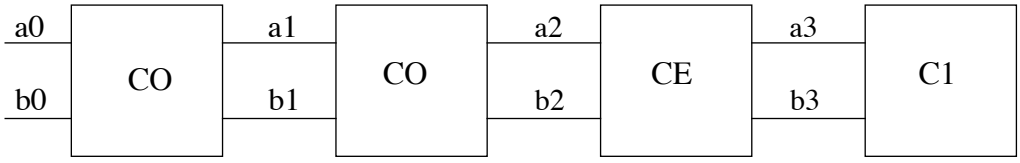

Figure 5.2: Modulo-11 Systolic Counter

```
procedure count11(sync a0, b0 ) is
sync  a1, b1, a2, b2, a3, b3
begin
  co(a0, a1, b0, b1)||
  co(a1, a2, b1, b2) ||
  ce(a2, a3, b2, b3) ||
  c1(a3, b3)
end
```

The behaviour of the circuit is shown in the trace of Figure 5.3[1]. 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.

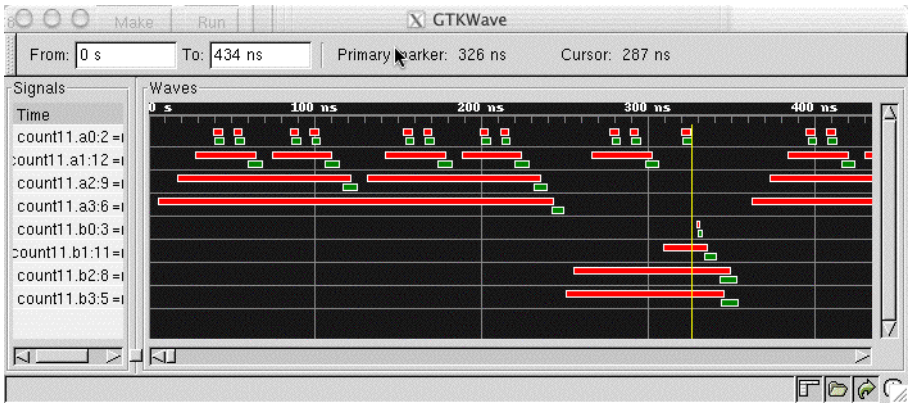

Figure 5.3: Behaviour of a Modulo-11 Systolic Counter

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 he 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.

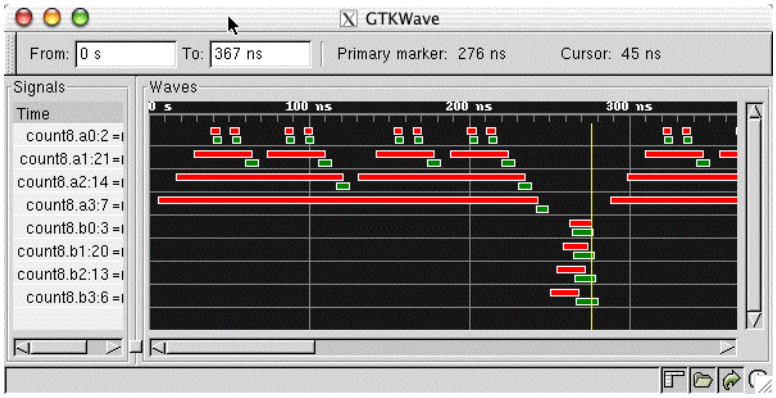

Figure 5.4: Behaviour of a Modulo-8 Systolic Counter

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:

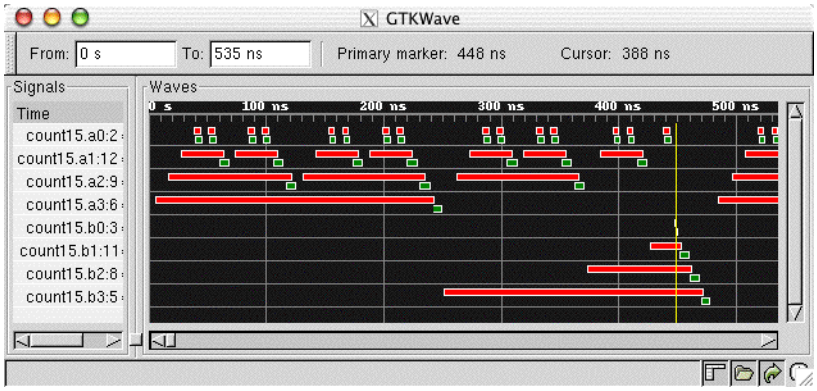

Figure 5.5: Behaviour of a Modulo-15 Systolic Counter

```
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
```

```
      sync b_left
    end
  end
```

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 ichange in the pattern of behaviour. Whetehr or not this translates to a change in performance depends on the relative speeds of the handshake components in the synthesised circuits..

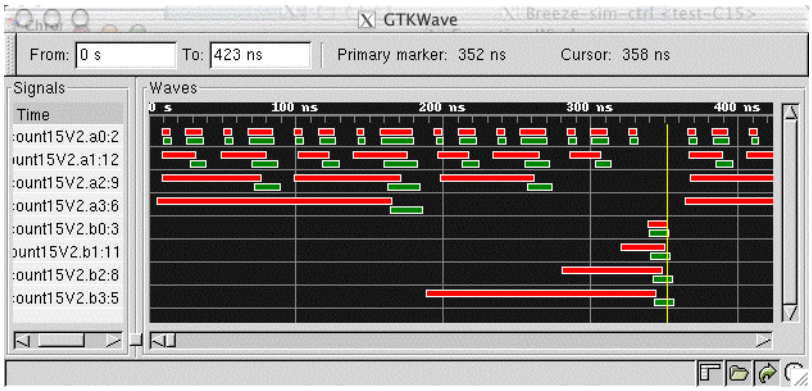

Figure 5.6: Behaviour of a Modulo-15 Systolic Counter

A decoupled all even cell

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:

```
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.

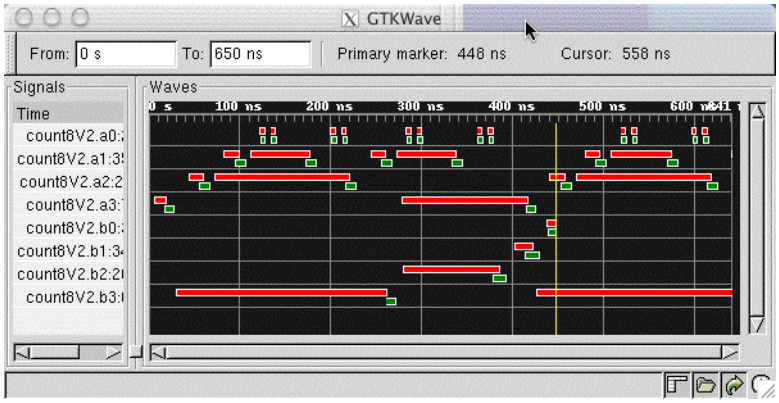

Figure 5.7: Behaviour of Non-Enclosed Modulo-8 counter

Parameterised version

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.

```
-- 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
begin
  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  ||   countN(isDecoupled, n/2, a_int, b_int)
  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

- 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:

```
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:

```
<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.

```
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.

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

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

```
) 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 assigment 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
```

6 Balsa Design Examples

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 arrray 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

```
-- 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
```

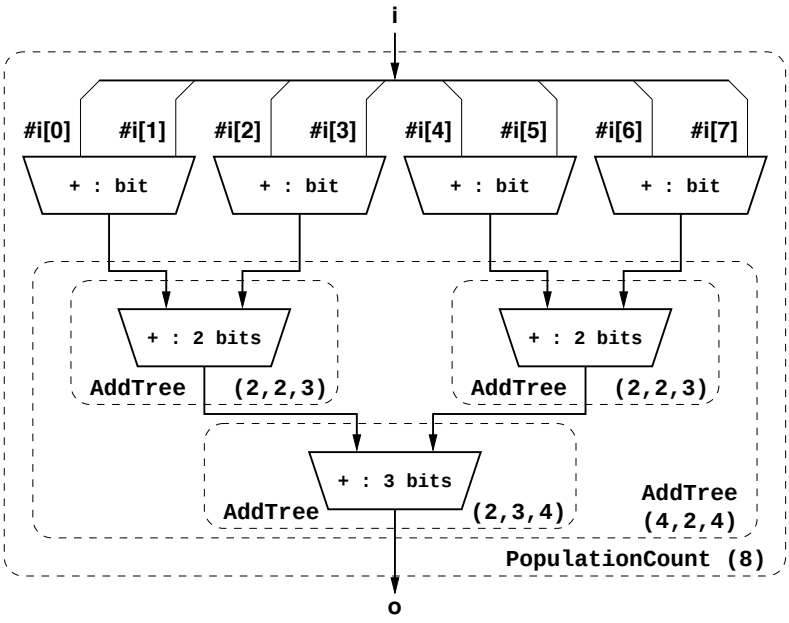

Figure 6.1: Structure of a bit-population counter

```
      o <- (i[0] + i[1] 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
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 <- i
      |  n = 2 then
        o <- (#i[0] + #i[1])  -- add bits 0 and 1
      else
```

```
        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)
```

Commentary on the code

Procedures `AddTree` and `PopulationCount` are parameterised. `PopulationCount` can used to count the number of bits set in any sized word. `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 `select` allow the contents of the input `i` to be referred to several times in the body of the `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 `for` loop.

```
for || c in 0..pairCount-1 then
  addedPairs[c] <- (#i[c*2] + #i[(c*2)+1])
```

It might be thought that a serial `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 `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 `AddTree` is awaiting all pairs to become valid: result deadlock!

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 *rorBody* which recursively creates sub-shifters capable of optionally rotating 1, 2, 4, 8 etc bits. The structure of the shifter is shown in

```
import [balsa.types.basic]

procedure ror (
  parameter X : type;
  input d : sizeof X bits;
  input i : X;
  output o : X
) is
begin
  loop
```

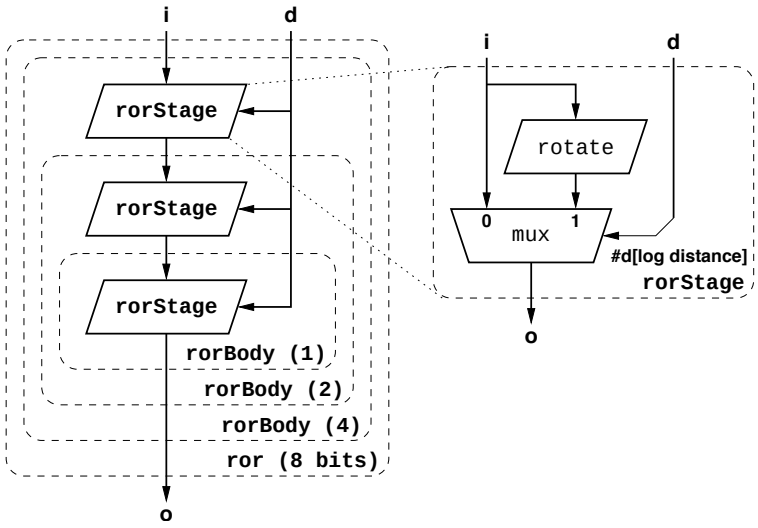

Figure 6.2:

```
select d then
  local
    constant typeWidth = sizeof X

    procedure rorBody (
      parameter distance : cardinal;
      input i : X;
    ) is
        output o : X
   ) is
       ) is
    local
      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
```

```
  end -- loop
end -- begin

procedure ror32 is ror (32 bits)
```

Testing the shifter

Another small Balsa test routine for exercising the shifter:

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

--test ror32
procedure test_ror32(output o : 32 bits)
is
  variable i : 5 bits
  channel shiftchan : 32 bits
  channel distchan : 5 bits
begin
  begin
    i:= 1;
    loop
      shiftchan <- 7 || distchan <- i;
      i:= (i+1 as 5 bits)
    while i < 31 end
  end || ror32(distchan, shiftchan, o)
end -- begin
```

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

-- ArbTree: a tree arbcall which outputs a channel number
--  prepended onto the input channel's data. (invokes itself
--  recursively to make the tree)
procedure ArbTree (
  parameter inputCount : cardinal;
  parameter depth : cardinal; -- bits to carry from inputs
  array inputCount of input i : depth bits;
  output o : (log inputCount) + depth bits
) is
begin
  case inputCount of
    0, 1 then print error, "can't build an ArbTree with fewer than 2 inputs"
```

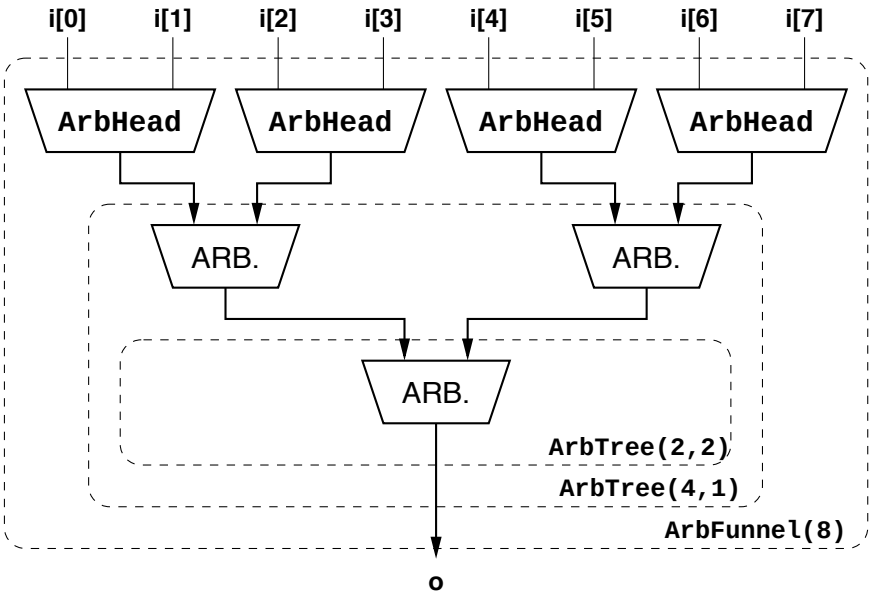

Figure 6.3: 8-input arbiter

```
  |  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
```

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

A description allowing arbitrary sized arbiters can be found in (*FurtherEx/ArbTree/arbgen.balsa*).

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.

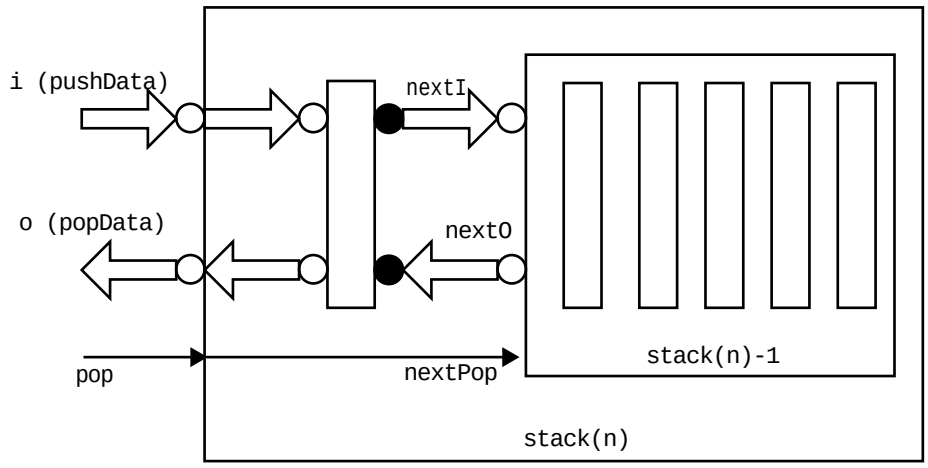

Figure 6.4: A Recurslively Defined Stack

Operations on the stack consist of either pushing daa 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 the ability to choose between ouput 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 propogated down the stack.

```
import [balsa.types.basic]

-- The stack description
procedure stack (
  parameter depth : cardinal ;
  input i : byte ;
  output o : byte ;
```

```
  sync pop
) is
  variable x : byte
begin
  if depth = 1 then
    loop
      select i then
        x := i
      | pop then
        o <- x
      end -- select i
    end -- loop
  else local
      channel nextI, nextO : byte
      sync nextPop
    begin
      stack (depth - 1, nextI, nextO, nextPop) ||
      loop
        select i then
          nextI <- x ;
          x := i
        | pop then
          o <- x ;
          sync nextPop || nextO -> x
        end -- select i
      end -- loop
    end -- local
  end -- if depth = 1
end -- procedure stack

procedure stack8 is stack(8)
```

Commentary on the code

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`).

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.

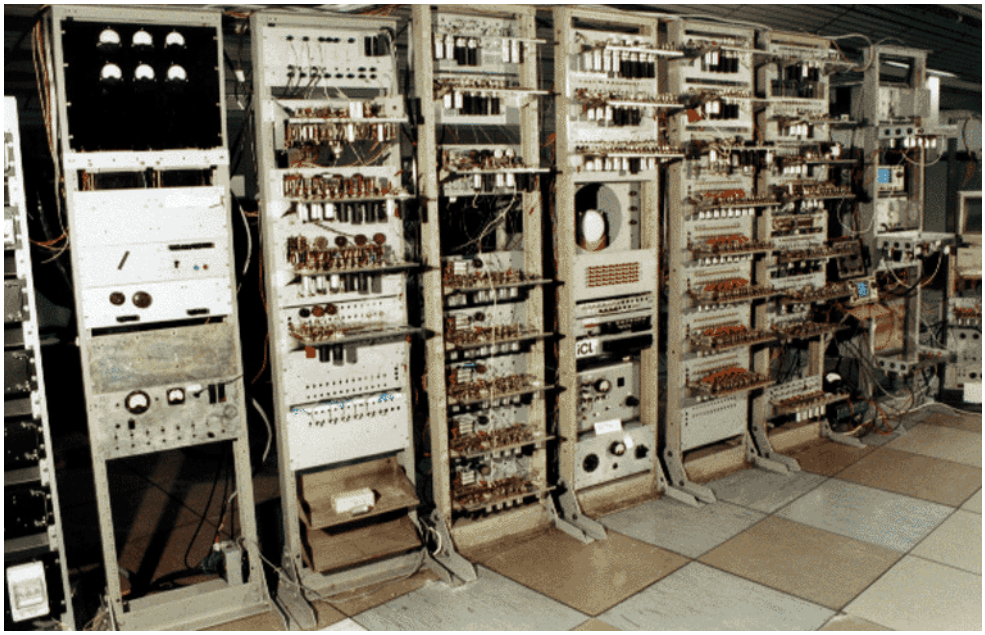

Figure 6.5: A rebuild of the orginal SSEM

The original machine had only 7 instructions:

```
JMP      ; PC := M[Addr]                    indirect jump
JRP      ; PC := PC + M[Addr]               relative jump
LDN      ; ACC := -M[Addr]                  load negative
STO      ; M[Addr] := ACC                   store result
SUB      ; ACC := ACC - M[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:

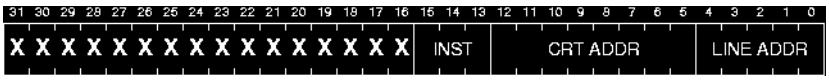

Figure 6.6: SSEM instruction format

The CRT address referes 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.

SSEM types

```
-- Basic types
type word is 32 bits
```

```
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
```

Channel and Variable Declarations

```
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
```

Useful functions and shared procedures

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

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

-- Memory operations, shared procedures
  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
```

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

Decode and excute procedure

```
--missing instrcution 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.

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.

Builtin Functions

In order to manipulate builtin 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 builtin function must have a declaration in Balsa as well as a definition in C. In Balsa, a typical builtin function declaration looks like this:

```
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:

```
    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 builtin 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 builtin and simple bitwise types values can be passed into builtin 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.

Builtin functions can also have parameters in the same way as parameterised procedures to allow the typing of their ports to be varied between instanced 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 builtin 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:

```
    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 builtin 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 builtin functions rather than specialised simulation handshake components. A statement such as:

```
print “Hello”, v
```

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

```
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;
```

```
  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

-- FileWrite : write a string to a file, returns the file object
function FileWrite (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)
```

```
        end
      end
    end
  end
end

procedure B1632 is BalsaMemory ({16, 32})
```

sim.portio

```
-- 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
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
```

```
        i -> then
          BalsaWriteLogLine (portName, "reading",
       <- NumberToString (X, i, radix, underscoreSpacing, showLeadingZeroes))
        end
  end
end

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

-- BalsaPrintInputPortFromValue : supply the given value to the port “o” each
-- time an input happens on that port
procedure BalsaInputPortFromValue (
  parameter X : type;
  parameter portName : String;
  input value : X;
  output o : X
) is
begin
  value -> then
    loop
      o <- value;
      BalsaWriteLogLine (portName, "writing", <- ToString (X, value))
    end
  end
end

-- BalsaInputPortFromFile : source values for port o from the given file
procedure BalsaInputPortFromFile (
  parameter X : type;
  parameter portName : String;
  input file : File;
  output o : X
) is
  variable line : String
  variable value : X
begin
  file -> then
    loop while not FileEOF (file) then
      line := FileReadLine (file);
      value := FromString (X, line, line);
```

```
      o <- value;
      if StringLength (line) /= 0 then
        BalsaWriteLogLine (portName, "comment", <- line)
      end;
      BalsaWriteLogLine (portName, "writing", <- ToString (X, value))
    end
  end
end
```

sim.sim

```
-- BalsaSimulationTime : get the current simulation time as a string.
--   This function must be provided genuinely builtin by any simulation system
function BalsaSimulationTime is builtin : String

-- BalsaGetCommandLineArg : get the value of a keyed command line argument
-- from the simulator based on the key
--  This function must be provided genuinely builtin by any simulation system
function BalsaGetCommandLineArg (key : String) is builtin : String
```

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:

```
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):

```
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

BalsaSim_BuiltinLibrary_<libraryname> (where <libraryname> is the last component of the dotted path to that libraries Balsa/Breeze file) to call this function and to perform any other initialisation necessary for that library. A macro, BALSA_SIM_REGISTER_BUILTIN_LIB, is provided to insert the head of this initialisation function. The complete C file for the HelloWorld example is:

```
#include <stdio.h>
#include <balsasim/builtin.h>
static void HelloWorld (BuiltinFunction *function,
   BuiltinFunctionInstanceData *instance)
{
   fprintf (stderr, "Hello, Balsa user\n");
   instance->result->words[0] = 1; /* Ignore this for now */ }
   BALSA_SIM_REGISTER_BUILTIN_LIB (hello)
{
BalsaSim_RegisterBuiltinFunction ("HelloWorld", 0, 0,
   HelloWorld, 1, NULL, 0);
}
```

The header file *balsasim/builtin.h* provides the definitions of the types used in the file and the prototype for BalsaSim_RegisterBuiltinFunction. This file (which can be found in the *src/libs/balsasim* directory of the Balsa distribution and *include/balsasim* of a Balsa installation) also includes the files *balsasim/object.h*, *balsasim/parameter.h* and (through *parameter.h*) *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 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 their 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).

Compiling HelloWorld

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

```
balsa-make-builtin-lib hello hello.c
```

This should create *hello.la*, *hello.o*, *hello.a* and either *hello.so...* or *hello.dylib...* files depending on your machine architecture. The Balsa declaration file can be compiled with:

```
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.

Invoking HelloWorld

A short Balsa description such as:

```
import [hello]
procedure try is
begin
  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 S*imDemo.balsa*, the folllowing commands generate a default test harness.

```
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 precedure 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.

```
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

```
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 as described in “Setting the BALSATECH environment variable” on page 126.

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

`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 simplifes 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.

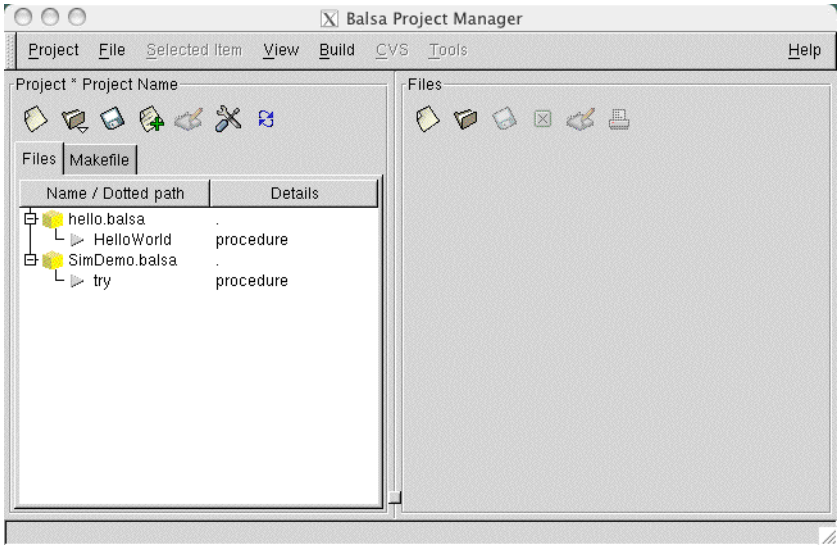

Figure 7.1: The simulation balsa files.

Figure 7.2: Adding the C language description file.

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 balsa-mgr, right-click on the *test1* test fixture name attached to the *try* procedure and select “Add Implementation”. The Verilog implenetation is added to the test1 test harness.
6. Click on the Makefile tab in the left-hand pane in the balsa-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:

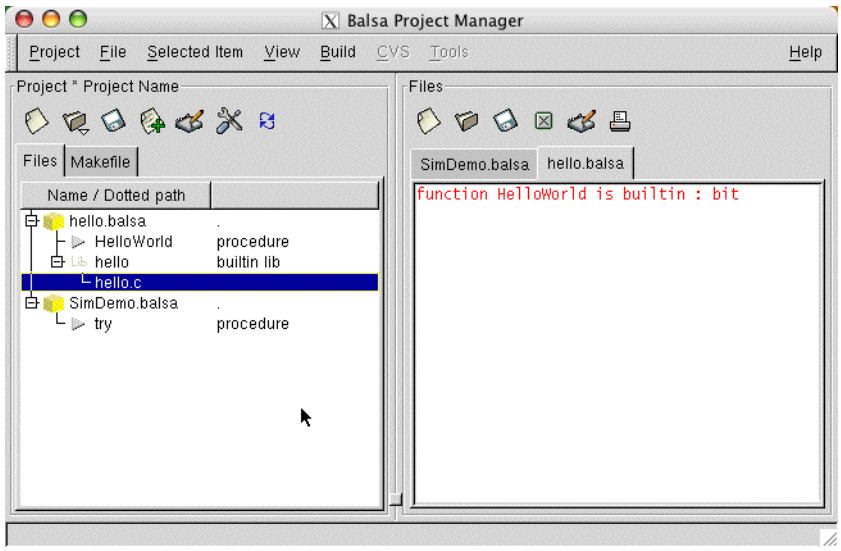

Figure 7.3: Project with library added

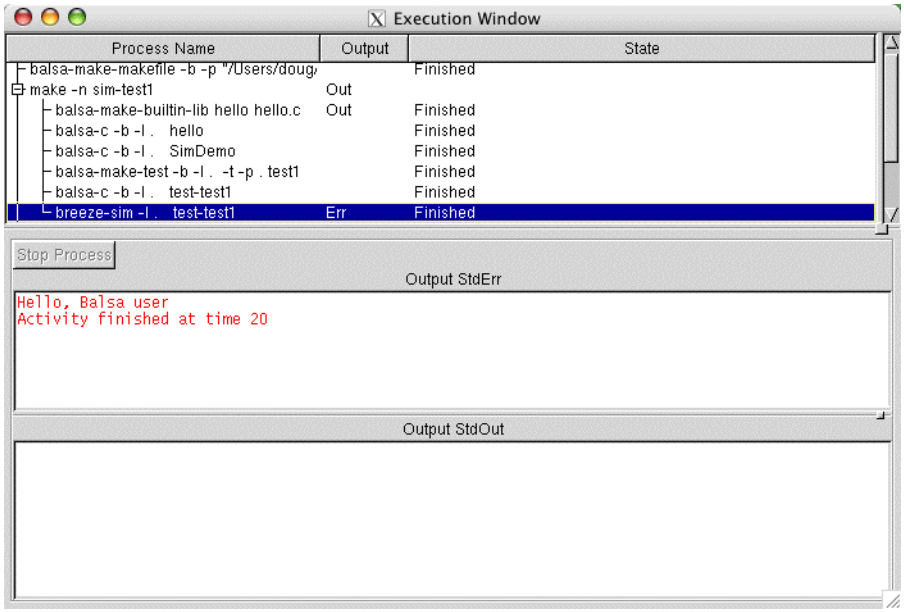

Figure 7.4: Running the simulation.

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:

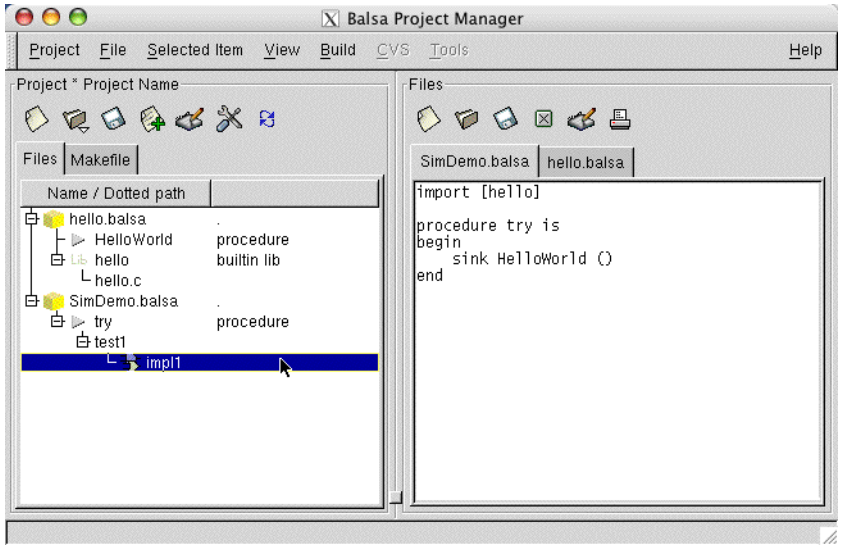

Figure 7.5: A test verilog test harness added.

```
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:

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

and is equivalent to the "pure" Balsa function:

```
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:

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

```
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*

```
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:

```
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:

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

On deallocation of the object in `instance->objects[0]`, DeleteBalsaFile will the 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:

```
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

`initialising' call to the builtin function's C function in order to resolve any uncertain argument and result widths. This function call is initiated by the simulation system noticing that the user has passed a width of 0 as the resultWidth argument, or 0 as any element of the argumentWidths argument to BalsaSim_RegisterBuiltinFunction. ToString's registration looks like this (notice the {0} passed as argumentWidths):

```
BalsaSim_RegisterBuiltinFunction ("ToString", 1, 1, Builtin_ToString,
   64, (unsigned []) {0}, 1);
```

In order to distinguish the initialising call to the C function (here this function is called Builtin_ToString) from "genuine" calls, the `instance->portWidthsAreResolved` will be false during the initialising call. This leads to a generalised form of C implementation of a builtin function with an enclosing if statement around it's body. `ToString` resolves its port width with this code (with error checking removed):

```
static void Builtin_ToString (BuiltinFunction *function,
   BuiltinFunctionInstanceData *instance)
{
   if (! instance->portWidthsAreResolved)
   {
      instance->argumentWidths[0] =
         ABS(instance->parameters[0]->info.type->size);
   } else {
      …
   }
}
```

Note that the `argumentWidths` array which is modified is the array within the instance structure and **not** the one within the function structure which must be invarient across instances of the builtin function. A function which has its result width changed during an initialising call must similarly change the `instance->resultWidth` value rather than any element of function. Any remaining argument or result widths which remain 0 after the initialising call are flagged as error by the simulation system and will cause the simulation to terminate.

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 hand 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

replaced by a new object for the current call. A loop decrements each of the reference counts of the instance–>objects elements and then checks the decremented reference count against 0. Objects with a 0 reference count then have their "–>data" payload deallocated using the appropriate destructor function and the BalsaObject structure is reused (with a new reference count of 1) for the current call's object. Objects with a reference count greater than 0 are stored elsewhere in the system and so should not be deallocated. Those objects' elements in the `instance->objects` array are therefore overwritten by pointers to newly allocated BalsaObject structures (with initial reference counts of 1 again) and the task of deallocating the original object then falls to the last BuiltinVariable or other builtin function to hold a reference to the object.

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 malloced 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 deallocaing 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.

BalsaFile

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 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 witten 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 117440512
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
```

```
10756: chan ‘o’ reading 57344
11297: chan ‘o’ reading 28672
11861: chan ‘o’ reading 14336
12397: chan ‘o’ reading 7168
12938: chan ‘o’ reading 3584
13479: chan ‘o’ reading 1792
14043: chan ‘o’ reading 896
14593: chan ‘o’ reading 448
15157: chan ‘o’ reading 224
15721: chan ‘o’ reading 112
Ended test
```

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:

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

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

```
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
end -- procedure BalsaOutputPortToLogX
```

This produces the output in binary with leading zeros with each 4 bit field separated by an underscore.

```
230: chan ‘o’ reading 0000_0000_0000_0000_0000_0000_0000_1110
727: chan ‘o’ reading 0000_0000_0000_0000_0000_0000_0000_0111
1245: chan ‘o’ reading 1000_0000_0000_0000_0000_0000_0000_0011
1749: chan ‘o’ reading 1100_0000_0000_0000_0000_0000_0000_0001
2267: chan ‘o’ reading 1110_0000_0000_0000_0000_0000_0000_0000
2785: chan ‘o’ reading 0111_0000_0000_0000_0000_0000_0000_0000
3326: chan ‘o’ reading 0011_1000_0000_0000_0000_0000_0000_0000
3839: chan ‘o’ reading 0001_1100_0000_0000_0000_0000_0000_0000
4357: chan ‘o’ reading 0000_1110_0000_0000_0000_0000_0000_0000
4875: chan ‘o’ reading 0000_0111_0000_0000_0000_0000_0000_0000
5416: chan ‘o’ reading 0000_0011_1000_0000_0000_0000_0000_0000
5943: chan ‘o’ reading 0000_0001_1100_0000_0000_0000_0000_0000
6484: chan ‘o’ reading 0000_0000_1110_0000_0000_0000_0000_0000
7025: chan ‘o’ reading 0000_0000_0111_0000_0000_0000_0000_0000
7589: chan ‘o’ reading 0000_0000_0011_1000_0000_0000_0000_0000
8111: chan ‘o’ reading 0000_0000_0001_1100_0000_0000_0000_0000
8629: chan ‘o’ reading 0000_0000_0000_1110_0000_0000_0000_0000
9147: chan ‘o’ reading 0000_0000_0000_0111_0000_0000_0000_0000
9688: chan ‘o’ reading 0000_0000_0000_0011_1000_0000_0000_0000
10215: chan ‘o’ reading 0000_0000_0000_0001_1100_0000_0000_0000
```

```
10756: chan ‘o’ reading 0000_0000_0000_0000_1110_0000_0000_0000
11297: chan ‘o’ reading 0000_0000_0000_0000_0111_0000_0000_0000
11861: chan ‘o’ reading 0000_0000_0000_0000_0011_1000_0000_0000
12397: chan ‘o’ reading 0000_0000_0000_0000_0001_1100_0000_0000
12938: chan ‘o’ reading 0000_0000_0000_0000_0000_1110_0000_0000
13479: chan ‘o’ reading 0000_0000_0000_0000_0000_0111_0000_0000
14043: chan ‘o’ reading 0000_0000_0000_0000_0000_0011_1000_0000
14593: chan ‘o’ reading 0000_0000_0000_0000_0000_0001_1100_0000
15157: chan ‘o’ reading 0000_0000_0000_0000_0000_0000_1110_0000
15721: chan ‘o’ reading 0000_0000_0000_0000_0000_0000_0111_0000
Ended test
```

The rotation of the 3 ‘1’s can now be clearly seen. If it is desired to produce the ouptut 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* illustrates 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 builtin function
  -- calls.  You can either use this module or make your own use of the
  -- builtin 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 builtin 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
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
  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
```

```
  variable data : DataWidth
  variable addr : AddrWidth
begin
  print “dumping memory”;
  addr := 0;
  loop
    addr_bus <- addr || data_bus -> data || rNw <- 1;
    print “<0x“, NumberToString (AddrWidth, addr, 16, 0, 1) , “> 0x“,
    NumberToString (DataWidth, data, 16, 0 ,1) ;
    addr := (addr + 1 as AddrWidth)
  while addr /= 0
  end
end

procedure ex2 is
  variable mem : BalsaMemory
  channel datafile : String
  channel addr_bus : AddrWidth
  channel read_bus : DataWidth
  channel write_bus : DataWidth
  channel rNw : bit
begin
  mem := BalsaMemoryNew ();
  BalsaMemory ( {addr_width , data_width},
    <- mem,
    addr_bus,
    rNw,
    write_bus, read_bus) ||
  begin
    load_mem(<- "data", addr_bus, write_bus, rNw) ;
    proc(addr_bus, write_bus, read_bus, rNw) ;
    dump_mem(addr_bus, read_bus, rNw)
  end
end
```

A Processor Test Harness

This example in Simulation/Processor ties together many of the previous examples of using the builtin 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 which 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 cellsfrom 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.

Option	Values	Notes
suggest-buffer	set/unset	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.
cad	se	Substitutes behavioural assign statements for feed-through components when importing into Silicon Ensemble and substitutes supply 0/1 for Vdd and Gnd. This is the default route.
	cadence	Currently, this option only makes sense for the amuST library. It is used for as part of the design flow for transistor-level back-annotated simulations Feed-through components are replaced with a special component to allow verilog-in to generate the correct output.
logic	DIMS	Implements “helper” cells – those cells composed from the basic cell library – in a DIMS style
	NCL	Implements “helper” cells – those cells composed from the basic cell library – using NCL style gates. In many circumstances smaller implementations result.
	Balanced	Creates balanced circuits where the notional path delays through the DIMS circuts are matched in an attenpt to defeat Differential Power Analysis attacks in security applications such as smartcards
variable	SR	Variables stored in “standard”SR latches
	Spacer	Each variable latch is reset to a NULL state before a writie operation in an attenpt to defeat Differential Power Analysis attacks in security applications such as smartcards
	NCL	Variables are stored in pipleline style latches. More efficient for 1-of-4 codes.
n-of-m mapping	set/unset	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.
sim	icarus vxl ncv vcs modelsim cver	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.

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 targetting 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

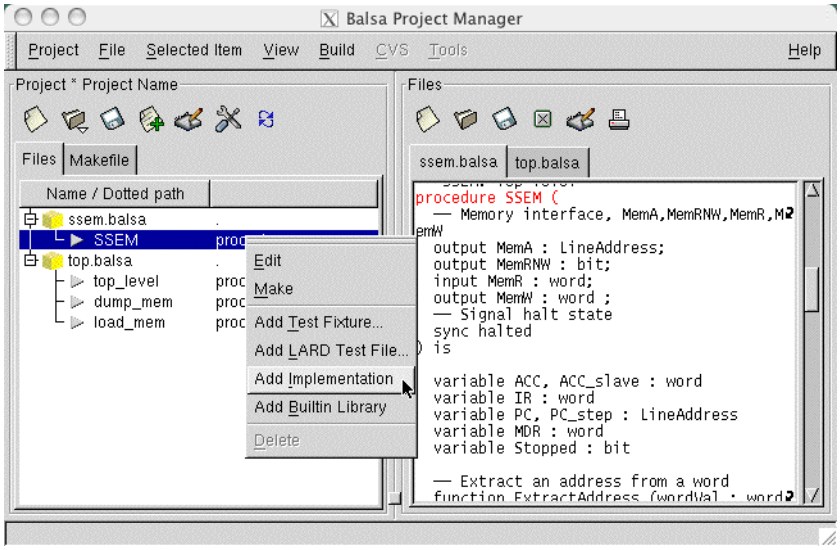

Figure 8.1: Adding an implementation.

shown in Figure 8.1) causing a dialogue box to be spawned. The user can change the nakme of the implementation and the default *balsa-netlist* options (see Section, “Balsa Reference,” on page 125). Clicking on the technology tab reveals the technology and style options. shown in Figure 8.2

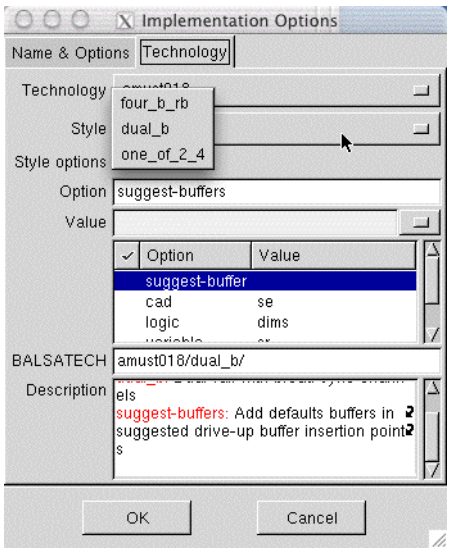

Figure 8.2: Choosing an implementation style.

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.

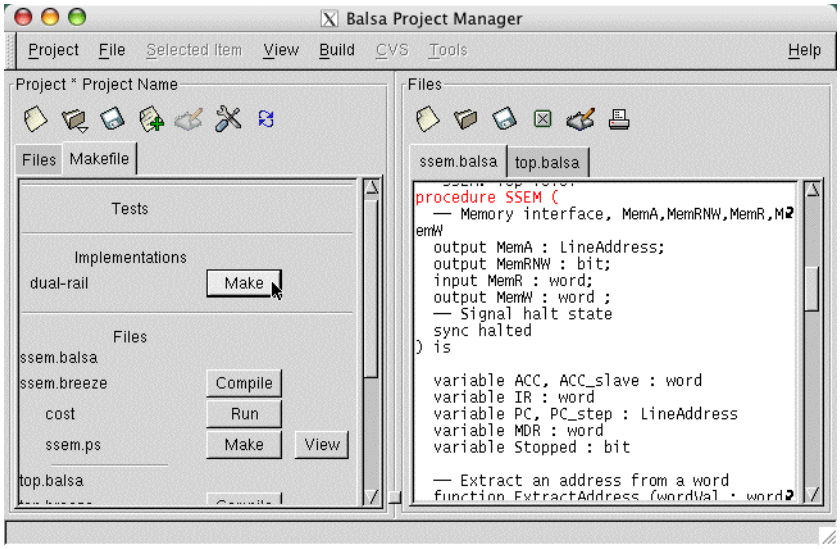

Figure 8.3: Making the implemetation.

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

9 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 110)**.** 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

Technologies and Styles

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.

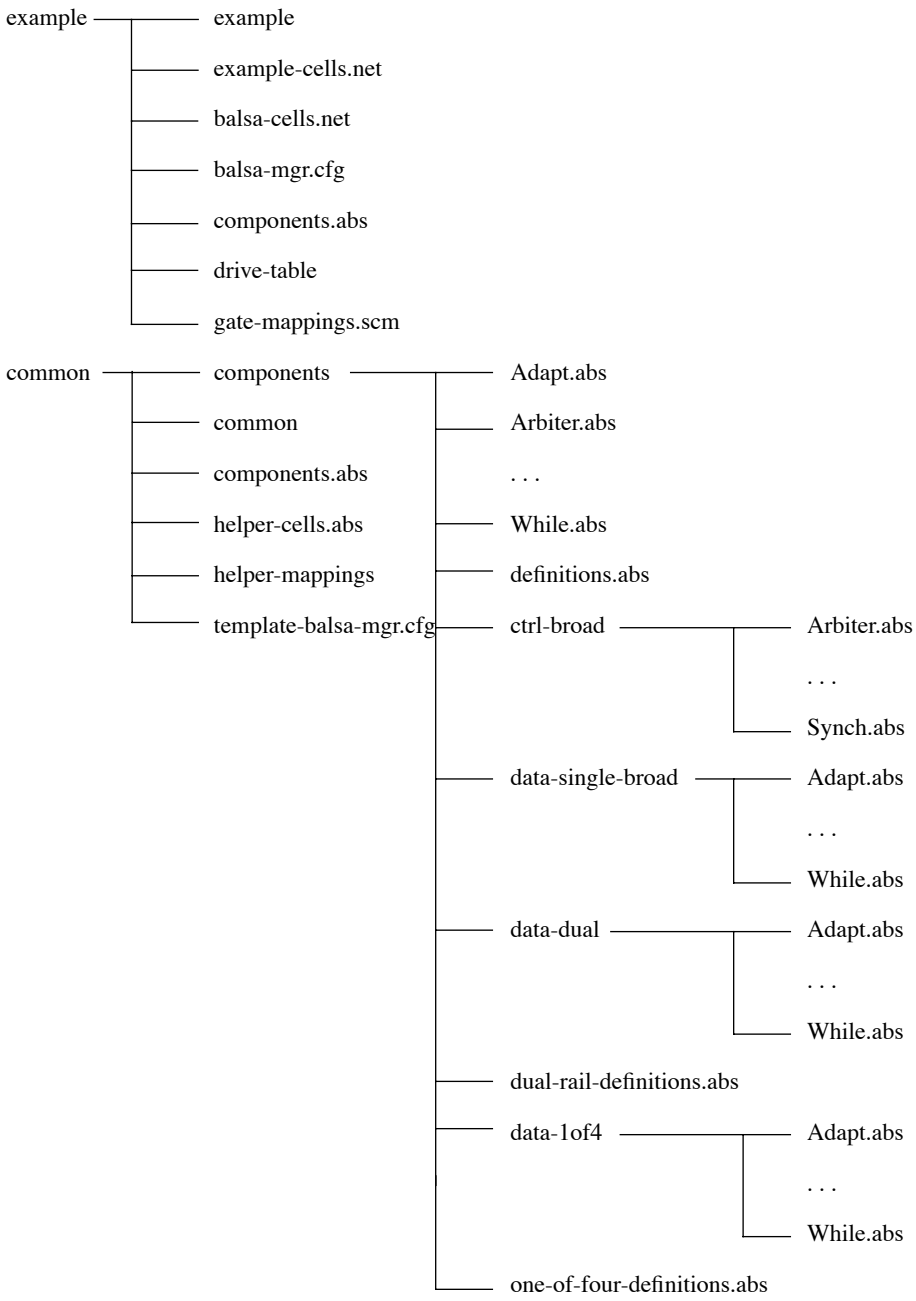

Figure 9.1: Directory structure for the example technology and common components

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.

Directory structure

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.

common directory

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.

the <tech> directory

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 105.

<technology>-cells.net: a file in .net format (see "Netlist," on page 122) 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* 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 115.

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.

```
(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")
```

Figure 9.2: A typical configuration file.

technology. 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 prefered 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 conections 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:

`tech-cell-name-max-length` determines the maximum number of characters for instance names in the netlist by default this is set at 1024 characters.

`tech-map-cell-name-import`, `tech-map-cell-name-export` allow balsa-netlist to import and export any name mappings of cells to or from a mappings file in the local directory, allowing different balsa-designs to keep consistently mapped cell names, if the tech-map-cell-name or tech-cell-name-max-length options are used.

Each option takes the name of a simple import/export function: `net-simple-cell-name-import` and `net-simple-cell-name-export` respectively

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
```

The port declaration lists the ports “sense” (whether passive or active), “direction” (input or output), type (if data port), and, if arrayed port, its low index and cardinality.

The two sections above both include a type declaration to specify the type of the expression. The types allowed are defined in “Types,” on page 120.

`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:

```
(identifier expression)
```

The complete grammar for expressions is given in "The ABS Grammar," on page 118 and includes operators such as *, /, and, if etc. It also contains several builtin functions:

```
pop-count          ;; the number of bits set in a binary 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 interger 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 builtin 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:

```
(lambda (params*)
   (let-expression?)
   (body-expression)
)
```

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

```
(let
   (identifier 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:

```
(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:

```
(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.

```
(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 1 (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")))
   )
)
(connections
   (connect (req "guard") (req (each "guardInput")))
)
```

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

9.5. Adding a new technology

Adding new technologies is straight-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`, `inverter`, `xor`, `buffer`, `2-1-mux`, `1-2demux`, `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 new full balsa-cells.net and a gate-mappings file that must be concatenated to the existing file as it only contains new mappings for cells that have been implemented by `balsa-make-helpers` – the reason why these files are handled differently by `balsa-make-helpers` is historical and of course has been forgotten.
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 113

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.

Implementation Style	Channel Sense	Port Sense	Portions
Four Phase Broad/ Reduced Broad	Push	Passive Input Active Output	req ack data
	Pull	Active Input Passive Output	req ack data
Dual Rail	Push	Passive Input Active Output	req0 req1 ack
	Pull	Active Input Passive Output	req ack0 ack1
One of Four	Push	Passive Input Active Output	req0 req1 req2 req3 ack
	Pull	Active Input Passive Output	req ack0 ack1 ack2 ack3

These portions can be accessed from channels/bundles by the portioning operators:

```
(req bundle)(ack bundle)(data bundle)
(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 and gate operator operating on two nodes, a and b, of width 2 producing a result on node c, also two bits wide:

```
(and (node c) (node a) (node b))
```

would produce two and gate operators:

(and (slice 0 1 (node c)) (slice 0 1 (node a)) (slice 0 1 (node b)))

(and (slice 11 (node c)) (slice 11 (node a)) (slice 1 1 (node b)))

which would be mapped into .net gates as:

```
(and2 (“c_0n” 0) (“a_0n” 0) (“b_0n” 0))
(and2 (“c_0n” 1) (“a_0n” 1) (“b_0n” 1))
```

and into a verilog netlist as:

```
(and2x1 c_0n[0],a_0n[0],b_0n[0])
(and2x1 c_0n[1],a_0n[1],b_0n[1])
```

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 118.

The abs language also provides the facility to add customisable cells as illustrated below:

```
(cell “cell-name” singleton-slice …)
```

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:

```
(constant value output-slice)
```

creates a constant from the decimal value, by tying wires to VCC or GND circuits.

```
(print args)
```

prints debugging messages that can be viewed when instances of the component are being generated, takes unlimited number of arguments.

```
(macro identifier args)
```

The macro operator is a method for creating complex abs expressions dependent on some parameters. A macro is lambda defined, as described, in the `defines` section. The lambda is called using the `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:

```
(if expr
gate
gate
)
```

where, *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 118.

```
(cond
(condition-expr  gate ...)
(condition-expr  gate ...)
. . .
(else gate ...)
)
```

Each condition-expr is evalutated 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,

```
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
           (numeric-type #f (param "width")) 0 (param "readPortCount"))
9     )
10       (symbol
11       (centre-string "FV")
12    )
13    (implementation
14       (style "four_b_rb" (include tech "common"
          "data-single-broad/FalseVariable"))
15       (style "dual_b"
16          (nodes
17             ("writeSig" 1 0 1)
18             ("writeSigPart" (param "width") 0 1)
19          )
20          (gates
21             (or (node "writeSigPart")(req0 "write")(req1 "write"))
22             (c-element (node "writeSig")(smash (node "writeSigPart")))
23             (s-element (node "writeSig") (ack "write") (req "signal")
                 (ack "signal"))
23             ; data read ports
24             (and (combine (ack1 (each "read")))
25             (combine (dup-each (param "width")  (req (each "read"))))
26             (combine (dup (param "readPortCount")  (req0 "write"))
27             )
28             (and (combine (ack0 (each "read")))
29             (combine (dup-each (param "width") (req (each "read"))))
30             (combine (dup (param "readPortCount")  (req1 "write")))
31             )
32          )
33          (connections)
34       )
35    )
36 )
```

Figure 9.5: Description of dual rail FalseVariable

- "signal" - sync port to enclose the "read" port activity within a handshake on the "write" port.
- "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 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 `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 c-element of line 22, the `smash` slice operator breaks down (node "writeSigPart") into individual single-wire slices, and so this operator instantiates a tree of C-elements the width of the datapath.
2. An 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 s-element is instantiated.
3. The read-ports to the read-channels are instantiated with the two `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 `each` on line 24 creates a slice for each channel in the read-port array. This is combined to a single-slice with the `combine` command, The input arguments to the and-gate highlight the difference between the `dup` and `dup-each` commands. The `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 `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:

```
(slice 0 1 (req (bundle "read" 0)) ... (slice width 1 (req (bundle "read" 0))
(slice 0 1 (req (bundle "read" 1)))
```

rather than:

```
(slice 0 1 (req (bundle "read" 0)) ... (slice 0 1 (req (bundle "read"
readPortCount)) (slice 1 1 (req (bundle "read" 0))
```

which would be produced by the `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:

`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 foreach 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) (1 "AND2") (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 absract 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.

`<technology>-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.

iinstances lists of all the instances comprising the circuit. The format of instance declaration is:

```
(instance instancename ("connection" "connection" ...))
```

where, *instancename* is the name of the instance as it appears in either the *<technology>*`-cells` or `balsa-cells` files. The connections are either nets or ports of the circuit, and are ordered in the ordering given in the .net files of the technology.

attributes Attribute declarations are of the form:

```
(attributename value)
```

Attributes currently in use are:

cell-type: defines the circuit to be a helper cell or a balsa-generated component allowing netlists to be created with helper cell descriptions removed.

global-ports: allows ports of a helper cell to be defined as global, which are then propagated through the breeze netlist to the top level, this allows, for example, explicit reset signals on helper-cells.

feedthrough: allows the insertion of assignment statements in components to avoid unnecessary buffering in designs. The arguments are the port indices of the left and right handsides of the assignment statement.

simulation-initialise: currently only configured for verilog netlists, this option signals balsa-netlist to insert verilog initialisation code into the final netlist to force certain signals into known states. The arguments to the attribute are a list of pairs (*signal-name value*), only signal-bit signals can be assigned. The resulting verilog code requires two defines to be set in the testbench `balsa_simulate`, a boolean to determine when simulation code is being used, and `balsa_init_time`, which determines the length of time the signals should be forced to their designated value before being released.

Balsa-netlist takes the .net netlist and maps it to specific netlist formats, this includes changing the instance declarations, channel naming schemes and node declarations. In formats where there is a restriction on the length of circuit names, balsa-netlist creates a new abbreviated name, in order to keep track of the original component it keeps track of this mapping in <technology>.map, in the invocation directory. Then every time this name mapping needs to take place the .map file is searched, and where possible the previous mapping is used.

9.8. The BALSATECH environment variable

```
<technologyname>/<stylename>/<styleoptions>*
```

The implementation style of a circuit is determined by the BALSATECH environment variable. This sets the technology, implementation style and also any options available for the implementation style.

e.g.

```
export BALSATECH=example/dualb/variable=spacer:logic=balanced
```

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 vaules, 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
- `one_of_2_4` – one-of-four delay insensitive encoding with return to zero signalling

9.9. The ABS Grammar

Components component description:

(primitive-part ⟨partname⟩
⟨parameter-expr⟩
⟨port-expr⟩
⟨symbol-expr⟩
⟨implementation-expr⟩
)

⟨parameter-expr⟩ **(parameters**
(**(**"⟨param-name⟩" ⟨type-expr⟩**)**)*
)

⟨port-expr⟩ **(ports**
(**(**⟨port-type⟩ "⟨portname⟩" ⟨port-sense⟩ ⟨port-direction⟩ ⟨type-expr⟩**)**)+
)

⟨port-type⟩ **port**
| **sync-port**
| **arrayed-port**
| **arrayed-sync-port**

⟨port-sense⟩ **passive**
| **active**

⟨port-direction⟩ **input**
| **output**

⟨symbol-expr⟩ **(symbol**
(centre-string "⟨symbol⟩"**)**
)

⟨implementation-expr⟩ **(implementation**
(**(style** "⟨stylename⟩" ⟨include-expr⟩ | ⟨style-descr⟩**)**)+
)

Styles Descriptions of implementation styles

⟨style-descr⟩ ⟨define-expr⟩
⟨node-expr⟩
⟨gate-expr⟩
(⟨connection-expr⟩)?

⟨define-expr⟩ **(defines**
(⟨bound-expr⟩)*
)

⟨node-expr⟩ **(nodes**
(**(** " ⟨nodename⟩ " ⟨width⟩ ⟨low-bit-index⟩ ⟨cardinality⟩ **)**)*
)

⟨gate-expr⟩ **(gates**
(**(**⟨gate-operator⟩**)**)*
)

⟨connection-expr⟩ **(connections**
(**(connect** ⟨input-slice⟩ (⟨output-slice⟩)+**)**)*
)

Gates Descriptions of gates

⟨gate-operator⟩
    ⟨Fixed Gate⟩
  | ⟨Stretchable Gate⟩
  | ⟨Control Gate⟩
  | ⟨Other Gate⟩

⟨Fixed Gate⟩
    (constant ⟨value⟩ ⟨output-slice⟩**)**
  | **(s-element** ⟨request-in-slice⟩ ⟨ack-in-slice⟩ ⟨request-out-slice ⟩ ⟨ack-out-slice⟩**)**
  | **(xor2** ⟨output-slice⟩ ⟨input-slice0⟩ ⟨input-slice1⟩**)**
  | **(mux2** ⟨output-slice⟩ ⟨input-slice0⟩ ⟨input-slice1⟩ ⟨select-slice⟩**)**
  | **(demux2** ⟨input-slice⟩ ⟨output-slice0⟩ ⟨output-slice1⟩ ⟨select-slice⟩**)**
  | **(inv** ⟨output-slice⟩ ⟨input-slice⟩**)**
  | **(keeper-inv** ⟨output-slice⟩ ⟨input-slice⟩**)**
  | **(latch** ⟨enable⟩ ⟨input-slice⟩ ⟨output-slice⟩**)**
  | **(latch-n-enable** ⟨enable-slice⟩ ⟨input-slice⟩ ⟨output-slice⟩**)**
  | **(tri-buffer** ⟨enable-slice⟩ ⟨input-slice⟩ ⟨output-slice⟩**)**
  | **(tri-inv** ⟨enable-slice⟩ ⟨input-slice⟩ ⟨output-slice⟩**)**
  | **(mutex** ⟨input-sliceA⟩ ⟨input-sliceB⟩ ⟨output-sliceA⟩ ⟨output-sliceB⟩**)**

⟨Stretchable Gate⟩
    (gnd (⟨output-slices⟩)+**)**
  | **(vcc** (⟨output-slices⟩)+**)**
  | **(connect** ⟨input-slice⟩ (⟨output-slices⟩)+**)**
  | **(and** ⟨output-slice⟩ (⟨input-slices⟩)+**)**
  | **(nand** ⟨output-slice⟩ (⟨input-slices⟩)+ **)**
  | **(or** ⟨output-slice⟩ (⟨input-slices⟩)+ **)**
  | **(nor** ⟨output-slice⟩ (⟨input-slices⟩)+ **)**
  | **(c-element** ⟨output-slice⟩ (⟨input-slices⟩)+ **)**

⟨Control Gate⟩
    ⟨if-gate-operator⟩
  | ⟨cond-gate-operator⟩
  | ⟨case-gate-operator⟩
  | ⟨gate-gate-operator⟩

⟨if-gate-operator⟩
(if ⟨expr⟩
    ⟨gate-operator⟩
    ⟨gate-operator⟩
)

⟨cond-gate-operator⟩
(cond
    (**(**⟨condition-expr⟩ ⟨gate-operator**)**) +
    (**(else** ⟨gate-operator⟩**)**)?
)

⟨case-gate-operator⟩
(case ⟨expr⟩
    (**((** ⟨test-value⟩)+**)** ⟨gate-operator⟩**)**)+
    (**(else** ⟨gate-operator⟩**)**)?
)

⟨gates-gate-operator⟩
(gates (⟨gate-operator⟩)+**)**

⟨Other Gate⟩
    (constant ⟨value⟩ ⟨output-slice⟩ **)**
  | **(print** (⟨arg⟩)* **)**
  | **(macro** ⟨macro-name⟩ (⟨macro-args⟩)***)**
  | **(encode** ⟨option⟩ **(**(⟨input-slices⟩)+**)** ⟨output slice⟩**)**
  | **(decode** ⟨option⟩ ⟨input-slice⟩ **(**(⟨output-slices⟩)+**))**
  | **(cell** “⟨cell-name⟩” (⟨singleton-slice⟩)* **)**

⟨macro-args⟩
    (⟨identifier⟩)+

⟨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 preceeded 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⟩
| ⟨string-type-expr⟩

⟨named-type-expr⟩ **(named-type** ⟨name⟩**)**

The type identified by named-type are useful predeclared types (in *balsa/types/basic* and *balsa/types/synthesis)* such as cardinality or boolean..

⟨numeric-type-expr⟩ **(numeric-type** ⟨signedness⟩ ⟨width⟩**)**

⟨alias-type-expr⟩ **(alias-type** ⟨newname⟩ ⟨oldname⟩**)**

⟨array-type-expr⟩ **(array-type** ⟨type-expr⟩ ⟨lowIndex⟩ ⟨elementCount⟩**)**

⟨enumeration-type-expr⟩ **(enumeration-type** ⟨signedness⟩ ⟨width⟩ ⟨enum-list⟩**)**

⟨record-type-expr⟩ **(record-type** ⟨width⟩ ⟨fields⟩**)**

⟨string-type-expr⟩ ⟨case-spec⟩ (**;** ⟨case-spec⟩)*

This type is only used a parameters to a select few gates which take a specification string

⟨enum-list⟩: (**(**⟨name⟩ ⟨value⟩**)**)+

⟨fields⟩ (**(**⟨name⟩ ⟨type-expr⟩**)**)+

⟨case-spec⟩ ⟨range⟩ (**,** ⟨range⟩)*

⟨range⟩ **[0-9]** (**..** **[0-9]**)*

⟨signedness⟩ **#t**
| **#f**

Expressions expression types

⟨expr⟩ ⟨lambda-dec-expr⟩
| ⟨lambda-call-expr⟩
| ⟨if-expr⟩
| ⟨arith-expr⟩
| ⟨fn-expr⟩
| ⟨scheme-expr⟩
| ⟨encoding-expr⟩
| ⟨param-expr⟩
| ⟨bind-name⟩
| ⟨primitive-expr⟩

⟨lambda-dec-expr⟩ **(lambda** ⟨identifier⟩ **(**⟨param-list⟩**)** ⟨body-expr⟩**)**

⟨lambda-call-expr⟩ **(**⟨identifier⟩ (⟨expr⟩)∗**)**

⟨if-expr⟩ **(if** ⟨expr⟩ ⟨expr1⟩ ⟨expr2⟩**)** ;;⟨expr⟩ is consequent, ⟨expr⟩ is alternative.

⟨arith-expr⟩ **(**⟨arith-op⟩ (⟨expr⟩)+**)**

⟨fn-expr⟩ **(pop-count** ⟨expr⟩**)**
| **(find-set-bit** ⟨expr⟩**)**
| **(find-clear-bit** ⟨expr⟩**)**
| **(style-option** ⟨expr⟩**)**
| **(bit-length** ⟨expr⟩**)**
| **(bit-set?** ⟨expr⟩ ⟨expr⟩**)**
| **(bit-xor** ⟨expr⟩ ⟨expr⟩**)**
| **(..** ⟨expr⟩ ⟨expr⟩**)**

| | **(print** (⟨expr⟩)∗**)**
| | **(note** (⟨expr⟩)∗**)**

⟨scheme-expr⟩ **(expt** ⟨exponent-expr⟩ ⟨expr⟩**)**
| **(mod** ⟨expr⟩ ⟨expr⟩**)**
| **(min** ⟨expr⟩ (⟨expr⟩)***)**
| **(max** ⟨expr⟩ (⟨expr⟩)***)**
| **(quotient** ⟨expr⟩ ⟨expr⟩**)**
| **(and** ⟨expr⟩ (⟨expr⟩)+**)**
| **(not** ⟨expr⟩**)**
| **(or** ⟨expr⟩ (⟨expr⟩)+**)**
| **(assoc** ⟨expr⟩ ⟨expr⟩**)**
| **(cons** ⟨expr⟩ ⟨expr⟩**)**
| **(car** ⟨expr⟩**)**
| **(cdr** ⟨expr⟩**)**
| **(cadr** ⟨expr⟩**)**
| **(caar** ⟨expr⟩**)**
| **(let** ⟨let-expr⟩ ⟨expr⟩**)**
| **(list** (⟨expr⟩)***)**
| **(length** ⟨expr⟩**)**
| **(reverse!** ⟨expr⟩**)**
| **(append** (⟨expr⟩)***)**
| **(null?** ⟨expr⟩**)**
| **(odd?** ⟨expr⟩**)**
| **(pair?** ⟨expr⟩**)**
| **(string?** ⟨expr⟩**)**
| **(string-append** (⟨expr⟩)***)**
| **(make-string** ⟨expr⟩ (⟨expr⟩)?**)**
| **(substring** ⟨expr⟩ ⟨expr⟩ (⟨expr⟩)?**)**
| **(string-set!** ⟨expr⟩ ⟨expr⟩ ⟨expr⟩**)**
| **(string-length** ⟨expr⟩**)**
| **(number->string** ⟨expr⟩**)**

⟨encoding-expr⟩ **(complete-encoding** ⟨expr⟩**)**

The argument to complete encoding is type specification string. It is used to make sure the decode/encode gate specifications are correct.

⟨param-expr⟩ **(param** " ⟨param-name⟩"**)**

⟨primitive-expr⟩ **#t**
| **#f**
| ([**0-9**])*

⟨param-list⟩ (⟨identifier⟩)*

⟨let-expr⟩ (⟨bound-expr⟩)+)

⟨bound-expr⟩ **(**⟨identifier⟩ ⟨expr⟩**)**

⟨param-list⟩ (⟨expr⟩)*

⟨arith-op⟩ **+** | **-** | ***** | **/** | **=** | **/=** | **>** | **<** | **>=** | **<=**

9.10. Netlist Format

Netlist format of .net netlists

⟨netlist⟩ (**(** ⟨net-circuit-decl⟩**)**)*

⟨net-circuit-decl⟩ **(circuit** ⟨name⟩ ⟨net-ports⟩ ⟨net-nets⟩ ⟨net-instances⟩ ⟨ net-optional⟩ **)**

⟨net-ports⟩ **(ports** (⟨net-port⟩)* **)**

⟨net-port⟩ **(** ⟨portname⟩ ⟨net-direction⟩ ⟨cardinality⟩ **)**

⟨portname⟩ ⟨channelname⟩_⟨cardinality⟩⟨portid⟩

⟨portid⟩ | **r | a | d**
| **r0d | r1d | r2d | r3d**
| **a0d | a1d | a2d | a3d**

⟨net-direction⟩ **input**
| **output**
| **inout**
| **hiz**

⟨net-nets⟩ **(nets** (⟨net-net⟩)* **)**

⟨net-net⟩ **(** ⟨name⟩ ⟨cardinality⟩ **)**

⟨net-instances⟩ **(instances** (⟨net-instance⟩) * **)**

⟨net-instance⟩ **(instance** ⟨name⟩ ⟨net-instance-connections⟩ (⟨net-instance-name⟩)? **)**

⟨net-instance-connections⟩ **(** (⟨net-instance-connection⟩)* **)**

⟨net-instance-connection⟩ **(** ⟨name⟩ ⟨index⟩ **)**
| **(** ⟨name⟩ ⟨index⟩ ⟨cardinality⟩ **)**
| ⟨name⟩
| ⟨net-vector⟩
| **unconnected**

⟨net-vector⟩ **(vector** (⟨net-instance-connection⟩)* **)**

⟨net-optional⟩ **(attributes** (⟨net-attribute⟩) * **)**

⟨net-attribute⟩ **(** ⟨net-attribute-name⟩ ⟨value⟩ **)**

⟨attributes⟩ **cell-type** ⟨cell-name⟩
global-ports ⟨portname⟩
feedthrough ⟨port-indices⟩
simulation-reset ⟨simulation-signal⟩

⟨port-indices⟩ (**[0-9]**)+ ((**[0-9]**)+)*

⟨simulation-signal⟩ (**(**⟨net-instance-connection⟩ ⟨simulation-value⟩**)**)+

⟨simulation-value⟩ **0** | **1** | **x** | **z**

10 Balsa Reference

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.

10.1. Balsa programs

balsa-c {<switch>}* <block/file-name>

The switches are:

-I	<path>	append <path> to import file path (*--import*)
-e		discard import path (*--discard-import*)
-o	<directory>	directory for output intermediate files (*--output*)
-f		flatten all procedure calls (*--flatten-calls*)
-O		DON'T optimise generated HC's (*--no-optimise*)
-b		inhibit banner (*--no-banner*)
-t	<distance>	tabs indent by <distance> places (*--tab*) Used for identifying correctly column numbers in the source code when error reporting.
-v		be verbose (*--verbose*)
-c	<option>	compilation option (*--compile-option*) <option> can be: *var-read-split* -- split variables on read bitfields as well as writes **Deprecated** code generation features: *no-wire-forks* -- don't use the WireFork component as a replacement for permanent Forks *use-masks* -- use Mask components instead of slice
--		accept no more switches
-B		don't generate a Breeze file (*--no-breeze*)
-F		generate a flat Breeze file (*--flatten-files*)
-i		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>	default print command behaviour. Type can be: (runtime|report|error|warning|fatal) (*--print-type*)

balsa-netlist {<switch>}* <block/file-name>

The switches are:

-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*)
-n	<format>	Dump a netlist in the given format (edif, verilog, compass ...) as well as any other scheduled netlist writes, several -n can be used (*--make-other-netlist*). 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*)
-I	<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*)

10.2. Setting the BALSATECH environment variable

11 The Balsa Language Definition

Summary

The syntax of the balsa language is given in Table 11.1. An extended form of BNF is used to describe the syntax. A term (*a*)* 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 ⟨⟩.

11.1. 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, sizeo, sync, then, type, val, variable, while, xor`.

11.2. Language Definition

⟨bin-digit⟩	::=	(**`0`** | **`1`**)
⟨oct-digit⟩	::=	(**`0`** ... **`7`**)
⟨dec-digit⟩	::=	(**`0`** ... **`9`**)
⟨hex-digit⟩	::=	(**`0`** ... **`9`** | **`a`** ... **`f`** | **`A`** ... **`F`**)
⟨letter⟩	::=	(**`a`** ... **`z`** | **`A`** ... **`Z`**)
⟨identifier⟩	::=	(⟨letter⟩ | _) (⟨letter⟩ | ⟨dec-digit⟩ | _)*

Table 11.1: Balsa Language Definition

⟨literal⟩	::=	(**1** ... **9**) (⟨dec-digit⟩ | _)*
	|	**0** (⟨oct-digit⟩ | _)*
	|	(**0b** | **0B**) (⟨bin-digit⟩ | _ | **x** | **X** | **?**)+
	|	(**0x** | **0X**) (⟨hex-digit⟩ | _ | **x** | **X** | **?**)+
	|	**?**
⟨string-char⟩	::=	(⟨letter⟩ | ⟨dec-digit⟩ | | **!** | **#** | **$** | **%** | **&** | **'** | **(** | **)** | ***** | **+** | **,** | **-** | **.** | **/** | **:** | **;** | **<** | **=** | **>** | **?** | @ | **[** | **]** | **^** | _ | **`** | **{** | **|** | **}** | **~**)
⟨string⟩	::=	**"** (⟨string-char⟩)* **"**
⟨file⟩	::=	(**import [** ⟨dotted-path⟩ **]**)* ⟨outer-declarations⟩
⟨dotted-path⟩	::=	⟨identifier⟩ (**.** ⟨identifier⟩)*
⟨outer-declarations⟩	::=	(⟨outer-declaration⟩)*
⟨outer-declaration⟩	::=	**type** ⟨identifier⟩ **is** ⟨type-declaration⟩
	|	**constant** ⟨identifier⟩ **=** ⟨expression⟩ (**:** ⟨type⟩)?
	|	**procedure** ⟨identifier⟩ **is** ⟨identifier⟩ (**(** ⟨procedure-formals⟩ **)**)?
	|	**procedure** ⟨identifier⟩ (**(** ⟨procedure-formals⟩ **)**)? **is** (**local**)? ⟨inner-declarations⟩ **begin** ⟨command⟩ **end**
	|	**function** ⟨identifier⟩ (**(** ⟨function-formals⟩ **)**)? **=** ⟨expression⟩ (**:** ⟨type⟩)?
	|	**if** ⟨expression⟩ **then** ⟨outer-declarations⟩ (**|** ⟨expression⟩ **then** ⟨outer-declarations⟩)* (**else** ⟨outer-declarations⟩)? **end**
⟨type-declaration⟩	::=	⟨type⟩
	|	**new** ⟨type⟩
	|	**record** ⟨identifiers⟩ **:** ⟨type⟩ (**|** ⟨identifiers⟩ **:** ⟨type⟩)* (**end** | (**over** ⟨type⟩))
	|	**enumeration** ⟨identifier⟩ (**=** ⟨expression⟩)? (**|** ⟨identifier⟩ (**=** ⟨expression⟩)?)* (**end** | (**over** ⟨type⟩))
⟨identifiers⟩	::=	⟨identifier⟩ (**;** ⟨identifier⟩)*
⟨type⟩	::=	⟨identifier⟩
	|	⟨expression⟩ (**signed**)? **bits**
	|	**array** ⟨range⟩ **of** ⟨type⟩
⟨function-formals⟩	::=	⟨identifiers⟩ **:** ⟨type⟩ (**;** ⟨identifiers⟩ **:** ⟨type⟩)*
⟨procedure-formals⟩	::=	⟨formal-parameters⟩
	|	⟨formal-ports⟩
	|	⟨formal-parameters⟩ **;** ⟨formal-ports⟩

Table 11.1: Balsa Language Definition

⟨formal-parameters⟩	::=	**parameter** ⟨identifiers⟩ **:** ⟨type⟩ (**;** **parameter** ⟨identifiers⟩ **:** ⟨type⟩)*
⟨formal-ports⟩	::=	⟨formal-port⟩ (**;** ⟨formal-port⟩)*
⟨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**
⟨channel-options⟩	::=	(**multicast**)?
⟨expression⟩	::=	⟨identifier⟩
	|	⟨literal⟩
	|	⟨string⟩
	|	(⟨identifier⟩)? **{** ⟨expressions⟩ **}**
	|	⟨identifier⟩ **'** ⟨identifier⟩
	|	⟨unary-operator⟩ ⟨expression⟩
	|	**sizeof** ⟨identifier⟩
	|	⟨expression⟩ ⟨binary-operator⟩ ⟨expression⟩
	|	⟨expression⟩ **.** ⟨identifier⟩
	|	⟨expression⟩ **[** ⟨range⟩ **]**
	|	**(** ⟨expression⟩ **as** ⟨type⟩ **)**
	|	**(** ⟨expression⟩ **)**
	|	⟨identifier⟩ **(** (⟨expressions⟩)? **)**
⟨expressions⟩	::=	⟨expression⟩ (**,** ⟨expression⟩)*

Table 11.1: Balsa Language Definition

⟨unary-operator⟩	::=	(**-** | **+** | **not** | **log** | **#**)
⟨binary-operator⟩	::=	(**+** | **-** | ***** | **/** | **%** | **^** | **=** | **/=** | **<** | **>** | **<=** | **>=** | **and** | **or** | **xor** | **@**)
⟨command⟩	::=	**continue**
	|	**halt**
	|	⟨channel⟩ **->** ⟨lvalue⟩
	|	⟨channel⟩ **->** ⟨channel⟩
	|	⟨channels⟩ **->** **then** ⟨command⟩ **end**
	|	⟨channel⟩ **<-** ⟨expression⟩
	|	**sync** ⟨channel⟩
	|	⟨lvalue⟩ **:=** ⟨expression⟩
	|	⟨block⟩
	|	⟨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**
	|	**arbitrate** ⟨channel-guard⟩ **|** ⟨channel-guard⟩ **end**
	|	**print** ⟨expressions⟩
	|	⟨identifier⟩ **(** (⟨procedure-actuals⟩)? **)**
⟨channels⟩	::=	⟨channel⟩ (**,** ⟨channel⟩)*
⟨channel⟩	::=	⟨identifier⟩
	|	⟨identifier⟩ **[** ⟨expression⟩ **]**
⟨lvalue⟩	::=	⟨identifier⟩
	|	⟨lvalue⟩ **.** ⟨identifier⟩
	|	⟨lvalue⟩ **[** ⟨expression⟩ **]**
⟨block⟩	::=	(**local** ⟨inner-declarations⟩)? **begin** ⟨command⟩ **end**
	|	**[** ⟨command⟩ **]**
⟨guards⟩	::=	⟨expression⟩ **then** ⟨command⟩ (**|** ⟨expression⟩ **then** ⟨command⟩)*

Table 11.1: Balsa Language Definition

⟨case-guard⟩	::=	⟨case-matches⟩ **then** ⟨command⟩
	|	**for** ⟨identifier⟩ **in** ⟨case-matches⟩ **then** ⟨command⟩
⟨case-match⟩	::=	⟨expression⟩
	|	⟨expression⟩ **..** ⟨expression⟩
⟨case-matches⟩	::=	⟨case-match⟩ (**,** ⟨case-match⟩)*
⟨channel-guard⟩	::=	⟨channels⟩ **then** ⟨command⟩ (**|** ⟨channels⟩ **then** ⟨command⟩)*
⟨procedure-actuals⟩	::=	⟨actual-parameters⟩
	|	⟨actual-channels⟩
	|	⟨actual-parameters⟩ **,** ⟨actual-channels⟩
⟨actual-parameters⟩	::=	⟨actual-parameter⟩ (**,** ⟨actual-parameter⟩)*
⟨actual-parameter⟩	::=	⟨expression⟩
	|	(**type**)? ⟨type⟩
⟨actual-channels⟩	::=	⟨actual-channel⟩ (**,** ⟨actual-channel⟩)*
⟨actual-channel⟩	::=	⟨identifier⟩
	|	⟨actual-channel⟩ **[** ⟨range⟩ **]**
	|	**<-** ⟨expression⟩
	|	**->** ⟨lvalue⟩
	|	⟨block⟩
	|	**{** ⟨actual-channel⟩ (**,** ⟨actual-channel⟩)* **}**
	|	⟨actual-channel⟩ **@** ⟨actual-channel⟩

Table 11.1: Balsa Language Definition

12 The Breeze Language Definition

Summary

Breeze is the intermediate language used for compiled Balsa prgrams. It serves as a repository for libraries and 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.

The syntax of the breeze language is given in Table 12.1. An extended form of BNF is used to describe the syntax. A term (*a*)* 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 ⟨⟩.

⟨dec-digit⟩	::=	(**0** ... **9**)
⟨lc-letter⟩	::=	(**a** ... **z**)
⟨letter⟩	::=	(**a** ... **z** | **A** ... **Z**)
⟨positive⟩	::=	(**1** ... **9**) (⟨dec-digit⟩)*
⟨natural⟩	::=	(**0** | ⟨positive⟩)
⟨integer⟩	::=	(⟨natural⟩ | **-** ⟨positive⟩)
⟨identifier⟩	::=	**"** (⟨letter⟩ | _) (⟨letter⟩ | ⟨dec-digit⟩ | _)* **"**
⟨boolean⟩	::=	(**#t** | **#f**)
⟨symbol⟩	::=	⟨lc-letter⟩ (⟨lc-letter⟩ | ⟨dec-digit⟩ | _ | **:**)*
⟨string-char⟩	::=	(⟨letter⟩ | ⟨dec-digit⟩ | | **!** | **#** | **$** | **%** | **&** | **'** | **(** | **)** | ***** | **+** | **,** | **-** | **.** | **/** | **:** | **;** | **<** | **=** | **>** | **?** | **@** | **[** | **]** | **^** | _ | **`** | **{** | **|** | **}** | **~** | **** | **"**)
⟨quoted-symbol⟩	::=	**"** ⟨symbol⟩ **"**

Table 12.1:

⟨string⟩	::=	**"** (⟨string-char⟩)* **"**
⟨breeze-file⟩	::=	(⟨import⟩)* (⟨definition⟩)*
⟨import⟩	::=	**(import** ⟨identifier⟩ **)**
⟨definition⟩	::=	⟨constant-defn⟩
	|	⟨type-defn⟩
	|	⟨part-defn⟩
	|	⟨balsa-defn⟩
	|	⟨netlist-defn⟩
	|	⟨composition-defn⟩
⟨constant-defn⟩	::=	**(constant** ⟨identifier⟩ ⟨integer⟩ ⟨type⟩ **)**
⟨type-defn⟩	::=	**(type** ⟨identifier⟩ ⟨type⟩ **)**
⟨type⟩	::=	**(numeric-type** ⟨boolean⟩ ⟨positive⟩ **)**
	|	**(enumeration-type** ⟨boolean⟩ ⟨positive⟩ (**(** ⟨identifier⟩ ⟨integer⟩ **)**)+ **)**
	|	**(record-type** ⟨positive⟩ (**(** ⟨identifier⟩ ⟨type⟩ **)**)+ **)**
	|	**(array-type** ⟨type⟩ ⟨integer⟩ ⟨positive⟩ ⟨type⟩ **)**
⟨part-defn⟩	::=	**(breeze-part** ⟨identifier⟩ **(ports** (⟨part-port⟩)* **) (attributes** (⟨part-attribute⟩)* **) (channels** (⟨channel⟩)* **) (components** (⟨component⟩)* **))**
⟨balsa-defn⟩	::=	...
⟨part-port⟩	::=	**(sync-port** ⟨identifier⟩ ⟨port-sense⟩ (⟨option⟩)* **)**
	|	**(port** ⟨identifier⟩ ⟨port-sense⟩ ⟨port-direction⟩ ⟨type⟩ (⟨option⟩)* **)**
	|	**(arrayed-port** ⟨identifier⟩ ⟨port-sense⟩ ⟨port-direction⟩ ⟨type⟩ ⟨integer⟩ ⟨positive⟩ ⟨type⟩ (⟨option⟩)* **)**
	|	**(arrayed-sync-port** ⟨identifier⟩ ⟨port-sense⟩ ⟨integer⟩ ⟨positive⟩ ⟨type⟩ (⟨option⟩)* **)**
⟨port-sense⟩	::=	(**active** | **passive**)
⟨port-direction⟩	::=	(**input** | **output**)
⟨part-attribute⟩	::=	**(is-procedure)**
	|	**(is-function)**
	|	**(is-permanent)**
	|	⟨view-attribute⟩
	|	⟨position-option⟩
	|	⟨option⟩
⟨position-option⟩	::=	**(at** ⟨natural⟩ ⟨natural⟩ **)**

Table 12.1:

⟨option⟩	::=	**(** ⟨symbol⟩ (⟨value⟩)* **)**
⟨value⟩	::=	⟨integer⟩
	|	⟨identifier⟩
	|	⟨boolean⟩
	|	⟨symbol⟩
	|	**(** (⟨value⟩)* **)**
⟨channel⟩	::=	**(sync** ⟨identifier⟩ (⟨option⟩)* **)**
	|	**(** ⟨channel-sense⟩ ⟨positive⟩ ⟨identifier⟩ (⟨option⟩)* **)**
⟨channel-sense⟩	::=	(**push** | **pull**)
⟨component⟩	::=	**(component** ⟨identifier⟩ **(** ⟨parameter⟩
⟨parameter⟩	::=	(⟨integer⟩ | ⟨identifier⟩)
⟨channel-no⟩	::=	⟨positive⟩
	|	**(** (⟨positive⟩)+ **)**
⟨view-attribute⟩	::=	**(view** ⟨identifier⟩ ⟨view-spec⟩ ⟨options⟩ **)**
⟨view-spec⟩	::=	**(lines-spec** (⟨string⟩)+ **)**
	|	**(list-spec** ⟨quoted-symbol⟩ ⟨value⟩ **)**
⟨netlist-defn⟩	::=	**(breeze-netlist** ⟨identifier⟩ **(ports** (⟨net-port⟩)* **)** **(attributes** (⟨breeze-net-attribute⟩)* **)** **(nets** (⟨net-net⟩)* **)** **(instances** (⟨net-instance⟩)* **))**
⟨net-port⟩	::=	**(** ⟨identifier⟩ ⟨net-direction⟩ ⟨positive⟩ **)**
⟨net-direction⟩	::=	(**input** | **output** | **inout** | **hiz**)
⟨net-net⟩	::=	**(** ⟨identifier⟩ ⟨positive⟩ **)**
⟨net-instance⟩	::=	**(instance** ⟨identifier⟩ **(** (⟨net-instance-connection⟩)* **)** (⟨option⟩)* **)**
⟨net-instance-connection⟩	::=	⟨net-single-instance-connection⟩
	|	**(vector** (⟨net-instance-connection⟩)+ **)**
⟨net-single-instance-connection⟩	::=	⟨identifier⟩
	|	**(** ⟨identifier⟩ ⟨natural⟩ **)**
	|	**(** ⟨identifier⟩ ⟨natural⟩ ⟨positive⟩ **)**
	|	**unconnected**

Table 12.1:

⟨breeze-net-attribute⟩	::=	⟨view-attribute⟩
	|	⟨option⟩
⟨composition-defn⟩	::=	**(`breeze-composition`** ⟨identifier⟩ **(`ports`** (⟨part-port⟩)* **)** **(`attributes`** (⟨comp-attribute⟩)* **)** **(`nets`** (⟨net-net⟩)* **)** **(`instances`** (⟨comp-instance⟩)* **))**
⟨comp-instance⟩	::=	**(`instance`** ⟨identifier⟩ **(** (⟨comp-instance-connection⟩)* **)** (⟨option⟩)* **)**
⟨comp-instance-connection⟩	::=	⟨net-single-instance-connection⟩
	|	⟨comp-single-instance-connection⟩
	|	**(`vector`** (⟨comp-instance-connection⟩)+ **)**
⟨comp-single-instance-connection⟩	::=	**(** ⟨comp-portion-connection⟩ ⟨natural⟩ **)**
	|	**(** ⟨comp-portion-connection⟩ ⟨natural⟩ ⟨positive⟩ **)**
	|	⟨comp-portion-connection⟩
⟨comp-portion-connection⟩	::=	⟨identifier⟩
	|	**(** ⟨portion⟩ ⟨identifier⟩ **)**
	|	**(** ⟨portion⟩ ⟨identifier⟩ ⟨natural⟩ **)**
⟨comp-attribute⟩	::=	⟨view-attribute⟩
	|	⟨position-option⟩
	|	⟨option⟩
⟨portion⟩	::=	⟨portion-name⟩
	|	**(** ⟨portion-name⟩ ⟨natural⟩ **)**
⟨portion-name⟩	::=	(**`req`** | **`ack`** | **`data`**)

Table 12.1:

13 References

[1] Kees van Berkel. *"Handshake CIrcuits - an Asynchronous Architecture for VLSI programming"*. Cambridge International Series on Parallel Computerss 5, Cambridge University Press ,1993

[2] Theseus Logic Inc. <http://www.theseus.com>

[3] Part 2 of "Principles of Asynchronous Circuit Design: A Systems Perspective", Eds Sparsø & Furber, Kluwer Academic Publishers, ISBN 0-7923-7613-7, 2001

[4] http://www.cs.man.ac.uk/apt/projects/lard/index.html

[5] A. Bardsley, "Implementing Balsa Handshake Circuits", Ph.D. thesis University of Manchester, 2000.