Balsa – A Tutorial Guide

1 Introduction

1.1 What is Balsa?

Balsa is 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 system [ref] 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 predicatable changes at the circuit implementation level. This is important if optimisations and design-tradeoffs are to be made easily and contrasts with a VHDL description in which small changes in the specification may make radical alterations to the resulting circuit (indeed it is possible to describe systems in VHDL that are not synthesisable at all!).

1.2 Basic Concepts

A circuit described in Balsa is compiled into a communicating network composed from a small (~35) 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 circuit to to one *active* port of another handshake circuit. 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.

The compilation function results in circuit fragments with both active input and output ports. Circuits constructed from compositions of compiled circuit fragments must have their interconnecting ports connected by *passivator* components. The passivator synchronises the 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.

1.3 Design Flow

Balsa comprises the following collection of tools:

- *balsa-c*: the compiler for the Balsa language. The ouput of the compiler is an intermediate language *Breeze.*

- *breeze2ps*: a tool which produces a postscript file of the handshake circuit graph.
- *breeze2lard*: a translater that converts a *breeze* description to a *LARD* behavioural model.
- *breeze-cost*: a tool which gives an area cost estimate of the circuit.
- *balsa-md:* a tool for genrating makefiles
- *balsa-mgr:* a graphical front-end to *balsa-md* with project management facilities. In this release, this tool is not fully functional and is included for experimentation only.

1.4 Manual Structure

The manual is structured as a tutorial. Each section introduces new concepts or elaborates features introduced previously. The language is illustrated by a few simple circuit examples which are developed throughout the tutorial. Although each section has a main theme, unrelated language features may also be highlighted during the explanation of the circuit description. All the programs listed in this tutorial can be found in the *examples* subdirectory.

2 Getting Started

The following environment variables need to be set:

- **BALSAHOME=**<installation directory>[1]
- **PATH=**${PATH}:${BALSAHOME}/bin

2.1 A Single-Place Buffer

The buffer circuit is the HDL equivalent of the <*hello wo*rld> program. A Balsa program would be:

```
(-- buffer1a.balsa
    Balsa program defining an 16-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]
public
type word is 16 bits

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

1. Only required for binary-only distributions

Commentary on the Code

This balsa description builds a single-place buffer, 16-bits wide. The circuit requests a word 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. This little program 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. A full list of the current definitions is given in *$BALSAHOME/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.

privacy: The keyword `public` makes subsequent declarations visible by other modules which may *import* the compiled file. A similar keyword, `private`, hides subsequent declarations. private definitions must appear before public definitions and there may only be one instance of each of the keywords in a file.

typing: the `type` keyword introduces a user-defined data type – `word`. The result of this declaration is to introduce a numeric type, *word*, which will be implemented as a bit-vector, 16-bits wide. As will be seen later, Balsa is **strongly** typed. Numeric expressions may be assigned to variables defined a being of type `word` as long as they evaluate to `0 … 65535`. If *word* had been defined as:

```
type word is 16 signed bits
```

then word would still have been implemented as a bit-vector 16-bits wide, but numerical expressions of type `word` would only be valid in the Balsa program if they were in the range `[-32768 , +32767]`

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 16-bit input datapath and an 16-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 `local variable x` (of type `word` and therefore 16-bits wide) is declared implying that a 16-bit wide storage element will be appear in the synthesised circuit.

The behaviour of the circuit is obvious from the code: 16-bit values are transferred from the environment to the storage variable, *x*, and then sequentially output from the variable to the environment. This sequence of events is continually repeated (`loop` … `end`).

channel assignment: the assignment operator “`->`” is a channel assignment and implies a communication or handhake over the channel. 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.

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

2.2 Compiling the Circuit

```
balsa-c buffer1a
```

The compiler runs silently, 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 *balsac* command. The file extension *.breeze* **is** of significance to the compilation systen

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:

```
breeze2dot buffer1a.
```

2.3 The Synthesised Circuit.

The resulting handshake circuit is shown in Fig. 1. Note that this is not actually taken from the

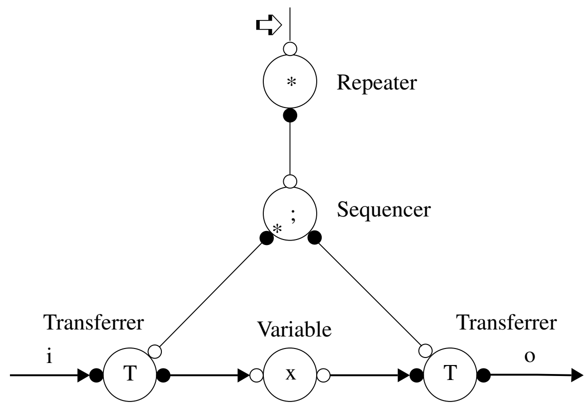

Figure 1: Handshake circuit for a Single Place Buffer

output of breeze2dot, but has been redrawn to make the diagram more readable. Although it is not necessary to understand the exact operation of the compiled circuit, an understanding of the structure is important 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.

Handshake components are represented by circles; active ports are denoted by filled bubbles on components; passive ports are denoted by open bubbles on components. Data-bearing ports are connected by a line with an arrow head to indicate the direction of data transfer. Control-only channels are connected by a plain line. The circuit complexity of many handshake elements is low: for example, a transferrer may be implented using only wires.

The port at the top, denoted by ➪ is an *activation* port which *encloses* 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 *repeater* which initiates a handshake with the *sequencer*. This component first isuues a handshake to the left-hand *transferrer*, causing data to be moved to the storage element in the *variable* element. The sequencer then handshakes with the right-hand transferrer 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.

3 Composing Circuits

Having built a single place buffer, an obvious goal is a pipeline of stages. Initially consider a two-place buffer; there are a number of ways we might describe this:

3.1 A First (naive) 2-place Buffer

```
-- buffer2a.balsa: a 2- place buffer
-- Sequential version with assigment between variables

import [balsa.types.basic]

public
type word is 16 bits

procedure buffer (input i : word; output o :word) is
local variable x1, x2 : word
begin
  loop
    i -> x1;        -- Input  communication
    x2 := x1;       -- Implied cmmunication
    o <- x2         -- Output communication
  end
end
```

In this example we explicitly introduce 2 storage elements, *x1* and *x2*. The contents of ther 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.

3.2 An Alternative (naive) 2-place Buffer

```
--  buffer2b.balsa: 2 place buffer
-- Sequential version with an explicit internal channel

import [balsa.types.basic]

public
type word is 16 bits

procedure buffer (input i : word; output o :word) is
local variable x1, x2 : word
      channel chan : word
begin
  loop
    i -> x1;        -- Input  communication
    chan <- x1 ||  chan -> x2;
    o <- x2         -- Output communication
  end
end
```

In this version the channel, which was 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

Operation of buffer2a and buffer2b

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 afer this sequence of operations is complete can new data from the environment be read into *x1* again.

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.

3.3 A 2-place buffer using parallel composition

```
-- buffer2c.balsa
-- a 2-place buffer using parallel composition
import [balsa.types.basic]
import [buffer1a]
public
type word is 16 bits

procedure buffer2c (input i : word; output o : word) is
local channel c : word
begin
        buffer (i, c) ||
        buffer (c, o)
end
```

Commentary on the Code

In the program 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 feaures 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 instantiaton 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 a temporal parallelism. The two buffers are physically connected in series.

4 Placing Multiple Structures

If we wish to extend the number of places in the 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. One technique is to use the `for` construct together with compile-time constants.

```
-- buffer_n.balsa: n-place paramterised buffer
import [balsa.types.basic]
import [buffer1a]
public
type word is 16 bits

constant n = 8

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

Commentary on the Code

This example introduces:

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.

arrayed channels: procedure ports and locally declared channels may be arrayed. Each channel can be referred to by a numeric or enumerated index, but each channel is distinct from the point of view of handshaking 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 a parallel composition – as in the example above – or sequential. In the latter case, “;” should be substituted for || in the loop specifier.

A more flexible approach uses paramterised procedures and is discussed later [see “Parameterised Procedures” on page 26].

5 Micropipeline Style Circuits

The previous examples have created “pull” style pipelines in which the pipeline control logic has effectively demanded data from its environment. This is a consequence of the active-ported nature of circuit fragments synthesised by the Balsa compilation function. There are many situations where it is more appropriate to use a “push” style, micropipeline-like approach. The balsa *select* construct may be used to implement unbufferred, data-driven, unarbitrated choice. A simple micropipeline circuit is:

```
-- buffer1s.balsa: Single place micropipeline push-style buffer
import [balsa.types.basic]
public

procedure buffer (input i : byte; output o : byte) is
local variable x : byte
begin
  loop
    select i then
      x := i      -- store the data
    end;
  o <- x
  end
end
```

Commentary on the Code

This example illustrates:

predefined types: In this example an 8-bit wide buffer is specified using the predefined type `byte` imported from balsa.types.basic. This is actually the first example in which this statement has been required.

input selection: The `select` statement is used to define a data-driven input. The action of storing data in the latch variable *x*, is initiated by the request signal from the environment. The handshake on the input port encloses the actions defined in the body of the `select` statement which in this case is the assignment into the latch.

In general the `select` statement allows a choice of inputs; in this example there is no choice: only a single input is defined. `select` is a very flexible construct and its use is explored in more detail in later examples.

6 Data Typing

In the subsequent sections, various flavours of counters are described in Balsa to illustrate several new language features. More sophisticated systolic counters, better suited to an asynchronous approach are described later in the manual [see “Systolic Counters” on page 35]. Consider first a simple binary counter. A variable is required to hold the count value; on every count command, the variable must be incremented. Care must be taken to ensure that Balsa’s strict typing rules are obeyed.

6.1 A Simple Modulo-16 Counter

```
-- count16a.balsa: a simple active-ported counter
import [balsa.types.basic]
public

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

Commentary on the Code

In an asynchronous world, there are no clocks as such to count. Instead, this circuit counts handshakes on the input control port *aclk*. The program waits for the input handshake to complete, then places the curent value of the internal 4-bit count variable on the output port and finally increments the variable. Note that the counter is not initialised.

strict typing

It is not possible to write the simple statement `count_reg := count_reg + 1` because the addition of a 4 bit variable and the literal 1 is potentially 5 bits. The result therefore has to be cast back into a 4-bit nibble using the `as` construct. Note, that for simplicity of parsing, it is unfortunately necessary to parenthesise the expression that recasts the result.

Other features of the programme are:

sync channels: A sync channel is one without any associated data: it performs purely a control function. In this example the port *aclk* is declared as a sync port. The `sync` statement in the loop body waits for a handshake to complete on the *aclk* port.

active-ported inputs: Balsac usually generates circuits with active ports. Thus this counter initiates a request on the *aclk* port and waits for the environment to respond with an acknowledgement to complete the handshake.

auto-assignment: expressions of the form `x := x + y` are allowed in Balsa. However, the resulting implementation is not quite as simple as might be implied by the code. This issue is elaborated later [see “Auto-assignment” on page 11].

7 Exploiting the *select* Statement

The previous program generates a circuit with active input ports. If this is not what is required, the select construct can be used to change the active *aclk* port to a passive port. In this case the handshake is initiated by a request from the environment and the counter will acknowledge this request to complete the handshake.

7.1 A Passive-Input Modulo-16 Counter

```
-- count16b.balsa: Simple counter with a passive input port
import [balsa.types.basic]
public

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

Commentary on the Code

The circuit waits for a request on the channel *aclk* and then completes the handshake by issuing an acknowledge signal. The 4-bit count variable is placed on the output channel and the variable then incremented.

This program illustrates:

input selection: As in the micropipeline example, the `select` statement is used to generate a passive port for the *aclk* handshake.

continue statement: The `continue` statement is equivalent to `skip` in CSP: it is a null statement; however omitting it will cause a syntax error because an action (even a null action) is required between the `then` and `end` delimiters

7.2 Re-ordering the Handshakes

The *select* statement in the previous example is merely used to generate a circuit module with a passive input channel rather than one with the default active input channel. However, the communication is still atomic. The `select` statement allows actions to be nested within the enclosing handshake as illustrated by the next example.

```
-- count16c.balsa: Simple counter with a passive input port.
import [balsa.types.basic]
public

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

```
  end
end
```

Commentary on the Code

The `select` statement here encloses the incrementing of the counter variable with the output channel communication (`count <- count_reg`). Only after the increment operation is the input communication completed and the module driving the counter circuit allowed to proceed. The question of which of the form of the select statement is better obviously depends on the context of its use: there is little to choose between the two forms in the examples above. Hovever, the second form allows more flexibility by allowing multiple actions to take place within the body of the statement.

8 Auto-assignment

As mentioned earlier [see "auto-assignment" on page 9], auto-assignment constructs of the form `x := x + y` 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. In fact the following program, in which an temporary variable is explicitly introduced, has an identical implementation to the previous program

```
-- count16d.balsa: similar to count16c, but with a
-- temporary register used to avoid auto-assignment
import [balsa.types.basic]
public

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

Since auto-assigment generates twice as many variables as might be suspected, it is probably better practice to avoid auto-assigment, explicitly introduce the extra variable and then re-write the program to hide the sequential update thereby avoiding any time penalty. The previous example becomes:

```
-- count16e.balsa: Optimised: write-back overlaps output assignment
import [balsa.types.basic]
public

procedure count16 (sync aclk; output count : nibble) is
local variable count_reg, tmp : nibble
begin
  loop
    select aclk then
      tmp := (count_reg + 1 as nibble)||
      count <- count_reg;
```



```
      count_reg := tmp
    end
  end
end
```

In this example, incrementing of the temporary variable is overlapped with the output of the previous value of *count_reg* by use of the `||` operator.

9 Conditional Statements

9.1 if … then …else

Balsa has a conventional `if … then …else` statement. Its use is illustrated in the next example which turns the previous binary counter into a decade counter.

```
-- count10a.balsa: an aysnchronous decade counter
import [balsa.types.basic]
public

type C_size is nibble
constant max_count = 9

procedure count10(sync aclk; output count: C_size) is
local
  variable count_reg : C_size
  variable tmp : C_size
begin
  loop
    select aclk then
       if count_reg /= max_count then
          tmp := (count_reg + 1 as C_size)
       else
          tmp := 0
       end ||  count <- count_reg ;
       count_reg := tmp
    end -- complete select H/S
  end -- loop end
end
```

Commentary on the Code

Since this is not a pure binary counter, an explicit test must be made to reset the counter to zero when the terminal count is reached. The example also illustrates:

/= operator: Not equals is denoted by **/=**

use of type and constant declarations: Good programming practice applies as much to HDLs as to conventional programming languages. By defining a local type *C-size* and by using the *constant* declaration, we have pulled magic numbers out of the body of the code thereby making it more readable and more easy to change to a different length counter – say a modulo-99 counter.

9.2 Parallel Guard Evaluation

Balsa does not have an `elseif` keyword. Nested `if … then …else` statements may be written:

```
if condition1 then
  procA
else if condition2 then
  procB
end else
  procC
end
```

Note the sequencing implicit in this coding. The test for *condition2* is made after the test for *condition1* has completed. If it is known that the various conditions (guards) are mutually exclusive, the statement may be written:

```
if condition1 then
  procA
|  condition2 then
  procB
else
  procC
end
```

The **|** operator causes *condition1* and *condition2* to be tested in parallel. The result is undefined if more than one guard is satisfied.

10 Record Types

Balsa possesses record types which are are bitwise compositions of (possibly different) pre-declared types. Records are similar to C structs except that a defined order and packing for the elements of the types is defined. The use of a record data structure is illustrated by extending the decade counter to be a loadable decade counter:

```
-- count10b.balsa: an aysnchronous loadable decade counter

import [balsa.types.basic]
public

type C_size is nibble
constant max_count = 9

type In_bundle is record
  ld_data : C_size ;
  ld_ctrl : bit
end

procedure updown10 (input in_sigs: In_bundle; output count: C_size) is
local
  variable count_reg : C_size
  variable tmp : C_size
begin
  loop
    select in_sigs then
      if  in_sigs.ld_ctrl = 0 then
        count_reg := in_sigs.ld_data
```



```
      else
        if count_reg /= max_count then
          tmp := (count_reg + 1 as C_size)
        else
          tmp := 0
        end || count <- count_reg ;
        count_reg := tmp
      end -- end if
    end -- complete select H/S
  end -- loop end
end
```

Commentary on the Code

The interface to the the counter consists of

1. an input bundle of the data to load into the counter together with a load/count control signal
2. an output bundle comprising the current count value.

A data type, *In_bundle*, is declared which consists of data to be loaded into the counter (*ld_data*), and an active low control signal (*load_l*) which determines whether the counter counts or loads on the handshake. Record selection is performed by the "." operator:

```
count_reg := in_sigs.ld_data
```

selects the *ld_data* bit of the input bundle, *in_sigs*, and assigns it to the internal variable *count_reg*.

At each handshake on the input, the input data bundle is read. If the load/count bit indicates a data load, the internal count register is loaded with the data in the input bundle, otherwise the count is incremented. When the count reaches 9, the counter wraps to zero. This simple interface is similar to a conventional synchronous counter in which the request signal of the input handshake corresponds to the clock signal. One difference is that the output count appears on a channel and therefore the environment must complete a handshake on the output for the count to continue.

Padding record structures

A record structure can be forced to be padded to a given length:

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

The 5-bit record is forced to be padded to 8 bits by use of the *over* keyword. Even if padding is not required, 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
```

11 Case Statements

The case statement is a multiway decision maker that tests whether an expression matches one or more possible values. Its use is illustrated in the next example which extends the loadable decade counter to become a loadable up/down decade counter.

```
-- count10c.balsa: an aysnchronous up/down loadable decade counter
import [balsa.types.basic]
public

type C_size is nibble
constant max_count = 9

type In_bundle is record
  ld_data : C_size ;
  ld_ctrl : bit;
  dir : bit
end

procedure updown10 (input in_sigs: In_bundle; output count: C_size) is
local
  variable count_reg : C_size
  variable tmp : C_size
begin
  loop
    select in_sigs then
      if  in_sigs.ld_ctrl = 0 then
        count_reg := in_sigs.ld_data
      else
        case in_sigs.dir of
          0 then             -- counting down
            if count_reg /= 0 then
               tmp := (count_reg - 1 as C_size)
            else
               tmp := max_count
            end || count <- count_reg
        | 1 then               -- counting up
            if count_reg /= max_count then
              tmp := (count_reg + 1 as C_size)
            else
              tmp := 0
            end || count <- count_reg
        end ; -- end case statement
        count_reg := tmp
      end -- end if
    end -- complete select H/S
  end -- loop end
end
```

The "|" symbol separates different branches of the *case* statement. The keyword *also* is a synonym for "|". If all possible guard vaules of the *case* statement are not enumerated, the compiler reports a non-fatal warning. A single guard may match more than one value of the guard expression and subrange expressions can be used, for example:

```
case n of 0,2 then CmdA
| 1, 3 .. 5, 7 then CmdB
| 6 then CmdC
end
```


CmdA is executed is *n* is 0 or 2, CmdB is executed if *n* is 1 or 3 or 4 or 5 or 7 and CmdC is executed if *n* is 6.

12 Enumerations

Enumerated types are very useful for enhancing the readability of the code. The previous example can be rewritten:

```
-- count10d.balsa: an aysnchronous up/down decade counter
-- illustrating use of enumeration for readability

import [balsa.types.basic]
public

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
  ld_data : C_size ;
  ld_ctrl : mode;
  dir : dir
end

procedure updown10 (input in_sigs: In_bundle; output count: C_size) is
local
  variable count_reg : C_size
  variable tmp : C_size
begin
  loop
    select in_sigs then
      if  in_sigs.ld_ctrl = load then
        count_reg := in_sigs.ld_data
      else
        case in_sigs.dir of
          down then          -- counting down
            if count_reg /= 0 then
               tmp := (count_reg - 1 as C_size)
            else
               tmp := max_count
            end || count <- count_reg
        | up then              -- counting up
            if count_reg /= max_count then
              tmp := (count_reg + 1 as C_size)
            else
              tmp := 0
            end || count <- count_reg
        end ; -- end case
        count_reg := tmp
      end -- end if
```

```
      end -- complete select H/S
    end -- loop end
  end
```

The program illustrates:

enumeration: the direction signal, *dir*, is declared as an enumerated type. Named values in an enumeration are implemented as numeric values starting at zero and incrementing by one from left to right. An enumerated type has an associated type which is the smallest numeric type that can hold all of the enumeration values. Here, *dir* can be represented by a single bit.

Enumerated values can be aliased, i.e. many names can be given to the same value. Consider the definition of a type defining the opcode a simple processor in which load and store instructions share the same opcode field in the instruction word. A possible type definition might be

```
type opcode is enumeration
  ADD, ADC, SUB, SBC, AND, OR, LD, ST=LD, BR
over 3 bits
```

Aliased names must have been previously defined. It is incorrect to write:

```
  ADD, ADC, SUB, SBC, AND, OR, LD=ST, ST, BR  -- incorrect
```

Note that enumerated types may be padded just as records by means of the *over* keyword. In the declaration specifiying the enumeration as being over 3 bits is redundant but is defensive programming giving the advantage compile time checking.

There is a minor problem with *case* statements and enumerated types. If the number of enumerated values defined is less than the maximum possible number that could be held by the underlying numeric type, then a *case* statement that evaluates that enumerated type will generate a non-fatal warning message because not all possible values of the underlying type will have been covered by the guards of the case statement. For example if variable *hue* has type *colour* which was declared:

```
type colour is enumeration
  red, blue, green
end
```

then the use of *hue* in the following *case* statement:

```
case hue of
  red then CmdA
| green then CmdB
| blue then CmdC
end
```

will, because only 3 of the possible 4 enumerations of the underlying type have be covered, result in the warning message:

```
warning: some case guard values are not covered
```

name space separation: Separate name spaces (variables, types, functions) are used by the Balsa system. Thus the declaration in the type declaration of *In_bundle*:

```
dir : dir
```

is valid and declares *dir* to be a field of the predefined type *dir.*

enclosed input selection: Use of the select statement has been described before. However, it is worth emphasing how advantage is taken of its communication-enclosed semantics. The

handshake is not completed until the matching *end* of the *select* statement. Thus multiple reads are possible without the need for communications to terminate in a variable. In this example, the program waits until the input bundle becomes valid. The various sub-fields of the bundle are read at different points during the passage through the loop body before the hanshake is completed at which point the environment is free to release the data. A minor restriction is that a field can only be read once within a *select* statement.

13 Efficiency Considerations.

In Balsa, statements instantiate hardware. A statement repeated will cause duplication of gates in the generated circuit. Sometimes this may be unavoidable, but often optimisations may be made by re-arranging the code. In the previous example (`count10d`), the statement which assigns the count register to the output channel is repeated in each of the branches of the *case* statement. It is possible to move the assignment to the end of the case statement as shown below:

```
procedure updown10 (input in_sigs: In_bundle; output count: C_size) is
local
  variable count_reg : C_size
  variable tmp : C_size
begin
  loop
    select in_sigs then
      if  in_sigs.load_l = load then
        count_reg := in_sigs.ld_data
      else
        case in_sigs.dir of
          down then          -- counting down
            if count_reg /= 0 then
               tmp := (count_reg - 1 as C_size)
            else
               tmp := max_count
            end
        | up then              -- counting up
            if count_reg /= max_count then
              tmp := (count_reg + 1 as C_size)
            else
              tmp := 0
            end
        end || count <- count_reg ; -- output assignment moved to here
        count_reg := tmp
      end -- end if
    end -- complete select H/S
  end -- loop end
end
```

The effect of this on the compiled circuit size can be determined by the utility *breezecost.* The programme as originally written has a cost[1] of 2588. The more efficient form above has a cost of 2270.

1. The cost reported is an approximate area given some unstated layout assumptions for an unspecified fabrication process. The actual figure for any given circuit will vary between releases of this software and is not an accurate estimate of the absolute area consumed.

14 Shared Procedures

There is one further optimisation that can be made: the program instantiates two adders: one for incrementing and one for decrementing. Since these two units are not used concurrently, area can be saved by sharing a single adder. The following program illustrates the use of shared procedures in Balsa:

```
-- count10f.balsa: an aysnchronous up/down loadable decade counter
-- uses shared adder
import [balsa.types.basic]
public

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
  ld_data : C_size ;
  load_l :  bit;
  dir : dir
end

type inc is 2 signed bits

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

   -- define shared procedure
   shared update_ctr is begin
     tmp := (count_reg + inc as C_size)
   end

begin
  loop
    select in_sigs then
    begin
      if  in_sigs.load_l = load then
        count_reg := in_sigs.ld_data
      else
        case in_sigs.dir of
          down then           -- counting down
            inc := ( -1 as inc);
            if count_reg /= 0 then
              update_ctr()
            else
              tmp := max_count
            end
        | up then              -- counting up
            inc := 1 ;
            if count_reg /= max_count then
```



```
              update_ctr()
            else
              tmp := 0
            end
          end || count <- count_reg ;
          count_reg := tmp
        end -- end if
      end
    end -- complete select H/S
  end -- loop end
end
```

Commentary on the Code

A local shared procedure, *update_ctr*, is defined. The procedure adds a local variable, *inc*, which is set to +1 for counting up and to -1 for counting down. The local variable is defined to be the minimum width necessary to hold the values -1 and +1: 2 signed bits. Note that it is necessary to perform an explicit cast to type *inc* when assigning the literal -1 to variable *inc*.

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

- shared procedures can not have any arguments
- shared procedures must not use local channels.

If a shared procedure uses elements of the channel referenced by a *select* statement, the procedure must be declared as local within the body of that *select* block.

Discussion

Running *breezecost* on the example `count10f` above shows a total cost of 2017: a saving of ~10%. Here the adder is only 4 bits wide, so saving an adder brings only minor returns. For larger size, structures, the savings can be siginificant. As an illustration, if *C_size* is defined to be 16 bits instead of the 4-bit nibble, the sizes of circuits `count10e` and `count10f` become 6902 and 5038 respectively.

Balsac is moderately clever in the type of adder that is instantiated. Savings can be made if the adder takes different length operands, a situation that occurs often when incrementing a full word by small integers. Defining the local variable *inc*, which as defined is limited to values -1 and +1, to be the same type as *C_size* increases the cost of the circuit from 5038 to 6242 in the case when *C_size* is 16 bits!

15 Arrays

Arrays are numerically indexed compositions of values of the same type. A good illustration of their use can be seen in the specification of a register bank in the example below:

```
-- regbank1.balsa (NB broken)
import [balsa.types.basic]

public
type  RegNo is 3 bits
type  RegData is 16 bits

type  RegCtrl is record
```

```
    Read0 : RegNo ;
    Read1 : RegNo ;
    Write : RegNo ;
    DoRead0 : bit ;-- if true read port 0
    DoRead1 : bit ;-- if true read port 1
    DoWrite : bit  -- if true write data
  end

  procedure RegBank (
    input RegCtrl : RegCtrl;
    input WritePort : RegData ;
    output ReadPort0, ReadPort1 : RegData
  ) is local
    variable RegBank : array 0..7 of RegData
  begin
    loop
      select RegCtrl then
        if RegCtrl.DoRead0 then ReadPort0 <- RegBank[RegCtrl.Read0] end ||
        if RegCtrl.DoRead1 then ReadPort1 <- RegBank[RegCtrl.Read1] end ||
        if RegCtrl.DoWrite then
          select WritePort then
            RegBank[RegCtrl.Write] := WritePort
          end
        end
      end
    end
  end
```

Commentary on the Code

This program synthesises a register bank containing 8 registers with two independent read ports and a single write port. The declaration of the register bank, illustrating the use of arrays is:

```
variable RegBank : array 0..7 of RegData
```

where *RegData* is a type determining the width of each register. The register bank is controlled by a bundle of signals, *RegCtrl,* defined in a record structure. The bundle includes the address of the registers to be read/written and 3 bits enabling the read/write operations. On receipt of the register control word, the data from the addressed registers are put on the output channels or read from the write port.

Unfortunately this program is flawed. As written, the read and write operations are allowed to take place concurrently leading to a potential conflict on a register. The compiler will issue a fatal error message to this effect. One solution is to separate the read and write operations as shown in the example `regbank2`:

```
procedure RegBank (
  input RegCtrl : RegCtrl;
  input WritePort : RegData ;
  output ReadPort0, ReadPort1 : RegData
) is local
  variable RegBank : array 0..7 of RegData
begin
  loop
    select RegCtrl then
      if RegCtrl.DoRead0 then ReadPort0 <- RegBank[RegCtrl.Read0] end ||
      if RegCtrl.DoRead1 then ReadPort1 <- RegBank[RegCtrl.Read1] end ;
      if RegCtrl.DoWrite then
        select WritePort then
```



```
          RegBank[RegCtrl.Write] := WritePort
        end
      end
    end
  end
end
```

Here, the two read operations are allowed to occur concurrently, but any write operation can only take place after any read operations have completed. This circuit, whilst correct, has explicit sequencing: the *read* control bits are tested before the *write* control bit. An alternative specification which eliminates this sequencing, and is cheaper, is given in example `regbank3`:

```
procedure RegBank (
  input RegCtrl : RegCtrl;
  input WritePort : RegData ;
  output ReadPort0, ReadPort1 : RegData
) is local
  variable RegBank : array 1..7 of RegData
begin
  loop
    select RegCtrl then
      if RegCtrl.DoWrite then
        select WritePort then
          RegBank[RegCtrl.Write] := WritePort
        end
      else
      if RegCtrl.DoRead0 then ReadPort0 <- RegBank[RegCtrl.Read0] end ||
      if RegCtrl.DoRead1 then ReadPort1 <- RegBank[RegCtrl.Read1] end
      end
    end
  end
end
```

Since reads and writes to a variable are constrained to be mutually exclusive, the test for a register access is moved to an *if … then … else* statement.

16 Multiplexing and Demultiplexing

16.1 An unbuffered data mixer

In a synchronous system, multiplexers are used to choose between two or more data streams on the basis of separate control or select inputs. In an asynchronous system a more common requirement is to merge/mix/multiplex two or more data streams on the basis of the validity of those data streams themselves. A Balsa description of such a multiplexer could be:

```
-- mux1.balsa: unbuffered MUX
import [balsa.types.basic]

public
procedure mux (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
  end
end
```

Commentary on the Code

If input *a* becomes valid, it is passed to the output c. Similarly, if input *b* becomes valid, it is transferred to output *c*. The program illustrates a number of important points:

variable-like channels: Normally, a channel has to be assigned to an internal variable in a procedure. However, within the body of a select statement, a channel can be treated as a variable. Here, channels *a* and *b* can be directly assigned to the output channel *c*.

unbuffered behaviour: Because local variables are avoided by use of the *select* construct, there is no bufferring in the multiplexer definition giving the expected behaviour for a multiplexer.

choice: This is the first example in which select has been used to choose between different inputs. Previously, *select* has been used solely for its enclosed handshake behaviour.

mutual exclusion: The behaviour of *select* is undefined if more than one input becomes valid at the same time. For correct behaviour of this circuit, the designer is assuming that mutual exclusion of the inputs is enforced by the environment in which the circuit is used. Often, this is not an onerous obligation to satisfy: the designer, by knowledge of the behaviour of the circuit in which the mux is to be used, will be able to guarantee mutual exclusion. However, if the sequencing of the inputs *a* and *b* is truly unpredictable, the *arbitrate* handshake component must be substituted to give an arbitrated choice

```
-- mux2.balsa: unbuffered arbitrated data mixer
import [balsa.types.basic]

public
procedure mux (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
  end
end
```

16.2 Demultiplexers

A demultiplexer takes an input data bundle and steers it to one of several output channels. A Balsa description of a simple 1-to-2 unbuffered demultiplexer is:

```
-- Dmux1.balsa: unbuffered 1-2 demultiplexer
import [balsa.types.basic]

type ibundle is record
  data: byte;
  sel : bit
end

public
procedure mux (input I:ibundle; output O1, O2 :byte) is
begin
  loop
    select I then
      if I.sel = 0
        then O1 <- I.data
        else O2 <- I.data
      end
```



```
      end
    end
  end
```

The input to the demultiplexer consists of a bundle of 8 bits together with a control bit which indicates the output channel to which the data is to be steered.

17 Tupling and Bit-Slicing

Balsa has operators for combining elements and for extracting or slicing elements from an array.

17.1 Tupling

```
-- combine1.balsa
import [balsa.types.basic]

type typeC is record
  a: byte ;
  b: nibble
end

public
procedure combine (input a: byte ; input b: nibble; output c : typeC) is
begin
  loop
    select a, b then
      c <- {a, b}
    end
  end
end
```

This simple example illustrates how tuples can be created by enclosing elements of the tuple by {}. The compiler must be able to infer the type of the resulting structure; there is only one commmon case where this is a problem: this arises when the tuple is cast.

```
-- combine2.balsa

import [balsa.types.basic]

type typeC is record
  a: byte ;
  b: nibble
end

public
procedure combine (input a: byte; input  b: nibble; output c : 12 bits) is
begin
  loop
    select a, b then
      c <- ( typeC {a, b} as 12 bits)
    end
  end
end
```

Here the output channel is declared as being 12 bits. The tuple has to be cast to this size so that the assigment to channel *c* is correctly typed and the compiler has to informed by as to the original type of the tuple by prefixing it with its type, *typeC.*

17.2 Array Concatenation Operations

Arrays [ref] can be concatenated in expressions using the "**@**" operator, for example

```
variable a, b : byte
variable c: array 4 of byte
variable d : array 6 of byte

c:= {a,b,a,b}      -- tuple construction
c:= {a,b} @ {a,b}  -- array concatenation
d:= c @ {a,b}
```

17.3 Array Slicing

Arbitrary bits fields can be extracted, or sliced, and cast into other sized types:

```
variable x : array 0 ..7 of 4 bits
variable y : array 0.. 1 of 4 bits

y:= x[1..2]        -- returns a slice of x of type array 0 .. 1 of 4 bits
d:= c[0..2] @ c[3..3] @ {a,b}
d:= c[0..2] @ {c[3]} @ {a,b}
```

Array slices always return array values which are based at index 0. The order of the indices in the range specification is immaterial. Thus: `x[1..2]` and `x[2..1]` are equivalent.

Here is an example from the instruction decoder of a simple toy microprocessor:

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

variable ICtrl is 8 signed bits  -- bottom 5 bits contain an immediate value
variable Imm16 : word

Im16:= (((ICtrl as array 0..7 of bit)[0..4] as Imm5) as Word)
```

The least-significant 5 bits of the variable *Ictrl* hold a signed immediate value.

```
(ICtrl as array 0..7 of bit)
```

casts the variable *Ictrl* into an array of bits from which an arbitrary subrange can be extracted or sliced:

```
(ICtrl as array 0..7 of bit)[0..4]
```

extracts the bottom (least significant) 5 bits from the variable *Ictrl.*

```
((ICtrl as array 0..7 of bit)[0..4] as Imm5)
```

casts the least significant 5 bits from the variable Ictrl and converts them into a signed number

```
Im16:= (((ICtrl as array 0..7 of bit)[0..4] as Imm5) as Word)
```

Finally, the 5-bit signed immediate field is sign-extended to become a 16-bit signed number.

18 Parameterised Procedures

18.1 A Variable-buffer definition

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 a library without knowing the depth of the pipeline when that part is instantiated. Example `pbuffer1` defines a single place buffer with a parameterised width:

```
-- pbuffer1.balsa - parameterised buffer example
import [balsa.types.basic]
public
-- single-place, parameterised-width buffer definition
procedure Buffer (
  parameter X : type ;
  input i : X;
  output o : X
) is
local variable x : X
begin
  loop
    i -> x ;
    o <- x
  end
end
```

Commentary on the Code

The buffer definition given earlier [see "A Single-Place Buffer" on page 2] is modified by the addition of the *parameter* declaration which defines X to be of type *type* - i.e. *X* is identified as being as being a type to be refined later rather than a value. 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.

Calling parameterised procedures

Once a parameterised procedure has been defined, it can be used by other procedures:

```
-- pbuffer1a.balsa - calling parameterised a procedure
import [balsa.types.basic]
import [pbuffer1]

public
-- instantiate a byte-wide single place buffer
procedure test (input a :byte ; output b : byte) is
begin
  Buffer over type byte of a,b
end
```

Note the syntax for instantiating the parameterised buffer:

```
<param-procedure-name> over <parameters> of <channels>
```

Renaming procedures

New names can be bound to procedures. This is particularly useful in the case of instances of parameterised procedures.

```
-- pbuffer1b.balsa - renaming a (parameterised) procedure
import [balsa.types.basic]
import [pbuffer1]

public
-- Give a new name for a byte-wide buffer
procedure BufferB is Buffer over type byte

-- instantiate a byte-wide single place buffer
procedure test ( input a :byte ; output b : byte) is
begin
  Buffer8 (a,b)
end
```

Here, the name *Buffer8* is bound to the instance of the parameterised procuedure *Buffer*. The familiar form of procedure call can then be used:

```
Buffer8 (a,b)
```

Actually, for compatibility with the syntax for instantiated parameterised procedures, the procedure calls above could be written:

```
Buffer8 of a, b
```

This notation is general and all procedure calls can use this form.

18.2 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 where the depth is also parameterised.

```
-- pbuffer2.balsa - parameterised pipeline example
import [balsa.types.basic]

public

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

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



```
begin
  if n = 1 then            -- single place pipeline
    buffer(i, o)
  | n >= 2 then
    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 Buffer0 is BufferN over 4, type byte
```

Introduced in the code are:

cardinal: this a type for representing parameters which should range over natural numbers.

print: the print statement enables messages to be reported via the compiler. the command has the form:

```
print <expression-list>
```

where `expression-list` consists of a list of manifest constants and may include one or more string literals. The first element of the list is expected to be, but does not have to be, of the type BalsaError which is a builtin enumeration type with elements “fatal”, “error”, “warning” and “report” indicating the severity the compiler should report on printing the remaining expressions. A “fatal” error will cause the compiler to exit immediately. An “error” error will stop the compiler generating breeze intermediate files, but parsing of the source file and further error reporting will continue. A “warning” error will generate a warning message, but code generation will continue. The “report” error will be logged to an error logfile. The expression-list may contain the name of a variable or parameter which evaluates to a constant value thus:

```
if param = 0 then
  print warning, "Parameter ", param, " is zero"
end
```

19 Recursion

19.1 An n-fold Handshake Generator

Some problems have a naturally recursive definition (and implementation). As an initial example consider a procedure that generates a fixed number handshakes for each call. A simple solution would use the *for* construct, but a more elegant (and less expensive) approach is to use the recursive approach below:

```
-- GenHS. A recursive procedure generating n Handshakes for each call
import [balsa.types.basic]
public

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
  |  n = 2 then
      sync o ; sync o
  else
    local
      shared doNext is begin
        Repeat over n/2 of (o)
      end
    begin
      if (n as bit) then     -- n is odd
        sync o
      end ;
      doNext () ; doNext ()
    end
  end
end

procedure Gen5 is Repeat over 5

procedure test (sync o) is
begin
  Gen5(o)  -- Generate 5 Handshakes
end
```

Commentary on the Code

The procedure *Repeat* takes a parameter *n* which is the number of handshakes to issue on the sync channel *o*. If *n* is odd, a preliminary handshake is issued. *Repeat* is then called twice, each time with a parameter half that of the previous *n*. The recursion terminates when *n* is 1 or 2.

19.2 Multiplying Handshakes

The handshake generator above can be employed in a handshake multiplier-circuit: for each input handshake, a fixed number of output handshakes are generated

```
-- MultHS: repeat: handshake on `o' `n' times for each input
import [balsa.types.basic]
import [GenHS]

public
procedure MultHS (parameter n : cardinal; sync i ; sync o ) is
begin
  loop
    select i then continue end ;
    Repeat over n of (o)
  end
end

-- Here is a x5 gnerator
procedure MultHS5 is MultHS over 5
```


19.3 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 2 n/2-way multipexers connected by internal channels to a 2-way multiplexer.

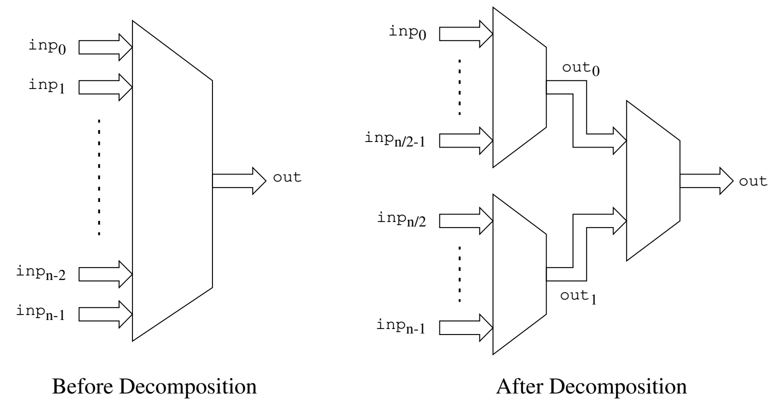

Figure 2: Decomposition of an n-way Multiplexer

```
-- Pmux1.balsa: A recursive parameterised MUX definition
import [balsa.types.basic]
public

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] -> inp then
         out <- inp
       end
     end
  |  n = 2 then
       loop select inp[0] -> inp then
               out <- inp
            |      inp[1] -> inp then
               out <- inp
            end
       end
  else
    local
      channel out0, out1 : X
      constant mid = n/2
```

```
        begin
          PMux over type X, mid of inp[0..mid-1],out0 ||
          PMux over type X, n-mid of inp[mid..n-1],out1 ||
          PMux over type X, 2 of {out0,out1},out
          end
    end
  end

  -- Here is a 5-way multiplexer
  procedure PMux5Byte is PMux over type 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. Ther recursive decomposition stops when the number of inputs is 2 or 1 (specification of a multiplexer with zero inputs generates an error).

20 Makefile Generation

Makefiles are commonly used in Unix to specify and control the processess by which complicated programs, or variants thereof, are compiled. Specifying the dependencies involved is often tedious and error prone. The Balsa system has a utility, *balsa-md*, to automatically generate the makefile for a given program. The generated makefile knows not only how to compile a Basla module with multiple imports, but also how to generate and run test-harnesses for the simulation environment used by Balsa – LARD.

20.1 Running balsa-md

usage

```
balsa-md {<switch>}* {<block/filename>}*
```

switches

```
-o <filename>      Name of target makefile (Default: Makefile)
-h or -?           Display list of switches
-t <procname>      Make a test harness gererating rule for procedure <procname>
-T <Basename>      Use <basename> as the name of the test-harness file
-U <variable>      Don't define <variable> in the generated file
-D <var> <value>   Define the variable <var> with value <value>
```

20.2 Using the makefile

The makefile which is generated contains several rules for producing different aspects of the compilation/simulation environment. The use of some of these rules is illustrated below

make: The default rule. The module is compiled and a LARD test-harness is produced if one was requested when generating the makefile **and** if the test-harness file does not already exist. Users can thus customise the test-harness to support their own requirements without fear that subsequent *makes* will overwrite and lose the customised test-harness.

make sim-pre: This rule compiles the test-harness (compiling any out-of-date balsa files) into runnable lard bcode. The LARD interpreter is not run.

make sim: This rule compiles the test-harness (compiling any out-of-date balsa files) into runnable LARD bcode and then calls the LARD interpreter to run the simulation.

make sim-win: This rule compiles the test-harness (compiling any out-of-date balsa files) into runnable LARD bcode and then calls the LARD interpreter to run the simulation in the windowed channel-viewer environment.

make-tidy: removes all derived files except for the test-harness files.

make-clean: removes all derived files **including** test-harness files. Use this option if you change the external interface to the top-level procedure for which you have generated a default test-harness file.

make depend: regenerates the makefile – you never need to type *balsa-md* again.

Examples

Consider the simple modulo-16 counter (count16a). A simple makefile can be generated by:

```
balsa-md count10a
```

Note that the *.balsa* extension to the source filename is not required. If the makefile is required to be able to generate a LARD test harness, the **-t** flag is required:

```
balsa-md-t count10 count10a
```

Note that `count10` is the name of the top-level procedure within the filename (`count10a.balsa`).

21 Simulation

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

1. Default LARD Test-Harness.

   If the -t option has been given to `balsa-md`, running `make sim` will generate a LARD test-harness and run it. The test-harness reads data from a file for each input port. of the module under test. Data sent to output channels appears on the standard output. This method needs no knowledge of LARD at all. Note that if the interface to procedure under test is changed, a new test-harness must be generated. By default, the makefile will not do this: 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 LARD test harness can then be generated for the Balsa tes-harness. Again no detailed knowledge of LARD is required.

3. Custom LARD Test Harness.

   For some applications, it may be necessary to write a custom test-harness in LARD. The makefile generated test-harness may be used as template.

21.1 Example Simulation 1

Assume the makefile for `count10a` [page 12] has been generated as shown above. Running `make sim` gives the following output:

```
0: chan 'aclk': sync
2: uninitialised variable read (0) 'V0_count_reg[0..3]'
2: uninitialised variable read (0) 'V0_count_reg[0..3]'
2: uninitialised variable read (0) 'V0_count_reg[0..3]'
3: chan 'count': reading 0
9: chan 'aclk': sync
12: chan 'count': reading 1
18: chan 'aclk': sync
21: chan 'count': reading 2
27: chan 'aclk': sync
30: chan 'count': reading 3
36: chan 'aclk': sync
39: chan 'count': reading 4
45: chan 'aclk': sync
48: chan 'count': reading 5
54: chan 'aclk': sync
57: chan 'count': reading 6
63: chan 'aclk': sync
66: chan 'count': reading 7
72: chan 'aclk': sync
75: chan 'count': reading 8
81: chan 'aclk': sync
84: chan 'count': reading 9
90: chan 'aclk': sync
93: chan 'count': reading 0
99: chan 'aclk': sync
102: chan 'count': reading 1
```

The simulation runs for ever until interrupted. The output *count* can be seen incrementing after each handshake on channel *aclk* and rolling over as expected from 9 to 0. The numbers on the left hand side represen LARD simulation time. In the current model, these figures are somewhat arbitrary. The simulation reveals that there is an initial communication on the output channel *count* before any input. This should not be surprising: the Balsa source specifies exactly that, but it may not be what was intended by the circuit designer. Furthermore, the trace announces (several times, alas) that variable `count_reg` (width 4 bits) is uninitialised. Again, this is obvious from the Balsa source; should this be a problem, explicit initialisation code must be added to the procedure. A weakness of the simulation environment is that the uninitialised state of `count_reg` is **not** reflected in the trace – output *count* is shown as being initially zero rather than undefined. Beware! – heed the uninitialised variable warnings.

21.2 Example Simulation 2

The test-harness can be coerced into reads from a input channel from a file by defining the variable TESTOPTS to be passed to the *breeze2lard* command. Consider the loadable up-down counter example `count10d.balsa` [page 16]. The makefile can be configured to generate a test-harness that associates files with input channels. For example

```
balsa-md -t updown10 -D TESTOPTS "-f in_sigs data" count10d
```


causes the file `data` to be assigned to the input channel `in_sigs`. The data contained in this file should be of the same type as the channel which in this example is a symbolic enumerated record. The contents of the file `data` are shown below

```
{8, load, up}           load the counter
{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
{0, count, down}        count down
{0, count, down}        count down
{0, count, down}        count down
{0, count, down}        count down
```

The format of the file is one value per line – anything that follws on the line is treated as a comment. The output from the LARD simulation Kgenerated by `make sim` is:

```
balsa-li test-harness.bcode balsa run
0: chan 'in_sigs': comment 'load the counter'
0: chan 'in_sigs': writing {8,load,up}
9: chan 'in_sigs': comment 'count to 9 '
9: chan 'in_sigs': writing {0,count,up}
13: chan 'count': reading 8
22: chan 'in_sigs': comment 'count & wrap to 0'
22: chan 'in_sigs': writing {0,count,up}
26: chan 'count': reading 9
35: chan 'in_sigs': comment 'count to 1'
35: chan 'in_sigs': writing {0,count,up}
39: chan 'count': reading 0
48: chan 'in_sigs': comment 'count down to 0'
48: chan 'in_sigs': writing {0,count,down}
52: chan 'count': reading 1
61: chan 'in_sigs': comment 'count down'
61: chan 'in_sigs': writing {0,count,down}
65: chan 'count': reading 0
74: chan 'in_sigs': comment 'count down'
74: chan 'in_sigs': writing {0,count,down}
78: chan 'count': reading 9
87: chan 'in_sigs': comment 'count down'
87: chan 'in_sigs': writing {0,count,down}
91: chan 'count': reading 8
100: chan 'in_sigs': comment 'count down'
100: chan 'in_sigs': writing {0,count,down}
104: chan 'count': reading 7
113: chan 'in_sigs': comment 'count down'
113: chan 'in_sigs': writing {0,count,down}
117: chan 'count': reading 6
Program terminated
```

The rest of the test sequence illustrates the counter counting up and wrapping round to zero, and then vounting down. Note how the symbolic enumerated values are passed to the simulation output along with the comments in the data file.

21.3 Balsa as a simulation language

The parameterised multiplexer [page 30] provides an example of how to use Balsa to create a simulation wrapper. The default test-harness which is generated for the component is not very satisfactory. Instead, a small Balsa program can be written to generate some simple test patterns. LARD is still used as the simulation language and environment, but it is not necessary to be able to write LARD programs. The code below does not pretend to be an exhaustive test of the multiplexer; it is however illustrative of the techniques employed in using Balsa to generate test harnesses.

```
-- test_pmux.balsa - A test-harness for Pmux5
import [balsa.types.basic]
import [pmux1]

public
procedure test (output out :byte) is
local
  type ttype is sizeof byte + 1 bits
  array 5 of channel inp : byte
  variable i : ttype
begin
  begin
    i:= 1;
    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
```

Commentary on the code

Balsa has a *while* construct which is used to generate a number of test vectors. Each pass of the loop is used to “walk” a “1” along the input variable. Balsa has no shift construct[1] and so the shifting is accomplished by setting the variable *i* to 1 and then doubling[2] *i* each pass around the loop. The loop variable *i* is declared as being 1 bit wider than the type of the input ports by using the *sizeof* operator in the type definition of *ttype*. If variable *i* were of type byte, the *while* loop would never terminate, because i would wrap around to zero.

Values are assigned to the 5 local channels sequentially. These channels are connected (by name) to the input channels of PMux5Byte. By operating the multiplexer concurrently, using the parallel compositon operator, every assignment to a channel causes the multiplexer to be activated. For the output of the multiplexer to be visible during simulation, its output channel is made a parameter of the test procedure.

22 Systolic Counters

Kees Van Berkel describes [ref] an elegant counter circuit possessing the property of constant response time and a constant upper bound on the power consumption regardless of the length

1. Resolving typing ambiguities at the language level is problematic.
2. The multiply operator is not yet implemented, so doubling is achieved by addition.

of the counter. The basic idea is to recursively divide the counter into a set of even or odd components.

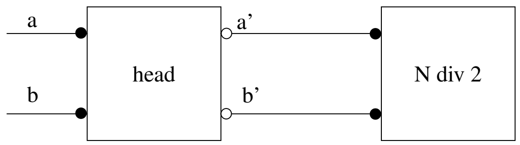

Figure 3: Counter Decompostion

The head cell effectively doubles each `a’` (or doubles +1) each communication of the N/2 counter and passes `b’` along `b` after N commications along `a`. The head cell is either a *Count-Odd* cell (CO) for odd N or a *Count-Even* cell (CE) for even N. The derivation of the cells is given in [ref]. Note that ports *a* and *b* are active ports whereas *a’* and *b’* are passive ports. The counter is “primed” by handshakes flowing from right to left. The head cell chooses between handshakes arriving on a’ and b’. However, the sequencing guarantees mutually exclusive use of the channels so that a non-arbitrated select construct may be used to implement the choice.

22.1 A systolic modulo-11 counter

11 can be written as:

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

A modulo-11 counter can therefore be constructed as:

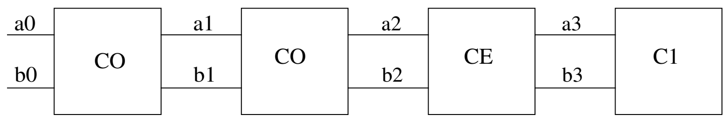

Figure 4: Modulo-11 Systolic Counter

A program for a modulo-11 counter in Balsa is:

```
-- Scount11a: systolic counter Kees van Berkel style
-- uses Balsa style select
-- non recursive, non parameterised flavour to expose all the wires

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

```
procedure c_even(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

procedure c_odd(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
end

procedure c_1(sync a, b) is
begin
  loop
    sync a; sync b
  end
end

procedure count11(sync a0, b0) is
local sync a1, b1, a2, b2 ,a3, b3
begin
  c_odd(a0, a1, b0, b1) ||
  c_odd(a1, a2, b1, b2) ||
  c_even(a2, a3, b2, b3) ||
  c_1(a3, b3)
end
```

22.2 Graphical Simulation of the modulo-11 counter.

The operation of the modulo-11 counter is best understood by using graphical viewer of the LARD simulation environment:

```
balsa-md -t count11 Scount11a
make sim-win
```

When the LARD channel viewing windows appear, run the simulation from the *Simulation* menu in the *li: test-harness.bode* window.. The Time View window is shown in Fig. 5. The red bars indicate asserted request signals and the green bars asserted acknowledge signals. The counter can clearly be seen to be counting by 11. The viewer displays all internal channles (including the activation channels). In general, the display is clearer if the LARD view is instructed to hide them

22.3 Problems with the Balsa-Style select.

The enclosed, non-buffered, semantics of the Balsa select statement leads to performance problems. This is not obvious from the `Scount11a` example above However, it is exposed by a

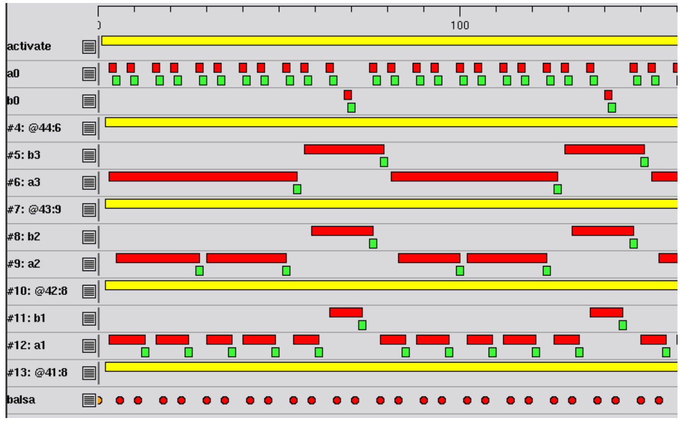

Figure 5: Lard Time-View for `Scount11a`

modulo-16 counter (`Scount16a`). Counting starts with an internal handshake at the far end, but no module completes the handshake on its right until the handshake on its left has completed. The result is a highly sequential mode of operation.. In Fig. 6 below, the activation channels have been removed from the display and for clarity the default channel names have been replaced by the channel names in the specification.

Figure 6: Scount16a (with Balsa-style Select)

22.4 Decoupling using a Tangram-style Select

Better perfomance can be obtained by using a Tangram style of select which incorporates an internal buffer thereby achieving a degree of decoupling between adjacent modules

```
-- Scount16b: systolic counter from Kees van Berkel.
-- uses Tangram style select
```

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

procedure c_even(sync a_left, a_right, b_left, b_right) is
local 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

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

procedure c_1(sync a, b) is
begin
  loop
    sync a; sync b
  end
end


procedure count16(sync a0, b0 ) is
local sync  a1, b1, a2, b2, a3, b3, a4, b4
begin
  c_even(a0, a1, b0, b1)||
  c_even(a1, a2, b1, b2) ||
  c_even(a2, a3, b2, b3) ||
  c_even(a3, a4, b3, b4) ||
  c_1(a4, b4)
end
```


The behaviour of the modulo-16 counter using a Tangram style select is shown in Fig. 7.

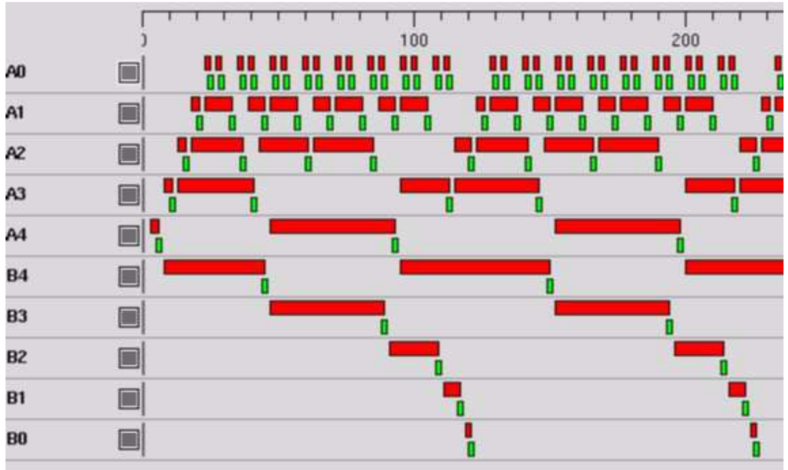

Figure 7: Scount16b (Tangram-style select)

As can be seen, much better performance can be obtained by exploiting the decoupling inherent in the Tangram style select. The cost (reported by *breezecost*) is however 310 compared to 191 for the Balsa-style select.

22.5 Count_Odd Behaviour for Balsa & Tangram Selects

Each odd cell initially handshakes on its left and then awaits a handshake on its right. Since that handshake is available, the select can immediately complete and there is no need to decouple that stage. A modulo-15 counter comprises only odd stages and is therefore a good vehicle for comparing the two types of select. The response of the two circuits is shown in Fig. 8: there is no significant different in the performance.

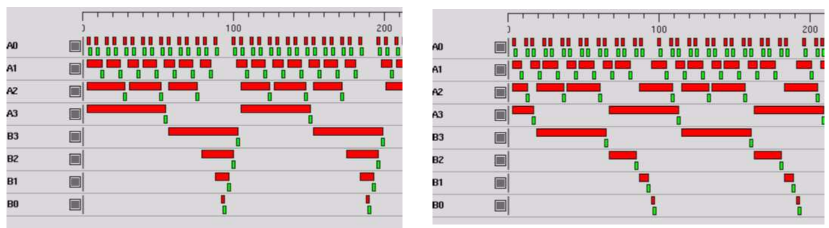

Figure 8: `scount15b` (Balsa-select left and Tangram-select right)

22.6 Mixed Select Counters

Since the relatively expensive Tangram style select is only required for the count_even module, a cost effective general counter can be constructed from count_odd modules using the Balsa-style select and count-even modules using the Tangram style select.

22.7 Recursive-Parameterised Version

It is rather tedious to specifiy arbitrary counters using the explicit composition techniques employed in the previous examples[1]. Balsa's parametisation and recursive features provide an elegant means of specifying arbitrary systolic counters. The following example uses a Tangram style select for the count-ven modules and a Balsa style select for the count-odd modules.

```
-- SRcountc: recursive, parameterised flavour systolic counter
-- uses Tangram style Select for even and Balsa Style for Odd
import [balsa.types.basic]
public

procedure countN (parameter N : cardinal ; sync a, b) is
local -- declare local procedures and channels

  procedure c_even(sync a_left, a_right, b_left, b_right) is
  local 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

  procedure c_odd(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
  end

  procedure c_1(sync a, b) is
  begin
    loop
      sync a; sync b
    end
  end
```

1. It is however easy to identify the channels connecting the various count_odd and count_even modules thereby easing the interpetation of LARD time-view plots


```
  -- declare two local channels
  sync a_int, b_int
begin
  if N = 0 then print error, "Parameter n should not be zero"
  |  N = 1 then c_1(a, b)
  else
    if (N as bit) then -- Odd
      c_odd(a, a_int, b, b_int)
    else
      c_even(a, a_int, b, b_int)
    end  ||   countN over N/2 of a_int, b_int
  end
end

; OK instantiate an arbitrary counter
procedure Ctest is countN over 53
```

23 Omissions & Future Developments

The current version of the manual is not complete: not all of the current Balsa language features have been described. A revised version will be issued in about 6 months. Balsa has been used in anger to generate the DMA controller for AMULET3I, however the tutorial does not cover any route from the netlist format to silicon.

The test-harness route and simulation environment is undergoing a period of rapid change. Future releases will greatly improve this area: a graphical project manager will allow multiple project configuarations and the current (painful) restriction of only one test harness per makefile will be removed. A prototype of the new project manager is available, but several of the bindings to the buttons are not yet operational although some of the functionality is available via the switches to the individual components of the balsa toolkit –-if you can work out how to use them!!.

Please report any bugs or infelicities in the manual to *balsa@cs.man.ac.uk*