Balsa – Demonstration Notes

Introduction

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 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!).

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 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.

Demonstration Notes

This demonstration will be run in conjunction with a powerpoint presentation. Mostly you will be invited to look at simple Balsa programs, use some of the supporting tools, make simple changes to existing programs etc. At the end of the each exercise, the presentation will show the output, or program that should have been obtained. For the last third of the session, you are invited to design a small stack-based calculator in Balsa and interface it to a 2 line x 20 character. LCD display. A simulator is available that directly interprets Balsa code and provides a model of the LCD display. When you have debugged your design, a mechanism has been provided to invoke a Xilinx backend remotely (in Manchester) and to return a Xilinx bit-file. This bit-file can be downloaded to one of 4 prototyping boards and run, thus demonstrating the complete Balsa to Hardware concept.

Even though Balsa is a small language, there is not time to demonstrate all its features, still less to elaborate on design styles and pitfalls. We hope you will gain a flavour of the current capabilites (and limitations) of the system . A complete language definition is appended to this back of this handout.

Obtaining Balsa

Balsa can be downloaded from w*ww.cs.man.ac.uk/amulet/projects/balsa/*
Queries about Balsa & bug reports should be emailed to *balsa@cs.man.ac.uk*

Exercise 1: A Single Place Buffer

Objective: understanding & compiling the simplest balsa program

1. Go to the directory *~/Balsa/Buffers*
2. Examine the file *buffer1a.balsa* using your favourite editor
3. Compile the buffer description:
   ```
   balsa-c buffer1a
   ```
4. List the files created (& examine them if really curious)

Exercise 2: A Two Place Buffer

Objective: use of parallel composition & modular compilation; viewing the handshake circuit graph.

1. Open file *buffer2c.balsa*
2. Edit the *buffer2c.balsa* to replace the '?'s by appropriate channel names.
3. Compile the description of the 2-place buffer:
   ```
   balsa-c buffer2c
   ```
4. Create a postscript plot of the handshake circuit graph & view it

```
breeze2ps buffer2c
gv buffer2c
```

5. Recompile the circuit with the hierarchy flattened & examine the handshake circuit:

```
balsa-c -f buffer2c
breeze2ps buffer2c
gv buffer2c.ps
```

Exercise 3: A modulo-16 Counter

Objective: understanding Balsa type enforcement; generating Makefiles; simple simulation.

1. Go to directory: *~/Balsa/Counters*
2. Open file: *count16a.balsa*
3. Generate a Makefile with test-harness rules

```
balsa-md -t count16 count16a
```

4. Examine the Makefile
5. Run the simulation (terminate with Ctrl-C)

```
make sim | more
```

6. Invoke the graphical simulation viewer

```
make sim-win
```

7. Start the simulation (Simulation ⇒ Run) and then stop it (Simulation ⇒ Stop)
8. Observe the timing of the request (red) and acknowledge (green) signals
9. The “blobs” on the viewer are cursor-sensitive: information is displayed on mouse-overs.
10. Close the simulation

Exercise 4: A Passive Input Counter

Objective: understanding Balsa’s select statement

1. Open the file *count16c.balsa*
2. Remove the results of the previous compilation and generate a new Makefile

```
make clean
balsa-md -t count16 count16c
```

3. Simulate the new description

```
make sim-win
```

4. Observe the impact of the enclosed handshake semantics of the select mechanism

Exercise 5: A modulo-10 Counter

Objective: introducing *if … then …else* statements; intoducing parallelism

1. Open the file *count10a.balsa.*
2. Modify the code so thaat the count value is output in parallel with the updating of the temporary register *tmp*.
3. Compile the new description and simulate the design.

```
make clean
balsa-md -t count10 count10a
make sim-win
```

4. Check that the counter does indeed wrap round.

Exercise 6: A Loadable Up-Down Counter

Objective: illustration of more Balsa data types; simulation using data in a file; cost estimation

In this exercise, we are going to supply a bundle comprising both data and control to the counter’s input port. The bundle consists of a bit indicating whether the counter should count up or down, a bit indicating whether the counter should count or load, and a 4-bit data value to be loaded (if a load is specified).

1. Open the file *count10d.balsa*
2. Now define a type *dir* as an enumeration

```
type dir is enumeration
   down, up
end
```

3. Now define a type *mode* as an enumeration of *load* & *count*
4. Now define the input bundle as a record structure

```
type In_bundle is record
   ld_data: C_size;
   ld_ctrl: mode;
   dir: dir
end
```

5. Note how the name space separation between variables and types allows the name *dir* to be overloaded
6. The counter description already tests to determine if a load or a count operation has been specified. Complete the main body of the program to test whether a count-up or a count down operation is to be executed. Another *if* clause could be used, but for variety, a *case* statement will be employed.
7. The body of the *else* clause is given below. Fill in the code for the case when the counter is counting up.

```
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
    -- fill in this part of the choice
    end || count <- count_reg
end ; -- end case statement
```

When simulating this design, it is convenient to specify a set of values for the bundle on the input channel *in_sigs*. The Makefile can be coerced into making this happen.

8. Compile the completed description & generate a test harness that will read test stimuli from a file – in this case arbitrarily called *data.* Inspect the contents of file *data* and then type:

```
make clean
balsa-md -t updown10 -D TESTOPTS "-f in_sigs data" count10d
make sim-win
```

9. The simulation stops rather ungraciously upon reaching the end of the data files. Ignore the error message. Observe how the symbolic enumerated values of the data bundle are shown on the output of the simulation trace along with comments in the data file.
10. Quit the simulation
11. A estimate of the cost, (based on some unstated layout assumptions for an unspecified backend technology) of a particular circuit can be obtained by

```
make cost
```

12. Make a note of the notional cost of this circuit.

Exercise 7: Sharing Hardware

Objective: reducing circuit cost at the description level.

Balsa statements instantiate hardware. A statement repeated will cause a duplication of gates. It is often possible to avoid duplication by re-arranging the code or by the use of shared procedures - similar to subroutines in a normal programming language. Example *count10f.balsa* illustrates both these approaches.

Firstly the output assignment `count <- count_reg`, which was duplicated in the two clauses of the `case` statement has been moved to the end of the case statement. Secondly, a shared procedure is introduced: *update_ctr* which handles both the increment and decrement operations. A local variable *inc* is introduced which takes the values -1 or +1.

1. Open the file *count10f.balsa* & browse the counter description
2. Compile the counter description & determine its cost

```
make clean
balsa-md count10f
make cost
```


3. Compare the cost with that obtained from the previous description.

There are a couple of minor restrictions in the use of shared procedures:

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

Exercise 8: Parameterised & Recursive Structures

Objective: defining a parameterised structure recursively; using balsa as a test language

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 or depth of the buffer. A more interesting example is provided by an n-wide multiplexer. The code is found in *pmux1.balsa*.

Balsa itself can be used to generate a test-harness for other Balsa-generated components. The file *test_pmux.balsa* is an example, generating values to each of the multiplexer input channels in turn.

1. change directory to *~/Balsa/Muxs*
2. generate a Makefile in the usual way & run the simulation (which will terminate in the usual unpleasant manner)

```
balsa-md -t test test_pmux
make sim-win
```

Exercise 9: Design of a Stack Based Calculator

Objective: exercising your Balsa skills; taking a design through from description to working hardware

In this exercise it is proposed that you build a stack-based calculator implementing basic operations such as push, add, subtract (or reverse subtract). This exercise will go all the way from Balsa description to actual hardware. A number of prototyping boards (containing a Xilinx FPGA and a 2 line x 20 character LCD isplay) are available to run the finished design. A special LARD test-harness has been constructed for you which communicates with a GTK widget modelling the LCD. The route from Balsa to a Xilinx configuration bit-stream (and encapsulation within a test-harness that represents the prototyping board) requires the use of proprietary Viewlogic and Xilinx software (not available under linux). A mechanism has therefore been provided whereby a debugged Balsa design can be submitted to Manchester for compilation returning a Xilinx download file.

1. Change directory to *~/Balsa/Calculator*
2. Design a parameterised stack. If this proves too tricky in the time available, a version can be found in *stack.balsa.*

3. Design a simple calculator. The specification is up to you, but to conserve resources on the Xilinx FPGA, the depth of the stack should not be more than 8. You may of course make life simple and dispense with the stack all together, but **do not change the specification of the channel interfaces to the calc procedure**. The test-harness (simulation and hardware) assume these interfaces.
4. Simulate your design using the special simulator:

```
balsa-calc-sim calc
```

5. When satisfied, submit the design for remote compilation. Details of how to do this will be available separately.
6. When compiled, download the design using *balsa-async-mgr*: a version of a widget used in the teaching laboratories at Manchester to interface to the prototyping boards.
7. Test your calculator on real hardware: be amazed, be very amazed!

LCD Memory Map

If you wish to be clever with the display, you will need to know how to address it. Constants are defined in the skeleton of the code in *calc.balsa* for moving the cursor, scrolling the display and for writing digits and characters. The memory map of the display is given below.

line 1	80	81	93	94	A7
line 2	C0	C1	D3	D4	E7
	visible window			ram outside visible display	

LCD Memory Map on Reset

The memory map wraps so that for cursor left or right movements, address 0xA7 is adjacent to address 0xC0 and address 0x80 is adjacent to 0xE7. Thus if the cursor were at address 0x80, a cursor shift left would take it to position 0xE7, outside the visible range of the display.

The figure below indicates the addressing after a scroll right. If the cursor were at address 0x80, a cursor shift left would again take it to address 0xE7, but this time the cursor would be visible as the 1st character of the 2nd line of the display.

line 1	A7	80	92	93	A6
line 2	E7	C0	D2	D3	E6
	visible window			ram outside visible display	

LCD Memory Map After Scroll-Right