SIMULATION AND ANALYSIS OF SYNTHESISED ASYNCHRONOUS CIRCUITS

L. Janin, A. Bardsley, D.A. Edwards

Department of Computer Science
The University of Manchester
Manchester M13 9PL, U.K.
{janinl, bardsley, doug}@cs.man.ac.uk

Abstract: Recent advances in automated synthesis tools for asynchronous circuits have made possible the design of large self-timed circuits. However, these new tools are still weak in their simulation and debugging capabilities because asynchronous circuits pose different challenges and opportunities in these areas from conventional clocked circuits. Balsa is such a tool intended for the synthesis of large asynchronous circuits. Two major additions to the Balsa asynchronous circuit synthesis system are presented in this paper: a simulation system and a visualisation system specifically addressing asynchronous circuit simulation. Both operate at the Handshake Component level extending Balsa with new debugging, profiling and validation capabilities whilst at the same time improving simulation speed by over four orders of magnitude compared with the previously used asynchronous modelling system. The new framework is evaluated on a Balsa-synthesised ARM-compatible asynchronous processor.

Keywords: Handshake Circuits, asynchronous circuits, simulation, synthesis, Balsa

1. INTRODUCTION

Balsa [Edwards and Bardsley, 2001] is one of the recently developed tools intended to automate the description and synthesis of large asynchronous circuits. It uses a Handshake Circuits methodology [Van Berkel, 1993] and has successfully contributed to the design of major asynchronous circuits such as a DMA controller for the Amulet3 asynchronous processor [Bardsley and Edwards, 2000] and SPA, an asynchronous ARM V5T compatible processor entirely described in Balsa, composed of ten thousand lines of code and synthesised into over a million transistors [Plana et al., 2002]. However, the Balsa framework lacks adequate native debugging and simulation tools. To avoid this problem, conventional tools are employed to simulate and debug the lower level netlists automatically generated by the Balsa synthesis. However, the netlist level is too far removed from the Balsa level for the tools to be convenient to use.

This paper describes a simulation and visualisation system operating at the Handshake Component level, available as an extension of the Balsa framework, and intended to help the designer of large asynchronous circuits by offering new debugging, profiling and validation capabilities, as well as improved simulation speed. The new simulation and visualisation architecture deals with the following characteristics of asynchronous systems:

- Asynchronous hardware systems exhibit a high degree of fine-grained concurrency implying the need for
  a) a fast simulation kernel,
  b) a detailed simulation trace for profiling,
  c) a scalable environment for debugging with a suitable visualisation technique for conceptualising the concurrency,
  d) validation at the source (Balsa) language level;
- Asynchronous circuits are often non-deterministic and suffer from complicated timing relationships between events and emergent timing behaviour. Precise timings are very difficult to obtain at this level of simulation, and a small timing difference can lead to a completely different order of execution of the components of the system;
- Deadlocks are hard to avoid, analyse and debug;
- Asynchronous modules connected by delay insensitive interfaces also offer opportunities for simplification of the simulation scheduler and indeed for distributed simulation, although this latter aspect is not considered further in this paper.

1.1. Handshake Circuits

Handshake Components are parameterisable components used as an implementation technology independent intermediate for synthesis in a similar manner to the EDIF LPM component set [EDIF]. Unlike the EDIF component set, each of the terminals of the

Handshake Components is accompanied by request/ acknowledge signalling to indicate when the data on that terminal is ready and when the data has been accepted by the party connected to that terminal. In this way Handshake Components communicate solely by taking part in data handshakes and are connected together solely by channels including this handshake signalling.

The use of cooperative handshaking in Handshake Circuits (compositions of Handshake Components) allows circuits to be built which do not require a global clock for internal synchronisation. A Handshake Circuit thereby presents a very flexible, modular interface to the world. Handshake Circuits can also be themselves partitioned into modular subcircuits (to map onto a number of ICs/FPGAs for instance) by separating the circuits' components into groups and using the channels connecting those groups as the interfaces between those groups. Figure 1 shows how two Handshake Components are connected by a channel. Here, the connection is between a Fetch component "T" and a Case component "@". The Fetch component presents a handshake request and data to the Case component using an 'active' port (with a filled circle), which the Case receives on a 'passive' port. Data follows the direction of the request in this example and the acknowledgment flows in the opposite direction. In this figure, individual, physical request/acknowledgement and data wires are explicitly shown. Data is carried on separate wires from the signalling (it is 'bundled' with the control) although this is not necessarily the case with other data/signalling encoding schemes. Normally, Handshake Circuit diagrams show the channel as a single arc where the control and direction can be discerned from the passive/active nature of the ports connected and an arrowhead on the arc indicates data direction.

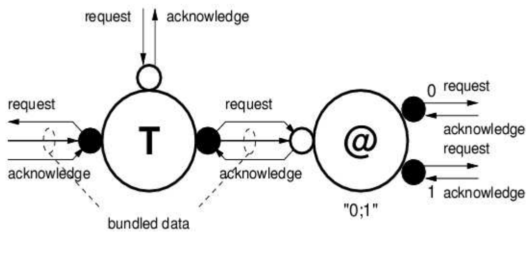

Figure 1: Two connected Handshake Components

1.2. Delay Insensitivity

Methodologies exist (DI codes, dual rail encoding, NULL Convention Logic) to implement channel connections with 'delay-insensitive' signalling where the timing relationships between individual wires of an implemented channel do not affect the functionality of the channel [Seitz, 1980; Rem, 1990; Martin, 1990; Fant and Brandt, 1997]. The methodologies can be used to implement Handshake Circuits which are robust to naive implementation, process variations and interconnect delay properties. In their interfaces to other asynchronous circuits, Handshake Circuits usually match data rates by the use of request/ acknowledge signalling avoiding potentially dangerous explicit synchronisation between clock domains.

Where asynchronous circuits are interfaced to synchronous circuits, the problems of synchronisation are no worse than between different synchronous clock domains.

1.3. Balsa

The aim of Balsa is to provide syntax-directed compilation without needing to optimise a flat netlist at the gate-level to produce suitably small and fast circuits. Improvement of the area/performance characteristics of a Balsa design is usually performed by modifying the Balsa description for a circuit and then testing the effect of that modification in simulation (design iteration). Syntax-directed compilation enables the designer to have a clear understanding of the effect of source description modifications on the implementation. This makes the process of design refinement (by rewriting the description) much simpler than for less transparent synthesis mechanisms.

1.4. History

The original simulation system for Balsa was based on LARD, the Language for Asynchronous Research and Development, which is a modelling language for asynchronous circuits [Endecott and Furber, 1994]. The Handshake Circuit generated from a Balsa description was transformed to a LARD program, which was in turn compiled and simulated. The first version of the LARD simulator was a language interpreter, leading to a simulation speed worse than that obtained during simulation of the post-synthesis layout. An improved version of the simulator was written later, where LARD was transformed into C code, compiled and executed. The simulation speed was about 64 times faster, but the successive transformations and compilations of the code lost much of the original structure of the Balsa description. This made the process of identifying erroneous Balsa code from simulation results a very difficult task.

Apart from simulation through LARD, Balsa designers have been extensively using conventional simulators in order to simulate the netlists generated by the Balsa synthesis. Unfortunately, this solution also makes difficult the process of mapping the detected errors onto the original Balsa source code, which results again in a tedious debugging stage. Balsa is now simulated directly at the Handshake Circuit level, four orders of magnitude faster than the original simulator, significantly faster than the post-synthesis simulators, and providing good debugging functionalities. In the new simulation flow described in this paper, LARD is no longer employed although it can still be useful for the description of some test harnesses.

2. ARCHITECTURE

The simulation and visualisation system described in this paper is implemented as an extension of the existing Balsa synthesis framework.

The global architecture of the Balsa development framework is shown in figure 2. The framework consists of a collection of software and scheme scripts communicating via files. The flow starts with a Balsa description, compiled into a Handshake Circuit description, and splits into two branches: the original synthesis branch and the new simulation and visualisation branch. Synthesis is used to transform a high-level Balsa description of an asynchronous VLSI circuit into a netlist of combinational logic, registers and asynchronous cells, as described in [Bardsley, 1998; Bardsley, 2000]. Simulation is used to debug, profile and validate a Balsa design at the Handshake Component level, and is described in the rest of this paper. This section introduces the different components of the simulation system and the flows of data running inside the simulation system and between it and the rest of the framework, together with the structures chosen to represent these flows.

2.1. Components Of The Simulation System

The simulation system is divided into two parts: the Handshake Circuit simulator, able to process low-level Handshake Communication at high speed, and the visualisation system, whose aim is to display both the simulator activity and some inferred results on top of an organised high-level graph representation of the Balsa circuit. The simulator is a single program whose inputs and output are described in the next subsections, and the visualisation system is expanded as two components: the Handshake Circuit visualisation, composed of the graph layout, navigation and animation processes and *GTKWave*, which is an external module used to display waveforms of the Handshake Channels [GTKWave].

Not represented on the diagram are two tools used to manipulate Balsa projects. Projects are used to group the different Balsa files describing a circuit together with the descriptions of the test harnesses. The first tool, balsa-md, generates a Makefile from a project description in order to automate the successive calls to the various Balsa scripts. The second tool, balsa-mgr, is a graphical IDE for Balsa, able to manipulate the project files and to give access to the Balsa programs through the interface.

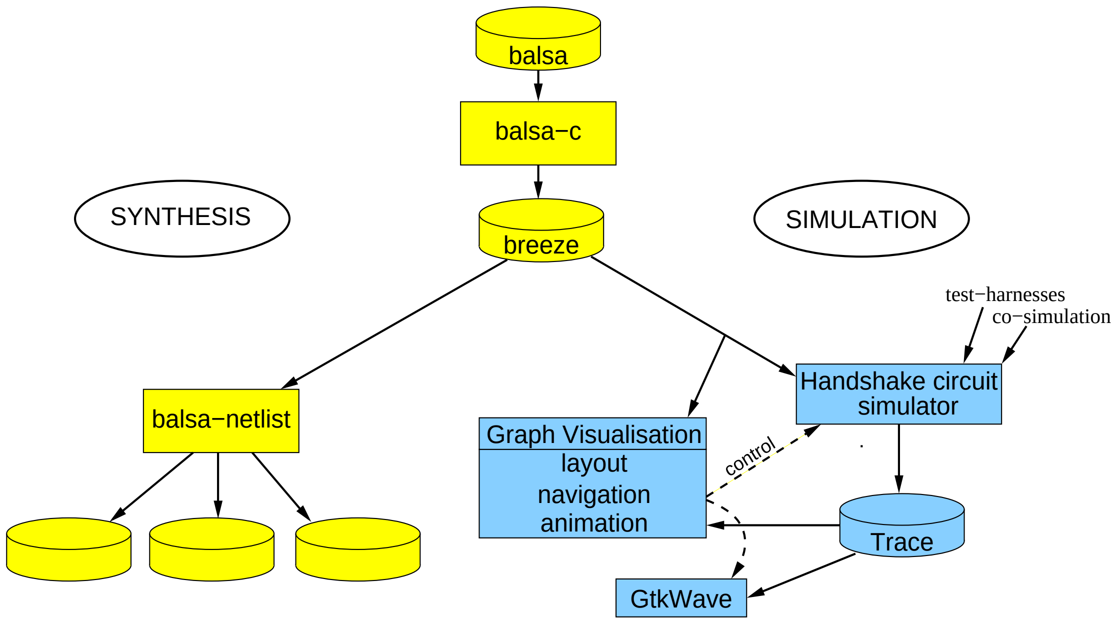

Figure 2: Organisation of the Balsa framework

2.2. Modelling Handshake Circuits

A Handshake Circuit is a collection of Handshake Components linked together by communication channels. It can be modelled as a graph whose edges and vertices are respectively its components and channels. This basic representation allows the use of standard graph layout software for the visualisation of Handshake Circuits. For example, the tool *dot* [AT&T Research] is used to produce a static view of a Handshake Circuit as a PostScript document.

Although the Handshake Circuit graph alone is sufficient for the synthesis process, it needs to be annotated to allow the debugging software to map the Handshake Circuit onto the original Balsa description, and so be able to report simulation faults with reference to the Balsa source code. The visualisation system can also take advantage of this extra information to show simultaneously the original source code and the compiled Handshake Circuit mapped onto it.

In order to map the Handshake Circuit onto the Balsa source code, a way of addressing this source code must be defined. A logical way is to use the Balsa file names and positions in the files. The compilation process is so transparent that before optimisation, a one-to-one correspondence exists between the Balsa constructs and the Handshake Communication channels [Bardsley, 2000]. It is then easy to make the correspondence between the Balsa description and the Handshake Circuit by associating a Balsa position information to each Handshake Channel.

This is simple and sufficient for error reporting. However, it does not take into account the Balsa structure as it is developed during compilation: A Balsa description where a non-shared procedure is called from two different places would be compiled into a circuit containing two sub-circuits corresponding to this procedure. In a situation like this, a visualisation system where the Handshake Circuit is mapped onto the Balsa description should obviously show the two instances of the procedure in an expanded form.

A method of extracting the structure of the Balsa description has been developed for the compiler, leading to a tree representation of the structure hierarchy. The combination of this tree with the graph representing the Handshake Circuit produces a clustered graph representation, which contains at the same time the information of the Handshake Components, the communicating channels and the Balsa hierarchy. All this information is used by the visualisation system to produce a detailed view for debugging the Balsa description and the generated circuit together.

2.3. Trace Format

The trace format is used to transfer information from the simulator to the visualisation system. The information required to be transferred between these two components comprises the channel and component activities as well as some timing information. On one hand, channel activity is reported as a sequence of events corresponding to the handshake protocol, and gives some information about the data and control flows inside the Handshake Circuit. On the other hand, component activity provides information for the estimation of the power consumption. Both channel and component traces are interleaved with some timing information in order to allow timing estimations.

At this level, the data and control flows are not distinguished, as they both are simulated as channel communications. The distinction between the different types of flow is done later by the visualisation or analysis process.

A critical aspect of this operation is the huge amount of data flowing through. Two orthogonal methods for reducing the amount of traced information are detailed in section 3.2.3: a time-based method based on checkpointing, and a space-based method based on the user control to select some regions of interest, limiting the traced data to these regions.

2.4. Test harnesses and co-simulation

The Balsa language was developed to describe only synthesisable structures. Consequently, accesses to computer resources such as reading data files cannot be done at the Balsa language level. In order to provide such accesses, the simulator integrates a test harness interface, giving the designer the possibility to connect the ports of their Balsa circuit to some external components. Two possibilities are offered for these external components:

- The designer can choose to use some of the special test harness components described as part of the simulator. Simulating these embedded components is efficient for speed and their integration in the visualisation system makes them easy to be debugged. Although these components' main role is to give access to files and a virtual console, another special component simulates the behaviour of a memory module, in the same way as memory generators are used for the generation of an efficient layout of the memory system. These components are described in section 3.3.3.
- The designer can describe their own components (asynchronous or not) in the language of their choice, and co-simulate the Balsa Handshake Circuit and the other description thanks to the co-sim-

ulation interface. At the time of writing, the only supported language for co-simulation is LARD, but it is planned to define a generic interface that can be used with any language.

3. HANDSHAKE CIRCUIT SIMULATOR

Design space exploration is performed in Balsa by making changes to the source Balsa language description. Design iteration is used to evaluate the effect of these changes. For this to be an effective technique, a simulator must be fast and reflect the speed and structure of the real circuit.

The new simulation system for Balsa has been developed around two axes:

- Design analysis: to provide the designer with relevant information for debugging and optimising his circuit
- Speed: necessary for practical design iteration and validation.

The choice of simulating at the Handshake Circuit level is explained first, followed by the description of the simulator itself, covering the solutions adopted to handle the various problems of deadlocks, non-determinism and data analysis.

3.1. Advantages of Simulating at the Handshake Component Level

Simulation of a Balsa description can be performed at several distinct levels of abstraction [Bardsley, 1998]:

- Language level behavioural simulation,
- Handshake Component simulation,
- Gate level simulation,
- Switch and analogue extracted layout based simulation.

The described Balsa simulator is working at the Handshake Component level. This is a good compromise between the direct simulation of the high-level Balsa description and the simulation of the synthesised netlist.

A language level behavioural simulation presents the advantage of providing an easy access to variable values and structures for inspection and debugging purposes. However, the simulation itself is not any easier at this level than at the Handshake Circuit level. On the contrary, the nested Balsa structures add a complexity which is not present in the flat structure of the Handshake Circuits: A Handshake Circuit can be modelled by a simple graph structure of Handshake Components where the component set is well defined and does not change for every modification in the language. This implies a simple structure for the Handshake Circuit simulator and avoids the need for too many simulator updates. On the contrary, the major drawback of a behavioural simulation at the Balsa level is that any change in the Balsa language requires an update of the simulator.

Technology independence is a useful factor for a Balsa language simulator: The handshake protocol and data encoding (the way the data and the request and acknowledge signals are encoded on wires) do not have to be chosen prior to the simulation. This is not the case for the simulation of Handshake Circuits and simulations at lower levels where the communication protocol between components and the data encoding have to be specified. However, knowing such properties of the circuit brings some useful information such as the possibility of visualising data and control flows as they would appear inside the real hardware circuit.

The lowest two simulation levels correspond to the simulation of the netlist generated from the Handshake Circuit by the synthesis tool (see figure 2) at the gate level or later as an extracted layout. Their main advantage is to provide more precise timing simulations as well as to enable estimations such as electromagnetic emissions, but at the cost of some additionnal simulation processing time. Gate level and layout simulators are already available as synchronous tools, and can be used with Balsa-synthesised circuits, although without any automatic way for the simulator to reference the Balsa source code for error reports or flow analysis, both useful for debugging purposes.

In summary, simulating at the Handshake Component level provides the following advantages:

- Fast simulation,
- Simple simulator (only 40 to 50 standard Handshake Components, linked together by easily simulated wires),
- Good possibilities of circuit analysis exploiting the data and control flows,
- One-to-one correspondence with the Balsa source code.

One may prefer simulating at a lower level (gates or layout) for the increased precision. As said previously, this is still possible through the use of conventional circuit simulation tools.

3.1.1. Choice of the handshake protocol

As the simulation of Handshake Circuits is technology dependent, the handshake protocol and data encoding have to be defined prior to the simulation.

The Balsa framework is constructed in such a way that different back-end technologies and implementation styles can be used for synthesis and thus the simulation framework must take account of these options where appropriate.

Data encoding defines how the request and acknowledge signals are mixed with the data to define how a Handshake Channel will be synthesised as a set of wires. The current implementation of the simulation scheduler authorises only a fixed number of request and acknowledge signals per channel, whatever the width (in bits) of the data is. In practice, this means that only single-rail data encodings can be simulated.

Both the 2-phase and 4-phase single-rail handshake protocols have been implemented as libraries of Handshake Components used by the simulator. The former provides a better speed as half as many events are flowing in the circuit. However, the later provides more information about the data flows thanks to its return-to-zero phase.

3.2. A Simulator Designed For Design Analysis

Design analysis is the main purpose of this simulation system. It allows the designer to debug and optimise the Balsa description of a circuit, and is basically divided into two activities: firstly, the debugging functionalities, dealing with the asynchronous problems of deadlocks and non-determinism, as well as with the analysis of the data and control flows; And secondly, the profiling functionalities, dealing with the estimation of time and power consumption inside the circuit.

3.2.1. Deadlocks

Without implicit global clock control, the control logic in asynchronous design is more complex than in the synchronous world since each module of the design needs hardware to perform synchronisation, to wait for data, and to trigger other modules when it has produced its data. The use of explicit communications between modules increases the risk of introducing deadlocks: distributed control through which a circle of unresolvable dependencies causes all activity to cease. This problem can be introduced by design errors. Ideally, deadlocks should be detected and then avoided at a very early stage in the design process. Unfortunately, current formal validation techniques [Barringer et al., 1996] cannot cope with large designs, hence the use of extensive simulation to give good confidence in the design functionality.

Handshake Circuits have only two ways of stopping their execution: the acknowledgment of the main control signal (reset) or a deadlock. The former indicates the successful completion of the simulation, and is characterised by the absence of any pending control signal in the circuit. A real circuit built in hardware and ending in such a manner would need to be reactivated before being able to operate again. The deadlock situation indicates that the activity has been stopped due to a missing acknowledge/request event. Unfortunately, this doesn't tell us if the missing event is due to a normal or to an erratic behaviour.

Different types of deadlocks must be distinguished, leading to different actions of the simulator:

- *Valid deadlock*. This deadlock arises when a circuit designed to run forever (think of a pipeline circuit for example) has processed all the available input data. The circuit has correctly sent a request on its input data port, but never received any answer, leading to the deadlock situation. This is the normal and only way for the circuit to finish when it has consumed every test vector. The simulator must then stop without indicating an error.
- *Error deadlock*. This type of deadlock is due to a real error in the Balsa description, and requires the simulator to stop and generate a complete enough description of the Handshake Component and channel states for debugging.
- *Error in co-simulation deadlock*. This is a high-level deadlock between two or more co-simulation systems. The problem is that each simulator has its own local view of the whole circuit, and thus cannot detect individually such deadlocks. A process tracking high-level communications is necessary.
- *Temporary deadlock*. Not really a deadlock, this situation arises when the external environment (test harnesses, or other simulators in the case of a co-simulation) is taking a very long simulation time to process its data, and thus appears to be dead from the point of view of the Balsa simulator. In this situation, the simulator should wait until an external event is available. This is not precisely a deadlock, as "temporary" indicates that the deadlock situation will be solved after an undefined period of time. However, the distinction between this type of deadlock and the "error in co-simulation" deadlock type is difficult to make in practice. This type of deadlock can be avoided if processes are able to indicate a minimal potential timestamp of their next event, as proposed by [Chandy and Misra, 1979].

The difficulty is to be able to detect the correct type of deadlock in any situation. The solution proposed here is to check some properties characterising the different types of deadlocks. Furthermore, simpler properties can be used when the interface ports of the simulated circuit are of certain (simple) types, namely in the case of a single simulation (not a co-simulation) or when the input ports are linked to fixed-length files.

The simplest case is a single simulation without any port other than the reset port. This special case of a circuit without port is trivial and any deadlock will be immediately reported as an Error deadlock.

A more interesting and very oftenly encountered case is the search for deadlocks through successive validation tests: The simulation consists of a single process with input ports linked to fixed-length files and output ports linked to specified-length files (or any other interface providing an immediate acknowledge to any incoming request event, in order to prevent any Temporary deadlock, and whose final expected size is provided). In this situation, only the Valid and Error deadlocks can occur and a quick detection of an Error deadlock can be done by negating the property "Valid deadlock => no input pending & output files completed".

The general case can be distinguish by the properties enunciated by [Chandy and Misra, 1979] as *Deadlock conditions*: A set of processes h in a network is said to be (Error) deadlocked at some stage of the computation if and only if

- *termination condition*: not all the processes in h have terminated and
- *executability condition*: no process in h is executable and
- *closure condition*: if h_i in h is waiting on edge e, and e is incident on h_j, then h_j is in h.

In the case of a co-simulation environment, the Error and Error in co-simulation deadlocks are distinguished respectively by one of the simulators and by a separate process tracking high-level communications between simulators.

3.2.2. Non-determinism

Because of non-determinism, repeated executions of the same asynchronous design for the same input may give different outputs.

In a simulation at the Handshake Component level, a repeatable behaviour can be enforced if the following conditions hold:

- Inputs from external environment are the same;
- Initial values for the different components of the system are the same;
- Individual processes are deterministic.

In Balsa, the only source of non-determinism is one particular Handshake Component called *Arbiter*, used to guarantee the mutual exclusion of two passive input channels' communications by passing a single communication at a time onto one of the two active output channels [Bardsley, 1998]. Arbiters are explicitly introduced by the Balsa "arbitrate ... end" statement. Designers always try to minimise the use of arbitration. However, the simulator should be able to handle the few cases where arbiters are required and non-determinism problems can appear.

The non-deterministic behaviour of the arbiter appears when both inputs' communications arrive at the same time or within a small time window. Here lies the absurdity of the situation: timing estimations of the simulation at the Handshake Component level are quite poor and far from what happens in the real circuit. Two requests arriving at the same time in an arbiter during the handshake circuit simulation – leading to a non-deterministic behaviour – would probably have arrived quietly one after the other on the real hardware. In the same way, but far more problematic: Two requests arriving at the same time on the real hardware circuit could arrive at different times in the simulator, avoiding the important detection of the non-deterministic situation. In order to detect such cases, the Arbiter component can work with *time windows*: When a communication is received on one of its inputs, the component waits for a possible request on its other input during a specified amount of time before being able to decide if its behaviour should be deterministic (one request received during the lapse of time) or non-deterministic (two received requests). The choice of the delay is critical: A too short delay would miss the detection of some non-deterministic situations, whereas a too long delay would lead to false detections of non-deterministic situations.

In the case of a non-deterministic situation, the behaviour of the arbiter has to be defined. A random behaviour, or different deterministic behaviours can be used: always choosing the first input, or always the second one; starting with the first input and switching; choosing the same input as the one received last, or alternating; etc. In the current implementation, this solution is used together with the checkpointing system. This allows the designer to rewind a simulation to the non-deterministic points and manually define the desired behaviour before restarting the simulation.

Another possible solution is to create two branches of the simulation every time a non-deterministic situation occurs. However, this method also requires a way to avoid the creation of 2^n simulation branches, which can be done by exploiting the fact that – most of the time – two branches differ only during a certain amount of time before going back to an identical state. This requires a way of analysing different branches fairly quickly in order to join them back when their execution becomes identical. This last idea has not been implemented, although it would be a way to reflect every possible behaviour of the VLSI circuit.

3.2.3. Data and Control Flows

Observing the flow of data and the state of control is useful for debugging, as it allows the designer to verify that data and threads of execution evolve correctly inside the circuit. Based on the trace format (section 2.3) which contains the history of control state changes, both flows are enclosed in the sequence of handshake signals – requests, acknowledgements and data transfers – and will be extracted and separated at the visualisation system level.

Two difficulties have been encountered: Firstly, the amount of information to be collected can be gigantic, generating some trace files of many gigabytes and slowing down the simulation by a couple of orders of magnitude. Secondly, the visualisation of so much information can be problematic. This second point is addressed in section 4.

A couple of simple ways for reducing the amount of traced information are used in this system: a time-based method based on checkpointing, and a space-based method based on the user control to select the desired channels. These methods are orthogonal, and have limitations making them insufficient for handling efficiently large designs. Checkpointing is limited in the actual framework to the simulation of a single Balsa description, and is thus unavailable for co-simulations. Letting the user choose which channels are important works well for designs smaller than a few hundreds of channels, but is impractical for larger designs. Current research is intended to ease the user's task by providing a graphical view of the channels mapped onto a clustered graph reflecting the structure of his Balsa description.

Observing the flow of data and the state of control can also be useful for optimising the circuit, since the designer can detect if some control flows are not finishing as early as expected, for example when they are waiting for useless return-to-zero phases of some components in a 4-phase protocol.

3.2.4. Profiling

Estimates of time and power at the Handshake Component level are available for Balsa simulations. These currently are imprecise results intended to help detecting performance bottlenecks inherent in an architecture and to provide a basis for circuit optimisation. Each component is assigned a specific duration and power consumption, and the visualisation system is able to integrate them over specified periods of time and specified areas, giving a rough estimation of the time and power consumption for the execution of specific actions inside the different parts of the circuit. The channels are deliberately ignored during the profiling process, as no information about the future length of the layout wires is known at this level. As for the data and control flows, the profiling information is exported from the simulator to the visualisation system via the trace file. In the current implementation, the timing and power information are fixed for each component, giving very imprecise results since, for example, the same component is used for both the *Add* and the *And* operations. This will probably be improved in a future version of the simulator. However, the estimations provided by this model are sufficient for detecting the main performance bottlenecks of a circuit.

3.3. A Simulator Designed For Speed

The previous section was dealing with design analysis, necessary for the debugging and optimisation phases of a circuit design. As explained in section 3.1, simulating at a lower level would provide the designer with more precise timing and power estimations, helpful for a better understanding of the circuit behaviour. However, this would come at the price of a slower simulation, and a simulator a few orders of magnitude faster can often counter-balance the lack of precision inherent to the high-level simulation of the Balsa Handshake Circuits. Moreover, a fast simulation is important for evaluating quickly a design during the process of design space exploration by iteration.

3.3.1. Component-oriented simulation

The Handshake Circuit simulator is based on a component-oriented world view, which is based on the discrete event scheduling model. Using this technique, a discrete event simulation is viewed as a collection of components that interact with each other by exchanging messages through communication ports. Besides components, the simulation contains a simulation engine that is responsible for synchronising components. An event-oriented view is adopted to model individual components, i.e., the component has one or more event handlers each of which performs corresponding actions upon the arrival of a certain type events. Components receive and schedule asynchronous events by pops and pushes onto an array of time queues modelling discrete times between now and now plus the (constant) maximum duration of a component activity.

3.3.2. Scheduler

This section describes the scheduler implemented in the simulator, and particularly the option taking advantage of the delay-insensitive (DI) nature of the Balsa circuits.

This is a standard event-based simulator working with time queues: Each Handshake Component is implemented as a set of possible handshake events (request and acknowledge for each port) which, when executed, pushes some messages onto the time queues in order to activate the following connected components at specific times.

Asynchronous circuits described in Balsa present the important property of being delay-insensitive (DI) when coupled with a DI protocol (see Introduction). The simulator assumes a single-rail protocol which is normally non-DI. However, the manner in which it is implemented ensures the DI property of the communications during the simulation: The data value always arrives in the destination component before the request or acknowledge event it is bundled with.

The advantage of DI circuits at the scheduler level is their ability to be executed “really asynchronously”: An activated component can wait as long as it wants before being processed without changing the behaviour of the circuit: It is the same situation as when the component’s activation wire takes a very long time to transmit the event, which is not a problem in a DI environment. An example is given in figure 3, where each parallel column of components can be executed independently to calculate the formula $\frac{-(x+1)}{2} + \frac{2}{x-1}$. However, an inattentive simulator would simulate the Handshake Components line by line, thus interleaving the execution of both columns. A better simulator (in the same way as a real person) would execute the first column, and then the second one, thus making a better use of the data locality (better use of the cache memory). In both cases, the Add component has to wait for its two inputs to acknowledge before being able to carry on its work.

In order to achieve this result, the scheduler can be simplified: The time queue necessary to execute line by line is not useful anymore, and the direct execution without time queue processes the column serially: Each component requests the data from the next component, and this request is directly processed in the order shown in figure 4.

This scheduler has been used successfully in conjunction with the 2-phase handshake protocol, doubling the simulation speed. Unfortunately, timestamps of the handshake events as ordered by this scheduler are in a different order than what they would be in a real execution of the circuit. Although this is not important for circuit validation, in a debugging context, out-of-order messages prevent the correct analysis of the different flows of data and control. As a consequence of this, this special scheduler is not used when debugging. It is only useful when speed is very important, as a 4 times improvement is obtained when coupled with the 2-phase handshake protocol, compared to the 4-phase protocol used with the normal scheduler.

A point not discussed yet is what happens in a case such as “x<-1 || x->var”, i.e. when a read and a write or two writes are occurring in parallel. The answer is simple: Balsa does not allow such constructs, which, if needed, must be described more precisely with the use of arbitration.

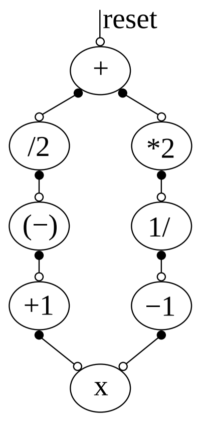

Figure 3: Handshake Circuit for $\frac{-(x+1)}{2} + \frac{2}{x-1}$

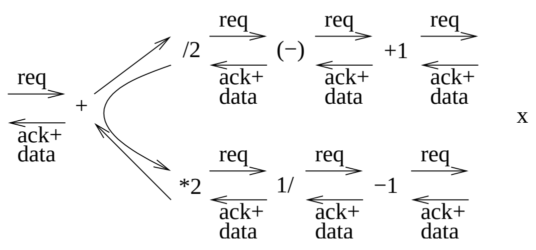

Figure 4: Execution order of the components of the Handshake Circuit in figure 3

3.3.3. Special test harness components

Test harnesses generally require access to the computer resources (output to screen or files, input from files) and thus can not be described with the Balsa HDL. For this reason, they were originally described using LARD, the language originally used at the University of Manchester for modelling the behaviour of asynchronous circuits.

Unfortunately, the synchronisations required to co-simulate the LARD and Balsa languages, added to the slow simulation speed of LARD were increasing the simulation time considerably. Furthermore, the simulations of LARD and Balsa were visualised by different software, making it difficult for the user to observe both of them together.

Some special test harness components have thus been designed closer to the Balsa level, available for the simulation without any loss of speed, and provided with a direct interface in the Balsa visualisation system. These components were originally specially integrated for the simulation of the SPA processor and provide read and write accesses to files and a console; a specific memory component simulates a configurable memory.

3.3.4. Co-simulation, Distributed simulation

Co-simulation is used for the moment only for the execution of test-harnesses described with the LARD language. The co-simulation interface will be extended in the future to allow other languages to be co-simulated with Balsa.

Another use of the co-simulation interface is to co-simulate multiple Balsa designs. Doing this on different computers is equivalent to running a distributed simulation. A couple of points in favour of the distributed simulation of Balsa are:

- The static structure of a Balsa-described design is easy to divide and distribute: The analysis of data and control flows gives a way to determine frontiers with the fewest communications;
- A scheduler similar to the one presented earlier (in section 3.3.2) can be used to reduce the need for time synchronisation between simulators;
- The automatic synchronisation of the components of asynchronous circuits makes them ideal for distributed simulations.

4. VISUALISATION

The process of understanding a program involves reverse engineering the source code [Rugaber 1992]. [Chikofsky and Cross II, 1990] give the following definition of reverse engineering: "Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.".

The visualisation system presented in this section is a reverse engineering tool used for program understanding. It extracts important information from three sources – the Balsa source description, the compiled Handshake Circuit, and the simulation trace – and builds a dynamic representation of the system in order to help designers debug their circuits.

The visualisation system is composed of three main modules working with Handshake Circuits, plus a couple of components for extra visualisation at other levels:

- The layout module, able to organise Handshake Components and groups thereof in a big picture, hopefully with short and non-overlapping communication channels, mapped onto a clustered graph reflecting the structure of the Balsa description;
- The navigation module, key of the scalability of the system;
- The animation module, able to paint components and communication channels into different colours, according to their state and activity, in order to visualise data and control flows;
- An interface for test-harness components;
- GTKWave, an external module used to display waveforms of the Handshake Channels.

Spaced-based visualisation is provided by the layout and navigation modules, whereas time-based visualisation is available through the animation module.

An illustration of the visualisation system is given in figure 5, showing a global view of the Balsa description of SPA microprocessor. The next section describes the layout, navigation and animation views that allow such a complex representation be of practical use to a designer.

4.1. Layout

The aim of the layout module is to show the state of the whole Handshake Circuit at a given time on one picture. This serves two roles: As a still image, it shows the overall design of the circuit and as an animated image, it will be used by the animation module described later.

The layout module is based on the following requirement: To be able to organise every Handshake Component on a plane in a readable manner. The term “in a readable manner” is translated here to “minimising the size and the overlapping wires (communication channels)”, and is simplified in the current research to “minimising the size of the communication channels”. Another idea taken into account is to try to reflect the structure of the Balsa description on the visualised circuit, i.e. components declared close to each other in the Balsa description should appear close to each other on the visualised picture of the circuit.

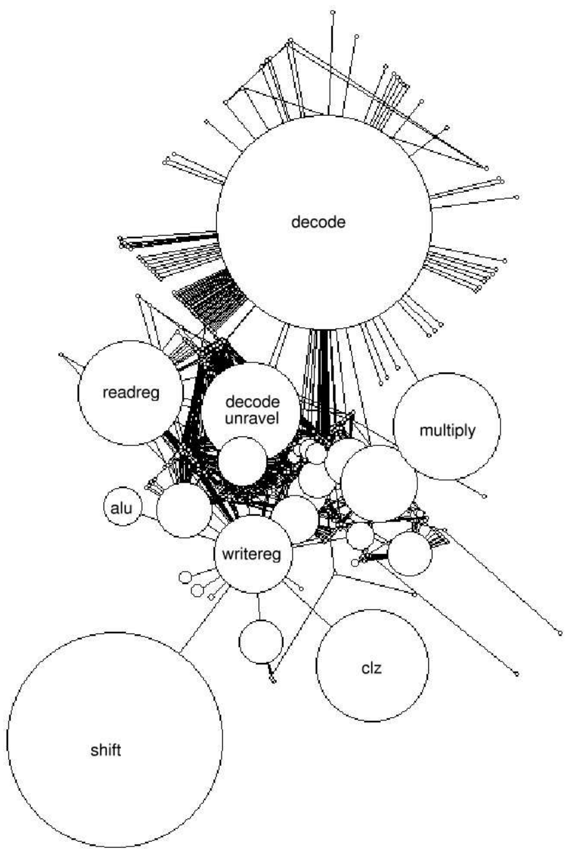

Figure 5: Global view of a circuit structure (The SPA microprocessor core)

The chosen technique for organising the Handshake Components is based on a force-directed layout system [Di Battista, Eades et al., 1998], where each component or group of components is described as a particle with a size. These simple rules apply: If two particles are too close to each other (i.e. the distance between them is less than the sum of their sizes), they repulse each other; otherwise, if the two particles are linked (components linked by a channel for Handshake Communication), they attract each other. If the two particles are far from each other and not linked, they do not interact.

At the same time, the layout of the graph of the Handshake Circuit is mapped onto the clustered graph reflecting the structure of the Balsa description. This allows low-level information such as data flows to be visualised on the high-level Balsa structure. Furthermore, the reduction/expansion of clusters provides a means to display the circuit at different levels of abstraction.

4.2. Navigation

The aim of the navigation module is to show precisely any chosen part of the circuit, and to navigate inside the circuit quickly and intuitively. This serves the debugging functionalities, and is here to help the designer locate quickly and accurately the faulty parts of a design.

The navigation module is the key for a scalable visualisation system: Firstly, it interacts with the layout module to provide functionalities such as zoom, pan, and manipulation of clusters, necessary for the visualisation of large designs. Secondly, it provides the possibility to select intuitively regions of interest, a necessary action to reduce the amount of traced information (see section 3.2.3). It is also used to select regions for profiling.

4.3. Animation

Once the components of the circuit are organised in an easily readable way, a representation of the circuit structure is available. Based on the trace format described in section 2.3, the role of the animation module is to add further information to the static picture in order to represent the data and control flows, and the changing activity of the components during the simulation.

This is done here by representing each component or channel state by a colour, and animating the colours as the simulation system updates the state of the components and channels.

The advantage of such an animation system is its ability to show all the information available from the Balsa description and from the execution of the simulation of the system, and then let the user decide what he wants to focus on. Debugging is made easier through the visualisation of the parallel activity: Every thread of execution of a simulation can be shown simultaneously, and the observer can focus on one specific thread, observe its activity, and can easily observe its merging with another thread or its splitting into two threads. Moreover, every thread is ensured not to overlap with any other in the visualisation area, whereas they often overlap on a source file description.

This animation system also provides some interesting debugging features for deadlocks and livelocks. When a deadlock situation arises, the program stops, leaving the guilty components in a specific colour and the trace of the components before them in another colour, making it less difficult to debug. In a livelock situation, the colours can be observed circling in an endless loop.

Furthermore, the one-to-one correspondence between the Balsa description and the visualised Handshake Components makes it easy to link any error located on the visualised circuit with its corresponding location in the Balsa description.

4.4. Other visualisation tools

4.4.1. Test harness components

An interface is provided for visualising the activity of the test harness components. Input from files and output to files and console are displayed together with the channel activity. The special test harness Memory component gets a more complex interface where it is possible to visualise and edit the contents of the simulated memory.

4.4.2. GTKWave

GTKWave is an external module used to display waveforms of the Handshake Channels. It is directly linked to the Handshake Circuit visualisation system, which provides the user interface to select which channels are to be displayed in GTKWave. In return, GTKWave is used to select the periods of time over which the power consumption needs to be estimated by the profiling module.

5. CONCLUSIONS

The simulation and debugging of asynchronous circuits bring some new possibilities of optimisation to the general event-driven simulation of VLSI circuits, but asynchronous circuits also require some new features for being debugged efficiently. This paper has described a new simulation system and visualisation system aimed at easing the design of such circuits.

The simulator described here shows some good results concerning scalability and simulation speed thanks to the use of simple Handshake Circuits. The debugging and visualisation aspects of the described framework take advantage of the one-to-one correspondence between the high-level description language Balsa and the Handshake Circuits in order to provide an efficient debugging interface for handling non-determinism problems, data and control flow analysis and deadlock detection.

Profiling at the Handshake Circuit level, based on fixed values of time and consumption assigned to each type of component, and disregarding the channel activity, is rather imprecise. However, it is sufficient for detecting the main performance bottlenecks inside a circuit.

Further work includes the co-simulation of Balsa with other languages, and a basic interface is already in place for co-simulating Balsa with the LARD asynchronous behavioural language.

ACKNOWLEDGEMENTS

This research is supported by the EPSRC Grant GR/N19618.

REFERENCES

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

Bardsley A. 1998, “Balsa: An Asynchronous Circuit Synthesis System”. *M.Phil. Thesis*. The University of Manchester.

Bardsley A. 2000, “Implementing Balsa Handshake Circuits”. *Ph.D. Thesis*. The University of Manchester.

Bardsley A. and Edwards D.A. 2000, “Synthesising an asynchronous DMA controller with Balsa”. In *Journal of Systems Architecture*, 46. Pp1309-1319.

Barringer H., Fellows D., Gough G.D., Jinks P., Marsden B. and Williams A. 1996. “Design and Simulation in Rainbow: A framework for Asynchronous Micropipeline Circuits”. In *Proceeding of the European Simulation Symposium*. Genoa, Italy.

Chandy K.M. and Misra J. 1979, “Deadlock Absence Proof for Networks of Communicating Processes”. In *Information Processing Letters*, 9, 4, November 1979, Pp185-189.

Chikofsky E.J. and Cross II J.H. 1990. “Reverse engineering and design recovery: A taxonomy”. In *IEEE Software*. Volume 7(1), Pp13-17.

Di Battista G., Eades P., Tamassia R. and Tollis I. 1998, *Graph Drawing: Algorithms for Geometric Representations of Graphs*. Prentice-Hall.

EDIF Library of Parameterized Modules. URL. *http://www.edif.org/lpmweb/*.

Edwards D.A. and Bardsley A. 2001, “Balsa - An Asynchronous Hardware Synthesis System”. In *Principles of Asynchronous Circuit Design*. Pp153-218. ISBN 0-7923-7613-7.

Endecott P.B. and Furber S.B. 1994, “Modelling and simulation of asynchronous systems using the LARD hardware description language”. In *Proceedings of the 12th European Simulation Multiconference*. Manchester, Society for Computer Simulation International. Pp39-43.

Fant K.M. and Brandt S.A. 1997, *NULL Convention Logic Tech. Report*, Theseus Logic Inc. 140, 485 N. Keller Rd. Maitland, FL 32751.

GTKWave. URL: http://www.cs.man.ac.uk/amulet/tools/gtkwave/.

Martin A.J. 1990, “The Limitations to Delay-Insensitivity in Asynchronous Circuits”. In *6th MIT Conference on Advanced Research in VLSI Processes*.

Plana L.A., Riocreux P.A., Bainbridge W.J., Bardsley A., Garside J.D. and Temple S. 2002, “SPA – A Synthesisable Amulet Core for Smartcard Applications”. In *Proceedings of Async’2002*. Manchester, UK.

Rem M. 1990, “The Nature of Delay Insensitive Computing”. In *Higher Order Workshop*, Banff.

Rugaber S. 1992, “Program Comprehension for Reverse Engineering”. In *Proceedings of the AAAI Workshop on AI and Automated Program Understanding*. San Jose, California.

Seitz C.L., 1980. “System Timing”. In *Introduction to VLSI Systems*, C. Mead and L. Conway, eds. Addison Wesley, chapter 7. ISBN 0-201-04358-0.

Van Berkel K. 1993, “Handshake Circuits – an Asynchronous Architecture for VLSI Programming”. In *International Series on Parallel Computers.* Volume 5. Cambridge University Press.

BIOGRAPHIES

Lilian Janin is a PhD student in the Computer Science Department at the University of Manchester. He received his M.Sc. in Advanced Computer Science at the University of Manchester and his degree of Engineer from the Institut d’Informatique d’Entreprise, France. His current research interests are in simulation and visualisation of large asynchronous systems.

Dr Andrew Bardsley received the degrees of B.Sc. (Computer Engineering, 1996), M.Phil and Ph.D (Computer Science, 1998, 2000) from the University of Manchester. Since then he has worked as a researcher in the Amulet group, working on the Balsa synthesis system.

Dr Doug Edwards is a Senior Lecturer in Computer Science. He received a B.Sc. in Physics and Electronic Engineering and M.Sc. and Ph.D. degrees from the University of Manchester. His current research interests are in synthesis and simulation of large asynchronous systems. Previous work has been in the areas of Computer Networking, routing algorithms and hardware routing engines for PCB layout and formal verification of hardware.