Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language

Philip Endecott, Stephen Furber,
Dept. of Computer Science,
University of Manchester,
Oxford Road,
Manchester.
M13 9PL U.K.
pbe@cs.man.ac.uk

Abstract

LARD is a hardware description language which uses CSP-like channel communication to describe the behaviour of asynchronous VLSI systems. This communication abstraction makes LARD a much more productive language for this type of modelling than conventional languages such as VHDL. LARD simulations can be useful for debugging, for performance analysis, and for validating the behavioural model against a corresponding gate-level schematic. The latter requires co-simulation if the model is non-deterministic. The LARD toolkit has been implemented in a very flexible fashion and is readily adapted to other tasks.

1. Introduction

The development of ever more complex VLSI circuits has been underpinned by the existence of suitable simulation. Since a whole chip must be fabricated in one go the designer must rely entirely on simulation results while he designs, debugs, evaluates and refines his work.

For many years mainstream VLSI design has been dominated by the synchronous paradigm where the timing of the whole system is controlled by a single global clock. This approach has been very successful as it makes it easy to reason about the behaviour of the system; timing can always be described in whole numbers of clock cycles.

Unfortunately a number of factors are coming together to make globally synchronous design less attractive in the long term. These factors include:

- As chip sizes and clock speeds increase, signals will take more than one clock cycle to get from one part of a chip to another. We are already seeing chips that use several clocks for distinct sub-blocks. The Semiconductor Industry Association projects that chips will have over 10 000 non-overlapping time zones by around 2012 (SIA 1997).
- High frequency clock signals cause significant electromagnetic compatibility problems. This is particularly worrying in important applications such as mobile telephony, where it makes the goal of a single-chip phone hard to achieve.
- In many applications low power consumption is a goal. The global nature of a clock signal means that logic everywhere operates continuously, using power on every clock cycle even when there is no work to do.

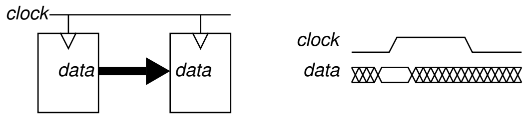
As a result of these factors, asynchronous (or self-timed) design, for many years not much more than an academic curiosity, has seen a resurgence of interest. In an asynchronous VLSI system, the timing at interfaces between blocks is controlled by a local handshaking protocol with communications occurring when both sides are ready, rather than under the control of a global timing signal.

The AMULET group at the University of Manchester has made a number of contributions to the re-establishment of asynchronous logic. The AMULET1 microprocessor (Woods et al. 1997), developed between 1991 and 94, runs standard ARM code and as such was the first asynchronous implementation of a commercial instruction set. AMULET2 (Furber et al. 1997) took this proof of concept and developed a more useful chip with increased performance. AMULET3 (Gilbert and Garside 1997) is currently under development; this chip is a complete asynchronous embedded system with an ARM compatible processor core, RAM, ROM, a DMA controller, an on-chip bus, an external memory interface and a synchronous peripheral subsystem. This chip has been designed with telecommunications applications in mind and we have hopes that it will find commercial exploitation.

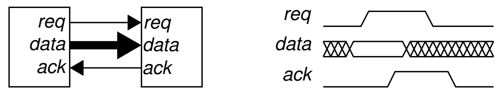
Despite its apparent advantages the further development of asynchronous logic faces several hurdles. One such hurdle is the lack of modelling languages and related tools, and as was noted earlier tools such as simulators are crucial for complex VLSI designs. Many of the tools in use today have been developed by people with a synchronous mind set, and description and simulation of asynchronous systems using them can be awkward.

For the AMULET1 and AMULET2 designs we used a conventional modelling language and simulator, and found progress difficult. We identified this as a major obstacle to the progress of the project and decided to develop a new high-level modelling language and simulation environment for AMULET3. This paper will describe the resulting language, which is called LARD (Language for Asynchronous Research and Development). Developing the initial model of AMULET3 took an order of magnitude less time than had been the case for the earlier designs, thanks to the more appropriate semantics of the new language; we count this as a substantial success.

This paper starts by explaining how conventional hardware description languages such as VHDL are inefficient for modelling asynchronous systems, and describes the main features of LARD that addresses this deficiency. We then look at how LARD simulations can help to analyse the functional correctness and performance of the design, and how co-simulation between the LARD simulator and a transistor level simulator can help to validate a circuit implementation against the behavioural model.



(a) Synchronous communication



(b) Asynchronous communication

Figure 1: Synchronous and Asynchronous Communication Protocols

Finally we look at the way in which the LARD simulation toolkit has been implemented, and show how the flexibility of its implementation makes it applicable to a wider range of applications than just VLSI design.

2. Communication Abstractions in LARD

Conventional hardware description languages such as VHDL model communications between blocks using signals. These signals have semantics similar to shared variables.

In asynchronous systems, timing information about the validity of data signals is provided by local timing signals, often in the form of request and acknowledge signals. Figure 1 compares the synchronous timing protocol with one possible asynchronous protocol. To model the asynchronous protocol using VHDL's signals we must explicitly describe the behaviour of the data, request and acknowledge signals.

Although an explicit description of the protocol may be appropriate later in the design process, for initial modelling we would prefer to see the communication as an abstract atomic action. The channel communication primitives of CSP (Hoare 1978) and occam are an ideal model and have been used in the proprietary asynchronous hardware description language Tangram from Philips (van Berkel et al. 1991). A slight variation on this approach has been adopted in LARD. The following listings illustrate how VHDL (left) and LARD (right) can be used to describe a simple block that reads from its input I and sends a function of the value read to its output O.

```
process begin
  wait_until I_req=1;
  v:=I_data;
  I_ack<=1;
  wait_until I_req=0;
  I_ack<=0;
  O_data<=f(v);
  O_req<=1;
  wait_until O_ack=1;
  O_req<=0;
  wait_until O_ack=0;
end process;
```

```
forever(
  I?(
    v:=?I
  );
  O!f(v)
)
```

Some of the complexity of the VHDL code shown here could be hidden by writing subroutines to send and receive on a channel. However this becomes difficult when we try to perform two communications concurrently as the actions have to be interleaved, and the fact that the request and acknowledge ports go in opposite directions means that they cannot be grouped together in a single record type. More fundamentally the underlying simulation would still operate in terms of the signalling protocol which would be visible in trace displays and so on at run time.

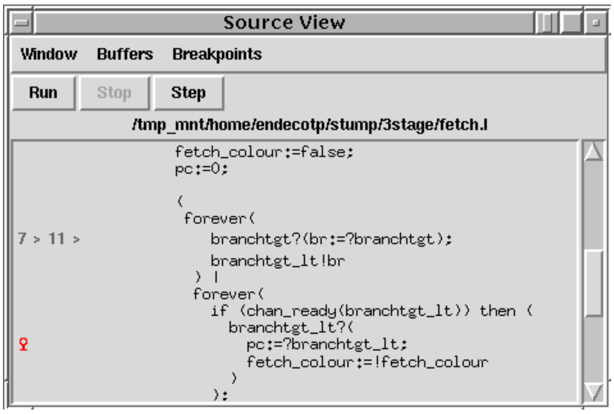


Figure 2: Screenshot of Source View window

This communication abstraction is the main feature that makes LARD suitable for describing asynchronous VLSI systems. In addition LARD has fine-grained concurrency, with statements composed either sequentially with `;` or concurrently with `|`. The other features of the language are similar to other high level languages: subroutines, hierarchical scope, a complex type system, libraries and so on. With the AMULET3 model taking 10 000 lines of code these structured programming features are essential.

3. The Role of Simulation in Asynchronous Design

The relatively abstract nature of LARD allows us to model our designs at an earlier stage of the design process than had been the case with other description languages. In the past, by the time we had developed a model that could be used to give feedback about the chosen architecture, it was almost too late to consider changes. Now we are able to get feedback much more quickly; in fact the AMULET3 model took an order of magnitude less time to develop to the point where it could run ARM programs than the AMULET2 model did. With these more abstract models, what useful questions could we answer using simulation?

Functional correctness

The first question that the designer will ask is "does it work?". Much of the code in the AMULET3 model is purely functional and uninteresting from the theoretical point of view, but as with any design it is possible to inadvertently introduce errors. It is therefore essential that the tools provide the designer with an environment for debugging with at least the functionality that a C programmer would get, for instance. Figure 2 shows a screenshot of the LARD source debugger. This offers standard facilities such as breakpoints and single stepping.

More subtle bugs such as deadlocks are also possible in asynchronous systems. Deadlocks can be easy to diagnose since the simulation will stop with the source viewer indicating exactly the state in which no more progress was possible. In contrast, when debugging a synchronous system the simulation will often run on for a long time after the actual error occurred, making it much harder to track down the error.

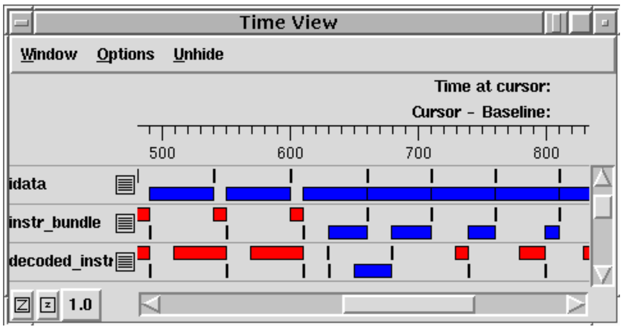


Figure 3: Channel Activity Display

Unfortunately deadlocks may not manifest themselves unless particular timing constraints are met. In general our models are set up with estimated typical delays so that performance analysis predicts typical performance. However we expect our designs to function correctly with any delays as they should be delay insensitive at the block level. Others have adopted a state-space exploration approach for determining the deadlock freedom of a model (Visser et al. 1997), but this approach may not be tractable for complex designs. Using LARD we are able to run modified simulations with randomised delays which we hope will eventually encounter any deadlock or similar problem that is present. Although we have not formally analysed this approach, it does quickly find all deadlocks in our test models.

Performance analysis

Having established the functional correctness of a model, the next question that the designer asks is "how fast does it go?". This is normally answered by running a benchmark simulation and measuring the time taken.

In synchronous systems, benchmark execution times are easy to interpret. The time taken is the number of clock cycles multiplied by the clock period. The clock period is determined by the system's critical path. Each block is either on the critical path or not on the critical path; if it is on the critical path, making it faster will reduce the clock period, and if it is not it won't make any difference.

In contrast the temporal behaviour of asynchronous systems can be much more complex. In some cases, the delay in one block may be masked by a longer delay in another block. In other cases small changes in a delay may propagate throughout the system in a "chain reaction", having an almost chaotic effect on overall performance. Measuring and understanding these effects is important and the LARD toolkit provides various facilities to help with this task.

Figure 3 shows the channel activity display. Each horizontal trace represents the activity on one communication channel over time. The trace has two parts; the upper bars represents the activity of the sender and the lower bars represents the activity of the receiver. Presence of the bar indicates that that end of the channel is ready to communicate; when both are present, the transfer takes place.

By studying this display we can see whether a channel is being starved or blocked. A starved channel has predominantly lower bars, indicating that an upstream block is being a bottleneck. The top trace has this characteristic. A blocked channel has predominantly upper bars, indicating that a downstream block is causing a backlog. The bottom trace behaves in this way for all but one of the communications shown.

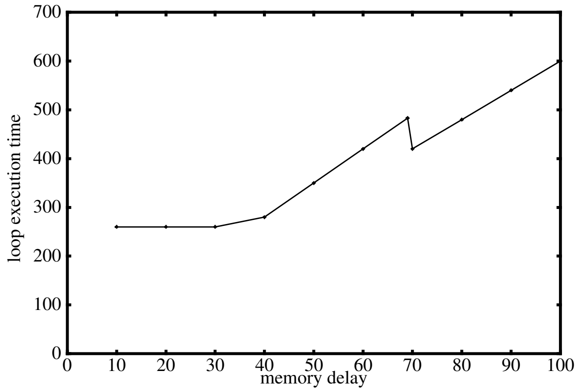


Figure 4: Complex timing relationship measured using a LARD simulation

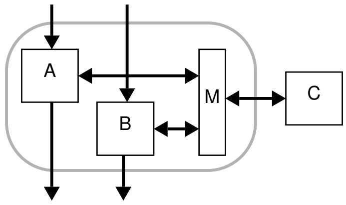


Figure 5: Example Non-Deterministic System

LARD simulations can also quantify the effect on performance of delays. For example, figure 4 shows how the speed of one block influences the overall performance of the system in a complex way. Predicting what influence the speed of this block will have analytically may be very hard; simply measuring its effect using simulation is straightforward.

Interaction with transistor-level simulation

Most of our designs have been taken from behavioural models to gate level schematics manually rather than using automatic synthesis. The reasons for this include the experience of our designers with this approach, the fact that the synchronous ARM is implemented largely by hand and we aim to achieve comparable performance, and the fact that asynchronous synthesis tools are not as well developed as synchronous ones.

Having developed a gate-level schematic by hand we need some way to validate that it implements the same functionality as the equivalent behavioural model. The standard approach to this problem is to simulate the behavioural model in a testbench environment and extract test vectors at the periphery. These vectors are then used to drive the inputs to the gate level simulator and to check its outputs.

Unfortunately this approach is difficult with our LARD models because of the non-deterministic behaviour of some asynchronous systems. For example, consider the system shown in figure 5. Blocks A and B share access to block C via a mutual exclusion circuit M. (In a real system, C could be an expensive but infrequently used resource such as a multiplier which we don't want to duplicate). When A and B want to access C at the same time, M blocks one until the other has completed. Which one M chooses first depends on the exact timing of A and B's requests.

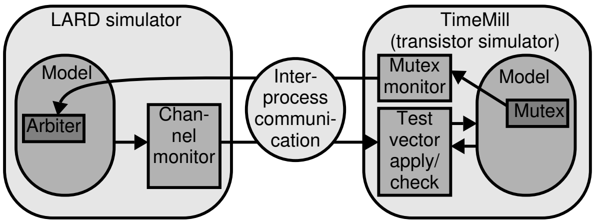


Figure 6: Cosimulation between LARD and TimeMill

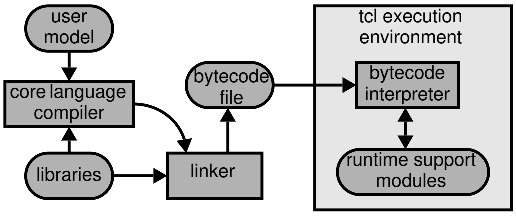


Figure 7: Organisation of the LARD toolkit

Although the order in which A and B are granted access to C is non-deterministic, the eventual outcome as seen from the "outside world" is the same. So generating test vectors for the whole of the system shown in figure 5 has no special problems. On the other hand, the non-determinism is visible if we want to generate test vectors for the subsystem comprising A, B and M (outlined).

In order to generate the correct vectors we need to ensure that the non-deterministic choices made by our behavioural model are the same as the non-deterministic choices made by the transistor-level simulator. This suggests that we need to carry out cosimulation between the LARD simulator and the transistor-level simulator.

Figure 6 shows the approach that we use. The transistor-level simulator has been enhanced with the addition of code to monitor the behaviour of the mutual exclusion circuits. This information is transmitted to the LARD behavioural simulator to control the abstract descriptions of the same choices. In turn, the LARD simulator records activity at the input and output channels to the block being validated and transmits this information to another part of the transistor-level simulator. This translates from the LARD atomic communications to signal transitions and drives the inputs to the transistor-level simulator. Output signals are checked against their expected values and discrepancies are reported.

4. Implementation of the LARD toolkit

One of the key requirements for the LARD implementation was flexibility. Although we started with some idea of what our language and simulation environment needed to do, we wanted to be able to incorporate feedback from the users to add new features as quickly as possible.

As a result the implementation strategy that was chosen is more like a general-purpose programming language than a special-purpose simulation system. The organisation of the tools is shown in figure 7.

The user's source code is compiled into bytecode for a simple virtual machine. The interpreter for this bytecode is embedded in an execution environment that uses the tcl language

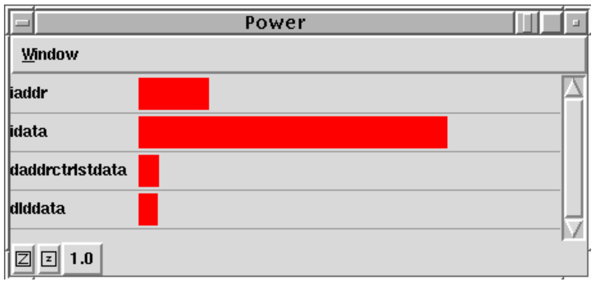


Figure 8: Power display

and tk graphical user interface system (Ousterhout 1994). Recent work has replaced the interpreter with a native code translator for increased performance (Rogers 1998).

Much of the functionality of the language is implemented in libraries which are linked with the user's code. This includes the channel communication primitives which are implemented using shared variables by a channel communication library. In fact even more primitive operations are implemented using libraries to keep the core language as small as possible. In this way the language can evolve based on users' feedback without having to delve inside the core language compiler each time.

Many of these libraries can communicate at runtime with tcl modules in the execution environment. For example the channel communication library sends information about channel activity to the tcl module which provides the trace view display (figure 3). This approach allows us to extend the capabilities of the execution environment without changing the bytecode interpreter.

As an example of this flexibility, consider the power display shown in figure 8 which was not part of the original implementation. Our requirement was to monitor the activity on each channel in terms of the total number of bits that have changed. This measure is a good estimate of the power that will be used by that channel. Implementation of this display required only:

- Small modifications to the channel communication library to record the activity, and store this data in variables accessible to the tcl code: about 10 lines of LARD.
- A tcl module to display the bar graph: about 100 lines of tcl.

To provide a general mechanism for observing and controlling a simulation an animation display has been implemented. This display provides a simple drawing program where each element can be given a tag. Code within the LARD model can then change the attributes of these elements based on their tags. Attributes that can be changed include the colour of box and arrow elements and the string shown in a text element.

Figure 9 shows an animation of this type for the AMULET3 processor. Each instruction is given a colour as it enters the processor's pipeline, and its progress down the pipeline can be observed by following this colour. To make the display more meaningful we can slow down the LARD simulation so that simulated time elapses at a fixed rate relative to real time.

It is also possible to control the simulation by linking sliders and buttons on the display to LARD variables. Figure 10 shows a display where the delays in the model can be controlled by sliders, and the effects on performance monitored.

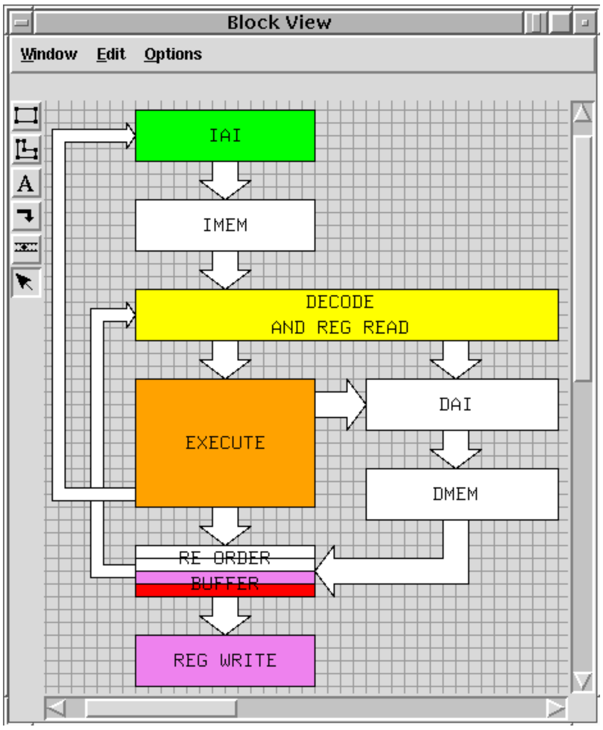


Figure 9: AMULET3 animation

5. Conclusions

LARD is a hardware description language based on CSP-like channel communication which has been developed for behavioural modelling of asynchronous VLSI systems. We have found that the more abstract style of modelling that it provides makes it a much more productive language for this type of modelling than traditional hardware description languages.

The language is now mature and stable enough to take on problems of significant complexity.

From the outset LARD's implementation has been made as flexible as possible. This flexibility has allowed us to add functionality as needed without having to re-write the core system. For example, we have combined the LARD simulator into a co-simulation environment alongside a transistor-level simulator. This allows us to avoid the problem of non-determinism which occurs with asynchronous systems.

The flexibility extends to the execution environment where our use of the tcl/tk language means that the user interface is easily extended.

This flexibility also means that LARD can be adapted to application domains other than its original purpose. LARD could be used to model any system that has fine-grained concurrency, CSP-like communication, timing, and requiring a complex run-time user interface.

LARD is freely available along with comprehensive documentation on the web. The LARD home page is at **`http://www.cs.man.ac.uk/amulet/projects/lard`**

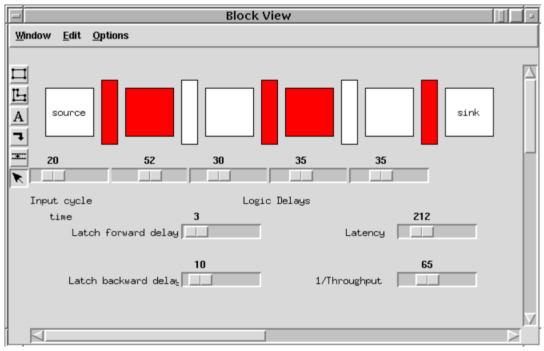


Figure 10: Interactive pipeline simulation

6. References

van Berkel, K.; J. Kessels; M. Ronken; R. Saeijs; and F. Chalij. 1991. "The VLSI programming language Tangram and its translation into handshake circuits". In *Proceedings of the European Conference on Design Automation* (Amsterdam). 384-389.

Furber, S.B.; J.D. Garside; S. Temple; and J. Liu. 1997. "AMULET2e: An Asynchronous Embedded Controller". In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Eindhoven, The Netherlands, April 7-10). IEEE, Los Alamitos, CA. 290-299.

Gilbert, D.A.; and J.D. Garside. 1997. "A Result Forwarding Mechanism for Asynchronous Pipelined Systems". In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Eindhoven, The Netherlands, April 7-10). IEEE, Los Alamitos, CA. 2-11.

Hoare, C.A.R. 1978. "Communicating Sequential Processes". *Communications of the ACM* 21, no.8 (Aug.): 666-677.

Ousterhout, J.K. 1994. *Tcl and the Tk Toolkit*. Addison Wesley. ISBN 0-201-63337-X.

Rogers, I. 1998. "A LARD front-end for Dynamite". Project report, Dept. of Computer Science, University of Manchester, M13 9PL, U.K.

Semiconductor Industry Association. 1997. *The National Technology Roadmap for Semiconductors*. 4300 Stevens Creek Boulevard, Suite 271, San Jose, CA 95129.

Visser, W.; H. Barringer; D. Fellows; G. Gough; and A. Williams. 1997. "Efficient CTL* Model Checking for the Analysis of Rainbow Designs". In *Proceeding of the Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (Montreal, Canada, October). Chapman & Hall.

Woods, J.V.; P. Day; S.B. Furber; J.D. Garside; N.C. Paver; and S. Temple. 1997. "AMULET1: An Asynchronous ARM Microprocessor". *IEEE Transaction on Computers* 46, no.4 (Apr.): 385-398.