From Behavioural Models to Silicon: Tools for the AMULET3 Project

Phil Endecott, Steve Furber

AMULET Group, University of Manchester, M13 9PL U.K.

pbe@cs.man.ac.uk

Abstract

A new high level hardware description language called LARD has been developed for use in the AMULET3 asynchronous microprocessor project. Behavioural models written in this language act as specifications for the manual circuit implementation of the system. In order to validate the circuit implementation we compare behavioural model simulations with transistor level simulations. To perform this comparison we must first overcome a number of problems, including the mapping of atomic channel communication to signal transitions, reconciling different timing characteristics and coping with nondeterminism. This paper describes the solutions adopted to these problems.

	AMULET1	AMULET2	AMULET3[a]
Years	1991-1994	1994-1996	1996-1998
References	4, 10, 12	1, 4, 5	7, 8
Man years	5	12	17
Transistors	58 000	454 000	650 000
Principal tools	ASim Compass	ASim Forcage Compass TimeMill	LARD Petrify Compass TimeMill

Table 1: The AMULET Processors

a. Projected data

1. Introduction

AMULET3 is the third in a series of asynchronous microprocessors developed at the University of Manchester (see table 1). Each of these processors has been a large and complex design requiring substantial support from hardware description language and computer aided design tools.

For the later stages of the design process we have been able to use conventional tools familiar to synchronous designers. This is because the tasks of schematic entry, automatic place and route, layout entry, analogue simulation, switch-level simulation, etc., are independent of the synchronous or asynchronous nature of the circuits.

In the earlier stages of the process novel tools are required. For behavioural modelling a synchronous designer could employ cycle-based simulation. Equivalent abstractions for asynchronous design, such as channel communication, are not provided by conventional tools.

For the AMULET1 and AMULET2 designs we used a proprietary hardware description language called ASim, originating from ARM Ltd. Like other conventional languages ASim lacks any abstractions to simplify the behavioural modelling of asynchronous systems. As a result the ASim models of AMULET1 and AMULET2 were structural, describing the processor in terms of micropipeline macromodules [11] such as call, toggle, select etc. along with datapath components and other logic. The lack of a simpler behavioural model made architectural experimentation difficult as it was necessary to implement a modification in terms of these macromodules before it could be evaluated.

For AMULET3 we have developed a new, more abstract hardware description language called LARD. This language allows us to model at the behavioural level using abstractions such as channel communication. As a result of this change architectural experimentation has become much easier and we have observed a very large increase in designer productivity at the architectural modelling stage.

The downside to the more abstract nature of the model is that producing a corresponding circuit is now more difficult. With AMULET1 and AMULET2 much of the circuit implementation consisted of transcribing the ASim description into schematics. With the new language the task of converting a behavioural description to schematics requires more effort.

Some methodologies automate the transformation of the behavioural specification into a circuit implementation. Possible approaches include syntax-directed translation, as employed by Tangram [2], and speed independent synthesis, as used by Petrify [3]. These approaches have the advantage that the resulting circuits are correct by construction, and if we trust the algorithms used extensive testing of the resulting circuits is not necessary.

We have used Petrify for AMULET3; in particular we have used it for many small control circuits. However we still prefer to take a manual approach to most of our circuit implementations. Reasons for this include:

- the high performance and small size of full custom bounded delay datapath circuits;
- the complexity of our circuits exceeding the capacity of the tools available to us;
- the level of experience and expertise that we have developed in this approach;
- the fact that our benchmark architecture, the synchronous ARM processor, has been implemented in this way.

One major disadvantage to manual implementation is that since human designers introduce random errors it is necessary to check the implementation against the specification. Ideally we would like to prove the equivalence of the LARD model and the circuit, but in the absence of tools to facilitate this we compare LARD simulations and transistor-level simulations until we gain sufficient confidence that the two models behave in the same way.

For AMULET1 and AMULET2 the similarity of the ASim and circuit descriptions made this comparison of simulation results relatively straightforward. On the other hand LARD is sufficiently different from the circuit implementation to make comparison more complex. Particular problems that we have encountered include:

- How do we know that our simulations have sufficiently exercised the LARD model and circuit to reveal any problems?
- In comparing the two simulations, how do we map from the abstract idea of channel communication to our chosen implementation technology?
- How do we reconcile the different timing characteristics of the LARD model and circuit simulation?
- What happens when the LARD model describes a non-deterministic circuit?

This paper describes the approaches to these four problems that have been developed for the AMULET3 project. The next section provides an introduction to the LARD language and toolset. Section 3 describes how we generate test sequences from the LARD simulation and check for test coverage. Section 4 explains the problem of mapping from LARD channel communication to four-phase logic and describes the solution that we have adopted. Section 5 considers what happens when the LARD model and the circuit simulation have different timing behaviours. Section 6 describes a possible cosimulation approach to validation of non-deterministic systems.

2. The LARD Hardware Description Language[1]

LARD was initially designed with these objectives:

- To provide atomic channel communication, as in Tangram and occam. Removing the need to explicitly model request and acknowledge signals and signalling protocols greatly simplifies the modelling.
- To provide high level language programming features.
- To include a front end with productivity features such as a trace display, source level debugger, visualisation etc.
- To be robust enough to cope with the complexity of the AMULET3 processor.
- To be flexible enough to allow us to add substantial extra functionality at a later stage when we learn from experience what is needed.
- To be simple enough to allow us to get it up and running quickly.

The language that evolved out of these requirements includes structured programming features at least as useful as those provided by other modern languages. LARD provides:

- A sophisticated type system with arrays, records, enumerated types, and polymorphic and overloaded functions.
- Facilities for libraries and separate compilation.

1. The LARD toolkit is freely available and can be found along with comprehensive documentation at:

`http://www.cs.man.ac.uk/amulet/projects/lard`

```
pipeline_latch(in:var(chan(T:type)),
               out:var(chan(T))
              ):expr(void)=(
  v:var(T).
  forever(
    in?(v:=?in);
    out!v
  )
).

increment_pipe_stage(in:var(chan(int)),
                     out:var(chan(int))
                    ):expr(void)=(
  C:var(chan(int)).
  pipeline_latch(in,C) |
  forever(
    C?(
      wait_for(10);
      out!(?C+1)
    )
  )
).

inc_by_3(in:var(chan(int)),
         out:var(chan(int))
        ):expr(void)=(
  C1:var(chan(int)).
  C2:var(chan(int)).
  "stage1" is increment_pipe_stage(in,C1) |
  "stage2" is increment_pipe_stage(C1,C2) |
  "stage3" is increment_pipe_stage(C2,out)
).
```

Figure 1: Example LARD program

- Hierarchical scope.
- A concise and extensible syntax.
- A standard library for arithmetic and logical operations, input/output, string handling etc.

To support the behavioural modelling of asynchronous systems LARD also provides:

- Fine grain concurrency.
- Channel communication.
- A scheduler with explicit modelling of time.

The example program in figure 1 illustrates some of these areas. The program constructs a three stage pipeline where each stage increments the data value. Each stage consists of a pipeline latch and code implementing an incrementer.

Concurrent composition is provided by the vertical bar operator (**|**). Sequential composition is provided by semicolon (`;`).

Channel communication is implemented using `!` and `?`. The syntax is

chan `!` *value*	Send *value* to channel *chan*.
chan `?` *expr*	Wait for a value to arrive on *chan*, then evaluate *expr*. The sender does not continue until *expr* has completed.
`?`*chan*	Within the *chan* `?` *expr* construct, `?`*chan* returns the value that has been received.

Note that in Tangram we would write `c?v` to store a value from channel *c* into variable *v*. In LARD receiving a value from a channel is split into two operations, and we must write the more verbose `c?(v:=?c)`. The advantage of this different notation is that it allows us to describe blocks that perform processing and possibly output communications between an input arriving and it being acknowledged. In circuit terms this corresponds to a block that does not have latches on its inputs. Such constructs are common in the AMULET3 model.

In our design approach having some timing information in the model at an early stage is essential as many of our early design decisions are concerned with performance issues. We need to be able to annotate our code with estimates of delays in critical areas and then measure the resulting performance. The incrementer in figure 1 illustrates both delay modelling and the LARD channel receive syntax:

```
C?(
  wait_for(10);
  out!(?C+1)
)
```

The LARD toolset consists of a compiler, a linker and a standard library which are used in a similar way to conventional programming language tools. The result of the compilation is a program in a bytecode language for a simple virtual machine.

The interpreter for this virtual machine is embedded within the tcl (tool control language) [9] environment. By using tcl it becomes possible to extend the execution environment to monitor and interact with the simulation. Simulations can run offline in batch mode, in interactive command line mode, or with a graphical user interface provided by the tk toolkit. Facilities provided by the execution environment include:

Source code debugging with breakpoints and single stepping. The complexity of the AMULET3 design makes software style debugging essential.

Channel activity traces (see figure 2). These diagrams show timing information and help to identify any bottlenecks in the system.

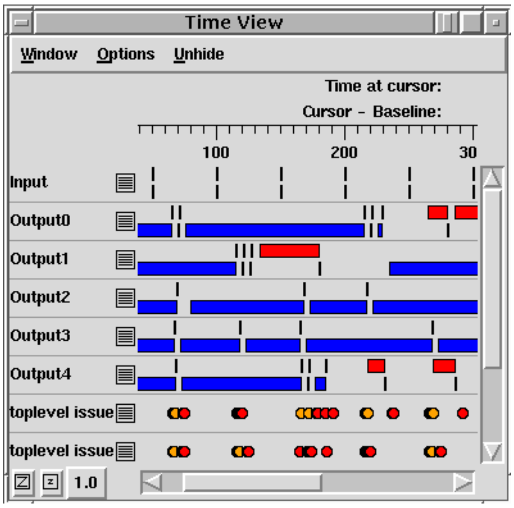


Figure 2: Channel Activity Trace

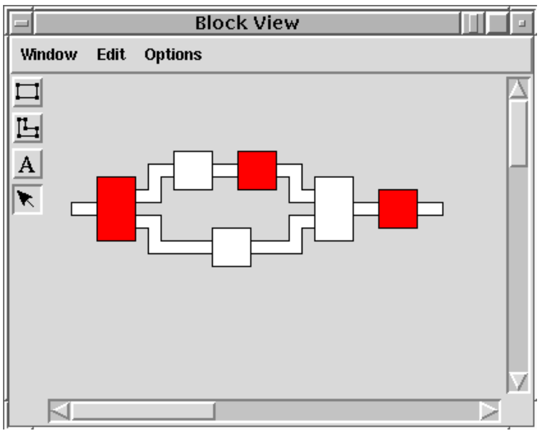


Figure 3: Animated Block Diagram

Animated block diagrams (see figure 3, which shows a snapshot of which blocks are active in a complex fork-join pipeline). Controls, for example sliders to adjust delays or buttons to activate interrupts, can be embedded in the diagrams.

Power estimation based on channel activity (see figure 4). This display is based on the number of bits that have changed in the data communicated by the channel.

Together these facilities have made the LARD language and toolkit a highly productive environment for architectural modelling at the behavioural level. The AMULET3 model comprises approximately 6000 lines of LARD source code, and executes ARM code at around 10 instructions per second in batch mode.

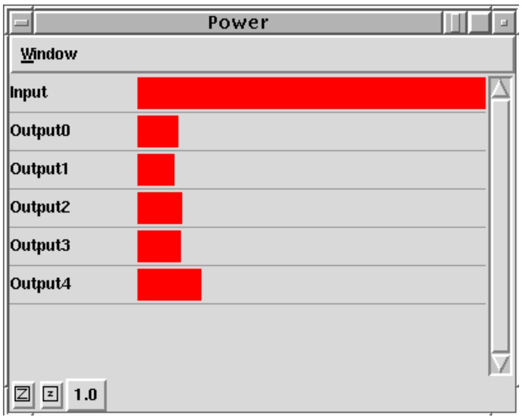


Figure 4: Power Viewer

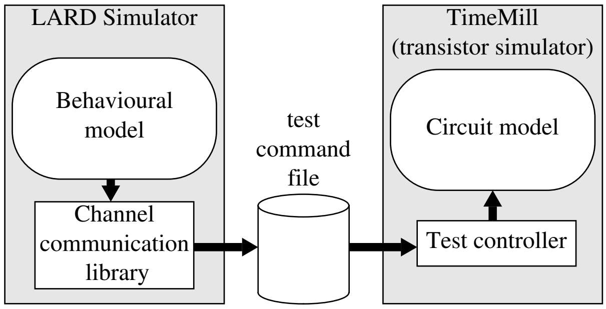


Figure 5: Test Generation and Application Scheme

Once the behavioural model is complete it acts as a specification for the design of the circuit implementation.

3. Test Generation and Coverage Analysis

Having designed a circuit we wish to show that it behaves in the same way as the LARD block to which it corresponds. We do this by running a LARD simulation and extracting data about the input and output channel behaviour of the block of interest. This data is then used to drive a transistor level simulation of the circuit and the behaviour is compared. The general scheme is show in figure 5.

The top one or two levels of hierarchy of the LARD model correspond closely with the circuit hierarchy. Up to this point both the LARD model and circuit consist primarily of sub-blocks interconnected by bundled data channels with well behaved protocols. Beyond this point the LARD model becomes more behavioural and the circuit decomposes the channels into distinct event and data signals. When testing circuits against the corresponding LARD blocks we normally work with the smallest blocks that maintain well defined bundled data channel interfaces. In the case of AMULET3 the processor core is divided into about twenty such blocks.

Having isolated a suitable block in the LARD model it is exercised in one of two ways:

- The block is embedded in a specially constructed testbench. This testbench applies a pattern of possibly pseudo-random inputs to the block and consumes and discards its outputs.
- The block is exercised in situ in the processor model. The memory model is loaded with an ARM machine code test program and the whole model is allowed to free run.

The former approach has the advantage that the inputs can be more readily controlled, and simulation times may be shorter. The latter approach has the advantage that no extra code need be written.

Whatever the source of test stimulus our concern at this point is to establish whether the block has been sufficiently exercised to give us confidence that the corresponding circuit matches its specification if it behaves in the same way. With this aim a LARD test coverage analysis tool has been added to the LARD toolkit.

The execution environment has been extended to write a log file identifying unreached portions of the code. A postprocessing tool converts this log file into a list of function names and line numbers. With this feedback the designer can modify the test program to give greater coverage.

Similarly our transistor level simulator is able to report circuit nodes that have not been toggled during the simulation.

Note that there is no guarantee that because a line of code has been executed or a node has gone low and high that its effects can be observed from the outputs of the block; significant responsibility still rests with the designer in this area.

4. Test Generation for 4-Phase Channel Communication

This section describes in more detail how test stimulus for the transistor model is generated from observing the activity of the LARD model.

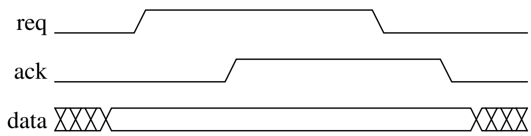


Figure 6: The four phase broad protocol

The LARD execution environment has been extended to allow activity on the input and output channels of the block under test to be captured and written to a file. The activity on the input channels will form the basis of the input stimulus for the circuit simulation. The activity on the output channels will be used to validate the output of the circuit simulation.

Mapping from the abstract channels of the LARD model to their circuit implementations is not necessarily straightforward. Firstly a number of different protocols may be used by the circuit: either two phase or four phase, and in the latter case with either the early or broad data validity scheme [6]. Furthermore the request and acknowledge signals may be active high or active low. The protocol used by each channel is specified by the designer when the test generation system is configured.

For simplicity only the four phase broad protocol with active high request and acknowledge is considered in the rest of this paper. This protocol is shown in figure 6.

Because each atomic communication in the LARD model corresponds to a sequence of signal transitions in the circuit, each channel transaction must be mapped to a sequence of commands for the circuit simulator[1]. An input transaction is translated to the following commands:

```
set data ....
wait_for setup_time
set req 1
check ack 1
set req 0
check ack 0
```

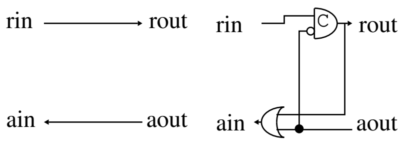
An output transaction is translated to:

```
check req 1
check data ....
set ack 1
check req 0
set ack 0
```

This mapping is straightforward when one channel transaction is seen in isolation. Unfortunately a problem

1. The notation that is used for the simulator commands is a hypothetical simplified language.

```
forever(
  in?(
    out!(?in)
  )
)
```



(a) **(b)** **(c)**

Figure 7: Example of Alternative Inactive Edge Behaviours

arises when we consider how these commands should be interleaved when two or more transactions overlap. This problem results from the inherently two phase nature of the LARD channel communication. Consider the trivial lard code and two circuits shown in figure 7. The LARD code shown in (a) simply waits for a value to arrive on the input and sends that value to the output. Note that there is no latch involved.

The simplest implementation of this function uses two wires to join input request and acknowledge to output request and acknowledge as shown in (b). In this case the simulator commands required (excluding data for simplicity) would be as follows:

```
set rin 1
check rout 1
set aout 1
check ain 1
set rin 0
check rout 0
set aout 0
check ain 0
```

On the other hand it would be equally acceptable for the designer to choose the circuit shown in (c). This circuit exhibits the same behaviour as the two wire circuit for the active edges of the protocol, but optimises the behaviour on the inactive edges: it does not wait for aout+ before sending ain+. Note that as the LARD model is protocol independent it says nothing about the expected behaviour of the inactive edges. A suitable sequence of simulator commands for this circuit would be:

```
set rin 1
check rout 1
check ain 1
set rin 0
set aout 1
check rout 0
set aout 0
check ain 0
```

The problem is that for any LARD program there may be several equally valid circuit implementations, each requiring a unique and incompatible test sequence. Furthermore the number of possible interleavings grows substantially as the number of inputs and outputs increases.

The solution adopted is to record only the active edges of the request and acknowledge signals in the test file, and to add an extra mechanism to the circuit simulator to process the inactive edges autonomously. For the four phase broad protocol the required autonomous behaviour for an input channel is:

- req+ is sent in response to a file command.
- req- is sent autonomously in response to ack+.
- ack- is checked against a file command.

For an output channel:

- req+ is checked against a file command, and ack+ is sent in response.
- req- is ignored.
- ack- is sent in response to a file command.

Similar mappings exist for the other protocols.

The test commands for the example in figure 7 are then as follows[1]:

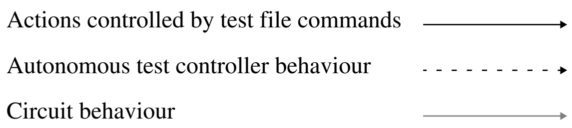
```
send_event rin
check_event rout
send_event aout
check_event ain
```

Figures 8 and 9 show how this single test sequence correctly validates both the circuit variants of figure 7.

The extra mechanism that implements the autonomous behaviour is the test controller shown in figure 5. As well as autonomous control of the inactive edges of the request and acknowledge signals, and control of the active edges as required by a control file, the test controller is also responsible for setting and checking bundled data signals, and checking that they meet their bundling constraints. The test controller is implemented in C using the foreign language interface of our transistor level simulator, TimeMill.

This behaviour is, in some sense, equivalent to connecting four phase to two phase and two phase to four phase converters to the input and output channels, as shown in figure 10. Here a subcircuit with a four phase broad protocol interface is interfaced to a two phase environment by means of toggle elements and exclusive OR gates. This environment behaves similarly to the LARD model.

1. A 2-phase notation is used here, since only active edges are recorded.



Key to figures 8 and 9

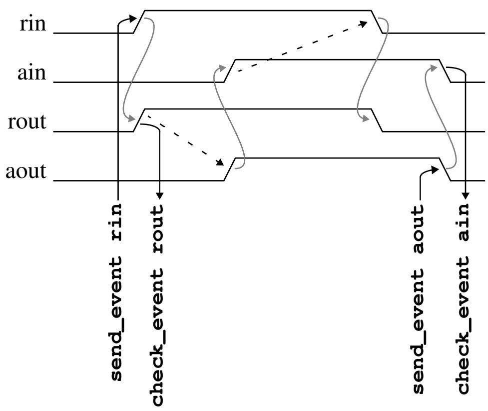


Figure 8: Testing the circuit of figure 7 (b)

5. Coping with Interleaved Communications

A further problem of interleaving occurs when concurrent events have different timing properties in the LARD model and the circuit simulation. Consider the LARD program in figure 11 (a).

In this code, after a value arrives on input *in* functions *f* and *g* are computed concurrently and their results are sent to outputs *out1* and *out2* respectively. In the LARD model the definitions of *f* and *g* will contain **`wait_for`** statements giving estimates of their propagation delays. These delays are useful for estimating the performance of the system from the LARD simulation, but they do not form part of the specification for the circuit.

Figure 11 (b) shows a possible circuit for the control part of the implementation of this block. It contains two matched delays. These delays may be determined from analogue simulation results, or be implemented as matched paths. Although these delays are likely to be similar to the delays specified in the LARD model, they need not be exactly the same.

The implication is that in the LARD model the events on out1 and out2 may be interleaved in one way, but in the circuit they may be ordered in another equally valid way. The test system must accept all valid circuit behaviours.

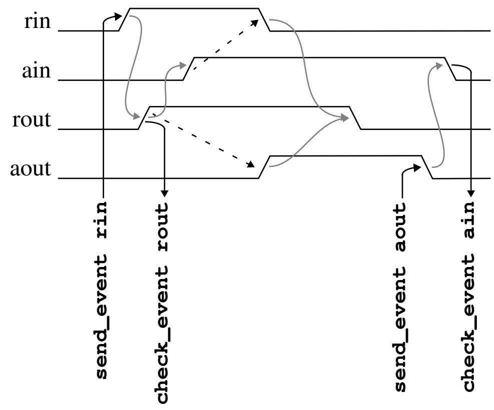


Figure 9: Testing the circuit of figure 7 (c)

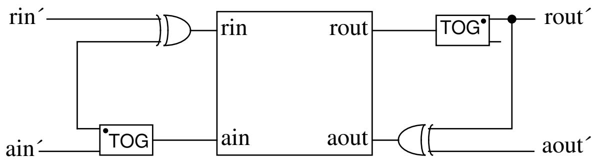
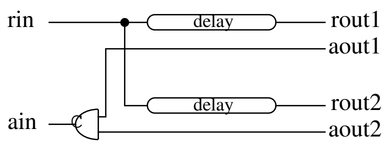


Figure 10: Two Phase to Four Phase Conversion

```
forever(
  in?(
    out1!f(?in) |
    out2!g(?in)
  )
)
```



(a) **(b)**

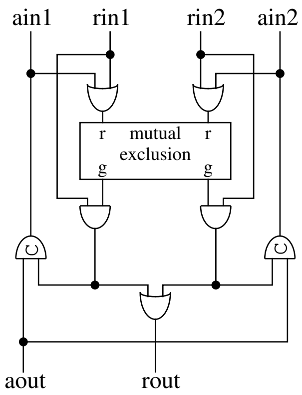
Figure 11: Illustration of Concurrent Output Interleaving

The solution to this problem is to delay communications in and out of the block under test in the LARD model so that all enabled output events have occurred before any responses from the environment are returned. With this modification a single test sequence that will accept all valid circuits can be produced. In the case of the code in figure 11 the sequence would be:

```
send_event rin
wait_for long_time
check_event rout1
check_event rout2
send_event aout1
send_event aout2
wait_for long_time
check_event ain
```

```
forever(
  case (select(in1,in2))
  when 1 => in1?(
                  out!(?in1)
               )
  when 2 => in2?(
                  out!(?in2)
               )
)
```

(a)



(b)

Figure 12: LARD model with Arbitration

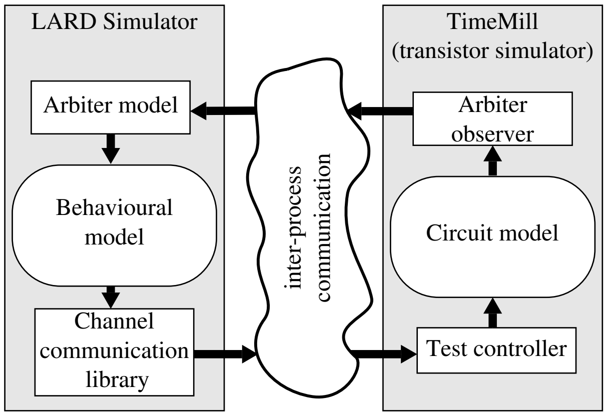


Figure 13: Cosimulation

```
forever(
  case (arbitrate(in1,in2,"arbiter_name"))
  when 1 => in1?(
                  out!(?in1)
               )
  when 2 => in2?(
                  out!(?in2)
               )
)
```

Figure 14: LARD Model Updated for Cosimulation

It is now possible to define the behaviour of the **`check_event`** command. When an active edge occurs on a signal the test controller sets a flag. **`check_event`** clears the flag. If an active edge occurs while the flag is still, or if a **`check_event`** command is encountered and the flag is clear, an error is indicated.

6. Arbitration and Cosimulation

The approach of delaying responses from the environment until all enabled output events have occurred solves the problem of interleaving for all deterministic circuits. However in some circuits the interleaving of events is more significant and a different approach is required. Consider the LARD program shown in figure 12 (a).

The code describes a block with two inputs, *in1* and *in2*, and a single output *out*. The select statement waits for data to arrive on either of the two inputs, and in combination with the case statement it causes the value from that input to be copied to the output. This is an arbitrating call block. A possible circuit implementation (excluding data) is shown in figure 12 (b).

When data arrives on both inputs simultaneously the select statement makes an arbitrary choice. In the corresponding circuit implementation this corresponds to arbitration between the two input request signals.

Clearly it is possible for the LARD model to make one choice and for the transistor level simulator to make the other choice. With the approach described so far this would cause the validation to fail even though the circuit behaviour is acceptable.

The solution to this problem is to adopt cosimulation. The schemes described so far run the LARD simulation, extracted a test command file, and then run the transistor simulation. Now the LARD simulation and the transistor simulation are run concurrently with bidirectional communication. The LARD simulation sends test commands to the transistor simulation as before. At the same time the transistor simulation sends data back to the LARD model to describe the behaviour of its arbiters. This data is used to steer the LARD arbiters to the same outcomes. The organisation is shown in figure 13.

Like the test controller the arbiter observer is a C program written using the TimeMill foreign language

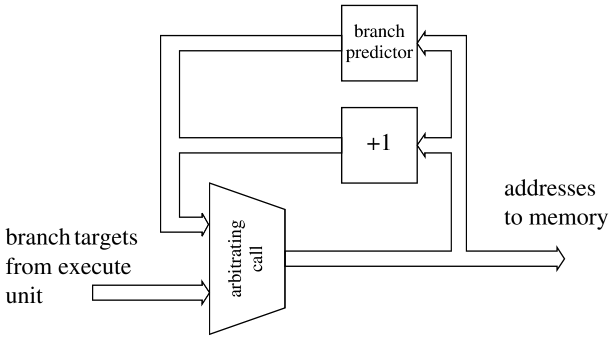


Figure 15: The AMULET3 Instruction Address Interface

interface. It observes the grant wires of each mutual exclusion element.

This change requires that the LARD program is modified as shown in figure 14. The select call has been replaced by a new function, arbitrate. When cosimulation is not active arbitrate behaves as select. When cosimulation is active the string argument to arbitrate is used to identify an arbiter in the transistor simulation that this LARD construct must follow.

An example of a block where cosimulation is necessary for validation is AMULET3's instruction address interface, shown in figure 15. The role of this block is to generate a stream of instruction fetch addresses to be sent to the memory. These addresses are normally computed by the PC incrementer and the branch predictor. When a branch is mispredicted the execute unit sends the correct branch target to this block. The arrival of this branch target is asynchronous to the internal generation of addresses, so an arbiter selects between the branch target and an address from the incrementer / branch predictor. Instructions fetched between the misprediction and the correct target are eventually discarded by the execute unit, and the number of such instructions is variable. It is likely that the LARD model and the transistor model will disagree over the number of such instructions.

With the cosimulation approach the choice made by the transistor model of the arbiter in the instruction address interface can be used to control the corresponding choice in the LARD model. In this way the address streams are kept in step and validation should function correctly.

7. Conclusions

At the time of writing the tools described here have been implemented and evaluated on benchmark designs, with the exception of the cosimulation system. Other members of the group are working on circuit implementation of various blocks of the AMULET3 processor, and will shortly be making significant use of these tools.

Our previous processor implementations have shown that conventional tool flows can be adapted for use with asynchronous designs, but with some drawbacks. Behavioural modelling has made the AMULET3 design a great deal simpler, but the tools required include more than just behavioural simulation. Synchronous designers have a large array of tools available to them and if asynchronous design is to compete we must provide equivalent functionality. The LARD toolkit provides much of what is needed.

References

[1] AMULET2e Datasheet. http://www.cs.man.ac.uk/amulet/AMULET2e_uP.html

[2] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, F. Chalij, "The VLSI programming language Tangram and its translation into handshake circuits", Proc. European Conference on Design Automation, Amsterdam, 1991, pages 384-389.

[3] J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakovlev, "Synthesizing Petri nets from state-based models", Proc. ICCAD'95, November 1995, pages 164-171.

[4] S. B. Furber, "ARM System Architecture", Addison Wesley Longman, 1996. ISBN 0-201-40352-8.

[5] S. B. Furber, J. D. Garside, S. Temple, J. Liu, "AMULET2e: An Asynchronous Embedded Controller", Proc. Async'97, Eindhoven, pages 290-299. http://www.cs.man.ac.uk/amulet/publications/papers/async97_A2e.html

[6] S.B. Furber, J. Liu, "Dynamic Logic in Four-Phase Micropipelines", Proc. Async'96, Aizu-Wakamatsu, Japan, March 18-21 1996.

[7] D. A. Gilbert, "Dependency and exception handling in an asynchronous microprocessor", Ph.D. Thesis, Department of Computer Science, University of Manchester, U.K., 1997, submitted for examination.

[8] D. A. Gilbert, J. D. Garside, "A Result Forwarding Mechanism for Asynchronous Pipelined Systems", Proc Async'97, Eindhoven, pages 2-11. http://www.cs.man.ac.uk/amulet/publications/papers/async97_queue.html

[9] J. K. Ousterhout, "Tcl and the Tk Toolkit", Addison Wesley, 1994, ISBN 0-201-63337-X.

[10] N. C. Paver, "The Design and Implementation of an Asynchronous Microprocessor", Ph.D. Thesis, Department of Computer Science, University of Manchester, U.K., 1994. http://www.cs.man.ac.uk/amulet/publications/thesis/paver94_phd.html

[11] I. E. Sutherland, "Micropipelines", Communications of the ACM, Volume 32, No. 6, June 1989, pages 720-738.

[12] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, S. Temple, "AMULET1: An Asynchronous ARM Microprocessor", IEEE Transactions on Computers, April 1997, Vol. 46, No. 4, pages 385-398.