# DESIGN AND SIMULATION OF ASYNCHRONOUS MICROPIPELINE CIRCUITS IN RAINBOW

Howard Barringer, Donal Fellows, Graham Gough,
Pete Jinks, Brian Marsden, Alan Williams*

Department of Computer Science, University of Manchester, Manchester, M13 9PL, UK
email: howard@cs.man.ac.uk

## KEYWORDS

VLSI & simulation, CAD, simulation interfaces, animation software, hierarchical models.

## ABSTRACT

We describe the development of the Rainbow hardware design environment for supporting the design and simulation of asynchronous systems. Rainbow contains a suite of asynchronous hardware description languages for modelling systems based on Sutherland's micropipeline asynchronous design philosophy. Full interworking is supported between Rainbow sub-languages so that the various components of a design can be described in the most suitable way. The framework offers rapid design description and simulation at a high level, so that the functionality and performance of the design can be assessed and experimented with at an early stage in the design cycle. We illustrate one of the Rainbow sub-languages (*Green*) through a simple micropipeline register design and examine its operation with the simulator.

## INTRODUCTION

Digital hardware systems have, for the past 30 years, been designed principally using a synchronous approach, and involving global clocking strategies. Recently, there has been a resurgence of interest in asynchronous hardware design techniques (Birtwistle and Davis, 1995), with the aim of achieving low power combined with high speed operation, which is becoming increasingly difficult with synchronous systems.

Various asynchronous design methodologies have emerged, in particular Sutherland's micropipeline philosophy (Sutherland, 1989). Extensive designs have been built using this; for example at the University of Manchester, the AMULET Group have been investigating the design of, and implementing, asynchronous micropipeline versions of the ARM microprocessor (Furber, 1995). Currently, confidence in design correctness is increased by running validation programs through a (commercially available) hierarchical simulation/modelling tool. It has been observed that although the use of micropipelining techniques leads to systems which may be easily composed, it is remarkably easy to construct complicated feedback loops of asynchronous pipeline stages that may deadlock (Paver, 1994). So far, simulation has revealed several instances of this problem (but it is of course not known how many others remain undetected). In general, engineers developing asynchronous digital systems have been hampered by the lack of suitable design analysis and verification support tools, to check for liveness, activity, as well as 'equivalence', that are capable of supporting the currently achievable size of design.

*The authors acknowledge support from the Engineering and Physical Sciences Research Council via research grant GR/K42073.

### The Rainbow Philosophy

In this work, we are building a formally-based design description/representation and analysis environment to support the design of asynchronous systems throughout the design life-cycle. The aim is to embed formal analysis techniques into a framework for asynchronous hardware development that can be demonstrated to bring greater benefit and confidence to digital systems engineers and architects in their design work over the more usual traditional routes of mental exercise and simulation.

We are developing a hardware description framework called *Rainbow*, in close collaboration with the AMULET hardware engineers. Rainbow offers a range of description styles that engineers may use during design development, for example for future asynchronous versions of the ARM microprocessor. In particular, Rainbow can be viewed as being composed of a collection of sub-languages, where full interworking between these is supported; design components from one language can be instantiated within a design described using a different sub-language.

At present, Rainbow includes a dataflow-style language, called *Green*, based on micropipeline communication primitives and a control-flow sequential language, called *Yellow*, that is similar to OCCAM (Burns, 1988), but uses an Ada-like rendezvous communication (Li, 1982; Sommerville and Morrison, 1987). A formal semantics is defined for each sub-language, enabling Rainbow to form the basis of specialised formal verification and analysis tools for supporting a design. The sub-languages are interfaced at the semantics level.

This paper concentrates on the Green language, specifically on its application to the design, specification and simulation of micropipelines. Further details of the Rainbow language suite may be obtained in (Rainbow, 1996).

## DATAFLOW DESCRIPTION IN GREEN

The sub-language Green describes a micropipeline circuit structurally as a dataflow network whose nodes are primitive pipeline elements such as buffers, functions and multiplexers. The nodes are connected together by point-to-point stateless channels (i.e. wires). Data flows along channels according to a rendezvous style of synchronised communication. Being a static dataflow language, whose processing elements are data-driven and need all of the required input values to be present before they can fire, Green has similarities to other dataflow languages (Herath et al., 1992). However, there is no implicit buffering between elements, and so each output of a (stateless) element must be ready to accept a value before that element can fire. The (state-full) buffer element is the exception, since an empty buffer can only accept a value on its input, and a full buffer can only produce a value on its output. Clearly, elements with no outputs (sinks) or no inputs (sources) may always accept or produce values. These characteristics reflect the intended use of Green as a language for describing asynchronous micropipeline hardware, which of course has a static structure, and is based on the micropipeline handshake communication primitive.

### Simple Micropipeline Operation

Figure 1 shows a micropipeline fragment, consisting of three buffers A, B, C, and a duplicate element `Dup` that copies the value in buffer A to both buffers B and C. Buffers A and C initially contain some data values $x$ and $y$, respectively; buffer B is empty. Each bundled-data channel is expanded to reveal the request (`req`), data (`data`) and acknowledge (`ack`) wires. The network operates in the following way: since buffer A is full, the request wire from A to the duplicate becomes high, indicating that data is available to be forwarded. In turn, the duplicate sets the request wires on its outputs to buffers B and C. As buffer B is empty, the data can be accepted, and the `ack` wire back to the duplicate becomes high. As buffer C is full, the data cannot be accepted, so the request is blocked. Working backwards, the duplicate cannot acknowledge its input (from buffer A) until both of its outputs have been acknowledged, so the acknowledgement is blocked. Only when buffer C can accept the data and returns an acknowledgement can the duplicate element return an acknowledgement to buffer A. Only when this acknowledgement is received can the contents of buffer A be discarded, and A is then free to accept a further data item.

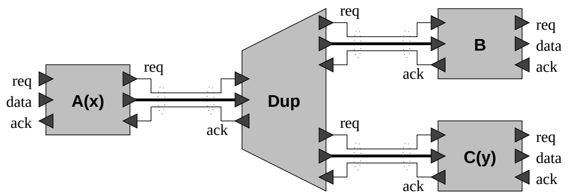


Fig. 1. Simple Micropipeline

In Rainbow we abstract away from the explicit micropipeline handshake, and consider the higher level rendezvous communication — each request/data/acknowledge wire bundle is now represented by a single channel carrying only data, as illustrated in the examples below.

### Micropipeline Elements

The principal micropipeline elements are described below. Other more complex elements can be derived from these basic elements; see (Rainbow, 1996) for more details.

**Buffer:** An empty buffer reads a data item from its single input channel, and stores it. Once it has acknowledged receipt of this item (but not before), it then presents the item on its single output. Once a forwarded item has been acknowledged, but not until then, the buffer is ready to read a new input value. Clearly, a full buffer cannot receive a new value, and an empty buffer cannot output a value. Buffers may be initialised or left empty at the start of a simulation.

**Function:** This element accepts data on various input channels. When data is available on all inputs, it is transformed into data on its output channels. No input channel can be released until all outputs have been released (acknowledged). Functions are, by definition, stateless.

**Duplicate:** This specialised function copies values (unchanged) on its input channel to all of its outputs. Only when the data on all outputs has been consumed is the input channel released.

**Sources (file, random, constant):** These can be regarded as specialised functions which generate a stream of data items on their (single) output channel. There are no input channels associated with these devices. A file is used to supply a finite user-defined stream of values. A random source supplies an unbounded stream of random numbers and a constant produces a stream of the same value. Therefore a source is potentially able to present a data element at any time after the previous one has been acknowledged.

**Sink (including files):** This will absorb an unbounded number of data items; it is always ready to accept another data item. It has a single input channel, but no output.

**Ports:** These are input or output points in a network, connecting channels to the external context.

**Flow Control:** This class contains elements which manipulate input/output data streams, possibly according to the data that is supplied on a control input. For example, the `merge` element is a stream-based multiplexer. It reads a control data item (a boolean for example), and transfers a data item from the selected input port to the output port. No data item is read from the unselected input ports. Similarly, the `split` element is a stream-based demultiplexer. It takes an input data stream and a control data stream, and writes the data item on the chosen output port. No data item is produced on the output ports not chosen.

An `arbitrate` element merges its two input streams (non-deterministically) into a single stream (its output). If items are available from both inputs, then an *internal* choice will be made between the two.

A more generalised element type is the `table`, which combines the functionality of functions and flow control elements. In a sense, `split`, `merge` and functions are all special cases of tables.

## EDITING AND ANIMATION

A schematic editing tool has been implemented that can be used for creating and editing Green networks in a hierarchic fashion. It has been embedded in an extensible design environment, which currently also contains a simulator so that designs can be animated — the environment invokes the simulator and receives back information for each simulation step showing the changes to pipeline elements. It displays the changes as colour coded annotations to each wire, showing both the values which are actually transferred between elements, together with the values which will *potentially* be output by an element, but cannot be because the next element is not ready to accept.

### Operation of the Simulator

The simulator operates in dataflow fashion, using a succession of forward and reverse *sweeps* through the network. In the forward sweep, all possible routes by which data could be forwarded through the network are followed, in an attempt to advance data (or data available flags) as far as possible. In the reverse sweep, a similar procedure is followed for all possible routes back through the network along which acknowledgements could be returned. Each step (pass) of the simulator invokes one forward sweep followed by one reverse sweep. A *step* button in the editor allows for the execution of a single pass; the new locations of the data in the network are then displayed. The simulator monitors the number of micropipeline elements whose status has changed, and the number of request lines which are high, and reports either deadlock or completion, if there has been no activity during a pass, as appropriate.

In its simplest form, the simulator algorithm works as follows. The forward sweep repeatedly and recursively calls a procedure *progress* for each output wire from each element, starting at each source or constant, until a sink is reached. Data (more strictly, its availability) is propagated forwards until an element is reached which cannot yet receive the data. Similarly, a function *regress* implements the reverse sweep by recursively scanning backwards along each input wire, starting at each sink, until a source is reached. In effect, acknowledgements are propagated backwards. This algorithm guarantees that all possible routes through the network are checked in each direction.

It should be noted that although the user is primarily interested in, and is provided with, details of changes in status of the wires connecting the network elements, the simulator operates in quite a different way, being element-centred. Each element is considered to *own* its input and output wires. The connectivity analysis ensures that a given input wire for one element matches a corresponding output wire for another one.

### Animation of a simple register

Figure 2 shows a Green model of a simple register. This has two inputs (`control` and `datain`) and one output `dataout`. The control input determines whether the data is being read (1) or written (0). Initially a data value 4 is present on `datain`, and a write (0) instruction is present on `control`. The latter selects the upper input to the merge and the upper output from the split. `Buffer(5)` initially contains the value 5 (hence its name); such initialisation is necessary to avoid deadlock in the event that the first control request is a read. The other buffer is uninitialised.

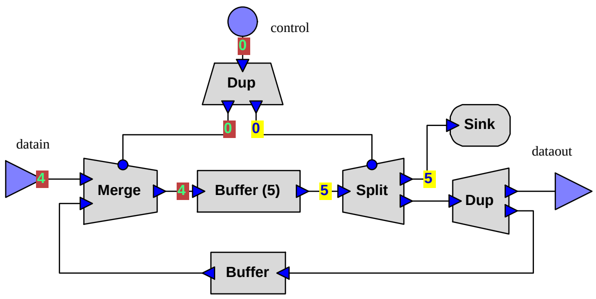


Fig. 2. State of register after one simulation step.

During the progress phase of the first simulation step, the write instruction makes the data value on `datain` (4) available to `Buffer(5)` via the merge, but this is blocked as this buffer is full. Wires waiting for acknowledgement are shown with data values on a dark background. It also causes the contents of `Buffer(5)` to be piped through the split to the sink, where it is discarded. In the regress phase, an acknowledgement is returned from the sink to the split. This releases its data input, `Buffer(5)`, which is now empty and will be able to accept another input value during the next simulation step. Acknowledged wires are shown with data values on a light background. The split control input is also released, but as the left hand output of the duplicate has not yet been acknowledged, `control` is not acknowledged. As the value on `datain` could not be copied to `Buffer(5)`, `datain` remains unacknowledged as well.

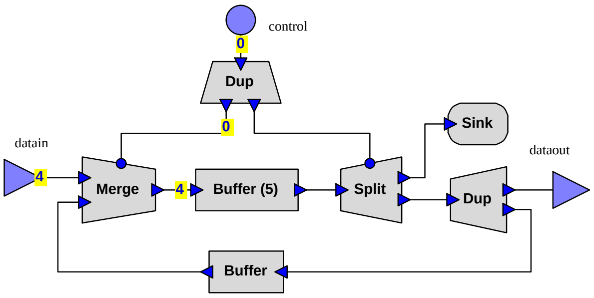


Fig. 3. State of register after two simulation steps.

Referring to Figure 3, `Buffer(5)` now accepts the new data value, and an acknowledgement is returned via the merge to the left hand output of the duplicate, and to `datain`, which is released. As both outputs of the duplicate have now been acknowledged, `control` is also released, and both `control` and `datain` will be able to present new values during the third simulation step.

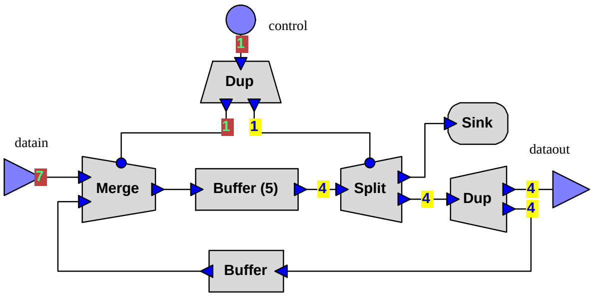


Fig. 4. State of register after three simulation steps.

In step three of the simulation, a read instruction is issued, by placing the value 1 on `control`, as shown in Figure 4. The contents of `Buffer(5)` have been forwarded via the split to `dataout` and `Buffer`. Note that the purpose of `Buffer` is to prevent the deadlock which could occur in a single buffer implementation. Acknowledgements have been returned via the split to `Buffer(5)` and the top duplicate node. As earlier, since `Buffer` does not acknowledge until the regress phase of the simulation step, its new contents cannot be forwarded to the merge yet. Hence the chosen merge input remains unavailable, and its control input remains blocked. As in the write cycle, `control` cannot be released since one output from the top duplicate remains unacknowledged.

Briefly, in step four (Figure 5), the contents of `Buffer` have now been forwarded to `Buffer(5)`, and `control` has been acknowledged via the merge. `Buffer` is now empty

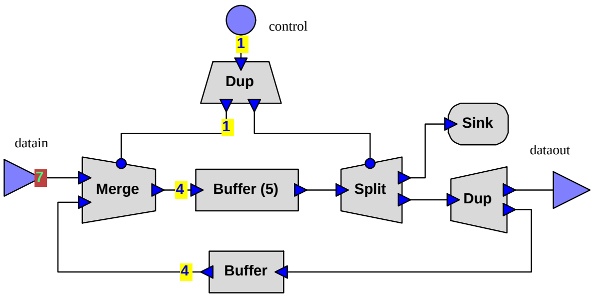


Fig. 5. State of register after four simulation steps.

and `datain` will remain blocked until a write request is received at `control`.

### Support for Hierarchies

In the editing tool, a design can be encapsulated and then used as a single node in a network at a higher level in a design hierarchy. The input and output ports of the encapsulated design will then be connected to channels in the parent network. For example, the register design from Figure 2 can be encapsulated and then included as a node in the network shown in Figure 6, so that the user is not shown details of what happens inside the register block during a simulation. Clearly, this approach is essential for large networks, otherwise the amount of information displayed would become unmanageable.

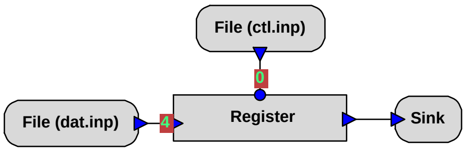


Fig. 6. Encapsulation

## SUMMARY AND FURTHER DEVELOPMENTS

We have outlined some of the main features of the Green sub-language from Rainbow, and shown how a simple micropipeline can be defined, represented pictorially, and simulated. This example includes most of the basic micropipeline elements which have been defined so far.

There is also a textual version of the (pictorial) Green described here, allowing a more conventional textual programming language style of design entry. For example, Figure 7 shows the textual Green version of the register. Further details are contained in (Barringer et al., 1996a).

A compositional semantics is defined for Green and Yellow, by compiling them into APA (Rainbow, 1996), a special process algebra that is similar to other process

```
green Register(input dataIn, readWrite;
               output dataOut) = {
 chan rdchan.

 using readWrite merge
    1: rdchan >> buffer >>
 || 0: dataIn >>
 end_merge
 >> buffer(5) >>
 using readWrite split
    1:  >> rdchan
 || 0: sink
 end_split $

 rdchan >> dataOut $
}.
```

Fig. 7. Textual Green Example: Basic Register

algebras, such as SCCS (Milner, 1989), but which has the micropipeline handshake as its communication primitive, and richly structured actions, similar to those used in EPA (Barringer et al., 1996b). In turn, APA has an operational semantics for generating derivation trees corresponding to each APA term.

The Rainbow design framework, containing a suite of formally-based sub-languages, offers an environment for describing asynchronous micropipeline designs in various styles. The sub-languages are integrated so that different parts of a design can be described in different styles, but can still be simulated and analysed as a whole. Rainbow will provide the basis for formal reasoning tools that we are developing for analysis of micropipeline designs.

It is intended that further sub-languages will be added to Rainbow:

***Red*:** supports behavioural/specification descriptions at a level above Yellow, using, for example, temporal logic or stream transformers (Broy et al., 1992).

***Blue*:** operates at a level below Green, and exposes handshaking control, similar to the CCS models of AMULET described in (Liu, 1995).

The current simulation and editing tools will be extended to cover the other sub-languages in the Rainbow framework, so that different parts of a design can be described in the most suitable way. For example, the (pictorial) Green editor/simulator described here will be integrated with textual Green and Yellow editing/simulation tools.

One other aspect which is being considered is the speed at which different elements in a design operate. The current simulation model takes no account of the passage of real time, so that only a subset of the behaviours of a design will be generated by the simulator. This is particularly so for the arbitrator, which, in the absence of data on either input channel, will choose whichever input arrives first during the simulation. This could clearly depend on the order in which the network is progressed.

## REFERENCES


Barringer, H., Fellows, D., Gough, G., Jinks, P., and Williams, A. (1996a). A formally based framework for supporting design and analysis of asynchronous hardware systems. In *Proceedings of the Northern Formal Methods Workshop*, Ilkley, UK.

Barringer, H., Gough, G., Monahan, B., and Williams, A. (1996b). A Process Algebra Foundation for Reasoning about Core ELLA. *The Computer Journal*, 39(4):303–324.

Birtwistle, G. and Davis, A., editors (1995). *Asynchronous Digital Circuit Design*. Springer.

Broy, M., Dederich, F., Dendorfer, C., and Fuchs, M. (1992). The Design of Distributed Systems: An Introduction to FOCUS. Technical Report TUM-19202, Institut für Informatik, Technische Universität München.

Burns, A. (1988). *Programming in Occam 2*. Addison-Wesley.

Furber, S. (1995). Computing Without Clocks: Micropipelining the ARM Processor. In Birtwistle, G. and Davis, A., editors, *Asynchronous Digital Circuit Design*, pages 211–262. Springer.

Herath, J., Yamaguchi, Y., Toda, K., Mattingley, R., Saito, N., and Yuba, T. (1992). Comparison of Dataflow Computing Models. In Sharp, J. A., editor, *Data Flow Computing: Theory and Practice*, pages 16–34. Ablex.

Li, W. (1982). An Operational Semantics of Tasking and Exception Handling in Ada. Technical Report CSR-99-82, University of Edinburgh.

Liu, Y. (1995). *AMULET1: Specification and Verification in CCS*. PhD thesis, Department of Computer Science, University of Calgary.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall, Hemel Hempstead, Herts, England.

Paver, N. (1994). *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester.

Rainbow (1996). The Rainbow Project Web Pages, Department of Computer Science, University of Manchester. URL: `http://www.cs.man.ac.uk/fmethods/projects/AHV-PROJECT/ahv-project.html`.

Sommerville, I. and Morrison, R. (1987). *Software Development with Ada*. Addison-Wesley.

Sutherland, I. (1989). Micropipelines. *Communications of the ACM*, 32(6).