Submission to CHDL'97

Abstract Modelling of Asynchronous Micropipeline Systems using Rainbow

Howard Barringer, Donal Fellows, Graham Gough, Alan Williams*
Department of Computer Science, University of Manchester, UK

email: {howard,fellowsd,graham,alanw}@cs.man.ac.uk

correspondence:	Professor Howard Barringer Department of Computer Science, University of Manchester, MANCHESTER M13 9PL, UK
phone:	(+44) 161-275-6248
fax:	(+44) 161-275-6211

Abstract

We consider Sutherland's Micropipeline methodology for asynchronous hardware systems and highlight problems of design representation. The Rainbow hardware design framework supports such description, simulation and analysis of micropipeline systems. It provides a set of sub-languages each offering a different description style. Components described in the different languages can be fully integrated within a single design. The framework offers abstract design description and rapid simulation at a high level for early simulation and analysis.

CHDL'97 Topics: Hardware Description Languages, Formal Methods, Design Systems and Tools.

*The University of Manchester acknowledges support from the Engineering and Physical Sciences Research Council via research grant GR/K42073.

1 Introduction

Over the past three decades, the capabilities of digital hardware systems have increased significantly, enabling the production of single-chip VLSI devices comprising millions of gates. This progress has been possible not only because of the advancements in circuit fabrication technology but also because of the development of highly sophisticated CAD support tools, so that much of the design process can be at least partially automated. Fast prototyping, early simulation and experimentation, and sophisticated design synthesis tools[1, 2] are available to the hardware engineer.

However, most chip level digital systems developed have operated synchronously, using a global clocking strategy. The speed of the circuit is determined by the fastest clocking rate which can be achieved; this in turn is determined principally by the maximum computation times of clocked stages, and by the time taken to distribute the clock signal across each device. As faster and larger systems are made possible by new fabrication techniques then it becomes increasingly difficult to design fast clocked stages and the associated clock distribution networks.

A second major concern in hardware design is power consumption, mainly because of the demand for portable battery-operated applications for which modern systems are being developed. This directly conflicts with the requirement for high speed, since the faster a synchronous system is clocked then the more power it consumes — all of the system is drawing power the whole of the time. Novel and complex schemes for slowing down the clock or switching off those parts of a system which are not required at any particular time are now being employed to increase the speed/power ratio.

Recently, there has been renewed interest in asynchronous design methodologies at the chip level[1], with the aim of obtaining the high speed but low-power operation desired. Different parts of an asynchronous system operate autonomously, with various communication strategies employed to ensure reliable data transfer between components. This means that asynchronous system designers are no longer constrained by limits in clock distribution, or by the clock period determined by worst-case delays through circuit stages. Instead the design of each computing stage can often be optimised for typical delay cases, possibly leading to much simpler designs. This saves both on design time and on circuit area, which would otherwise be required for improving the computation speed in the worst-case. Additionally, because components do not need to operate when they have nothing to compute then there will be potentially reduced power consumption. There are also new problems introduced when adopting an asynchronous design style. For example, although a designer is less constrained through the absence of the global clock, this leaves the possibility of introducing deadlock into a circuit, which can be difficult or impossible to trace, especially by conventional validation techniques which must rely on non-exhaustive simulation. Also, it is more difficult to estimate the performance of a system because the speed is no longer governed by the clock, it now being determined by the complex interaction of data-dependent delays through the separate components of the pipeline.

The advantages of asynchronous design have been difficult to demonstrate in practice, because of the relatively limited experience so far gained in asynchronous design and the lack of asynchronous design tools, in comparison to those available for synchronous systems. For example, the AMULET Group at Manchester have been investigating the design of asynchronous code-compatible versions of the ARM processor[3]. The first version of the AMULET1 processor has offered performance only comparable to its synchronous counterpart. However, this apparent lack of advantage must be considered in light of the smaller development resources available — the synchronous version has evolved over the past ten years whereas AMULET1 was designed in 3 years — and the current lack of suitable asynchronous design support tools. Furthermore the AMULET1 has been designed to be code-compatible with the synchronous ARM processor, therefore imposing constraints on the design decisions that could be made. All benchmark tests are also oriented towards synchronous systems, running the processor at maximum speed. In practice, a processor may be idle for some of the time and it is under these conditions that the

[1] In fact, many of the former Manchester computer architectures, such as Atlas and MU5, were of asynchronous design.

asynchronous design will have the advantage.

The renewed interest in asynchronous design has been supported by the emergence of novel design methodologies, in particular Sutherland's Micropipeline method[4], which simplify somewhat the design process by defining reliable methods for circuit composition — the AMULET design largely follows this style. Although certain asynchronous design methods, such as Philip's Tangram system[5], are also equipped design description languages, these have not yet emerged for micropipelines, especially for higher-level abstract description (however, a Tangram-like approach to micropipeline description is currently being investigated at Manchester). For example, with the currently available tools on the AMULET project it is necessary to virtually complete a detailed gate-level design description before full simulation of the design is possible. Accompanying analysis tools are also required for deadlock detection, for estimating performance early in the design process, and for functional verification during design transformation and implementation.

Our project is to develop such an asynchronous hardware description framework for micropipelines, called Rainbow, together with the necessary underlying formal basis to support the development of formal analysis tools. Our philosophy is to develop application-specific representations and techniques and to embed these into a conventional hardware development system. The engineer will be able to describe designs at the micropipeline level and then simulate these abstract descriptions before having to implement the full low-level circuit.

The Rainbow framework contains a suite of sub-languages offering a variety of design views, so that a multi-view description can be constructed for a single design, using the most appropriate description style for each component. The languages support structural data-flow and algorithmic control-flow descriptions at the micropipeline level. Also under development are languages for lower-level descriptions where the explicit handshake control signals are visible will also be included, and a high-level language for design specification. In the latter, design properties will be described using temporal logic and trace specifications.

In this paper, we first describe micropipelines (section 2), and then show how these have been modelled using existing languages, in particular CCS. We then introduce Rainbow (section 3) and describe how designs are modelled. Section 3.1 then contains an example based on part of the AMULET1 processor. Finally the underlying semantics for Rainbow is outlined (section 4).

2 Micropipelines

In his Turing Award Lecture[4], Sutherland describes the Micropipeline approach for achieving modularity and composibility for asynchronous design elements. A micropipeline contains buffers which use a simple request/acknowledge handshake communication in order to transfer data from one buffer to the next. Stateless functional elements are introduced between buffer stages. When one buffer has placed a data value on its output then it signals the following buffer via its request line. It then waits until it receives an acknowledge signal from the following buffer indicating that the data value has been consumed and is no longer needed. The buffer can now change the output value by accepting a new value on its input for storage. The key to this simple protocol is the *bundled-data* assumption[4]. This ensures that when a request is received, then the data value associated with the request is ready for consumption and that it will be maintained until an acknowledge is sent. In practice this means that a possibly data-dependent delay needs to be introduced so that a request signal is not sent until data is ready, having passed through any functional circuits connected to the buffer. If this assumption is satisfied then the resulting circuit elements are delay-insensitive[6], so that it is easy to compose elements operating at different processing speeds and still achieve correct functioning. Note that this would cause considerable difficulties with globally clocked synchronous circuits, because all of the stages between buffers would have to be specially designed to operate within a single clock cycle. In the asynchronous case,

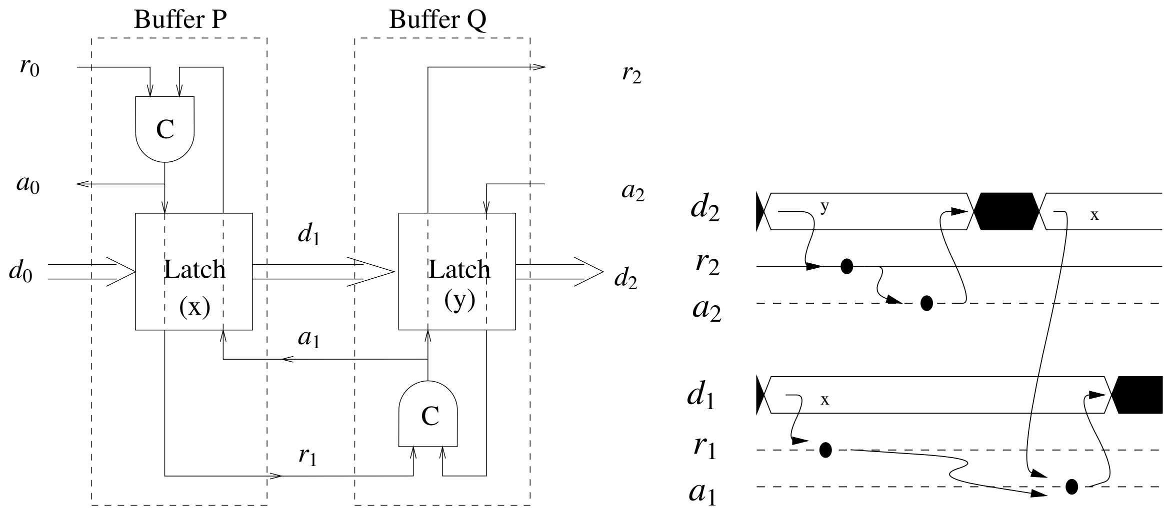

Figure 1: Basic Micropipeline

the worst that would happen if a particular component was slow to compute under certain conditions is that performance would be degraded.

Another feature of micropipeline circuits is that elastic buffers can be easily constructed, so that they can store a variable number of values and therefore assist in achieving efficient operation of a system — if at some stage one component A is operating faster than a component B which follows it, then a micropipeline FIFO can be inserted to temporarily absorb the intermediate values until B has had a chance to catch up. This frees A to continue processing. At a later stage then B may be running faster than A but can then empty the FIFO. This means that the design of components may be simplified since they can be designed to operate for an average delay with the micropipeline FIFOs acting as temporary sources and sinks.

There are new problems which now arise, due to this design style having asynchronously evolving components operating at different speeds, interspersed with elastic buffering. A major consideration is the problem of deadlocks introduced by possibly data-dependent feedback or stream combining networks. Therefore new analysis tools are required to assist the designer in avoiding this. A second consideration is the performance of the system; with components operating at variable data-dependent speeds then novel data-flow analysis tools are required so that the designer can estimate system performance at a high level. At present the only method available to the designer for performance estimation and deadlock detection is via intensive low-level simulation. At the research level, verification tools such as SMV[7] or Petri nets[8] are being developed and performance estimation techniques using timed process algebra have been investigated[9].

Let us now describe the operation of a micropipeline in more detail, in order to highlight the problems involved in design representation. Figure 1 shows a two-stage micropipeline without any intermediate function. This consists of a datapath, with two buffers holding data values x and y. Data is input via d_0 and output via d_2, transferring between the latches via d_1. The remainder of the circuit shows the request/acknowledge control, which can be implemented via *transition-signalling*[4]. The exact value of the control signals is not important, only that they have changed. It is therefore only necessary to consider *events* in the control. Micropipeline components are constructed from basic control and data elements. For example, in the two buffers, C-gates perform event conjunction on control signals: only when an event has been detected on *both* inputs does an event occur on the output. The latches then store data values, being controlled by the request/acknowledge events. Other low-level event control elements are described in [4].

Consider how the micropipeline outputs y and then moves x from buffer P to buffer Q. The signal diagram in Figure 1 shows the ordering relation between events, with bullets indicating when event occur on the control wires. First, Q makes y available on output d_2, and is then able to signal on r_2 that y is ready. The component connected to the output can receive y and when it has done this it can send an acknowledge on a_2. Buffer Q is now free to accept x on its input. Meanwhile independently buffer P has placed x on its output d_1 and has signalled a request on r_1. The acknowledge a_2 and request r_1 are combined in the C-gate to signal that x can be latched in buffer Q; then the new request is sent on r_2 and an acknowledge on a_1. Buffer P is now able to accept a new value, when the next request signal arrives on r_0. At this stage, buffer Q is now storing x and buffer P is 'empty'.

A slightly more complex circuit is shown in Figure 2. The internal details of the data-path and control-path elements have been hidden in each micropipeline component, although the external request/acknowledge controls for each communication channel are still visible. The circuit contains three buffers: the output

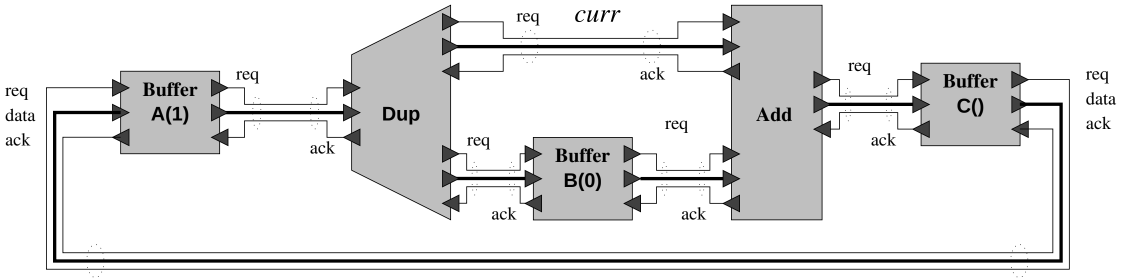

Figure 2: Micropipeline Circuit for Fibonacci Calculator

from buffer A is copied by the duplicate element Dup to both channel $curr$ and to buffer B. These are then fed into an adder function Add which writes its result into buffer C. The result is then fed back to A. Assume that A contains the value 1 and B contains the value 0. This means that the values from Dup will be consumed at different times. When Dup receives a request from A then it sends requests to both $curr$ and B. Now Dup will wait until it receives acknowledges from *both* of $curr$ and B before sending the acknowledge to A. The function element Add waits until both of its inputs are available before calculating the result and sending a request on its output to buffer C. When C accepts the value, it sends an acknowledge to Add, which can then send acknowledges both to B, and to Dup for channel $curr$. One copy from Dup has now been consumed, with A still containing the value 1, B empty and C now containing 1. In the next cycle, B can now accept the second copy of 1 produced by Dup and send back an acknowledge. Dup is now finally able to acknowledge A since both output values have now been consumed.

Since the output from C is fed back to A, the resulting circuit calculates the series of Fibonacci numbers — $curr$ supplies the current number and B supplies the last number in the series. This relies of course on the values supplied by Dup being consumed independently.

2.1 Low-level Modelling of Micropipelines

Formally based models of micropipelines usually capture the request/acknowledge control signalling between components, ignoring data as far as possible. This has been utilised with some success in [10] when constructing models of the AMULET1 processor using CCS. Having constructed these models, then the standard tools in the Concurrency Workbench[11] were used to check for deadlock.

For example, the behaviour of an initially full single-place buffer would be represented by the following CCS term:

$$Buf \triangleq \overline{rout} \,.\, aout \,.\, rin \,.\, \overline{ain} \,.\, Buf$$

where rin, ain and $rout$, $aout$ are the request/acknowledge pairs on the input and output of the buffer, respectively. Buf receives the input request rin and then immediately acknowledges this with $\overline{ain}$ thereby releasing the element connected to its input so that it can continue with further processing. The buffer then sends out a request $\overline{rout}$ to indicate that it has a data item ready for output — it then waits until it receives $aout$, indicating that the item has been consumed by the element connected to its output, before returning to its original empty state.

An initially empty buffer is represented by:

$$BufEmpty \triangleq rin \,.\, \overline{ain} \,.\, \overline{rout} \,.\, \overline{fibout} \,.\, aout \,.\, BufEmpty$$

The output action $fibout$ is present so that the CCS control model for the Fibonacci circuit contains at least one external port; otherwise it will appear to be deadlocked.

A Duplicate node would be modelled by the following CCS term:

$$\begin{aligned}
Dup \triangleq\ & rin \,.\, (\overline{r1out} \,.\, (\overline{r2out} \,.\, (a1out \,.\, a2out \,.\, \overline{ain} \,.\, Dup \\
&\qquad\qquad\qquad\qquad + a2out \,.\, a1out \,.\, \overline{ain} \,.\, Dup) \\
&\qquad\qquad\quad + a1out \,.\, \overline{r2out} \,.\, a2out \,.\, \overline{ain} \,.\, Dup) \\
&\quad + \overline{r2out} \,.\, (\overline{r1out} \,.\, (a1out \,.\, a2out \,.\, \overline{ain} \,.\, Dup \\
&\qquad\qquad\qquad\qquad + a2out \,.\, a1out \,.\, \overline{ain} \,.\, Dup) \\
&\qquad\qquad\quad + a2out \,.\, \overline{r1out} \,.\, a1out \,.\, \overline{ain} \,.\, Dup))
\end{aligned}$$

Dup receives an input request rin and then needs to interleave the output request/acknowledge signals (i.e. $rin, ain, rout, aout$) as long as rin and $rout$ precede ain and $aout$, respectively, before sending an acknowledge for its input $\overline{ain}$. Similarly the two-input adder can receive its inputs in either order:

$$Add \triangleq r1in \,.\, r2in \,.\, Add1 + r2in \,.\, r1in \,.\, Add1$$

$$Add1 \triangleq \overline{rout} \,.\, aout \,.\, (\overline{a1in} \,.\, \overline{a2in} \,.\, Add + \overline{a2in} \,.\, \overline{a1in} \,.\, Add)$$

Having received an acknowledge $aout$ for its output, representing that the value output by Add has been consumed, the agent $Add1$ sends acknowledges to both inputs, again in either order.

In order to model the circuit of Figure 2 then three instances of Buf would need to be placed in parallel with Dup and Add:

$$\begin{aligned}
Fib \triangleq\ & (Buf[ch1r/rout, ch1a/aout, ch7r/rin, ch7a/ain] \mid \\
& Dup[ch1r/rin, ch1a/ain, ch2r/r2out, ch2a/a2out, ch3r/r1out, ch3a/a1out] \mid \\
& Buf[ch2r/rin, ch2a/ain, ch4r/rout, ch4a/aout] \mid \\
& Add[ch3r/r1in, ch3a/a1in, ch4r/r2in, ch4a/a2in, ch5r/rout, ch5a/aout] \mid \\
& BufEmpty[ch5r/rin, ch5a/ain, ch7r/rout, ch7a/aout] \\
&)\backslash\{ch1r, ch1a, ch2r, ch2a, ch3r, ch3a, ch4r, ch4a, ch5r, ch5a, ch7r, ch7a\}
\end{aligned}$$

The CCS model operates at the explicit request/acknowledge level, encoding each micropipeline communication handshake directly as a sequence of reqest/acknowledge events. This means that careful attention needs to be paid during construction of the model in order to control combinatorial explosion in the state-space of the composite CCS terms, by removing request/acknowledge interleavings[12], applying similar ideas to those used in partial order state-space reduction[13].

3 The Rainbow Approach

Micropipeline modelling techniques such as CCS expose all of the internal operation of the request/acknowledge handshake, potentially leading to rapid growth in the state-space of the resulting terms. Special state-space reduction techniques must then be employed to remove as many of the interleavings as possible.

The strategy is to make use of the generic expressiveness of CCS by capturing the micropipeline handshake via a suitable encoding, and then to analyse the result with the aim of producing tractably sized models.

Our approach is to start by constructing a set of design representation languages so that a user can describe a system using the most appropriate view. For micropipeline designs then a structural data-flow representation is provided, where the micropipeline handshake communication is used as the *atomic* communication in the language, its behaviour being captured directly by the semantics of the language. This represents an abstract modelling approach as opposed to one using encoding/hiding. It means that data transfer is considered as an atomic event within the model, instead of encoding it as a sequence of request/acknowledge events. Therefore each channel in the language models a complete bundled-data micropipeline connection, abstracting away from the explicit request/acknowledge/data channels visible in the CCS model. By adopting this approach we aim to construct smaller models from the outset, instead of having to continually apply state-space reduction techniques to control the growth. These reductions essentially come for free by being inherent in the language. Furthermore, since we capture dataflow, we have functional analysis capabilities, as well as control analysis.

We have developed a language called Green for directly describing micropipeline networks. This static dataflow language uses explicit fixed length buffers for introducing state, together with stateless functional elements and data stream controllers for controlling the flow of data through the various pipelines. A textual version of Green allows the designer to define complex data types and expressions in the usual way, as well as providing a dataflow network description constructs. It uses somewhat similar primitives to other dataflow languages[14], including merge and split for dataflow control, arbitrate for (internal) non-deterministic stream merging, as well as buffers and pure functional units. A *Table* construct provides a large degree of flexibility in the description of dataflow components, so that an element can control the consumption and generation of data values on its input and output streams. The network descriptions can also be constructed using a graphical version of Green resulting in hierarchical dataflow diagrams of a design, possibly with embedded textual fragments. The two formats of the language are interchangeable.

In addition to these structural dataflow languages, other views of a design can be created by using different specialised languages. The Yellow language supports algorithmic style descriptions, by including sequencing. It is a CSP-like language, but uses an Ada-like *rendezvous* communication. Again, communication between elements uses micropipeline handshaking.

The Red language will support design specification, utilising trace descriptions and/or temporal logic to describe system behaviour abstractly. In contrast, the Blue language will actually expose the internal working of the micropipeline communication, allowing the designer access to the request/acknowledge control signalling, resulting in descriptions similar to those shown for CCS.

These languages will be fully integrated at the semantic level, so that design analysis can be conducted in any Rainbow fragment. This also allows mixed-view descriptions of a design to be constructed and simulated within Yellow, Green and Blue. For example, a Green network description will be able to instantiate a Yellow description in one of its components, or vice versa. This reflects the way in which hardware engineers are observed to develop a design description: a structural architectural diagram is built showing the overall structure of a system; this could be given in Green in our system. The detailed implementation of individual blocks is not fully described at this stage, but just their externally visible behaviour is described; Yellow would be used for these parts. At this stage, the design can be simulated so that the engineers can experiment with different configurations at an early stage. Then, each sub-block can be developed in turn, replacing the Yellow behavioural descriptions with Green network implementations. Integration of the different sub-languages of Rainbow is achieved via the underlying uniform formal semantics, provided by APA as described in section 4.

As an example, the micropipeline circuit in Figure 2 can be represented by the Green network shown in

Figure 3, where each request/data/acknowledge bundled data connection is replaced by a single channel. This gives the complete description of the system, i.e. including the datapath, so that the same model can be used for simulation, analysis, functional and control verification, and later for translation into hardware. Figure 3 shows the annotated design during a simulation; the channel labels with a light

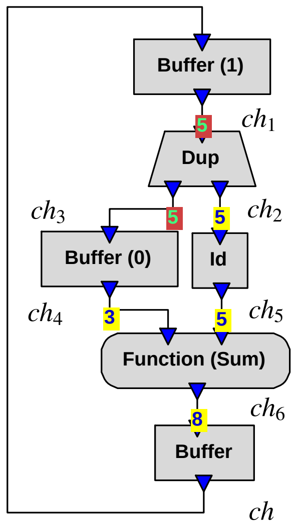

Figure 3: Visual Green Network for Fibonacci Calculator

background indicate values that have successfully been transferred, and the labels with a dark background show values that are ready to be transferred but are currently blocked. Refer to [15] for more details of the Green editing and simulation tools.

Alternatively, an equivalent textual version of Figure 3 can be given:

```
green Fib(value a,b,c) = {
chan ch.
fun Add(x,y) = x+y.
ch >> Buf(a) >> Dup() >>
      { >> | Buf(b) } >> Add()
          >> Buf(c) >> ch $
} end.
```

The pipe operator '>>' simply passes values from the output(s) of one elements to the input(s) of the next element. If there is more than one input/output stream then these can evolve independently. For example the '`Dup`' element generates two output streams containing copies of its input — these values are received independently by the following elements, one an identity indicated by >> and the second buffer `Buf(b)`, which are placed in parallel using '|'. A local channel `ch` connects the output buffer to the input. The user-defined function `Add()` adds its two inputs together. The following is part of a trace giving the behaviour of `Fib(2,1,-)`; it has no inputs or outputs so there are no values labelling the transitions. The behaviour of the circuit is therefore monitored by examining the values stored in the 3 buffers; the three arguments to *Fib* represent the states of buffers A, B and C, respectively:

$$
\begin{aligned}
&Fib(2,1,-)\\
&\qquad \longrightarrow Fib(2,-,3) \longrightarrow Fib(-,2,3) \longrightarrow Fib(3,2,-)\\
&\qquad \longrightarrow Fib(3,-,5) \longrightarrow Fib(-,3,5) \longrightarrow\\
&\qquad Fib(5,3,-) \longrightarrow \boxed{Fib(5,-,8)} \longrightarrow Fib(-,5,8)(*)\\
&\qquad \longrightarrow Fib(8,5,-)
\end{aligned}
$$

Here, '$-$' indicates that a buffer is empty. Notice how first of all a value is written into buffer C, consuming the value stored in buffer B, together with one of the copies produced by the 'Dup' element, leaving buffer B empty. Then, the second copy is written to buffer B, with buffer A emptying at the same time. Buffer C can then write the result back to buffer A and the evaluation cycle can begin again. The boxed state '$\boxed{Fib(5,-,8)}$' corresponds to that shown in Figure 3; the derivation of the two transitions in row $(*)$ is described in more detail in section 4.

This behaviour is in contrast to that which would be produced using an encoding of micropipeline communication such as CCS, where the interleaving of all of the request/acknowledge signals would have to be generated and then removed by considering equivalences between states. This is achieved directly in Rainbow by defining a micropipeline behaviour for components and composition operators.

3.1 Abstract Model of Address Interface

Figure 4 shows a larger Rainbow design example based on a simplified version of the AMULET1 address interface[16]. This is used to illustrate how mixed-view hierarchical descriptions can be created in Rainbow[2]. The design generates increasing sequences of PC instruction addresses, output to both `PC Pipe` and `Memory`, by cycling values in the `Inc` — `PCStore` — `MAR` loop, with `Inc` incrementing the value each time. Also, new address or data values can be input asynchronously via the `Write` and `A Bus` inputs, for output to the memory or to generate fresh instruction address; here, the necessary control information is also supplied along with the address/data values. An arbiter is therefore required to merge the circulating PC values with the new inputs. Data values do not enter the loop. Multiple load/store

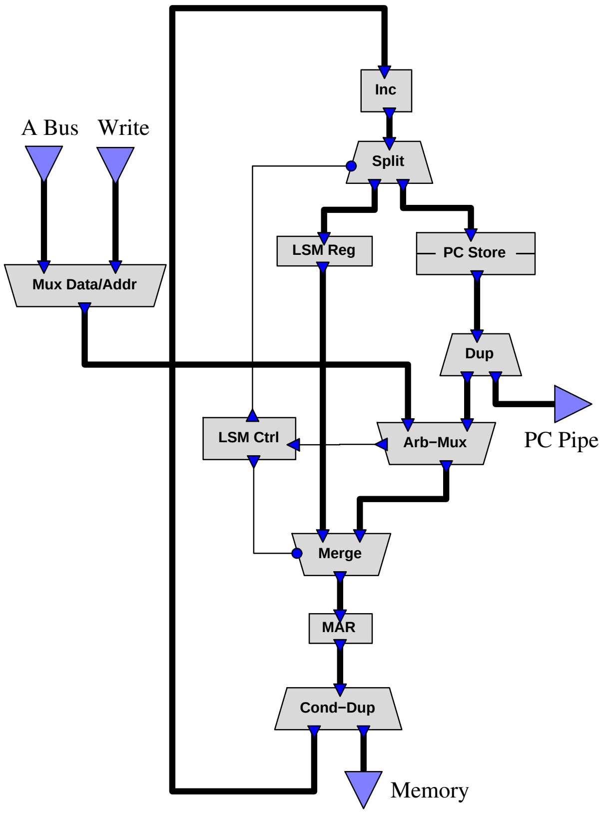

Figure 4: Visual Green Network for Address Interface

[2]A structural description of the address interface is given here, and so the design contains schematic and textual fragments of Green.

(LSM) memory addresses are calculated by inserting a base address into the loop and then following the 'Inc — LSM Reg — MAR' loop for the required number of times (which we assume to be fixed here), while the PC generation is suspended. Once finished the LSM addresses is discarded and PC generation resumes. A branch is performed by inserting the branch address into the loop and discarding the old PC value.

In this Green implementation the `ArbMux` block acts as the main dataflow controller. We concentrate on this block and give a textual Green description, containing dataflow fragments, tables and functions:

```
type OldNew =  enum(old,new).           type ValTag =  enum(black, white, newdata, lsm).
type LSMcount = [0..15].                type LSMCtrl = enum(pcloop,lsmloop,throwaway).
type address = [0..1024].               type data =    [0..1024].
type val =     union(address,data).     type tag_val = (ValTag, val).
type tagged_tagval = (OldNew, tag_val).

green ArbMux (input oldpc: address;
              input newval: val;
              output outval: val;
              output lsmctrl: tag_val) = {
 chan arbval: val.

   arbitrate origin(old, oldpc) || origin(new, newval) end_arbitrate
       >> arbval$
    {arbval || oldcolour } >> ArbMuxControl >>
          {newaddr || buffer(2,black) >> oldcolour} $
    {arbval || counter()} >> LSMControl >> { outval || lsmctrl } $
} end
```

`ArbMux` merges the PC stream `oldpc` and the new input stream `newval` using an arbiter; a flag is attached so the following controllers can determine the source of the arbiter output `arbval`. Figure 5 shows the table `ArbMuxControl`, which determines whether `arbval` should be passed on or thrown away. The PC value is tagged with a 'colour' which changes when a branch is taken, so that obsolete PC values circulating in the loop can be discarded. Also, when an LSM operation finishes, the final address needs to be thrown away. Each row in the table contains an input pattern (to the left of the '=>'), to be matched

```
 table ArbMuxControl (input newval: tagged_tagval;
                      input oldcol: ValTag;
                      output newaddr: val;
                      output newcol: colour) = {

  var v: val.
  var nc: ValTag.
  // Table rows:
     (old, (lsm, v)),     -                         => (lsm, v),     -
  or (old, (nc, v)),      oldcol when (nc == oldcol)    => (nc, v),      oldcol
  or (old, (nc, v)),      oldcol when not(nc == oldcol) => - ,           oldcol
  or (new, (lsm, v)),     oldcol                    => (lsm, v),     oldcol
  or (new, (nc, v)),      oldcol                    => (nc, v),      nc
  or (new, (newdata, v)), -                         => (newdata, v), -
 } end
```

Figure 5: Green Table for `ArbMuxControl`

against values on the input streams. The `when` clauses in the specify additional guards. When there is a match, then the values for the output streams are generated, determined by the expression list to the right of '=>'. Each pattern contains literals such as `old`, variables such as `oldcol` that bind to the value on the same-name input stream, or '-' which means that a value is not required on that particular input channel — if a value is present on a channel whose entry contains '`-`' in a successful pattern-match, then that input will simply not be consumed. The outputs in each row consist of expressions which utilise the variables bound in that row. Alternatively, '`-`' can again be used to mean that no value is to be written to an output stream. For example, the last row in `ArbMuxControl` is executed when the pattern `(old, (newdata, v))` is on input `newval`; no value is required on input `oldcol`. In this case, the value `(newdata, v)` is output to channel `newaddr` but no value is output to channel `newcol`. This row is executed when a new data value has been input to the address interface and simply needs to pass out to memory.

The other parts of `ArbMux` will not be described in detail. Table `LSMControl` controls whether address values follow the PC LSM loop. The Green procedure `counter` determines how many LSM addresses are generated, and the function `origin` simply attaches a source tag to each value. The full description for `ArbMux` appears in the appendix.

It is interesting to observe that the kind of schematic shown in Figure 4 also reflects the way the AMULET engineers wish to view a design when mentally analysing datapath activity and explaining its operation. For example, Figure 6 shows a typical diagram as used in the literature for AMULET1[16].

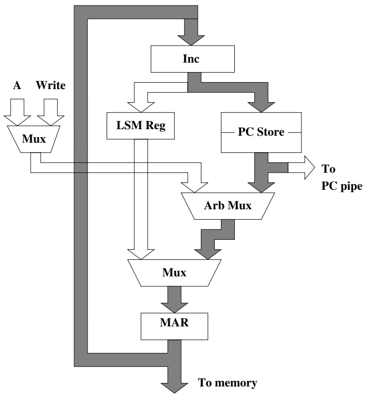

Figure 6: Diagram of Address Interface used by AMULET

4 Formal Semantics of Rainbow: APA

APA is a process algebra designed for representing the behaviour of asynchronous systems described in Rainbow. It has some similarities to standard process algebras, such as CCS[17], CSP[18] or LOTOS[19], using some familiar process operators and a semantics defined operationally using SOS-style transition rules. However, the composition operators have been designed to support bundled-data micropipeline communication — the communication primitives involved use an Ada-like rendezvous, and their semantics resembles the Ada semantics presented in [20]. APA supports value-passing and has richly-structured transition labels, similar to those used in our previous work on the hardware design language ELLA and EPA[21]. For example, the action '$(ch_4?5, ch_5?3 \ / \ ch_6!8)$' shows the values 5 and 3 being input on channels ch_4 and ch_5 respectively, with the simultaneous output on channel ch_6 of the value 8. To aid readability, when both input and output bindings are needed, they are separated by ' / '. These features lead to a compact semantic representation of Rainbow designs, providing the basis for simulation formal analysis. The development of APA extends our work of developing application-specific process algebraic semantic representations which naturally express the semantics of the source language at the most suitable level. This reduces the amount of encoding which would be necessary if already existing general process algebras were used, helping to clarify rather than obscure the intended meaning of Rainbow.

We make a distinction between the process terms, which model hardware system connectivity, and the embedded action algebra for modelling data. The actions are input/output maps, from channels to value expressions. There is also a 'zero' action $\boxtimes$ for representing 'chaos' – any trace containing a zero action will be considered to be improper and will be ignored during subsequent analysis. This simplifies the definition of the APA transition rules.

The constructs available in APA reflect those available in Rainbow, including buffers, duplicators, merge/split and functions. The combinators include parallel, pipe and process call, which behave in a similar way to the corresponding operators described in other standard process algebras such as CSP or CCS. For example, the APA parallel operator combines the transition labels of the two component processes by matching input and output channels, as in CCS. However, its operation is slightly more complicated because of the map structure of the transition labels; if the values carried by same-name channels do not match then a chaos transition labelled by $\boxtimes$ results.

For Green communication, APA provides the pipe operator with implicit channel naming, as well allowing explicitly named channels. Instead of using explicit control channels, micropipeline communication is modelled directly by introducing 'postponed' input/output actions. When all of the inputs to the construct are available, then the body can begin execution and the outputs are generated. Consumption of the inputs is then postponed until the body has completed and all of the outputs have been consumed.

APA is used to provide a single uniform semantics for all of the *Rainbow* sub-languages, thereby supporting interworking — suitable translations from *Rainbow* into APA are defined. As may be expected, much of the translation is straightforward, since many Rainbow components have exact counterparts in APA. Channel declarations and scoping rules simply translate into channel hiding in APA.

To illustrate the behaviour of APA and its relation to the Rainbow languages, we use the Green network in Figure 3 for the Fibonacci calculator, and derive the transition from '$Fib(5,3,-)$' to '$Fib(-,5,8)$'. We give instances of transition rules for the various basic elements used and then compose these using rules for pipe and parallel[3]. The following transition rules show how a full buffer can output its value to become empty, and vice versa[4]:

$$\mathcal{B}[3] \xrightarrow{ch_3!3} \mathcal{B}[-] \qquad\qquad \mathcal{B}[-] \xrightarrow{ch_6?8} \mathcal{B}[8]$$

We can also perform a 'postponed' action, such as '$(ch_1!5)$', where a process term indicates that it can *potentially* perform a particular transition, but it does not actually change state:

$$\mathcal{B}[5] \xrightarrow{(ch_1!5)} \mathcal{B}[5]$$

The 'Dup' process can produce its outputs independently, storing the other unconsumed action for output later:

$$\mathsf{Dup} \xrightarrow{(ch_1?5)\ /\ ch_2!5} \mathsf{Dup}_{ch_3!5}$$

Note that the input action is postponed and is therefore not consumed at this stage. The functions simply accept their input value(s) and calculate the output in a single step:

$$Add \xrightarrow{ch_4?5, ch_5?3\ /\ ch_6!8} Add \qquad\qquad Id \xrightarrow{ch_2?5\ /\ ch_5!5} Id$$

These components can now be assembled using pipe and parallel operators, whose behaviours are illustrated in the following transition rules:

[3]Further details of Rainbow are available from the project web-site, which is regularly updated, URL: `http://www.cs.man.ac.uk/fmethods/projects/AHV-PROJECT/ahv-project.html`

[4]For the purposes of this paper, we present each component transition with the input/output channels already renamed, so that it is easier to match corresponding actions. For example, in the first buffer transition, the output channel for the buffer has already been renamed ch_3, and in the second the input channel to the buffer has been renamed ch_6.

$$\boxed{\mathsf{Pipe}}\ \frac{\mathcal{B}[5] \xrightarrow{(ch_1!5)} \mathcal{B}[5] \qquad \mathsf{Dup} \xrightarrow{(ch_1?5)\ /\ ch_2!5} \mathsf{Dup}_{ch_3!5}}{\mathcal{B}[5]\ >>\ \mathsf{Dup} \xrightarrow{ch_2!5} \mathcal{B}[5]\ >>\ \mathsf{Dup}_{ch_3!5}}$$

$$\boxed{\mathsf{Par}}\ \frac{Id \xrightarrow{ch_2?5\ /\ ch_5!5} Id \qquad \mathcal{B}[3] \xrightarrow{ch_4!3} \mathcal{B}[-]}{Id \mid \mathcal{B}[3] \xrightarrow{ch_2?5\ /\ ch_5!5, ch_4!3} Id \mid \mathcal{B}[-]}$$

For the parallel operator, each component can evolve independently. For the pipe operator, the output(s) of the first component must match the input(s) of the following component. These separate stages can now be assembled, again using the pipe rule:

$$\boxed{\mathsf{Pipe}}\ \frac{\mathcal{B}[5]\ >>\ \mathsf{Dup} \xrightarrow{ch_2!5} \mathcal{B}[5]\ >>\ \mathsf{Dup}_{ch_3!5} \qquad Id \mid \mathcal{B}[3] \xrightarrow{ch_2?5\ /\ ch_5!5, ch_4!3} Id \mid \mathcal{B}[-] \qquad Add \xrightarrow{ch_4?5, ch_5?3\ /\ ch_6!8} Add \qquad \mathcal{B}[-] \xrightarrow{ch_6?8} \mathcal{B}[8]}{\mathcal{B}[5]\ >>\ \mathsf{Dup}\ >>\ (Id \mid \mathcal{B}[3])\ >>\ Add\ >>\ \mathcal{B}[-]\ >> \xrightarrow{\varepsilon} \mathcal{B}[5]\ >>\ \mathsf{Dup}_{ch_2!5}\ >>\ (Id \mid \mathcal{B}[-])\ >>\ Add\ >>\ \mathcal{B}[8]}$$

Therule has been applied several times, so that all of the components are 'connected'. The effect of the transition is for the adder to compute the next Fibonacci number (5+3) and store this in buffer C, with buffer B emptying. One of the values from the Dup element has been consumed, the second is ready to be written to buffer B in the next step. The two transitions for calculating the next Fibonacci number are as follows:

$$\begin{aligned}
&\mathcal{B}[5]\ >>\ \mathsf{Dup}\ >>\ (Id \mid \mathcal{B}[3])\ >>\ Add\ >>\ \mathcal{B}[-] \\
&\quad \xrightarrow{\varepsilon} \mathcal{B}[5]\ >>\ \mathsf{Dup}_{o_2!5}\ >>\ (Id \mid \mathcal{B}[-])\ >>\ Add\ >>\ \mathcal{B}[8] \\
&\qquad \xrightarrow{\varepsilon} \mathcal{B}[-]\ >>\ \mathsf{Dup}\ >>\ (Id \mid \mathcal{B}[-])\ >>\ Add\ >>\ \mathcal{B}[8]
\end{aligned}$$

5 Summary

We have outlined the problems encountered in hardware design and the potential benefits of adopting an asynchronous design style in comparison to the traditional synchronous approach. In particular, Sutherland's Micropipeline design methodology is considered. However, the advantages of asynchronous design are not observed in practice, due to the current lack of adequate design representation languages and support tools comparable to those available for synchronous design. The problems encountered when applying generic description languages are illustrated, leading to inefficient representations because of the low level at which the encoding must be given. We introduce the Rainbow design framework in order to raise the level of description to that of micropipelines — the primitives of the languages operate at the micropipeline level. The potential advantage of this approach has been illustrated by showing the efficient semantic representation achievable. Rainbow offers mixed-view descriptions so that different parts of a design can be described in the most appropriate way. Smooth interfacing between elements is attained by defining the semantics uniformly via a single underlying process algebra APA. Prototype design editing and animation tools have been developed for the Green fragment of Rainbow; integration with Yellow is in progress. APA/Rainbow will also provide the basis for the development of formal analysis tools.

6 References

[1] J.D. Morison and A.S. Clarke. *ELLA2000: A Language for Electronic System Design*. McGraw-Hill, Maidenhead, Berkshire, England, 1993.

[2] Louis Baker. *VHDL Programming With Advanced Topices*. Wiley, 1993.

[3] S. Furber. Computing Without Clocks: Micropipelining the ARM Processor. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, pages 211–262. Springer, 1995.

[4] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6), June 1989.

[5] K. van Berkel. *Handshake Circuits: an Intermediary between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[6] A.J. Martin. Programming in VLSI: From Communicating Process to Delay-Insensitive Circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, pages 1–64. Addison-Wesley, 1990.

[7] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Technical Report CMU-CS-92-131, Carnegie Mellon University, May 1992.

[8] A. Yakovlev, V. Varshavsky, V. Marakhovsky, and A. Semenov. Designing an Asynchronous Pipeline Token Ring Interface. In *Asynchronous Design Methodologies*, South Bank University, London, May 1995.

[9] Chris Tofts. Some Formal Musings on the Performance of Asynchronous Hardware. Technical Report UMCS-96-2-2, Department of Computer Science, University of Manchester, February 1996.

[10] Ying Liu. *AMULET1: Specification and Verification in CCS*. PhD thesis, Department of Computer Science, University of Calgary, September 1995.

[11] R. Cleaveland, R. Parrow, and J. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. Technical Report ECS-LRCS-89-83, LFCS, University of Edinburgh, August 1989.

[12] C. Tofts and Y. Liu and G. Birtwistle. State Space Reduction for Asynchronous Micropipelines. In *Proceedings of the Northern Formal Methods Workshop*, Ilkley, UK, September 1996.

[13] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Number 1032 in LNCS. Springer-Verlag, 1996.

[14] Carlos Delgado Kloos. *Semantics of Digital Circuits*. Number 285 in LNCS. Springer-Verlag, 1987.

[15] H. Barringer, D. Fellows, G.D. Gough, P. Jinks, B. Marsden, and A. Williams. Design and simulation in rainbow: A framework for asynchronous micropipeline circuits. In *Proceedings of the European Simulation Symposium (ESS'96)*, Genoa, Italy, October 1996.

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

[17] R. Milner. *Communication and Concurrency*. Prentice Hall, Hemel Hempstead, Herts, England, 1989.

[18] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[19] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[20] Wei Li. An Operational Semantics of Tasking and Exception Handling in Ada. Technical Report CSR-99-82, University of Edinburgh, January 1982.

[21] H. Barringer, G.D. Gough, B.Q. Monahan, and A. Williams. A Process Algebra Foundation for Reasoning about Core ELLA. *The Computer Journal*, 39(4):303–324, 1996. Also see reports available on the World-Wide Web, URL: `http://www.cs.man.ac.uk/fmethods/projects/ella-project.html`.

A Textual Green **ArbMux** for Address Interface

```
type OldNew = enum(old,new).            type ValTag = enum(black, white, newdata, lsm).
type LSMcount = [0..15].                type LSMCtrl = enum(pcloop,lsmloop,throwaway).
type address = [0..1024].               type data = [0..1024].
type val = union(address,data).         type tag_val = (ValTag, val).
type tagged_tagval = (OldNew, tag_val).

green ArbMux (input oldpc: address;
              input newval: val;
              output outval: val;
              output lsmctrl: tag_val) = {

 table ArbMuxControl (input newval: tagged_tagval;
                      input oldcol: ValTag;
                      output newaddr: val;
                      output newcol: colour) = {

  var v: val.
  var nc: ValTag.
  // Table rows:
     (old, (lsm, v)),     -                            => (lsm, v),     -
  or (old, (nc, v)),      oldcol when (nc == oldcol)    => (nc, v),      oldcol
  or (old, (nc, v)),      oldcol when not(nc == oldcol) => - ,           oldcol
  or (new, (lsm, v)),     oldcol                        => (lsm, v),     oldcol
  or (new, (nc, v)),      oldcol                        => (nc, v),      nc
  or (new, (newdata, v)), -                             => (newdata, v), -
 } end

 table LSMControl (input ival: tagged_tagval;
                   input count: LSMcount;
                   output oval: val;
                   output lsmctrl: LSMCtrl) = {
  var v: val.
  var nc: ValTag.
  var on: OldNew.
     (old,(nc, v)),  -     when (nc == oldcol)    => (nc, v),   pcloop
  or (old,(nc, v)),  -     when not(nc == oldcol) => - ,        throwaway
  or (on, (lsm, v)), count when (count == 0)      => -,         throwaway
  or (on, (lsm, v)), count when (count > 0)       => (lsm, v),  lsmloop
  or (on, (nc, v)),  -                            => (nc, v),    pcloop
 } end

 green counter (value max; output cntr: LSMcount) = {
   function notmax(value x): bool = {x < max}.
   function inc(value x): integer = {x+1}.
   {source(0) || i} >> while notmax() do inc() >> cntr end_while
 } end

 function origin(value tag: OldNew; value v: val): tag_val = {
     (tag, v) }.
 chan arbval: val.

   arbitrate origin(old, oldpc) || origin(new, newval) end_arbitrate
       >> arbval $
    {arbval || oldcolour } >> ArbMuxContrl >>
          {newaddr || buffer(2,black) >> oldcolour} $
    {arbval || counter()} >> LSMControl >> { outval || lsmctrl } $
} end
```