The Green Component of the Rainbow Hardware Description System

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

October 1996

By
Donal K. Fellows
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Recently there has been much work in the field of asynchronous hardware, but these efforts have been largely without a formal basis or structured tool support. At the same time, much research has been focussed on the theoretical properties of asynchronous systems, though this work has mainly concentrated on software applications.

The *Rainbow* project is an attempt to create a formally-based language framework that permits both rapid development and testing of micropipelined hardware systems, and formal verification of the same systems without the transcription errors and tractability problems that abound when a final circuit layout is translated into a logic like CCS.

This document discusses the *Green* language component of *Rainbow*, which presents a dataflow view of micropipelined hardware, both in a pictorial and a textual form. Also discussed is a development environment for *Green*.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Chapter 1

Introduction

The focus of single-device digital hardware design for the past thirty years has been on synchronous systems, where the operating philosophy is that of allowing a fixed time between each system state change and then forcing the state of the whole system to update at the same moment. This leads to a system that is fairly easy to conceptualise when the design is small. For large designs, there are problems, some of which are highlighted in [Fur95].

One of the main problems is the distribution of the clock signals used to control when the state of all the devices change. These clock signals need to be delivered to all the latches on the microprocessor at about the same time so that the states of all the inputs to the latches are stable and consistent. Failure to distribute the clock signals correctly leads to clock skew and devices with poorly defined behaviour, particularly at high clock speeds. This results in the clock distribution circuitry occupying up to a quarter of the silicon area, when this might be put to better use elsewhere. With a large microprocessor, there can be on the order of a million logic gates changing in step, even if a lot of this switching does not actually have a useful effect from the point of view of executing instructions. With that many gates switching in step at high frequencies, the power requirement of these microprocessors becomes significant in relation to the system as a whole, especially in embedded and low-power applications.

With synchronous design, another of the major concerns is making sure that the execution of the data processing elements between each latch stage can be accomplished within the clock cycle time (after allowing for signal stabilisation time). This makes designers concentrate on building all their computation stages so as to always terminate within that fixed length of time (or a multiple of it), resulting in an increase in design complexity due to the break up of otherwise simpler components into portions which can always reach a stable state within one clock cycle, with the interfaces between the components not always being well defined.

The concern with making sure all the timings fit with each other also adversely impacts upon the time to develop optimisations. This is because it is necessary to increase the speed of every single calculation stage in the design in order to increase the overall speed of the system. There is no easy route to optimising just the most heavily used parts of the design which could otherwise speed up the average processing, only slowing things down when rarer events happen.

Synchronous designers have a number of solutions to these problems. In particular, the use of gated clocks (where the clock signal is only propagated to those parts of the circuit where processing is taking place) and sleeping circuits (where power is only delivered to active portions of the circuit) are used to cut power consumption. The use of specialised tools to support synchronous design also helps a lot with the management of the system complexity, but there are still problems with the total area of the device devoted to timing signal propagation and control.

1.1 The Asynchronous Solution

One of the solutions to these problems was to turn to asynchronous hardware design[BD95], where there is no global clocking strategy, instead having separate modules operating independently. This solves the problems with global clock distribution by getting rid of global clocks, increases modularity by encouraging the development of systems with well-defined interfaces, and allows the use of slower designs in non-critical sections, since the other parts of the design can collect the result when it arrives and continue performing useful processing in the meantime. The increased modularity also encourages rapid development by allowing the use of simpler (and slower) designs during initial development even in critical sections, while allowing the substitution of high-performance implementations of the same functionality at a later stage in development. Indeed, the modularity can even allow sections of a design to be described in a wholly algorithmic style so that other parts of the design can be developed first, providing a much faster turnaround in the early parts of the design cycle.

There are many different styles of design for asynchronous hardware[Asy95], including self-timed circuits[McM92], the handshake circuits of Tangram[vB92], and Sutherland's micropipelines[Sut89]. Indeed, it is easier to consider the synchronous circuit design method as a kind of asynchronous scheme (where every functional unit shares a control signal, the clock, with all the others) than to split the world into synchronous circuits and asynchronous circuits. This also means that most of the solutions developed for synchronous designs also apply in the asynchronous domain, some coming naturally from the type of circuit itself, and some through the use of design tools.

The main problems with asynchronous design are that there is relatively little experience with it, particularly within a commercial setting, and consequently, only a small community working in the field. This means that there is a general lack of representations for the efficient description of asynchronous hardware designs, and a lack of tools to support the development of asynchronous hardware. It is these last two problems that *Rainbow* is addressing.

One of the main problems that hardware designers have with the current toolsets when designing asynchronous hardware is that the tools that they are using tend to have been developed for use in the synchronous domain. In particular, VHDL[Coe89] has a very large number of tools available commercially for it, but it remains difficult to produce asynchronous hardware with VHDL since the semantics of the language itself is not oriented towards channel-based communication. Instead, VHDL is based on communication via shared wires, and it has little support for fine-grained parallelism which could be used to otherwise handle the cases where single processes have to produce output values on multiple channels at the same time. VHDL is also a very large language, and providing formal support for it is a daunting task in terms of its scale at the very least.

1.2 Micropipelines

One of the emerging design methods for asynchronous hardware is Sutherland's micropipelines (see Figure 1.1 for a four-phase bundled-data implementation). These are built with small functional units interacting with each other only through the use of bundled-data communications channels based on the idea of using a request signal from the transmitting end of the channel to indicate that there is data available on the channel, and an acknowledge signal from the receiving end of the channel to indicate that the data has been received and processed, with the data being guaranteed valid between the request and the acknowledgement. These communications semantics are usually implemented using a cell library that provides the basic semantics of the control communications, with the data being passed using data buses as in conventional synchronous design.

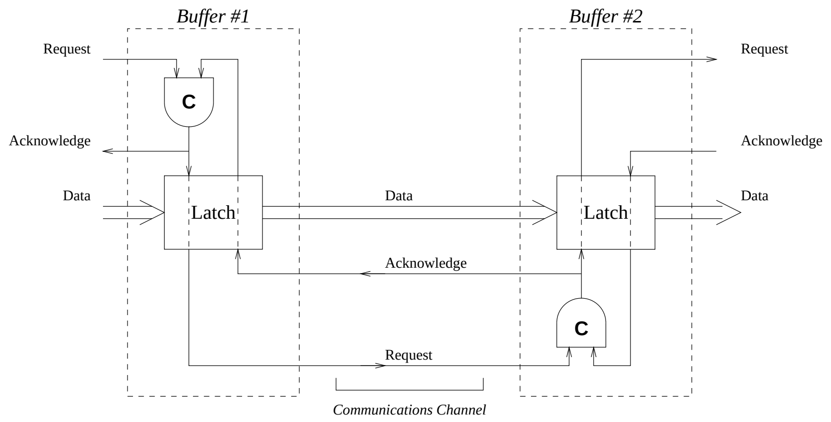


Figure 1.1: Basic Micropipeline Channel

As can be seen from Figure 1.1, a channel communicating between two buffers implemented by latches, the acknowledgement on the communications channel is only generated once both a request is received and the latch has loaded the value from the data bus into itself. As this is a four-phase (level-sensitive) design[1], the request signal can be regarded as the request line going high and the acknowledge signal can be regarded as the acknowledge line going low.

Another way of looking at micropipelined hardware is to think of it as a dataflow language[CDJ84], where each stage in a micropipeline corresponds to a node in a data-flow network. The main differences between micropipelines and dataflow languages are in the fact that with hardware, the flow network has to be static and the amount of buffering on any of the communications links has to be finite. This is because hardware is not usually dynamically reconfigurable, and the quantity of silicon on which to implement any particular design is finite. This leads to the conclusion that it is easier to have explicit buffering nodes, especially as that follows the manner of thinking of the designers more closely (they want to know exactly where all the buffering is in a design, particularly since a lot of the deadlocks in an asynchronous design are due to queues of buffers being full, when

[1]Other ways of communicating between the different components in a micropipelined design are possible [vB94] but the differences can be abstracted away to leave a communication process approximately as described above.

there are more values to put in that queue, before the queue can be emptied).

The AMULET project at Manchester is building microprocessors based upon micropipelines, but the current design methodologies used are highly informal. For formal verification (such as deadlock checking on a CCS model of their design[Liu95, TLB96]) to be of use, it really needs to be used by the designers themselves on partial designs before the architecture is finalised — while changes are still relatively simple and cheap to implement — as opposed to after the production of the final circuit (at either the transistor level or the silicon level) has commenced. It is the intention for *Rainbow* [BFG+96b] to provide a framework on which the AMULET3 microprocessor is built, allowing the production and testing of initial designs without the significant time delays involved with producing designs at the circuit diagram level (the previous principal level of working).

1.3 Introducing *Rainbow*

There are a number of approaches to the formal modelling of asynchronous hardware design, each of which has its own limitations.

Tangram[vB92] is a static asynchronous hardware design language based on CSP[Hoa85]. It offers direct synthesis and simulation, and is specially targeted towards low-power applications.

Its main problems occur because it is a very strongly algorithm-oriented language, encouraging all descriptions to be made in a highly sequential form, making it difficult to add more parallelism to a design. It also does not produce micropipelined hardware, and in practice substantial parts of the designs produced use a data-pulling communication style (decreasing the potential efficiency, since functions cannot begin evaluation until the acknowledgement phase of the communication). There are also licensing problems with Tangram.

CCS[Liu95] has been used to successfully model the control of the AMULET1 microprocessor, but it suffers from not being able to model data[2] resulting in the reporting of deadlocks that would trivially be excluded otherwise. The fact that it does not model the semantics of micropipelined communications directly also counts against it, as this results in models that can grow to be considerably more complex than the design itself. On the other hand, it is a well-understood language with a number of tools developed to use it already[Mol94].

The *Rainbow* [Rai96] hardware description framework aims to provide a system for the design, analysis and production of micropipelined hardware, with an especial focus on microprocessors. As part of this, there are a number of principal components:

The Data/Type subsystem encompasses all the data values that can be communicated via a channel between the the sub-language components.

Green which provides a dataflow view of micropipelines.

Yellow which provides an algorithmic control-flow view (similar to Tangram and CSP, though with communications semantics more closely suited to working with micropipelines)

Blue which exposes the handshaking signals.

Red which works entirely on the basis of streams as opposed to individual values from them.

[2] Or at least not being able to model data in an efficient manner. An encoding can be specified, but without specialised tool support for this as well, adding data to a model renders the model itself extremely unwieldy.

Rainbow communications are strictly pushing, with data-accepting processes having no way to signal to data-producing processes that they are willing to accept data. Instead, they must wait until the producers signal that some data is available, and only send a return signal to indicate that the data has been consumed. This follows the communications style of micropipelines very closely.

1.4 This Document

This document discusses *Green* and the *Rainbow* development environment. In Chapter 2 it looks at the operation, meaning and interworking of the various methods of expressing *Green* designs, illustrating how sets of simple components can be built up to produce complex systems, culminating in the design of a simple microprocessor which uses some techniques and systems that might also be used in an AMULET design. In Chapter 3 it looks at the tools for the development and simulation of *Green* designs, discussing the issues involved in the development of a graphical environment for micropipelined circuit design.

Chapter 2

Green

As described in the introduction, micropipeline designs can be represented as a collection of nodes which pass data between themselves using a standard protocol, with each step performing an operation on the data. This closely fits with a dataflow language model, and it is this view that the *Green* component of *Rainbow* is intended to capture.

The main differences between *Green* and other dataflow languages[CDJ84] are that *Green* is *static* (i.e with a fixed number of processes and channels in a fixed configuration) and that the communication channels are unbuffered, requiring any value written to them to be consumed by the reader of the channel before another value can be written to the channel. Instead, explicit finite buffering is introduced, this fitting far more closely the manner in which hardware works. The amount of buffering clearly affects the potential blocking and deadlock properties of a design in that a node may now block during output as the value it is attempting to produce cannot be consumed (as opposed to other dataflow languages where nodes may always produce output, but can be delayed indefinitely waiting for input).

Green comes in two varieties, allowing designs to be expressed in whatever style best suits the particular application. These two *Green* sublanguages are intended to be used together, as illustrated by the example in Sections 2.2.2.8 and 2.3.4.8.

Textual *Green* is a static dataflow language, designed to be largely declarative. Procedures written with it are really just sets of components connected together with some basic pipeline combinators.

Pictorial *Green* is a way of describing designs in a fashion that is very easy to convert into textual *Green* netlists. It performs a large amount of information hiding, helping to emphasise the large-scale structure of a design at the expense of concealing details of some of the more complex components (such as functions). Pictorial *Green* is intended to be mainly used within the environment of Chapter 3.

Most nodes have no *major* state (buffers being the obvious exception) allowing the state in a design to be easily identified. However, every device that can produce several outputs in the same time-cycle has *minor* state — which channels have accepted values from the device's outputs and which have not. This does not affect the long-term trace behaviour of a deterministic design[1]. By comparison, languages like CSP or CCS have to implement this behaviour using state as their processes do not have the capability to perform outputs on several channels at once.

Each use of a device is an instantiation of that device, and not a call to it. Where design sharing is desired, it must be specifically implemented by the designer, since *Green* does not provide any general way of telling that accesses will be distinct in time.

Pictorial *Green* is defined by translation into textual *Green* as a netlist. Each layout becomes a procedure, with the functions and channels in that layout being

[1]Nondeterministic designs can be regarded as having the state associated with the choice being located at the nondeterministic nodes within the layout.

declared first (channel names are generated automatically, after taking into account any hints derived from user-supplied labellings) and then all the instances of the devices are translated into a parallel composition of the pipelines in Table 2.1. While this is not the most efficient translation, it is easy to automate.

Hierarchic nodes are implemented as nested procedure declarations (as all names are local to the scope in which they are declared) and instances of a particular hierarchic node would look very much like instances of a function, except for the differences in the associated declaration.

2.1 Types in *Green*

The data types used in *Green* are those used in *Rainbow*: integers, booleans, tuples of data types, unions of data types, subranges of enumerative[2] types, user-defined enumerations, records of data types, and arrays of a data type.

Rainbow also contains procedures, functions and channels, though these are not first-class data types. Channels only carry first-class data types and can be thought of as being derived from them.

Note that *Rainbow* is polymorphic, and that the type of a channel (unless specified otherwise at its declaration) is taken to be the least general type that encloses the type of all the values that could be passed on that channel. This principle extends throughout the whole of *Rainbow*, except that wherever a declaration is made of a complex object (like a function or procedure, which are the only places where a type variable[3] can be bound) the signature in the declaration must be contained in the signature of the object itself, at least with respect to type variables. This means that any instantiation of the object need only examine

[2]Integers and user-defined enumerations are enumerative, in that they have a natural ordering. Booleans are considered to be enumerations.

[3]*Rainbow* has type variables as in ML[MTH90], but they are more restricted in where they can be introduced — procedure and function signatures are the only legal places, though they can also be used in the bodies of those procedures and functions.

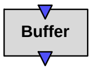
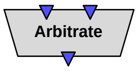
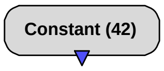
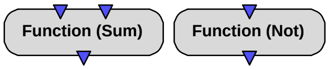
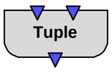
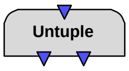
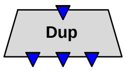
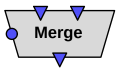
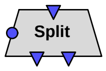
<table>
<tr><td>
</td><td>i >> buffer(1) >> o
or
i >> buffer(1,x) >> o</td></tr>
<tr><td>
</td><td>arbitrate i1 || i2 end_arbitrate >> o</td></tr>
<tr><td>
</td><td>source c >> o</td></tr>
<tr><td>
</td><td>i >> sink</td></tr>
<tr><td>
</td><td>function f(...) = ... end
⋮
{i1 || ... || im} >> f()
  >> {o1 || ... || on}</td></tr>
<tr><td>
</td><td>{i1 || ... || im} >> o</td></tr>
<tr><td>
</td><td>{i[0]>>o1 || ... || i[n-1]>>on}</td></tr>
<tr><td>
</td><td>{i>>o1 || ... || i>>on}</td></tr>
<tr><td>
</td><td>using c merge
    l1:i1 || ... || lm:im
end_merge >> o</td></tr>
<tr><td>
</td><td>i >> using c split
    l1:o1 || ... || ln:on
end_split</td></tr>
<tr><td>
</td><td>green hierDef(...) = ... end
⋮
{i1 || ... || im} >> hierDef()
  >> {o1 || ... || on}</td></tr>
</table>

Table 2.1: Basic Translations from Pictorial to Textual *Green*

the declared signature of the object to correctly type its use.

Type	Meaning
ν_α	Value of type α
σ_α	Unnamed stream of type α
χ_α	Named stream of type α
$\nu_{\alpha\times\beta}$	Tuple of values
$\sigma_\alpha \times \sigma_\beta$	Tuple of streams
$\nu_\alpha \rightarrow \nu_\beta$	Function of type $\alpha \rightarrow \beta$
$\sigma_\alpha \rightarrow \sigma_\beta$	Procedure with stream of type α as input and stream of type β as output
$\nu_\alpha \rightarrow (\sigma_\beta \rightarrow \sigma_\gamma)$	Parameterised procedure which takes an argument value of type ν_α to produce a procedure of type $\sigma_\beta \rightarrow \sigma_\gamma$
σ_ε	Stream which carries no values at all (the empty stream)
$\prod_{j=1}^{k+1} \alpha_j$	The product of all the types $\alpha_1 \ldots \alpha_{k+1}$ where $k \geq 0$ (i.e. $\alpha_1 \times \ldots \times \alpha_{k+1}$)
$\mathcal{I}$	Shorthand for integers
$\mathcal{B}$	Shorthand for booleans
$\sigma_{\alpha\cup\beta}$	The stream with the union of α and β for its type

Table 2.2: Type Notation Used in *Green* Definitions

When formally typing elements of *Green*, the notation in Table 2.2 is used. There are also several important features to note ($\Leftrightarrow$ denotes a commutative connectability[4] operator, and $\equiv$ denotes a type equivalence operator):

- A tuple of values is connectable with a value of tuples.

$$\nu_\alpha \times \nu_\beta \Leftrightarrow \nu_{\alpha\times\beta}$$

- A tuple of a stream of type α with the empty stream is connectable with a

[4]Types are connectable if instances of the one type can be connected to instances of the other. With values ν_α, this means that a stream σ_α of one type is the same as a stream of the other type. Connectable streams are the same across a piping operator ($\gg$)

stream of type α.

$$\sigma_\alpha \times \sigma_\varepsilon \Leftrightarrow \sigma_\alpha$$

$$\sigma_\alpha \Leftrightarrow \sigma_\varepsilon \times \sigma_\alpha$$

- Objects of type $\sigma_{(\alpha\rightarrow\beta)}$ are illegal, as are objects of type σ_{σ_α}. This, together with the equivalence:

  $$\nu_{\nu_\alpha} \equiv \nu_\alpha$$

  enforces the division of the type-system into first-class[5] and higher-order types.

- It is always possible to promote a stream of type σ_α to have the type $\sigma_{\alpha\cup\beta}$, though this promotion does not change the type of any of the values being passed on the stream itself (i.e., you can pass values ν_α on streams $\sigma_{\alpha\cup\beta}$ but you cannot pass values $\nu_{\alpha\cup\beta}$ on streams σ_α, and promoting the type of the input to a function does not promote the type of the function). When combined with the specialisation of type variables, this provides for a highly dynamic typing system where the only restrictions on the types that can be passed on a channel are usually those explicitly imposed by the designer.

2.2 Pictorial *Green*

The functionality of pictorial *Green* is defined by the nodes that are used in its designs, together with the channels connecting the nodes. The basic set of devices that may be placed in a design come in several varieties:

Stateful devices (Buffers). A buffer will accept a value on its input in one time-cycle, and then on a later time-cycle it will produce that value on its output and will not accept another input value until its stored value has been accepted by the device connected to the output channel.

[5]Only first-class types can be used for subscripts for stream-types.

Functional devices (Functions, Sources, Sinks, Duplicate, Tuple constructors and destructors). These all work by waiting until a value is available on each one of their inputs, and then produce a value on each of their outputs. Once those output values have been consumed, these devices indicate to all the input channels that the values on them have been consumed and become ready to accept the next values on the inputs.

Functional nodes can only perform combinatorial operations on their inputs, and anything more complex should be built using a network of other nodes to handle the storage of state and control the operation of the network. **Source**s and **Sink**s are special cases of functional devices which have no inputs or outputs respectively.

Channel manipulating devices (Splits and Merges). These (deterministic) devices are capable of reading from only a selection of their input channels or writing to only a selection of their output devices.

Nondeterministic devices (Arbitrators). These introduce nondeterminism by being able to choose between doing two (or more) actions. The choice is made internally if both actions are possible, but otherwise the choice is made according to which channel connected to the device is willing to supply a value. Once an action has been chosen, the device will not make another choice until that action has completed.

Special devices like memory banks, which are both stateful and nondeterministic, and tables, which allow the creation of devices with arbitrary, stateless, deterministic transition rules. These devices have behaviours that are not practical to model using the standard set of devices, and are described in more detail in the *Rainbow* reference manual.[BFGW]

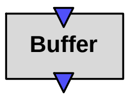
A restricted form of register bank is known to be implementable (see Figures 2.11 and 2.12) but the desired definition of blocking until data is put in, allowing simultaneous reads and writes, and performing the access across a number of cells is not known to have an implementation using devices from the basic set. It *is* known that tables are not generally implementable using the basic set of devices, but a useful subset of that behaviour can be implemented.

Other devices can be defined as subdesigns using **Hierarchy** nodes to encapsulate the contained behaviours and provide visual representations.

2.2.1 Description of Basic Devices

Each device communicates with the rest of the design containing it via the ports on its edge. These come in three types: inputs, outputs and control ports. Input ports are denoted by an inward-pointing triangle, output ports by an outward-pointing triangle, and control ports are denoted by a circle[6].

Buffer

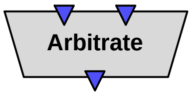


This reads a value from its input channel, and produces the value on its output channel — if the buffer is initialised, it behaves as if it had read a value immediately before the start of time. It may only perform one of the two operations at a time, and is the principal way of introducing state into a layout.

[6] There are both input and control ports to provide a visual distinction between those ports that are primarily for carrying data that is processed and controlled by the node, and those ports that connect to channels that control the behaviour of the node. There is no inherent semantic difference between the two kinds of input ports though — their visual difference is merely there as an aid to the understanding of controlled systems at a glance.

The type of the input channel is the same as the type of the output channel, and that type must contain the type of the initial value if there is one. Thus, if boolean values are only ever passed into the buffer, the output channel would be of type boolean. If that buffer was initialised to some integer, then the type of the input and output channels would actually be the union of the boolean and integer types, since this is the smallest type that encompasses both the type of the initialisation and the type of the input.

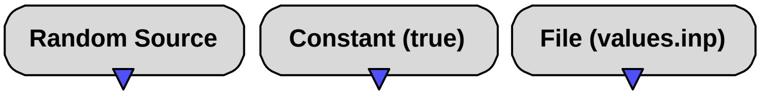
Arbitrate



This passes a value from one of its input channels to its output channel, with the choice between the channels (if both are available) being non-deterministic. The choice is made internally to the device, and once the choice has been made the communication that is associated with that choice will be performed, even if it leads the layout as a whole into deadlock.

The type of the output channel is the union of the types of the input channels.

Source



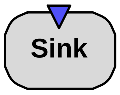
This injects values into a layout (these values may be specified directly as constants, generated randomly, or read from a file), with the type of the channel output from the source being the union of all the types of the data values that the source generates.

Constant sources always generate the same value, and are used to act as the heads of pipelines in the interior of a design, whereas file and random sources are usually used as heads of pipelines at the outermost level of the design where they represent the outside world; they are implemented by the environment, as opposed to being defined directly in textual *Green*.

File sources are used to inject external data into a design for simulation purposes, and produce the stream of values defined by the list of data values in the file named. If the file contains only a finite number of values, then the channel generated by the source node will also be finite.

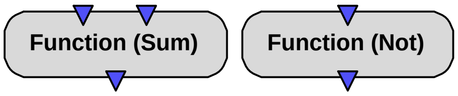
Random number sources[7] should solely be used as means of demonstrating a design, allowing the quick testing of a layout on data without the need for creating a file listing the values to place on the channel.

Sink



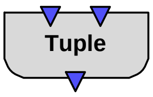
This absorbs values (of any type) from a layout. It is often used to represent the output of a network.

Function



This applies a (combinatorial) function to the value channel(s) entering it. If there are several channels entering it, it waits until a value is available on every channel before generating any output. If there are several output channels, it will not accept any more input until the output values have been consumed from all of its output channels.

Tuple

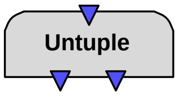


[7]The current implementation produces pseudo-random integers uniformly distributed over a fixed integer subrange.

This performs data tupling by taking the values presented on its input channels and combining them to form a single tupled output value. This device forces some synchronisation between the input channels, since it, like a **Function** node, requires all of its inputs to be available before producing an output, and because it generates the acknowledgement to all of its inputs at the same time (when it receives an acknowledgement on its outputs).

This device is an example of a specialised function node.

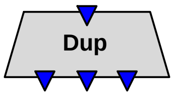
Untuple



This performs the reverse operation of **Tuple**, breaking apart a tuple into its components producing a channel for each of the elements of the tuple input. The type of the data coming in (in particular, the number of members of the tuple) should match the number of channels going out. All output values must be accepted before a new input value will be accepted.

This device is also an example of a specialised function node.

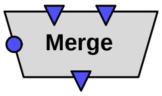
Duplicate



This passes a value from its input channel to all of its output channels (the number of which is a configurable parameter), waiting until all of the values have been successfully transmitted before accepting the next value. The types of the output channels are the same as the type of the input channel.

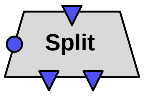
This can be regarded as a specialised function node, which does not perform any operation on the data flowing through it (apart from copying it).

Merge



This is a multiplexer, transmitting values from an input port chosen by the supplied control value (via the channel connected to the round port) to the output port. In keeping with the micropipeline style of communication, the control port undergoes full handshaking as well as the principal data communication channels. This causes synchronisation between the chosen input channel and the control channel.

Split



This is a demultiplexer, transmitting values from the input port to the output port chosen by the control channel. Like the **Merge** element, the control port undergoes full handshakes, causing synchronisation between the input channel and the control channel.

Hierarchy

This allows the creation of nodes with both user-defined behaviour and shape. The behaviour is defined by the embedded sub-design (which can be described using either pictorial *Green* or any other description form within *Rainbow*) and the external connections of this sub-design become the ports on the exterior of the hierarchic node. The physical appearance of the hierarchic node can be adjusted in the editor.

When producing a subdesign within a hierarchic node such as in Figure 2.1, the external interface of that design is produced by the introduction of external ports (which look like large versions of the ports on ordinary devices). Each of these corresponds directly to a port on the border of the hierarchic node containing the design.

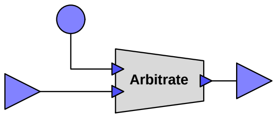


Figure 2.1: Simple Design with External Ports

Most devices are configurable in one form or another. **Sink** (since it always has exactly the same behaviour) and **Arbitrate** are the only non-configurable nodes, with the latter being due to the manner in which arbitration is implemented in hardware.

2.2.2 Pictorial *Green* Examples

These examples illustrate how to build a complex design using pictorial *Green*; they all have textual analogues in Section 2.3.4. The first three examples demonstrate basic layout construction. The next two examples demonstrate the creation and use of a complex design via hierarchic nodes. The final three examples build up to the definition of an entire microprocessor in *Green*.

2.2.2.1 A Pipeline

Figure 2.2 illustrates how a simple pipeline can be built. It applies a function (bitwise binary inversion) twice to a random stream of integers, with the output of the second function being dropped into a sink. Each of the value-manipulating stages of the pipeline is decoupled from the others by a **buffer**, which has the effect of only allowing one function to operate at a time. The operation of *Invert* is defined by:

$$\textbf{Invert} = \lambda x : \texttt{integer} \cdot x \text{ xor } -1$$

where xor has the usual bitwise definition, resulting in *Invert* performing a ones-complement operation on the bit-representation of the values input.

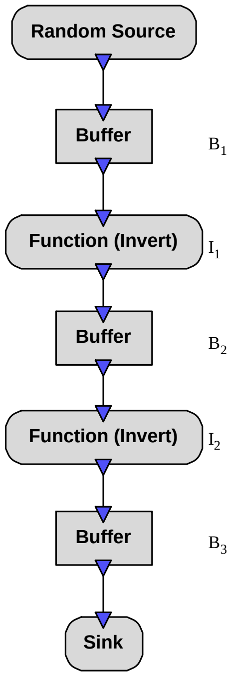


Figure 2.2: A Simple Pipeline

As an illustration of the operation of this circuit, consider the section between the first and the second buffers.

1. The first buffer (B_1) signals on its output that a value is available on the channel to the next element in the pipeline.

2. The inverter (I_1) delays this signal for long enough for it to put on its output data channel the inverted value and then signals to the next element in the pipeline that a value is available.

3. The second buffer (B_2), when it is empty and a value is available, latches that value into its internal state and signals back along the value that it

has been successfully received. At the same time, it signals forward that it has a value to pass on.

4. I_1 merely passes this acknowledgement straight through.

5. B_1, on receiving the acknowledgement, becomes ready to accept another value on its input.

See Section 3.4.1 for a more detailed example of the operation of a design.

2.2.2.2 A Substituter

This illustrates the operation of the control elements in pictorial *Green*. The layout in Figure 2.3 takes a stream of random values and substitutes the value 0 for every occurrence of the value 12 in that stream. It does this by testing whether the input value is equal to the value to replace, and feeding the output of this test into both a **split** (to allow the unwanted value to be discarded) and a **merge** (to allow the replacement value to be inserted into the output channel. No buffering is needed.

The layout as a whole works because the upper duplicate node does not release the value coming from the random source until both of its outputs have been acknowledged. When the value is equal to 12, this means both that the right-hand copy of the input value, and that the replacement value (0) has been successfully written from the constant. When the input value is not 12, the duplicate also acknowledges when the value output has been acknowledged (the critical path is marked with arrows, and is in both cases via the test and the controlling channel to the **Merge** node).

2.2.2.3 Cumulative Sum

This demonstrates the basics of building simple, non-deadlocking loops in *Green*. The layout in Figure 2.4 takes a channel of random values and generates the

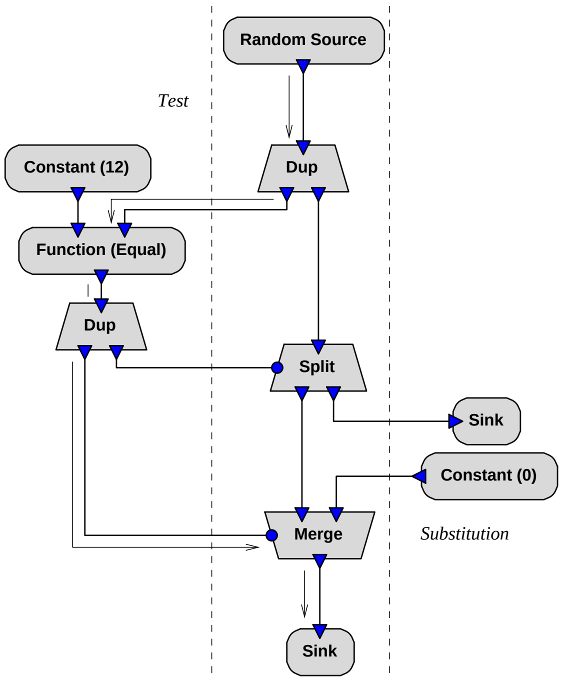


Figure 2.3: A Substitution on a Stream of Values

cumulative sum of those values by adding each new value to the sum of all the values preceding it. The main point to note in this example is the use of a loop with an initialised buffer in it. This must be initialised so that the sum **function** element can have both of its arguments available the first time it does an addition, and there must be another buffer in the loop as well, since buffers cannot be read from and written to at the same time[8]. This leads to a two-phase device where the first phase is when the calculation is being performed, and the second phase is when the current total is being passed back through the loop for the next input value.

In this design, the sink is used to represent the output from the layout, and the random source is used to represent the input to the layout.

[8]In general (for $n > 0$) if you have n values in a loop, then you must have at least $n + 1$ buffers or deadlock will occur. This is because you will need at least n buffers in the loop to act as value-sources during any time-step, together with at least one other buffer capable of acting as a value-sink.

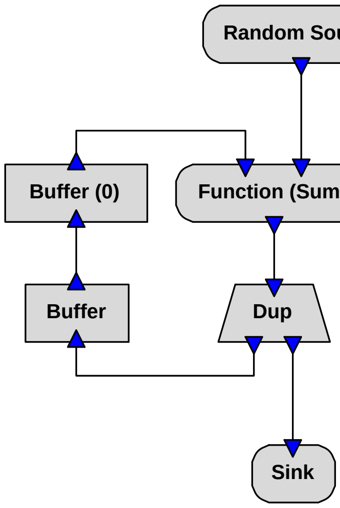


Figure 2.4: The Cumulative Sum of a Stream of Values

2.2.2.4 While Loop

This example demonstrates how a fairly basic design may be elaborated into a more complex version with improved functionality, as well as showing how an interface to a layout can be produced using external ports.

The purpose of a while loop is to perform the operation in the body *while* the test is true, but this really needs to be looked at in a functional manner (with the operations being performed on a current value) for a suitable pipelined meaning to be obtained at all.

First, we define the desired behaviour of a while loop.

1. It should take in a value from the input channel.

2. It should perform a loop test on this, and output the value if the test returns false.

3. Otherwise, it should pass the value through the body of the loop and pass the returned value back to the beginning. Like this, the value is duplicated for tests as many times as necessary.

4. If the body and the test portions of the loop are functional, then the loop as a whole should behave externally like a (single input) function. This provides an easy to understand interface.

This leads to a layout like Figure 2.5 (provided requirement 4 is ignored). The main features of this design are the use of external ports (to allow the while to communicate with the outside world) and arbitrate (to connect the input and the feedback into a single channel to the control code.

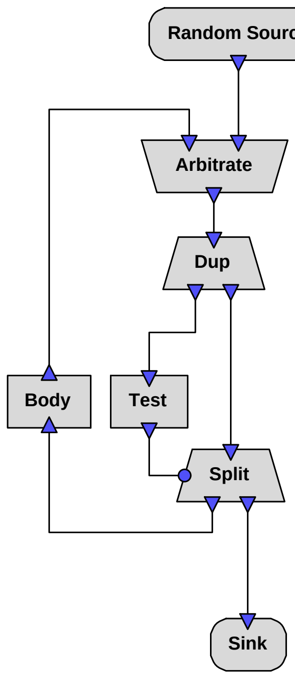


Figure 2.5: Basic While Loop

However, it suffers from two main shortcomings:

1. New values may be injected into the loop while old values are still being processed. After a decision to loop has been made, the **Arbitrate** should always choose from the loop-back side. Otherwise it should choose to take input from the external world.

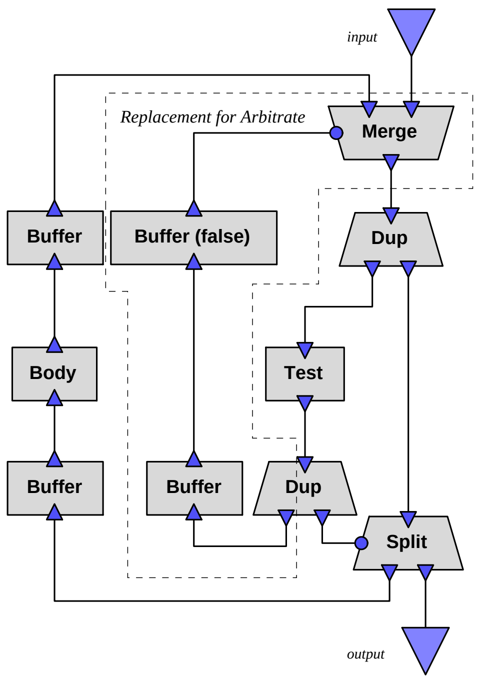


Figure 2.6: Improved While Loop

This can be avoided by using a **Merge** element controlled by a delayed copy of the output of the test (with the delay queue initialised so that the **Merge** lets through a starting value from outside the loop) to replace the **Arbitrate**.

2. If the user puts a **Function** node in as the body of the loop, the system will deadlock due to the lack of any buffering within the loop[9]. This can be corrected by moving the responsibility for buffering into the while loop itself.

Figure 2.6 is the result of the application of the improvements outlined above.

The new design still does not behave like a function, since it may release its (principal) external input before it has completed writing its (principal) external output. It also does not export its body and test blocks to the outside world, limiting the reusability of this particular layout.

[9]This forces the arbitrator (or the merge that replaces it) to be accepting values from both outside the loop and the loop itself at the same time, which is not allowed.

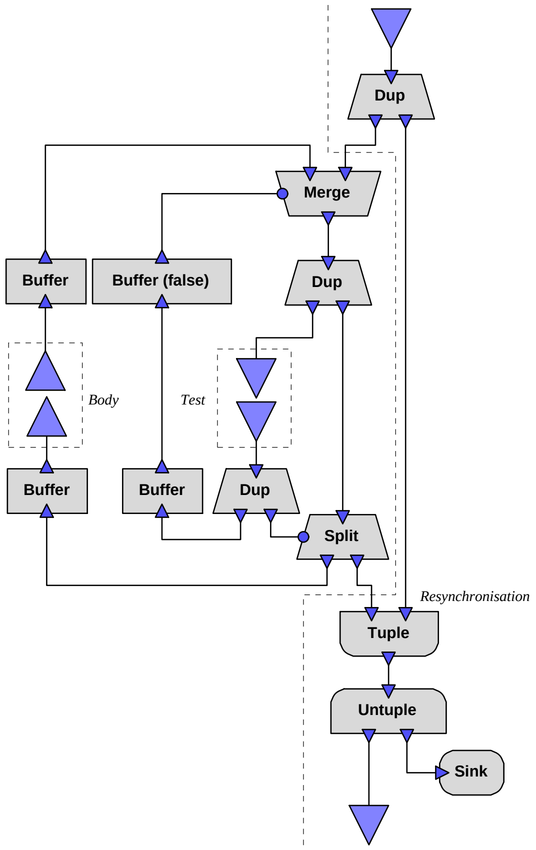


Figure 2.7: Full While Loop

These final minor problems can be cured by synchronising the external input with the external output by tupling a copy of the input to the while and the output from the while together, and then undoing this tupling and throwing that copy of the input away. Replacing the test and the body by external I/O port pairs leads to a layout as in Figure 2.7.

2.2.2.5 Fibonacci Calculator

This design demonstrates the use of a hierarchically defined node, and is based upon the while loop defined in Figure 2.7, but adding the additional processing elements to create a full Fibonacci number calculation algorithm. This is shown in Figure 2.8.

The additional processing is performed in the functions labelled “Init”, “Test”,

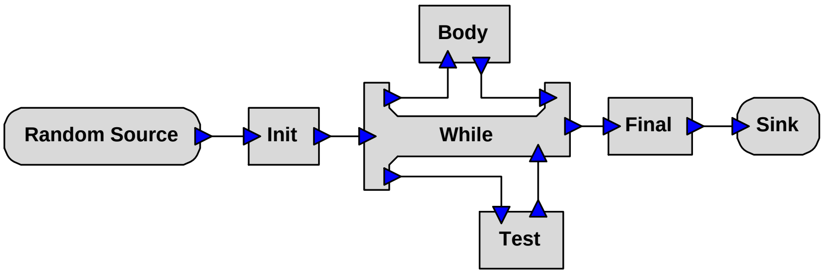


Figure 2.8: Fibonacci Number Generator

"Body" and "Final" which apply the following operations[10]:

$$\mathbf{Init} = \lambda x \cdot (x, 1, 1, 1)$$

$$\mathbf{Test} = \lambda (x, i, o, n) \cdot x > i$$

$$\mathbf{Body} = \lambda (x, i, o, n) \cdot (x, i + 1, n, o + n)$$

$$\mathbf{Final} = \lambda (x, i, o, n) \cdot o$$

2.2.2.6 Memory Element

In this example, the purpose is to create a storage element that can have a value read from it many (an undetermined number) times between each write.

In order to accomplish this, again it is convenient to start off by considering the desired properties of the system:

1. The act of dispatching a new value into the input channel should cause the value stored to be changed to the value supplied. This is what is expected of a memory element.

2. The system should only supply a value (on its output channel) when requested. This makes it fit in with the general pushing scheme of micropipelines.

3. The element should internally handle determining whether a read or a write

[10]These operations are implemented in hierarchic nodes due to the limitations of the current set of functions. This will change with the integration of the full *Rainbow* function evaluator package into the environment of Chapter 3.

was requested, and perform the appropriate action accordingly. In practice, this means either a dedicated control line saying whether to read or write, or an internal arbitrate unit, though ideally the element will allow simultaneous reads and writes.

4. The element should start out empty, and should not permit any reads to succeed until a write has succeeded.

This particular implementation uses (on the input side) a read channel (of an arbitrary type, since the values will not matter) and a write channel of the same type as the memory cell. The output channel will be of the same type as the memory cell.

The core of the memory cell is defined in Figure 2.9. This has the ability to output the stored value many times and to replace the stored value with one read from the outside world. It however requires a separate control channel, as well as having a buffer that is already initialised.

Initialisation is required, since otherwise any writes would result in the written value being thrown away and the first read of the memory cell would always block, no matter how many times the cell was written to first.

The requirement that the read control should come from one channel, and the write control and data from another is handled by extending the core memory cell with devices that generate the correct read/write controls from the untyped read-request stream and the write-stream of data values, as shown in Figure 2.10. It is at this level that non-determinism between reads and writes is introduced via the arbitrate device.

However, this design still fails to deal with the requirement that no read should happen until at least one write has occurred. This can be satisfied by diverting the first value written to the memory element directly into the storage in the core of the element, bypassing both the normal mechanism for emptying

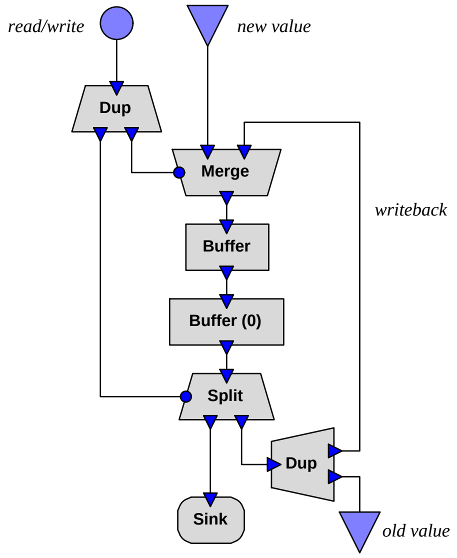


Figure 2.9: Initialised, Deterministic Memory Cell

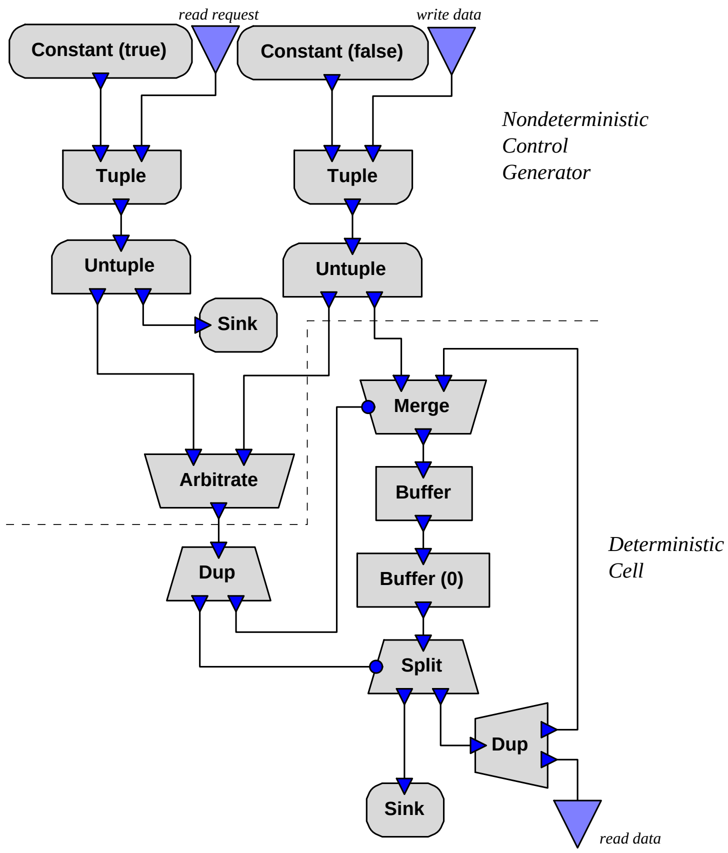


Figure 2.10: Basic Memory Cell

the register of its old value at the same time, and the arbitration control (this being necessary, as a read request might have already locked the arbitrator when the write arrives). At the same time, the register is made uninitialised, so that the value used to initialise the register has a space to go into. This leads to the full design as in Figure 2.11.

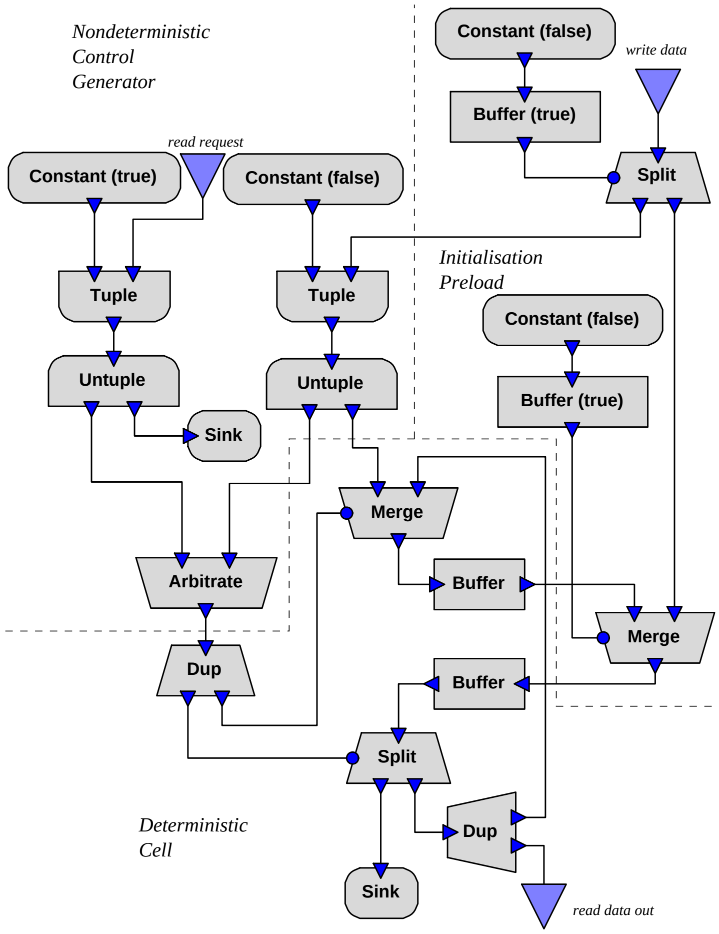


Figure 2.11: Full, Initialising Memory Cell

In this version, the first value on the write channel will be diverted directly into the (initially empty) store in the memory cell without triggering the output **Split** stage, and all values passed on the write channel after that are arbitrated with the requests on the read channel for data.

The constant/buffer combinations generate a series of values where the first value differs from the rest. Two copies of the sequences are needed since the number of values that need to be generated[11] is greater in the body of the loop than at the write input.

2.2.2.7 Small Register Bank

Often, what is actually wanted when building a microprocessor such as the AMULET[FDG+93] is an initialised (usually to zero) register bank, where the actions being performed by the bank can be determined precisely through its control ports, removing the need for arbitration.

This leads to the design illustrated in Figure 2.12 with four cells, though the principle extends easily to any number. Each cell is a unit as in Figure 2.9, embedded within the level describing the bank management circuit. The main features of this design are the use of four-way **Split** and **Merge**, and the way that there are separate read/write control and address control lines. This means that the register bank is a deterministic unit, and that there is no possibility of performing a read and a write at the same time.

This can be extended in the obvious manner to banks of as many registers as desired.

2.2.2.8 Simple Microprocessor

This last example is intended to how larger systems can be built in *Green*. This example is actually best suited to the greater expressive power of textual *Green*, even though the operation of the system is less clear there. This is because the design uses tables which have no visual analogue.

The design in Figure 2.13 describes a simple microprocessor, consisting of three principal operating units:

[11] This is only true for any finite sequence of read and write requests, where the number of read requests is non-zero.

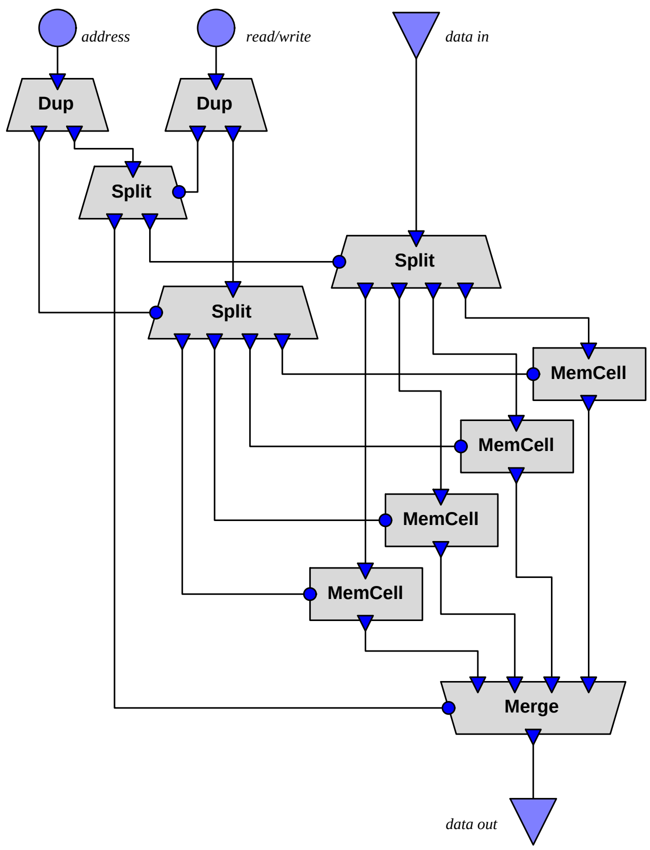


Figure 2.12: Small (4-place) Register Bank

1. The instruction decoder (layout in Figure 2.14). This unit controls the rest of the processor, causing instructions to be fetched from memory to itself, and then translating those instructions into the sequence of actions to be performed by the rest of the processor in order for the operation of the instruction fetched to be carried out.

   It is implemented using a table (part of textual *Green* described in Section 2.3) and a couple of buffers to form a finite state machine. The description of the table is given in Section 2.3.4.8

2. The register bank (layout in Figure 2.15). This is implemented on top of a (trivially) expanded version of the register bank in Figure 2.12 and merely adds the capability to take input values from several channels and produce output values on several channels (though only one action can be done at a time).

3. The ALU. This is implemented wholly by a *Green* table and is described in Section 2.3.4.8

   The ALU is stateless, performing combinatorial functions on its inputs to produce its outputs.

The processor operates by first causing the current PC value to be sent to memory as a read request while loading the incremented address value back into the PC at the same time. The value returned from memory is passed into the instruction decoder where it is then executed. The execution phase always takes three cycles (since an instruction may access the register bank up to three times, twice to read and once to write) which produces a five stage system with the bottleneck being due to the single control channel access to the register bank.[12] Currently, this design has not actually been simulated, since the simulator doesn't

[12]Though the decode subsystem is the bottleneck in this design, this is caused mainly by the requirement that accesses of the register bank be sequential.

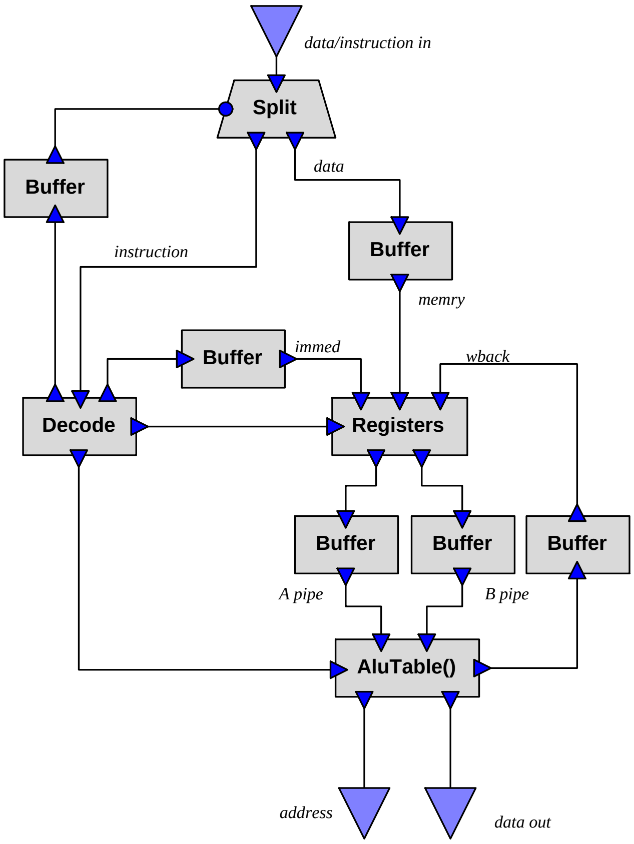


Figure 2.13: Toplevel Layout of a Simple Microprocessor

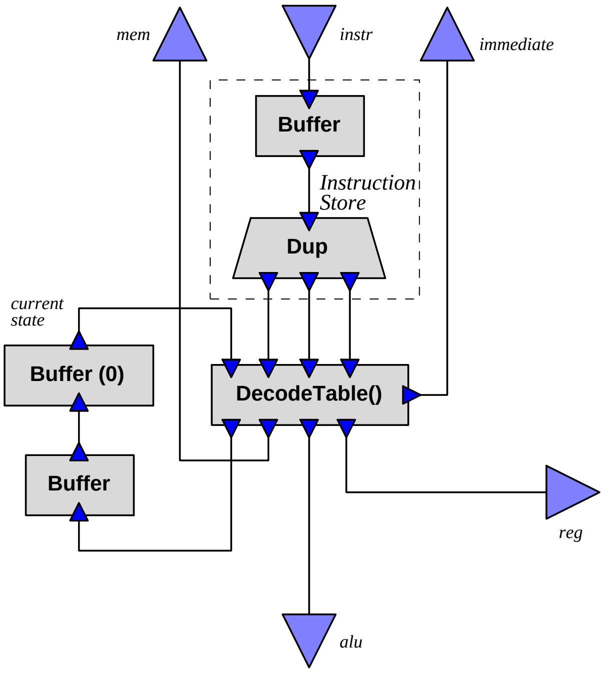


Figure 2.14: Instruction Fetch and Decode Subsystem of Microprocessor

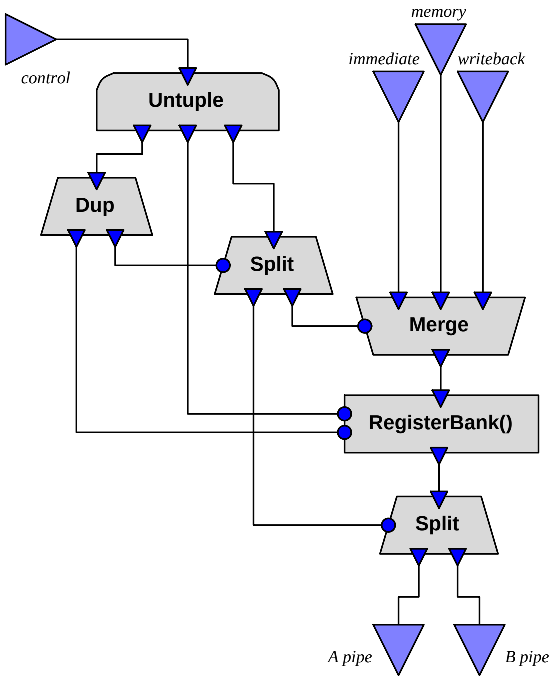


Figure 2.15: Register Subsystem of Microprocessor

currently have the ability to handle tables (which form an important part of the design), and verification lies even further off in the future.

The nesting of hierarchies involved in the microprocessor can be best illustrated here:

Microprocessor	
Decode	
DecodeTable()	
AluTable()	
Registers	
RegisterBank	
MemoryCell^n	n is the number of registers
Memory	This component is not described here. ...

For more information on the operations performed by the microprocessor and its instruction set, see Appendix B.

2.3 Textual *Green*

Textual *Green* is a way of expressing the dataflow micropipeline style of design using text. It provides the underlying semantics of pictorial *Green* and allows the description of much more complex objects such as user-defined functions and tables at the expense of making the overall layout of the design harder to understand intuitively. See Appendix A for the syntax of the *Rainbow* language, of which textual *Green* is a part.

There are four main parts of textual *Green*:

Types and expressions which are common to all parts of *Rainbow*.

Declarations which allow the creation of named elements like *Green* processes and tables.

Pipeline combinators for composing commands into pipelines.

Commands which form the basic elements of pipelines.

In pictorial *Green*, every channel is point-to-point. In textual *Green*, multiple readers are permitted on a channel as if there are implicit **Duplicate** nodes on every channel. Each value that is placed on that channel must be read by all the readers before another value can be communicated via the channel. *Rainbow* channels have no inherent restriction upon the number of times they can be read, but the sub-languages of *Rainbow* all place some restrictions on how channels may be accessed.

2.3.1 *Green* Declarations

Declarations introduce a new name with the meaning of the body of the declaration at the end of the declaration, so a name cannot be used in its own declaration.

This prevents all structural and functional recursion[13] by construction.

Aside from the the declarations associated with *Rainbow* as a whole (functions, constants, channels and types) there are three declarations that are specific to textual *Green*:

***Green* procedure** declarations are denoted by the keyword `green` and consist of a procedure signature followed by the body of the procedure, which is effectively a piece of local scope, the same as in the definition of $\{\ldots\}$ below. In a procedure definition there is one significant difference, since all the channels connected to the procedure must be referred to by their name in the procedure body. All top-level pipeline output from the body of the procedure is implicitly directed into **sink**s and every top-level pipeline has an empty stream (σ_ε) as its input.

A signature consists of a list of names, each associated with a class of object and a type. The classes of object that can be given a name are **input** and **output** channels, **value**s, **proc**edures and **function**s. Value parameters are used to pass constants to procedures and inputs to functions. Procedure and function parameters are there for building library components like a generalised sorting procedure which takes a comparison function as one of its parameters (the types of functions and procedures are considered to be their signatures).

Memory declarations create a memory bank with two input channels and an output channel. If the declaration of the memory bank is as:

```
memory M : integer^32
```

then it is as if there was a green procedure declaration with the following signature:

[13] Recursion does not make sense in the context of static hardware.

```
green M(input  readAddress:[0..31];
        input  writeAddressData:([0..31],integer);
  output readData:integer)
```

To read a memory cell in the bank, the address of the cell to read should be sent to the `readAddress` port, causing the value in the given cell to appear on the `readData` port.

To write a memory cell, the tuple of the address of the cell to write and the value to put there should be sent to the `writeAddressData` port.

Reads and writes can occur simultaneously, with the old contents of the cell being the value read if there is some conflict on a particular memory cell.

Table declarations allow the definition of a deterministic, *stateless* block where the actions that it can perform are listed in a tabular format, with the special option, indicated by a `-` symbol, of performing no action at all on any particular channel (either input or output). Once an action has been chosen (through exact matching of all the input patterns described in that rule) then all the outputs for that rule are generated, and the table only becomes ready to accept further inputs once all those outputs have been accepted and all the inputs that were used to trigger the rule acknowledged.

Tables can be regarded as special cases of *Green* procedures, and they share the same scheme for signatures.

An example of a table would be:

```
table Tab(input i,j:boolean; output k:boolean) =
        true,  -      =>  false
    or  false, true   =>  true
    or  false, false  =>  -
end
```

This creates a procedure *Tab* with two input channels i and j and an output channel k. The actions that an instance of the procedure can perform are:

1. Read a *true* value from i and produce a *false* value on k. No action on j would be performed.
2. Read a *false* value from i, a *true* value from j and produce a *true* value on k.
3. Read a *false* value from both i and j and perform no action on the output channel.

When a procedure (or function) is used, each of the parameters in its signature must be instantiated to an object with correct type. These arguments can be supplied using three mechanisms:

Positional: the style of argument passing that is most widely used in other programming languages (like C, Pascal, FORTRAN, etc.)

Named: allows the user of the procedure or function to explicitly associate each actual parameter with the name of a formal parameter. This may be particularly useful when the number of arguments is large, to avoid confusion between arguments. Named and positional arguments may not be used in the same instantiation.

Implicit: can only be used to supply input and output channels[14], but can be used with both positional and named arguments. These are the only way of connecting an unnamed channel to a procedure or function.

When using positional arguments with implicit arguments for inputs, all the input channels must be supplied implicitly (and the positions which they occupied

[14] Input channels may be connected to value arguments in functions only. With a procedure instantiation, only input arguments can have input channels attached.

should be completely omitted for the purposes of writing the use), and similarly with outputs. This means that where a procedure P has the (untyped) signature:

green P(input a,b; output c; value v:integer)

then the following are legal (assuming the types match) but other combinations of positional and implicit arguments are not:

$$\{x \| y\} \gg P(42) \gg z$$
$$\{x \| y\} \gg P(z, 42)$$
$$P(x, y, 42) \gg z$$
$$P(x, y, z, 42)$$

This restriction is not placed on combinations of named and implicit arguments, where it is always clear which arguments are being passed implicitly and which explicitly. This means that the following instance of P is also legal:

$$y \gg P(v = 42, a = x) \gg z$$

2.3.2 *Green* Combinators

There are three pipeline combinators (in order of precedence, from tightly binding to loosely binding):

$\gg$ is the pipeline connector; an infix operator which makes the output of the preceding command connect to the input of the following command, producing a new command.

It does not introduce any synchronisation between the channels except in the case where there is a single channel on one side, and multiple channels on the other side. In this case, the type of the single channel will be the tuple of the types of the multiple channels on the other side of the operator. When the multiple channels are on the input side, synchronisation between the channels is forced and none of the input channels will be acknowledged until the output channel has been acknowledged. When the multiple channels are on the output side, all the requests on the output channels are generated when a request is received on the input channel, and an acknowledgement on the input channel is generated only when all the acknowledgements from the output channels have been received.

It has the special syntax of also being permitted to start and terminate any pipeline (except where a pipeline terminator is used), though it has no meaning in these cases.

It flattens the tuples of streams that pass through itself, allowing:

$$(\sigma_\alpha \times \sigma_\beta) \times \sigma_\gamma \Leftrightarrow \sigma_\alpha \times \sigma_\beta \times \sigma_\gamma \Leftrightarrow \sigma_\alpha \times (\sigma_\beta \times \sigma_\gamma)$$

In general, the pipeline connector has (after applying the flattening rule) the type scheme:

$$[(s \to t) \times (t \to u)] \to (s \to u)$$

where

$$s = \prod_{j=1}^{k+1} \sigma_{\alpha_j} \quad t = \prod_{j=1}^{m+1} \sigma_{\beta_j} \quad u = \prod_{j=1}^{n+1} \sigma_{\gamma_j}$$

$ is the pipeline terminator (a postfix operator), which indicates that the output from this pipeline is discarded. It is shorthand for:

$$\gg \textbf{sink}$$

It can also be used as an infix operator to separate two pipelines where the output from the first is not wanted, and where the second does not take an input. This is shorthand for the form:

$$\gg \textbf{sink} \gg$$

The pipeline terminator has the type schemes:

Unary: $(s \rightarrow t) \rightarrow (s \rightarrow \sigma_\varepsilon)$

Binary: $[(s \rightarrow t) \times (\sigma_\varepsilon \rightarrow u)] \rightarrow (s \rightarrow u)$

where

$$s = \prod_{j=1}^{k+1} \sigma_{\alpha_j} \quad t = \prod_{j=1}^{m+1} \sigma_{\beta_j} \quad u = \prod_{j=1}^{n+1} \sigma_{\gamma_j}$$

$||$ is the parallel composition operator, with the input streams to the combination as a whole being the parallel composition of the input streams of both sides of the operator (and the analogue being true for the output channels). No synchronisation is implied here.

Where a channel of empty type (σ_ε) is placed in parallel composition with any other channel or group of channels, it is removed from the parallel composition. Thus, in terms of types,

$$\sigma_\alpha \times \sigma_\beta \times \sigma_\varepsilon \times \sigma_\gamma \Leftrightarrow \sigma_\alpha \times \sigma_\beta \times \sigma_\gamma$$

The parallel composition operator has the type scheme:

$$[(s_L \rightarrow t_L) \times (s_R \rightarrow t_R)] \rightarrow (s \rightarrow t)$$

where

$$s_L = \prod_{j=1}^{m+1} \sigma_{\alpha_j} \quad s_R = \prod_{j=m+2}^{m+n+2} \sigma_{\alpha_j} \quad s = \prod_{j=1}^{m+n+2} \sigma_{\alpha_j}$$
$$t_L = \prod_{j=1}^{p+1} \sigma_{\beta_j} \quad t_R = \prod_{j=p+2}^{p+q+2} \sigma_{\beta_j} \quad t = \prod_{j=1}^{p+q+2} \sigma_{\beta_j}$$

2.3.3 *Green* Components

There are thirteen basic *Green* commands:

- **<u>sink</u>** is a node that swallows any input presented to it (this is very similar to the pipeline terminator, except in that it is a command, and not a combinator). It has type:

$$\sigma_\alpha \rightarrow \sigma_\varepsilon$$

- **buffer**(*length,init*) is a buffer (or FIFO queue) of the given *length*, optionally initialised with *init* (the initial argument is omitted for an empty buffer). Where an initial value is supplied, it is only used for the last element in the queue (so it is available immediately) and the rest of the queue is empty. Buffers have type:

$$(\nu_{\mathfrak{I}} \times \nu_{\alpha}) \rightarrow (\sigma_{\alpha} \rightarrow \sigma_{\alpha})$$

  where an initial value is supplied, and otherwise:

$$\nu_{\mathfrak{I}} \rightarrow (\sigma_{\alpha} \rightarrow \sigma_{\alpha})$$

- **arbitrate** *parallel_pipelines* **end_arbitrate** nondeterministically joins the output of the two or more *parallel_pipelines* into a single channel in a manner analogous to the **Arbitrate** node in pictorial *Green*. It can be regarded as having type:

$$\left(\prod_{j=1}^{k}(\sigma_{i_j} \rightarrow \sigma_o)\right) \rightarrow \left[\left(\prod_{j=1}^{k}\sigma_{i_j}\right) \rightarrow \sigma_o\right]$$

- {*declarations parallel_pipelines*}. This indicates a portion of local scope with a list of *declarations* followed by a parallel composition of *pipelines*, and has exactly the same type as the composition of pipelines in its body.

- **source** *expression* creates a constant value **source**, always ready to supply values to the channel on its output. It has type:

$$\nu_{\alpha} \rightarrow (\sigma_{\varepsilon} \rightarrow \sigma_{\alpha})$$

- *chan_identifier* indicates a named channel access. If there is a non-empty channel input to this command, then it is a write to the named channel. The value in the named channel is always the output channel of this command[15].

[15] The exception to this rule is when the output channel of this command goes into a pipeline terminator, when the command behaves like a pure channel write.

Pipeline access commands have one of the following two types, depending on whether it is used in a channel-reading or a channel-writing context:

$$\textbf{Writing:} \quad \chi_\alpha \to (\sigma_\alpha \to \sigma_\alpha)$$
$$\textbf{Reading:} \quad \chi_\alpha \to (\sigma_\varepsilon \to \sigma_\alpha)$$

- *proc_identifier* (*arguments*) indicates a procedure or function instantiation (actually an instantiation of the procedure which applies the function to each value on the inputs to generate each value on the outputs).

  A function instantiation has the following type after allowing for explicitly supplied arguments (procedure instantiations are similar):

  $$(\nu_\alpha \to \nu_\beta) \to (\sigma_\alpha \to \sigma_\beta)$$

  where

  $$\alpha = \prod_{j=1}^{k} \alpha_j \quad \text{and} \quad \beta = \prod_{j=1}^{m+1} \beta_j$$

- **using** *pipeline* **split** *labelled_pipelines* **end_split**

- **using** *pipeline* **merge** *labelled_pipelines* **end_merge**

  These commands are analogous to **Split** and **Merge** pictorial nodes respectively. The output of the *pipeline* in the **using** clause corresponds to the control input on the pictorial nodes, and the body (consisting of pipelines labelled with the value of the control associated with connecting the input (or output) with the main input (output) port on the pictorial node) corresponds to the pipelines attached to the data input (output) sides of the pictorial nodes.

  The types of **split** and **merge** are as follows, where *ctlty* is the control input, *lblty* is the labelling of the pipelines in the bodies of the commands, and *sbdyty* and *mbdyty* are the pipelined bodies of the commands:

split: $(ctlty \times lblty \times sbdyty) \rightarrow splty$

merge: $(ctlty \times lblty \times mbdyty) \rightarrow mrgty$

where

$$\begin{array}{ll} ctlty = \sigma_{\varepsilon} \rightarrow \sigma_{\alpha} & lblty = \nu_{\alpha} \rightarrow \nu_{\mathcal{J}} \\ sbdyty = \prod_{j=1}^{k+2} (\sigma_{in} \rightarrow \sigma_{out_j}) & mbdyty = \prod_{j=1}^{k+2} (\sigma_{in_j} \rightarrow \sigma_{out}) \\ splty = \sigma_{in} \rightarrow \left(\prod_{j=1}^{k+2} \sigma_{out_j} \right) & mrgty = \left(\prod_{j=1}^{k+2} \sigma_{i_j} \right) \rightarrow \sigma_{out} \end{array}$$

- **if** *chooser* **then** *pipeline* **else** *pipeline* **end_if** is like a combination of **split** and **merge**, where the *chooser* type is forced to be a boolean and the number of arms is consequently two. An **if** may be implemented by the fragment of pictorial *Green* in Figure 2.16, and it works by directing values along two pipelines according to the values supplied on the chooser stream. When the *chooser* pipeline supplies a true value, the **if** command will provide the input value to the true (**then**) branch and accept a value from the true branch before acknowledging and accepting another value from the *chooser* pipeline. When the *chooser* value is false, the **if** command behaves the same, except using the false (**else**) branches each time. There is no implicit buffering anywhere in an if command.

  If commands have the following type scheme:

$$[(\sigma_{\varepsilon} \rightarrow \sigma_{\mathcal{B}}) \times (\sigma_{in} \rightarrow \sigma_{out}) \times (\sigma_{in} \rightarrow \sigma_{out})] \rightarrow (\sigma_{in} \rightarrow \sigma_{out})$$

  The first argument is the chooser, the second is the true pipeline, and the third is the false pipeline.

- **case** *chooser* **of** *labelled_pipelines* **end_case** is a generalised version of **if** which allows more than two cases and different types of *chooser* value

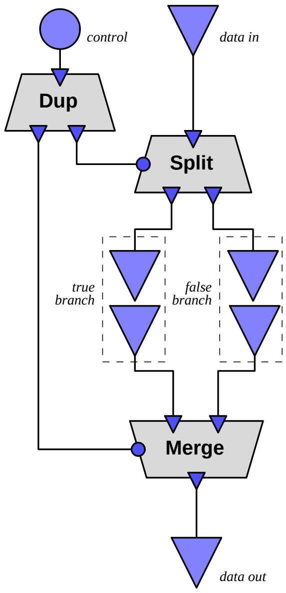


Figure 2.16: Implementation of an If Command

through a labelling as in the **merge** and **split** commands. It has the type:

$$\left[(\sigma_\varepsilon \to \sigma_\alpha) \times (\nu_\alpha \to \nu_{\mathbb{J}}) \times \left(\prod_{j=1}^{k+1} (\sigma_{in} \to \sigma_{out})\right)\right] \to (\sigma_{in} \to \sigma_{out})$$

- **while** *condition_pipeline* **do** *body_pipeline* **end_while** applies the *body* of the **while** to each input value repeatedly until the *condition* does not hold of the value, when it produces that resulting value as the output. One possible implementation of a **while** command is given in Figure 2.7.

  The **while** command has the type scheme:

  $$[(\sigma_\alpha \to \sigma_{\mathbb{B}}) \times (\sigma_\alpha \to \sigma_\alpha)] \to (\sigma_\alpha \to \sigma_\alpha)$$

- **for** *constant* **do** *pipeline* **end_for** applies the pipeline to each value in the input channel a fixed number of times (as specified by the constant argument). The input remains locked until the final value is accepted from the output. For example, if f is a function and n is a constant integer, then:

  for n **do** $f()$ **end_for**

is functionally equivalent to the operation of g when $g = f^n$.

The **for** command can be implemented in terms of the **while** command above (the iterator is not available to the body) but **for** is not specified like that in order to encourage better optimisation of its uses.

The **for** command has the type scheme:

$$[\nu_{\mathfrak{I}} \times (\sigma_\alpha \to \sigma_\alpha)] \to (\sigma_\alpha \to \sigma_\alpha)$$

2.3.4 Examples

These examples all implement exactly the layouts seen in the figures in Section 2.2.2. Note that random sources are represented by the procedure `Random()`.

2.3.4.1 A Pipeline

```
green Pipeline() =
    Random() >> buffer(1) >>
    Invert() >> buffer(1) >>
    Invert() >> buffer(1) >> sink
end
```

2.3.4.2 A Substituter

```
green Substituter() =
    chan thevalue,istwelve end
    function test(value x:integer) : boolean =
        x==12
    end

    Random() >> thevalue >> test() >> istwelve$
```

```
        thevalue >> if istwelve then
            sink$ source 0 >>
        else
            {}
        end_if >> sink
    end
```

It is easy to extend this scheme to perform different substitutions on the same stream of values using a case command:

```
    green Subst() =
        chan thevalue end
        function choprange(value x:integer) : integer =
            x mod 4
        end

        Random() >> thevalue >>
        case thevalue >> choprange() >>
        of  0: {}             // Do nothing for multiples of 4
         || 1: {}             // ... or if we are 1 larger
         || 2: sink$ source 2 // ... but map these to 2
         || 3: sink$ source 7 // ... and these to 7
        end_case >> sink
    end
```

Note that the use of the $ terminator above is not strictly necessary, as a $\gg$ operator would have been sufficient there. The use of the terminator is still recommended as being good style.

2.3.4.3 Cumulative Sum

```
green CumulSum() =
    chan feedback end

    { Random() >>
    || feedback >> buffer(2,0) >>
    } >> Sum() >> feedback >> sink
    // Could have omitted sink from line above
end
```

2.3.4.4 While Loop

The final version of the while loop from Figure 2.7, in addition to the special syntax it has in textual *Green*, can be written as:

```
green While(input  i, testresult, bodyresult;
            output o, testdata, bodydata) =
    chan loopout, synch end

    {loopout || i} >> synch $
    synch.0 >> o $

    using testresult >> buffer(2,false) merge
        true:  bodyresult >> buffer(1) >>
     || false: i >>
    end_merge >> testdata
      >> using testresult split
        true:  >> buffer(1) >> bodydata$
     || false: >> loopout$
```

```
    end_split
end
```

Note the loss of the sink from the tuple extraction since textual *Green* can read from only part of a channel, unlike pictorial *Green* which is restricted (without the use of complex functions) to handling all the data from a tupled channel.

2.3.4.5 Fibonacci Calculator

While could be implemented using code similar to that in Section 2.3.4.4.

```
green Fibonacci() =
    function Init(value x:integer) =
        (x, 1, 1, 1)
    end
    function Test(value x) =
        x.0 > x.1
    end
    function Body(value x) =
        (x.0, x.1+1, x.3, x.2+x.3)
    end
    function Final(value x) =
        x.2
    end

    Random() >> Init() >>
    while Test() do
        Body()
    end_while >> Final() >> sink
end
```

2.3.4.6 Memory Element

The core memory cell described in Figure 2.9 can be built in textual *Green* using the following:

```
green MemCell(input  writeData, readWrite;
              output readData) =
    chan wbchan end

    using readWrite merge
        true:  wbchan >>
     || false: writeData >>
    end >> buffer(1) >> buffer(1,0) >>
    using readWrite split
        true:  >>wbchan
     || false: {}
    end$

    wbchan >> readData
end
```

The full version (with the initialisation from the write data channel) from Figure 2.11 can be described in textual *Green* by:

```
green MemCell_InitArb(input  readRequest, writeData;
                      output readData) =
    green tfff(output o) =
        source false >> buffer(1,true) >> o
    end
    green tuun(input  iv;
```

```
          output ob, ov;
          value  ib) =
    chan c end
    {source ib || iv} >> c$
    c >> {ob || ov}
end
chan
    wdin1, wdin2, wdata, access, midpoint
end

writeData >> using tfff() split
    true:  >> wdin2
 || false: >> wdin1
end_split$

arbitrate
    readRequest >> tuun(true) >> {{} || sink}
 || wdin1 >> tuun(false) >> {{} || wdata}
end_arbitrate >> access$

using access merge
    true:  readData >>
 || false: wdata >>
end_merge >> buffer(1) >> midpoint$

using tfff() merge
    true:  wdin2 >>
```

```
            || false: midpoint >>
           end_merge >> buffer(1) >> using access split
               true:  >> readData
            || false: sink
           end_split
       end
```

2.3.4.7 Small Register Bank

The register bank of Figure 2.12 is described in textual *Green* by:

```
       green RegBank(input  writeData, address, readWrite;
                     output readData) =
           chan readAddress, writeAddress,
                c0, c1, c2, c3
           end

           readWrite >> using address split
               0: c0
            || 1: c1
            || 2: c2
            || 3: c3
           end$

           address >> using readWrite split
               true:  readAddress
            || false: writeAddress
           end$
```

```
        writeData >> using readAddress split
            0: MemCell(readWrite=c0)
         || 1: MemCell(readWrite=c1)
         || 2: MemCell(readWrite=c2)
         || 3: MemCell(readWrite=c3)
        end >> using writeAddress merge
            0: {}
         || 1: {}
         || 2: {}
         || 3: {}
        end >> readData
    end
```

2.3.4.8 Simple Microprocessor

The full microprocessor design is not presented here, but rather just the enumerated type declarations and tabular portions that make up the parts of it not already described in figures 2.13–2.15. The DecodeTable definition is the major part of a naïve translation of a *Yellow* definition into a state machine. More efficient encodings of the same system (especially when viewed at the level of the Decode unit as a whole) are probably possible.

```
type regSources     = enum(immed,memry,wback) end
type regDesinations = enum(aPipe,bPipe) end
type memTypes       = enum(isInstr,isData) end
type opcodes        = enum(LOD,STO,LDI,ALU1,ALU2) end
type alucodes       = enum(
    FetchInc,ReadMem,WriteMem,
    MOV,NEG,NOT,
    ADD,SHFT,AND,OR,XOR,CMP
```

```
) end


table DecodeTable(input  thisState,instr1,instr2,instr3;
                  output nextState,regCtl,aluCtl,memCtl,
                         immediate) =
    var x,y,z,op end

    // Instruction Fetch
       0,-       ,-,-   => 1,(out,0,aPipe),FetchInc,isInstr,-
    or 1,-       ,-,-   => 2,(in ,0,wback),-       ,-      ,-

    // Mem->Reg
    or 2,(LOD,x,y),-,-      => 3,-            ,-   ,isData ,-
    or 3,-,(LOD,x,y),-      => 4,(out,y,aPipe),ReadMem ,-  ,-
    or 4,-,-,(LOD,x,y)      => 0,(in ,x,memry),-       ,-  ,-

    // Reg->Mem
    or 2,(STO,x,y),-,-      => 3,(out,y,bPipe),-       ,-  ,-
    or 3,-,(STO,x,y),-      => 4,(out,x,aPipe),WriteMem,-  ,-
    or 4,-,-,(STO,x,y)      => 0,-            ,-       ,-  ,-

    // Imm->Reg
    or 2,(LDI,x,v),-,-      => 3,-            ,-       ,-  ,v
    or 3,-,(LDI,x,v),-      => 4,-            ,-       ,-  ,-
    or 4,-,-,(LDI,x,v)      => 0,(in, x,immed),-       ,-  ,-

    // op(Reg)->Reg
    or 2,(ALU1,x,op,y),-,-  => 3,-            ,-       ,-  ,-
```

```
    or 3,-,(ALU1,x,op,y),-   => 4,(out,y,aPipe),op      ,-  ,-
    or 4,-,-,(ALU1,x,op,y)   => 0,(in ,x,wback),-       ,-  ,-

    // op(Reg,Reg)->Reg
    or 2,(ALU2,x,op,y,z),-,- => 3,(out,z,bPipe),-       ,-  ,-
    or 3,-,(ALU2,x,op,y,z),- => 4,(out,y,aPipe),op      ,-  ,-
    or 4,-,-,(ALU2,x,op,y,z) => 0,(in ,x,wback),-       ,-  ,-
end

table AluTable(input  operation,A,B;
               output writeback,address,data) =
    function cmp(value x,y:integer):integer =
        (((x>y) cast integer) shl 1) \/
        (((x=y) cast integer) shl 0)
    end
    var x,y end

    // Special Operations
       FetchInc,x,- => x+1     ,x,-
    or ReadMem ,x,- => -       ,x,-
    or WriteMem,x,y => -       ,x,y
    // Unary Operations
    or MOV     ,x,- => x       ,-,-
    or NEG     ,x,- => -x      ,-,-
    or NOT     ,x,- => ~x      ,-,-
    // Binary Operations
    or ADD     ,x,y => x+y     ,-,-
    or SHFT    ,x,y => x shl y ,-,-
    or AND     ,x,y => x/\y    ,-,-
```

```
        or OR      ,x,y => x\/y    ,-,-
        or XOR     ,x,y => x^y     ,-,-
        or CMP     ,x,y => cmp(x,y),-,-
    end
```

2.3.5 Semantics

The formal semantics of textual *Green* is defined by translation into the process algebra *APA* [Wil96], which is similar to CCS except that its communication command is better suited for the data-pushing style of micropipelines, and the communication of data values is modelled directly, as opposed to being implemented on top of value-free communication. The syntax of the communication process definition is:

$$\mathbf{sync}\{\mathit{ibinds}\}\{\mathit{obinds}\}(\mathit{bodyproc})$$

where *ibinds* is a list of input actions, *obinds* is a list of output actions, and *bodyproc* is a process. The execution of this works by waiting for all the input actions listed to become available, and once this is the case, the names in those actions are bound to the values on the channels and the body and output actions are performed. When all the output actions have been performed, and the body has been evaluated, the input actions complete and the process terminates.

An example of a function node (applying an addition operation to two streams producing a single result) is:

$$\mathit{Add}(i_1, i_2, o_1) \triangleq \mathbf{loop}\ \mathit{true} \longrightarrow \mathbf{sync}\{i_1?x, i_2?y\}\{o_1!(x+y)\}(\mathbf{skip})$$

and an example of a buffer is:

$$\mathit{Buf}(i_1, o_1) \triangleq \mathbf{loop}\ \mathit{true} \longrightarrow \{\mathbf{sync}\{i_1?x\}\{\}(r := x); \mathbf{sync}\{\}\{o_1!r\}(\mathbf{skip})\}$$

Defined in the *APA* core are a number of basic processes (see Table 2.3) which implement the semantics of most of pictorial *Green*. The definition of *Buf*() above is that of $\mathcal{B}_\varepsilon$

***APA* Symbol**	**Approx. Meaning**	**Parameter**
$\mathcal{A}$	Arbitrate	—
$\mathcal{B}_\varepsilon/\mathcal{B}_x$	Buffer	empty/containing x
$\mathcal{C}_x$	Constant	generating value x
$\mathcal{D}_n$	Duplicate	n-way (data) output
$\mathcal{F}_f$	Function	of function f
$\mathcal{M}_n$	Merge	n-way (data) input
$\mathcal{N}$	Sink	—
$\mathcal{S}_n$	Split	n-way (data) output
$\mathcal{W}$	While	—
$\mathcal{R}_n$	Repetition (for)	repeated n times
$\mathcal{I}_s$	Input	named-channel s
$\mathcal{O}_s$	Output	named-channel s

Table 2.3: Basic *APA* Operators

Chapter 3

The *Rainbow* Design Environment

One of the key features of *Green* is the pictorial sublanguage, which provides an intuitive interface to pipeline programming. The successful use of such a language depends on the ability to create, update, simulate and verify designs created with it. The *Rainbow* environment is intended to provide a framework for the performance of these actions, enabling the development of *Green* designs in conjunction with the rest of *Rainbow* in as natural fashion as possible.

3.1 Environment Design Issues

The key issues of the *Rainbow* editing and simulation environment are:

- Extensibility through its foundation on Tcl/Tk[Ous94]. This allows the easy addition of new components to the environment, even by the users of the system. The use of Tcl also decreases the development cycle time, since simple changes can be made to an already running system, and complex modules can be tested in isolation.

- Development of the system through the production of trial versions and giving these to the target audience for feedback, ensuring high levels of user satisfaction. This has lead to the use of intuitive actions to manipulate designs, allowing access to complex functionality through a simple[1] interface. This is greatly assisted by the use of graphical metaphors for design elements, which naturally leads to an inherently intuitive design construction style.

- Information overloading and management, so that the total amount visible at any time is controlled, with the focus of the users attention displaying additional context-related information. This is enhanced through the use of colour to highlight the state of the systems under simulation.

- Design management support, allowing the manipulation of complex hierarchic designs, thus helping users to manage the complexity of their own designs.

The biggest gains from the graphical environment, that it allows the intuitive management of very large amounts of information, is also one of its biggest problems, since it makes it hard to specify components of designs where a graphic fails to capture the nature of the actions being performed. In particular, type, function and table declarations are not good in a graphical form since they are hard to compress into a visual form without overwhelming the rest of a design.

The lack of enforced naming also causes problems in that the usual schemes for displaying traces of the actions on communications channels (as used in both the Cadence and ELLA design environments) are difficult to use here. The alternative of direct display of values is available, but this tends to become unwieldy when the values being passed have large textual representations.

[1] A simple interface is one which does not need a large explanation to enable the user to use it, or the system underlying it.

3.2 Architecture of System

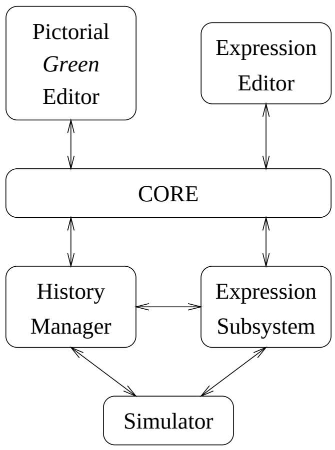


Figure 3.1: Basic Architecture of the *Rainbow* Environment

Figure 3.1 shows the basic arrangement of the environment. The main reason for the division into these modules are that it allows the separate development and testing of the editing modules. The six main modules in this model (though more can be added to increase the functionality of the overall system) are:

Core: manages communication between the editors and provides basic services common to other systems. Services provided include both the hierarchy manager and the object manager. The core also manages the gross scheduling of the environment.

Simulator: performs all the simulation services, taking a flattened net-list of a design and producing a trace of the actions performed by the design (doing so in a demand-driven fashion). The action of functional nodes is performed by calling the expression subsystem.

History Manager: performs net-list flattening and converts the events generated by the simulator into the displayed state histories for each communications channel. It then issues the commands to the displayed channels to display the values carried by them during the step in the simulation that the user is currently examining.

Expression Subsystem: provides the expression compilation, evaluation and formatting services.

Pictorial *Green* Editor: allows the user to perform the graphical editing of a pictorial *Green* design, with the core managing the multiple windows that are used in hierarchic editing.

Expression Editor: provides a user interface to expression writing, interfaced to the *Green* editor and the expression subsystem which provide it with its underlying services and its usage context.

The user interface and connecting components are written using Tcl/Tk for easy extensibility, and the simulation, history and expression engines are written in C for efficiency and speed.

3.3 Description of Editor

As can be seen in figure 3.2, the editor consists of a scrollable area (which expands to contain the layout placed within it) and a tool bar beside it. A device can be added to the current layout by dragging a copy of it from the tool bar at the side and dropping it where it is desired. Commonly used operations on the whole design are accomplished by pressing the button associated with that operation on the button-bar at the side of the window, though all of these actions (as well as others that are less commonly performed) are available in the menubar at the top of the window as well.

When devices are wired together, a check is made that it is only by connecting an input to an output. The other main constraint imposed by the system is the

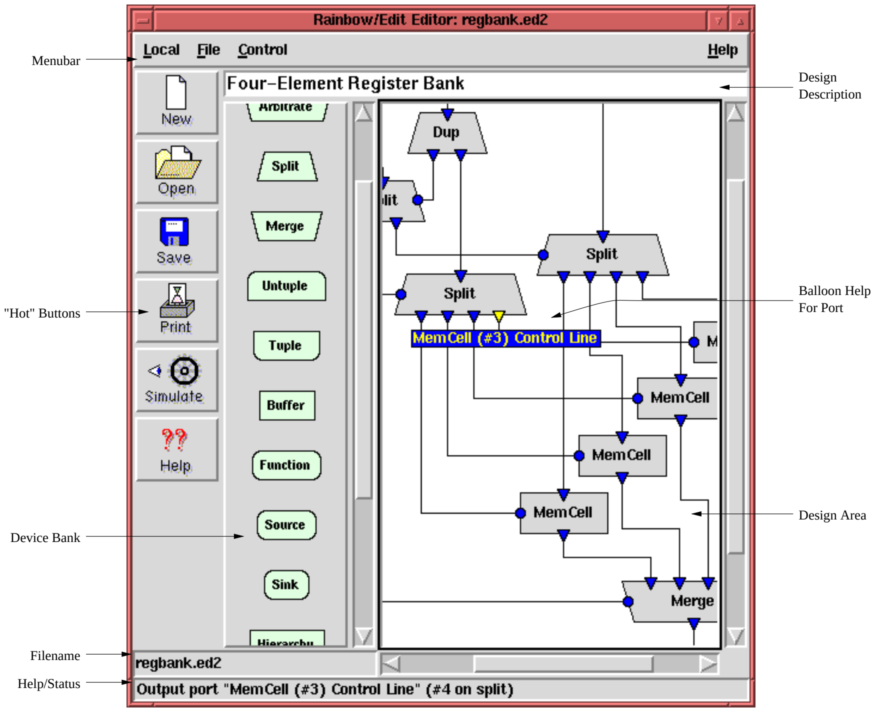


Figure 3.2: The Main Environment Screen

direct linking of the number of external ports in a design to the number of ports on the hierarchic node containing that design.

Highlighted devices with labels defined will have those labels displayed in a help balloon (as illustrated for one of the ports of a four-way split). The use of pop-up balloons allows considerably more information to be presented on a diagram than would be otherwise displayed. In order to assist in design layout, ports may be dragged round the edge of the device to which they are attached. Since a particular sub-design might be needed in several higher-level designs, a sub-design can be written to and read from disk independently of the top-level of a particular design hierarchy.

The production of printed images and encapsulated PostScript diagrams is done through a single interface, except with a printing command being the target in the first case, and a file being the target in the second case.

3.4 Animation of Simulator

An essential feature of the environment is the ability to simulate a design created with the editor with the events happening during simulation being displayed directly on the design screen so that the user can check that a design they have created behaves as they expect. Channel values are calculated by the simulator, and are displayed directly on the wires, with one form of label (dark lettering on a light background) indicating that a value is present and has been acknowledged, and another form of label (light lettering on a dark background) indicating that though the value is present, it is not flowing in this step due to being blocked somewhere in the pipeline by a device which is either busy elsewhere or waiting for a another value to synchronise with.

The simulation may be advanced either a step at a time, made to run continuously, or told to go any particular step. Since a full history is maintained, allowing the examination of previous steps without restarting the simulator, the target step may be anywhere in the entire history of the system, and not just in the future, and stepping and running can performed from those points onwards as well as from the final state in the simulation.

The hierarchy of editors may be navigated during simulation, although modification of any of the nodes is prevented (so preserving consistency during the simulation).

3.4.1 An Example Simulation

In the example in Figures 3.3–3.5, a system is simulating which produces on its output the tuple of the value input together with the previous value input.

In the first step, the initial value in the buffer, 0, together with the value from the input (represented by a random source) are tupled together producing the output (13,0). At the same time, the input value is also directed into the buffer,

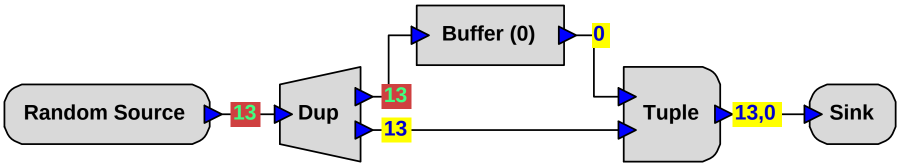


Figure 3.3: First Simulation Step

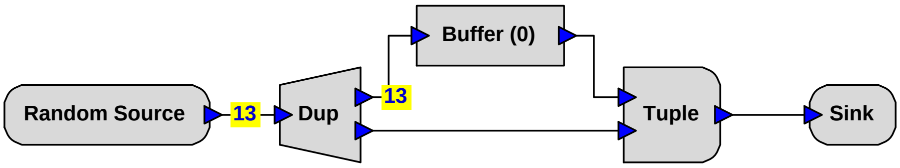


Figure 3.4: Second Simulation Step

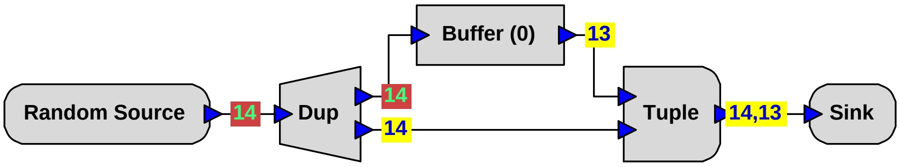


Figure 3.5: Third Simulation Step

but this value cannot be accepted since the buffer is still in its value-writing mode of operation, leaving the input value blocked at the buffers input, with this blocking propagating back through the duplicator to the source.

In the second step, the buffer is now able to read and accepts the value from the duplicator, releasing the block. Note that the other output of the duplicator no longer presents a value as it has already been accepted in step two.

Finally, the third step behaves in the same way as the first step, with the exception that the buffer now contains a different value (the value 13, as this was the value input in the previous step), and this is what is tupled with the new random input value to produce the output (14,13).

3.4.2 Brief Description of Simulator

The simulator[2] works by taking a net-list, which it performs some elementary checking on (principally that all the inputs and outputs are connected).

The simulator then evaluates a step by first pushing values from all source-like nodes (**Source**s and **Buffer**s with contents) as far as possible — until they encounter a sink, empty buffer or unsatisfied synchronisation point such as a two-input function where one of the inputs is not yet available. This models the request stages of the transactions. **Arbitrate** nodes delay their internal decision until the end of the push-forward stage, allowing a random decision to be made in the cases where both channels have asked to perform an action in the same time step, and allowing the unit to function deterministically in all other cases. When an **Arbitrate** node makes a choice, it then behaves like an ordinary node until it receives its acknowledgement, and the value chosen from its inputs gets passed on to the succeeding nodes in the network.

Once all values have been pushed forwards as far as possible, acknowledgements are pushed back from all sink-like nodes (**Sink**s and **Buffer**s with no value contained previously) that have just accepted a value using very similar rules to the forward stage except applied in reverse (this being slightly simpler due to the fact that data is not being passed at this point). This models the acknowledgement stages of the transactions.

When the actions of a particular step have all been performed, the simulator waits until the editor requests the evaluation of the next step.

A trace of the actions performed by the simulator is passed back to the history management unit of the core editor. The history manager determines what values are to be displayed on each channel at any given time. It then issues commands to the pictorial *Green* editor so that the values are displayed on the channels.

[2] The simulator was developed separately by Brian Marsden, and has the ability to operate as a stand-alone system.

For a fuller description of the simulator system, see [BFG$^+$96a].

3.5 Discussion

3.5.1 Other Hardware Design Systems

3.5.1.1 Cadence

Cadence is a very large system, based on VHDL[Coe89] which has a graphical layout system as one of its major components. However, it suffers in the high-level micropipeline design field due to the use of VHDL as its base. Some of the principal problems are the fact that wires drawn in Cadence are not full-handshake communications channels but rather something closer to real wires. This means that in order to implement a communications channel, the request, data and acknowledgement wires have to be explicitly created. Furthermore, since there are no primitives for handling pipeline communications directly, the user is forced into creating them for themselves. This last problem is compounded by the fact that VHDL requires concrete types to be specified for all wires, rather than being able to infer them from what those wires are connected to, with the net result that code reuse between pipelines carrying different types is virtually nonexistent.

Cadence is also (at least partially) based on a scripting mechanism, but as this is a custom language that is only used in this tool, it is difficult for someone without substantial background to modify.

Cadence does have a number of good features though, like its hierarchic library manager, its debugging tools and the ability to use foreign code to handle complex parts of a simulation. This makes the overall system good enough to perform simulations of entire microprocessors which can then run a full operating system, a good thing since it permits the taking of statistics using actual applications before the production of any hardware.

3.5.1.2 ELLA

The ELLA environment[BGMW94] is based on a Lisp system, so like both the *Rainbow* editor and Cadence, it can be extended by the user, and the dependence on a widely known language is an advantage in this respect. However, the ELLA graphical design tool is considerably more limited than the *Rainbow* editor, due to the fact that its subcomponents have a fixed shape, and the ports on them, despite being possible to drag round the device, do not have their layout configuration preserved between editing sessions. It also is not possible to directly place a design within another design, requiring that the user go through the library manager instead. The most serious handicap with the ELLA system was that it was a research system — there were many features missing which gave an overall bad impression. However, the degree of integration between the tools that were implemented was a good aspect, with it being easy to take designs produced using either the graphical and textual input systems, and check that they were the same through simulation and formal verification. The ability to perform symbolic execution (so that unknown values are propagated through the system and form part of the histories of the devices) was also useful to the analysis of a design.

3.5.1.3 PEP

This system[Gra94], also based largely on Tcl/Tk, offers much greater quantities of formal verification support than the above two systems, being based on PetriNets. It is a generic system without specialised support for hardware design, and it also doesn't handle data dependencies, resulting in the reporting of more deadlocks in the system than actually exist. The intuitive way that the execution of a design can be followed is an excellent feature though.

3.5.2 A Brief History of the Environment

The environment started initially as a system for producing diagrams of layouts due to the limitations of the drawing tools available at the time[3]. The basic set of devices were those thought to be most common in an abstracted micropipeline scheme. The development of this version was greatly eased by ability of Tcl/Tk to support rapid application prototyping, allowing the production of a system visually very similar to the final version very shortly after the commencement of development.

Once it became clear from consultation with the engineers in the AMULET group that the graphical system could be useful as more than just a conceptualisation aid, it was decided to add a mechanism for simulating designs made with this system. Direct annotation of the values flowing along the communications channels was chosen as the simulation output representation, as this provided the most direct metaphor of what was happening. The other main techniques of simulation output representation (such as signal diagrams) suffer from the fact that they disassociate the communications channel from the values that are passing along it without an obvious means of providing an intuitive connection between the two (pictorial *Green* channels being unnamed).

The editor system was completely reimplemented so as to decrease the complexity of the graphical editor (which made the whole system object-oriented in order to localise functionality as much as possible) while adding greater capabilities (like draggable ports and hierarchic nodes). Following this reorganisation, the

[3]This was before a reasonable syntax for textual *Green* had been developed, though the semantic issues were being studied closely at the time. It was only after experimenting with pictorial *Green* for several months that an initial syntax for *Green* was developed.

current view of the system as an environment for supporting the whole of *Rainbow* (and thus modelling the design philosophy of *Rainbow* as a set of interacting components) came into being.

Chapter 4

Conclusions and Future Work

4.1 Conclusion

The aims of *Green* (and *Rainbow*) are to provide a language framework for the development of asynchronous bundled-data micropipelined hardware upon which tools can operate, providing services such as simulation, power estimation, and deadlock checking.

It has been demonstrated that *Green* is a language capable of building complex components out of its basic building blocks, with the basic set of pictorial *Green* being sufficient for most designs, and the full power of the tabular descriptions allowing the creation of such functionality as is not easily described in a pictorial form. This process has been illustrated through the development of while loops and register banks, together with examples that use these newly created constructs.

The static semantics of textual *Green* have been presented through the use of types, and introduced the dynamic semantics through a translation (partially given in this document) into the process algebra *APA*. The use of types has provided a very easy way to describe how *Green* components fit together, especially as it allows the use of a single type scheme to describe the whole of *Green* (and

Rainbow) while allowing the imposition of restrictions on what types can be passed on channels (especially preventing the passing of channels and functions on channels). The fact that it encourages the view that procedures are functions over streams was an added bonus.

It has also been shown that a design environment, a prerequisite for structured design development in a pictorial language, allows the intuitive display of event-based simulation information through the annotation of communications channels with the values they are carrying during that simulation step. Since the environment is designed as an extensible framework, enabling the modular design of tools which can then be fitted into a coherent whole, it is in a position of being able to be extended to handle more sophisticated analyses within the same mixed pictorial/textual paradigm.

4.2 Future Work

There are a number of areas which still need substantial work in both *Green* and the environment. The incorporation of these within the existing environment is facilitated by the component-based structure of *Rainbow* and the the environment as a whole is thus easier to expand to cope with the changes outlined below.

There is still much more work to be done on the common semantic framework of *Rainbow*, particularly at the interfaces between the components. As more languages get added to *Rainbow* (currently both *Blue* and *Red* are not defined, with only the basic desired properties of the two languages known) there are bound to be more complex semantic questions. Of particular concern are the problems that will happen when *Blue* gets added, as that exposes the handshaking within the communications of the rest of *Rainbow*. The development of tools to perform some form of semantic analysis (such as checking for deadlocks) will also be a major component of the work ahead.

An immediate task is the adding of a system of modules to *Green* (and the rest of *Rainbow*) so that users can import procedures and devices from other peoples work and export those parts of their own system that they want to make available to the rest of world. Once modules have been implemented, it will become possible to create libraries of devices and procedures (such as making available the memory elements, the while loop and the if block discussed in this document to users of pictorial *Green*).

One aspect of VHDL that was particularly good was the ability to simulate entire computer systems through the use of foreign code modules to simulate the core parts of the operating system. Adding such capabilities to *Rainbow* (in particular, to textual and pictorial *Green*, the levels that it is intended that designers will mostly produce hardware models at) would allow much more complete testing of designs than is currently possible.

There is also the question of whether the current type system would be better for having the ability to build infinite data objects (such as lists) in some components of *Rainbow*, allowing the expression of very complex behaviour without resorting to foreign code modules. The abstract types of SML look like an attractive model, especially since they offer a restricted interface to the underlying data structure (which would hopefully be easier to reason about).

On the environment front, it would be very nice to include a textual entry system together with an automatic conversion from textual to pictorial *Green* [1] (the reverse process is simple to perform due to the definition of pictorial *Green* in terms of textual *Green*). This has two components: the analysis of the text into a netlist of pipelines, and the laying out of those pipelines so as to make the design easy to understand without post-layout adjustments.

Currently, there is a compiler for *Yellow* and *Green* under development[2] but

[1] This is currently the subject of a third-year project.
[2] This is work by Peter Jinks.

an animation system to go with this is also required. As part of this, there is the need to develop a suitable virtual machine model for executing pieces of *Yellow* and *Green* (and the other *Rainbow* languages when they are defined), though the production of output from the compiler in other target forms (such as value-passing CCS, Occam or VHDL) is also desired.

Work also needs to be done on determining whether the systems in the current environment are capable of scaling to large designs. A system is not of much practical use if it cannot be used by the engineers on the systems that they are developing, and relatively little work has been done in this area to date.

The AMULET group would like the capability added to *Rainbow* to perform estimations of time taken to perform a simulation, the electrical power that would be consumed during such a simulation, and the area of silicon occupied by the design should it be fabricated naïvely from the design. Adding these estimates would require changes to the language (which currently has no way of representing the data required for such estimations) and the development of a tool to perform the calculations to derive the final figures involved.

Another feature that it would be nice to add to *Green* (or *Yellow*, where it might be better suited) would be the capability to specify that a module could be shared between pipelines without needing a heavyweight mechanism like arbitration. This feature is one of the major points of Tangram (and the closely related language, Balsa), and its inclusion within *Rainbow* would be a bonus when comparing the two languages.

However, the *Rainbow* language framework was designed to be extensible, with the capacity to add extra functionality being one of the first design aims of the whole system. The development environment reflects this, and its foundation on an extensible framework intended to encourage modular design should make the further addition of systems a relatively painless process, even encouraging end-users to add their own enhancements over the core system through a well-documented and easily learnt extension language.

Bibliography

[Asy95] *Asynchronous Design Methodologies*, South Bank University, London, May 1995.

[BD95] Graham Birtwistle and Al Davis, editors. *Asynchronous Digital Circuit Design*. Springer, 1995.

[BFG+96a] 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.

[BFG+96b] H. Barringer, D. Fellows, G.D. Gough, P. Jinks, and A. Williams. A formally based framework for supporting design and analysis of asynchronous hardware systems. In NFMW96 [NFM96].

[BFGW] H. Barringer, D.K. Fellows, G.D. Gough, and A. Williams. The Rainbow Language Reference Manual. *Unpublished Work*.

[BGMW94] H. Barringer, G.D. Gough, B.Q. Monahan, and A. Williams. The ELLA Verification Environment: A Tutorial Introduction. Technical Report UMCS-94-12-2, University of Manchester, December 1994. See [Man96].

[CDJ84] Fred B. Chambers, David A. Duce, and Gillian P. Jones, editors. *Distributed Computing*. Academic Press, 1984.

[Coe89] D. Coelho. *The VHDL Handbook*. Kluwer, 1989.

[FDG+93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993. Available via World-Wide Web, URL: `ftp://ftp.cs.man.ac.uk/pub/amulet/papers/VLSI93.ps.Z`.

[Fur95] 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.

[Gra94] Bernd Grahlmann. An Introduction to the Principles, the Functionality and the Use of the PEP System. In E. Best and H. Fleishhack, editors, *PEP Workshop, University of Humboldt, Berlin, 1994*, October 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Liu95] Ying Liu. *AMULET1: Specification and Verification in CCS*. PhD thesis, Department of Computer Science, University of Calgary, September 1995.

[Man96] Department of Computer Science Technical Report Series, The Computer Library, University of Manchester, available on the World-Wide Web, URL: `http://www.cs.man.ac.uk/csonly/cstechrep/index.html`, 1996. British Library International Standard Serial

Numbers (ISSN) for Departmental Technical Reports: ISSN 1361 - 6153 (printed versions), ISSN 1361 - 6161 (ftp versions).

[McM92] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 1992.

[Mol94] F. Moller. *The Edinburgh Concurrency WorkBench*. Department of Computer Science, University of Edinburgh, 7.0 edition, November 1994. Also see CWB Web pages, URL: `http://www.dcs.ed.ac.uk/home/cwb/`.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NFM96] *Proceedings of the Northern Formal Methods Workshop*, Ilkley, UK, September 1996.

[Ous94] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[Rai96] Rainbow. 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`, 1996.

[Sut89] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6), June 1989.

[TLB96] C. Tofts, Y. Liu, and G. Birtwistle. State Space Reduction for Asynchronous Micropipelines. In NFMW96 [NFM96].

[vB92] K. van Berkel. *Handshake Circuits: an Intermediary between Communicating Processes and VLSI.* PhD thesis, Eindhoven University of Technology, 1992.

[vB94] K. van Berkel. VLSI Programming of Asynchronous Circuits for Low Power. Technical report, Philips Research Laboratories, Eindhoven, The Netherlands, September 1994.

[Wil96] A. Williams. APA Definition. *Unpublished Work*, 1996.

Appendix A

BNF Syntax for Textual *Rainbow*

$$
\begin{array}{lcl}
\textit{expression} & ::= & (\ \textit{expression}\ (,\ \textit{expression})^{+}\ ) \\
 & | & \textbf{if}\ \textit{expression}\ \textbf{then}\ \textit{expression}\ \textbf{else}\ \textit{expression}\ \textbf{end\_if} \\
 & | & \textit{expression}\ \hat{}\ \textit{expression} \\
 & | & \textit{expression} \wedge \textit{expression} \\
 & | & \textit{expression} \vee \textit{expression} \\
 & | & \textbf{not}\ \textit{expression} \\
 & | & \textit{expression} > \textit{expression} \\
 & | & \textit{expression} < \textit{expression} \\
 & | & \textit{expression} >= \textit{expression} \\
 & | & \textit{expression} <= \textit{expression} \\
 & | & \textit{expression} = \textit{expression} \\
 & | & \textit{expression}\ \textbf{!=}\ \textit{expression} \\
 & | & \textit{expression}\ \textbf{cast}\ \textit{type} \\
 & | & \textit{expression}\ \textbf{fit}\ \textit{type} \\
 & | & \textit{expression}\ \textbf{shl}\ \textit{expression} \\
 & | & \textit{expression}\ \textbf{shr}\ \textit{expression}
\end{array}
$$

| *expression* + *expression*
| *expression* -*expression*
| *expression* * *expression*
| *expression* / *expression*
| *expression* **mod** *expression*
| **abs** *expression*
| ~ *expression*
| *expression* . *expression*
| *expression* [*expression*]
| (*expression*)
| *A_NUMBER*
| *qualified_name*
| *qualified_name* **in** *A_NAME*
| *invoke*

type ::= *A_TYVAR*
| *qualified_name*
| (*type*)
| **enum** (*names*)
| **union** (*type* (, *type*)$^+$)
| (*type* ^ *type*)
| (*type* (, *type*)$^+$)
| (*type* (|| *type*)$^+$)
| (*A_NAME* : $type_1$, . . . , *A_NAME* : $type_{k+1}$)
| [*expression* .. *expression*]

program ::= *declaration** *declaration*
| *declaration** *invoke*

invoke	::=	*qualified_name* ($expression_1$, ... , $expression_k$)
	|	*qualified_name* ($named\_arg_1$, ... , $named\_arg_{k+1}$)
named_arg	::=	*A_NAME* = *expression*
declaration	::=	*statefull*
	|	*stateless*
	|	*decl_g_only*
	|	*decl_y_only*
decl_g	::=	*statefull*
	|	*stateless*
	|	*decl_g_only*
decl_y	::=	*statefull*
	|	*stateless*
	|	*decl_y_only*
t_decl	::=	*stateless*
	|	**var** *typed_names* **end**
statefull	::=	**copy** *typed_name_list* = *expression* **end**
	|	**chan** *names* **:** *type* **end**
	|	**chan** *names* **end**
stateless	::=	**alias** *typed_name_list* = *expression* **end**
	|	**constant** *typed_name_list* = *expression* **end**
	|	**type** *A_NAME* = *type* **end**
	|	**function** *A_NAME f_signature* = *f_body* **end**
decl_g_only	::=	**memory** *names* **:** *type* **end**
	|	**green** *A_NAME p_signature* = *block_g* **end**
	|	**table** *A_NAME t_signature* = *table_body* **end**
decl_y_only	::=	**reg** $r\_signature_1$, ... , $r\_signature_{k+1}$ **end**

	$\mid$	**yellow** *A_NAME* *p_signature* = *block_y* **end**
typed_name_list	::=	*typed_names*
	$\mid$	**(** *typed_names* **)**
typed_names	::=	$typed\_name_1$ **,** ... **,** $typed\_name_{k+1}$
typed_name	::=	*A_NAME* (**:** *type*)?
names	::=	A_NAME_1 **,** ... **,** A_NAME_{k+1}
p_signature	::=	**(** p_params_1 **;** ... **;** p_params_k **)**
p_params	::=	**input** *names* **:** *type*
	$\mid$	**input** *names*
	$\mid$	**value** *names* **:** *type*
	$\mid$	**value** *names*
	$\mid$	**output** *names* **:** *type*
	$\mid$	**output** *names*
	$\mid$	**proc** *names* **:** *p_signature*
	$\mid$	**proc** *names*
	$\mid$	**function** *names* **:** *f_signature*
	$\mid$	**function** *names*
t_signature	::=	**(** t_params_1 **;** ... **;** t_params_{k+1} **)**
t_params	::=	**input** *names* **:** *type*
	$\mid$	**input** *names*
	$\mid$	**output** *names* **:** *type*
	$\mid$	**output** *names*
f_signature	::=	**(** f_params_1 **;** ... **;** f_params_k **)** **:** *type*
	$\mid$	**(** f_params_1 **;** ... **;** f_params_k **)**

f_params	::=	**value** *names* **:** *type*
	|	**value** *names*
	|	**function** *names* **:** *f_signature*
	|	**function** *names*
f_body	::=	{ *stateless** *expression* }
table_body	::=	{ *t_decl** *t_body* }
t_body	::=	t_row_1 **or** ... **or** t_row_{k+1}
t_row	::=	*i_pattern* **when** *expression* => *o_pattern*
	|	*i_pattern* => *o_pattern*
i_pattern	::=	$i\_patelem_1$ **,** ... **,** $i\_patelem_{k+1}$
i_patelem	::=	**-**
	|	*t_item*
t_item	::=	*A_NAME*
	|	*A_NUMBER*
	|	**(** t_item_1 **,** ... **,** t_item_{k+1} **)**
o_pattern	::=	$o\_patelem_1$ **,** ... **,** $o\_patelem_{k+1}$
o_patelem	::=	**-**
	|	*expression*
r_signature	::=	*A_NAME* (**:** *type*)? (**:=** *expression*)?
qualified_name	::=	*A_NAME* **::** *A_NAME*

	$\mid$	*A_NAME*
block_g	::=	{ *decl_g** *parallel_g* }
parallel_g	::=	*sequence_list*$_1$ $\parallel \ldots \parallel$ *sequence_list*$_{k+1}$
labelled_parallel_g	::=	*labelled_sequence_g* ($\parallel$ *labelled_sequence_g*)$^+$
labelled_sequence_g	::=	*expression* **:** *sequence_list*
sequence_list	::=	$\gg$ *sequence*$_1$ **$** $\ldots$ **$** *sequence*$_{k+1}$ $\gg$
	$\mid$	$\gg$ *sequence*$_1$ **$** $\ldots$ **$** *sequence*$_{k+1}$ **$**
	$\mid$	$\gg$ *sequence*$_1$ **$** $\ldots$ **$** *sequence*$_{k+1}$
	$\mid$	*sequence*$_1$ **$** $\ldots$ **$** *sequence*$_{k+1}$ $\gg$
	$\mid$	*sequence*$_1$ **$** $\ldots$ **$** *sequence*$_{k+1}$ **$**
	$\mid$	*sequence*$_1$ **$** $\ldots$ **$** *sequence*$_{k+1}$
sequence	::=	*command_dot_g*$_1$ $\gg \ldots \gg$ *command_dot_g*$_{k+1}$
command_dot_g	::=	*command_g* (. *A_NAME*)*
command_g	::=	**sink**
	$\mid$	**buffer** **(** *expression* **,** *expression* **)**
	$\mid$	**buffer** **(** *expression* **)**
	$\mid$	**arbitrate** *sequence_list* $\parallel$ *parallel_g* **end_arbitrate**
	$\mid$	*block_g*
	$\mid$	**source** *expression*
	$\mid$	*A_NAME*
	$\mid$	*invoke*
	$\mid$	*using* **split** *labelled_parallel_g* **end_split**
	$\mid$	*using* **merge** *labelled_parallel_g* **end_merge**

	|	**if** *parallel_g* **then** *parallel_g* **else** *parallel_g* **end_if**
	|	**case** *parallel_g* **of** *labelled_parallel_g* **end_case**
	|	**for** *expression* **do** *parallel_g* **end_for**
	|	**while** *parallel_g* **do** *parallel_g* **end_while**
using	::=	**using input** $\gg$ *parallel_g*
	|	**using** *parallel_g*
block_y	::=	$\{$*decl_y** *command_y*$_1$ *y_sep* ... *y_sep* *command_y*$_k\}$
y_sep	::=	**;**
	|	||
command_y	::=	*block_y*
	|	*invoke*
	|	*qualified_name* **!** *expression*
	|	*qualified_name* **!**
	|	*qualified_name* *modifier_y** **:=** *expression*
	|	**choice** *choices_y* **end_choice**
	|	**loop** *choices_y* **end_loop**
	|	**select** *acceptguard* (**or** *acceptguard*)* **end_select**
	|	*acceptguard*
choices_y	::=	*guardedchoice* (**or** *guardedchoice*)*
modifier_y	::=	[*expression*]
	|	. *A_NAME*
guardedchoice	::=	*expression* $->$ *command_y*
	|	*expression* $->$ **skip**

acceptguard	::=	{ *chanlist* **during** *command_y* } $->$ *command_y*
	|	{ *chanlist* **during** *command_y* }
	|	{ *chanlist* } $->$ *command_y*
	|	{ *chanlist* }
chanlist	::=	$chanread_1$ **,** . . . **,** $chanread_{k+1}$
chanread	::=	*qualified_name* **?** *A_NAME*

Appendix B

Simple Microprocessor

B.1 Instruction Set

There are five main classes of instruction, distinguished by the different actions that the processor performs internally in each of the cases:

Class	Written	Meaning
1. Memory Load	`LOD x,y`	$R_x := Mem[R_y]$
2. Memory Store	`STO x,y`	$Mem[R_x] := R_y$
3. Load Immediate	`LDI x,v`	$R_x := v$
4. Unary ALU Operation	`ALU1 x,op,y`	$R_x := op(R_y)$
5. Binary ALU Operation	`ALU2 x,op,y,z`	$R_x := op(R_y, R_z)$

All arithmetic is performed on signed values. The number and size of the registers is not defined in the model.

The following unary ALU operations are supported:

`MOV` Performs no operation on the value. Identity.

`NEG` Performs two's complement negation on the value.

`NOT` Inverts every bit in the value.

The following binary ALU operations are supported:

`ADD` Performs addition, $y + z$

`SHFT` Performs a (left-)shift operation, $y \times 2^z$

`AND` Performs bitwise boolean And, $y \wedge z$

`OR` Performs bitwise boolean Or, $y \vee z$

`XOR` Performs bitwise boolean Xor, $y \neq z$

`CMP` Compares for equality, placing in bit 0 whether the values are equal, and in bit 1 whether the first value is greater than the second.

The next instruction is fetched from the location identified in register 0 (usually identified as the PC) at the end of the current execute cycle. Absolute jumps can be performed by loading the PC, and relative jumps by adding offsets to it. Adding a calculated value performs a conditional.

Load Immediate of a positive value is guaranteed to fill at least the bottom half of the designated register with the value. Any bits not in the value supplied will be set to zero.

B.2 Sample Code

The code below is a (possible) translation of the following program in a Pascal-like pseudo-language:

```
var x:integer := 10;
var y:array [0..15] of integer;

do
    y[x] := 0;
    x := x-1;
```

```
until x=0;
```

```
LDI   1, #0          ;    zero     ⇐ 0
LDI   2, #1          ;    one      ⇐ 1
ALU1  3, NEG,2       ;    minusone ⇐ −one
LDI   4, #10         ;    x        ⇐ 10
LDI   7, #64         ; --- address of the array
ALU1  5, MOV,PC      ; loop:
ALU2  6, ADD,7,4     ; --- address of array element
STO   6, 1           ;    y[x]     ⇐ zero
ALU2  4, ADD,4,3     ;    x        ⇐ x + minusone
ALU2  6, CMP,4,1     ;    c        ⇐ x = zero
ALU2  6, AND,6,2     ; --- mask the correct bit
ALU2  PC,ADD,PC,6    ; --- skip next if c is true (bit set)
ALU1  PC,MOV,5       ;    goto loop
ALU2  PC,ADD,PC,3    ; --- terminate by jumping here
```

B.3 Full Definition of Microprocessor

```
// Full declaration of the microprocessor
//
// Donal K. Fellows, University of Manchester
//            fellowsd@cs.man.ac.uk    Sun Sep 22 17:26:06 BST 1996
//
// This version with only 8 registers, and no memory definition
```

```
// ----------------------------------------------------------------
// Basic nature of the registers
//
constant registerInit = 0 end        // Initial contents of registers
type registerContents = integer end  // Type of registers
type regIds           = [0 .. 7] end // Register ‘addresses’

// ----------------------------------------------------------------
// Type declarations.
//
type regSources     = enum(immed,memry,wback) end
type regDesinations = enum(aPipe,bPipe) end
type memTypes       = enum(isInstr,isData) end
type opcodes        = enum(LOD,STO,LDI,ALU1,ALU2) end
type regOps         = enum(in,out) end

type alucodes       = enum(
    FetchInc,ReadMem,WriteMem, // internal operations
    MOV,NEG,NOT,               // one argument
    ADD,SHFT,AND,OR,XOR,CMP    // two arguments
) end

type instructions   = union(
    ([LOD .. STO],  regIds,regIds),
    ([LDI .. LDI],  regIds,registerContents),
    ([ALU1 .. ALU1],regIds,[MOV .. NOT],regIds),
    ([ALU2 .. ALU2],regIds,[ADD .. CMP],regIds,regIds)
) end
```

```
type regControl     = union(
    ([in .. in],  regIds,regSources),     // read, to-reg,  from-pipe
    ([out .. out],regIds,regDestinations) // write,from-reg,to-pipe
) end

type memoryContents = union(
    instructions,    // From the 'text' segment
    registerContents // From the 'data' and 'stack' segments
) end               // These should have the same bit-sizes, somehow

// ----------------------------------------------------------------
// Table declarations
//
table DecodeTable(input  thisState,instr1,instr2,instr3;
                  output nextState,regCtl,aluCtl,memCtl,
                         immediate) =
    var x,y,z: regIds;
        v:     registerContents;
        op:    alucodes
    end

    // Instruction Fetch
       0,-        ,-,-       => 1,(out,0,aPipe),FetchInc,isInstr,-
    or 1,-        ,-,-       => 2,(in ,0,wback),-       ,-      ,-

    // Mem->Reg
    or 2,(LOD,x,y),-,-       => 3,-            ,-       ,isData ,-
    or 3,-,(LOD,x,y),-       => 4,(out,y,aPipe),ReadMem ,-      ,-
    or 4,-,-,(LOD,x,y)       => 0,(in ,x,memry),-       ,-      ,-
```

```
    // Reg->Mem
    or 2,(STO,x,y),-,-        => 3,(out,y,bPipe),-        ,-      ,-
    or 3,-,(STO,x,y),-        => 4,(out,x,aPipe),WriteMem,-      ,-
    or 4,-,-,(STO,x,y)        => 0,-              ,-        ,-      ,-

    // Imm->Reg
    or 2,(LDI,x,v),-,-        => 3,-              ,-        ,-      ,v
    or 3,-,(LDI,x,v),-        => 4,-              ,-        ,-      ,-
    or 4,-,-,(LDI,x,v)        => 0,(in, x,immed),-        ,-      ,-

    // op(Reg)->Reg
    or 2,(ALU1,x,op,y),-,-    => 3,-              ,-        ,-      ,-
    or 3,-,(ALU1,x,op,y),-    => 4,(out,y,aPipe),op       ,-      ,-
    or 4,-,-,(ALU1,x,op,y)    => 0,(in ,x,wback),-        ,-      ,-

    // op(Reg,Reg)->Reg
    or 2,(ALU2,x,op,y,z),-,- => 3,(out,z,bPipe),-        ,-      ,-
    or 3,-,(ALU2,x,op,y,z),- => 4,(out,y,aPipe),op       ,-      ,-
    or 4,-,-,(ALU2,x,op,y,z) => 0,(in ,x,wback),-        ,-      ,-
end

table AluTable(input  operation,A,B;
               output writeback,address,data) =
    function cmp(value x,y:registerContents) : registerContents =
        (((x>y) cast registerContents) shl 1) \/
        (((x=y) cast registerContents) shl 0)
    end
    var x,y:registerContents end
```

```
    // Special Operations
       FetchInc,x,- => x+1     ,x,-
    or ReadMem ,x,- => -       ,x,-
    or WriteMem,x,y => -       ,x,y
    // Unary Operations
    or MOV     ,x,- => x       ,-,-
    or NEG     ,x,- => -x      ,-,-
    or NOT     ,x,- => ~x      ,-,-
    // Binary Operations
    or ADD     ,x,y => x+y     ,-,-
    or SHFT    ,x,y => x shl y ,-,-
    or AND     ,x,y => x/\y    ,-,-
    or OR      ,x,y => x\/y    ,-,-
    or XOR     ,x,y => x^y     ,-,-
    or CMP     ,x,y => cmp(x,y),-,-
end

// ------------------------------------------------------------------
// Sublevel declarations
//
green Decode(input  i:      instructions;
             output memCtl: memTypes;
             output immed:  registerContents;
             output regCtl: regControl;
             output aluCtl: alucodes) =
    chan instr,thisState,nextState end

    i >> buffer(1) >> instr$
```

```
    nextState >> buffer(2,0) >> thisState$

    DecodeTable(thisState,instr,instr,instr,
                nextState,regCtl,aluCtl,memCtl,immed)
end

green Registers(input  control:regControl;
                input  immVal,memVal,aluVal:registerContents;
                output pipeA,pipeB:registerContents) =
    green RegBank(input  writeData: registerContents;
                  input  address:   regIds;
                  input  readWrite: regOps;
                  output readData:  registerContents) =
        // Basic memory cell - nested declaration
        green MemCell(input  writeData: registerContents;
                      input  readWrite: regOps;
                      output readData:  registerContents) =
            chan wbchan end

            using readWrite merge
                out: wbchan >>
             || in:  writeData >>
            end >> buffer(1) >> buffer(1,registerInit) >>
            using readWrite split
                out: >>wbchan
             || in:  {}
            end$
```

```
    wbchan >> readData
end

chan readAddress, writeAddress,
     c0, c1, c2, c3, c4, c5, c6, c7
end

readWrite >> using address split
    0: c0 || 1: c1 || 2: c2 || 3: c3
 || 4: c4 || 5: c5 || 6: c6 || 7: c7
end$

address >> using readWrite split
    out: readAddress
 || in:  writeAddress
end$

writeData >> using readAddress split
    0: MemCell(readWrite=c0)
 || 1: MemCell(readWrite=c1)
 || 2: MemCell(readWrite=c2)
 || 3: MemCell(readWrite=c3)
 || 4: MemCell(readWrite=c4)
 || 5: MemCell(readWrite=c5)
 || 6: MemCell(readWrite=c6)
 || 7: MemCell(readWrite=c7)
end >> using writeAddress merge
    0: {} || 1: {} || 2: {} || 3: {}
 || 4: {} || 5: {} || 6: {} || 7: {}
```

```
        end >> readData
    end

    chan sourcePipeline, destPipeline end

    control.2 >> using control.0 split
        in:  >> sourcePipeline
     || out: >> destPipeline
    end_split$

    using sourcePipeline merge
        immed: immVal >>
     || memry: memVal >>
     || wback: aluVal >>
    end_merge >>
    RegisteBank(address=control.1, readWrite=control.0)
    >> using destPipeline split
        aPipe: >> pipeA
     || bPipe: >> pipeB
    end_split$
end

// ------------------------------------------------------------------
// Main declaration
//
green Microprocessor(input  fromMem:         memoryContents;
                     output memAddr,memData: registerContents) =
    chan a,b,i,m,w: registerContents;
         instrs:    instructions;
```

```
        mc,rc,ac // Inferred types
    end

    instrs >> Decode(memCtl=mc, regCtl=rc, aluCtl=ac) >> i$

    fromMem >> using mc >> buffer(1) split
        isInstr: >>instrs
     || isData:  >>m
    end_split$

    {i>>buffer(1) || m>>buffer(1) || w>>buffer(1)} >>
    Registers(control=rc)
    >> {buffer(1) || buffer(1)} >>
    AluTable(operation=ac)
    >> {w || memAddr || memData}
end
```