A Formally Based Framework for Supporting Design and Analysis of Asynchronous Hardware Systems

Howard Barringer, Donal Fellows, Graham Gough, Pete Jinks, Alan Williams
Department of Computer Science, University of Manchester, UK*
email: howard@cs.man.ac.uk

1 Introduction

Digital hardware systems have, for the past 30 years, been designed using a synchronous approach, involving global clocking strategies. Recently, there has been renewed interest in asynchronous design techniques[1], with the aim of obtaining low-power designs combined with high speed operation, a combination which is becoming increasingly difficult to achieve with synchronous systems. Progress has been supported by the emergence of various asynchronous design methodologies, such as Sutherland's micropipeline philosophy[2], which sufficiently simplify the design process to render it tractable for commercially sized systems. However, although there is a perceived advantage in adopting an asynchronous style, this is more difficult to achieve in practice due to the lack of sophisticated asynchronous design support tools corresponding to those now available for supporting synchronous design. In particular, asynchronous hardware engineers require an abstract, high level description language to support rapid design evaluation and iteration; such a language requires supporting simulation and design analysis tools, for checking properties such as deadlock/progress and performance.

In this work, we aim to develop a design environment for micropipeline systems that offers a description framework, containing a suite of asynchronous hardware description languages (HDLs) equipped with formal semantics; this provides the basis for formal analysis tools. The work, especially the development of the framework, is being conducted in close collaboration with the AMULET Group at the University of Manchester. They have been investigating commercial-scale asynchronous micropipeline design, including versions of the ARM RISC processor[3]. They have identified particular features required of an asynchronous design environment that current systems lack, restricting design development.

Our philosophy, illustrated in past work on the synchronous HDL ELLA[4], is to embed application-specific formal analysis techniques into the more traditional hardware development system, so that the engineer is presented with a familiar design interface via the HDL, and is thus still able to use conventional design tools in the design cycle. The underlying formal design representation will act as the basis both for simulation tools and for standard formal analysis and verification methods. We will also develop novel analysis techniques that exploit characteristics of the design in order to provide more tractable analysis methods, again illustrated by our work on ELLA[4].

In this paper, we describe the *Rainbow* hardware description framework, and outline the characteristics of the underlying semantic model.

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

2 Rainbow

The *Rainbow* hardware description framework offers a range of description styles for asynchronous systems, in order to meet the requirements of hardware engineers, such as those in the AMULET group. The framework can be considered to be a collection of distinct sub-languages, offering multiple design styles and views of a design. However, full interworking between these is supported, so that the different components within a single design can be described in the style that is most appropriate or convenient. The framework offers rapid design description and simulation at a high level, so that the functionality and performance of the design can be assessed and experimented with at an early stage in the design cycle.

Rainbow includes a dataflow-style language based on the micropipeline communication primitives, which are similar to the Ada rendezvous communication. It also includes a control-flow style language for CSP-like descriptions, but again using rendezvous communication. Each language is given a formal semantics, enabling the language suite to form the basis of specialised formal verification and analysis tools for supporting a design. Ultimately, the framework will contain the following languages:

***Red*:** supports behavioural/specification descriptions using, for example, temporal logic or stream transformers[5].

***Yellow*:** describes designs using a control flow style and Ada-like rendezvous communication[6, 7].

***Green*:** supports hierarchical structural descriptions based on micropipeline primitives. It has schematic and textual versions — it is possible to convert from a schematic to a textual description, and to instantiate textual descriptions of components within a schematic.

***Blue*:** operates at a level below Green, and exposes handshaking control, similar to the CCS models of AMULET described in [8].

The formal semantics of Yellow and Green are defined by a specialised process algebra called *APA* (Asynchronous Process Algebra), which also operates at the micropipeline level: basic micropipeline components and combinators are modelled by atomic components and combinators in APA (see §3 below).

Figure 1 shows the prototype Rainbow framework: the user interacts with the system at the *Rainbow* level by describing a design using a combination of the sub-languages, indicated by the intersecting ovals. A single uniform semantics for Rainbow is defined by a translation to APA — a common set of data expressions is embedded into both. The internal APA representation will provide the basis for formal analysis and simulation tools, the results of which can be visualised via annotation of the design descriptions.

2.1 *Rainbow* Components

Visual Green enables a design to be entered as a schematic network hierarchy of communicating nodes. A set of basic nodes is provided, for buffers, functions and dataflow control, corresponding to those micropipeline primitives commonly used in asynchronous designs. Apart from these primitives, nodes may also contain Visual Green sub-networks, or components from other Rainbow sub-languages, such as Textual Green or Yellow. In this way a mixed-view design description can be built up. The resulting networks are similar to static dataflow networks described in [9], although in Green, the wires are stateless, and explicit buffering is used instead.

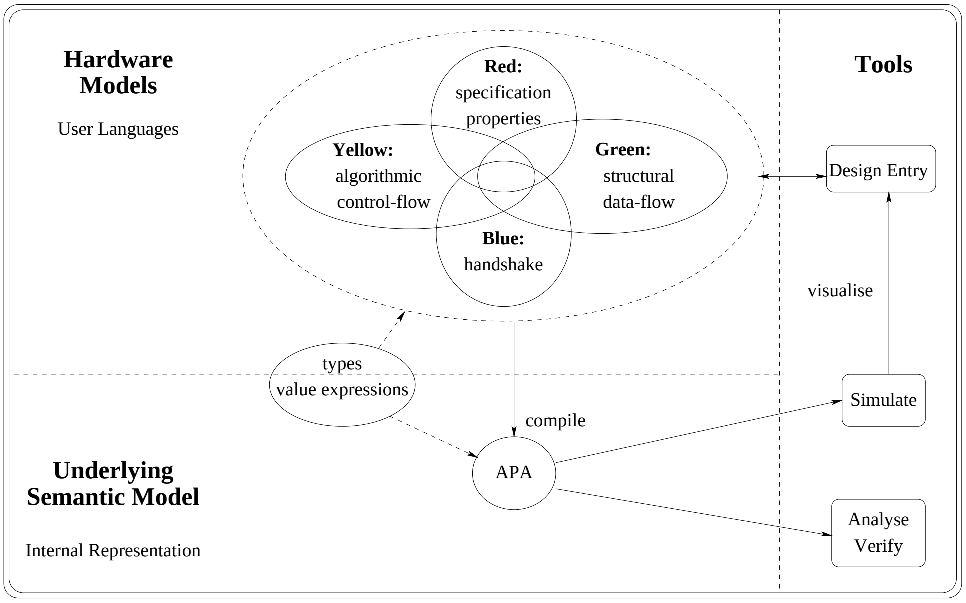

Figure 1: The Rainbow System

A textual dataflow description can be constructed using Textual Green procedures. In particular, the `table` construct provides a large degree of flexibility in the description of dataflow components, so that a node can control the consumption and generation of data values on its input and output streams.

In contrast, Yellow allows *evolving* descriptions to be constructed, using a control-flow style, similar in many respects to CSP[10] or Ada[7]; for example the basic call/accept communication primitives used are adaptations of those given in Ada. Again, other *Rainbow* components, such as Green procedures, can be instantiated.

All languages support a common set of built-in types such as integers and booleans equipped with the usual operators, together with user-defined data, function and type expressions. Further details appear in [11].

2.2 Rainbow Example: a Simple Processor

We introduce Rainbow by presenting a simple example, which illustrates the different styles available for describing designs. We show how the styles can be combined so that each part of a design can be described in the most effective way. The example used[1] is a simple microprocessor unit (*SMPU*), similar to the Move Machine[12]. Figure 2 shows the basic block structure of SMPU described in Visual Green. The design has the following components:

`decode()`: sequences the fetch/decode/execute phases of each instruction cycle, decodes the newly fetched instruction and then issues the appropriate controls to `reg_bank()` and `alu_block()`, via the `regIFC` and `aluIFC` control channels, respectively.

`reg_bank()`: takes its input either from the memory, or from the ALU during write-back; it can also take an immediate value supplied via `regIFC`. It has two output channels, but only writes to these sequentially. The program counter (`PC`) is considered to be just another register (register 0).

[1] The design has been constructed for illustrating Rainbow, and is therefore not necessarily particularly efficient.

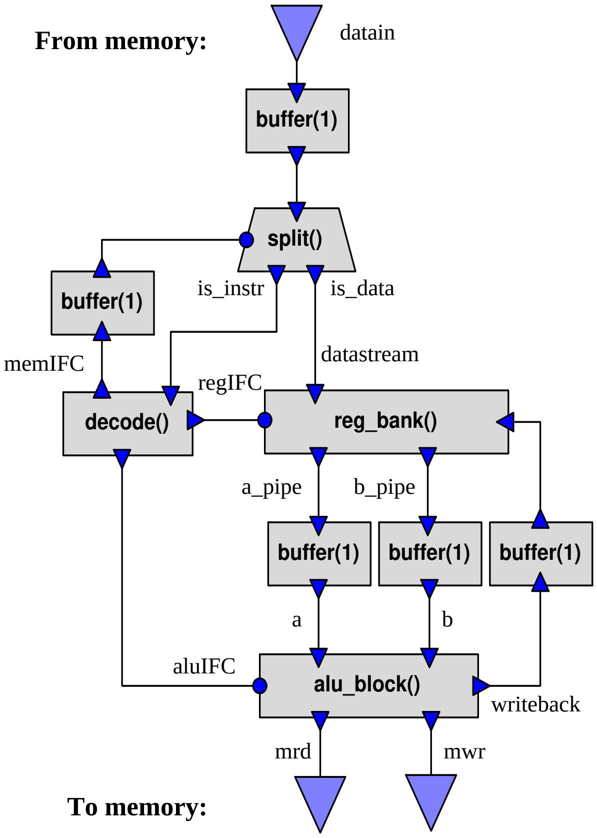

Figure 2: Visual Green SMPU

`alu_block()`: contains both the arithmetic unit and data/address output control. One or both of the data inputs from `reg_bank()` are read and the results are output either to memory for read/write or to `reg_bank()` for write-back. The block also increments PC.

Briefly, the operation of SMPU is controlled by the sequence of controls supplied by `decode()`. First, a new instruction is fetched from the memory address given by PC: `decode()` instructs `reg_bank()` to read PC, which then passes the address through `alu_block()` to the memory-read output channel `mrd`, as well as being incremented and stored back in `reg_bank()`. When the desired instruction arrives back at `datain`, `decode()` decodes and executes the instruction by issuing controls to `reg_bank()` and `alu_block()` determined by the instruction type and argument values. When two register values are required, then these must be read sequentially. The operations within the instruction cycle are not pipelined — introducing this may be considered as part of a future design step.

The Visual Green SMPU design in Figure 2 is interpreted as a static dataflow diagram with only explicit buffering, indicated by the single place buffer nodes `buffer(1)` — there is no implicit buffering in the wires, although nodes may themselves contain state. The nodes use synchronous micropipeline communication to pass (bundled) data values along the wires.

We first describe in more detail the behaviour of the basic Green elements which appear in Figure 2 — the user-defined nodes are then described below. The four buffers in Figure 2 are all single-place and uninitialised. An empty buffer is able to consume a value presented on its input, thus decoupling the input and output, so that the node supplying the input can proceed with further operations. At the next time-step the value is made available on the output — the buffer waits until this is consumed before being able to read from its input again.

```
table alu_block (input  aluIFC: aluctrl;
                 input  a,b: val;
                 output mrd: memread;
                 output mwr: memwrite;
                 output writeback: val) = {
    FETCHINC, a, - => a,   -,     a+1       //a is pc value
 or FROMMEM,  a, b => a+b, -,     -         //a is base addr, b is offset
 or TOMEM,    a, b => -,   (a,b), -         //a is addr, b is value
 // --- Binary ALU operations ---
 or ADD,      a, b => -,   -,     a+b
 or SUB,      a, b => -,   -,     a-b
 or AND,      a, b => -,   -,     a/\b
 or CMP,      a, b => -,   -,     a>b
 // --- Unary ALU operations ---
 or MOV,      a, - => -,   -,     a
 or NOT,      a, - => -,   -,     not a
}
end
```

Figure 3: Table for `alu_block()`

The `split()` node at the top of Figure 2 writes the memory input at `datain` either to `decode()`, when it receives a value `is_instr` on its control input `memIFC`, or to `reg_bank()`, when it receives value `is_data`. Note that `split()` does not produce an output until values are available on both its control and input channels. A flow control node such as `split()` is stateless, so that it only releases its inputs when all of the values on its output have been consumed. Also notice that the input/output channels on a node such as `split()` can progress independently, when required — for example, the `split()` node consumes both its inputs at the same time, but only produces a value on one of its outputs.

Finally the input and output ports `datain`, `mrd`, `mwr` are always able to source or sink a new value.

The design style shown in Figure 2 is typical of the block-level descriptions often seen when describing, for example, the behaviour of processors. The internal structure of the top-level nodes can then be described using any Rainbow sub-language procedure; here, we use a mixture of Textual Green and Yellow procedures to describe them, the latter especially offering a more behavioural descriptive style. Design components in the different sub-languages are able to communicate via procedure calls — a Yellow procedure can be invoked in a Green network by making a call to it, in parallel with other Rainbow fragments, and vice versa. Communication between components is made via channel arguments, using the synchronous micropipeline model.

2.3 Tabular Description: `alu_block()`

Green provides the `table` construct for describing in a natural way the general (conditional) dataflow behaviour of controllers and decoders. Figure 3 shows the table declaration for `alu_block()`, consisting of input/output channel arguments, followed by a list of table rows. Each row contains an input pattern (to the left of the '`=>`'), to be matched against values on the input streams. When there is a match, then the values for the output streams are generated, determined by the expression list to the right of '`=>`'. Each pattern contains literals such as `FETCHINC`, variables such as `a,b` that bind to the value on the same-name input stream, or '`-`' which means that a value is not required on that particular input channel — if a value is present on a channel whose entry contains '-' in a successful pattern-match, then that input will simply not be consumed. The outputs in each row consist of expressions, such as 'a+1' which utilise the variables bound in that row. Alternatively, '-' can again be used to mean that no value is to be written to an output stream. For example, the first row in table alu_block() is matched when the value FETCHINC is on input channel aluIFC, and some value is present on input channel a; no value is required on input b. In this case, the value a is output to channel mrd and the incremented PC value 'a+1' is output to channel writeback. The result of this is that a new instruction is fetched from memory address 'a' given by the current PC value, and the new incremented PC value a+1 is output, ready for writing back to the register bank. The functionality of the ALU can easily extended by adding further rows to alu_block(). Note that '//' marks the start of a comment.

Tables are restricted so that at most one row pattern is matched. They can be used to implement many of the other (stateless) basic constructs in Green, including functions or the split() node introduced above:

```
table my_split (input x: integer; input c: bool;
                output a, b: integer) = {
     x, TRUE  => x, -
  or x, FALSE => -, x
  } end
```

They are ideal for describing general dataflow behaviour, and it is envisaged that they will be used for fast prototyping, to be replaced at a later stage by more structural designs. They can also be used for describing certain parts of a processor design, such as instruction decoders, ready for straightforward translation into hardware, using PLAs for example.

2.4 Yellow Description: decode()

The decode() node is used to sequence the fetch-decode-execute-store instruction cycle within the SMPU. The sequential control-flow behaviour required by decode() is best described using Yellow[2]. The Yellow process is embedded in the top-level Green SMPU network via a call to the definition. Note that the node is designed to operate forever, via the infinite loop construct 'loop TRUE -> ··· end_loop'.

A new instruction register i, of type instr is declared by the statement 'reg i: instr end'. The first section of code issues controls for fetching the next instruction from memory and for directing it back into decode() — deadlock is prevented by buffering the memIFC control for the SMPU split():

```
par
       call memIFC!is_instr
   || { call regIFC!(ARG_A,0) ; call regIFC!(WB,0) }
   ||   call aluIFC!FETCHINC
   ||   accept instruction?ins do i:=ins end
end_par;
```

The par ··· end_par construct executes the component expressions, separated by '||', in parallel. The first call statement writes the value is_instr to output channel memIFC, so that the next value input from memory (i.e. the new instruction) will be directed to decode(). The next line contains two sequenced call statements to channel regIFC, separated by ';'. The reg_bank() is instructed by the first call to read register 0 to the a_pipe in the SMPU,

[2]The complete definition of SMPU, including the decode() and reg_bank() procedures, appears in the appendix.

and by the second to write back the incremented PC value. The final `call` statement directs `alu_block()` to fetch an instruction, by performing a memory read, and to increment the PC. Finally, the `accept` statement will read the new instruction when it returns from memory, and will then execute its body '`i:=ins`', which stores the new instruction in register `i`.

When a new instruction has been stored, then the decode section is entered. Here, the single `choice` statement outputs appropriate controls and values to `reg_bank()` and `alu_block()`, determined by the type of instruction. For example, when a load instruction of type `oplod` is received, the first branch of the `choice` is executed:

```
i in oplod ->
  par
      { call regIFC!(ARG_A, i.base) ; call regIFC!(ARG_B, i.off) ;
          call regIFC!(LODM, i.dst) }
   || call { aluIFC!FROMMEM, memIFC!is_data }
  end_par
```

Again the `par` statement allows parallel execution of the calls, while the sequence operator '`;`' ensures that the controls to `reg_bank()` are given in an appropriate order. Note that fields of record values are accessed using field names, such as '`i.base`', which yields the `base` field of instruction '`i`'.

2.5 Green Sub-Network: `reg_bank()`

Apart from describing Green networks schematically, a textual version of the language can be used. This offers a more familiar programming language-style of description, and will be useful for describing large Green networks, especially those involving complex data types, when a schematic representation may become unreadable. The definition of `reg_bank()` begins with local type declarations for enumerated types. Next, the `RegisterControl` table generates the internal controls required for each register bank operation. The `Registers` and internal channels are then declared:

```
memory Registers : (val ^ regid) end

chan ImmVal:val end             chan Src:srcsel end
chan RegId:regid end            chan RegAction:rdwr end
chan Dst:dstsel end             chan RBinval:val end
```

The `memory` statement declares an array storing values of type `val`, with addresses ranging over type `regid`. It is accessed via channels and takes a read channel input of type `regid`, a channel output of type `regid`, and a write channel input tuple of type `(val, regid)`.

The Green register bank consists of a series of sub-networks, separated by '`$`' and operating independently. The first statement simply connects input and output channels to `RegisterControl()`:

```
regIFC >> RegisterControl() >> {Src || ImmVal || RegId || RegAction || Dst}$
```

The pipe operator '`>>`' makes the (micropipeline) connection. The parallel operators '`||`' mean that the components (in this case channels) can progress independently. The second statement:

```
using Src
merge
    imm: ImmVal>>
 || mem: mem_stream>>
 || alu: alu_stream>>
end_merge >> RBinval$
```

is a textual form of the merge operator used for stream multiplexing. This uses the control channel Src to select one of the three input streams; for example, if Src = imm then the value on channel ImmVal is directed to RBinval (any values on the other two streams are blocked). A merge node can only output a value when the control input and the selected data input values are available; it will then not consume these inputs until the output has been consumed.

The final statement in reg_bank() first constructs the input value to the memory Registers, using a textual form of split, and then directs any output from Registers using a second split. The first split receives a data input from RegId and directs this according to the value on channel RegAction;

```
RegId >>
using RegAction
split
     read: { _ }
 || write: { _ || RBinval }
end_split
>> Registers() >>
```

If RegAction = read then RegId is passed through to the read input port of Registers unchanged, denoted by the identity operator '{ _ }'; this defines the register whose value is to be output from Registers. If RegAction = write, then RegId is tupled with the input data value on RBinval and the pair is then passed on to the write channel input of Registers. The two (independent) streams output by split() are then piped into the register bank core Registers, which will perform a read or write on the register given by RegId depending upon whether it receives a value on its first input stream or on its second.

Any value read from Registers is then directed by the following split() to one of the output channels (a_pipe or b_pipe) of reg_bank(), as determined by the value on Dst:

```
using Dst
split
    a: >>a_pipe
 || b: >>b_pipe
end_split$
```

The complete description for SMPU given in the appendix also contains the necessary type declarations, using integer subrange, enumerated, tuple, record and (discriminated) union types — these have their usual meaning. A Textual Green version of Figure 2 is also included here, so that the two alternative presentations of Green can be compared.

3 Formal Semantics of Rainbow: an Overview of APA

APA is a process algebra designed for representing the behaviour of asynchronous systems described in Rainbow. It is similar to standard process algebras, such as CCS[13], CSP[10] or LOTOS[14], using some familiar process operators and with its semantics defined operationally using SOS-style transition rules. However, the composition operators have been designed to support bundled-data micropipeline communication — the communication primitives involved use an Ada-like rendezvous, and their semantics resembles the Ada semantics presented in [6]. APA components operate at the level of abstraction corresponding to Rainbow; APA therefore supports value-passing and has richly-structured transition labels, similar to those used in our previous work on ELLA and EPA[4]. These features lead to a compact semantic representation of Rainbow designs, providing the basis for simulation formal analysis. The development of APA extends our work of developing application-specific process algebraic semantic representations which naturally express the semantics of the source language at the most suitable level. This reduces the amount of encoding which would be necessary if already existing general process algebras were used, helping to clarify rather than obscure the intended meaning of Rainbow.

As with EPA, we make a distinction between the process terms, which model hardware system connectivity, and the embedded action algebra for modelling data. The actions are input/output maps, from channels to value expressions. There is also a 'zero' action $\boxtimes$ for representing 'chaos' – any trace containing a zero action will be considered to be improper and will be ignored during subsequent analysis. This enables us to reduce the use of side conditions when defining the semantics for process operators, in turn enabling to the effect of process operators to be lifted to the action level, and defined by total action functions. Any improper action combination simply returns $\boxtimes$.

The constructs available in APA are essentially a subset of Yellow and Textual Green. They include parallel, sequence, choice, loop and process call, which behave in a similar way to the corresponding operators described in other standard process algebras such as CSP or CCS. For example, the APA parallel operator combines the transition labels of the two component processes by matching input and output channels, as in CCS. However, its operation is slightly more complicated because of the map structure of the transition labels; if the values carried by same-name channels do not match then a chaos transition labelled by $\boxtimes$ results.

Communication in APA is provided by an extended `accept` construct that takes a (possibly empty) list of output actions as an additional argument — it is the behaviour of this element that enables both Yellow and Green communication to be modelled uniformly. Instead of using explicit control channels, micropipeline communication is modelled directly by introducing 'postponed' input/output actions. When all of the inputs to the construct are available, then the body can begin execution and the outputs are generated. Consumption of the inputs is then postponed until the body has completed and all of the outputs have been consumed.

APA is used to provide a single uniform semantics for all of the *Rainbow* sub-languages, thereby supporting interworking — suitable translations from *Rainbow* into APA are defined. As may be expected, much of the translation is straightforward, since many Rainbow components have exact counterparts in APA. For the remaining elements, the translations to APA are generally based on standard encodings. For example, the Green pipeline operator '`>>`' translates into a parallel composition with channel renaming and hiding; several channels may be piped independently between two processes. Yellow registers utilise an evaluation environment, similar to that used in [6]. Channel declarations and scoping rules simply translate into channel hiding in APA. The Green `table` construct can be viewed essentially as a definition of a transition set — each row represents one or more possible transitions which the process can make, after substitution of actual values for input variables, and subsequent evaluation of expressions on the outputs.

Further details of APA and its role in providing the semantics for Rainbow appear in [11].

4 Summary

We have introduced the prototype Rainbow framework that supports mixed-view descriptions of asynchronous micropipeline designs. A uniform semantics for all Rainbow sub-languages is defined via APA, providing a straightforward definition of the behaviour of a design described using combinations of sub-language components. A simple processor design has been used to illustrate the natural description styles available in Rainbow, allowing a designer to describe design modules in a variety of ways, using the most appropriate method for each module.

Rainbow and APA will provide the basis for simulation and analysis tools currently under development.

5 References

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

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

[3] AMULET. The AMULET Group Publications and Bibliography. Available via World-Wide Web, URL: `http://www.cs.man.ac.uk/amulet/publications/index.html`.

[4] Formal Verification Support for ELLA, IED/SERC Project 4/1/1357. Reports are available on the World-Wide Web, URL: `http://www.cs.man.ac.uk/fmethods/projects/ella-project.html`.

[5] Manfred Broy, Frank Dederich, Claus Dendorfer, and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-19202, Inistitut fü Informatik, Technische Universität München, 1992.

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

[7] I. Sommerville and R. Morrison. *Software Development with Ada*. Addison-Wesley, 1987.

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

[9] J. Herath, Y. Yamaguchi, K. Toda, R. Mattingley, N. Saito, and T. Yuba. Comparison of Dataflow Computing Models. In John A. Sharp, editor, *Data Flow Computing: Theory and Practice*. Ablex, 1992.

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

[11] H. Barringer, D.K. Fellows, G.D. Gough, and A. Williams. Rainbow: Towards a Hardware Description Framework for Micropipeline Designs. Draft technical report, University of Manchester, 1996.

[12] J.S.H. van Gageldonk. The Asynchronous Move Machine: Verification using CCS. Master's thesis, Eindhoven University of Technology, 1994.

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

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

A Rainbow Description of SMPU Design

```
// SMPU - a Simple Microprocessor Unit
// --------------------------------------------------------------------

// --------------------------------------------------------------------
// Constants and Types:

// Some constants:
constant MAX_ADDRESS = 127 end                constant MAX_VALUE   = 127 end
constant MAX_REGISTERS   = 15  end

// Data types:
type address = [0..MAX_ADDRESS] end                      // integer subrange
type data    = [0..MAX_VALUE] end
type val     = union(address, data) end                  // union

// ALU opcodes:
type aluarith1 = enum(MOV, NOT) end                      // enumeration
type aluarith2 = enum(ADD, SUB, AND, CMP) end
```

```
type alumem    = enum(FETCHINC, FROMMEM, TOMEM) end
type aluctrl = union(aluarith1, aluarith2, alumem) end

// Instruction formats:
type oplod  = (base:regid, off:regid, dst:regid) end       // record
type opsto  = (addr:regid, arg:regid) end
type opldi  = (dest:regid, immval: val) end
type opalu1 = (opcode:aluarith1, src:regid, dst:regid) end
type opalu2 = (opcode:aluarith2, src1: regid, src2: regid, src3: regid) end
type instr  = union(oplod, opsto, opldi, opalu1, opalu2) end

// Memory types:
type memctrl  = enum(is_data, is_instr) end        type memread  = val end
type memwrite = (val, val) end                     type memin    = union(instr,val) end


// Register types:
type regid      = [0..MAX_REGISTERS] end
type unregcode  = enum(ARG_A, ARG_B, LODM, WB) end
type unregop    = (unregcode, val) end
type binregop   = (val, val) end
type regctrl    = union(binregop, unregop) end

// ----------------------------------------------------------------------

table alu_block (input  aluIFC: aluctrl;
                 input  a,b: val;
                 output mrd: memread;
                 output mwr: memwrite;
                 output writeback: val) = {
    FETCHINC, a, - => a,   -,     a+1       //a is pc value
 or FROMMEM,  a, b => a+b, -,     -         //a is base addr, b is offset
 or TOMEM,    a, b => -,   (a,b), -         //a is addr, b is value
 // --- Binary ALU operations ---
 or ADD,      a, b => -,   -,     a+b
 or SUB,      a, b => -,   -,     a-b
 or AND,      a, b => -,   -,     a/\b
 or CMP,      a, b => -,   -,     a>b
 // --- Unary ALU operations ---,
 or MOV,      a, - => -,   -,     a
 or NOT,      a, - => -,   -,     not a
}
end

// ----------------------------------------------------------------------

yellow decode(input instruction: instr;
              output memIFC: memctrl;     output regIFC: regctrl;
              output aluIFC: aluctrl) = {
  loop TRUE -> {
    reg i: instr end
    //---------------------------------------------------------------------
    // Fetch (pc is at register 0):
    par
           call memIFC!is_instr
      || { call regIFC!(ARG_A,0) ; call regIFC!(WB,0) }
      ||   call aluIFC!FETCHINC
      ||   accept instruction?ins do i:=ins end
    end_par;
    //---------------------------------------------------------------------
    // Execute
    choice
```

```
      i in oplod ->
        par
            { call regIFC!(ARG_A, i.base) ; call regIFC!(ARG_B, i.off) ;
                call regIFC!(LODM, i.dst) }
         || call { aluIFC!FROMMEM, memIFC!is_data }
        end_par
     or
      i in opsto ->
        par
            { call regIFC!(ARG_A, i.addr); call regIFC!(ARG_B, i.arg) }
         || call aluIFC!TOMEM
        end_par
     or
      i in opldi ->
        call regIFC!(LDIM, i.dst, i.immval)
     or
      i in opalu1 ->
        par
            { call regIFC!(ARG_A, i.src) ; call regIFC!(WB,i.dst) }
         || call aluIFC!i.opcode
        end_par
     or
      i in opalu2 ->
        par
            { call regIFC!(ARG_A, i.src1) ; call regIFC!(ARG_B, i.src2);
                call regIFC!(WB, i.dst) }
         ||   call aluIFC!i.opcode
        end_par
    end_choice
  } end_loop
}
end

// ------------------------------------------------------------------------
green reg_bank (input regIFC: regctrl;   input mem_stream: val;
               input alu_stream: val;
               output a_pipe, b_pipe: val) = {

  type srcsel = enum(imm, mem, alu) end     type rdwr = enum(read, write)   end
  type dstsel = enum(a, b)          end

  table RegisterControl (input regIFC: regctrl;
                         output src: srcsel;    output immval: val;
                         output id: regid;      output rw: rdwr;
                         output dest: dstsel) = {
   var r, v: val end
      (ARG_A,r) => -,   -, r, read,  a
   or (ARG_B,r) => -,   -, r, read,  b
   or (LODM, r) => mem, -, r, write, -
   or (WB,   r) => alu, -, r, write, -
   or (r,    v) => imm, v, r, write, -
  }
  end

  memory Registers : (val ^ regid) end

  chan ImmVal:val end         chan Src:srcsel end
  chan RegId:regid end        chan RegAction:rdwr end
  chan Dst:dstsel end         chan RBinval:val end

  // Create the control values:
   regIFC >> RegisterControl() >> {Src || ImmVal || RegId || RegAction || Dst}$
```

```
  // Generate any data input to the register bank:
  using Src
  merge
      imm: ImmVal>>
   || mem: mem_stream>>
   || alu: alu_stream>>
  end_merge >> RBinval$

  // Generate the input to the register bank, and  direct any output:
  RegId >>
  using RegAction
  split
       read: { _ }
   || write: { _ || RBinval }
  end_split
  >> Registers() >>
  using Dst
  split
      a: >>a_pipe
   || b: >>b_pipe
  end_split$
} end

// ----------------------------------------------------------------------

green SMPU(input DataIn: memin;
          output mrd: memread;    output mwr: memwrite) = {

  chan memIFC, regIFC, aluIFC, writeback, instrStream, dataStream end

  DataIn >> buffer(1) >>
  using memIFC >> buffer(1)
  split
      is_instr: >> decode() >> {memIFC || regIFC || aluIFC}
   || is_data:  >> dataStream
  end_split $
  {
      aluIFC
   ||
      { regIFC || dataStream || writeback >> buffer(1) } >> reg_bank() >>
         {buffer(1) || buffer(1)}
  } >>
    alu_block() >> {mrd || mwr || writeback} $
} end

// ----------------------------------------------------------------------
// Top-level call to SMPU:
chan datain, mrd, mwr: val end
SMPU(datain, mrd,mwr)
```