Design and Verification of the Rollback Chip using HOP: A Case Study of Formal Methods Applied to Hardware Design [1]

GANESH GOPALAKRISHNAN[2]
RICHARD FUJIMOTO[3]

UUCS-91-015

Ganesh Gopalakrishnan
Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

Richard Fujimoto
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

September 5, 1991

Abstract

The use of formal methods in hardware design improves the quality of designs in many ways: it promotes better understanding of the design; it permits systematic design refinement through the discovery of invariants; and it allows design verification (informal or formal). In this paper we illustrate the use of formal methods in the design of a custom hardware system called the 'Rollback Chip' (RBC), conducted using a simple hardware design specification language called 'HOP'. An informal description of the requirements of the RBC is first given, followed by a behavioral description of RBC stating its desired behavior. The behavioral description is refined into progressively more efficient designs, terminating in a structural description. Key refinement steps are based on system invariants that are discovered during the design, and proved correct during design verification. The first step in design verification is to apply a program called PARCOMP to derive a behavioral description from the structural description of the RBC. The derived behavior is then compared against the desired behavior using equational verification techniques. This work demonstrates that formal methods can be fruitfully applied to a non-trivial hardware design. It also illustrates the particular advantages of our approach based on HOP and PARCOMP. Last, but not the least, it formally verifies the RBC mechanism itself.

[1]Keywords: Custom Hardware Design, Formal Specification, Verification, Process Composition, Parallel Discrete Event Simulation, Time Warp

[2]Supported in part by NSF Award MIP-8902558

[3]Supported in part by NSF Awards DCR-850-4826 and CCR-8902362

1 Introduction

As custom architectures become more and more complex, the time required to functionally validate them is becoming prohibitive. Today, most custom architectures are validated through functional simulation. Even very small systems cannot be exhaustively tested. Although it may be possible to judiciously pick (based on experience) a set of simulation vectors which can validate *small* systems to a high level of confidence, this is extremely difficult, if not impossible to do, for large custom hardware designs.

In order to validate systems systematically with a measurable degree of confidence, formal verification has been widely advocated. An ideal design approach would consist of top-down design starting from a formal specification, where each design refinement step has been pre-validated through formal means. In practice, designs are seldom carried out strictly top-down; computer architects rely heavily on their experience and complete large portions of their designs in a seemingly unstructured way. In this approach also, it is possible for the architect to engage in his/her "leaps

of inspiration", but very soon thereafter capture the results of such "creative thinking" in a formal notation. A drawback of this approach is that it requires the designers to be experts in both architecture and formal verification methods. Another extreme approach would be to let the architect to complete the design and then hand-over the verification task to a team well versed in formal verification. A disadvantage of this task is that the formal verification team often finds it very difficult to unearth, and formally specify most of the semantics preserving transformations done by the architect. Also, the proof cannot be well-structured and made readable—an important requirement if the proof is to be believed, and if something is to be learned from the proof.

A good compromise is to employ team effort in which architects work closely with the formal specification writers. In this approach the specification and the design evolve together, and are maintained to ensure consistency. By the time the design is finished, the specification writer is well aware of the architectural design steps, and the architect is well aware of the specification techniques used. The final design representation (in whatever form it is—e.g. HDL, netlist, VLSI mask) can then be verified against an abstract specification of the system.

We have completed such a design exercise on a real hardware design called the Roll Back Chip [1, 2]. The Roll Back Chip (RBC) is a custom architecture that helps speed up the state-saving and roll-back portions of parallel discrete event simulations using Time Warp [3]. The design of the RBC evolved side by side with a specification for it. Specifications were written in our hardware description language HOP. Top-down design was followed to some extent, but many of the crucial refinement steps were simply based on the experience of one of the authors as an architect, and captured after the fact by the other author whose primary role has been that of formal specification. The final version of the RBC that was implemented (see [2]) is essentially the final design specification in HOP that is verified in this paper.

Our Design Methodology

Our design methodology is one of *performance directed design refinement with validation based on formal verification.* Large application specific hardware systems have, in addition to their functional correctness requirements, many demanding performance requirements. Unless both performance and functional correctness can be guaranteed, the final design is considered to be useless. Performance is, unfortunately, an empirical quantity that is usually measured in practice through simulation studies. Satisfactory analytical models of performance are often difficult to obtain.

In a typical design cycle as employed today, an architect begins with an informal (e.g. English + block diagrams) description of the system to be designed. He/she then picks one promising architecture and describes this architecture in a typical simulation (e.g. Simscript) or programming (e.g. C) language, and runs simulations. If the results look promising, and if the simulation reveals no functional errors, the design is further refined. Otherwise, the architect backs up one or more levels in the refinement process and examines another architecture.

Typical optimization techniques employed by architects are:

- lazy updates of data structures,
- pipelining, and
- making more frequently operations cheaper at the expense of less frequently used ones.

In this process, it is easy to come up with many performance optimization techniques that appear correct, and may even be found correct by a large number (e.g. millions) of simulation vectors. However, it is precisely such optimization techniques that introduce many subtle bugs that are often never revealed, or are revealed at awkward moments during the system's operation! For custom hardware systems deployed in the real-world—especially safety-critical—applications, such mistakes in validation can be extremely costly, and cannot be tolerated.

Although we are not offering a complete solution to the above problem, in this paper we are proposing a design methodology wherein each design refinement is formally described, so that the correctness of the refined behavioral description can be established formally against the highest level behavioral description. The initial design specification describes an abstract data type that supports several operations. At the highest level, only the functional correctness of these operations (and not performance) is of importance. Each refinement step adds detail to the state representation used, by augmenting the underlying data structures with new ones, or even replacing the data structures completely. The correctness of these transformations usually relies on *system invariants* (examples: "no duplicates in an associative memory"; "a state changing operation need not finish until some other operation that utilizes the state as well as produces an external output is triggered"). Although computer architects are typically well-aware of these kinds of invariants, in current design approaches, they seldom write them down because they consider the digression to be wasted effort, that benefits no one.

In our approach, such invariants are noted and later used in the formal proof of the system. The RBC system has been refined through many levels of refinement, and each refinement decision, taken based on *performance considerations*, has been shown to be correct during our proof. Demonstration of this methodology in a specific hardware design is perhaps the single most important contribution of our paper.

Our design methodology is outlined in figure 1. We express the initial specification of the system as a data type [4] supporting a collection of operations. This data type is then refined into a concrete representation, where the representation is chosen based on performance considerations. Simulation experiments are run to gain some confidence in the functional correctness of the refinement, and also to check that adequate performance can be obtained. This process is repeated through many levels, until the final architecture is clearly identifiable. After each refinement step, the design (or certain aspects—e.g. invariants) of it are formally described in our HDL, 'HOP'.

Once the final structural description is obtained, the actual hardware system can be constructed. We then apply an algorithm called PARCOMP to the structural specification. PARCOMP, which is part of the HOP system, infers a behavioral specification corresponding to the structural specification. In other words, PARCOMP simulates all possible interactions amongst the submodules, and captures these interactions in the form of a behavioral description. This behavioral description is indistinguishable from the external (i.e. behavioral) view of the structural description from which it was derived. The inferred behavioral description produced by PARCOMP is then compared against the initial specification using equational verification techniques. The proof of correctness can in principle, be automated using systems such as the Boyer-Moore prover [5], HOL prover [6], or the CLIO prover [7]. At present, however, only a manual proof has been completed.

Related Work

Formal verification of software and hardware systems has been proposed as an alternative to simulation and testing as early as the 60s. Much work has concentrated on software system verification [8, 9]. One of the early proposals for hardware verification came in the early 70s and has since been followed by a large body of work [10]. There has since been an extensive growth both in variety of techniques, examples, and verification tools [11, 12].

In hardware verification, much work has centered around showing the *functional correctness* of proposed implementations of hardware systems against their behavioral specifications. Though in many cases, verification of detailed timing behavior [13] as well as circuit behavior [14] has been included, these efforts have been restricted to small designs.

In this paper, we address the task of showing functional correctness of designs. More specifically, we address the task of showing that the *input output mappings* of the actual implementation agree with that of the specification. The specification usually doesn't provide any timing information

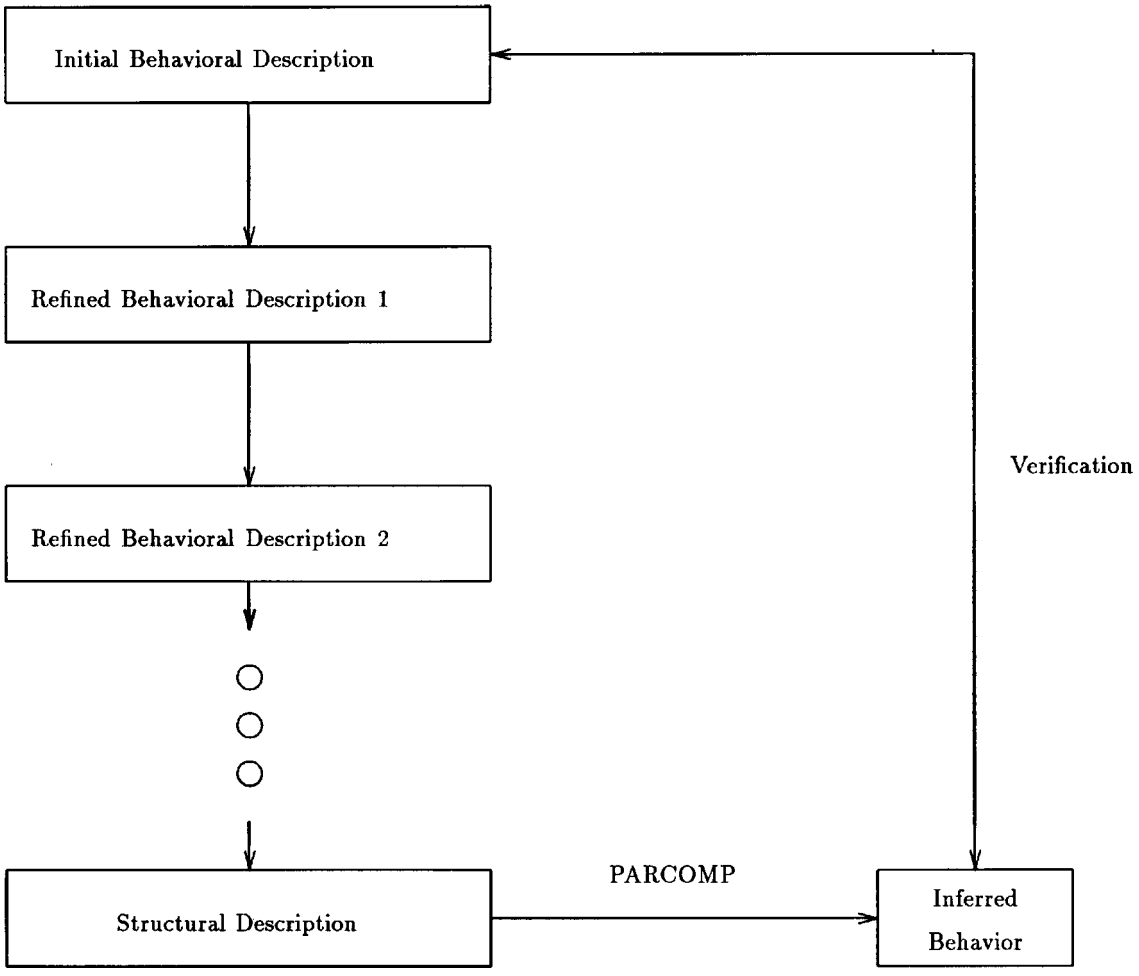


Figure 1: Our Design Methodology

about the design. The implementation does usually provide both functional and timing (at a cycle-level) information. The detailed cycle-level timing information contained in the implementation is simplified to an extent that only the *causal orderings* are retained; details of the exact number of cycles, etc., are ignored.

Among work that fall into this category, most examples deal with the verification of simple microprocessors [15, 16]. The microprocessors that were verified were not built with the design objective of high performance. A notable exception in this area is the work of [7] which considers the verification of a pipelined microprocessor. In that work, however, the pipelined design was not obtained through a process of top-down refinement that was geared towards obtaining a high-performance design.

The work presented in this paper differs from other works in the area of functional verification in the following important regards:

- in hardware verification, most examples chosen for illustration have been CPUs; we verify a complex special-purpose memory subsystem, for a change;
- we refine the initial behavioral specification top-down through *many* levels, each time refining the internal state representation and re-defining the operations of the module to more efficient versions; **the final version bears little resemblance to the initial specification**; (Each of these refinements can be separately verified, using techniques presented in this paper. However, this is beyond the scope of the paper.)
- our refinement steps lead to a complex implementation (structural specification). From this, we automatically infer a behavioral description and verify this *derived behavior* against a much simpler and believable specification of the *desired behavior.*

Less formal (rigorous) approaches than formal verification are commonly used in the validation of real-world hardware systems. In [17], the 'verification' (through random, and non-exhaustive tests) of a very complex cache controller has been discussed. The highest level specification of the protocol was written only in English. The cache controller was exercised using test-stubs that randomly generated a large number of possible multiprocessor interaction sequences. Most tests incorporated both test stimuli and expected responses, thus allowing these tests to run without human intervention (except upon noticing an error) for several weeks. Although this approach offers important benefits, the lack of a formal high level specification (beyond an English description) makes it **impossible to even define what verification means**. Also, running random tests for several *weeks* does not cover even a fraction of the state space of large architectures; their state space is so large that to fully cover all possible tests, it will take several *millennia.*

In our work, we specify hardware systems essentially as abstract data types, as in [4]. It is well known that data type axioms can be very easily obtained from such specifications, and used both for formal verification and test vector generation. Conducting both formal verification and test vector generation based on the same formal specification has the dual advantages of exhaustively proving the design for all inputs and state (achieved by verification), and of validating the assumptions about the real-world employed during verification (achieved by testing). This dual approach has been advocated in [18].

Organization

In section 2, we present an informal description of the RBC. We then give the initial formal description RM1 (section 3). We discuss the first optimization in section 4, involving the use of a data structure called the written-bits array. We then discuss a non-trivial optimization in section 5, involving the use of a second data structure called the roll back history. After presenting

the structural specification corresponding to the actual design of the RBC (section 6), we go on to apply PARCOMP to it, to obtain the inferred behavioral description, RM2 (section 7). In section 8, we prove that RM2 is functionally equivalent to RM1. Key results are discussed in our conclusions. An Appendix containing deferred details is included.

2 Informal Description of the RBC

A critical problem that parallel computers must address is synchronization between concurrent computations. Recently, optimistic methods of synchronization have been developed that are based on *rollback*. In optimistic synchronization, one detects synchronization errors at runtime, and invokes a rollback mechanism to recover. This is in contrast to more traditional *conservative* synchronization techniques that utilize blocking to *avoid* synchronization errors.

The Time Warp mechanism is perhaps the most well known optimistic protocol [3]. Some successes have been reported in using Time Warp to parallelize discrete event simulation problems, e.g., see [19, 20]. In Time Warp, the parallel computation consists of some number of processes that communicate by exchanging timestamped messages. The parallel computer is assumed to be a collection of processors, each with local memory, that communicate by exchanging messages. Time Warp provides an elegant, distributed mechanism for rolling back erroneous computations that may have spread across a network of processors.

Because optimistic synchronization methods rely on rollback, one must periodically save the state of each process. It has been observed that this is a serious overhead for processes containing a significant amount of state [19]. The rollback chip is a component that has been developed which provides hardware support for state saving.

The following capabilities are required to support state saving in Time Warp:

- It must be possible to take a snapshot of a process's data segment at any instant of time. A *mark* operation is issued to mark the current state of the process as one that may later have to be restored via a rollback.
- If rollback becomes necessary, the state of the process will have to be restored to a previous snapshot. The operation *rollback* deletes the most recent snapshot. In practice, several rollback operations must be issued to roll back to a specific frame.
- It must be possible to read and write the data segment in essentially the same way as conventional memory. $read(a)$ reads the contents of memory location a, and $write(a, d)$ writes data d into location a. In the absence of rollback, read and write operations have the same semantics as those operations in conventional memory systems. When a rollback occurs, writes performed by computations that were rolled back must be erased. Here, we will assume all read and write operations access a single word of data.
- Storage utilized by old snapshots must be reclaimed. The operation *advance* discards the oldest snapshot. The mechanism for determining when it is safe to garbage collect snapshots is beyond the scope of the present discussion, but is discussed in [3].
- Finally the *reset* operation initializes the system.

To date, several designs of the rollback chip have been studied. Extensive simulation studies have been completed to provide an initial check for correct behavior, and to evaluate its performance [1]. A commercial version of the RBC is under development, and an initial prototype using off-the-shelf components has been realized [2]. VLSI layouts for portions of the RBC have also been developed [21]. More recently, the RBC has been specified in the language HOP. This specification was used both to verify that the behavior of the proposed mechanisms are correct, but also to aid development of new, provably correct improvements to the proposed mechanisms.

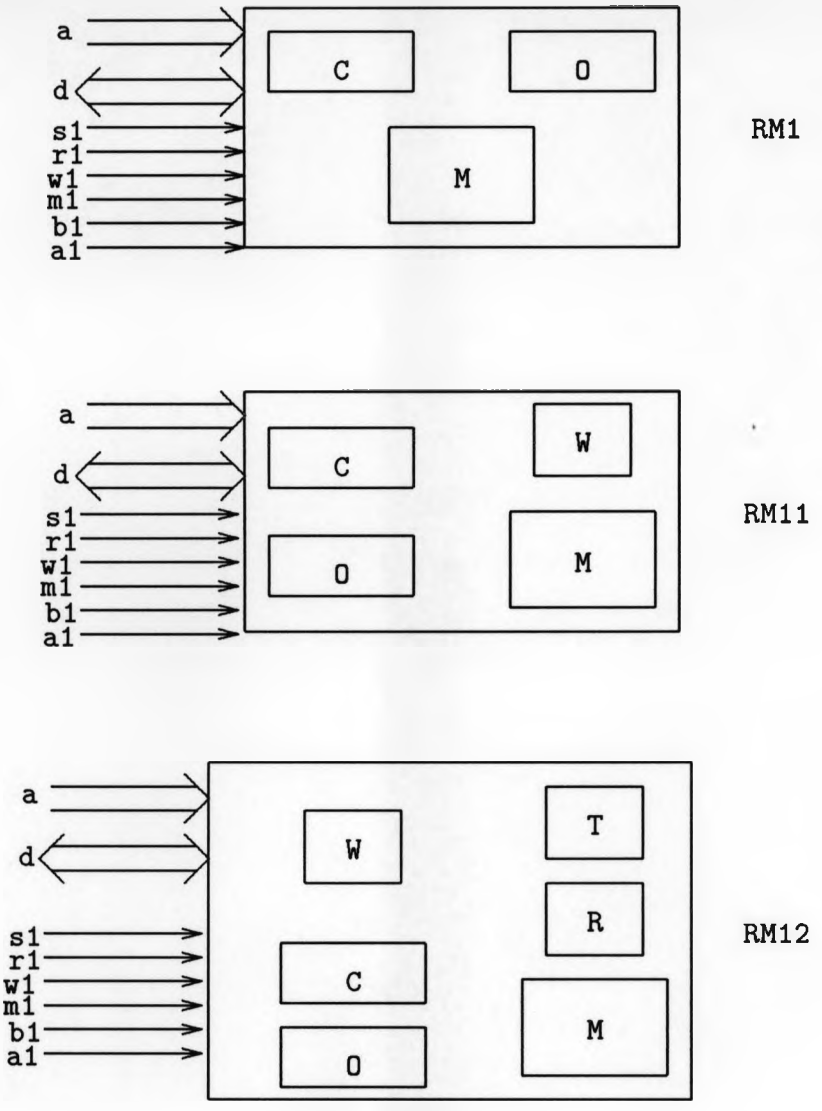


Figure 2: Initial Specification, RM1, and Later Refinements

3 Initial Specification, RM1

In this section, we present the specification of the desired behavior of the rollback chip. This system consists of the rollback chip hardware and the associated main memory (together known as the "rollback chip system" or the RBC). The CPU views this system as a black-box supporting the operations *reset*, *read*, *write*, *mark*, *rollback*, and *advance* (figure 2). The *desired behavior* of the RBC is specified using a model called RM1, illustrated in figure 2. This specification is not an efficient realization of the RBC operations; its purpose is only to specify their semantics. In particular, a new snapshot is created by copying the entire state vector into a new area of memory, which would not be done in an efficient implementation.

In RM1, the various snapshots are stored in a circular buffer. This is done to capture the finite memory capacity of any realization. Each element of the buffer, referred to as a *frame*, contains a single snapshot. Two pointers, `O` and `C`, point to the oldest and most recent snapshot, respectively. Read and write operations always access the frame pointed to by `C` ("frame `C`"). A *mark* operation creates a copy of the newest frame and adds it to the queue. The *rollback* operation discards the

newest frame, and the *advance* operation discards the oldest frame.

Level RM1 is described using the data type

$$\texttt{Mtype} \times \texttt{Ctype} \times \texttt{Otype}$$

where `Mtype` is the type of the entire version controlled memory (containing all the snapshots), `Ctype` is the type of the pointer pointing to the current frame, and `Otype` is the type of the pointer pointing to the oldest frame. These types can be described in a Pascal-like notation, thus:

```
(* Nframes is the number of frames maintained        *)
(* maxwordaddr is the largest address within a frame *)

Mtype     = array [0..(Nframes-1)] of frametype  (* array containing all snapshots *)
frametype = array [0..maxwordaddr] of word       (* one snapshot *)
Ctype     = 0..(Nframes-1)                       (* current frame *)
Otype     = 0..(Nframes-1)                       (* oldest frame *)
word      = integer                              (* one word of memory *)
```

The operations on RM1 are abbreviated as follows: `s1` denotes *reset*, `r1` denotes *read*, `w1` denotes *write*, `m1` denotes *mark*, `b1` denotes *rollback*, and `a1` denotes *advance*. We use the operation names *reset*, *read*, etc., while speaking about these operations in the abstract (for example while discussing these operations outside the context of a specific design such as RM1 or RM2).

3.1 Terminology and Notations

Our intention of providing level RM1 is to clearly present the desired functionality of the RBC system. Specific *timing requirements* (e.g. the number of cycles that each operation should take) are *not* being prescribed. Therefore, we shall use a purely functional [22] notation. We select the syntax of the functional programming language Miranda[1] for several reasons. First of all, the Miranda notation is very close to standard mathematical notation. Secondly, Miranda is polymorphically typed, and implements one of the most rigorous type-checking algorithms [23]. This helps eliminate virtually all type inconsistencies in the specification. Thirdly, specifications written in Miranda can be simulated and tested for specific test cases, subject to rigorous analysis [24], as well as checked for for consistency and *sufficient completeness* [25]. Finally, the functional sub-language employed in HOP is essentially the same as the one used in Miranda.

In this paper, we present level RM1, RM11, and RM12 in Miranda. Then HOP specifications of the structure of the roll back chip are provided. Behavioral inference using PARCOMP is then discussed. The behavior inferred by PARCOMP is outlined in the syntax of HOP. Then, the inferred behavior is shown in full detail in Miranda. Finally, the inferred behavioral description, RM2, is verified against the initial specification, RM1.

3.1.1 A Brief Introduction to Miranda

Miranda is a *polymorphically typed* [23] functional language. A Miranda program consists of a collection of function definitions (each preceded by an optional type declaration) and an expression that is evaluated with respect to the function definitions. Miranda programs are *referentially transparent*, which means that an identifier in a given scope denotes only one value. The values denoted can be either simple things such as numbers or complex things such as functions. Referential transparency makes Miranda programs easy to understand and reason about. Functions are treated as "first class citizens": they can be passed as arguments, returned as results, and stored in

[1] Miranda is a trademark of Software Research Corp., UK

data structures. Expressions in Miranda are *lazily* evaluated; *i.e.*, they are evaluated only as and when needed. In addition to promoting efficiency, lazy evaluation permits succinct (and unusual in the traditional sense) expression of concepts. For example, the Miranda program to obtain the least common multiple (LCM) of two numbers is:

```
lcm::num->num->num

lcm n1 n2
   = 0, if n1=0 \/ n2=0
   = hd [x | x <- [min[n1,n2]..]; (x mod n1 = 0); (x mod n2 = 0)], otherwise
```

The specification consists of two lines. The first line specifies the type *signature* of function `lcm`. The type signature says that function `lcm` takes two nums (*i.e.*, integers) and returns a num. In Miranda, the signature of a function is of the form `a->b` where `a` is the domain type and `b` the range type. A function with signature `a->b->c` represents a function that takes an element of type `a` and returns a function of type `b->c`; naturally, this function, when given an object of type `b` returns an object of type `c`. Thus, a function with signature `a->b->c` represents a two-ary function that can be *partially applied* to arguments (*i.e.* it can be applied to one argument, and then to another).

The first clause of `lcm` returns `0` if either `n1` or `n2` is `0`. *Otherwise*, the second clause in the description of `lcm` is executed. This clause is presented using the notation of *list comprehension.* It resembles the standard mathematical construction to specify sets:

$$\{x \mid predicate(x)\}.$$

The construct `[min[n1,n2]..]` represents the set of natural numbers starting at the minimum value of `n1` and `n2`. The construct `x <- [min[n1,n2]..]` means "x is a natural number greater than or equal to `min[n1,n2]`". The predicate part of the list comprehension is the check that `x` is divisible, with zero remainder, by `n1` and `n2`. A list of all such numbers is (conceptually) generated, and the `hd`, *i.e.*, the first such number is returned. Computationally, only the requisite numbers starting from 0 are generated, and *as soon as* the LCM is found, the evaluation terminates.

As another example, a Miranda program that performs the bit-wise and of two bit vectors, represented as lists, is shown below:

```
band bv1 bv2   = map and (transpose [bv1,bv2]), if #bv1 = #bv2
               = error"Bitand applied to unequal-length vectors", otherwise
```

Here, we omit the type definition: the Miranda system is capable of *inferring* the type of `band`. This specification says that the bit-wise and of the bit-vectors `bv1` and `bv2` is obtained by `map`ping function `and` onto the `transpose` of the lists `bv1` and `bv2`, provided the lists are of the same length. As an example, `transpose` takes a list containing two lists, `[[True,False,True],[False,True,False]]`, and returns the list `[[True,False],[False,True],[True,False]]`. The construct `map and` applies `and` to every sublist; *e.g.* `and[True,False]`, `and[False,True]`, and `and[True,False]` are obtained. The construct `#bv1 = #bv2` checks to see that the lists are of the same length. If the lengths are not the same, an error is raised (the `otherwise`) clause.

In Miranda descriptions used in this paper, we use the *array* data type and a two-dimensional array data type quite extensively. Their operations are now introduced.

The type `array` is introduced as an abstract data type in Miranda; its definition is provided in the appendix. Operations supported by `array` are:

1. `cv::maxind_type -> (ind_type -> *) -> array *`: Given the maximum index `maxind` into the array (of type `maxind_type`) and a function that maps indices in the range `0..maxind` to a value of *any type at all* (indicated by the `*` symbol), an array of `*`-typed elements is createdv, and initialized such that location `i` contains `(f i)`.

Usage example: `cv 3 factorial`

2. `empty::maxind_type -> array *`: Given the maximum index into the array (3), `empty` creates an un-inialized array which is indexable by indices in the range `0..3`.

   Usage example: `empty 3`

3. `indok::array * -> ind_type -> bool`: Checks to see if the given index is within bounds.

   Usage example: `indok (cv 3 factorial) 22` returns `false`

4. `maxind::array * -> maxind_type`: Returns the maximum index permitted by the array. The minimum index is assumed to be `0`.

   Usage example: `maxind (empty 3)` returns `3`.

5. `ar2l::array * -> [(ind_type,*)]`: Returns the contents of the array `a` in the range [0..maxind a] as an association list of `(index,value)` pairs.

   Usage example: `ar2l (cv 3 factorial)` returns `[(0,0),(1,1),(2,2),(3,6)]`.

6. `ar2litems::array * -> [*]`: Same as above, except that the indices are stripped.

   Usage example: `ar2litems (cv 3 factorial)` returns `[0,1,2,6]`.

7. `l2ar::[*] -> array *`: The length of the given list of items is measured, and an array of suitable size is created, and returned.

   Usage example: `l2ar [0,1,2,6]` is the same as `cv 3 factorial`.

8. `upd::array * -> ind_type -> * -> array *`: Updates the array at a given index (2) with a value (34).

   Usage example: `upd a 2 34` returns `a` with location `2` set to `34`.

9. `ind::array * -> ind_type -> *`: Index the given array and retrieve the value at the indexed location.

   Usage example: `ind (cv 3 factorial) 3` returns `6`.

10. `indable:: array * -> ind_type -> bool`: Checks to see if the array has been *defined* at the location where indexing is being attempted.

    Usage example: `indable (upd (empty 3) 2 22) 2` returns true, because location `2` has been initialized.

Two dimensional arrays are also used in our descriptions. Operations supported by them are as follows.

1. `empty2d::num->num->(array (array *))`: Create an empty two-dimensional array with the given maximum X and Y index values. The element type is not fixed a priori (indicated by the * operator).

   Usage example: `empty2d 7 3` creates an un-initialized array of size `[0..7]` by `[0..3]`.

2. `c2d::num->num->(num->num->*) -> (array (array *))`: Creates an initialized two-dimensional array. The first two arguments are as for `empty2d`. The third argument is a two-ary function `f_i_j`: its role is to initialize location `i,j` to `f_i_j i j`.

Usage example: `(c2d 7 3 (+))` creates a `[0..7]` by `[0..3]` array such that location `i,j` is initialized to `i+j`.

3. `u2di::(array (array *))->num->(num->*) -> (array (array *))`: Updates the 2d array at first coordinate `i`, and for all permissible second coordinates `j`, with `(f j)`.

   Usage example: `(u2di a n f)` gives a new array `a'` such that `ind (ind a' n) j`, for any `j`, returns `(f j)`.

4. `u2dj::(array (array *))->num->(num->*) -> (array (array *))`: Similar to `u2di` except that the update happens at `j`, for all `i`, with `(f i)`.

   Usage example: `(u2dj a n f)`.

5. `u2dij:: (array (array *))->num->num->* -> (array (array *))`: Defined to be `(upd a i (upd (ind a i) j v))`.

   Usage example: `(u2dij 2 2 43)`.

Additional details about Miranda, and supporting definitions, will be introduced as and when necessary.

3.2 Description of the RBC Operations

In this section we present the RBC operations at level RM1 in Miranda. Details omitted from the Miranda descriptions can be found in the Appendix.

3.2.1 Data Types used at Level RM1

The state at level RM1 is modeled using a three-tuple data type `rm1_type`. This, and other related type definitions are given below. Type names are introduced using the connective `==`. Comments begin with a `||`, and extend through the remainder of the line.

```
rm1_type      == (m_type,cmf_type,omf_type) || A three-tuple type
m_type        == array frame_type || array version_type of frame_type
version_type  == num              || 0..maxversion
frame_type    == array word_type  || array addr_type of word_type
addr_type     == num              || 0..maxwordaddr
word_type     == num              || one memory word
cmf_type      == num              || 0..maxversion - type of CMF
omf_type      == num              || 0..maxversion - type of OMF
```

3.2.2 Operation s1

Operation *reset* defined at level RM1 is denoted by `s1`. We specify the behavior of `s1` functionally, as follows.

```
s1::rm1_type->rm1_type
s1(m,c,o) = (empty2d maxversion maxwordaddr, 0, 0)
```

Applying `s1` on RM1 whose state is the triple `(m,c,o)` yields the triple on the right-hand side. `empty2d maxversion maxwordaddr` creates the initial version controlled memory. The CMF and OMF pointers are reset to `0`, following `s1`.

3.2.3 Operation r1

```
r1::(rm1_type,addr_type)->word_type
r1((m,c,o),a) = (ind (ind m c) a)
```

Operation r1 indexes m with c (the current mark frame) first, and indexes this array with a (the address of the word being read). In other words, c points to the latest version (frame) in memory m. This frame is first obtained by (ind m c) and is then indexed with the address a.

3.2.4 Operation w1

```
w1::(rm1_type,addr_type,word_type)->rm1_type
w1((m,c,o),a,d) = (u2dij m c a d, c, o)
```

Operation w1 writes data d in the current frame c at address a.

3.2.5 Operation m1

```
m1::rm1_type->rm1_type
m1(m,c,o) = (upd m newc (ind m c), newc, o), if newc ~= o
          = error"Mark: CMF wraps around & equals OMF", otherwise
            where newc = (c+1) mod nframes
```

Operation m1 copies the contents of the frame pointed to by c into the frame pointed to by (c+1) mod nframes, assuming that an overflow (indicated by the error check) does not occur. Thus, starting a new version through the mark operation is tantamount to copying over the entire *current frame* into the *new frame being allocated.* This is, of course, a succinct description of the actual intended effect, and not how the final implementation works.

From the above description of m1, it can be seen that the CMF pointer c can wrap-around, thereby causing an overflow, if too many mark operations are issued. More sophisticated implementations of the mark operation are possible; for example, overflows can be prevented by copying the top nframes frames of version controlled memory into backup store as in a paged virtual memory system. These possibilities are not explored here.

3.2.6 Operation b1

The actual RBC system supports a version of the rollback operation that rolls back over multiple frames. We have deliberately simplified the *rollback* operation in such a way that it rolls back by only one frame. This greatly simplified our proof, as well as simplifies our presentation thereof. The practical utility of the proof, however, is not diminished because the actual rollback operation can be simulated using multiple invocations of our version of the rollback.

```
b1::rm1_type->rm1_type
b1(m,c,o) = (m, (c-1) mod nframes, o), if c ~= o
          = error"Rollback: CMF underflows & equals OMF", otherwise
```

Operation b1 discards the current version by decrementing c, assuming that an underflow does not occur.

3.2.7 Operation a1

We have similarly simplified the operations *advance* so that it advances by one frame only. The actual RBC system supports an advance operation that advances over many frames. The latter can be simulated using the former.

```
a1::rm1_type->rm1_type
a1(m,c,o) = (m,c,(o+1) mod nframes), if c ~= o
          = error"Advance: OMF overflows & equals CMF", otherwise
```

Operation **a1** increments the oldest markframe pointer **o**, thereby discarding the oldest version being held. This is a form of *fossil collection.* Error checks are handled as with **mark**.

4 Optimization 1: Written bits array

RM1 is intended to capture the functional behavior of the rollback chip. A direct implementation of this specification, however, will clearly be inefficient because each mark operation requires making a copy of the entire data segment (one frame) of the process. We will now informally develop the RM2 specification through a series of refinements to the RM1 specification. A hardware realization of the RM2 specification will then be presented, as well as a formal specification using HOP. We will then verify that the RM1 and RM2 specifications are equivalent.

The first optimization is illustrated in figure 2 as RM11. We eliminate the data copying aspect of the mark operation. The mark operation simply increments the CMF register, making *mark* very efficient.

The principal consequence of the above modification is the stack may now contain "holes"—locations where no valid data has been stored. Immediately after the mark operation, the CMF register points to a frame containing only holes. Clearly, the hardware must be able to distinguish holes from valid data, so a new data structure called the *written bit array* is defined.

A single bit is associated with each word in the stack. This bit is set if valid data has been written into that word of the stack, and reset otherwise. Bit WB[f][a] is the bit associated with the word at address a ($Amin \leq a \leq Amax$) in frame f ($0 \leq f <$ **Nframes**). Initially, the written bits for frame 0 are set, and all other written bits are cleared.

This optimization is supported by the use of extra storage locations in the form of the WB array. This storage overhead is tolerably small.

The write operation must be modified. In addition to writing data into the frame on top of the stack, the write operation must also set the written bit corresponding to the word of data being written (i.e., wb[CMF][a] is set during writes to address a). This change does not increase the cost of *write*, as the setting of WB can be done in parallel with the write. This change requires more memory bandwidth, however.

The read operation cannot simply read data from the frame on top of the stack because there may not be any valid data stored there. Instead, the read operation must return the most recently written version of the data, excluding versions that were discarded because of rollback operations. Therefore, a read to address a must search back through the stack, starting from the CMF frame, looking for the first *set* written bit, and read the corresponding word of data. The frame containing the *most recent version* is called the MRV frame, and is defined as the frame such that $wb[MRV][a] = 1$ and $wb[f][a] = 0$ for all integers f such that $MRV < f < CMF + 1$. The read operation returns the data stored in STK[MRV][a].

A caching mechanism was developed to eliminate written bit searches for most read operations, in another version of the RBC [1]; however, this model is beyond the scope of the present discussion.

The above mentioned linear search involved during *read* is actually implemented more efficiently, as follows. Written bits are actually stored packed in words of Nframes bits. These words can be read in one cycle. The search for the first set written bit, scanning these Nframes bits from one end to the other, can then be implemented using a *circular priority encoder* (a priority encoder that treats the Nframes bits coming into it in a "circular" (modulo) fashion. The most significant bit of our circular priority encoder at any time is defined to be the CMF-*th* bit of the input word).

4.1 Types used at Level RM11

The state of level RM11 is modeled using `rm11_type`. This, and related types, are described below.

```
rm11_type      == (m_type,cmf_type,omf_type,wbstore_type)
wbstore_type   == array wb_type    || array addr_type of wb_type
wb_type        == array bool       || array version_type of bool
```

4.2 The Written Bits array

The written bits array is a two-dimensional array of bits. We specify the written bits array through its operations.

Operation `createw` creates the initial state of the written bits array:

```
createw::maxaddr_type->maxversion_type->wbstore_type
createw maxa maxv = c2d maxa maxv v0set
                    where v0set i j = (j=0)
```

Operation `createw` takes the maximum address in a frame, `maxa`, and maximum number of versions, `maxv`, and creates a two-dimensional array using function `c2d`. Function `v0set`, which is used to fill the array, returns true for all `i,j` where `j = 0`. In effect, the written bits array created initially specifies that for every address, the *zero*th version is the latest.

Operation `setmrv` is used to record that version `v` is the latest one for address `a`:

```
setmrv::wbstore_type->addr_type->version_type->wbstore_type
setmrv w a v = u2dij w a v True
```

Operation `clrmrv` is used to discard version `v` belonging to every address `a`:

```
clrmrv::wbstore_type->version_type->wbstore_type
clrmrv w v = u2dj w v f
             where f i = False
```

The written bits at coordinates `i,v`, for all `i` in the allowed range of addresses, are cleared.

Operation `clrmrva` clears only the written bit at the given coordinates `a` and `v`:

```
clrmrva::wbstore_type->addr_type->version_type->wbstore_type
clrmrva w a v = u2dij w a v False
```

Operation `mrv` examines the written bits located at the first coordinate `a` of the written bits array. The second coordinate into the written bits array is started at location `c`, where `c` is the current mark frame (CMF), and the `mrv` operation *scans* the written bits such that if location i is being currently examined in the process of scanning, the next location to be examined will be $i-1$, if i is currently greater than 0, and $maxv$ if i is currently equal to 0. While so scanning, `mrv`

detects the first set written bit and returns its location. If there are no set written bits, **mrv** returns the number of a distinguished location, called **aframeaddr**. This corresponds to the *archive frame*, whose purpose will be explained shortly. Briefly, operation **mrv** locates the most recent version for a given address, **a**.

Operation **mrv** is specified using a series of helping functions. Function **circmod** behaves similar to function **mod** when applied to positive integers. When applied to negative integers, it simulates "wrap around" in the circular buffer of versions:

```
circmod::num->num->num
circmod x nitems = x mod nitems, if x >= 0
                 = x + nitems, otherwise
```

For example, `circmod 0 nitems` is `0`, and `circmod -1 nitems` is `nitems-1`; the latter example shows the "wrap around" effect.

Function **genvers** generates the list of versions examined in the process of scanning from **c**, the CMF, to **o**, the OMF. The type signature specifies that **genvers** takes two nums and returns a *list of* nums:

```
genvers::num->num->[num]
genvers c o   = error"genvers: c or o are out of range",
                  if (c > maxversion) \/ (o > maxversion) \/ (c < 0) \/ (o < 0)
              = [o], if c = o
              = c:(genvers (circmod (c-1) nframes) o), otherwise
```

Operation **genvers** raises an error if **c** or **o** are out of range. Else, if **c=o**, **genvers** returns a list containing **o** alone; this is the only version to be examined during scanning. Else, **genvers** generates the list obtained by adding **c** to the head of the list (through the *cons* operation **:**) to the list of versions generated from `(circmod (c-1) nframes)` to **o**.

For example, `(genvers 3 1)`, when `nframes = 4`, returns `[3,2,1]`. The expression `(genvers 2 3)`, when `nframes = 4`, returns `[2,1,0,3]`. Notice how the wrap-around occurs beyond version `0`.

Finally, function **mrv** is:

```
mrv::wbstore_type->addr_type->cmf_type->omf_type->version_type
mrv w a c o = fst(hd lset), if lset ~= []
            = aframeaddr, otherwise
              where lset = [(v,ind (ind w a) v) | v<-(genvers c o); ind(ind w a) v]
```

Operation **mrv** first forms the list of `(version, written-bit)` pairs in the range being scanned. It is expressed through the following list comprehension

```
[(v,ind (ind w a) v) | v<-(genvers c o); (ind (ind w a) v)]
```

If this list is, for example, `[(1,False),(0,False),(3,True)]`, **mrv** returns `1`. If this list is, as another example, `[(1,False),(0,False),(3,False)]`, **mrv** returns **aframeaddr**.

The RBC operations may now be rewritten for level RM11.

4.2.1 Operation s11

The reset operation at level RM11, `s11`, creates an uninitialized memory `m`, as with operation `s1`. However, note that operation `s1` created the memory using the expression `(empty2d maxversion maxwordaddr)` while operation `s11` creates the memory using the expression `(empty2d nframes maxwordaddr)`. The constant `nframes` is defined to be one more than the constant `maxversion`. We provide one more frame for the memory at level RM11 to serve as the "archive frame", whose purpose will be explained later. The CMF and OMF are initialized to 0 by `s11`. Finally, the written bits array is initialized to indicate that for every address `a`, the most recent version is at frame `0`, *i.e.*, at the first frame that is going to be used.

```
initial_wb::wbstore_type
initial_wb   = createw maxwordaddr maxversion

s11::rm11_type->rm11_type
s11(m,c,o,w) = (empty2d nframes maxwordaddr, 0, 0, initial_wb)
```

Operation *read* fetches data from the `mrv` location of the memory:

```
r11::(rm11_type,addr_type)->word_type
r11((m,c,o,w),a) = ind(ind m (mrv w a c o)) a
```

Operation *write* updates `m` at coordinates `c,a` with `d`. It also records that the most recent version at address `a` resides at version `c`, through the `setmrv` operation.

```
w11::(rm11_type,addr_type,word_type)->rm11_type
w11((m,c,o,w),a,d) = (u2dij m c a d, c, o, setmrv w a c)
```

Operation *mark* increments the CMF pointer, modulo `nframes`. It also does `clrmrv` for every address `a`, at the new current version (`(c+1) mod nframes`). This makes sure that when the CMF pointer wraps around, "stale" written bits won't be seen.

```
m11::rm11_type->rm11_type
m11(m,c,o,w)  = (m, newc, o, clrmrv w newc), if newc ~= o
              = error"Mark: CMF wraps around & equals OMF", otherwise
                where newc = (c+1) mod nframes
```

Operation *rollback* simply decrements the CMF. It does not clear the written bits as *mark* does this "for *rollback*".

```
b11::rm11_type->rm11_type
b11(m,c,o,w)  = (m, (c-1) mod nframes, o, w), if c ~= o
              = error"Rollback: CMF underflows & equals OMF", otherwise
```

Finally, operation *advance* updates the memory such that: (i) for every version older than OMF that has a set written bit, data from that version is moved into the archive frame, for every address `a`. Since the archive frame is part of `m`, we model this process through function `archive`.

```
a11::rm11_type->rm11_type
a11(m,c,o,w)  = (archive m w o, c, (o+1) mod nframes, w), if c ~= o
              = error"Advance: OMF overflows & equals CMF", otherwise

archive::m_type->wbstore_type->num->m_type
archive m w o
= foldr oneupd m [0..maxwordaddr]
  where
  oneupd a m
  = u2dij m aframeaddr a (ind ind_m_copy_source a),
       if (indable ind_m_copy_source a)
  = m, otherwise
    where ind_m_copy_source = (ind m copy_source)
          copy_source = o, if ind (ind w a) o
                      = aframeaddr, otherwise
```

Function **archive** calls function **foldr**. Function **foldr** helps form right-linear trees of function applications. For example

```
foldr (+) 23 [4,5,6] = (4 + (5 + (6 + 23)))
```

In the definition of function **archive**, we pass as the first argument of **foldr** the function **oneupd**. Function **oneupd** performs one update of the memory **m**. Operation **oneupd** is repeatedly applied for **a** ranging through **0** through **maxwordaddr**. For example,

```
foldr oneupd m [0..2] = (oneupd 0 (oneupd 1 (oneupd 2 m)))
```

Operation **archive** is implemented by repeatedly updating the memory **m** at version **aframeaddr** and address **a** with data **(ind ind_m_copy_source a)**. Let us examine where this data comes from.

Expression **(ind ind_m_copy_source a)** retrieves the data at address **a** from the appropriate version. This version is determined by the value of the expression **ind_m_copy_source**: it is the version pointed to by the OMF value **o** if the written bit at version **o** is set (as shown by the test **(ind (ind w a) o)**); it is version **aframeaddr** if this bit is not set.

Thus, the data at version **o** and address **a** is saved if the OMF bit is set; otherwise, the data present at version **aframeaddr** and address **a** is copied back to that location itself. This last aspect—that of copying data from a location back to itself is simply an artifact of the definition (to make keep the definition succinct); this feature will not be implemented in the final hardware!

Lastly, as part of the operation **advance**, OMF is incremented.

5 Optimization 2: Roll Back Histories

An important optimization pertaining to the *rollback* operation is presented in this section. As mentioned in section 3, in addition to popping frames from the stack, the rollback operation must also clear the written bits corresponding to these discarded frames. Clearing these written bits will be relatively expensive, so an optimization called the *rollback history* (RBH) mechanism was developed (level RM12 of figure 2).

The rollback history mechanism avoids explicitly clearing written bits when a rollback occurs, allowing the rollback operation to be implemented very efficiently. Rather than clearing the written bits on each rollback, they are cleared "lazily," i.e., just before they are examined by subsequent read and write operations. Therefore, the question that must be answered is "which written bits

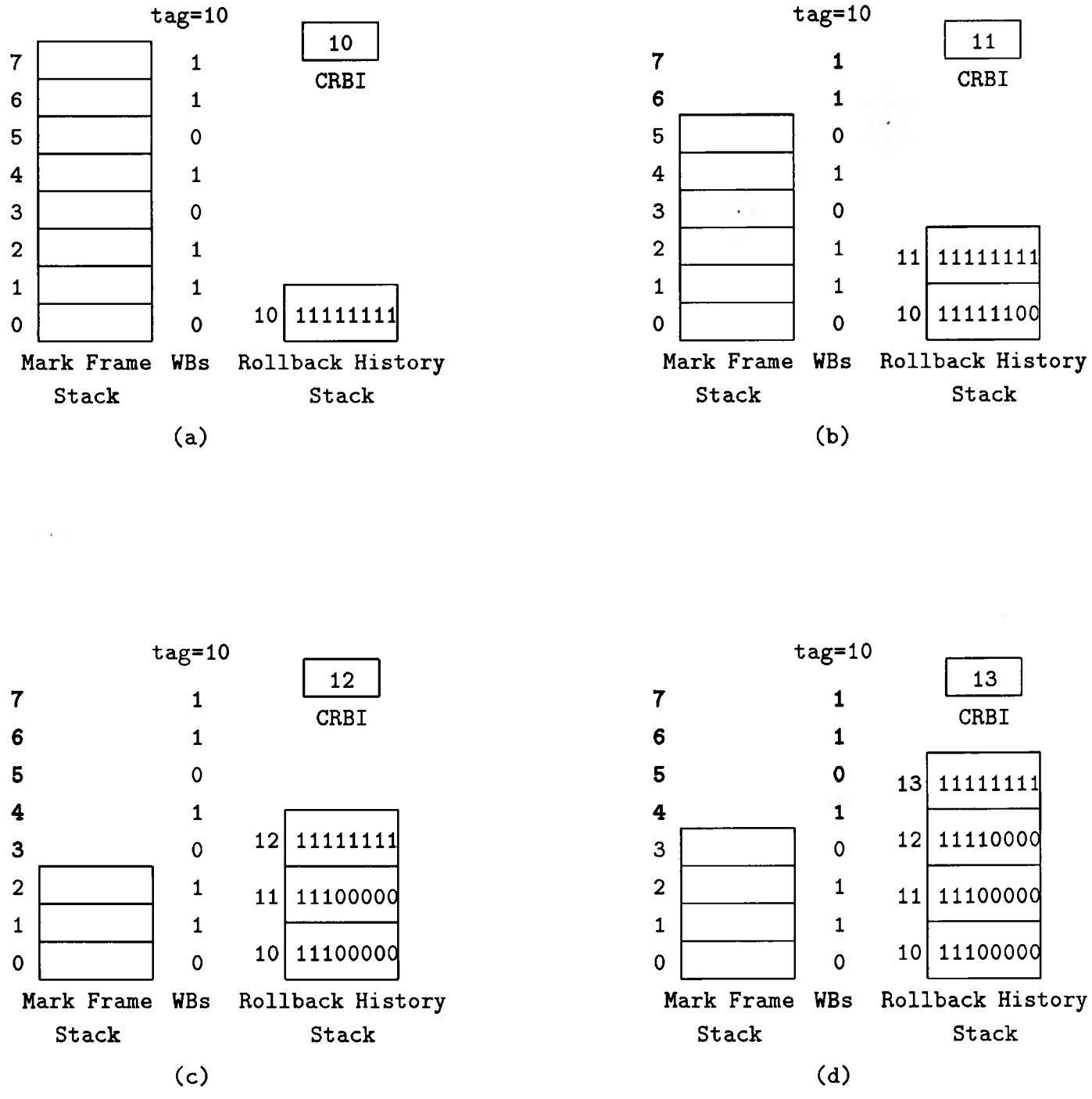


Figure 3: Examples Referring to the Rollback Histories stack (RBH)

must be cleared when they are read from the written bit memory?" The *rollback history (RBH)* stack provides the information necessary to answer this question.

A word of the written bit memory is defined as the collection of written bits associated with a single word of version controlled memory (i.e., WB[f][a] for all f). A "tag" value is included with each word that indicates when the written bits were last updated. This tag value is actually a count of the number of rollback operations that have been issued to the RBC thus far.

One word for eight frames of the mark frame stack, the corresponding written bits, and the tag are shown in figure 3a. We see that valid data has been stored into frames 1, 2, 4, 6, and 7, and no valid data was stored into the other frames. The tag value of 10 indicates that the written bits were last updated sometime after the 10th rollback. Further, assume that no rollbacks have occurred since the 10th rollback, so the written bits are up-to-date.

Suppose that a rollback now occurs to frame 5. This means that the written bits for frames 6 and 7 should become zero because those frames have been discarded. Rather than explicitly clearing the written bits, we instead remember that a rollback to frame 5 has since occurred, and bits 6 and 7 are now actually zero. We do this by storing a mask vector equal to 11111100 into RBH[10] (the leftmost bit refers to frame 0, the rightmost to frame 7). In general, zero bits of the mask vector correspond to discarded frames, and one bits indicate frames that remain in the stack. When the written bits (01101011) are later read, we also (1) read the tag (10), (2) read the corresponding RBH entry (RBH[10] = 11111100), and perform a bitwise logic AND on the written bits and RBH value (yielding the desired value, 01101000). The state of the system after this rollback is shown in figure 3b. An additional entry has been added to the RBH stack, as will be described next.

In general, RBH[i] indicates, *the "deepest" rollback (i.e., smallest destination frame number) that has occurred since time ith rollback.* This implies that a new element is pushed onto the stack after each rollback. This explains the new entry at RBH[11] in figure 3b. The new element always contains a mask of all one bits because no rollbacks have occurred since the last one (obviously!).

Each rollback operation requires that the following operations be performed:

1. The counter indicating the number of rollbacks that have occurred thus far, called the CRBI (current roll back index), is incremented. The current value of the CRBI register is stored as the tag whenever written bits are stored into the written bit memory.

2. RBH[CRBI] is set to a bit vector of all ones, as described above.

3. For each mask vector RBH[i] for i<CRBI, clear any bits of the mask that correspond to frames that were discarded as a result of the rollback.

The latter operation would seem to be expensive because it appears that every element of the RBH stack must be checked. One can optimize this operation by observing that the deepest rollback since the ith rollback (stored in RBH[i]) is no deep than the deepest rollback since the i-1st rollback. Thus, if no bits were cleared in the mask for RBH[i], we are guaranteed that no bits will be cleared in RBH[i-1]. Thus, when updating the RBH, we can scan the RBH elements from high indices to low, and stop scanning once we discover that no mask bits were cleared.

Suppose that in the above example the rollback to frame 5 is followed by a rollback to frame 2, two mark operations occur that push two new frames onto the mark frame stack, followed by a rollback to frame 3. The rollback to frame 2 will push a new vector of 1's onto the RBH, clear three additional bits of RBH[10] (corresponding to the three additional frames that were discarded), and clear bits 3 through 7 of RBH[11], leaving the RBH in the state shown in figure 3c. The subsequent marks and rollback operation (to frame 3) leave the RBH stack in the state shown in figure 3d. This last rollback creates a new entry RBH[13] and updates RBH[12]. It is discovered that RBH[11] is *not* modified, so there is no need to check the remainder of the stack.

Here, we will assume the RBH stack is allowed to grow without bound. A mechanism has been developed to "forget" old RBH elements, as described in [1], but is beyond the scope of the present discussion.

5.1 Types used at Level RM12

The RBC state at this level is modeled using type `rm12_type`. This, and related types, are described below.

```
rm12_type      == (m_type',cmf_type,omf_type,wbstore_type,tstamp_type,
                   rbh_type)
m_type'        == array frame_type || array [0..maxversion+1] of frame_type
rbh_type       == [num]            || list of 0..maxversion + "infinity"
tstamp_type    == array time_type  || array addr_type time_type
time_type      == num              || time-stamps for each wbword in wbstore
```

5.2 Formal Description

We implement the RBH stack using `rbh_type`. The `infinity` word can be implemented suitably, using a number that does not belong to the set of valid versions. One example is `infinity = 9999`.

The initial state of the RBH stack is realized through the `creater` operation:

```
creater::rbh_type
creater = [infinity]
```

When a rollback occurs, the depth of the deepest rollback since time t, for every time t, is updated using operation `updater`.

```
updater::rbh_type->version_type->rbh_type
updater r d   = infinity:(minimize r d)

minimize::[*]-> * ->[*]
minimize [] d         = []
minimize (h:rest) d   = h:rest, if d >= h
                      = d:(minimize rest d), otherwise
```

The key function here is `minimize`. This function takes a list representing the rollback history stack and a "destination" value `d` that represents the deepest of the versions to be discarded following a rollback. If the RBH stack is empty (represented by the notation `[]`), then there is nothing to be done. If the RBH stack is a list with head `h` and tail `rest` (`h:rest`), no changes are effected if `d >= h`; else, we replace the head with `d`, and recursively apply `minimize` to the tail of the list.

The depth of the deepest rollback since time `t` is returned by function `indexr`. It simply reads off the `l-1-t`th element, where `l = #r` is the length of the RBH stack. (Function `#` returns the length of a list.)

```
indexr::rbh_type->time_type->version_type
indexr r t = r!(#r-1-t)

time::rbh_type->time_type
time r = (lengthr r)-1
```

Given an RBH stack r, the "current time" is calculated by function time.

We now describe the operations at the level RM12. These operations are efficient versions of the operations defined at level RM1; however they still do not correspond to a hardware realization of the RBC; this would be the purpose of level RM2, which is yet to come in this paper. The state of the RBC at level RM12 is rm12_type.

5.2.1 Operation definitions at level RM12

Operation s12 resets the timestamps associated with the written bits, t, through the expression cv maxwordaddr zerofn. It also creates an empty RBH stack, creater. The other components of the state are initialized as before.

```
s12::rm12_type->rm12_type
s12(m,c,o,w,t,r)
          = (empty2d nframes maxwordaddr, 0, 0,
             initial_wb, cv maxwordaddr zerofn, creater)
             where zerofn i = 0
```

Operation r12 reads data from location clearedmrv of the memory.

```
r12::(rm12_type,addr_type)->word_type
r12((m,c,o,w,t,r),a) = ind(ind m (clearedmrv w t r c o a)) a
```

Function clearedmrv simulates the process of clearing the written bits read off the wbarray using information contained in the RBH, and then obtaining the most recent version. First of all, the written bits that ought to have been cleared corresponding to address a are determined through the expression (indexr r (ind t a)). This value, called the RBH version, or rv, is then passed to function fixw along with the other arguments of fixw.

Function fixw returns w unchanged if the most recent version after "fixing" the written bits is the archive frame (the first clause in the definition of fixw). This means that all the written bits were already cleared or ended up getting cleared by fixw. It returns (foldr oneupd w (genvers mrv' rv)), if o, rv, and mrv' are all "lined up" (determined by predicate lineup).

Predicate lineup makes sure that rv is within the range defined by the values o and mrv', where mrv' is the first set written bit scanning backwards from c *without clearing any bits that ought to have been cleared.* We can see how lineup is defined—a, b and c "line up" if b is a member of the list generated by genvers. (Function genvers was defined earlier.)

Coming back to the foldr expression below, it applies function oneupd to w for every value in the range generated by genvers mrv' rv. Each application of oneupd clears one written bit that ought to have been cleared.

```
clearedmrv::wbstore_type->tstamp_type->rbh_type->cmf_type->omf_type->
            addr_type -> version_type
clearedmrv w t r c o a = mrv (fixw w a (indexr r (ind t a)) c o) a c o

fixw::wbstore_type->addr_type->version_type->cmf_type->omf_type->wbstore_type
fixw w a rv c o = w, if mrv' = aframeaddr
                = foldr oneupd w (genvers mrv' rv), if lineup o rv mrv'
                = w, otherwise
                  where
                      oneupd v w = u2dij w a v False
                      mrv' = mrv w a c o

lineup::num->num->num->bool
lineup a b c = member (genvers c a) b
```

Operation **w12** fixes stale written bits in the wbarray and also sets the written bit corresponding to the word being written. This is achieved through the expression *setmrv(fixw w a (indexr r (ind t a)) c o)*. Also notice that while writing the written bits, we update the time-stamp of the written bits at address a, the address being written. The other components of the state are updated as before.

```
w12::(rm12_type,addr_type,word_type)->rm12_type
w12((m,c,o,w,t,r),a,d)
= (u2dij m c a d, c, o, setmrv(fixw w a (indexr r (ind t a)) c o) a c,
   upd t a (time r), r)
```

Operation **m12** performs the **updater** operation on the RBH, using the version **newc** as argument. The purpose of this step, an optimization step, is as follows.

During a *mark* operation, when the CMF pointer wraps around from frame F to frame $F+1$ *mod nframes*, stale written bits will be encountered. One (straight-forward) solution would be to clear *all* the written bits at all addresses a, corresponding to frame number `(c+1) mod nframes`. This is again an operation reminiscent of **r11** in that written bits are being industriously cleared. A better solution would be to do the following:

- increment c to `(c+1) mod nframes`;
- *simulate* a rollback by one version;
- do a mark again.

This simulated one-version rollback is "remembered" by the RBH unit. Thus, instead of clearing written bits when mark causes CMF to wrap around, we remember that the bits ought to have been cleared, and proceed!

This optimization was, in fact, discovered in the process of verification.

```
m12::rm12_type->rm12_type
m12(m,c,o,w,t,r) = (m, newc, o, w, t, updater r newc), if newc ~= o
                 = error"Mark: CMF wraps around & equals OMF", otherwise
                   where
                   newc = (c+1) mod nframes
```

Operation rollback does an **updater** on the RBH, thereby remembering which written bits ought to have been cleared.

```
b12::rm12_type->rm12_type
b12(m,c,o,w,t,r) = (m, (c-1) mod nframes, o, w, t, updater r c), if c ~= o
                 = error"Rollback: CMF underflows & equals OMF", otherwise
```

Finally, *advance* does the archiving as before, except that when words are read from memory, the written-bits clearings supported by RBH are invoked.

The archiving is achieved through function **archive2**. It is worth noticing how the source address, for reading the data to be archived, is determined. This address, **copy_source**, is o, if the most recent version of data for address a starting with version o is the OMF o itself. If not, the value of **copy_source** is the archive frame itself.

```
a12::rm12_type->rm12_type
a12(m,c,o,w,t,r) = (archive2 m w o c r t, c, (o+1) mod nframes, w, t, r),
                     if c ~= o
                 = error"Advance: OMF overflows & equals CMF", otherwise

archive2::m_type'->wbstore_type->omf_type->cmf_type->rbh_type->
          tstamp_type->m_type'
archive2 m w o c r t
= foldr oneupd m [0..maxwordaddr]
  where
  oneupd a m
  = u2dij m aframeaddr a (ind ind_m_copy_source a),
                                    if (indable ind_m_copy_source a)
  = m, otherwise
    where ind_m_copy_source = (ind m copy_source)
          copy_source = o, if (mrv  (fixw w a (indexr r (ind t a)) c o)
                                    a o o) = o
                       = aframeaddr, otherwise
```

6 Structural Specification

The final realization of RM2 is shown in figure 4. This figure provides a hardware realization of the ADT RM12 discussed in the previous section. It is this version that was run through PARCOMP, producing the inferred behavior. Similar to the description of the RM1 level, the inferred behavior RM2 is also a collection of functional programs that map RM2 states to RM2 states. The differences with RM1 are: (a) RM2 programs and data structures are more involved; (b) they will execute with much more space and time efficiency, as can be checked through performance simulation studies; (c) last, but not the least, they reflect all the design decisions taken by the designer; thus, when formally verified, the entire design is verified for structural, temporal, and functional correctness.

Each submodule in the schematic of RM2 includes data inputs and outputs, as well as its control points (in the form of *input events*, described below). The RBH unit is ideally built in custom VLSI. The WB+TS store comprises the WAC counter also. One operation, `write`, will now be discussed with respect to this schematic.

When `write` is invoked, written bits for the input address word are read from the `written bits` store. The time-stamp corresponding to this address is read from the `time-stamp tag` store, and used to index the RBH unit to read a masking word from the RBH. This word is used to mask bits read from the `written bits` store, through the help of WBAND. Then, the written bit corresponding to CMF is set with the assistance of the BITOR gate and `WBlatch` is loaded with this value. Concurrently, a write request is issued on the RAM, with CMF acting as the MSB of the address (this being the effective address). All the above happens in the first cycle of the `write`. In the second cycle of the `write`, the value latched in `WBlatch` is written into the written bits store, in place of the old value.

We now provide an overview of the HOP description of the architecture shown in figure 4. We first provide the ABSPROC descriptions of two representative modules: an and gate, and the WB+TS+WAC module. We then provide an overview of the REALPROC description describing the entire RM2 architecture.

6.0.2 Absproc of an and gate

The following is the description of a 32-bit and gate. It has name `and32`. We first declare a local type called `wbtype` which is a bit vector of size 32. We then declare the input ports `?in1` and

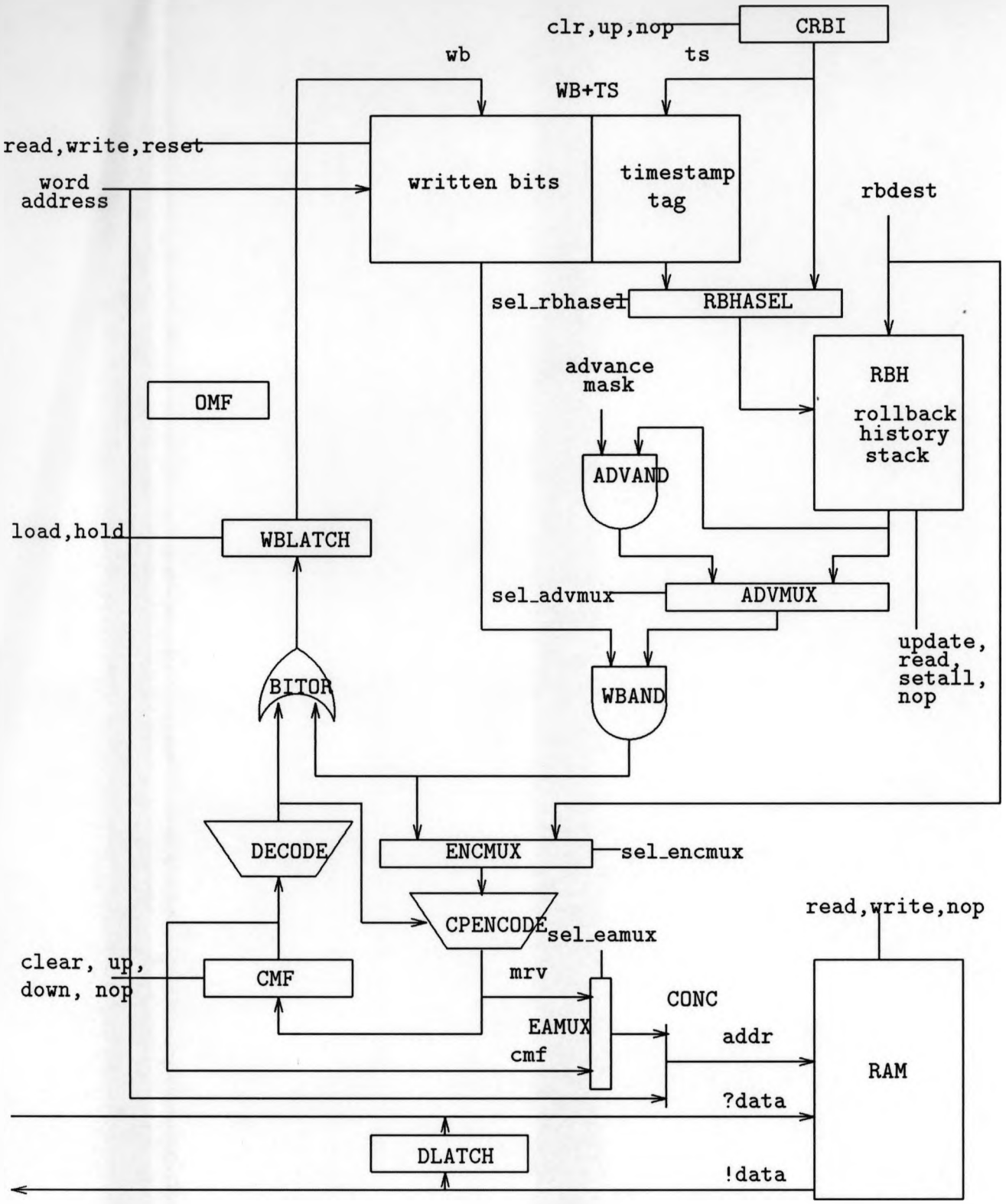


Figure 4: Structural Specification (Architectural Schematic)

?in2, and output port !out of the and gate. Finally the behavior of the and gate is specified in the section protocol.

The behavior of a HOP process is specified by modeling it as an idealized synchronous sequential system. The following process description captures the behavior of a combinational unit (and32). Process and32 simultaneously (simult) acquires two inputs, in1 and in2, through input ports (whose names are prefixed by ?) ?in1 and ?in2. It then outputs the bit-wise and of the inputs on output port !out. Then it continues to behave (become) as and32. More formally, the and32 module implements a mapping from its input streams to its output stream where the element of the output stream at time t is obtained by performing bit-wise and of the inputs arriving at times t.

```
((absproc and32)
 (type wbtype = (make-type vector-type :min-indx 0 :max-indx 31 :base-type bit))
 (port (?in1 ?in2 !out) of wbtype)
 (protocol
  (process and32 ()
   ( (simult (in1 = ?in1)(in2 = ?in2)
             (!out = (create-vector wbtype
                      (i (and (index-vector wbtype in1 i)
                              (index-vector wbtype in2 i)))))
     ) -> (become and32) )))
 (end and32))
```

6.0.3 Absproc of WB+TS+WAC

Unit WB+TS+WAC maintains the written bits (W) and the time-stamps (T) mentioned in level RM12. In addition, it maintains a counter called WAC that is used to step through all addresses in the range 0..maxwordaddr during the advance operation. Three vector types which are used in the specification are declared first. This is followed by the declaration of the ports. WB+TS+WAC also supports five operations, *nop*, *read*, *write*, *reset*, *clrwac*, *upwac*, and *rdwac*. These operations are invoked by asserting one of the input *events* supported by WB+TS+WAC. Each of the events supported by WB+TS+WAC corresponds to a Boolean condition that is typically encoded via control and clock inputs of WB+TS+WAC. During the top-down design of a large hardware system, the exact nature of these encodings is either unknown, or it is desirable to suppress such details. Events used in HOP serve this purpose by associating the module with a fictitious Boolean input that represents the encodings. A module that uses module WB+TS+WAC can generate one of the events supported by WB+TS+WAC through an *output event*. Again, in an actual hardware circuit, these events may be generated by suitable control encodings.

The behavior of WB+TS+WAC is specified by process wb+ts+wac. This process has one *control state* called wb+ts+wac and a three-component data state, namely (wbs, tss, wac). It begins with a choice construct, which is a deterministic construct that captures branching. The first two choices are based on whether the events inop and iread are found asserted. The next two choices are captured by a simult construct. Consider the first simult construct for detailed study:

```
((simult iread (wordaddr = ?wordaddr)
               (!wb = (index-vector wbarray wbs wordaddr))
               (!ts = (index-vector tsarray tss wordaddr)))
```

This simult construct has four arguments:

- iread, which is a control input event,

- `(wordaddr = ?wordaddr)`, which is a data query, that acquires a value through port `?wordaddr`, and assigns this value to `wordaddr`,
- `(!wb = (index-vector wbarray wbs wordaddr))`, which is a data assertion, that asserts on port `!wb` the value `(index-vector wbarray wbs wordaddr)`, and
- a data assertion on port `!ts`.

The entire specification of `wb+ts+wac` is now given.

```
((absproc wb+ts+wac maxwordaddr of int)
 (type
   wbtype = (make-type vector-type :min-indx 0 :max-indx 31 :base-type bit)
   wbarray = (make-type vector-type :min-indx 0 :max-indx maxwordaddr :base-type wbtype)
   tsarray = (make-type vector-type :min-indx 0 :max-indx maxwordaddr :base-type int)
 )
 (port (?wordaddr ?ts !ts) of int
       (?wb !wb) of wbtype
       !waciszero of bit);assume wac is a modulo counter. This o/p t when wac=0
 (event (inop iread iwrite ireset iclrwac iupwac irdwac) = tbd)
 (protocol
  (process wb+ts+wac (wbs  of wbarray
                      tss  of tsarray
                      wac  of int)
   (choice
    (inop   -> (become wb+ts+wac wbs tss wac))
    (ireset -> (become wb+ts+wac (reset-wb wbs) (reset-ts tss) wac))
    ((simult iread (wordaddr = ?wordaddr)
                   (!wb = (index-vector wbarray wbs wordaddr))
                   (!ts = (index-vector tsarray tss wordaddr)))
      -> (become wb+ts+wac wbs tss wac))

    ((simult iwrite (wordaddr = ?wordaddr) (wb = ?wb) (ts = ?ts))
      -> (become wb+ts+wac (update-vector wbarray wbs wordaddr wb)
                           (update-vector tsarray tss wordaddr ts)
                           wac))
    (iclrwac -> (become wb+ts+wac wbs tss 0))
    (iupwac  -> (become wb+ts+wac wbs tss (modadd1 wac maxwordaddr)))
    ((simult irdwac (!waciszero = (iszero wac))
                    (!wb = (index-vector wbarray wbs wac))
                    (!ts = (index-vector tsarray tss wac)))
             -> (become wb+ts+wac wbs tss wac))
 )))
 (defun
       (function reset-wb (wb of wbarray) to wbarray
               (create-vector wbarray
                       (i (create-vector wbtype (j F)))))
       (function reset-ts (ts of tsarray) to tsarray
               (create-vector tsarray (i 0)))
 )
 (end wb+ts+wac))
```

6.1 Inferring Behavior using PARCOMP

As referred to in section 1, PARCOMP infers the behavior of a network of synchronous system components specified in HOP. More specifically, the input to PARCOMP is a REALPROC defining

the architectural schematic. A REALPROC does three things: it instantiates previously declared ABSPROCs; it specifies connections between the data ports of the submodule ABSPROCs, and also between events of the submodule ABSPROCs; and finally, specifies which connections are exported and which are hidden.

Given this information, PARCOMP symbolically simulates the submodules through all possible executions. Since data path states and data inputs/outputs are kept symbolic, PARCOMP has to only explore various combinations of control flows (for every transition possible in a submodule at time t, the interaction of that transition with every other transition of other submodules at t is determined). In exploring combinations of control flows, PARCOMP eliminates control flow combinations that will not arise. For example if submodule A would make a transition based on input event `e`, if no other submodule generates event `e` at this time, and further, `e` is hidden from outside, the move through event `e` will not be taken by submodule A.

The schematic shown in figure 4 was the largest example run through PARCOMP, to date. Due to the size of RM2, we had to (manually) split the controller into separate ones, one for each operation. Also, the datapath was divided into two sections and these sections were composed separately before being composed together. All this was to conserve memory usage. Some statistics are provided in figure 5.

Operations	Ctrl States In Cartesian Prod.	Ctrl States In Final Process	Transitions Pruned During PARCOMP	Transitions In Final Output	Run time (secs) Lucid Lisp, 16M HP320
Reset	2	2	2686	3	11
Read	1	1	1343	2	5.5
Write	2	2	2686	3	11
Mark	1	1	105	4	0.82
Rollback	2	2	2686	3	11
Advance	4	4	6715	6	28

Figure 5: Performance of PARCOMP on the RBC

6.2 Simplified results of PARCOMP

The behavior inferred by PARCOMP in the syntax of HOP is not presented due to lack of space. We subjected the inferred behavior to simplifications, such as alpha-conversion of identifier names, and the application of the following rewrite-rules:

```
(if true a b) => a
(if false a b) => b
(index-vector vtype (create-vector vtype (I exp)) I) => exp
```

`create-vector` creates a vector of type `vtype` such that the `I`th location contains `exp`. `index-vector` indexes such a vector at location `I`, yielding `exp`. The simplified inferred behavior is outlined in figure 6 for the `rollback` and figure 7 for `read`. (Note that section 7 presents the inferred behavior for all the operations in our simple syntax, preparatory to verification.)

Rollback takes the state elements shown on the left column of this figure and the inputs shown in the middle column. It updates `RBHDPS` to an expression that, according to our conventions, is tantamount to pushing an "infinity" word. The `CMF` register is updated to the circular priority-encoded value of the `RBDEST` value, unless the `RBDEST` value is a vector of zeros (the `allzero` test),

```
State     | Inputs          | Next-state
-------------------------------------------------------------------------
RAM       | IRB             | RAM
DLATCH    | RBDEST=?RBDEST| DLATCH
OMF       |                 | OMF
AFRAMEREG|                 | AFRAMEREG
WB        |                 | WB
TS        |                 | TS
WAC       |                 | WAC
RBHDPS    |                 | (UPDATE-VECTOR RBHARRAY
          |                 |                (RBH-UPDATE RBHDPS RBDEST)
          |                 |                (+ CRBI 1)
          |                 |                (CREATE-VECTOR WBTYPE (I T)))
          |                 |
WBLATCH   |                 | WBLATCH
AMASKREG |                 | AMASKREG
CMF       |                 | (IF (ALLZERO RBDEST)
          |                 |      AFRAMEADDR
          |                 |      (CPENCODE RBDEST
          |                 |                (DECODE CMF)))
CRBI      |                 | (+ CRBI 1)
-------------------------------------------------------------------------
```

Figure 6: Inferred Behavior for Rollback

in which case the rollback has underflowed; in this case, the frame number of the archive frame is loaded into CMF.

The inferred behavior corresponding to the read operation reflects the value flows through the circuit even more clearly. Output data is generated by concatenating the effective frame address with WORDADDR. The effective frame address is generated using the expression that begins with (if (allzero ...)). This expression yields the archive frame address, if the written bits, after masking using WBAND, emerge to be all zeros. If not, this masked written bits word is encoded using the circular priority encoder and sent to the RAM.

7 Inferred Behavior, RM2

7.1 Specification of Level RM2

The state at level RM2 is described using the type rm2_type. This, and related type definitions, are listed below.

```
rm2_type       == (m_type',cmf_type,omf_type,wbstore_type,tstamp_type,
                   rbh_type', time_type, version_type)
rbh_type'      == [wbmask_type]    || [[list_of_bool_of_length_nframes]]
wbmask_type    == [bool]           || Each value in RBH is like a mask
maxversion_type== num              || Sundry types used in defining wbstore
maxaddr_type   == num              ||  operations.
```

The fields of rm2_type stand, respectively, for the Memory, CMF, OMF, Written bits array, Time stamp array, Rollback histories stack, current rollback Index, and Advance counter. This level represents the inferred behavior of the RBC, and was obtained by hand-transliterating the inferred behavioral description into Miranda. There are many reasons why this path was followed. First of all, this notation is consistent with the notations used during the process of top-down refinement. Also, as HOP stands, its functional notation is a bit involved (as can be seen from figure 7). Under

```
State     | Inputs
-------------------------------------------------------------------------------
RAM       |IREAD
DLATCH    |
OMF       |(WORDADDR = ?WORDADDR)
AFRAMEREG|
WB        |
TS        |
-------------------------------------------------------------------------------
 Output

(!DATA =
 (INDEX-VECTOR CONTENTS
               RAM
               (CONC WORDADDR
                     (IF (ALLZERO
                           (CREATE-VECTOR WBTYPE
                                          (I (AND
                                              (INDEX-VECTOR
                                               WBTYPE
                                               (INDEX-VECTOR WBARRAY WB WORDADDR)
                                               I)
                                              (INDEX-VECTOR
                                               WBTYPE
                                               (INDEX-VECTOR
                                                RBHARRAY
                                                RBHDPS
                                                (INDEX-VECTOR
                                                 TSARRAY TS WORDADDR))
                                              I)))))
                         AFRAMEADDR
                         (CPENCODE
                           (CREATE-VECTOR WBTYPE
                                          (I (AND (INDEX-VECTOR
                                                   WBTYPE
                                                   (INDEX-VECTOR WBARRAY WB
                                                                 WORDADDR)
                                                   I)
                                                  (INDEX-VECTOR
                                                   WBTYPE
                                                   (INDEX-VECTOR RBHARRAY RBHDPS
                                                                 (INDEX-VECTOR
                                                                  TSARRAY
                                                                  TS
                                                                  WORDADDR))
                                                   I))))
                           (DECODE CMF))))))
```

Figure 7: Inferred Behavior for Read

the space limitations of this paper, it would be difficult to discuss large examples such as the one we are considering without such a transliteration.

The main differences between RM2 and RM12 are: the RBH stack has now a more concrete implementation; the state representation of the RBC is more detailed; the process of searching for the most recent version of the written bits is supported by a *circular priority encoder.* We now discuss the RBH stack.

7.1.1 The RBH Stack

The RBH stack is implemented using a list of bit-vectors. We use primed names (e.g. `name'`) to distinguish this description from the earlier RBH description. Bit-vectors themselves are represented as a list of Booleans. In Miranda, we write the following type definition to describe the type of the RBH stack:

```
rbhtype' == [wbmask_type]
wbmask_type == [bool]
An example of an RBH state: [[True,True,False,True],[True,True,True,True]]
```

The `infinity` word is represented using a bit

```
infinity' = rep nframes True
For example, rep 4 True gives [True,True,True,True]. rep is defined by Miranda
```

The initial state of RBH is `[infinity']`. Since the bit pattern of words in RBH are used for masking written bits read from memory using the bit-and operation, the word `infinity'` also signifies that it will clear none of the written bits when anded.

The `update` operation is implemented as shown:

```
updater'::rbh_type'->version_type->cmf_type->omf_type->rbh_type'
updater' r d c o = infinity':(map (band dmask) r)
                       where
                       dmask
                       = (rep nframes True), if ~(lineup o d c)
                       = gendmask c d maxversion [], otherwise

band::[bool]->[bool]->[bool]
band bv1 bv2  = map and (transpose [bv1,bv2]), #bv1 = #bv2
              = error"Bitand applied to unequal-length vectors", otherwise
```

Function `updater'` uses function `band`, which takes the bit-wise and of two bit vectors. Function `gendmask` used in `updater'` generates the mask corresponding to the RBH destination, `d`. Essentially, it generates a bit vector such that the nth position of the bit vector has truth value `(member (genvers c d) n)`. In other words, if `n` is in the circular range defined by `c` and `d`, the bit value is `False`; else the bit value is `True`.

```
gendmask::cmf_type->version_type->num->wbmask_type->wbmask_type
gendmask c d n l = l, if n = (-1)
                 = gendmask c d (n-1) (~(member (genvers c d) n):l),
                       otherwise
```

Operation `index`, `time'`, and `fixw'` are implemented as shown

```
indexr'::rbh_type'->time_type->wbmask_type
indexr' r tim = r!(#r-1-tim)
```

We now describe the operations on RM2.

7.1.2 Operation s2

```
s2::rm2_type->rm2_type
s2(m,c,o,w,t,r,i,ac) =  (empty2d nframes maxwordaddr, 0, 0, initial_wb,
                         cv maxwordaddr zerofn, creater', 0, 0)
                         where zerofn i = 0
```

Operation `s2` creates an empty memory array. It resets `c` and `o` to `0`. The rollback history stack `r` is initialized through `creater'`. The rollback index register `I` is set to `0`, indicating that the current "time" is `0`. Finally, the advance counter—a local counter used during the *advance* operation—is set to `0`.

7.1.3 Operation r2

```
r2::(rm2_type,addr_type)->word_type
r2((m,c,o,w,t,r,i,ac),a)
      = ind (ind m ea) a
        where ea = aframeaddr, if zero (fixedwb w a r t c o) decc deco
                 = (cpe2num (cpe (fixedwb w a r t c o) decc deco)), otherwise
                   where decc = dec c
                         deco = dec o

fixedwb::wbstore_type->addr_type->rbh_type'->tstamp_type->cmf_type->omf_type
              -> wbmask_type
fixedwb w a r t c o = (band (ar2litems (ind w a)) (indexr' r (ind t a)))


dec::num->wbmask_type
dec d = (rep d False)++[True]++(rep (nframes-1 -d) False), if d < nframes
      = error"Decoding number outside the range nframes", otherwise
```

Operation `r2` returns data from memory `m` by first indexing `m` with a frame number called `ea`, and then indexing the frame thus retrieved with `a`, the given address. The value `ea` is the same as the version number of the archive frame, `aframeaddr`, if all the written bits end up being cleared after the operation `fixedwb`. In this case, the data is read from the archive frame. Otherwise, three things are done: function `fixedwb` is used to clear the written bits at address `a` that ought to have been cleared; a *circular priority encode* of the fixed written bits is done using function `cpe`; this value is then converted from a bit vector to an integer.

In the calculation of `ea`, we use the decoded values of the CMF and the OMF. These are calculated by the function `dec`, whose definition can be read: form a list as follows; replicate the `False` bit `d` times; then append (the `++` operator) a singleton list containing `True`; then append a list of length `nframes-1-d` containing `False`. This is a recipe for setting the `d`th position of the generated bit vector.

Function `cpe` works like an ordinary priority encoder, except that the word to be encoded is the one in the circular range defined by `c` and `o`; within this range, the priority of the bits is the highest at `c`, and decreases towards `o`. `cpe` locates the first set written bit in this circular range and outputs a `True` corresponding to its position, and outputs `False` for other positions. The implementation of `cpe` is shown in the appendix; basically its definition directly corresponds to its proposed hardware implementation.

7.1.4 Operation w2

Operation **w2** updates **m** with data **d** in the current frame **c** at address **a**. It also sets the written bits corresponding to the current frame in the written bits memory through the operation **bor**, after clearing 'stale' written bits in the written bits array through the operation **fixedwb**. In addition, the written bits are timestamped, to indicate when they were written, through the operation (upd **t a i**).

```
w2::(rm2_type,addr_type,word_type)->rm2_type
w2((m,c,o,w,t,r,i,ac),a,d)
= (u2dij m c a d, c, o,
   upd w a (l2ar (bor (fixedwb w a r t c o) (dec c))),
   upd t a i, r, i, ac)

bor::[bool]->[bool]->[bool]
bor bv1 bv2   = map or (transpose [bv1,bv2]), #bv1 = #bv2
              = error"Bitor applied to unequal-length vectors", otherwise
```

7.1.5 Operation m2

```
m2::rm2_type->rm2_type
m2(m,c,o,w,t,r,i,ac) = (m, newc, o, w, t, updater' r newc newc o, i+1,ac),
                                                          if newc ~=o
                     = error"Mark: CMF wraps around & equals OMF", otherwise
                        where newc = (c+1) mod nframes
```

Operation **m2** increments the current mark frame **c** by one, modulo **nframes**. Basically that is all that needs to be done in order to allocate a new frame. However, as discussed in section 5, the same optimization has been incorporated into **m2** as with operation **m12**:

- increment **c** by one, modulo **nframes**;
- simulate a one-step internal rollback (which decrements **c** by one);
- increment **c** by one again.

The advantages of this optimization are:

- During the *advance* operation, each location of the written bits memory **W** is subject to a *read* cycle, as opposed to a *read-modify-write* cycle that would be necessary without the optimization.
- In computations where only a fraction of all the addresses are written into over a span of **nframes** frames, not clearing the written bits can be a big win.
- Written bits are actually cleared only during a *write* operation.

The disadvantage is that the RBH stack grows by one following a *mark*; without this optimization, RBH will grow only following a *rollback*.

7.1.6 Operation b2

```
b2::rm2_type->rm2_type
b2(m,c,o,w,t,r,i,ac) = (m, (c-1) mod nframes, o, w, t, updater' r c c o,
                        i+1, ac), if c ~= o
                     = error"Rollback: CMF underflows & equals OMF", otherwise
```

Operation b2 decrements c to discard the current mark frame. It also updates the RBH stack to record which written bits ought to have been cleared following the rollback. Though b2 is defined as if it performs a one step rollback, in the actual implementation, a multi-step rollback is performed. The only difference between a multi-step rollback and a one step rollback is in the mask word used as argument to the upd operation. In the case of a one step rollback, a word with bit position c cleared is used. In a k step rollback, a mask word with bits c through c-k+1 cleared (this range is computed in a "circular mod" fashion as explained in section 4.2) are used. The verification technique changes very little, and so we study the version that is easier to follow, namely a one step rollback.

7.1.7 Operation a2

```
a2::rm2_type->rm2_type
a2(m,c,o,w,t,r,i,ac)= (archive2' m w o c r t, c,(o+1) mod nframes,w,t,r,i,0),
                      if c ~= o
                     = error"Advance: OMF overflows & equals CMF", otherwise

archive2'::m_type'->wbstore_type->omf_type->cmf_type->rbh_type'->tstamp_type
                ->m_type'
archive2' m w o c r t
 = foldr oneupd m [0..maxwordaddr]
   where
   oneupd a m
   = u2dij m aframeaddr a (ind ind_m_copy_source a),
                            if (indable ind_m_copy_source a)
   = m, otherwise
     where ind_m_copy_source = (ind m copy_source)
           copy_source = aframeaddr, if zero_cpword
                       = cp_cpword,  otherwise
                         where
                          cpword = (band (fixedwb w a r t c o)(dec o))
                          decc   = (dec c)
                          deco   = (dec o)
                          zero_cpword = (zero cpword decc deco)
                          cp_cpword   = (cpe2num (cpe  cpword decc deco))
```

For simplicity, operation a2 is implemented as if it were a one step operation. It discards the frame being pointed to by the oldest mark frame pointer o. Before discarding this frame, however, it may be the case that there are addresses for which only this frame (that is about to be discarded) has the most recent version of data. These data items are archived as before.

Significant differences between a12 and a2 can be noticed in the way copy_source is defined. In a12, we check to see whether the written bits for address a (after fixw has been performed) equals the OMF, o; if so, the OMF contents at address a has to be archived. In a2, we simply check if the result of circularly priority encoding the written bits (after fixw has been performed) results in a written bits word of all False; if so, no archiving is needed; else the location a of the OMF frame has to be archived.

8 Verification of RM2 Against RM1

As far as the external world is concerned, both RM1 and RM2 are systems whose internal states are observable only through the *read* operation. Also, both RM1 and RM2 must be first subject to a *reset* operation before any other operation can be applied on them.

Considering the above facts, the verification criterion by us states that RM1 and RM2 must be observably equivalent, defined as follows:

> Let op_i, $1 \leq i \leq N$ denote the ith operation in a sequence of N operations. Let arg_i, $1 \leq i \leq N$ denote the arguments of op_i. Let op_i^1 denote op_i as defined by RM1, and op_i^2 denote op_i as defined by RM2. Then, for any state σ_1 of RM1 and σ_2 of RM2, for all N,
>
> $$r1(op_N^1(\ldots op_1^1(s1(\sigma_1), arg_1)\ldots, arg_N), a) = r2(op_N^2(\ldots op_1^2(s2(\sigma_2), arg_1)\ldots, arg_N), a)$$

In other words, the data read from RM1 and RM2 should be the same after they both have been subject to an identical sequence of operations.

Since all possible states of the RBC can be created only through its operations *write*, *mark*, *rollback*, and *advance*, observable equivalence can be established through *generator induction*[4], *i.e.*, induction over sequences of state changing operations. Applying this technique leads to the generation of the following formulas, called *verification conditions*, or 'VCs'. (Note: The basis case for induction is included as part of the *reset* operation.) All these VCs have to be shown to be *true* for all states `S1` of RM1 and `S2` of RM2, all addresses `a` and `a'`, and all data `d`:

VC1: `r1(s1(S1),a) = r2(s2(S2),a)`

VC2: `(r1(S1,a) = r2(S2,a)) => (r1(w1(S1,a',d),a) = r2(w2(S2,a',d), a))`

VC3: `(r1(S1,a) = r2(S2,a)) => (r1(m1(S1),a) = r2(m2(S2),a))`

VC4: `(r1(S1,a) = r2(S2,a)) => (r1(b1(S1),a) = r2(b2(S2),a))`

VC5: `(r1(S1,a) = r2(S2,a)) => (r1(a1(S1),a) = r2(a2(S2),a))`

Let LHS2 stand for the left-hand side of the consequent (the part after `=>`) of any VC, RHS2 for the right-hand side of the consequent of the VC, LHS1 the left-hand side of the antecedent (the part before the `=>`) of the VC, and RHS1 the right-hand side of the antecedent. We shall now show the proof of correctness of VC1 and VC2. The other VCs have been proved similarly, but their proof not presented for lack of space.

8.1 Proof of `VC1`

The equality to be proved is

```
r1(s1(S1),a) = r2(s2(S2),a)
```

First of all, notice that `s1(S1)` can also be written as `(s1 S1)`. Expanding the left-hand side using the definitions of `r1`, we get

```
(ind (ind (m (s1 S1)) 0) a)
```

where function `m` selects the `m` component of the state triple `(m,c,o)`, at level RM1. Expanding the left-hand side using the definitions of `s1`, we get

```
(ind (ind (empty2d maxversion maxwordaddr) 0) a)      (1)
```

The right-hand side is

```
r2(s2(S2),a)                                          (1.5)
```

Let t selects the time_stamp array, t, component of the eight-tuple (m,c,o,w,t,r,i,ac). Similarly, define selector functions m, c, o, w, t, r, i, and ac. We have the following facts:

1. From the definition of s2,

   (ind (t (s2 S2)) a) = 0, for all a.

2. From the definition of s2,

   (ind (r (s2 S2)) 0) = infinity'.

3. From the definition of s2,

   (ind (w (s2 S2)) a) = (c2d maxaddr maxversion v0set), where v0set i j = (j=0).

4. band(wb,infinity') = wb, for all wb.

5. (c (s2 S2)) = (o (s2 S2)) = 0.

Expanding the right-hand side (1.5) using the above facts, and the definition of r2, we obtain

(ind (ind (m (s2 S2)) 0) a).

Now expanding using the definition of s2, we get

(ind (ind (empty2d nframes maxwordaddr) 0) a) (2)

We can assume that uninitialized memory arrays return an unspecified, but the same (in the case of RM1 and RM2) values. Therefore, we have (1) = (2) , and the proof of **VC1** is finished.

8.2 Proof of VC2

The goal is to show

LHS2 = RHS2,

that is,

r1(w1(S1,a',d),a) = r2(w2(S2,a',d),a).

Consider the left-hand side

r1(w1(S1,a',d),a).

Expanding it using the definition of r1 and w1, we get

```
  r1((u2dij (m S1) (c S1) a' d, c, (o S1)), a)
= (ind (ind (u2dij (m S1) (c S1) a' d) (c S1)) a)
```

Since the first-component of the address of ind matches that of u2dij, we simplify the above:

```
(ind (ind (u2dij (m S1) (c S1) a' d) (c S1)) a)
        = d, if (a = a')
        = (ind (ind (m S1) (c S1)) a), otherwise      (1)
```

Consider the fragment of RHS

w2(S2,a',d).

Expanding it using the definition of w2, we get

```
(u2dij (m S2) (c S2) a' d,
 (c S2),
 (o S2),
 upd (w S2) a' (l2ar (bor (fixedwb (w S2) a' (r S2) (t S2) (c S2) (o S2))
              (dec (c S2)))),
 upd (t S2) a' (i S2),
 (r S2),
 (i S2),
 (ac S2))                                                    (2)
```

Let the following abbreviations be defined

```
W1 = upd (w S2) a' (l2ar (bor (fixedwb (w S2) a' (r S2) (t S2) (c S2) (o S2))
                (dec (c S2)))),
T1 = upd (t S2) a' (i S2)
eff-mrv-frame = fixedwb W1 a (r S2) T1 (c S2) (o S2)
```

Now consider the whole RHS

```
r2(w2(S2,a',d),a).
```

Using (2), the above abbreviations, and the definition of r2, we get

```
(ind (ind (u2dij (m S2) (c S2) a' d) ea) a)
          where ea = aframeaddr, if zero (fixedwb W1 a (r S2) T1 (c S2) (o S2)) decc deco
                   = (cpe2num (cpe (fixedwb W1 a (r S2) T1 (c S2) (o S2)) decc deco)),
                                otherwise
                     where decc = dec (c S2)
                           deco = dec (o S2)                 (3)
```

Now, perform a case analysis. Consider the case a = a' on (3).

8.2.1 Case a=a'

Let us modify the above abbreviations to suit this case

```
W11 = upd (w S2) a (l2ar (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
                              (dec (c S2)))),
T11 = upd (t S2) a (i S2)
eff-mrv-frame11 = fixedwb W11 a (r S2) T11 (c S2) (o S2)
```

We then have

```
(ind W11 a) = (l2ar (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
                              (dec (c S2)))),
(ind T11 a) = (i S2)
eff-mrv-frame1  = fixedwb W11 a (r S2) T11 (c S2) (o S2)
                = (band (ar2litems (ind W11 a)) (indexr' (r S2) (ind T11 a)))
                = (band (ar2litems
                          (l2ar (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
                                     (dec (c S2))))
                        )
```

```
                (indexr' (r S2) (i S2))
        )
      = (band  (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
                    (dec (c S2)))
               (indexr' (r S2) (i S2))
        )
```

The proof seems stuck here, until we recognize that the following is a *system invariant*

```
(indexr' (r S2) (i S2)) = infinity'                                    (5)
```

In other words, as the designers of the RBC, we had arranged for the top of the RBH stack to always have the word **inf**. (The fact that this is indeed an invariant is proved as **Lemma1**, below.) Using this invariant, simplify **eff-mrv-frame1** to

```
eff-mrv-frame1 -- which is also equal to (fixedwb W11 a (r S2) T11 (c S2) (o S2))

      = (band  (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
                    (dec (c S2)))
               infinity'
        )
      = (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
             (dec (c S2)))
```

Since **eff-mrv-frame1** has a subterm (dec (c S2)) that gets bit-wise or-ed with the rest of the term, we can conclude that the following test used in equation (3), specialized according to case a=a', is false:

```
if zero (fixedwb W11 a (r S2) T11 (c S2) (o S2)) decc deco  --> reduces to false
```

We replace

```
(fixedwb W1 a (r S2) T1 (c S2) (o S2))
```

with

```
 (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
      (dec (c S2)))
```

in equation (3), giving us:

```
r2(w2(S2,a,d),a)   = (ind (ind (u2dij (m S2) (c S2) a d) ea) a)
                       where
                       ea = (cpe2num (cpe
                                      (bor (fixedwb (w S2) a (r S2) (t S2) (c S2) (o S2))
                                           decc)
                                      decc
                                      deco)
                             )
                             where decc = dec (c S2)
                                   deco = dec (o S2)                   (6)
```

We use **Lemma2** (proved later) which asserts:

```
(cpe (bor v decc) decc deco) = decc
```

and so

```
(cpe2num (cpe (bor v decc) decc deco)) = (c S2)
```

and so

```
r2(w2(S2,a,d),a)    = (ind (ind (u2dij (m S2) (c S2) a d) (c S2)) a)
                    = d.                                                    (7)
```

Notice that (1) also yields d, when a=a'. Thus VC2 holds for a=a'.

8.2.2 Case a <> a'

Using the definitions of r1 and w1, and using array axioms, LHS2 can be simplified to

```
(ind (ind (m S1) (c S1)) a)
```

Since the addresses a and a' do not match, we can simplify RHS2 to

```
(ind (ind (m S2) ea) a)
          where ea = aframeaddr, if zero (fixedwb W1 a (r S2) T1 (c S2) (o S2)) decc deco
                   = (cpe2num (cpe (fixedwb W1 a (r S2) T1 (c S2) (o S2)) decc deco)),
                                otherwise
                     where decc = dec (c S2)
                           deco = dec (o S2)                                (8)
```

LHS1 is, using the definitions of r1 and w1,

```
(ind (ind (m S1) (c S1)) a)
```

RHS1 can be expanded using the definition of r2 to exactly what RHS2 is. Thus, this case of a <> a' is also established. This finishes the proof of correctness of the *write* operation, with respect to the *read* operation.

8.2.3 Proof of Lemma1

We shall use a technique known as *generator induction* [4]. We notice that RBH is affected by operations s2, m2, and b2 only. Operation s2 initializes i to a value "i0", which is 0. It also initializes the RBH unit to creater'. We notice that creater' has an infinity' word on top; *i.e.*, indexr'(creater',i0)=infinity'. Thus, the basis case for the induction holds. After operation m2, i is increased by 1, to attain a value that we shall call "i1", and the RBH state advances to (updater' r newc newc o). From the definition of updater', we can see that it increases the length of the RBH by one and also pushes an infinity' word on top of the RBH stack. Therefore, the value of indexr'(r,i1) is also infinity'. Similar state updates are performed on RBH after operation b2 also. Thus, in all cases, the RBH unit has as its "top" entry (entry pointed to by i) the value infinity'. The above arguments establish the induction step. Thus, **Lemma1** is established.

8.2.4 Proof of Lemma2

To show

```
(cpe (bor v decc) decc deco) = decc
```

Informally, this is true because function **bor** (bit-wise or) sets the bit at position **decc**, which will be detected as the first set bit by the circular priority encoder function **cpe**, when it is scanning from decc towards deco. More formally, from the definition of **cpe**,

```
cpe wb dc do  = map pickcpo [(cpencode i dc do wb) | i<-[0..maxversion]]
                where pickcpo ((pout,inrng),(kout,cpout)) = cpout
```

We can see that in the above list-comprehension, **cpencode** will be called as

```
(cpencode i decc deco (bor v decc))
```

for i in the range `[0..maxversion]`.

From the way cpencode is defined, we can see:

1. the ith `cpcell` produces a **cpout** bit set to **true** if **din** is true, **inrng** is true, and **c** is true. (This stems from the clause

   `where cpout = din & inrng & (~kin\/c)` of the definition of **cpcell**, below.

2. Once cpout is set, kout is set true, thus killing the circular carry.

3. Thus, once the cell numbered **decc** has produced the **cpout** signal, the remaining cpout signals are reset to **False**. Thus, the lemma holds.

See the appendix for further details.

9 Conclusions

The design of the Roll Back Chip (RBC), a custom architecture that helps speed up the state-saving and roll-back parts of an implementation of distributed discrete event simulation using Time Warp, was presented. Specifications for the RBC system were written in our hardware description language HOP. Top-down design was followed to some extent, but many of the crucial refinement steps were simply based on the experience of one of the authors as an architect, and captured after the fact by the other author whose primary role has been that of formal specification. The final implementation of the RBC used in [2] is very close to the final design specification in HOP that was verified against a high-level specification of the RBC. Verification is greatly aided by the ability of PARCOMP to deduce behavioral descriptions from structural descriptions.

9.1 Lessons Learned

The exercise of designing the RBC system, writing its specification, and conducting its formal verification have taken nearly three years of part-time effort on part of both the authors.. This time-frame is not representative of how verification will ultimately be used in the industry. The reason for taking three years is mainly because we were developing both the specification of the RBC and the implementation of HOP side by side. This time period can be drastically reduced once sufficient support tools are developed. All these tools have well-understood functionality. Some of them are outlined below.

In the authors' estimate, the percentage of time allocated to various tasks is as follows:

Paper design: About six man-months and two graduate seminar classes (one quarter each) later, we had the initial specifications for the RBC system, and a simulator written in the C language;

Design iterations: Our initial design involved an on-chip written bits memory. This design was simulated in C. We concluded that this scheme would involve the use of large amounts of silicon area. Hence we developed a version of the RBC based on a special purpose Cache memory [1]. This took about three man-months.

Developing the Simplified Version: Following the commercial development of the RBC [2], we decided to try and verify this version. These decisions cost us a few man-months.

Developing HOP Specifications: Since the HOP effort was also going side-by-side, "teething troubles" forced us to take nearly six man-months to develop HOP specifications, and run them through PARCOMP.

Verification: Another two man-months were spent in writing out a formal proof of correctness of the inferred behavior.

Developing Miranda Specifications: For the purposes of exposition, it was felt that the HOP syntax was a bit tedious. Also, developing Miranda specifications has the added advantages of the ease of making the RBC specifications available for other groups to try their approaches on. This took only one week, thanks to the expressive power of Miranda.

The rest of the time was spent in coordinating between the authors through electronic mail.

9.2 How to do it again? Future Plans

We are convinced that formal verification is here to stay as a powerful technique to supplant other design validation techniques. Here are our concrete plans to make our future efforts in this area easier:

- *Re-implement HOP and PARCOMP:* The HOP system is currently being re-implemented, using the Standard ML functional language [18] as the implementation language. ML has nearly all the expressive power of Miranda, and also comes with much a more efficient compiler than Miranda. This will help in many ways, including the ability to present our results in the syntax of HOP itself.

- *Automate low-level reasoning steps:* We are planning to write an expression simplifier as part of PARCOMP to help carry out many of the proof-steps under human guidance.

- *Capture timing details:* The RBC system offers many possibilities for the overlapped execution of its operations. In the current design, these possibilities have not been fully exploited. Our notation is also presently incapable of *formally* expressing many of these operational details pertaining to resource contention, overlapped execution, synchronization, etc.

  Fortunately for us, we have developed a process+functional language called hopCP. A simulator as well as PARCOMP has been implemented for it by the first author's group [26, 27]. Plans are in place to specify the RBC in hopCP, conduct its formal verification, and develop an asynchronous version of the RBC.

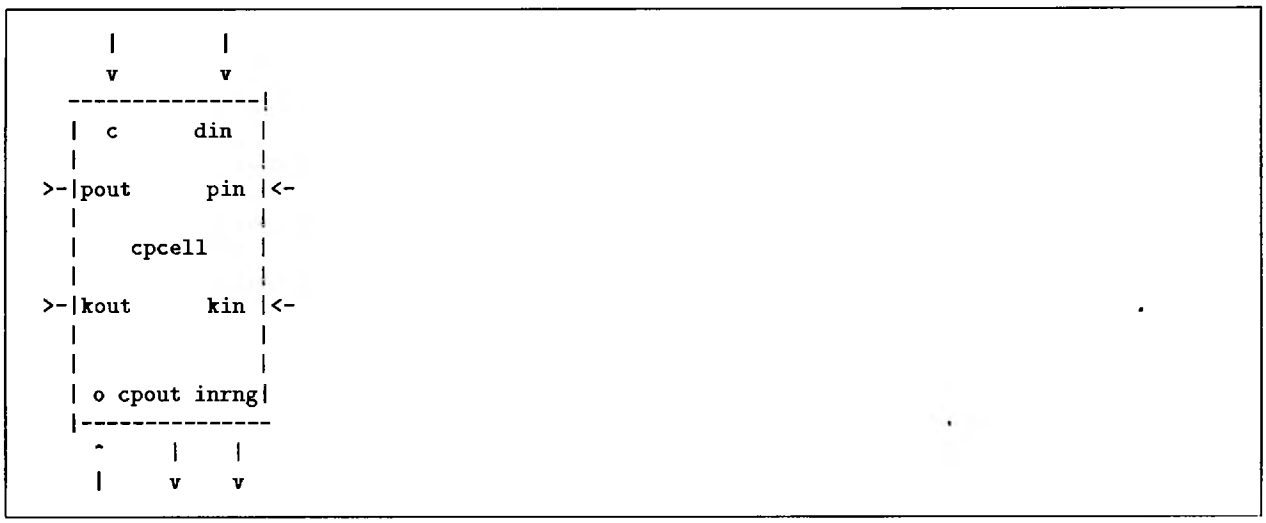
A The Specification of the Array Data Type

```
maxind_type == num
ind_type    == num

abstype array *
with cv::maxind_type -> (ind_type -> *) -> array *

     empty::maxind_type -> array *

     indok::array * -> ind_type -> bool

     maxind::array * -> maxind_type

     ar2l::array * -> [(ind_type,*)]

     ar2litems::array * -> [*]

     l2ar::[*] -> array *

     upd::array * -> ind_type -> * -> array *

     ind::array * -> ind_type -> *

     indable:: array * -> ind_type -> bool

     showarray::(*->[char]) -> array * -> [char]

array * == (maxind_type,[(ind_type,*)])

empty m = (m,[])

cv m f   = (m,[(i,f i) | i<-[0..m]])

indok a i = i<=(maxind a) & i>=0

maxind (m,l) = m

ar2l(m,l) = l

ar2litems(m,l) = map snd l

l2ar l = (#l-1,zip2 (index l) l)

upd a i e = (maxind a,(i,e):[(x,y)|(x,y)<-(ar2l a); x ~= i]), if indok a i
          = error ("Index "++(shownum i)++" out of range "),   otherwise

ind a i
        = hd projecti, if (indok a i) & projecti ~= []
        = error "Indexed item not found", if projecti = []
        = error ("Index "++(shownum i)++" out of range "), otherwise
          where
          projecti = [y | (x,y)<-(ar2l a); x=i]

indable a i = ([y | (x,y)<-(ar2l a); x=i] ~= [])

showarray f (m,l) =  "(" ++ (shownum m) ++ ",[])", if l=[]
```

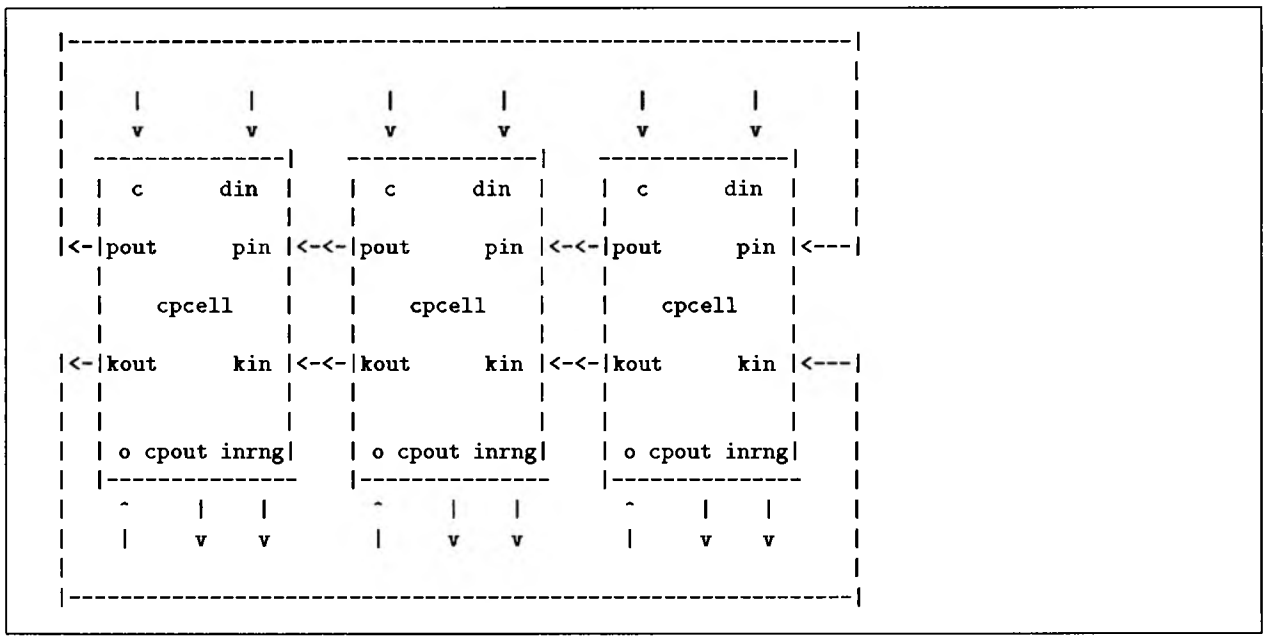
```
                = "(" ++ (shownum m) ++ ",\n[" ++
                   concat(map shc (init l))   ++
                   (sh (last l))              ++   "]\n)\n", otherwise
                    where
                    sh (x,y) = "(" ++ (shownum x) ++ "," ++ (f y) ++ ")"
                    shc x = (sh x)++","


|| Create a 2d array
empty2d::num->num->(array (array *))
empty2d d1 d2 =
               cv d1 f_i
               where
               f_i i = empty d2

c2d::num->num->(num->num->*) -> (array (array *))
c2d d1 d2 f_i_j =
               cv d1 f_i
               where
               f_i i = cv d2 f_j
                       where
                       f_j j = f_i_j i j

|| Update 2d array at i, for all j with (f j)
u2di::(array (array *))->num->(num->*) -> (array (array *))
u2di a i f =
            (upd a i new_islice)
            where
            new_islice =
                       (cv (maxind old_islice) f)
                       where
                       old_islice = (ind a i)

|| Update 2d array at j, for all i with (f i)
u2dj::(array (array *))->num->(num->*) -> (array (array *))
u2dj a j f =
            (iter 0 a)!(maxind a)
            where
               iter i x = x:(iter (i+1) (g i x))
                          where
                          g i x = u2dij x i j (f i)

|| Update 2d array at i,j with v
u2dij::(array (array *))->num->num->* -> (array (array *))
u2dij a i j v = (upd a i (upd (ind a i) j v))
```

B The Circular Priority Encoder Specification

Imagine a hardware implementation of "cpcell", as follows:



Then, replicating the cells as shown nframes times, with a wrap-around connection from the most significant pin and kin to the least significant pout and kout, constitutes the cpencode hardware.



Above, we show a three-bit circular priority encoder.

Notice that there is a combinational loop! However, the evaluation of this loop terminates, thanks to the use of many non-strict operators. (E.g. `and(False,anything) = False`.) Since Miranda is lazy, we can directly model this combinational loop in it! The definitions, leading up to the `cpe` function are now presented. First, define a "range computation cell" that is contained within the `cpcell`:

```
rngcell (c,pin,o)            || outputs are (pout,inrng) -this is a 'helper'
                = (True,True), if c&~o
                = (pin ,pin) , if (~c)&(~o)
                = (False,pin), if ~c&o
                = (False,True),if c&o
```

Now define the `cpcell`:

```
cpcell (c,din,pin,kin,o)      || outputs are ((pout,inrng),(kout,cpout))
            = ((pout,inrng),(kout,cpout))
              where cpout         = din & inrng & (~kin \/ c)
                    kout          = cpout
                    (pout,inrng) = rngcell(c,pin,o)
```

Now write cpencode, that returns the outputs for the i-th position. Here, decc is the decoded CMF, deco the decoded omf, and dinl the data input list.

The outputs of this function are:

- pout_i = the propagate signal generated for the inrng signal generation;
- inrng_i = the signal that tells whether the current bit i is within the circular range defined by CMF and OMF;
- kout_i = the kill signal generated by the i-th stage;
- cpout_i = the circularly priority encoded signal.

```
cpencode i decc deco dinl || gives ((pout_i,inrng_i),(kout_i,cpout_i))
      = cpcell(c_i,din_i,pin_i,kin_i,o_i)
        where
          ((pin_i,inrng_i'),(kin_i,cpout_i'))
                          = cpencode ((i+1) mod nframes) decc deco dinl
          (c_i,o_i,din_i) = (zip3 decc deco dinl)!i
```

Notice how the recursion proceeds. It won't "chase the tail" because, very soon, a term of the form and(False,X), and(X,False), or(True,X), *or* or(X,True) will be encountered, thus terminating the "tail chasing".

A simulation session further clarifies how this function operates:

```
Miranda cpencode 0 [False,False,True,False][True,False,False,False][False,True,False,False]
((False,True),(False,False))
Miranda cpencode 1 [False,False,True,False][True,False,False,False][False,True,False,False]
((True,True),(True,True))
Miranda cpencode 2 [False,False,True,False][True,False,False,False][False,True,False,False]
((True,True),(False,False))
Miranda cpencode 3 [False,False,True,False][True,False,False,False][False,True,False,False]
((False,False),(False,False))
```

References

[1] Richard Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan. Design and performance of special purpose hardware for time warp. In *15th Annual International Symposium on Computer Architecture, Honolulu*, pages 401–408, 1988.

[2] C. A. Buzzell, M. J. Robb, and R. M. Fujimoto. Modular VME rollback hardware for Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):153–156, January 1990.

[3] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[4] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[5] Boyer and Moore. *A Computational Logic.* Academic Press, 1979.

[6] Michael Gordon. HOL: A proof generating system for Higher Order Logic. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

[7] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, (9), September 1990.

[8] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *Computing Surveys*, 14(2):159–192, June 1982.

[9] Krzysztof R. Apt and Ernst-Rudiger Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991. *ISBN 0-387-97532-2.*

[10] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE Computer*, 21(7):8–20, July 1988.

[11] G.Birtwistle and P.A.Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[12] Luc Claesen, editor. *Proceedings of the IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium.* North Holland, November 1989.

[13] Daniel Weise. *Automatic Formal Verification of Synchronous MOS VLSI Designs.* PhD thesis, Dept. of EE and CS, MIT, 1986.

[14] Randal E. Bryant. Formal verification of memory circuits by switch-level simulation. *IEEE Transactions on Computer-Aided Design*, 10(1):94–102, January 1991.

[15] Avra Cohn. Correctness properties of the Viper block model: The second level. In G.Birtwistle and P.A.Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 1, pages 1–91. Springer-Verlag, 1989.

[16] Warren A. Hunt Jr. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs.* Elsevier Science Publishers B.V. (North Holland), 1987. (Proc of the IFIP WG 10.2 Working Conference with the same title.).

[17] David A. Wood, Garth A. Gibson, and Randy H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design & Test of Computers*, August 1990.

[18] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991. ISBN 0-521-39022-2.

[19] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[20] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[21] Mani Narayana and Surya Mantha. The design of a tlb for the roll back chip. VLSI Class Project Report, Dept. of Computer Science, Univ. of Utah, Winter 1988.

[22] Paul Hudak. Conception, evolution, and application of functional programming languages. *acm Computing Surveys*, (3):359–411, September 1989.

[23] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer System Sciences*, 17:348–375, 1978.

[24] David A. Turner. *Functional Programs as Executable Specifications, edited by C.A.R. Hoare and J.C.Shepherdson.* Prentice-Hall International Series in Computer Science, 1985.

[25] John V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[26] Venkatesh Akella. Action refinement based transformation of concurrent processes into asynchronous hardware. Ph.D. research in progress.

[27] Vankatesh Akella and Ganesh Gopalakrishnan. "Hierarchical Action Refinement: A Methodology for Compiling Asynchronous Circuits from a Concurrent HDL". In D. Borrione and R. Waxman, editors, *CHDL 91 - Computer Hardware Description Languages and their Application*, pages 319–338, Marseille, France, April 1991.