Verification of Regular Arrays by Symbolic Simulation

PRABHAT JAIN[1]
GANESH GOPALAKRISHNAN[2]
PRABHAKAR KUDVA

UUCS-91-022

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

October 28, 1991

Abstract

Many algorithms have an efficient hardware formulation as a regular array of cells, which can be implemented in VLSI as regular circuit structures. Bit-sliced microprocessors, pattern matching circuits, associative cache memories, fine-grain systolic arrays, and embedded memory-with-logic structures are representative of the regular array design style. In this paper, we illustrate a verification approach for regular arrays. Our approach for the verification of regular arrays combines formal verification at the high level and symbolic simulation at the low level(e.g., switch-level). The verification approach is based on a simple hardware specification formalism called HOP, a parallel composition algorithm for regular arrays called PCA, and a switch-level symbolic simulator(e.g., COSMOS). We illustrate our verification approach on the Least Recently Used(LRU) priority algorithm implemented as a two-dimensional array of LRU cells in VLSI. We also show a new technique of encoding input constraints as parametric boolean expressions on inputs to reduce the number of symbolic simulation vectors required for verification. The use of this technique in LRU array verification results in the simulation of only one *symbolic simulation vector independent of the size of the LRU array.*

[1]Supported in part by the University of Utah Graduate Research Fellowship
[2]Supported in part by NSF Award MIP-8902558

Verification of Regular Arrays by Symbolic Simulation

PRABHAT JAIN* (*jain@cs.utah.edu*)
GANESH GOPALAKRISHNAN† (*ganesh@bliss.utah.edu*)
PRABHAKAR KUDVA (*pkudva@cs.utah.edu*)

University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112

Keywords: Symbolic Simulation, Formal Verification of VLSI, Regular Array Verification, Input Constraints, Parametric Boolean Expressions

Abstract. *Many algorithms have an efficient hardware formulation as a regular array of cells, which can be implemented in VLSI as regular circuit structures. Bit-sliced microprocessors, pattern matching circuits, associative cache memories, fine-grain systolic arrays, and embedded memory-with-logic structures are representative of the regular array design style. In this paper, we illustrate a verification approach for regular arrays. Our approach for the verification of regular arrays combines formal verification at the high level and symbolic simulation at the low level(e.g., switch-level). The verification approach is based on a simple hardware specification formalism called HOP, a parallel composition algorithm for regular arrays called PCA, and a switch-level symbolic simulator(e.g., COSMOS). We illustrate our verification approach on the Least Recently Used(LRU) priority algorithm implemented as a two-dimensional array of LRU cells in VLSI. We also show a new technique of encoding input constraints as parametric boolean expressions on inputs to reduce the number of symbolic simulation vectors required for verification. The use of this technique in LRU array verification results in the simulation of only* one *symbolic simulation vector independent of the size of the LRU array.*

1 Introduction

Regular arrays and VLSI technology have a close connection: regular circuit structures in VLSI allow easy and area-efficient implementation of regular arrays. Regular structures are an important part of any VLSI methodology. One advantage of these regular structures is that they enable one to increase the regularity factor [13] of the design. For design productivity, it pays to have large regularity factor; it also pays from a layout viewpoint [7].

Many algorithms have an efficient hardware formulation as a regular array of cells and they can be implemented as area-efficient VLSI circuits. Logic and memory can be intermixed to build large regular structures. This mixing is particularly easy in the VLSI circuit domain. Examples of regular array designs include associative cache memories, pipelined multipliers, memory-with-logic structures, convolvers, and fine-grain systolic arrays. Iterative Logic Array(ILA)s of combinational and sequential cells form the basis of bit-sliced microprocessors and other easily testable regular designs

*supported in part by the University of Utah Graduate Research Fellowship.
†Supported in part by NSF Award MIP-8902558

[16]. With regular array designs being employed in numerous applications, as mentioned above, the verification of regular arrays becomes an important step in their design and implementation as VLSI circuits.

With the increase in the complexity of VLSI circuits, the use of a Hardware Description Language(HDL) in VLSI design is becoming necessary. Similarly, formal verification of VLSI circuits is becoming an essential step in the design of many large and complex VLSI circuits to ensure the correctness of these designs. The use of an HDL in VLSI design helps in the formal verification of designs and offers several other advantages. For instance, an HDL description can serve as an unambiguous specification of the design and facilitate easy integration with the design tools [17]. Several formal verification approaches have been suggested for the verification of digital circuits, but current formal hardware verification approaches cannot accurately model low-level circuit details. Since the simulators(e.g., switch-level) can model low-level circuit details accurately, an approach combining the capabilities of formal verification at the high level and symbolic simulation at the low-level can derive the advantages of both the approaches.

Bryant has proposed symbolic switch-level simulation for formal hardware verification [4]. His verification approach has been applied to verify a static RAM, data paths, and pipelined circuits [2, 3]. The combination of formal verification at the high-level and simulation-based verification at the low-level has been proposed in [9, 15]. Our verification approach for datapath and control circuits is based on a simple hardware specification formalism called HOP, a parallel composition algorithm called PARCOMP, and a switch-level simulator(COSMOS). The details of this verification approach are discussed in [9].

In this paper, we illustrate our verification approach for regular arrays which is based on HOP, PCA–a parallel composition algorithm for regular arrays, and COSMOS. We use the Least Recently Used(LRU) priority algorithm, implemented as a two-dimensional array of LRU cells in VLSI, as an example to illustrate our verification approach. In order to reduce the symbolic simulation effort, a new technique to encode the input constraints as parametric boolean expressions on inputs is incorporated in our verification approach. This encoding technique reduces the number of symbolic simulation vectors required for simulation; it also reduces the corresponding verification effort. The validity of this encoding technique relies on a property of the symbolic simulator similar to the monotonicity property of a ternary simulator. In the LRU verification, this technique reduces the number of symbolic simulation vectors required to *one*, independent of the LRU array size.

It is important to develop efficient ways to handle input constraints for the verification of regular arrays, because many regular arrays are designed to be operated under input constraints (*e.g.*: "inputs must be unary"). Designing regular arrays to operate under input constraints is a frequently employed circuit saving measure (*e.g.* internal decoders can be avoided). In such cases, it is the responsibility of the submodule that provides the inputs to guarantee that the constraints are never violated. This, in turn is often guaranteed by *circuit state invariants* that disallow the submodules from going into many of their state combinations.

1.1 Outline of Paper

The remainder of the paper is organized as follows. Section 2 outlines our verification approach for regular arrays. It also discusses a new technique of using parametric boolean expressions on inputs to encode input constraints, and reduce the simulation effort. Section 3 explains the LRU algorithm and its hardware implementation as a two-dimensional array of LRU cells. Section 4 shows the verification of the LRU algorithm using our verification approach and presents the results. Improvement in verification time with the use of our encoding technique is also reported. Section 5 concludes the paper and reports our ongoing effort in extending the verification approach discussed in this paper.

2 Verification Approach

Our verification approach for regular arrays combines formal verification at high-level and symbolic simulation at low-level to derive the advantages of both, in the framework of a simple hardware specification formalism called HOP. Formal verification is shown to be an effective technique for the verification of regular arrays at high level. Formal verification can provide important information about the circuit, such as *invariants*, to facilitate circuit verification at low level using symbolic simulation.

2.1 Overview of HOP

The language HOP supports the specification and functional simulation of hardware designs, and also assists in formal verification [10, 8]. In order to use HOP for verification, a reference behavioral specification (desired behavior) for the circuit is first written in HOP. This specification consists of a collection of *transitions*. Each transition of a HOP specification specifies the *present state* of the system, *control inputs* under which the transition is taken, *data input* values consumed by the system if the transition is taken, a boolean *guard* which must be true for the transition to be taken, *control outputs* generated when the transition is taken, *data outputs* generated when the transition is taken, and the *next state* attained. States, inputs, and outputs are modeled symbolically, *i.e.*, involving variables that range over bit vectors. In this paper, we leave out the guards in the transitions, in order to simplify PCA (to be discussed below). However, the prohibition of guards does not restrict the class of regular arrays that can be modeled, because the equivalent effect can be had through the use of *if_then_else* functions in specifying the next-state and the outputs.

Following the syntax suggested in [4], we write each HOP transition in the form shown in equation 1:

$$initial \quad \{actions\} \quad result \tag{1}$$

where *initial* specifies the initial system state, *action* specifies the control and data inputs applied, and *result* specifies the control and data outputs generated, and the next state attained. We shall refer to HOP transitions also as *transition assertions.*

A structural description of the circuit design is then written. A structural description consists of transition assertions for the *submodules* used, and a *netlist* specifying their interconnections. For the purposes of this paper, we shall stick to structural descriptions that describe rectangular arrays, where adjacent cells can be connected in the x direction or the y direction only. In addition, we also allow embedded "feed-through connections", by which values can be broadcast to all the cells along the x direction or along the y direction. Feed-through connections are also useful for realizing embedded busses. We do not consider this use of feed-through connections. Therefore, feed-through connections are assumed to be inputs only.

2.2 PCA: Parallel Composition Algorithm for Regular Arrays

A behavioral description can be derived from a regular array description written in HOP by employing an algorithm called PCA. This algorithm works as follows.

We are given:

1. one cell of the array in question (all cells are assumed to be identical);
2. a general description of the connectivity through recurrence relations of the general form[1]:
$$\forall\, i \in (XSIZE - 1)\,.\, j \in YSIZE\,.\, connect(cell[i,j].portp, cell[i+1,j].portq)$$
$$\forall\, i \in XSIZE\,.\, j \in (YSIZE - 1)\,.\, connect(cell[i,j].portp', cell[i,j+1].portq')$$
3. appropriate boundary conditions for the recurrence relations describing the connectivity;
4. description of the global connectivity through formulae of the form
$$\forall\, i \in XSIZE\,.\,(\forall\, j \in YSIZE\,.\, connect(cell[i,j].portp'', globalport[i]))$$
$$\forall\, j \in YSIZE\,.\,(\forall\, i \in XSIZE\,.\, connect(cell[i,j].portp''', globalport'[j]))$$
5. the array does not manifest any combinational cycles in any of its states of execution. (This is checked as described below.)

Each cell is described through a collection of possible transitions. An example is given in figure 2(b). The transitions captured in this example are:

$$\begin{aligned} State &= dps\ \ \{Iclkrise \wedge cin = col@ \wedge rin = row@\} \\ State &= (And\ (Or\ rin\ dps)\ (Not\ cin)) \qquad (2) \\ State &= (And\ (Or\ rin\ dps)\ (Not\ cin))\ \ \{Iclkfall \wedge w = ?w\} \\ State &= (And\ (Or\ rin\ dps)\ (Not\ cin))) \wedge (!e = (Or\ w\ dps) \qquad (3) \end{aligned}$$

[1]To simplify our notations, we use the convention from set-theory of treating numbers as sets: 0 is the empty set, and N, where $N > 0$, is the set containing 0 through $N - 1$. We also use the notation $cell[i,j].x$ to select aspect x (port or state) of $cell[i,j]$.

Here, *dps* stands for "data path state". We adopt certain port naming conventions, to enhance readability: $?p$ is an input port name, $!p$ an output port name, and $p@$ a global ("feed-through connected") port name. Notice that each transition essentially specifies, for each "step" of the computation, the next-state and output equations.

The steps in the PCA Algorithm are the following:

1. Describe each transition describing a cell through equations of the form

$$next\_state_i = f(present\_state, inputs)$$

$$output\_port_j = g(present\_state, inputs)$$

   where i and j range over all the state bits and the output ports, respectively.

   In case of the LRU, the next-state equation derived from the first transition (ignoring the clock input which is assumed to be global) is:

$$next\_state = (And\ (Or\ rin\ dps)\ (Not\ cin))$$

   The equation for the output on $!e$ is not of interest during the first transition. (It can be taken to be incompletely specified, or undefined.) The equation for the output on $!e$ for the second transition is

$$!e = (Or\ w\ dps).$$

2. Express the connectivity as described above. In case of the LRU, ports $!e$ and $?w$ connect, and the rest are feed-through connections.

3. Repeat the following steps for each transition, and its corresponding equations.

4. Replace the names of local ports used in the transition that are connected to a global port by the name of the global port. (In the LRU example, substitute in $col@$ and $row@$.)

5. Find out the dependency between $cell[i,j].port$ and: (i) $cell[i+1,j].port'$; or (ii) $cell[i,j+1].port''$, as follows. Treat connectivity equations as rewrite rules, and rewrite the port names used in transition equations using these rewrite rules. For the LRU, we get

$$cell[i,j+1].!e = (Or\ \ cell[i,j].!e\ \ cell[i,j+1].dps)$$

6. In general, in this rewriting process, two cases can arise:

   (a) The dependencies are acyclic (the normal case), as with the LRU: no node of a cell depends on itself;

   (b) A node of a cell depends upon itself. This can arise if two adjacent cells, or a two-by-two square of cells introduce a cycle that straddles them. When this is detected, PCA is aborted.

7. Generalize the dependencies. We have shown that [12] the dependencies of all regular arrays are expressible using one of the following higher order functions: *foldr*, *map*, *iterate*. We illustrate the use of *foldr* below. *foldr* takes a two-ary function *fn*, a boundary value *bv*, and a vector of items *vec*, and returns an item as follows:

   $$foldr(fn, bv, vec) = fn(vec[0], fn(vec[1], \ldots fn(vec[max], bv) \ldots)).$$

   An example, where $[1,2,3]$ denotes a vector, is:

   $$foldr('+', 0, [1,2,3]) = (1 + (2 + (3 + 0))).$$

   For the LRU, the $!e$ port output at the right boundary of the cell is obtained through generalization, as

   $$cell[i, YSIZE - 1].!e = foldr(Or,\ cell[i,0].?w,\ slice(cell[i,j].dps, i)) \tag{4}$$

   where $slice(cel[i,j].dps, i)$ returns the vector of *dps* values along the ith row. Similarly, PCA obtains generalized next-state equations for the entire array; for the LRU array, it is:

   $$cell[i,j].next\_dps = (And\ (Or\ \ rin[i]\ \ cell[i,j].dps)(Not\ \ cin[j])) \tag{5}$$

The fact that behaviors can be so generalized has been mentioned in [19]. We have implemented PCA in Standard ML of New Jersey, and it generalizes the behaviors of a very large class of regular arrays. An interesting feature of our algorithm is that it automatically constructs (in the *lambda* notation) the two-ary function to be used in the *fold* operation. It checks for cycles using a graph representation of the connectivity. (Details are skipped.) The inferred behavior of the LRU array that we use in the rest of this paper was automatically obtained using PCA.

2.3 Formal Verification

There are three levels at which formal verification can be conducted: verify that the requirements have been realized by an abstract data type model; verify that the abstract data type model is implemented correctly by the chosen regular array organization; and finally, verify that the transistor circuit realizes the regular array's intended functionality. In case of the LRU, these steps are: (i) LRU policy .vs. LRU array operations (*e.g.*, as reported in [14]); (ii) LRU array operations .vs. LRU regular array behavior determined by PCA; and (iii) PCA inferred behavior .vs. transistor circuits. We now touch upon (ii) in this section, and then detail (iii), which is the problem attacked in this paper.

In the verification problem numbered (ii), above, the abstract data type model of the system is compared against the inferred behavior of the cell array as follows:

- Discover a *homomorphism* h : `states_of_inferred_behavior` $\longrightarrow$ `states_of_adt_model`,

b1	b2	in1	in2	in3	in4
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0
Parametric Boolean Expressions		$b1 \wedge b2$	$b1 \wedge \neg b2$	$\neg b1 \wedge b2$	$\neg b1 \wedge \neg b2$

Figure 1: Input Constraints and Parametric Boolean Expressions

- For any sequence of operations s that begins with the `reset` operation, show that sequence s, when applied to the implementation (characterized by the derived behavior) results in state S', and sequence s when applied to the specification results in state S such that $h(S') = S$.
- Show that when the implementation is in state S', and the specification is in state S, such that $h(S') = S$, they (the implementation and the specification) produce identical outputs.

The above technique is routinely used in verification (e.g. [1, 6, 11]).

2.4 Handling Input Constraints

Every circuit relies on some input constraints for its correct operation. The circuit is expected to operate correctly when operated under the constraints it is designed for, and may malfunction if input constraints are violated. Thus, a circuit needs to be verified only for inputs satisfying the input constraints (unless it is required to know the circuit behavior under anomalous inputs, which we are not interested in). As mentioned in section 1, most regular arrays come with their own input constraints. Typical examples include arrays whose rows and/or columns can be selected (*e.g.* "only one row/column must be selected at a time"), arrays embedded within a larger module where the submodule that provides the input to the array guarantees its input constraints, arithmetic circuits employing various number encoding schemes, *etc.*

In our verification approach, we use switch-level symbolic simulation to verify regular arrays at low-level. We use parameterized boolean expressions on the inputs to encode the input constraints of the regular array being verified. The use of this new technique offers several advantages: it reduces the number of symbolic simulation vectors required, and thus reduces the simulation effort. It also reduces the corresponding verification effort. In addition, this technique can be applied when the modules of a large system are verified separately; the interface constraints of the module being verified separately can be encoded as parametric boolean expressions on the inputs of the interface, for efficient verification. This technique can also be applied to encode constraints among the state variables of the circuits (*e.g.* "the value in register r_1 must be greater than that in register r_2", *etc.*.)

To illustrate this technique, consider a circuit with four inputs `in1`, `in2`, `in3` and `in4` and the input constraint that one and only one input be 1 at any time for its correct operation. The four valid combinations of values of the inputs **in1, in2, in3** and **in4** are shown in Figure 1. This input constraint can be encoded using the two parameter boolean variables, say, **b1** and **b2**, such that

$$\begin{aligned}&(in1 \wedge \neg in2 \wedge \neg in3 \wedge \neg in4) \vee (\neg in1 \wedge in2 \wedge \neg in3 \wedge \neg in4) \vee \\ &(\neg in1 \wedge \neg in2 \wedge in3 \wedge \neg in4) \vee (\neg in1 \wedge \neg in2 \wedge \neg in3 \wedge in4) \\ &= \exists b1\, b2\,. \\ &\quad ((in1 = b1 \wedge b2) \wedge (in2 = b1 \wedge \neg b2) \wedge \\ &\quad (in3 = \neg b1 \wedge b2) \wedge (in4 = \neg b1 \wedge \neg b2))\end{aligned}$$

where $\mathcal{F} = \mathcal{G}$ means that $\mathcal{F}$ and $\mathcal{G}$ are logically equivalent.

Now, a parametric boolean expression can be obtained for each of the inputs as shown in Figure 1. These parametric boolean expressions capture the input constraint of the circuit. (Related techniques for obtaining parametric solutions of boolean equations have been reported in [5].)

In general, given n inputs $i_1, i_2, \ldots, i_n$, and the input constraint which results in m valid combinations of values of these inputs, the number of parametric boolean variables required to encode the input constraint is $\lceil \log_2 m \rceil$. The validity of the use of parametric boolean expressions, encoding the input constraints, on the inputs of the circuit for symbolic simulation relies on the following property of the symbolic simulator (roughly analogous to the monotonicity property assumed in [4]): Suppose the circuit in question is simulated with a scalar input i (ground value) that satisfies the input constraints, and the simulation results (next-states and outputs) be (s_1, o_1). Suppose the same circuit is now simulated with the parametric boolean expressions e at its inputs, and the simulation results thus obtained be (s_2, o_2). Then, for those instantiations σ such that $e\sigma = i$, we have $(s_2\sigma, o_2\sigma) = (s_1, o_1)$.

We derive the parametric boolean expressions for the inputs based on the input constraints of the regular array and use them in the symbolic simulation of the regular array.

2.5 Summary of the Verification Approach

In summary, the outline of the combined verification approach is given below. Many of the following steps have not been automated yet, but they are currently being automated.

1. Write the behavioral specification of the regular array in HOP.
2. Write the HOP description of a cell of the regular array.
3. Write a structural description in the form of a connectivity description of the cells in the regular array.
4. Derive a behavioral description from the structural description through PCA.

5. Verify that the *derived* behavior has the same externally observable behavior as the *desired* behavior.

6. Obtain a transistor level implementation of the regular array corresponding to the structural description in HOP.

7. Derive the parametric boolean expressions for inputs corresponding to input constraints of the regular array.

8. Substitute these parametric boolean expressions for the inputs in the symbolic expressions of the states and outputs of the regular array obtained through PCA, to get the reference specification required for the verification of symbolic simulation results.

9. Derive the symbolic simulation vectors based on the transitions of the derived behavior of the regular array, with parameterized boolean expressions on the inputs.

10. Load the circuit description into an appropriate simulator(e.g., COSMOS), apply symbolic simulation vectors obtained in the above step, and verify that the simulation responses match with the reference specifications of the circuit.

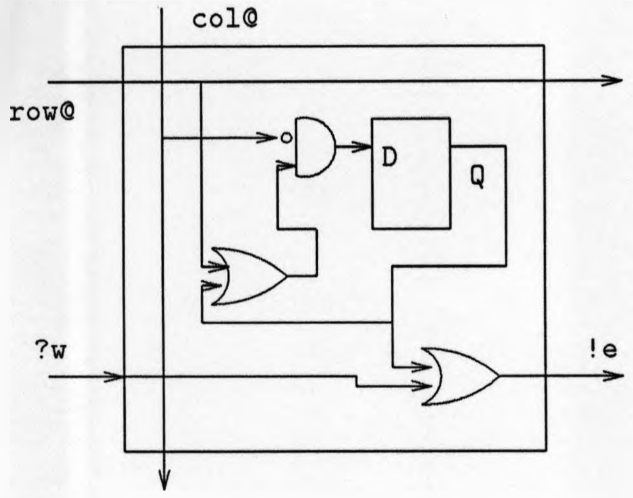
The above verification approach is illustrated on the LRU algorithm implemented as a two-dimensional regular array of LRU cells.

3 LRU Algorithm

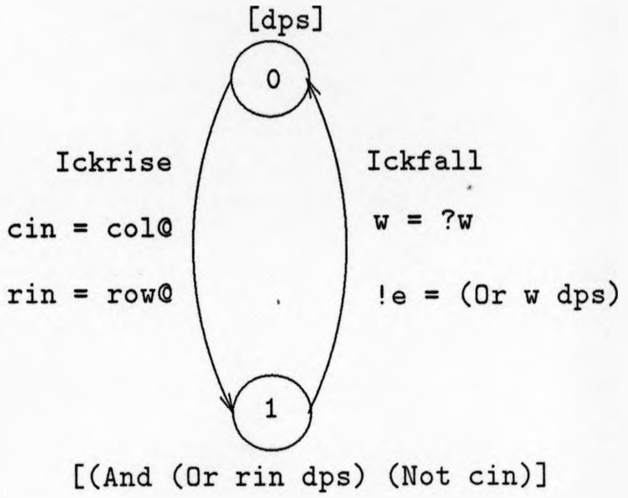
Least Recently Used(LRU) page-replacement policy is based on the common observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for a long time will probably remain unused for a long time. Thus, in LRU policy, when a page fault occurs, the page that has been unused for the longest time is thrown out.

The full implementation of LRU policy in software is expensive, as updation of the list of pages in the memory is required on every memory reference. There are several ways to implement LRU with special hardware. One hardware implementation of LRU algorithm which we consider here [18] maintains an array of $n \times n$ bits, initially all zeros, for a machine with n page frames. Whenever page k is referenced, the hardware sets all the bits of row k to 1 and sets all the bits of column k to 0. At any instant, the row with all bits set to 0 indicates the least recently used row, hence the least recently used page frame.

The LRU array is realized as a two-dimensional regular array of LRU cells. Each LRU cell of the regular array consists of a state bit which can be set to 1 by keeping the `row@` input to 1 and `col@` input to 0; the state bit can be set to 0 by keeping the `col@` input to 1. On rising edge of the clock–indicated by `Ickrise` (read:"control input `clkrise`") in the state diagram–the state bit of the LRU

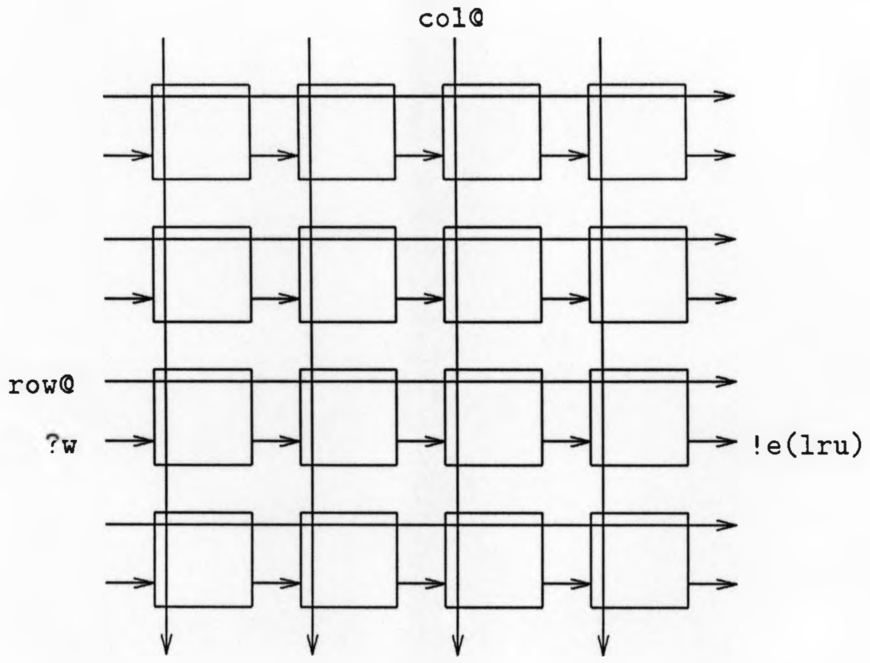


(a)



(b)

Figure 2: LRU cell and its HOP state diagram



Algorithm: Set row; reset col; find row with all zeros

Figure 3: An LRU Array

		col_0	col_1	col_2	col_3	LRU Output
		0	1	0	0	
row_0	0	r_{00}	0	r_{02}	r_{03}	$e_0 = r_{03} \vee r_{02} \vee r_{00}$
row_1	1	1	0	1	1	$e_1 = 1$
row_2	0	r_{20}	0	r_{22}	r_{23}	$e_2 = r_{23} \vee r_{22} \vee r_{20}$
row_3	0	r_{30}	0	r_{32}	r_{33}	$e_3 = r_{33} \vee r_{32} \vee r_{30}$

Figure 4: Expected values of simulation with inputs $row_1 : 1$ and $col_1 : 1$

cell is set to 0 or 1 depending upon `row@` and `col@` inputs. On falling edge of the clock–indicated by `Ickfall` in the state diagram–the output `!e` is computed as logical OR of `?w` input of the cell (which is `!e` output of the previous cell) and the state bit of the LRU cell. The output of each row is logical OR of the state bits of the LRU cells in the row. Functionality of the LRU cell is shown in Figure 2(a) and corresponding HOP state diagram is shown in Figure 2(b). A 4×4 LRU array is shown in Figure 3.

The operation of the LRU array relies on the input constraint that only the ith $(0 \leq i \leq 3)$ `row@` bit and the ith `col@` bit are 1, when page i is referenced.

4 Verification of The LRU Algorithm

Following the verification approach described in section 2, the LRU algorithm is verified at three levels. At the first level, the LRU array based algorithm is verified against an abstract specification of the LRU policy. This verification step ensures that the LRU policy realized as a two-dimensional array with operations as described in Section 3 above, implements the LRU policy correctly. For the LRU array algorithm described in the previous section, it has been shown that, at any instant, there is at least one row in the LRU array with all zeros and the rows with all zeros indicate the least recently used page frames. This verifies the LRU array algorithm at the first level, and proves that the LRU array algorithm is a correct formulation for the LRU policy. (A proof of this nature has been reported in [14].)

At the second level, the LRU regular array behavior determined by PCA is verified against the abstract specification of the LRU array algorithm. The formal verification at this level is based on the homomorphism relation between states of the inferred behavior and the states of the abstract specification. We are skipping the details of this proof in this paper.

Finally, the transistor level implementation of the LRU array corresponding to the structural description in HOP is verified against the PCA inferred behavior. However, the PCA inferred behavior cannot be directly used as the reference specification because PCA does not take input

b1	b2	row_0/col_0	row_1/col_1	row_2/col_2	row_3/col_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1
Expressions		$\neg$b1 $\wedge$ $\neg$b2	$\neg$b1 $\wedge$ b2	b1 $\wedge$ $\neg$b2	b1 $\wedge$ b2

Figure 5: Parametric boolean expressions for row and column inputs

constraints into account. Therefore we first obtain the PCA inferred behavior and then substitute into it the input and initial state values used during the transistor level symbolic simulation; *this* forms the reference specification.

The inferred behavior of the LRU array by PCA is shown in Section 2.2 through equations 4 and 5. The input constraint of the LRU array is that only bits row_i and col_i be 1, when page i $(0 \leq i \leq 3)$ is referenced. The LRU array is verified for all combinations of the input values which satisfy this input constraint. The LRU array is initialized to symbolic state values (as detailed below). Inputs satisfying the input constraint are applied, and the resulting new state and output values are compared against the expected values.

We considered two alternatives for the symbolic simulation and the corresponding verification of the 4×4 LRU array. In one alternative, we used *scalar values* satisfying the input constraint mentioned above on the row and column inputs. It required four symbolic simulation vectors to simulate and verify the LRU array. In the other alternative, we encoded the input constraint as parametric boolean expressions on the row and column inputs, with two parameter boolean variables b1 and b2, as shown in Figure 5. In both these symbolic simulation alternatives, it was possible to set the initial state of each cell of the LRU array to an independent symbolic variable.

This technique reduced the number of symbolic simulation vectors from four to one. In general, $\log_2 n$ parametric boolean variables are required to encode the input constraint of an $n \times n$ LRU array. Only one symbolic simulation vector (independent of the size n of the LRU array) is necessary.

In both the alternatives, the expected values for the state and output variables of the LRU array were obtained by substituting the input and initial state values used in the symbolic simulation into the PCA inferred behavior. For example, the expected (*i.e.* reference) symbolic expressions for the state and output of the LRU array, for input values $row_1 : 1$ and $col_1 : 1$, are shown in Figure 4. Initial symbolic state values for the LRU array are r_{ij} $(0 \leq i, j \leq 3)$.

The system times for running the symbolic simulation, and the corresponding verification, for a 4×4 LRU array, using symbolic simulator COSMOS on Sun 4, for the two alternatives are 0.2 seconds and 0.09 seconds, respectively. With the use of the encoding technique, the improvement in the simulation and verification time is expected to be significant for large LRU array sizes.

5 Conclusions and Future Work

This paper illustrated our verification approach for an important class of digital circuits–regular arrays. Not all regular arrays are alike, however. For example, some of the regular arrays employed in practice are purely combinational; some support storage within the cells without any logic blocks in the cells; some have logic and memory; some have cells that cannot affect its neighbors' state; some are fine-grained systolic arrays; etc. The reference behavioral specification that the designer would like to write for these arrays also varies according to the nature of the array. For example, for non-systolic designs, it is possible to specify simple and intuitive next-state (over one clock cycle) and output functions for the cells; in this case, the behavior inferred by PCA also specifies the behavior of the whole array over one cycle. For fine-grained systolic arrays, however, the highest level behavior that the designer would like to write may be a function (*i.e.*, mapping of states and inputs to states and outputs) that is realized in the array only over multiple clock cycles. Currently PCA can only infer what happens over one cycle; however, we are in the process of investigating extensions to PCA to infer *what happens over time.* When this work is finished, we expect our work to become applicable to systolic systems also.

We are also working on combining the verification technique for regular arrays presented in this paper, as well as the technique we presented in [9], so that large systems containing multiple regular arrays, as well as irregular structures can be verified. This technique will involve *partitioning* the system into its constituent regular arrays as well as irregular parts, and verifying these parts separately. The interface constraints of each of the partitions can be encoded using parametric boolean expressions, as described earlier.

As mentioned before, input constraints at the inputs of one module M_1 often arise because module M_2 that provides these inputs is never allowed to go into certain states (due to circuit state invariants). In such cases, while verifying M_2 separately, it would become necessary to initialize M_2 into its allowed states; our parametric encoding scheme can lend help here too.

References

1. Harry G. Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24:437–491, 1984.

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

3. Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proc. ACM/IEEE 28rd Design Automation Conference*, pages 397–402, June 1991.

4. Randall E. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-90-122, Computer Science, Carnegie Mellon University, March 1990. *Accepted for publication in the JACM.*

5. Eduard Cerny and Miguel A. Marin. A computer algorithm for the synthesis of memoryless logic circuits. *IEEE Transactions on Computers*, C-23(5):455–465, May 1974.

6. 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.

7. Lance A. Glasser and D. W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison-Wesley, Reading, MA., 1985.

8. Ganesh Gopalakrishnan and Richard Fujimoto. Design and verification of the rollback chip using hop: A case study of formal methods applied to hardware design. Technical Report UU-CS-TR-91-015, University of Utah, Department of Computer Science, 1991.

9. Ganesh Gopalakrishnan, Prabhat Jain, Venkatesh Akella, Luli Josephson, and Wen-Yan Kuo. Combining verification and simulation. In Carlo Sequin, editor, *Advanced Research in VLSI : Proceedings of the 1991 University of California Santa Cruz Conference.* The MIT Press, 1991. *ISBN 0-262-19308-6.*

10. Ganesh C. Gopalakrishnan. Specification and verification of pipelined hardware in HOP. In *Proc. Ninth International Symposium on Computer Hardware Description Languages*, pages 117–131, 1989.

11. 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.

12. Prabhakar Kudva. PCA: An algorithm for the Parallel Composition of regular Arrays, 1991. *Implementation in Standard ML, plus software documentation.*

13. B. Lattin. Vlsi design methodology: The problem of the 80's for microprocessor design. In C.L.Seitz, editor, *Proc. of the Caltech Conference on VLSI*, pages 248–252. MIT Press, 1979. *Pasadena, CA.*

14. Paliath Narendran. Verification of the lru algorithm, 1989. *Unpublished Memorandum.*

15. Carl-Johan Seger and Jeffrey Joyce. A two-level formal verification methodology using HOL and COSMOS. Technical Report 91-10, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., June 1991.

16. Thirumalai Sridhar and John P. Hayes. Design of easily testable bit-sliced systems. *IEEE Transactions on Computers*, C-30(11):842–856, November 1981.

17. Eliezer Sternheim, Rajvir Singh, and Yatin Trivedi. *Digital Design with Verilog HDL.* Automata Publishing Company, Cupertino, CA, 95014, 1990. ISBN 0-9627488-0-3.

18. Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1987. ISBN 0-13-637406-9.

19. Cheng-Wen Wu and Peter Cappello. Easily testable iterative logic arrays. *IEEE Transactions on Computers*, 39(5):640–652, May 1990.