Formal Verification of Memory Arrays

Manish Pandey

May 1997

CMU-CS-97-162

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Randal E. Bryant (Chair)
Allan L. Fisher
Rob A. Rutenbar
Richard Raimi, Motorola

This research was sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330, by the Defense Advanced Research Projects Agency (DARPA) under contract number DABT63-96-C-0071 and by a grant from Motorola.

Keywords: formal verification, symbolic simulation, symbolic trajectory evaluation, transistor-level, switch-level, memory arrays, symmetry, content addressable memory

Abstract

Verification of memory arrays is an important part of processor verification. Memory arrays include circuits such as on-chip caches, cache tags, register files, and branch prediction buffers having memory cores embedded within complex logic. Such arrays cover large areas of the chip and are critical to the functionality and performance of the system. Hence, these circuits are custom designed at the transistor level to optimize area and performance. Conventional simulation based verification approaches do not work for arrays, as it is infeasible to simulate the astronomical number of simulation patterns that are required to verify these designs. Therefore, we need to look at formal methods to ensure the correctness of these circuits.

We have adopted the formal technique of Symbolic Trajectory Evaluation (STE) to solve the array verification problem. STE uses a form of symbolic simulation to check whether a finite state system satisfies a formula expressed in a carefully restricted temporal logic. It can handle switch-level circuits and detailed system timing. However, STE does not resolve many fundamental issues important for verifying arrays. These include the state explosion problem, causing prohibitively large ordered binary decision diagrams (OBDDs) for certain classes of circuits, and the switch-level analysis bottleneck, limiting the size of switch-level circuits that can be analyzed prior to running STE.

Our thesis builds upon earlier work on STE to overcome these problems. We have developed techniques to exploit symmetry while verifying transistor-level circuits by STE. We show that exploiting symmetry allows one to verify systems several orders of magnitude larger than otherwise possible. We have verified memory arrays with multi-million transistors. The techniques we have developed also successfully overcome the switch-level analysis bottleneck. We believe that with our work, the problem of static random access memory (SRAM) verification is solved. We have developed techniques based on new Boolean encodings to efficiently verify content addressable memories (CAMs). Our encodings scale up well in terms of verification memory requirements, as compared to naive approaches. From our experimental results, and our case studies of PowerPC CAMs, we believe that we have solved the problem of verifying all the different types of CAMs that are found on a modern microprocessor. To facilitate the use of STE, we have developed an automated technique to identify the internal state nodes in transistor netlists. We have used the techniques developed in this thesis to successfully verify several memory arrays from state of the art PowerPC microprocessor designs.

Acknowledgements

I have had the good fortune to have Randy Bryant as my advisor in Carnegie Mellon. Much of what I have learned about verification, conducting research, and presenting it has been under his guidance. Many of the ideas in this thesis are a result of his suggestions. I was also fortunate to work with Richard Raimi during my internships in Motorola. His enthusiasm is contagious, and working with him helped me identify some of the core problems that I have attacked in this thesis. His suggestions and comments have made my thesis more readable. I would also like to thank Allan Fisher, and Rob Rutenbar, the other members of my committee. Their probing questions, suggestions and comments have improved this thesis in numerous ways.

A lot of what I know about VLSI design, CAD and computer architecture was learned from Akhilesh Tyagi, my advisor at the University of North Carolina, Chapel Hill. It was a privilege for me to have been his student. I have had the distinct pleasure of working with Derek Beatty in Carnegie Mellon, and during many of my summer internships in various companies. I have learned a great deal by working with him, and talking to him.

I have been fortunate to spend my last two summers as an intern in Motorola at the Somerset PowerPC design center. In these internships I have made many many friends, and I have learned a lot. These internships have helped shape my thesis in a significant way. I would like to thank Magdy Abadir for his guidance, encouragement and support for my work at Somerset. Also, I enjoyed working with Li Chung Wang and Neeta Ganguly and learned much from them. Many other friends and colleagues supported my work in Somerset, and made my stay there very pleasant. A short alphabetically arranged list includes Scott Butler, John Davis, Shantanu Ganguly, Charlie Malley, Maria Noack, Doug Parse, Raj Raina, Paul Reed, and Hemendra Talesra. My apologies for keeping the list short, for it would have included much of the Methodology and Tools group, and many from the processor groups. I am grateful to Motorola for allowing us to publish the verification results obtained during these internships. I am also grateful to Doug Parse for making available a grant from Motorola to support our research in Carnegie Mellon.

Many other friends and colleagues have contributed directly and indirectly to this work in numerous ways. A short, and incomplete list includes Yirng-An Chen, Ed Clarke, Kye Hedlund, Mark Horowitz, Alok Jain, Somesh Jha, Timothy Kam, Manpreet Khaira, Andreas Kuehlman, Carl Pixley, Suresh Rajgopal, Carl Seger and Gary York.

I would like to thank my parents for supporting me and encouraging me to follow whatever I have wanted. Finally, the biggest chunk of thanks surely go to my wife Madhulima for her constant love and support, for her unwavering faith in me that I could finish this. Madhu, I could never have done it without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Three fundamental trends mark the semiconductor industry today.

- Rapid advances in Very Large Scale Integrated circuit (VLSI) fabrication technology.
- Shortening design-to-market cycle times.
- Increasing reliance of society on digital systems.

These trends continue unabated. According to the Semiconductor Industry Association's National Technology Roadmap for Semiconductors [3], in AD 2010 semiconductor devices will have a minimum feature size of 0.07μm, microprocessors will contain 90 million logic transistors, several hundred million cache transistors, and they will run at frequencies over 1000 MHz. Thus, advances in fabrication technology are continuously fuelling the design of hardware systems which are larger and more complex than their predecessors.

The intense competition in the field is resulting in ever shorter design cycles, and delays can be very expensive. To amortize the mounting costs of development, and manufacturing, the market is increasingly mass production oriented. All of these issues point to the fact that errors in the design of digital systems can have severe financial implications. This is best illustrated by the $475 million cost of the Pentium floating point bug [118]. Furthermore, with increasingly widespread use of digital systems in areas such as aircraft fly-by-wire systems, and medical applications, design errors can lead to serious injuries, or even loss of life.

Therefore ensuring correctness of designs is an important task in the design of digital systems. Today, the most popular verification technique is the simulation of selected test cases. However, with increase in design complexity, simulation is proving to be increasingly inadequate. For a large complex system it is impossible to simulate all possible combinations of inputs, or sequences of inputs. A growing trend in the industry has been the shift away from manual simulation test pattern generation to random simulation test pattern generation. This partly avoids the problem of generating test cases which may be inadvertently biased. However, it does not solve the problem of uncovering an error which can occur only under remote circumstances, when a set of unlikely factors conspire together.

For these reasons, there has been an increasing interest in *formal verification* techniques for hardware designs. In formal verification, a mathematical model of the design is compared with a formal specification which describes the expected behavior of the design. The verification then uses rigorous, formalized reasoning to determine whether for all possible inputs, the behavior of the design is consistent with the specification.

Microprocessors are among the most complex digital systems being built today. Memory arrays such as caches, cache tags and translation lookaside buffers (TLBs) are important components of these systems. Verification of these arrays is an important part of verifying a microprocessor. The goal of our work described here is to develop techniques which substantially improve upon existing array verification practice.

Ahead, section 1.1 discusses memory arrays and their characteristics. Section 1.2 briefly surveys work in the area of formal verification, and it discusses the difficulties that most formal techniques have in handling arrays. It also discusses issues in using symbolic trajectory evaluation (STE) is to handle arrays. This is followed by section 1.3 which summarizes the goals of the thesis. Finally, section 1.4 discusses the thesis organization, and a summary of each of the major chapters.

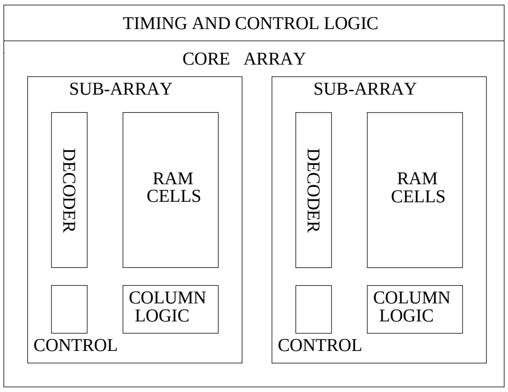

Figure 1.1: An array.

1.1 Verification of memory arrays

Verification of memory arrays is an important part of processor verification. Arrays are functional units in a processor with read and write memory structures containing multiple data locations [89]. Circuits classified as arrays include register files, caches, block address translators, tags, and branch target prediction buffers. These circuits typically consist of random access memory cores embedded in complex timing and control logic (Figure 1.1).

Arrays form an important part of microprocessors. On the PowerPC 604 chip [113], arrays include the following (marked with numbers on the chip layout in Figure 1.2):

1. Instruction Cache.

2. Instruction Cache Tags.

3. Instruction memory management unit (MMU) including the Instruction side Translation Lookaside Buffer (ITLB), and the Instruction Block Address Translator (IBAT).

4. Data MMU including the Data side Translation Lookaside Buffer (DTLB), and the Data side Block Address Translator (DBAT) arrays.

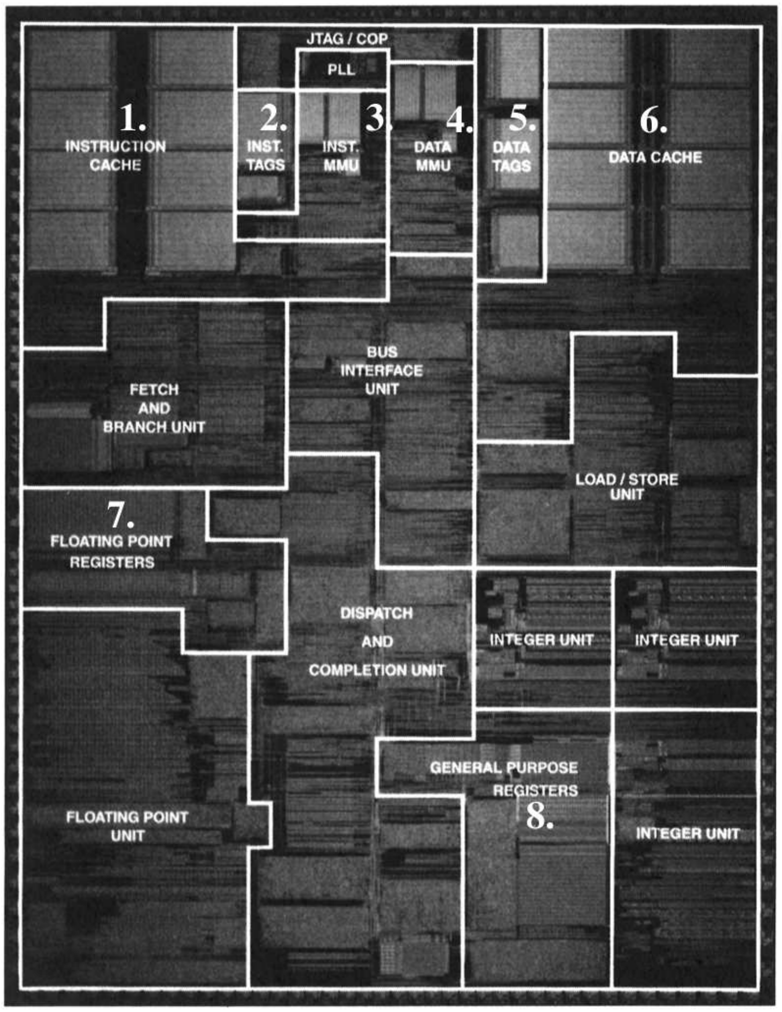

Figure 1.2: The PowerPC 604 Microprocessor (Copyright ©1994 MicroDesign Resources)

5. Data Cache Tags.

6. Data Cache.

7. Floating point register file.

8. General purpose register file.

These arrays are custom designed at the transistor level for optimizing *area* and *performance*. Arrays can occupy as much as 40-50% of the processor chip area. In the recently announced PA-7300LC processor [61] from Hewlett Packard, caches consume 49% and the TLB takes over 5% of the chip area. In the MediaProcessor chip announced by MicroUnity [119], tags and TLBs take over 5% of the chip area. The register file and the caches of this chip take 10% and 25% of the chip area respectively.

Arrays often come in the critical timing paths in a chip. For example, every instruction fetch requires that the instruction address be first translated by the TLB, and then whether or not the word is present in the instruction cache must be checked in the cache tags, all in one clock cycle. With the clock frequencies today, and in the foreseeable future, this is an important critical timing path in a chip.

Because of these area and performance constraints, arrays are not synthesized by automatic synthesis tools, rather they are custom designed. The designs include many features to maximize performance, including the use of self-timed components. For example, the MIPS-X register file generates an internal self-timed signal to precharge the bit lines before a read [36]. The register file uses two phase clocking. During the first phase, data is written into the register file, and during the second phase data is read from the register file. Prior to the read, the word lines of the memory cells must be precharged to high. Having a separate clock phase for the precharge makes a three phase clock necessary. The register-file design works around this by detecting the end of writes by means of a dummy column, and it then generates a precharge signal which is killed when the second phase begins.

These characteristics of arrays, including size, their design at the transistor level with sophisticated implementation techniques, and complex internal timing make array verification a difficult task. Simulation based techniques suffer from their inability to exhaustively verify these circuits. Often, for speeding up simulation, functional models are substituted for the memory core in an array. Such an approach is also not satisfactory, as it can fail to capture the detailed internal timing in the system, and thus mask design errors. Thus, to successfully verify memory arrays, we need to employ a formal technique which can handle designs at the transistor level. The formal technique should be automated, and it should be able to handle the large size of arrays. The technique we choose should facilitate the specification of the behavior of arrays. Of all the formal verification techniques, Symbolic Trajectory Evaluation (STE) comes closest to meeting the requirements stated above. However, it does not resolve many fundamental and pragmatic issues. These issues include the issue of array size, specification and efficient verification of array properties, and the identification of data storage nodes in arrays. We have addressed these specific issues in this thesis.

1.2 Related work

In this section we first discuss different formal verification approaches to place STE and our work in context. This discussion is quite broad and general. Specific pieces of work which are more closely related to our research have been referenced at the end of each chapter. In section 1.2.2 that follows, we discuss the problems in formally verifying memory arrays. Section 1.3 elaborates on the main contributions of this thesis.

1.2.1 Formal Verification

State Machine Analysis

Most automated approaches to formal verification are based either on state machine analysis or symbolic simulation. In state machine analysis, the verifier creates a finite state machine representation of the system and characterizes the system behavior for a number of transitions. Traditionally, many of these approaches used an explicit representation of the state transition graph, which made their use impractical for all but the smallest of finite state systems. The advent of reduced Ordered Binary Decision Diagrams (OBDDs) [15], and the use of OBDD based symbolic techniques to represent the state transition graph [30, 43] made it possible to verify systems with as many as 10^{20} to 10^{100} states, much larger than what could be verified using explicit state representation techniques. With the notable exceptions of the Murphi[45] and the SPIN [66] verification systems, symbolic state transition representations are used in most automated state machine analysis techniques today.

It should be noted that the number of states rises rapidly with the number of state storage elements in a design — a small register file contains 32 32-bit registers has a state transition graph over 10^{300} states. So even a technique that can handle 10^{100} states falls hundreds of orders of magnitude short in its capacity to handle the register file

Symbolic state machine analysis techniques necessitate the creation of an explicit representation of the system in terms of state variables and next state functions which update them. The program can then analyze properties of the machine, such as deciding the truth of a temporal logic formula, performing reachability analysis, and state machine comparison. In the symbolic model checking algorithm described by Burch et.al. [29], the underlying model can be non-deterministic and the user can even specify fairness constraints with temporal logic formulas. However, this great expressive power comes at a price. For each system state variable one needs to have two Boolean variables, one for the "old" value, and one for the "new" value. The next state relation is represented as a Boolean function of all these variables, where the function yields 1 when the old and the new state are related, and 0 otherwise. For large systems there are two fundamental problems that even symbolic techniques cannot resolve. The first is that the system transition relation representation can be prohibitively large. The second, which is a far more serious problem, is the prohibitively expensive representation of sets of states by their characteristic function during the state graph exploration. A number of approaches have focused on the use of partitioned transition relations to reducing the OBDD size during state machine exploration [55, 31]. However, while these techniques have succeeded in reducing the size of the transition relation, the problem of computing relational products, and representing characteristic function of sets of states still remains a significant headache.

For most hardware verification practitioners in the industry, model checking is synonymous with the SMV model checker [92], or its variants, which are based on symbolic state machine analysis. Clarke et al. first described a model checking algorithm for a powerful branching time temporal logic called computation tree logic or CTL in [37]. Formulas in this logic describe properties of computation paths, which are infinite sequences of states systems go through during their execution. In [32] Clarke et al. describe a symbolic version of the model checking algorithm by encoding the transition relation using OBDDs. McMillan discusses the details of symbolic model checking, and the SMV system in [92]. Coudert et al. independently developed symbolic CTL model checking and their work is described in [43]. In this work they describe the verification of CTL properties for deterministic finite state machines derived from the system's behavioral description in the LDS language [9]. Bose and Fisher [11] describe a model checking algorithm for a variant of CTL where symbolic simulation is used to characterize the next state function of the system. Their technique has the capability to work with detailed circuit representations, such as the switch-level model. By choosing to ignore the verification capabilities gained from ternary simulation [16], and by working only in the binary domain, this technique can be viewed as trading off the size of the circuits that can be verified, to allow the verification of system properties specified in a richer temporal logic, as compared to a approach like STE [27]. By representing the system behavior with next state functions, the work limits itself to the verification of deterministic finite state systems. Of course, the use of temporal logic for the specification and verification of hardware systems has long been an active research area. The subsection below on temporal logic discusses some of the related work in this area.

State machine analysis also includes techniques for determining whether two finite state machines are equivalent. Typically, in the hardware verification domain, the machine derived from a high-level system description like RTL or behavioral spec. is termed the specification machine, and the machine corresponding to the low-level system description, perhaps at the gate or the switch-level, is termed the implementation machine. Of course, there is no reason why the implementation and the specification machines may not be at the same abstraction levels. Some equivalence checking techniques have been described in [82, 40, 34]. Verity [82] has been designed to verify the equivalence of RTL design descriptions to switch-level implementations. It has been targeted especially at custom CMOS designs. Two key ideas simplify the complexity of equivalence checking for large designs, and they give Verity the capability of performing full-chip verification – identical state encodings, and identical hierarchical structure for both the specification and the implementation state machines. Coudert et al. describe the PRIAM system in [9, 88], where the specification and the implementation machines are extracted from the LDS HDL. Since the next state and the output functions are represented by OBDDs, and the state encodings are identical, determining equivalence is the trivial comparison of the corresponding next state and the output OBDDs. Coudert and colleagues extended the work to do equivalence checking for state machines without identical state encodings with a symbolic breadth first exploration of the product machine state graph [42, 41]. These ideas have also been explored by Camurati and colleagues in [34].

All of the above techniques require partitioning the circuit into combinational logic and latches. This is not always easy for transistor-level circuits that are based on memory structures because often parts of the state update and output logic are merged with the storage circuitry. Furthermore, most FSM equivalence approaches assume that the implementation and the specification machines start from the same state. Most memory arrays have only a limited reset capability, and therefore the initial state of the actual circuit usually cannot be predicted. A notable exception to this is the work by Pixley, [109], which decides the equivalence of two machines without a knowledge of their initial states.

As a part of state machine analysis, abstraction based techniques have been used to attack the state explosion problem. Techniques in this class attempt to verify that if a temporal logic formula holds in the abstracted system, then it also holds in the original transition system [86]. In this approach, the original transition system is never constructed and the abstractions work directly on an HDL-like language describing the system to produce the abstract transition system. Such an approach is not applicable here, for our starting point is a flattened transistor-level netlist, and these techniques do not work at this level of abstraction. Symmetry based approaches to attack the state explosion problem are more recent [50, 38, 69, 100]. These have been described in chapter 3.

Temporal Logic

While the section above mentions some of the more successful and automated applications of temporal logic to hardware verification, the idea of using temporal logic to study properties of concurrent, reactive systems is not new [90, 110, 111]. In a temporal logic, *temporal operators* are added to the usual propositional operators to allow reasoning about dynamically changing situations. There exist many flavors of temporal logic including linear time temporal logic (LTL) and branching time temporal logic (BTL). LTL versions of temporal logic [110] characterize time as a linear sequence of events and proof systems for LTL have been studied in [91, 84]. BTL versions of temporal logic characterize the behavior of systems over time as a set of branching possibilities such that at any given point of time there are multiple branching sequence of events (paths) each expressing a possible sequence the system may take in the future. Various flavors of BTL exist [48], including the well known CTL [37]. As stated above, symbolic model checking techniques can efficiently check the truth of a CTL formula for small systems. Temporal logics are suitable for expressing qualitative properties of systems like safety and liveness properties. With the exception of CTL model checking work by Clarke and colleagues, there is little use of temporal logic formalisms for automated verification of large finite state systems.

Symbolic Simulation

While symbolic simulation is best known today for simulation and verification of digital circuits [21, 117], its use for hardware verification can be traced to Darringer in [44]. The essence of his approach is to establish a simulation relation between the specification and the implementation programs, and show that the symbolic execution trees for the programs are the same. Darringer applies this technique for microcode verification.

Our version of symbolic simulation is an extension of conventional digital simulation, where a simulator evaluates circuit behavior using symbolic Boolean variables to encode a range of circuit operating conditions. Circuits nodes can take on symbolic values in addition to the constants 0, 1 and X. The use of symbolic Boolean values makes each run of a symbolic simulator equivalent to multiple runs of a conventional simulator, one for each assignment of 0/1 values to the symbolic Boolean variables introduced during the simulation.

Symbolic simulation became more practical after the introduction of OBDDs by Bryant [15]. Straightforward symbolic simulation is adequate for verifying combinational circuits or combinational portions of sequential circuits [88]. The efficient extraction of the logic functionality of a digital MOS circuit in terms of Boolean operations [19, 18], and the availability of OBDDs, made symbolic simulation of switch-level circuits feasible. The COSMOS simulation system developed at Carnegie Mellon

University [25] was the first to have the capability to do symbolic ternary switch-level simulation. In [23], Bryant describes the verification of of memory circuits using ternary switch-level simulation. The desire to leverage the power of ternary switch-level simulation, and the ability to do hardware verification by logic simulation [16] gave rise to the technique of Symbolic Trajectory Evaluation (STE). In STE, circuit behavior is expressed as a set of assertions in a carefully restricted specification language. The circuit is simulated with symbolic ternary simulation patterns derived from the assertions, and the response of the circuit is compared against that expected from the assertion (details in chapter 2). Early approaches to formalizing the concepts of STE can be traced in the following succession of papers from Bryant and colleagues — [5], [26], and [27]. Many utilities, and the core in the Voss verification system [116] come directly from the symbolic simulation based formal verification efforts at CMU. In [117], Seger and Bryant extend STE to work with a richer value domain, and a more generalized assertion syntax. Recent STE extensions by Jain and Bryant [72] allow the specification of system behavior while operating in non-deterministic environments.

Bose and Fisher [12] have used symbolic simulation, and the idea of representation functions [65] to verify pipelined systems. The representation function maps the state of a pipelined design into a that for a unpipelined design. The system behavior is specified as a set of Hoare like pre and postconditions.

Symbolic simulation based techniques have the capability of working with systems much larger than what be handled by traditional state machine analysis approaches like CTL model checking. Another advantage is their ability to work with detailed circuit models. However, successful application of symbolic simulation based approaches for verifying memory arrays requires overcoming significant fundamental and pragmatic issues, which is the subject of this thesis. We discuss this in more detail in section 1.3.

Theorem Proving

In theorem proving based approaches, the circuit is described as a hierarchy of components, and there is a behavioral description of each component in the hierarchy. The proof of a design's correctness is based on proofs of its component's correctness, which is obtained by composing the proofs of the sub-components at the lower level. The

HOL [57], Boyer-Moore [13] and the PVS [98] theorem provers have been successfully used to verify several hardware systems.

HOL [57, 56] is among the best known theorem provers which has been applied to hardware verification. HOL mechanizes proofs of theorems in higher order logic. The user interacts with HOL through ML, a strongly typed functional language [104], and the type system of the implementation guarantees that it has a sound proof system. Since HOL has its origins in Cambridge University, it is not surprising that a lot of work on formal verification from this university has centered around HOL. Among others, these include work by Camilleri [35], Joyce [76], and Melham [93].

The Boyer-Moore theorem prover [13], which is based on quantifier free first order logic, is more automated than the HOL theorem prover. A significant application of the Boyer-Moore theorem prover is the verification of the FM8501 microprocessor by Hunt [68]. The PVS (Prototype Verification System) theorem prover [98] is based on type higher order logic. PVS provides a specification language integrated with a theorem prover, and support procedures to ease the burden of developing proofs. Srivas and colleagues have used PVS to verify several hardware designs including a pipelined processor [120, 121], and a SRT division module [114].

However, the basic weakness of theorem proving approaches is that they require a large amount of user intervention to create specifications and perform proofs, which makes them unsuitable for automation. Attempts at automation of proofs have not been particularly successful, and proofs still require substantial interaction, and skilled guidance from the user. In the context of arrays, it worth noting that most theorem proving approaches, with the exception of Weise [124], use rather weak circuit models. This is a serious limitation, since arrays are designed at the transistor-level, and, therefore they should be verified at the switch-level.

Other Approaches

In a technique developed by Burch and Dill [33], the user provides behavioral descriptions of an implementation and specification. These descriptions are compiled into transition functions. The verifier attempts to proves that if the implementation and the specification start in any matching pair of states, then the result of executing any instruction will lead to a matching pair of states. The mapping between implementation and specification is defined by using a number of NOPs to flush out internal pipeline registers. This approach is geared towards verifying the correctness of high-level system descriptions. In contrast, we need a technique to verify the transistor-level implementation of a design. It assumes the correctness of data-path, and concentrates on proving the correctness of the control. This makes it unsuitable for our problem domain, where we want to verify transistor-level implementations of various hardware units like register-files, TLB's and cache tags.

1.2.2 Why are arrays difficult to verify?

Switch-level simulation has long been the predominant array verification technique in industry. However, the size and complexity of arrays precludes the use of a exhaustive set of simulation vectors. The typical verification run simulates only a small fraction of the system behavior space. Memory arrays include many complex features like self-timed components and multiple internal clock phases. This complexity, and their ever increasing size exacerbates the verification problem. Therefore, it is not surprising that often design errors slip by the verification process, only to reveal themselves in Silicon.

A multitude of reasons make arrays a difficult class of circuits to verify using most existing formal verification techniques. We have enumerated these below.

1. State explosion problem.

   The large number of state holding elements can result in the state explosion problem. The Boolean functions to represent the next state and the output functions for arrays can be very large, especially since arrays can contain over 10^4 state holding elements. For designs with non-trivial behavior, e.g., the cache tags unit, attempting to build the output function which depends on all the internal state values will most likely lead to unacceptably large BDDs. For the same reason, the construction of the transition relation and keeping track of the characteristic function will also be prohibitively expensive.

2. Arrays do not follow the classical FSM structure of combinational logic and feedback latches.

   Arrays are based on memory-like structures. Often they have state update and output logic merged with state storage which makes it difficult to separate the design into combinational logic and latches. Even if it were possible to separate an array into combinational logic and latches, such a model will likely fail to capture the complex internal timing inherent in most arrays.

Many verification techniques (e.g. [109]) do not allow combinational loops in circuits. Therefore, such techniques will not work in the case of cross coupled inverters, which is a commonly used circuit structure for memory circuits.

3. The switch-level analysis bottleneck

   Prior to running STE on a switch-level circuit, it is necessary to derive its excitation function by switch-level analysis of the circuit (Section 2.2). For large memory array circuits which have large channel connected components, this step becomes prohibitively expensive (Table 3.1). This analysis stage imposes a limit of a few hundred thousand transistors on the maximum circuit size that can be analyzed within practical space and time bounds. Clearly, this is not adequate, as microprocessors today have on-chip arrays like level 1 and level 2 caches containing well in excess of a million transistors.

4. Lack of information on state points in the circuit

   The absence of such information in the traditional design and verification methodology makes it problematic for both property verification tools, as well as machine equivalence checkers. To circumvent this problem, verification tools like Verity [82] require that the design has identical hierarchical partitioning in both the RTL, and the transistor-level views. Such a strict requirement affects the overall design partitioning in an adverse manner. Often, hierarchical partitioning at the RTL level is chosen to represent the high-level system functionality, and at the transistor-level, a hierarchical partitioning that is suitable for layout is desired.

1.3 Scope of the thesis

We start with the formal verification technique of STE to solve the problem of verifying memory arrays. However, direct application of STE in this domain is not without challenges. We classify the challenges into two categories — *fundamental*, and *methodology related*. The fundamental challenges include the state explosion problem which results in unacceptably large OBDD growth for certain classes of circuits, and the switch-level analysis bottleneck. The methodology related challenges include problems like having a general framework to specify and verify array properties, and relating low-level circuit implementation details to an abstract view of the circuit. We have built upon earlier work on STE to overcome these challenges. The principal contributions of this thesis are the following:

- Exploiting symmetry with STE

  We have developed techniques to exploit symmetry while verifying transistor-level circuits by STE. We show that exploiting symmetry can allow one to verify systems several orders of magnitude larger than otherwise possible. We have verified memory arrays with over a million transistors. The technique we have developed also successfully overcomes the switch-level analysis bottleneck. We believe that with our work, the problem of SRAM verification is solved.

- Verification of content addressable memories

  We have studied the structure of content addressable memories (CAMs). Using new Boolean encoding techniques we have developed techniques to efficiently verify this class of circuits. Our encoding techniques scale up well in terms of space requirements, as compared to naive encodings. From our experimental results, we believe that we have solved the problem of verifying all the different types of CAMs that are found on a modern microprocessor.

- Automated state node identification

  To facilitate the use of STE, we have developed an automated state node identification technique, and we have used this technique to successfully verify several industrial arrays.

- Application of the techniques developed in the thesis to several memory array designs from state of the art microprocessors.

  An integral part of our thesis has been the application of the techniques we have developed to real industrial designs. We have used our techniques to verify the following memory arrays from recent PowerPC processors — multi-ported register file, data cache tags unit, branch target address cache, and a block address translator array. The last two arrays are complex CAMs.

1.4 Thesis Overview

Chapter 2 discusses symbolic trajectory evaluation of switch-level circuits. It starts with our new *atom* formulation of STE which facilitates discussion of symmetry in circuits. The chapter discusses the methodology we have adopted for verification of memory arrays. In addition, the chapter also discusses the switch-level model, which is a prerequisite for understanding the circuit transformation techniques we use to create conservative approximations.

Chapter 3 first defines what a symmetry property means, and it then classifies symmetries in systems as *structural*, *data*, and *mixed* structural data symmetries. It discusses how these symmetries can be verified in a system. It discusses the use of conservative approximation to create reduced circuit models, and to partition circuits to expose highly symmetric regions of designs. The chapter then illustrates the application of the techniques developed to verify SRAM designs.

Chapter 4 is on the verification of content addressable memories (CAMs). This chapter starts with a discussion on the structure and properties of CAMs. It then develops techniques for generation of ternary vectors essential for verifying CAMs efficiently. It presents results which demonstrate the efficiency of our techniques.

Chapter 5 discusses the state node identification problem. It discusses the basic theory underlying the identification problem, and it then discusses the approaches we have used successfully for identifying state nodes in large memory arrays. We then present a general algorithm for state node identification and some results we have obtained on simple circuits.

Chapter 6 is on case studies on verification of several memory arrays from recent PowerPC microprocessors. These include a multi-ported register file, a data cache tags unit, a branch target address cache circuit, and a block address translator circuit. We discuss the high-level design of each of these circuits, some of the interesting properties we verified, and the bugs we discovered during the verification process.

Chapter 7 rounds off the thesis with an evaluation of the work and possible future research directions.

Note: Figure 1.2 has been reproduced from [60, page 9] with permission from MicroDesign Resources.

Chapter 2

Symbolic trajectory evaluation of switch-level circuits

We use symbolic trajectory evaluation (STE) to verify memory arrays. The level of abstraction most appropriate for modeling this class of circuits is the switch-level. To specify and verify the behavior of memory arrays, we have adopted a verification methodology which partitions the specification into two components — a set of assertions describing abstract system behavior, and an implementation mapping specifying low-level implementation details. This chapter gives a brief background on these basic aspects of our verification methodology.

2.1 Symbolic Trajectory Evaluation

2.1.1 Mathematical Notation and Background

Before we proceed with the main presentation, we briefly discuss some of the notation and the basic mathematical concepts used in this chapter.

Our notation generally uses upper case script letters like $\mathcal{P}, \mathcal{Q}, \mathcal{R} \ldots$ to denote sets. For example, $\mathcal{B}$ denotes the set of binary values $\{0, 1\}$, and $\mathcal{T}$ denotes the set of ternary values $\{0, 1, X\}$. Generally, lower case letters, $p, q, r, \ldots$, denote individual set elements.

A binary relation $\mathcal{R}$ on set $\mathcal{Q}$ is a subset of $\mathcal{Q} \times \mathcal{Q}$. Often, we use the infix notation $p\mathcal{R}q$ for $(p, q) \in \mathcal{R}$. Relation $\mathcal{R}$ on a set $\mathcal{Q}$ is a *partial order* if it is reflexive, symmetric, and transitive, i.e.,

$$\forall q \in \mathcal{Q}.\ q\mathcal{R}q$$
$$q_1\mathcal{R}q_2 \ \wedge\ q_2\mathcal{R}q_1 \Rightarrow q_1 = q_2$$
$$q_1\mathcal{R}q_2 \ \wedge\ q_2\mathcal{R}q_3 \Rightarrow q_1\mathcal{R}q_3$$

A *poset* is an ordered pair $\langle\mathcal{S}, \sqsubseteq\rangle$, where $\mathcal{S}$ is a set, and $\sqsubseteq$ is a partial order of $\mathcal{S}$. A commonly used poset in this thesis is the ternary value poset, $\langle\mathcal{T}, \sqsubseteq_{\mathcal{T}}\rangle$. In this poset, $\mathcal{T}$ equals $\{0, 1, X\}$, the discrete states of the switch-level mode, and the partial order $\sqsubseteq_{\mathcal{T}}$ is such that, $\forall a \in \mathcal{T}.\ a\sqsubseteq_{\mathcal{T}}a$, $X\sqsubseteq_{\mathcal{T}}0$ and $X\sqsubseteq_{\mathcal{T}}1$. The partial order $\sqsubseteq_{\mathcal{T}}$ is consistent with the information conveyed by the values in $\mathcal{T}$ since a 0 or a 1 conveys more information than an X in a circuit.

If $\langle\mathcal{S}, \sqsubseteq\rangle$ is a poset, $\mathcal{P} \subseteq \mathcal{S}$, then $q \in \mathcal{S}$ is a *lower bound* of $\mathcal{P}$ iff $q \sqsubseteq p$ for all $p \in \mathcal{P}$. A lower bound of $\mathcal{P}$ is called the *greatest lower bound* of $\mathcal{P}$, written $glb(\mathcal{P})$, if and only if $q \sqsubseteq p$ for every lower bound q of $\mathcal{P}$. Similarly, the *upper bound*, and the *least upper bound* written as $lub(\mathcal{P})$ are defined dually. For notational convenience, when we enumerate the elements of a set, e.g., $\mathcal{P} = \{p, q\}$, then we write $glb(\mathcal{P})$ as $glb(p, q)$, rather than $glb(\{p, q\})$. Similarly, $lub(\mathcal{P})$ is expressed as as $lub(p, q)$. A poset $\langle\mathcal{S}, \sqsubseteq\rangle$ has a universal lower bound $\perp \in \mathcal{S}$ iff $\perp \sqsubseteq p$ for all $p \in \mathcal{S}$. The *universal upper bound*, $\top$, is defined dually.

A poset $\langle\mathcal{S}, \sqsubseteq\rangle$ is a *complete lattice* if $lub(\mathcal{P})$ and $glb(\mathcal{P})$ exist for every subset $\mathcal{P} \subseteq \mathcal{S}$. A complete lattice has a universal upper bound $\top \in \mathcal{S}$, and a universal lower bound $\perp \in \mathcal{S}$. For example, the powerset $2^{\mathcal{P}}$ of set $\mathcal{P}$, with the subset relation $\subseteq$ is a lattice. The set union operation, $\cup$, and the set intersection operation, $\cap$, are the *lub*, and the *glb* operations respectively. Set $\mathcal{P}$, and the empty set $\emptyset$, are the universal upper bound, and the universal lower bound, respectively of this lattice.

If $\langle\mathcal{S}_1, \sqsubseteq_1\rangle, \langle\mathcal{S}_2, \sqsubseteq_2\rangle, \ldots \langle\mathcal{S}_n, \sqsubseteq_n\rangle$ are n complete lattices, then $\langle\mathcal{S}, \sqsubseteq\rangle$ is a complete lattice, where $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n$, and for any $p = (p_1, p_2, \ldots, p_n) \in \mathcal{S}$, and $q = (q_1, q_2, \ldots, q_n) \in \mathcal{S}$ $p \sqsubseteq q$, iff $p_i \sqsubseteq_i q_i$ for $1 \leq i \leq n$. $\langle\mathcal{S}, \sqsubseteq\rangle$ is a complete lattice.

Given a poset $\langle\mathcal{S}, \sqsubseteq\rangle$, and a mapping $f : \mathcal{S} \to \mathcal{S}$, we say that f is *monotone* if and only if

$$p \sqsubseteq q \Rightarrow f(p) \sqsubseteq f(q)$$

System behavior is expressed as a sequence of elements from some set $\mathcal{S}$. The set of all infinite sequences of elements of $\mathcal{S}$ is denoted by $\mathcal{S}^\omega$. While these sequences are infinite conceptually, most properties we are interested in can be determined from finite length prefixes of these sequences. Given a poset $\langle \mathcal{S}, \sqsubseteq \rangle$, we can extend the relation $\sqsubseteq$ to elements of $\mathcal{S}^\omega$ by a point-wise extension of $\sqsubseteq$. If $s_1 = {s_1}^0 {s_1}^1 \ldots$, and $s_2 = {s_2}^0 {s_2}^1 \ldots$ are elements of $\mathcal{S}^\omega$, then $s_1 \sqsubseteq s_2$ iff $s_1^i \sqsubseteq s_2^i$ for $i \geq 0$. Similarly, *lub* and *glb* can be extended point-wise.

2.1.2 Model structure and State Domain

Our view of circuits is quite abstract. We view circuits as consisting of a set of nodes and an excitation function which determines how the circuit nodes get updated at every time step. This abstract viewpoint can capture behavior at various levels of abstraction ranging from detailed switch-level models to register transfer level and high-level behavioral models. The important questions that arise are the following:

- What is the domain of values circuits operate over?
- Does this domain have some structure?
- How are the excitation functions generated and represented?

This section answers the first two questions. Section 2.2 answers the last question.

In this thesis, we predominantly deal with switch-level models of circuits. In this model node voltages are expressed by the 0,1,X discrete values of the ternary set $\mathcal{T}$. An early formulation of STE [27] worked directly on this ternary domain., with the natural information ordering of the ternary values as specified by the poset $\langle \mathcal{T}, \sqsubseteq_\mathcal{T} \rangle$. A later formulation of STE was generalized to handle lattice-structured value domains [117], and more complex forms of assertions. We can extend $\langle \mathcal{T}, \sqsubseteq_\mathcal{T} \rangle$ $\mathcal{T}$ by adding a top element, $\top$, to $\mathcal{T}$ such that for all elements a in $\mathcal{T} \cup \{\top\}$, $a \sqsubseteq_\mathcal{T} \top$. This approach, however, does not provide a suitable framework to allow the clear expression of symmetry properties of a circuit. This has motivated us to describe the *atom formulation* of STE, where the value domain of circuits is a set of *atoms*.

Intuitively, knowledge about about the state of a circuit is built up of *information atoms*, similar in the spirit of how the physical world we live in is composed of atoms[1].

[1]At least from the point of view of a chemist.

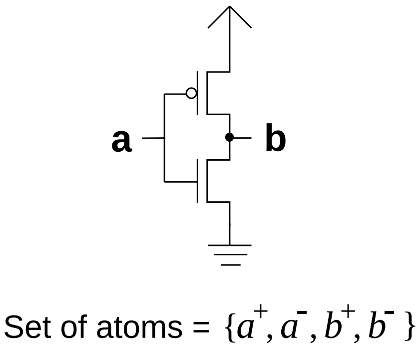

Figure 2.1: Set of atoms for an inverter circuit.

The state of a circuit consists of values at the circuit nodes. Thus, we need to define atoms associated with every circuit node, as described below.

Definition 1 *Let N denote the set of nodes of a circuit. For every node* n *of a circuit, we define two atoms, n^+ and n^-, indicating that node* n *has value 1 or 0 respectively. Let $\mathcal{A}$ denote the set of all the atoms of a circuit.*

Figure 2.1 shows the set of atoms for a two node circuit. An atom for a node restricts the value of the node. A set of atoms of a circuit restricts the values on the circuit nodes. For example, the atom set $\{a^+, b^-\}$ indicates a is 1, and b is 0. This motivates the following definition of a circuit state.

Definition 2 *Given the set of all the atoms of a circuit, $\mathcal{A}$, we define a circuit state S to be any subset of $\mathcal{A}$, and $\mathcal{S}$ to be the set of all possible states, i.e., $\mathcal{S} = 2^{\mathcal{A}}$.*

State set $\mathcal{S}$, together with the subset ordering $\subseteq$ forms a complete lattice, where states are ordered according to their "information content," i.e., how much they restrict the values of the circuit nodes. For example, the structure of the state domain for the circuit in Figure 2.1 is illustrated in Figure 2.2. In this diagram we indicate the set of atoms in each state. As the shaded regions indicate, states can be classified as being "partial", "complete", or "conflicting". In a partial state, some nodes have no corresponding atoms while others have at most one. In a complete state, there is exactly one atom for each node. In a conflicting state, there is some node n for which

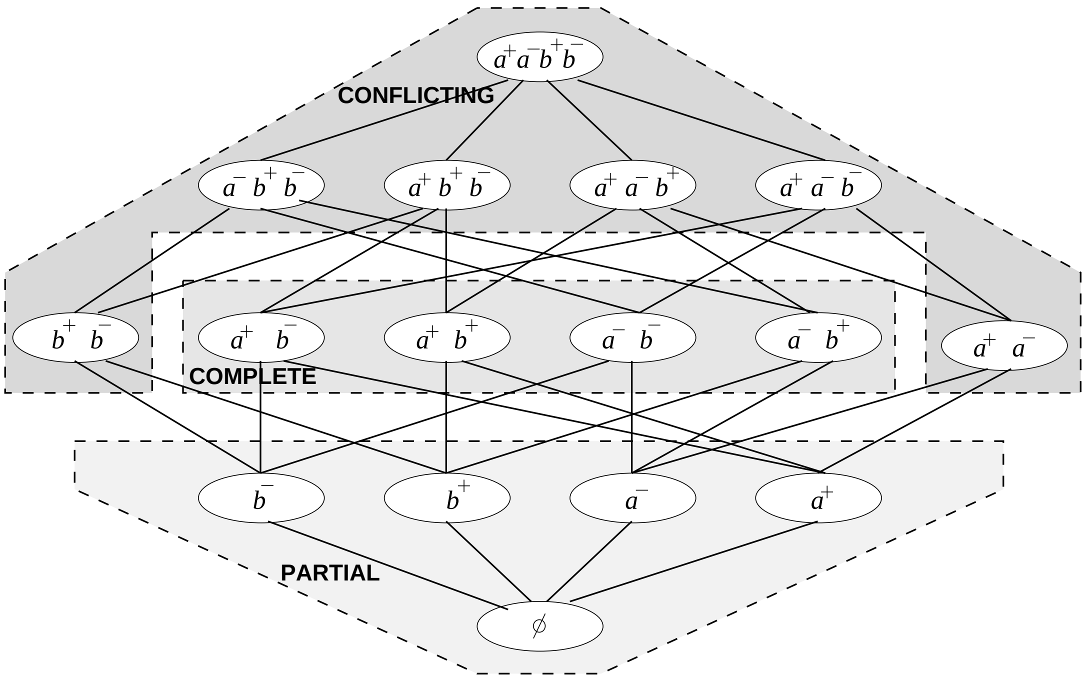

Figure 2.2: Structure of State Lattice for Two Node Circuit

both atoms n^- an n^+ are present. Such a state is physically unrealizable—it requires a signal to be both 0 and 1 simultaneously. Conflicting states are added to the state domain only for mathematical convenience. They extend the semilattice derived from a ternary system model into a complete lattice. Our state lattice has the empty set $\emptyset$ as its least element and the set of all atoms $\mathcal{A}$ as its greatest element. The set union and intersection operations are the *lub* and the *glb* operations, respectively, in the lattice.

Most traditional presentations of switch-level models describe circuit operation over the ternary domain $\mathcal{T}$. Each circuit node takes on one of the three distinct values from the set $\mathcal{T} = \{0, 1, X\}$, where the X value denotes an unknown or an indeterminate value. The *atom representation* of the state domain is closely related to the ternary domain. If a circuit state contains n^+, but not n^-, then node n has a value of 1. Similarly, the presence of n^-, and the absence of n^+ implies that n has a value of 0. If the circuit state contains neither n^+, nor n^-, then n has a value of

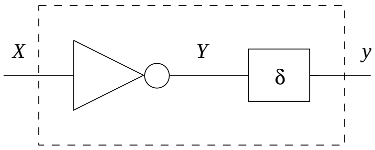

Figure 2.3: Y is the inverter excitation.

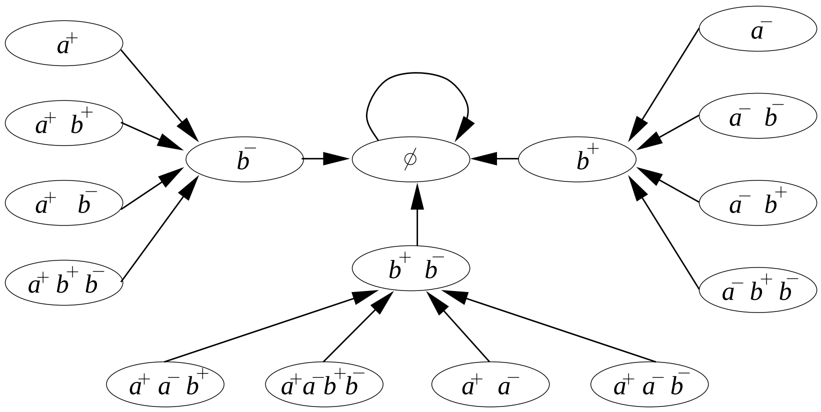

Figure 2.4: Excitation function of inverter in Figure 2.1.

X. For example, in the circuit of Figure 2.1, the set of atoms $\emptyset$, $\{a^+\}$, $\{b^-\}$, and $\{b^-, a^-\}$, represent the circuit states $(\mathsf{a} = X, \mathsf{b} = X)$, $(\mathsf{a} = 1, \mathsf{b} = X)$, $(\mathsf{a} = X, \mathsf{b} = 0)$, and, $(\mathsf{a} = 0, \mathsf{b} = 0)$, respectively. A circuit state such as $\{b^+, b^-\}$ is a conflicting state. Such conflicting states are mapped to a top element, $\top$, which is added to $\langle \mathcal{T}, \sqsubseteq_{\mathcal{T}} \rangle$ to extend it to a lattice [117].

are which contain positive and negative atoms for a node are mapped to a single element $\top$.

Since we attempt to model the behavior of physical systems with delays, the term *excitation function* is more appropriate for describing the function which specifies how circuit state is updated [28, pp.165-166]. Given a physical inverter with input x, output y and a delay δ, it may be modeled as an ideal zero-delay inverter, with input x, and output Y, followed by a delay element with input Y, and output y, as shown in Figure 2.3. The fictitious signal Y is the excitation of the physical inverter.

The behavior of a circuit is defined by its *excitation function* $Y\colon \mathcal{S} \to \mathcal{S}$. This function serves a role similar to the transition relation or next-state functions of temporal logic model checkers. We require this function to be monotonic over the information ordering, i.e., if two states are ordered $s_1 \subseteq s_2$, then their excitations must also be ordered: $Y(s_1) \subseteq Y(s_2)$. Intuitively, we can view a state as defining a set of constraints on the signal values. We require the excitation function to remain consistent as more constraints are applied. Since input nodes in a circuit are not constrained by the circuit itself, for any state $s \in \mathcal{S}$, $Y(s)$ does not contain any atoms corresponding to the input nodes. Figure 2.4 shows the excitation function for the inverter of Figure 2.1. A circuit model is defined by its lattice-structured state-set, and a monotonic excitation function over this state-set. Formally,

Definition 3 *A circuit model $\mathcal{M}$ is a tuple $\mathcal{M} = \langle \mathcal{S}, Y \rangle$, where $Y : \mathcal{S} \to \mathcal{S}$ is a monotonic excitation function.*

The behavior of a circuit can be represented as an infinite sequence of states. We define a *circuit trajectory* to be any state sequence $\sigma = \sigma^0\sigma^1\ldots$ such that $Y(\sigma^i) \subseteq \sigma^{i+1}$ for all $i \geq 0$. That is, the state sequence obeys the constraints imposed by the circuit excitation function. Below, the sequences σ_1, and σ_2, where $(s)^*$ indicates an infinite repetition of state s, are both trajectories of the inverter of Figure 2.1.

$$\sigma_1 = \emptyset \;\; \{a^+\} \;\; \{a^-, b^-\} \;\; (\{b^+, a^-\})^*$$
$$\sigma_2 = \{a^-\} \;\; \{a^+, b^+\} \;\; \{b^-\} \;\; (\emptyset)^*$$

As can be seen, these trajectories always obey the constraints imposed by the excitation function of Figure 2.4 — any state that contains the atom a^+ (a^-, resp.) constrains the succeeding state to contain the atom b^- (b^+, resp.).

$L(\mathcal{M})$ denotes the set of all trajectories of a circuit model $\mathcal{M}$. $L(\mathcal{M}, z)$ denotes the set of all trajectories $\sigma = \sigma^0\sigma^1\ldots$ of $\mathcal{M}$ such that $z \subseteq \sigma^0$, i.e., all trajectories which start with a state which is more constrained than z. The set $L(\mathcal{M}, \emptyset)$ equals $L(\mathcal{M})$.

We can extend the $\subseteq$ ordering on elements of $\mathcal{S}$, to sequences of elements of $\mathcal{S}$. This extended ordering is denoted as $\sqsubseteq$. If $\sigma_1 = \sigma_1^0\sigma_1^1\ldots$, and $\sigma_2 = \sigma_2^0\sigma_2^1\ldots$, then $\sigma_1 \sqsubseteq \sigma_2$ iff for all $i \geq 0$, $\sigma_1^i \subseteq \sigma_2^i$.

2.1.3 Specification Language

In STE, the specification language consists of a set of *trajectory assertions*. The simplest form of trajectory assertion has the form $[A \Longrightarrow C]$, where A, and C are *trajectory formulas*. A, the *antecedent* of the trajectory assertion describes the stimulus to the circuit over time, and C describes the expected response.

Trajectory formulas (TFs) have the following recursive definition:

1. **Atoms**: For any node n, atoms n^+ and n^- are TFs.
2. **Conjunction**: $(F_1 \wedge F_2)$ is a TF if F_1 and F_2 are TFs.
3. **Domain restriction**: $(E \rightarrow F)$ is a TF if F is a TF and E is a Boolean expression.
4. **Next time**: $(\mathbf{X}F)$ is a TF if F is a TF.

The Boolean expressions occurring in domain restriction operators, having the form $E \rightarrow F$, give these formulas a symbolic character. They can be thought of as "guards," i.e., F must hold for the cases where E evaluates to true. For the theoretical development, however, it is convenient to first consider the form where E is restricted to be either 0 (false) or 1 (true). A *scalar* trajectory formula obeys this restriction throughout its recursive structure. The extension to the symbolic case then simply involves considering the valuation of the expressions for each variable assignment. **X** is the *next time* temporal operator which causes advancement of time by one unit.

The truth of a scalar trajectory formula F is defined relative to a model structure and a trajectory. Let σ and $\sigma^0\tilde{\sigma}$ both be members of $L(\mathcal{M})$. $\sigma \models_{\mathcal{M}} F$, the truth of F relative to model $\mathcal{M}$, and a trajectory σ is recursively defined as:

1. (a) $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} a^+$ iff $a^+ \in \sigma^0$.
   (b) $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} a^-$ iff $a^- \in \sigma^0$.
2. $\sigma \models_{\mathcal{M}} (F_1 \wedge F_2)$ iff $\sigma \models_{\mathcal{M}} F_1$ and $\sigma \models_{\mathcal{M}} F_2$.
3. (a) $\sigma \models_{\mathcal{M}} (1 \rightarrow F)$ iff $\sigma \models_{\mathcal{M}} F$
   (b) $\sigma \models_{\mathcal{M}} (0 \rightarrow F)$ holds for every σ.

4. $\sigma^0\tilde{\sigma}\models_{\mathcal{M}}\mathbf{X}F$ iff $\tilde{\sigma}\models_{\mathcal{M}}F$.

A *defining sequence* of a trajectory formula F, denoted by δ_F, is the weakest possible sequence of states "consistent" the restrictions specified by F. We clarify this below. The recursive definition of this sequence is given as:

1. (a) $\delta_a^+ = \{a^+\}\emptyset\emptyset\emptyset\ldots$
   (b) $\delta_a^- = \{a^-\}\emptyset\emptyset\emptyset\ldots$
2. $\delta_{F_1\wedge F_2} = lub(\delta_{F_1}, \delta_{F_2})$
3. (a) $\delta_{0\rightarrow F} = \emptyset\emptyset\emptyset\ldots$
   (b) $\delta_{1\rightarrow F} = \delta_F$
4. $\delta_{\mathbf{X}F} = \emptyset\delta_F$

While δ_F is not necessarily a trajectory, it can be shown that $\sigma\models_{\mathcal{M}}F$ iff $\delta \sqsubseteq \sigma$. A defining trajectory is the weakest sequence of states that can be constructed, given the constraints specified in a trajectory formula. For example, in the definition above, the defining trajectory for a formula consisting only of a^+, is a sequence, the first element of which is the set $\{a^+\}$, and the remaining elements are the empty set $\emptyset$.

While δ_F is not a trajectory, we may combine it with the successor function Y, to get the *defining trajectory*, τ_F, of F. It can be shown that τ_F is the unique weakest trajectory satisfying F. We outline the construction of τ_F ahead. Let $\delta_F = \delta_F^0\delta_F^1\ldots$. Let $\tau_F = \tau_F^0\tau_F^1\ldots$ be the defining trajectory. Then, the successive elements of τ_F are given by the following construction:

$$\tau_F^i = \begin{cases} \delta_F^0 & \text{if } i = 0 \\ \delta_F^i \cup Y(\tau_F^{i-1}) & \text{otherwise} \end{cases}$$

The truth of a trajectory assertion $[A \Longrightarrow C]$ is defined with respect to a model $\mathcal{M}$, and a set of trajectories L of $\mathcal{M}$. Expressed as, $L\models_{\mathcal{M}}[A \Longrightarrow C]$, it is defined to hold iff for all $\sigma \in L$, $\sigma\models_{\mathcal{M}}A$ implies $\sigma\models_{\mathcal{M}}C$. Often, L equals $L(\mathcal{M})$, the complete set of trajectories of model $\mathcal{M}$. That $[A \Longrightarrow C]$ is true for every trajectory in this set is denoted as $\models_{\mathcal{M}}[A \Longrightarrow C]$. The existence of a defining trajectory for every trajectory formula considerably simplifies the test for determining the truth of an assertion. Given an assertion $[A \Longrightarrow C]$, we can verify that it holds for all elements of $L(\mathcal{M})$ by performing the test $\delta_C \sqsubseteq \tau_A$. The key result of STE, stated ahead, has been proved by Bryant and Seger in [117].

Theorem 1 $\models_{\mathcal{M}}[A \Longrightarrow C]$ *iff* $\delta_C \sqsubseteq \tau_A$.

Thus, to verify the truth of an assertion, all we need to do is construct a defining sequence, and a defining trajectory, and check that the former is weaker than the latter. Furthermore, we need check only an initial prefix of the two sequences, which is equals the "depth" of C. The depth of a formula F, denoted by $d(F)$, equals the maximum nesting of the next time **X** operator. This is stated as corollary 1.

Corollary 1 $\delta_C \sqsubseteq \tau_A$ *iff* $\delta_C^i \sqsubseteq \tau_A^i$ *for* $0 \leq i < d(C)$.

2.2 The switch-level circuit model

The *switch-level model* abstracts digital metal-oxide semiconductor (MOS) circuits as a network of nodes connected together by bidirectional transistor "switches." This model expresses transistor conductances and node capacitances by discrete strength and size values, and node voltages by discrete states $\{0, 1, X\}$. It can capture many of the important low-level features in MOS circuits such as ratioed, complementary, and precharged logic, and bidirectional pass transistors.

Prior to development of the switch-level model, the *Boolean logic gate model* was a popular abstraction of logic circuits. This model consists of a set of unidirectional logic gates connected by memoryless wires, in modeling MOS circuits. However, the inability of this model to capture many aspects of MOS circuit behavior has contributed to the widespread popularity of switch-level models since their inception. Since memory arrays are designed at the transistor-level, switch-level models are particularly appropriate for modeling this class of circuits.

Ahead, we briefly describe the MOSSIM II switch-level model developed by Bryant [17]. This was the first comprehensive formal model of switch-level networks combining transistors of different strengths, nodes of different sizes, and various node states and transistors conductances into a uniform mathematical framework. A brief description of *symbolic Boolean analysis* of switch-level circuits follows this.

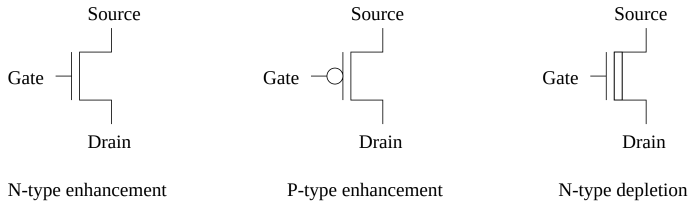

Figure 2.5: MOS transistors.

2.2.1 The model

In the switch-level model, a MOS transistor network consists of a set of nodes connected together by transistor switches. Nodes are of two types: *input*, and *storage*. An input node provides strong signals from sources external to the network, like power, ground and data inputs. Storage nodes are internal to the circuit, and they have states determined by the operation of the network and can retain these states in the absence of applied signals. Each storage node is assigned a size in the set $\{0,...,k\}$ to indicate in a highly idealized way its capacitance relative to other nodes with which it may share charge. The state of a node is represented by one of three logic values: 0, which indicates a low, 1 which indicates a high, and an X which represents an unknown or uninitialized value. Input nodes are assigned a size w, which is greater than the size of all the nodes and the strengths of all the transistors in the network.

A MOS transistor is a three terminal device with node connections *gate*, source, and drain. This device acts like a voltage controlled switch, depending on the value at its gate. Normally, there is no distinction between source and drain terminals – the transistor is a symmetric, bidirectional device. We distinguish between three types of transistors: n-type, p-type, and n-type depletion (Figure 2.5). A transistor acts as a switch between source and drain controlled by the state of its gate node. This switch may be open or closed, or it may have a conductance of unknown value. These three conduction states, open, closed and unknown are represented by the values 0, 1, and X respectively. Table 2.1 shows the conduction states of the three transistor types as a function of their gate node states. Each transistor has a strength in the set $\{k+1,...,w-1\}$. The strength of a transistor indicates its conductance when turned on relative to other transistors which may form part of a ratioed path.

Figure 2.6 shows a switch level circuit, consisting of the nodes a, b, c, k, s, t, p,

Gate	N-type	P-type	N-depletion
0	0	1	1
1	1	0	1
X	X	X	1

Table 2.1: Transistor state with gate values

Vdd, and GND, and the transistors T1, through T8. The node sizes and the transistor strengths are indicated by the numbers in parenthesis. The storage nodes in the circuit are s and t, which have sizes of 1, and 2, respectively. All transistors have a size of 3, except T6, which has a size of 4. The input nodes in the circuit, a, b, c, k, Vdd, and GND have a size of 5. The states of input nodes Vdd, and GND are fixed at 1, and 0, respectively.

Nodes in a switch-level network are connected together by directed *paths* of conducting transistors. Each path originates at a source node, and terminates at a destination node. The path has a *strength*, which roughly indicates the approximate amount of charge that can be supplied along the path from the source to the destination. In case of a path from an input node to a storage node, the strength of the path equals that of the weakest transistor in the path. In case of a path connecting two storage nodes, the strength of the path equals the size of the source node. The state of a node depends on the states of the source nodes of the strongest paths to this node.

2.2.2 The behavior of switch-level circuits

Most switch-level analysis and simulation tools partition the transistor-level network into a set of communicating components, termed *channel connected subnetworks* (CCSNs). Each CCSN consists of a set of storage nodes that can share charge, together with the transistors that connect them. Behavior within a CCSN can be difficult to analyze because of the bidirectional nature of transistors, and the multiple signal strengths. The interaction between the CCSNs is simpler. Each CCSN may be viewed as a sequential machine, with inputs, internal state, and outputs. The inputs

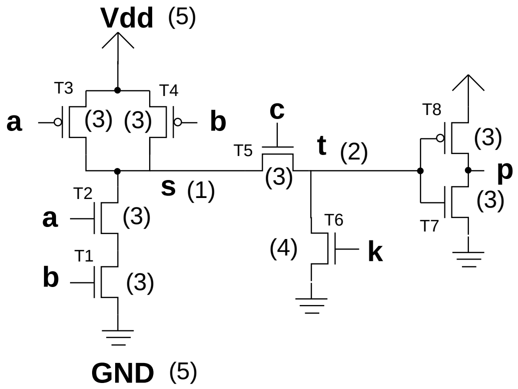

Figure 2.6: Example of a switch-level network.

consist of the transistor gate nodes, and input nodes connected to transistor source or drains. The storage nodes hold the CCSN state, and a subset of the CCSN nodes constitute the set of observable outputs of the CCSN. Given its initial state, and the present inputs, this sequential machine computes a new state, and new outputs. The entire transistor network is thus modeled as a system of communicating sequential machines. Figure 2.7 shows the circuit of figure 2.6 partitioned into CCSNs CCSN1, and CCSN2.

The *steady-state response* function of a CCSN describes its sequential behavior. This function specifies how the new CCSN node states are computed, given the initial storage node states, and the values at the inputs and the gates of the CCSN, and given that the transistor states are fixed long enough for the nodes to stabilize.

The *excitation function* of a CCSN gives the steady-state response of the CCSN nodes, when the transistors are held fixed in states determined by the initial storage node states and the inputs. An important property of the excitation function is its *monotonicity* over the $\{0, 1, X\}$ values. This property implies that if some inputs of this function were set to X, and a given output were 0 or 1, then changing the X inputs to 0 or 1 does not alter the output. This property is particularly important for verification, in view of the "information-content" ordering of the three values.

We use the *unit-delay model* to describe circuit delays. In this approach, a change in the state of a transistor gate is reflected as a change in the state of the transistor after a delay of one *time-step*. This time-step is used as the unit of time. In each

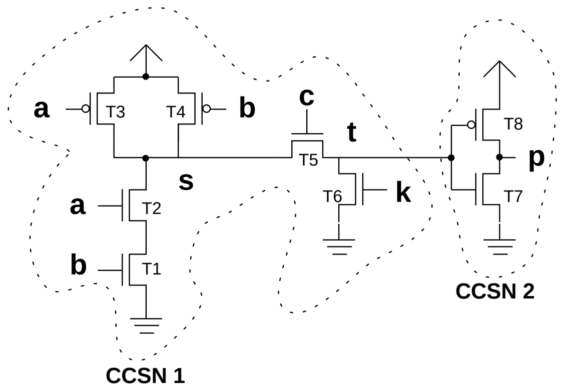

Figure 2.7: Circuit partitioned into CCSNs CCSN1 and CCSN2.

CCSN, given the logic levels at the inputs, and the storage nodes, a new logic level is computed for each storage node according to the CCSN excitation function. Then, after a delay of one time-step, the storage nodes are assigned the new logic levels just computed.

2.2.3 Computing the steady-state response

The steady-state response of a switch-level network can be obtained by a *symbolic Boolean analysis* of the network [19, 18]. In this approach, the problem of determining the network response is cast in terms of determining paths through the channel graph of a CCSN. The result of the analysis is a number of Boolean expressions which state how the ternary state of the CCSN is updated in each time-step.

To express and compute ternary quantities in the switch-level model in terms of Boolean operations, a "dual rail" encoding is used. Each ternary quantity x, is represented by a pair of Boolean values, $x.L$, and $x.H$, as shown in Table 2.2. For each node n, we introduce two Boolean variables, $n.L$, and $n.H$. The analysis problem can be specified as: for each node n in the circuit, derive the Boolean formulas $N.H$, and $N.L$, which represent the encoded value of the steady-state response at the node as a function of the initial node states.

The goal of symbolic analysis of a network is to derive Boolean expressions indicating the conditions under which conducting paths are formed in the network. To

x	$x.H$	$x.L$
0	0	1
1	1	0
X	1	1

Table 2.2: Dual rail encoding of x.

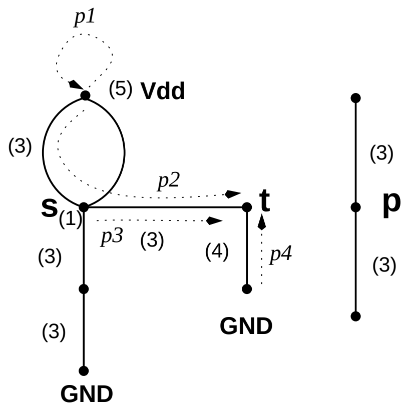

Figure 2.8: Rooted paths in channel graph.

express these concepts more precisely, we first define a *channel graph*, which has a vertex for each circuit node, and an edge for the channel of each transistor in a switch-level circuit. The CCSN defined earlier in section 2.2.2, is a connected component in this graph. The analysis process examines one CCSN at a time. The discussion ahead pertains to the analysis of a single CCSN.

A *rooted path* in the channel graph is a directed path between two nodes. A rooted path p originates at $root(p)$, and it terminates at $dest(p)$. Paths have a length, equal to the number of edges in it. A path can have a length of 0. The *strength* of a path reflects its relative charge transfer capacity. Figure 2.8 illustrates paths of different types in the channel graph for the circuit of Figure 2.6. $p1$ is a path of zero length, with a strength of 5. $p2$ is a path of strength 3 from the input node Vdd to the storage node t. $p3$ is a path of strength 1 from s to t. Note that the difference in strengths of $p2$, and $p3$ arises from the fact that $p2$ conveys charge from an input node which can potentially supply unlimited amount of charge, whereas node s can only convey the stored charge it has through path $p3$.

A rooted path is termed a *definite path*, if none of the transistors in the path are in the X state. The steady-state response of a node depends only on the paths to the node that are not *blocked*. Informally, a path is blocked if some intermediate node in the path is the destination of a stronger definite path. The charge from the stronger path overrides the charge from the weaker path. For example, if the transistor corresponding to path $p4$ in Figure 2.8 is on, then, $p2$ and $p3$, the weaker paths to node t, do not affect the steady-state response of t. If all the unblocked sources of charge to a node drive it to 0 (or 1), then the steady-state response equals 0 (or 1). Otherwise, if unblocked sources drive a node to conflicting values, then the node's response equals X. Let $\{\mathsf{m}_1, \mathsf{m}_2, ..., \mathsf{m}_k\}$ be the the set of nodes which are the origin of unblocked paths to node n. Let $m_1.H$, $m_1.L$, $\ldots m_k.H$, and $m_k.L$ be the pairs of Boolean variables to encode the states of these nodes. If N is the steady state response of node n then, given the dual rail encoding, it may be encoded as

$$N.H = m_1.H \vee m_2.H \vee \ldots \vee m_k.H$$
$$N.L = m_1.L \vee m_2.L \vee \ldots \vee m_k.L$$

The analyzer works with signals of one strength level at a time. It starts with the input signals, which are of the highest strength and works downward, each time adding in the effects of the paths at the next lower strength. For each strength level $w > s \geq 1$, the analyzer sets up and solves using Gaussian elimination three systems of Boolean equations which yield formulas $N.H_s$, $N.L_s$, and $clear_s(n)$ for every storage node n. $N.H_s$ and $N.L_s$ express the steady state response at the node, when all paths of strengths s and higher have been accounted for. $clear_s(n)$ expresses the condition when node n is not the destination of a definite path greater than or equal to s. It is used to set up equations for the next lower strength level. Thus, in the last iteration, expressions $N.H_1$ and $N.L_1$ are obtained, and these equal the steady state response of n.

The Boolean expressions generated in the process of symbolic analysis are represented by directed acyclic graphs (DAGs), where leaves denote Boolean variables and constants, and nodes denote Boolean operations. Each node of the DAG represents a Boolean formula, and often there is considerable amount of sharing in a DAG

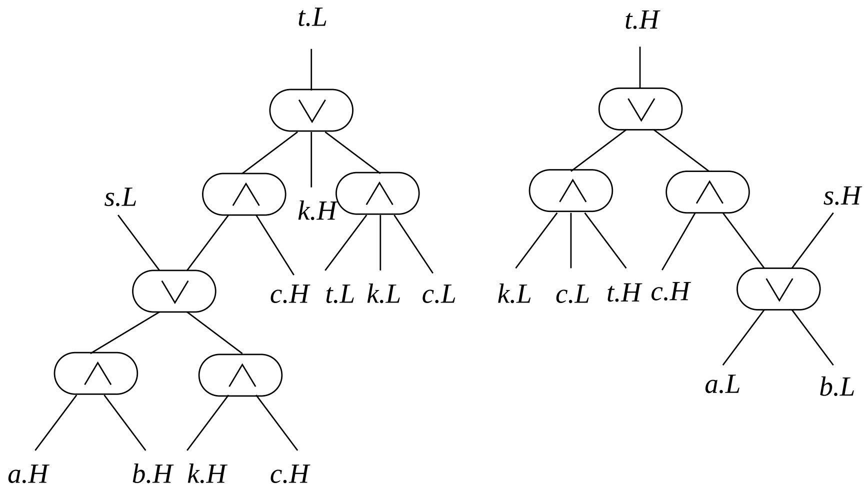

Figure 2.9: Results of switch-level analysis.

structure for the steady-state expressions for the storage nodes of a CCSN. Figure 2.9 shows the steady state expressions for nodes `s`, and `t` in component CSSN1 of the circuit in Figure 2.6.

Section 2.1.2 describes the behavior of systems by an excitation function $Y : \mathcal{S} \rightarrow \mathcal{S}$ over sets of atom. The system response obtained by the analysis above can be easily converted to our "atomcentric" point of view by a simple transformation. To ensure consistency between the two representations, this transformation ensures that if the high-rail $n.H$ (low-rail $n.L$) value of a node is 1, then the corresponding circuit state does not include n^- (n^+). Since the presence of n^- precludes the hi-rail from being 1, (dual true for lo-rail, and n^+), the transformation below converts the dual-rail DAGs to an atomcentric DAG.

1. Transform every $x.L$ value to x^+, and every $x.H$ value to x^-.

2. In the DAG, convert AND nodes to OR nodes, and OR nodes to AND nodes.

Figure 2.10 illustrates this transformation for the DAG of Figure 2.9.

Many MOS circuits can be partitioned into CCSNs that are quite small, with fewer than a 20 transistors. However, in some cases CCSNs can contain thousands of transistors. This is the case for memory arrays, where the transistors in the core of the

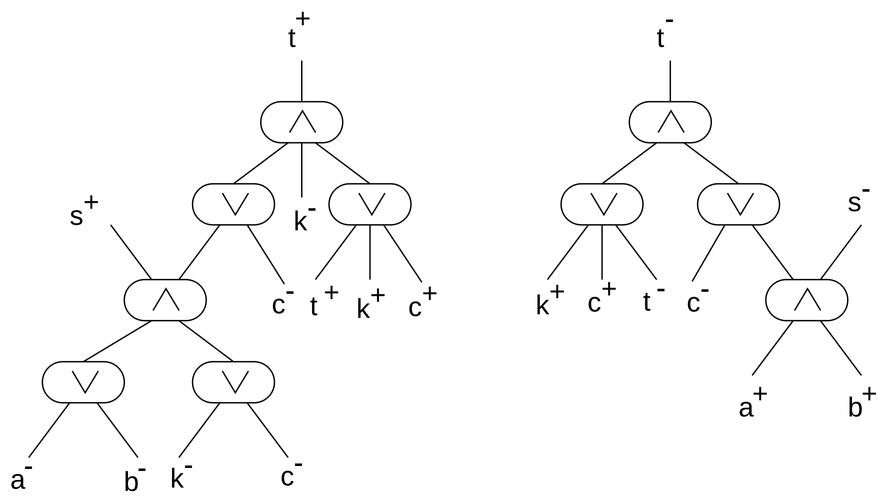

Figure 2.10: Atomcentric view of analysis results.

memory array form a large CCSN. Figure 2.11 shows the core of a 16-bit, 1 word/bit SRAM circuit, where the entire core consists of just two CCSNs. The transistors and the nodes included in one of these CCSNs have been shown in the shaded region of the figure. As can be seen, with increasing memory size, the size of the CCSN also increases proportionately. A 1K-bit SRAM circuit with a similar structure has two large CCSNs with over 3000 transistors each. Analysis of such CCSNs containing thousands of transistors is very expensive in terms of the time and memory required. This problem is discussed further, and a solution is described in Chapter 3.

2.3 Methodology for applying STE

STE is a powerful verification technique. However specifying system behavior directly as STE assertions is often awkward. These assertions include low-level details like circuit node names, detailed timing, including setup and hold times, clocking discipline etc. Such low-level details tend to obscure the abstract high-level behavior of the system which can be quite simple. This has motivated us to adopt a methodology for application of STE which was developed earlier by Beatty [6, 8]. In this section, we have discussed an outline of the methodology, and we have illustrated it with a small example.

•

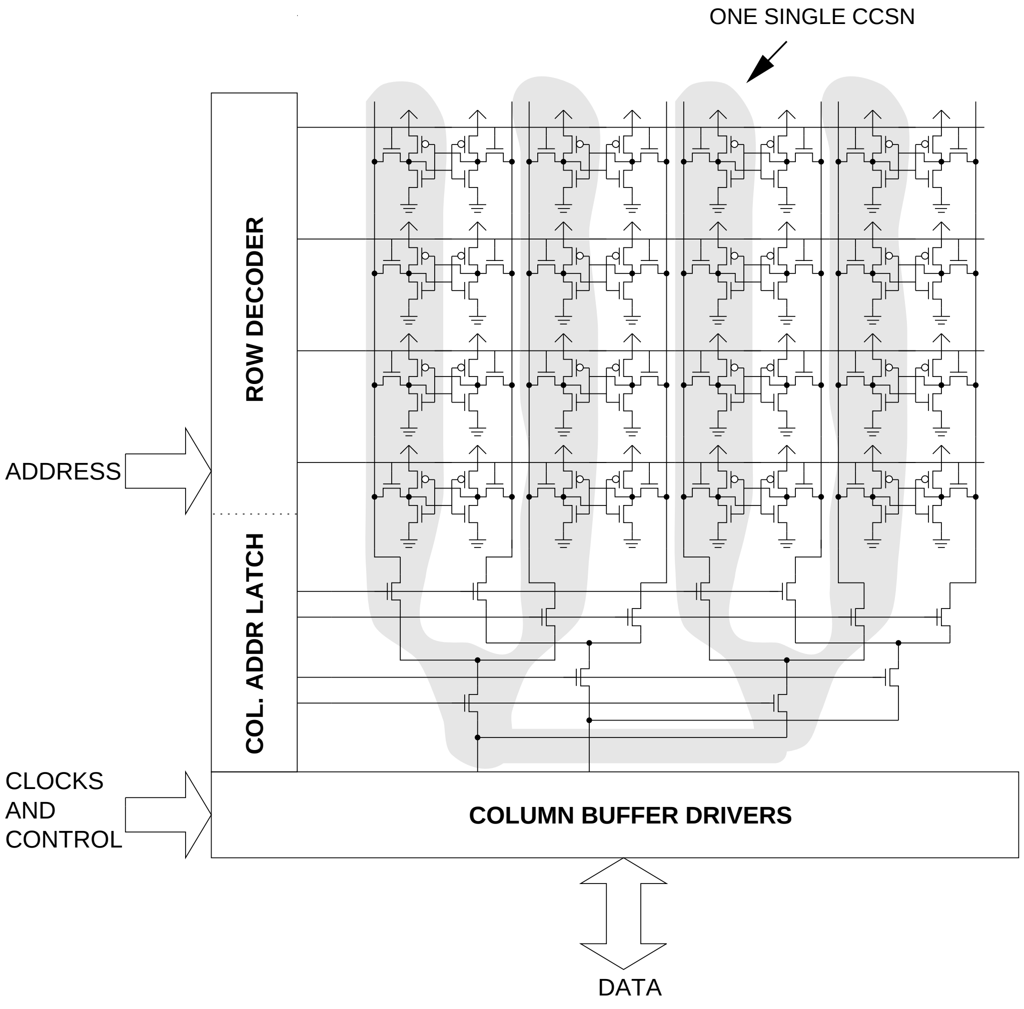

Figure 2.11: SRAM core CCSN

The methodology partitions the specification into two components. The first component is the *abstract specification*. It consists of a set of Hoare-like *abstract assertions* which specify the behavior of the system as a set of transitions over an abstracted system space. The second component is the *implementation mapping*, which describes how the abstract state is realized in the concrete implementation by mapping the abstract state values to values on circuit nodes at specified times. The abstract assertions are combined with the implementation mapping to yield STE assertions against which the circuit can be verified using STE. The overall goal of the methodology is to guarantee that the system being verified performs correctly under all execution sequences. The methodology establishes this guarantee by verifying individual operations and ensuring that they can be "concatenated" with others.

Ahead, section 2.3.1 sketches how the methodology establishes a relationship between the input-output sequences of an abstract finite-state specification, and the input-output signals of a concrete sequential design. For more details, the reader is referred to Beatty's thesis [6]. Following this, section 2.3.2 illustrates the methodology with an example.

2.3.1 Methodology Basis

Systems are modeled as *non-deterministic agents*, which accept input sequences and produce output sequences. Switch-level circuits, and abstract assertions both define non-deterministic Moore machines, and these can be viewed as agents. The notion of an implementation realizing a specification is defined ahead.

A realization (or design) **R** *implements* a specification **S** with respect to a mapping **I**, if and only if for any input sequence the specification might see, and any way of encoding that input for the design, any response the design produces must be a possible way of encoding some plausible response that the specification could have produced from the original input (Figure 2.12). This language containment property of agents is termed as the *obedience* property. In order to prevent trivial designs from being passed off as implementations, it is necessary for the mapping **I** to be *distinct*. Another non-triviality condition of mappings, *conformity*, is necessary to ensure that every legal abstract input sequence can be applied to the system.

While we have talked about only the input-output behavior of systems, agents usually have internal state. Thus, in order to show that one agent behaves according

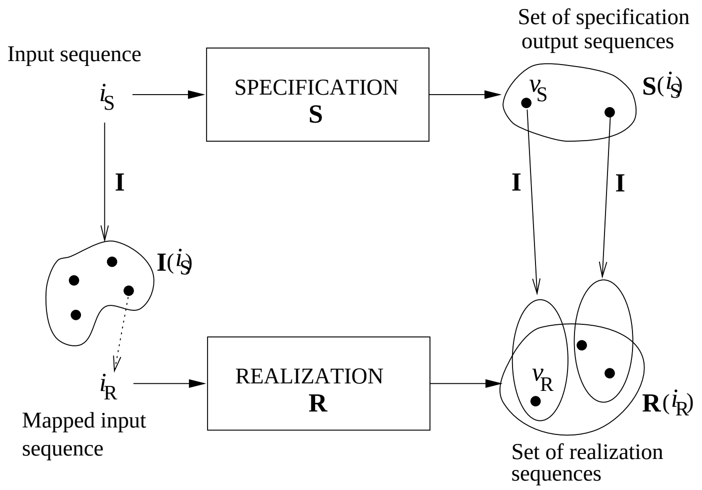

Figure 2.12: Set containment relationship between specification and realization.

to another, it is necessary to expose the internal state of the system. The development of the methodology in [6] proves that if we were to expose the internal state, and establish the obedience result for the agents with exposed state, the obedience result would still apply to the original agents.

A pair of states of a system, such that the system can make a transition from the first to the second, is termed a *transition* of the system. These transitions constitute the set of generators of the language, i.e., the system behavior. Therefore, in order to show behavior containment of the realization by the specification, it is sufficient to show containment of the realization transitions by the corresponding specification transitions. Since the realization can be pipelined and can have overlapped transitions, a *marked string* formalism was developed to take this into account. This last building block of the methodology bridges the gap between input-output sequences and individual system transitions.

2.3.2 Illustration of the Methodology

While the presentation in the previous section has been quite abstract, this section illustrates the pragmatic aspects of the methodology with a simple example.

Consider the 16-bit SRAM circuit in Figure 2.13. This circuit can read from a

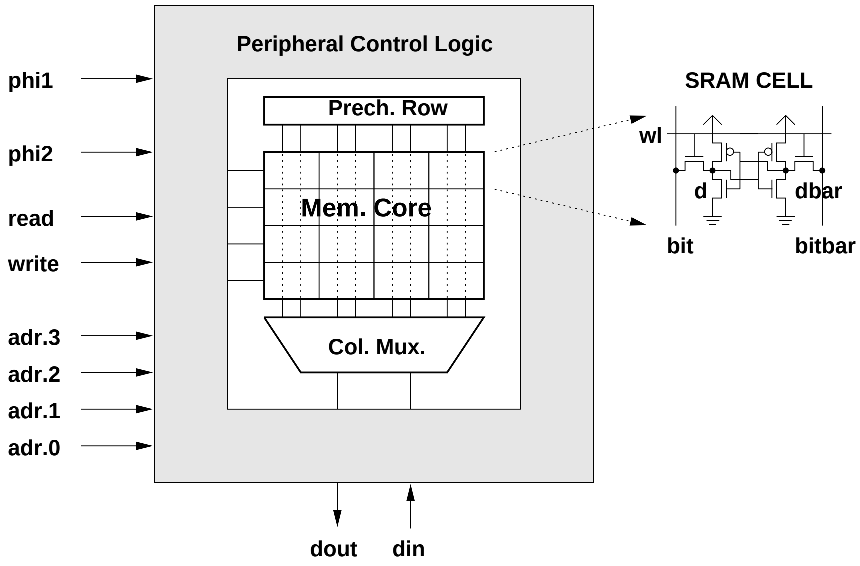

Figure 2.13: 16-bit SRAM

specified memory location, or write to a specified memory location. The read or write operation is specified by control signals at `read`, and `write`, and the address is specified at the address pins `adr.0` through `adr.3`. The circuit is clocked by a two phase non-overlapping clock at nodes `phi1`, and `phi2`.

Partitioning the high-level system functionality into a set of abstract operations of the system is the first step in the verification methodology. The *read* and the *write* operations are a natural part of the specification. However, it is also necessary to include the *nop* operation, to account for the system behavior when neither reads nor writes are being performed.

Specification of the abstract system state space, followed by a specification of each of the operations as a set of assertions over the abstract states is the second step in the methodology. The abstract state space is defined by a set of state variables. In our example, the abstract state includes state components `M[0..15]` corresponding to the state holding elements of the circuit. We also specify abstract state components `adr[0..3]`, `rd`, `wr`, `datain` and `dataout`, which correspond to the address pins, read and write pins, and the data in and data out pins in the circuit. Note that at the abstract level, we specify even the inputs as a part of the state. Such state components, however, are only controlled by the environment, and they can change non-deterministically. Rather than formally specify the syntax of assertions in detail, we illustrate their key aspects with simple examples (Details may be found in [6], or [71]).

Abstract assertions are of the form $[A \stackrel{\text{LEADSTO}}{\Longrightarrow} C]$, where the antecedent A defines a precondition on the system state, and the consequent C defines the constraints on the system state after one transition of the system. The syntax of A, and C is similar to that of the trajectory formulas, except that they do not contain the next time temporal operator $\mathbf{X}$. In fact, at the abstract state level, the abstract assertion $[A \stackrel{\text{LEADSTO}}{\Longrightarrow} C]$ may be thought of as a trajectory assertion of the form $[A \Longrightarrow \mathbf{X}C]$.

The read operation of the SRAM circuit described above may be specified as the following abstract assertion:

$$\begin{aligned}(\mathtt{adr} = i) \wedge (\mathtt{M}[i] = a) \wedge (\mathtt{rd} = 1) \wedge (\mathtt{wr} = 0)& \\ \stackrel{\text{LEADSTO}}{\Longrightarrow}& \\ (\mathtt{M}[i] = a) \wedge (\mathtt{dataout} = a)&\end{aligned}$$

Intuitively, this assertion specifies that if the value i appears at the address pins `adr`, the ith memory location contains the data value a, and a read is being performed, the value in the ith memory location remains unchanged, and the correct value appears at `dataout`. Symbolic values like a, and i are *case variables*, and they help specify a number of different combinations of non-symbolic values in a single symbolic assertion as above. In addition to the assertion above, it is also necessary to specify that reads are non-destructive, and the following assertion expresses this property.

$$(\mathtt{M}[i] = a) \wedge (\mathtt{rd} = 1) \wedge (\mathtt{wr} = 0) \stackrel{\text{LEADSTO}}{\Longrightarrow} (\mathtt{M}[i] = a)$$

Similarly, one may specify the write operation with a pair of assertions, the first of which states that a write to a given location updates the location correctly, and the second specifies that writes do not alter unaddressed locations. The nop operation simply involves showing that the data stored in the memory locations remains unchanged.

The third step in the methodology is specification of the *implementation mapping.* For the memory read operation we have illustrated the mapping in Figure 2.14. This

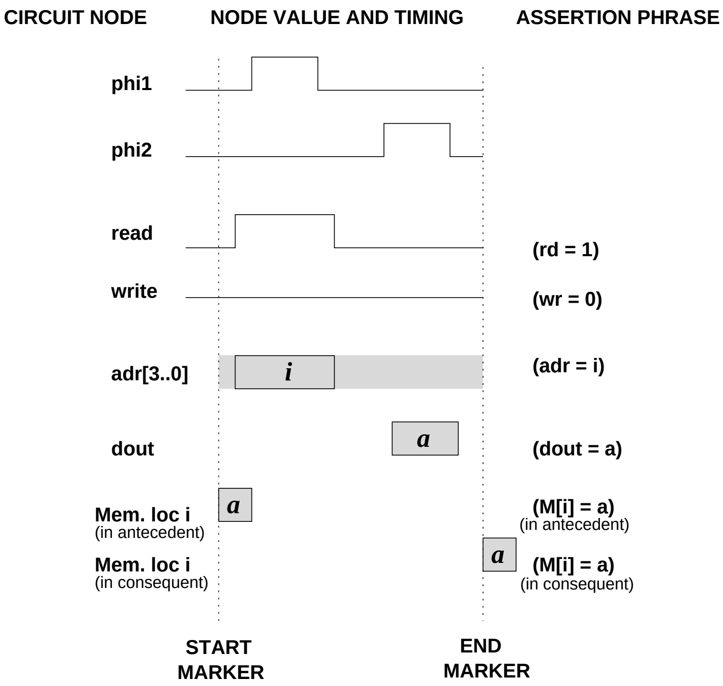

Figure 2.14: Implementation mapping.

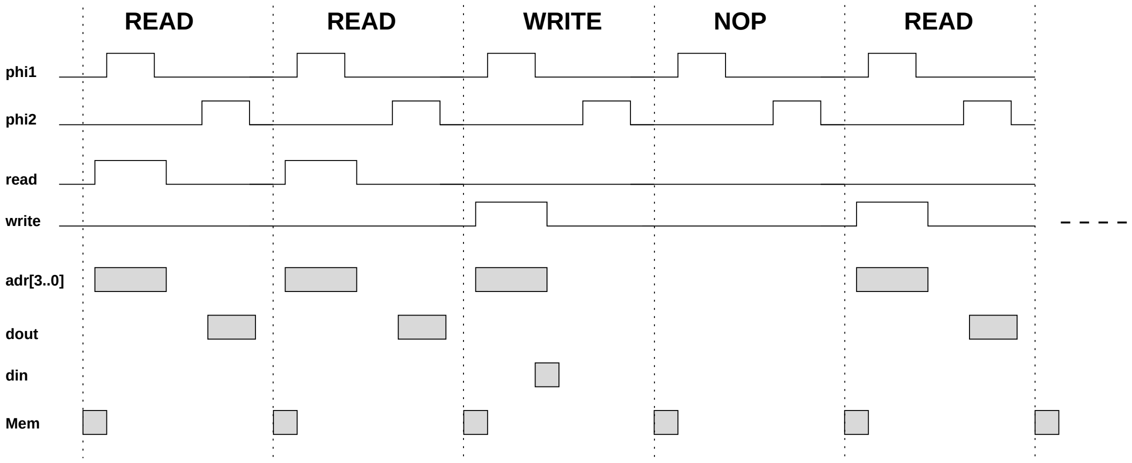

Figure 2.15: Sequence of SRAM operations aligned at markers.

mapping relates the abstract state to circuit timing and values on actual circuit nodes. The figure shows the actual circuit signals and their timings for each "phrase" in the abstract assertion. The mapping captures such implementation details as that a two phase clocking is used, and that the `read` signal is asserted and the address is provided when `phi1` is high. The dotted line at the left of the figure is the *start marker* indicating the nominal start of the operation, and the dotted mark to the right of the figure is the *end marker* indicating the end of the operation. These markers indicate how different fragments of circuit operation can be combined to yield a sequence of operations — the end marker of an operation must be lined up with the start marker of the next operation. Figure 2.15 shows a sequence of SRAM read, write and nop operations aligned at their start and end markers.

Note that variable i in the abstract assertion shown earlier is used as an array index. The implementation mapping represents it in binary form as a vector of symbolic Boolean variables. From the antecedent phrase ($\mathtt{M}[i] = a$), the implementation mapping will initialize each tag storage node k in the memory with a symbolic ternary function

$$f_k(i, Mem) = \begin{cases} tag, & \text{if } i = k \\ X, & \text{otherwise} \end{cases}$$

where X is the ternary constant of switch-level simulation. This technique, called *symbolic indexing*, is critical to the efficiency of STE on many memory arrays [6]. It is responsible for reducing the number of variables STE considers to a number logarithmic in the number of array locations. Symbolic indexing and more general ternary encodings are discussed further in Chapter 4.

2.4 Related work

2.4.1 Symbolic trajectory evaluation and its extensions

Our presentation of STE comes closest to the work by Bryant and Seger [117, 27]. However, recent work by Jain and Hazelhurst have extended the classes of properties that can be verified with STE based techniques.

In [72] Jain et al. have developed a more general form of trajectory assertions, where trajectory assertions are arbitrary *control graphs*, which consist of state vertices labeled with antecedent and consequent node formulas. Thus, one can have nested and interacting loops, which allow the specification of complex non-deterministic interface protocols at system boundaries [95]. This new formalism, while interesting and important, is not essential for arrays which generally have simple deterministic interfaces.

In [64] Hazelhurst et al. have generalized the theory of STE from a binary temporal logic to a quaternary temporal logic called TL. Formulas of the temporal logic TL include negation, disjunction, and an until operator which is similar to its LTL counterpart. These richer set of operators, and the four different truth values of the quaternary logic allow the specification of a broader set of properties, as compared to the original STE formulation.

2.4.2 Switch Level Modeling and Simulation

Many different switch-level modeling and simulation algorithms have been described in literature. A small subset of these includes [17], [51], [52], [4], [62], and [22]. However, rather than describe these in any detail, we refer the reader to the comprehensive survey on this topic by Bryant [20].

2.4.3 Methodology for Applying STE

The foundations of the STE methodology we use here were laid down by Beatty in his thesis [6]. Recent work by Jain [72] has extended Beatty's methodology to support more complex forms of implementation mappings which have arbitrary graph structures and include non-determinism.

Chapter 3

Symmetry

The term *symmetry* has evolved from the Greek word *summetros*, which means "of like measure." The American Heritage Dictionary [2] defines it as:

> *A relationship of characteristic correspondence, equivalence, or identity among constituents of a system or between different systems.*

The use of symmetry is widespread in science. Its applications range from stereochemistry, to quantum mechanics. Theoretical physicists employ symmetry theory in attempts to reconcile all known fundamental forces of nature — electroweak force, strong nuclear force and the gravitational force, as a part of the *grand unification theory* [87].

The use of symmetry in formal verification is recent [38], [69], [50]. The regularity in the structure of memory arrays strongly suggests the use of symmetry in their verification to cut down on space and time. For example, consider the verification of a SRAM circuit. If all memory locations are identical in the circuit, intuition suggests that it ought to be sufficient to verify only one memory location in the circuit. In this chapter, we formalize the notion of symmetry in a circuit. We show how symmetry can be exploited with STE to efficiently verify circuits. We illustrate the verification of a SRAM circuit using these ideas.

Section 3.1 describes our notion of symmetry as an *excitation function preserving state transformation*. A discussion on how symmetry properties allow one to infer the validity of an entire set of assertions from one valid assertion follows this. Section 3.3 shows how structural, and data symmetries can be verified by circuit graph isomorphism checks, and symbolic simulation, respectively. Circuit partitioning and reduction via *conservative approximations* is covered in section 3.4, and the following section illustrates the use of the above ideas for verifying SRAM circuits.

3.1 Symmetries of a circuit

We express both circuit operation and the specifications in terms of sets of atoms. We can therefore express symmetries in a circuit and the corresponding transformations of the specification in terms of bijective mappings over atoms, named *state transformations*.

Definition 4 *A state transformation, σ, is a bijection over the set of atoms: $\sigma : \mathcal{A} \to \mathcal{A}$. We can extend σ to be a bijection over states by defining $\sigma(s)$ for state s as $\cup_{a \in s}\{\sigma(a)\}$.*

As the term suggests, a state transformation σ takes a system state s and alters it to a new state $\sigma(s)$. Since σ is bijective, σ^{-1} exists. Also, if σ_1, and σ_2 are state transformations, then their composition $\sigma_1\sigma_2$ is also a state transformations.

Two types of state transformations, which alter circuit state in a "structured" manner are particularly interesting. These are the *structural*, and *data* transformations. Below, $s[a_1/b_1, ..., a_n/b_n]$ denotes the state obtained by simultaneously substituting atoms $a_1, \ldots, a_n$ for the atoms $b_1, \ldots, b_n$, respectively, in state s.

Definition 5 *A* structural *transformation is a state transformation which swaps the atoms for two different nodes. For nodes n_1 and n_2, we write $\mathsf{n}_1 \leftrightarrow \mathsf{n}_2$ to denote the transformation consisting of the swappings: n_1^+ with n_2^+ and n_1^- with n_2^-. Given $\sigma = \mathsf{n}_1 \leftrightarrow \mathsf{n}_2$, and a circuit state s, $\sigma(s) = s[n_1^+/n_2^+, n_1^-/n_2^-, n_2^+/n_1^+, n_2^-/n_1^-]$*

Definition 6 *A* data *transformation involves swapping the two atoms for a single node. For node n, we write $n^\pm$ to denote the transformation consisting of the swapping of n^+ with n^-. Given $\sigma = n^\pm$, and circuit state s, $\sigma(s) = s[n^+/n^-, n^-/n^+]$*

Intuitively, a structural transformation exchanges two nodes of a circuit (by swapping their atoms), and a data transformation complements the value at a circuit node by altering a positive atom of a node to a negative atom, and vice versa. Thus, these transformations provide us with a convenient mechanism to express circuit structure and circuit data handling related issues. Composing structural and data transformations allows us to express a variety of circuit transformations. To simplify notation, we will denote more complex transformations as a list of elementary transformations.

Our unified view of state transformations and excitations as functions mapping states into states allows us to succinctly express symmetry in a circuit as an excitation preserving transform. This closely parallels the definition of symmetry in [38] as a transition relation preserving state permutation.

Definition 7 *A state transformation σ is a* symmetry property *of a circuit with excitation function Y when $\sigma(Y(s)) = Y(\sigma(s))$ for every state s.*

That is, the excitation of the circuit on the transformed state $\sigma(s)$ matches the transformation of the excitation of s.

Lemma 1 *Symmetry transformations have the following properties.*
1. If σ is a symmetry property, and Y is an excitation function then $Y = \sigma Y \sigma^{-1} = \sigma^{-1} Y \sigma$.

2. σ is a symmetry property if and only if its inverse σ^{-1} is a symmetry property.

3. If σ_1 and σ_2 are symmetry properties, then their composition $\sigma_1\sigma_2$ is also a symmetry property.

Proof:
1. Since σ is a symmetry property, for every state s, $\sigma(Y(s)) = Y(\sigma(s))$, i,e, $\sigma Y = Y\sigma$. Therefore $\sigma^{-1}\sigma Y = \sigma^{-1} Y \sigma$, i.e., $Y = \sigma^{-1} Y \sigma$.

2. $\Rightarrow$. Since σ is a symmetry property, $\sigma Y = Y\sigma$. Therefore, $\sigma^{-1}(\sigma Y)\sigma^{-1} = \sigma^{-1}(Y\sigma)\sigma^{-1}$, which reduces to $Y\sigma^{-1} = \sigma^{-1}Y$. The other direction can be proved similarly.

3. σ_1 and σ_2 are symmetry properties. Therefore, $Y = \sigma_1^{-1} Y \sigma_1$, and $Y = \sigma_2^{-1} Y \sigma_2$.

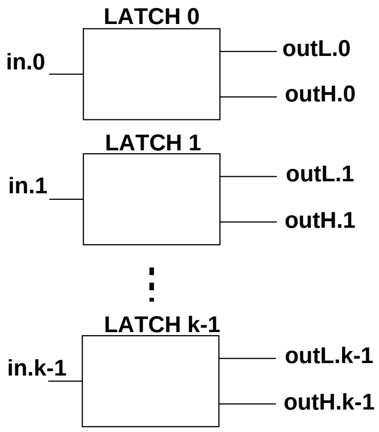

Figure 3.1: Illustration of the symmetries of a circuit

Substituting $\sigma_1^{-1} Y \sigma_1$ for Y in the second equation yields $Y = \sigma_2^{-1}(\sigma_1^{-1} Y \sigma_1)\sigma_2$, i.e., $Y = (\sigma_1\sigma_2)^{-1} Y \sigma_1\sigma_2$, i.e., $\sigma_1\sigma_2$ is a symmetry property. $\square$

Depending on their constituent state transformations, symmetry properties may be classified into one of the following three categories:

- *Structural symmetry* — consists entirely of structural transformations.
- *Data symmetry* — consists entirely of data transformations.
- *Mixed symmetry* — consists of both structural and data transformations.

Consider, for example, the circuit shown in Figure 3.1. This circuit consists of k identical latches. In each latch outL is a complement of the input, and outH has the same value as the input. Since the latches are identical, this circuit has a structural symmetry corresponding to the swapping of any pair of latches i and j, such that $0 \leq i, j < k$:

$$[\mathsf{in}.i \leftrightarrow \mathsf{in}.j, \mathsf{outL}.i \leftrightarrow \mathsf{outL}.j, \mathsf{outH}.i \leftrightarrow \mathsf{outH}.j]\,. \tag{3.1}$$

Each individual latch also stores data values 0 and 1 in a symmetric way, expressed for Latch 0 by the data symmetry:

$$\left[\mathsf{in}.0^{\pm}, \mathsf{outL}.0^{\pm}, \mathsf{outH}.0^{\pm}\right]. \tag{3.2}$$

Finally, each latch can also be viewed as a one-bit decoder—it sets one of its outputs high based on its input data. Such behavior for Latch 0 is expressed by a mixed symmetry:

$$\left[\mathsf{in.0}^{\pm}, \mathsf{outL.0} \leftrightarrow \mathsf{outH.0}\right]. \tag{3.3}$$

3.2 Verification under symmetry

Verifying a circuit involves checking a family of assertions against the circuit model. The presence of symmetry properties in the circuit often allows us to dramatically cut down on the number of assertions that need to be verified. This is because if an assertion G holds, and σ is a symmetry property of the circuit, then the assertion $\sigma(G)$ also holds. This is elaborated below.

We can extend σ to be a bijection over state sequences by applying σ to each state in the sequence. We can also extend state transformation σ to be a bijection over temporal formulas. First we define the extension of σ to Trajectory formulas(TFs).

Definition 8 *If F is a TF, then $\sigma(F)$ is recursively defined as*

1. *If F is the atom a, then $\sigma(F)$ is the transformed atom $\sigma(a)$.*
2. *$\sigma(F_1 \wedge F_2) = (\sigma(F_1) \wedge \sigma(F_2))$, where F_1 and F_2 are TFs.*
3. *$\sigma(E \rightarrow F) = (E \rightarrow \sigma(F))$, where F is a TF and E is a Boolean expression.*
4. *$\sigma(\mathbf{X}F) = (\mathbf{X}\sigma(F))$ where F is a TF.*

The effect of applying σ to a TF F is to replace every atom a in F by $\sigma(a)$. This idea is carried further, where σ may be applied to an assertion. The result of applying σ to the assertion $[A \Longrightarrow C]$ is the assertion $[\sigma(A) \Longrightarrow \sigma(C)]$, which is also denoted by $\sigma([A \Longrightarrow C])$.

Since a symmetry property is an excitation function preserving transformation, it follows fairly intuitively that the structure of the defining sequence and the defining trajectory of a TF F should remain invariant with respect to σ. This is formalized below in lemma 2, and lemma 3.

Lemma 2 *If temporal formula F has defining sequence δ_F, then its transformation $\sigma(F)$ will have defining sequence $\delta_{\sigma(F)} = \sigma(\delta_F)$.*

Proof: We prove this by induction over the structure of TFs.

- TF F is an atom a.

  $\sigma(\delta_F)$
  $= \sigma(\{a\}\emptyset\emptyset\emptyset\ldots)$ (From definition of δ_F)
  $= \sigma(\{a\})\emptyset\emptyset\emptyset\ldots$ (applying σ to every element of sequence)
  $= \delta_{\sigma(F)}$ (From definition of δ_F)

- $F = F_1 \wedge F_2$, where F_1, and F_2 are TFs.

  $\sigma(\delta_F) = \sigma(\delta_{F_1 \wedge F_2})$
  $= \sigma(lub(\delta_{F_1}, \delta_{F_1}))$ (Definition of δ_F)
  $= lub(\sigma(\delta_{F_1}), \sigma(\delta_{F_1}))$ (Bijection σ distributes over set union)
  $= lub(\delta_{\sigma(F_1)}, \delta_{\sigma(F_2)})$ (Inductive hypothesis)
  $= \delta_{\sigma(F_1) \wedge \sigma(F_2)}$ (Defintion of δ_F)
  $= \delta_{\sigma(F_1 \wedge F_2)}$

- If $F = 0 \rightarrow F_1$, where F_1 is a TF, it is easy to see that $\sigma(\delta_{0 \rightarrow F_1}) = \emptyset\emptyset\emptyset\ldots = \delta_{\sigma(0 \rightarrow F_1)}$.

- If $F = 1 \rightarrow F_1$, where F_1 is a TF, then

  $\sigma(\delta_{1 \rightarrow F_1})$
  $= \sigma(\delta_{F_1})$ (Definition of δ_F)
  $= \delta_{\sigma(F_1)}$ (Inductive hypothesis)
  $= \delta_{1 \rightarrow \sigma(F_1)}$ (Definition of δ_F)
  $= \delta_{\sigma(1 \rightarrow F_1)}$

- Finally, if $F = \mathbf{X}F_1$, where F_1 is a TF,

  $\sigma(\delta_{\mathbf{X}F_1})$
  $= \sigma(\emptyset\delta_{F_1})$ (Definition of δ_F)
  $= \emptyset\sigma(\delta_{F_1})$ (Apply σ to every element of sequence)
  $= \emptyset\delta_{\sigma(F_1)}$ (Inductive hypothesis)
  $= \delta_{\sigma(\mathbf{X}F_1)}$ $\square$

Lemma 3 *If σ is a symmetry property of a circuit model $\mathcal{M}$, then its defining trajectories for any temporal formula F will obey the symmetry: $\tau_{\sigma(F)} = \sigma(\tau_F)$.*

Proof: We prove this result by doing induction on the sequence of elements in the defining trajectory $\tau_F = \tau_F^0 \tau_F^1 \ldots \tau_F^i \tau_F^{i+1} \ldots$. In particular, we first show that $\sigma(\tau_F^0) = \tau_{\sigma(F)}^0$ (base case), and next we show that if $\sigma(\tau_F^i) = \tau_\sigma(F)^i$, then $\sigma(\tau_F^{i+1}) = \tau_\sigma(F)^{i+1}$(induction step).

- **Base case**:
  $\sigma(\tau_F^0)$
  $= \sigma(\delta_F^0)$ (Def. of defining trajectory)
  $= \delta_{\sigma(F)}^0$ (Lemma 2)
  $= \tau_{\sigma(F)}^0$ (Def. of defining trajectory)

- **Induction step**: Assuming $\sigma(\tau_F^i) = \tau_\sigma(F)^i$, and that σ is a symmetry property, we show that $\sigma(\tau_F^{i+1}) = \tau_\sigma(F)^{i+1}$.
  $\sigma(\tau_F^{i+1})$
  $= \sigma(\delta_F^{i+1} \cup Y(\tau_F^i))$ (Def. of defining trajectory)
  $= \sigma(\delta_F^{i+1}) \cup \sigma(Y(\tau_F^i))$ (Bijection σ distributes over set union)
  $= \delta_{\sigma(F)}^{i+1} \cup \sigma(Y(\tau_F^i))$ (Application of lemma 2)
  $= \delta_{\sigma(F)}^{i+1} \cup Y(\sigma(\tau_F^i))$ (Definition of symmetry property σ)
  $= \delta_{\sigma(F)}^{i+1} \cup Y(\tau_{\sigma(F)}^i)$ (Inductive hypothesis)
  $= \tau_F^{i+1}$

□

This brings us to the central theorem of this chapter, which is stated below.

Theorem 2 *For an assertion $[A \Longrightarrow C]$, and a symmetry property σ of model $\mathcal{M}$, $\models_{\mathcal{M}} [A \Longrightarrow C]$ if and only if $\models_{\mathcal{M}} [\sigma(A) \Longrightarrow \sigma(C)]$.*

Proof: The proof follows directly from the definition of a symmetry property, lemma 2, and lemma 3.

$\models_{\mathcal{M}} [A \Longrightarrow C]$
$\Leftrightarrow \tau_A \subseteq \delta_C$ (Theorem 1)
$\Leftrightarrow \sigma(\tau_A) \subseteq \sigma(\delta_C)$ (Bijection σ preserves the subset relation)
$\Leftrightarrow \tau_{\sigma(A)} \subseteq \sigma(\delta_C)$ (Lemma 3)
$\Leftrightarrow \tau_{\sigma(A)} \subseteq \delta_{\sigma(C)}$ (Lemma 2)
$\Leftrightarrow \models_{\mathcal{M}} [\sigma(A) \Longrightarrow \sigma(C)]$ (Theorem 1) □

Thus, proving that σ is a symmetry property of a circuit allows us to infer the validity of a transformed assertion once we verify the original. For example, suppose we verify that Latch 0 in Figure 3.1 operates correctly for input value 1, and also prove that the transformations defined by Equations 3.1 and 3.2 are indeed symmetry transformations. Then we can infer from Equation 3.1 that for all j, Latch j operates correctly for input value 1, and from Equation 3.2 that Latch 0 operates correctly for input value 0. Furthermore, by composing these two transformations, we can infer that for all j, Latch j will operate correctly for input value 0.

3.3 Verification of symmetry properties

To exploit symmetry, we verify an assertion $[A \Longrightarrow C]$, and given a set of symmetry properties, $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, we can conclude that $[\sigma_1(A) \Longrightarrow \sigma_1(C)]$, $[\sigma_2(A) \Longrightarrow \sigma_2(C)]$, $\ldots [\sigma_3(A) \Longrightarrow \sigma_3(C)]$ all hold. However, before drawing this conclusion, one must verify that every element of S is actually a symmetry property. The typical set of symmetry properties we work with is a group. Below, we give two basic definitions from group theory which we use ahead.

Definition 9 *Set S is a symmetry group for a model structure $\mathcal{M} = \langle \mathcal{S}, Y \rangle$ iff every element of S is a symmetry property of $\mathcal{M}$, and the following properties hold:*

1. *The identity element σ_e is in S, where $\sigma_e(a) = a$.*
2. *Every element of $\sigma \in S$ has an inverse $\sigma^{-1} \in S$ such that $\sigma\sigma^{-1} = \sigma^{-1}\sigma = \sigma_e$*

Definition 10 *A set $\langle S \rangle$ is termed a generator of a symmetry group S if repeated compositions of the elements in $\langle S \rangle$ can generate every element of S.*

The definitions above imply that rather than test every element of a set S for the symmetry property, it suffices to check only the generators of S. Thus, it is possible to prove the correctness of an entire set of assertions by simply verifying that each member of a set of generators for a group of transformations is a symmetry property.

For example, equation 3.1 represents a total of $k(k-1)/2$ symmetry transformations, corresponding to the pairwise exchange of any two latches. In general, one could argue that this circuit would remain invariant for any permutation π of the latches. Consider the transformation σ_π mapping the 6 atoms for each Latch i (two each for nodes $\mathsf{in}.i$, $\mathsf{outL}.i$ and $\mathsf{outH}.i$) to their counterparts in Latch $\pi(i)$. We could prove that each such transformation is a symmetry property, but this would require $k!$ tests. Instead, we can exploit the fact that any permutation π can be generated by composing a series of just two different permutation types. The "exchange" permutation swaps values 0 and 1, while the "rotate" permutation maps each value i to $i + 1 \bmod k$. Thus, proving that the state transformations given by these two permutations are symmetry properties allows us to infer that σ_π is a symmetry property for an arbitrary permutation π.

So, once the generators of a symmetry group are identified, the next step is to verify that they are indeed symmetry properties. We verify structural symmetries of a circuit by circuit graph isomorphism checks, and we verify data and mixed symmetries using symbolic simulation based techniques. We describe these techniques in sections 3.3.1, and 3.3.2.

3.3.1 Structural symmetry property verification

We can verify structural symmetries in our circuit models by checking for isomorphisms in the circuit-graph network. Since Anamos derives its representation of the excitation function from the network, any isomorphisms in the network graph imply structural symmetries in the excitation function. While it is also possible to verify structural symmetries using symbolic simulation, such an approach requires performing switch-level analysis on the circuit. This analysis can be prohibitively expensive (Table 3.1) for circuit components such as array cores with their large CCSNs.

Consider the problem of determining if $\mathsf{n}_1 \leftrightarrow \mathsf{n}_2$ is a symmetry property in a given circuit 3.2. If this symmetry were to hold in the circuit, the circuit 2 which is obtained from circuit 1 by swapping the labels of nodes n_1, and n_2 should be isomorphic to circuit 1. To perform this isomorphism test we use a graph coloring based algorithm described in [47, 46, 78, 7, 97]. This algorithm, which is described below, converts a circuit graph network into a *pseudo-canonical* form. This allows a fast and efficient test for isomorphism between two networks.

Given a the circuit graph for a switch-level circuit, we first construct a *coloring graph* which contains a *transistor vertex* for each transistor in the circuit, and a *node vertex* for each node and primary inputs in the circuit. The edges of this graph connect

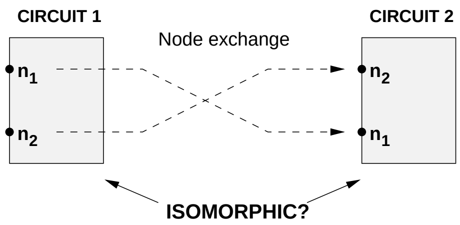

Figure 3.2: Structural symmetry verification problem

transistor vertices and node vertices, and they correspond to the node-transistor interconnections in the circuit. Since there are no edges which connect only two node vertices, or two transistor vertices, the coloring graph is bipartite. Once the coloring graph is constructed, the vertices of the graph are "colored" with integers. Based on isomorphism invariant vertex properties such as the number of edges incident on a vertex, all the vertices in the graph are assigned an initial color. Vertices are recolored repeatedly using a hashing function which combines the colors of the neighboring vertices, until all the vertices are colored uniquely. Then the nodes and transistors of the circuit are sorted to yield what is termed as the *quasi-canonical form* of the circuit network. This coloring and sorting technique guarantees that two networks that are not isomorphic will be colored differently. However, in some remote cases, it is also possible that two isomorphic networks may not be colored uniquely within a given fixed number of coloring iterations. However, we have not encountered this problem in our experiments (Section 3.6).

The problem we need to solve is slightly different that what is solved above. We need to swap two nodes in the circuit, and test if the two circuits, before and after the swap, are isomorphic. This does not work directly, as the isomorphism checks work purely on the structure of the network, and they ignore the node names. So the two circuits, before and after the node name swap will still reduce to the same quasi-canonical form. We work around this by using *structural labels*.

A structural label is a circuit graph (or an interconnection of nodes and transistors) which satisfies the following two properties:

- The structural label is not isomorphic to any subgraph of the original circuit.

- The label is not isomorphic to any other structural label.

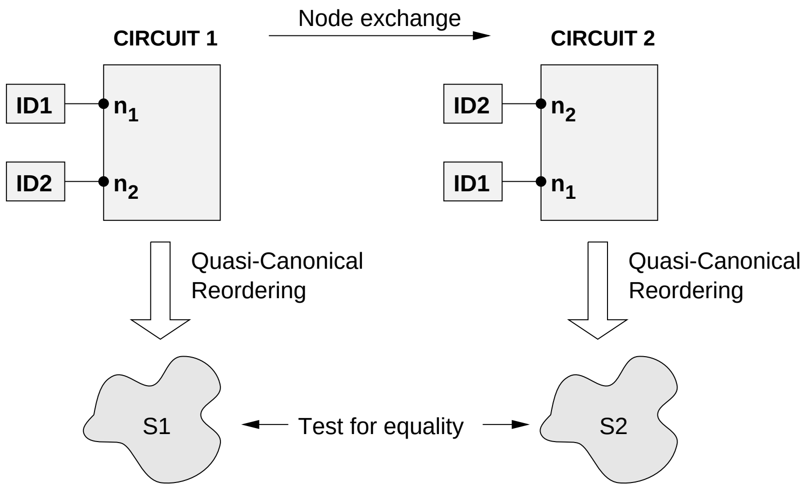

Figure 3.3: Verification of structural symmetry $\mathsf{n}_1 \leftrightarrow \mathsf{n}_2$

These properties allow us to use these labels to physically tag circuit nodes. Structural labels serve to uniquely tag a node in the circuit being tested for symmetry properties. So, structural labels are attached not only to the set of nodes being swapped, but also to the remaining circuit nodes. While verifying symmetry properties by graph isomorphism checks, it is necessary to attach structural labels to all the input and output nodes of a circuit. However, it is not necessary to consider the state nodes during our symmetry tests. If the circuit being verified satisfies all the assertions, then it has the desired IO behavior, and no further tests are necessary for the internal nodes. Of course, if some assertion fails, then further checks are necessary, and we may well discover a problem related to the state nodes.

For example, to solve the problem illustrated earlier in Figure 3.2, we can attach structural labels ID1 and ID2 to nodes n_1 and n_2 in circuit 1, and then flip these labels in circuit 2. We then can reduce the two circuits to the quasi-canonical form, and compare them (Figure 3.3).

The worst case time complexity for the graph coloring algorithm for a circuit with n transistors is $O(n^2 log(n))$. However, in practice we have seen both the time and memory scale nearly linearly with n. (Table 3.2).

3.3.2 Data and mixed symmetry property verification

Data and mixed symmetries can be verified by symbolic simulation. Data symmetries involve "switching polarities" of values on a node. Mixed symmetries involve "switching polarities" of the value on a node, and "exchanging" the values of two different nodes. To verify these symmetries it is necessary to check whether for every combination of circuit node values, the exchange and the polarity switching of the input node values results in changes in the output node values as specified by the symmetry property. Symbolic simulation is the ideal tool for such checks involving all possible combination of Boolean values. One symbolic Boolean variable is introduced for each circuit node. The circuit is simulated with these Boolean variables, and then with new Boolean values corresponding to the changes specified by the symmetry property. The results of the two simulations are compared to verify if the circuit obeys the symmetry property.

Consider, for example, the symmetry of Latch 0 specified in equation 3.2:

$$\left[\mathsf{in.0}^{\pm}, \mathsf{outL.0}^{\pm}, \mathsf{outH.0}^{\pm}\right].$$

We symbolically simulate the circuit with the symbolic value a at the input $\mathsf{in.0}$, and inspect the values at the output nodes $\mathsf{outL.0}$, and $\mathsf{outH.0}$. If the above symmetry holds for the latch, then complementing the value at node $\mathsf{in.0}$, should result in the values at circuit nodes $\mathsf{outL.0}$, and $\mathsf{outH.0}$ being complemented.

Consider also the mixed symmetry of Latch 0, specified in equation 3.2:

$$\left[\mathsf{in.0}^{\pm}, \mathsf{outL} \leftrightarrow \mathsf{outH.0}\right].$$

This symmetry can also be verified by symbolically simulating the circuit with a symbolic value a at the input $\mathsf{in.0}$, and observing the outputs, $\mathsf{outL.0}$, and $\mathsf{outH.0}$. The equation above specifies that complementing the value at $\mathsf{in.0}$ should result in the values at nodes $\mathsf{outL.0}$, and $\mathsf{outH.0}$ being exchanged, which can be easily checked.

3.4 Conservative approximations of circuits

Intuitively, a conservative approximation of a circuit is a "reduced" version of the circuit which, given any current circuit state, imposes fewer constraints on the next state the circuit can take. In terms of values, on circuit nodes, the reduced circuit produces more Xs than the original circuit, i.e., fewer atoms in the next state. We formalize this concept below.

Definition 11 *Let $\mathcal{M}'$ and $\mathcal{M}$ be circuit models over the same state set, having excitation functions Y' and Y, respectively. We say that $\mathcal{M}'$ is a* conservative approximation *of $\mathcal{M}$ if for every state s, $Y'(s) \subseteq Y(s)$. We denote this by $\mathcal{M}' \preceq \mathcal{M}$.*

We exploit conservative approximations to perform two important tasks:

- Create reduced models which take less memory to represent.
- Partition circuits to expose symmetric regions of a design.

Proving an assertion for a reduced circuit model, allows us to infer that the assertion holds for the original circuit. Theorem 3 below justifies this. The advantage of this is reduced circuit models are often a fraction of the size of the original circuit model, which results in smaller verification memory requirements. Conservative approximations provide a systematic way to reason about partitioned circuits, allowing us to verify the complete circuit by proving properties about each partition. This is particularly useful when the partitioning can expose highly symmetric regions of the circuit. In addition, if we can prove that a circuit has some structural symmetry, then we can create a "weakened" version of the circuit containing just enough circuitry to verify the behavior for one representative of the symmetry group.

3.4.1 Verification of Reduced Models

Below, we show that for any trajectory assertion $[A \Longrightarrow C]$, and models $\mathcal{M}$, and $\mathcal{M}'$, such that $\mathcal{M}' \preceq \mathcal{M}$, if the assertion $[A \Longrightarrow C]$ holds for $\mathcal{M}'$, then it should also hold for $\mathcal{M}$. We start with the proof of a simple result on the trajectories of $\mathcal{M}'$ and $\mathcal{M}$.

Lemma 4 *If F is any trajectory formula, and models $\mathcal{M}$, and $\mathcal{M}'$ are such that $\mathcal{M}' \preceq \mathcal{M}$, then the defining trajectories for the two models, τ'_F and τ_F, must be ordered $\tau'_F \sqsubseteq \tau_F$.*

Proof: Let $\tau_F = \tau_F^0 \tau_F^1 \tau_F^2 \ldots$, and $\tau'_F = {\tau'}_F^0 {\tau'}_F^1 {\tau'}_F^1 \ldots$. Using induction we show that for all $i \geq 0$, $\tau_F^i \subseteq {\tau'}_F^i$.

- **Base case**: From the definition of τ_F, $\tau_F^0 = {\tau'}_F^0 = \delta_F^0$, i.e., ${\tau'}_F^0 \subseteq \tau_F^0$.
- **Induction step**: Assuming ${\tau'}_F^i \subseteq \tau_F^i$ is true, we show that ${\tau'}_F^{i+1} \subseteq \tau_F^{i+1}$ also holds. Let $\delta_F = \delta_F^0\delta_F^1\delta_F^2\ldots$ be the defining sequence of F.
  ${\tau'}_F^i \subseteq \tau_F^i$
  $\Rightarrow Y({\tau'}_F^i) \subseteq Y(\tau_F^i)$ (Y is monotonic)
  $\Rightarrow Y'({\tau'}_F^i) \subseteq Y(\tau_F^i)$ ($\mathcal{M}' \preceq \mathcal{M}$)
  $\Rightarrow lub(\delta_F^{i+1}, Y'({\tau'}_F^i)) \subseteq lub(\delta_F^{i+1}, Y(\tau_F^i))$
  $\Rightarrow {\tau'}_F^{i+1} \subseteq \tau_F^{i+1}$

□

Therefore, as expected, "weakening" the model also weakens the trajectories of the model, and this immediately leads to the theorem below.

Theorem 3 *For any assertion* $[A \Longrightarrow C]$, *and* $\mathcal{M}' \preceq \mathcal{M}$, *if* $\models_{\mathcal{M}'} [A \Longrightarrow C]$, *then* $\models_{\mathcal{M}} [A \Longrightarrow C]$.

Proof Since $\models_{\mathcal{M}'} [A \Longrightarrow C]$, $\delta_C \sqsubseteq \tau'_F$ (Theorem 1). This result, when combined with $\tau'_F \sqsubseteq \tau_F$ (follows from lemma 4 above), gives $\delta_C \sqsubseteq \tau_F$, i.e., $\models_{\mathcal{M}} [A \Longrightarrow C]$. □

Thus, proving an assertion for a conservative approximation to a circuit model allows us to infer that the assertion holds for the original circuit.

3.4.2 Partitioning circuits via conservative approximations

We can view the partitioning of a circuit into different components as a process of creating multiple conservative approximations. For example, suppose we partition a circuit with nodes N into components having nodes N_1 and N_2, respectively, as illustrated in Figure 3.4. The set of nodes forming the interface between the components comprise the set $N_1 \cap N_2$. In this example, we assume the communication is purely unidirectional—N_1 generates signals for N_2. Suppose we wish to prove a property described by an assertion $[A \Longrightarrow C]$, where the atoms of C are contained only in N_2. We could then create conservative models $\mathcal{M}_1$ and $\mathcal{M}_2$ using the subset construction given by Equation 3.4 below. Taken individually, each of the two models is too weak to prove the assertion. Using the technique of waveform capture described ahead, we can record the output values generated by model $\mathcal{M}_1$ and use them in verifying the assertion with model $\mathcal{M}_2$. We describe this technique in greater detail below.

We start with the idea of creating conservative approximations by removing nodes of a circuit. Let N' be a subset of the set of circuit nodes N, and $\mathcal{A}'$ be the corresponding set of atoms. Then we can view the removal of those nodes not in N' as yielding a conservative approximation to the circuit with an excitation function Y' such that:

$$Y'(s) \quad = \quad Y(s \cap \mathcal{A}') \cap \mathcal{A}'. \tag{3.4}$$

Intuitively, $s \cap \mathcal{A}'$ eliminates atoms of all nodes other than in N', i.e., all sets the excluded nodes to X. The intersection of $Y(s \cap \mathcal{A}')$ with $\mathcal{A}'$ ensures that in the response, Y', atoms of all nodes other than in N' are eliminated.

Below, we first discuss the idea of *waveform capture*, and show how to construct a trajectory formula corresponding to the signal waveforms on a set of nodes.

Let N' be a subset of nodes in a circuit $\mathcal{M}$. Let $\tau_A = \tau_A^0 \tau_A^1 \tau_A^2 \ldots$ be the defining trajectory for the circuit $\mathcal{M}$ for a trajectory formula A. The technique of waveform capture records the occurrence of atoms on the nodes in N' as specified by the elements of the sequence τ_A, and it creates a temporal formula W_A describing this occurrence of atoms.

Let $\mathcal{A}_{N'}$ be the set of atoms corresponding to the nodes in N'. By eliminating all atoms from τ_A that are not in N', we construct a new sequence,

$$(\tau_A^0 \cap \mathcal{A}_{N'})\ (\tau_A^1 \cap \mathcal{A}_{N'})\ (\tau_A^2 \cap \mathcal{A}_{N'})\ \ldots.$$

From this sequence, we construct a trajectory formula

$$W_A = W_A^0 \wedge (\mathbf{X} W_A^1) \wedge (\mathbf{X}^2 W_A^2) \ldots \tag{3.5}$$

such that $W_A^i = a_1 \wedge a_2 \wedge \ldots \wedge a_k$, where $a_i \in (\tau_A^i \cap \mathcal{A}_{N'})$. The lemma below states the obvious consequence of such a construction.

Lemma 5 *Let A be a trajectory formula, and let N' be a subset of nodes of the circuit described by $\mathcal{M}$. If W_A is the trajectory formula constructed from τ_A, as described above, then $\models_{\mathcal{M}} [A \Longrightarrow W_A]$.*

Proof: Consider $\delta_{W_A} = \delta_{W_A}^0 \delta_{W_A}^1 \delta_{W_A}^2 \ldots$, the defining sequence of W_A. From the construction above, and the definition of defining trajectories, $\delta_{W_A}^i = \delta_{W_A^i}$, i.e.,

$\delta^i_{W_A} = (\tau^i_A \cap \mathcal{A}_{N'})$. Obviously, $(\tau^i_A \cap \mathcal{A}_{N'}) \subseteq \tau^i_A$. Therefore, $\delta_{W_A} \sqsubseteq \tau_A$, which proves $\models_{\mathcal{M}} [A \Longrightarrow W_A]$. $\square$.

Now we discuss the use of waveform capture to verify a property $[A \Longrightarrow C]$ for a larger design by partitioning it into smaller components and verifying these components individually. Prior to the discussion however, we prove theorem 4, which justifies waveform capture and combination. To simplify the proof of the theorem, we first show some useful results in lemma 6, 7, and 8.

Lemma 6 *If* $\models_{\mathcal{M}} [A \Rightarrow C]$, *then* $\models_{\mathcal{M}} [A \Rightarrow A \wedge C]$.

Proof: Since, $\models_{\mathcal{M}} [A \Rightarrow C]$, therefore, $\Rightarrow \delta_C \sqsubseteq \tau_A$, i.e., for $i \geq 0$, $\delta^i_C \subseteq \tau^i_A$. From definition of τ_A, for $i = 0$, $\tau^0_A = \delta^0_A$, and for $i > 0$, $\tau^i_A = lub(\delta^i_A, Y(\tau^{i-1}_A))$, i.e., for $i \geq 0$, $\delta^i_A \sqsubseteq \tau^i_A$. Therefore, for $i \geq 0$, $(\delta^i_A \cup \delta^i_C) \subseteq \tau^i_A$, i.e., $\delta_A \cup \delta_C \subseteq \tau_A$. Therefore, $\models_{\mathcal{M}} [A \Rightarrow A \wedge C]$.

Lemma 7 *If A and B are trajectory formulas, then* $\delta_B \sqsubseteq \tau_A \Rightarrow \tau_B \sqsubseteq \tau_A$

Proof We prove this by induction on elements of the sequences $\tau_A = \tau^0_A \tau^1_A \tau^2_A \dots$ and $\tau_B = \tau^0_B \tau^1_B \tau^2_B \dots$.

- **Base case**: $\tau^0_A = \delta^0_A$, and $\tau^0_B = \delta^0_B$. So, $\delta^0_B \subseteq \tau^0_A \Rightarrow \tau^0_B \subseteq \tau^0_A$ is trivially true.

- **Induction step**: Assuming that the relation holds for the *ith* element of the sequences, we show that the relation also holds for the $i+1$th element. Let $\delta^{i+1}_B \subseteq \tau^{i+1}_A$ be true. Below, we show that $\tau^{i+1}_B \subseteq \tau^{i+1}_A$.

$\tau^i_B \subseteq \tau^i_A$	(Inductive hypothesis)
$\Rightarrow Y(\tau^i_B) \subseteq Y(\tau^i_A)$	(Y is monotonic)
$\Rightarrow Y(\tau^i_B) \subseteq lub(Y(\tau^i_A), \delta^{i+1}_A)$	(Take the upper bound of $Y(\tau^i_A)$)
$\Rightarrow Y(\tau^i_B) \subseteq \tau^{i+1}_A$	(Definition of τ_A)
$\Rightarrow lub(Y(\tau^i_A), \delta^{i+1}_A) \subseteq \tau^{i+1}_A$	($\delta^{i+1}_B \subseteq \tau^{i+1}_A$ is an assumption)
$\Rightarrow \tau^{i+1}_B \subseteq \tau^{i+1}_A$	

$\square$

Lemma 8 *If* $\models_{\mathcal{M}} [A \Rightarrow B]$, *and* $\models_{\mathcal{M}} [B \Rightarrow C]$, *then* $\models_{\mathcal{M}} [A \Rightarrow C]$.

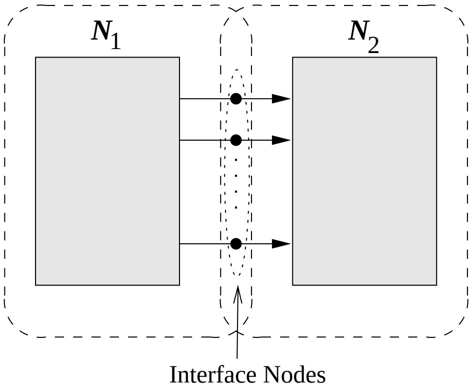

Figure 3.4: Illustration of Circuit Partitioning.

Proof From $\models_{\mathcal{M}} [B \Rightarrow C]$, and $\models_{\mathcal{M}} [A \Rightarrow C]$, we know that $\delta_B \subseteq \tau_A$, and $\delta_C \subseteq \tau_B$ are true. From $\delta_B \subseteq \tau_A$, and lemma 7, we can conclude that $\tau_B \subseteq \tau_A$. Therefore, $\delta_C \subseteq \tau_B \subseteq \tau_A$, i.e., $\models_{\mathcal{M}} [A \Rightarrow C]$ [1]. □

Theorem 4 *If $\mathcal{M}_C$ and $\mathcal{M}_D$ are conservative approximations of $\mathcal{M}$, and $\models_{\mathcal{M}_C} [A \Rightarrow F]$, and $\models_{\mathcal{M}_D} [A \wedge F \Rightarrow C]$, then $\models_{\mathcal{M}} [A \Rightarrow C]$.*

Proof: If $\models_{\mathcal{M}_C} [A \Rightarrow F]$, then $\models_{\mathcal{M}} [A \Rightarrow F]$ (from Theorem 3), and thus $\models_{\mathcal{M}} [A \Rightarrow A \wedge F]$ (from Lemma 6). Similarly, if $\models_{\mathcal{M}_D} [A \wedge F \Rightarrow C]$, then $\models_{\mathcal{M}} [A \wedge F \Rightarrow C]$ (Theorem 3). From this, using Lemma 8, we can conclude that $\models_{\mathcal{M}} [A \Rightarrow C]$.

Let τ_A^1 be the defining trajectory generated by model $\mathcal{M}_1$ for antecedent A. We construct a trajectory formula W_A describing the occurrence of the atoms corresponding to the nodes in $N_1 \cap N_2$ as described above. One refinement that should be performed is to record the values up to the maximum depth of the next-time operators in C (Corollary 1). Our construction ensures that $\models_{\mathcal{M}_1} [A \Longrightarrow W_A]$, and therefore, $\models_{\mathcal{M}} [A \Longrightarrow W_A]$. Using model $\mathcal{M}_2$, we then verify the assertion $[A \wedge W_A \Longrightarrow C]$, and if it holds, we know that $\models_{\mathcal{M}} [A \wedge W_A \Longrightarrow C]$ also holds. Effectively, we "play back" the waveforms on the interface nodes. Using theorem 4, we show that for any model $\mathcal{M}$ and any temporal formula F, if $\models_{\mathcal{M}} [A \Longrightarrow F]$ and $\models_{\mathcal{M}} [A \wedge F \Longrightarrow C]$, then $\models_{\mathcal{M}} [A \Longrightarrow C]$, and therefore this pair of verifications is sufficient to prove the desired property.

[1] Lemma 7 and 8 appear in a modified form in [63]

3.4.3 Partitioning with Bidirectional Communication

For partitions in which the communication between partitions is bidirectional, this approach described above can be generalized to an iterative process, creating a series of waveforms $W_1, W_2 \ldots, W_k$ representing successively stronger approximations to the communication patterns between the two partitions. As shown earlier in Figure 3.4, the two components in Figure 3.5 have nodes N_1, and N_2, and the set of nodes forming the interface between the two is $N_1 \cap N_2$. Their respective models are $\mathcal{M}_1$ and $\mathcal{M}_2$, which are both conservative approximation of the full circuit model $\mathcal{M}$.

As a first step towards proving, $\models_{\mathcal{M}} [A \Longrightarrow C]$, we show that $\models_{\mathcal{M}_1} [A \Longrightarrow W_1]$, i.e., $\models_{\mathcal{M}} [A \Longrightarrow W_1]$ where W_1 is constructed from the waveform on the nodes communicating from N_1 to N_2 (STEP 1). At each step, the dotted box around the partition indicates the active partition. In the next step, using waveform W_1, and antecedent A, we prove $\models_{\mathcal{M}_2} [A \wedge W_1 \Longrightarrow W_2]$, i.e., $\models_{\mathcal{M}} [A \wedge W_1 \Longrightarrow W_2]$ (STEP 2). From $\models_{\mathcal{M}} [A \Longrightarrow W_1]$, and $\models_{\mathcal{M}} [A \wedge W_1 \Longrightarrow W_2]$, we can infer $\models_{\mathcal{M}} [A \Longrightarrow W_2]$. In this manner, we generate successively "stronger" waveforms, until we reach the point where $\models_{\mathcal{M}} [A \Longrightarrow W_k]$, and $\models_{\mathcal{M}} [A \wedge W_k \Longrightarrow C]$ can both be shown true.

3.4.4 Creation of conservative approximations

Creation of a conservative approximation of a circuit intuitively means creating a reduced version of the circuit which produces more Xs than does the original circuit. The limiting case of the conservative approximation of a circuit is one whose every node produces only an X for every input sequence. Such a "strict" approximation is of little use, however. We would like to create conservative approximations in a more controlled manner, so that we can selectively disable desired portions of the circuit. The following two techniques are our means of doing so (figure 3.6):

- Attach "X-drivers" to internal circuit nodes.
- Strengthen transistors which are adjacent to X-drivers.

A X-driver is a strong source of Xs. Attaching a X-driver to a node is equivalent to converting the node to an input node set to the constant value X. The X-driver is analogous to the Vdd and ground nodes, which are strong sources of 1

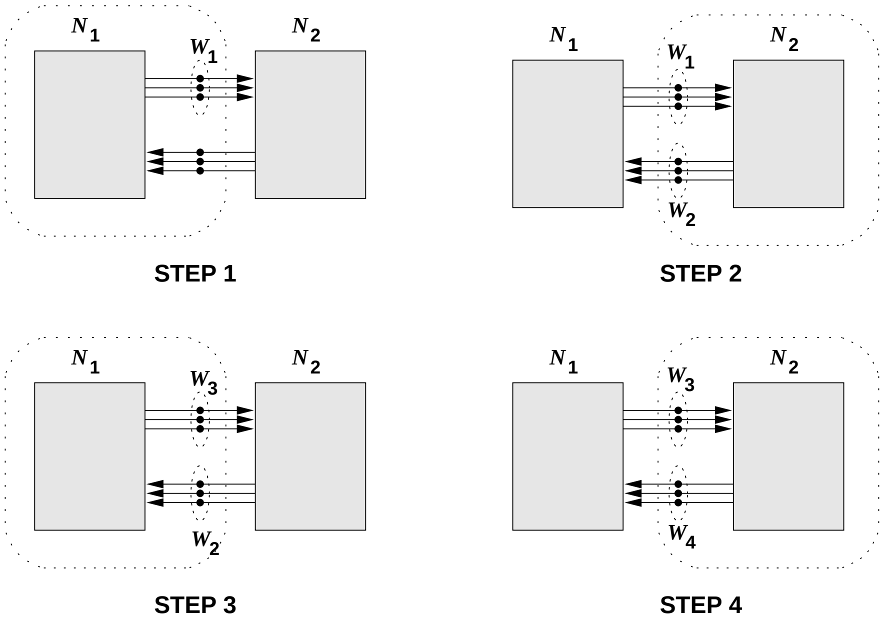

Figure 3.5: Bidirectional waveform capture

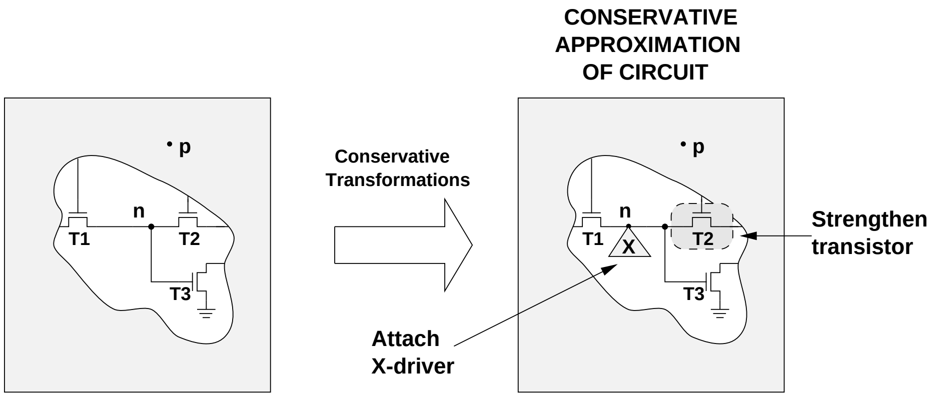

Figure 3.6: Creation of a conservative approximation of a switch-level circuit

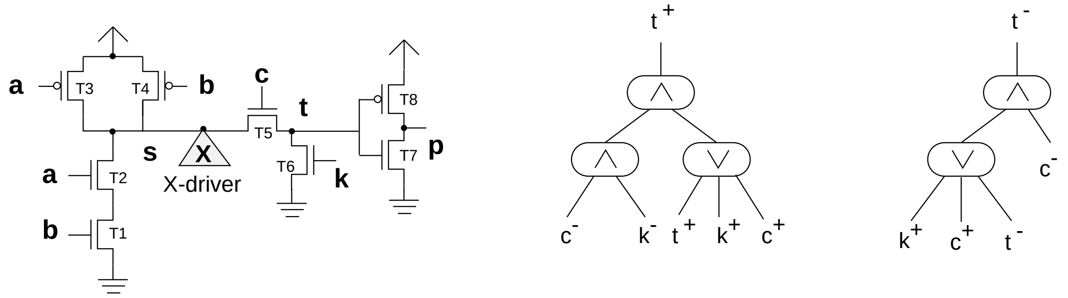

Figure 3.7: Conservative approximation of CCSN1.

and 0, respectively. An accompanying technique we employ to create a conservative approximation is strengthening transistors adjacent to X-drivers. Intuitively, this strengthening aids the propagation of Xs through the circuit. As will be shown later, both these techniques monotonically move all the circuit nodes towards Xs.

As an example, suppose we wish to create a reduced model for the circuit in Figure 2.7 by eliminating nodes a, b, and s. Then we could describe the remaining portions of CCSN1 by the excitation expressions shown in Figure 3.7. One can see that these expressions were obtained from those of Figure 2.7 by simplifying the result of setting the leaves for all eliminated atoms to false. This conservative approximation could be used to verify circuit operation for the cases where node c is set to 0. We have modified Anamos to generate these simplified expressions directly, avoiding the need to ever generate a complete model. In particular, we would replace node s in the example circuit by a X-driver.

One final task which remains is to show that the two circuit transformation techniques discussed above actually create a conservative approximation. As mentioned in section 2.2.3, the steady state response of a node in a switch-level network depends on all the unblocked paths to that node. In figure 3.8, consider the path p from src to dest. If the path were unblocked before the X-driver is attached to node n, then, two possibilities arise after the X-driver is introduced. The first is that the path still remains unblocked, and the node response stays the same. The second is that a stronger path q from n blocks p, i.e., $|q| > |p'|$, where p' is a prefix of p originating at the root. In this case, the stronger path q will dominate the response at dest, i.e., dest will monotonically move towards X. The effect of strengthening transistors

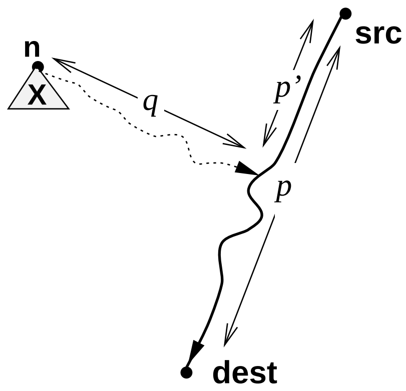

Figure 3.8: Paths in a switch-level circuit conservative approximation

adjacent to a X-driver is similar — stronger paths from the X-driver may dominate existing paths, sending their destination towards X. Note that the argument above also accounts for the case when n is on p. In such a case, the length of path q is 0.

3.5 Putting it together: Verifying a SRAM circuit

Consider the 16-bit (1 bit/word) SRAM circuit shown in Figure 3.9. This circuit consists of the the following major components — row decoder, column address latches, column multiplexer (Mux) and the memory cell array core. To simplify the discussion here, many essential SRAM components like precharge column, write-drivers etc. have not been shown in the figure. This is a standard organization followed in many larger industrial SRAM arrays [53]. In order to verify this circuit we must show that the read and write operations work correctly. For example, if a memory location is addressed and read from, then the correct value must appear at the output. Similarly, if a write operation is done, the addressed memory location should be updated correctly. Such properties can be expressed with STE assertions.

The machinery we have built in the previous sections allows us to verify the read or write operation for only one location and, from the symmetry in the SRAM circuit, conclude that the operation works for every location. We expand on this below,

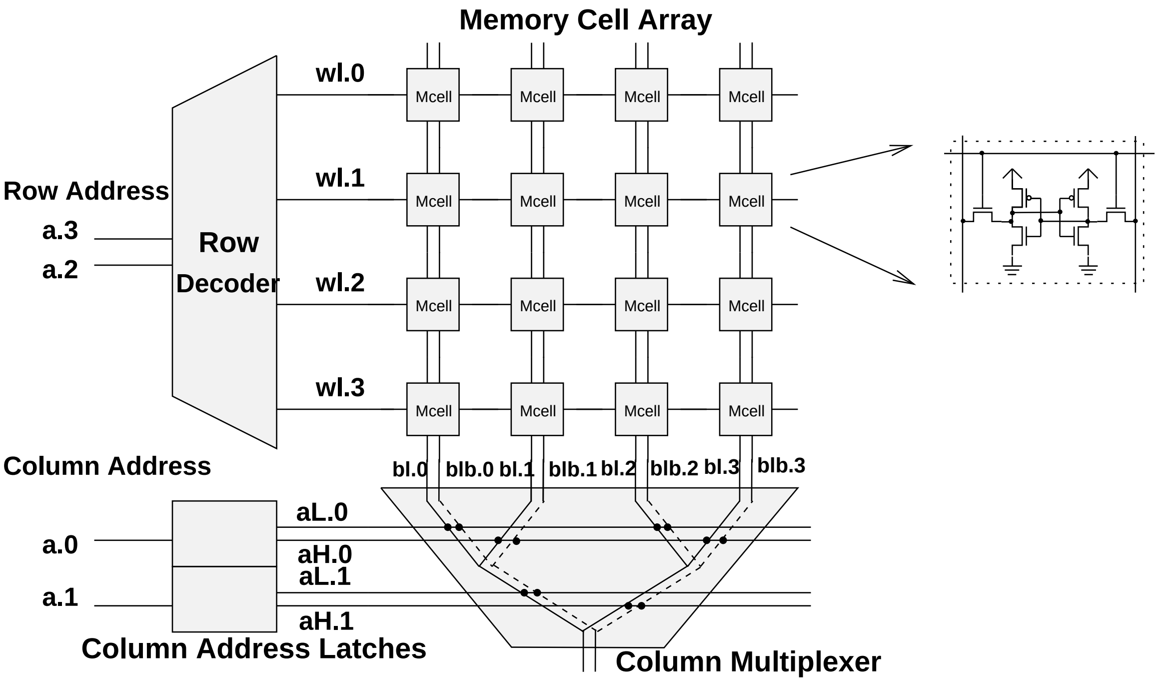

Figure 3.9: SRAM circuit

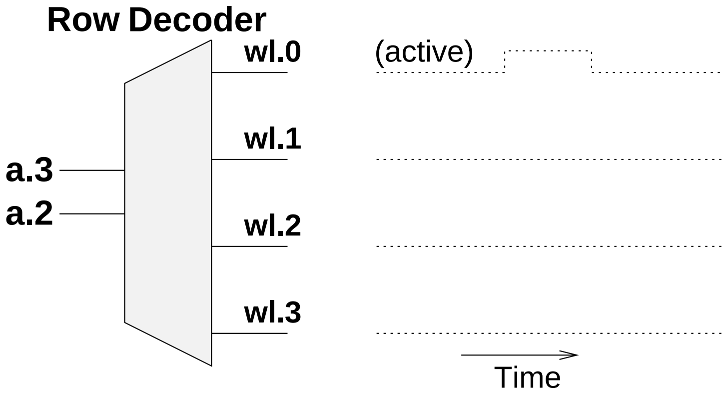

Figure 3.10: Row decoder and signal waveforms on word lines for row address 00.

starting with a discussion of SRAM symmetries.

3.5.1 Symmetries of a SRAM

Consider the decoder in Figure 3.10. For any memory operation, the value of the row address assigned to nodes a.2 and a.3, causes one of the word lines wl.0, wl.1, wl.2 and wl.3 to be active. The figure shows that wl.0 is active for row address 00. The same waveform occurs on the active word line regardless of the address. This mixed symmetry of the decoder is expressed by the group of transformations generated by transformations σ_0 and σ_1:

$$\sigma_0 = [\mathsf{a.2}^{\pm}, \mathsf{wl.0} \leftrightarrow \mathsf{wl.1}, \mathsf{wl.2} \leftrightarrow \mathsf{wl.3}]$$
$$\sigma_1 = [\mathsf{a.3}^{\pm}, \mathsf{wl.0} \leftrightarrow \mathsf{wl.2}, \mathsf{wl.1} \leftrightarrow \mathsf{wl.3}]$$

Transformation σ_i indicates that complementing bit i of the row address causes an exchange of signal waveforms for each pair of word lines j and k such that the binary representations of j and k differ at bit position i. The column address latches obey the "decoder" symmetry expressed by Equation 3.3.

The mixed symmetries of the decoder and the column address latches can be verified by symbolic simulation, where a single run of the simulator with n symbolic Boolean values at the circuit inputs is equivalent to 2^n runs of a conventional simulator with 0-1 values. For example, to verify that σ_0 is a symmetry of the decoder, we

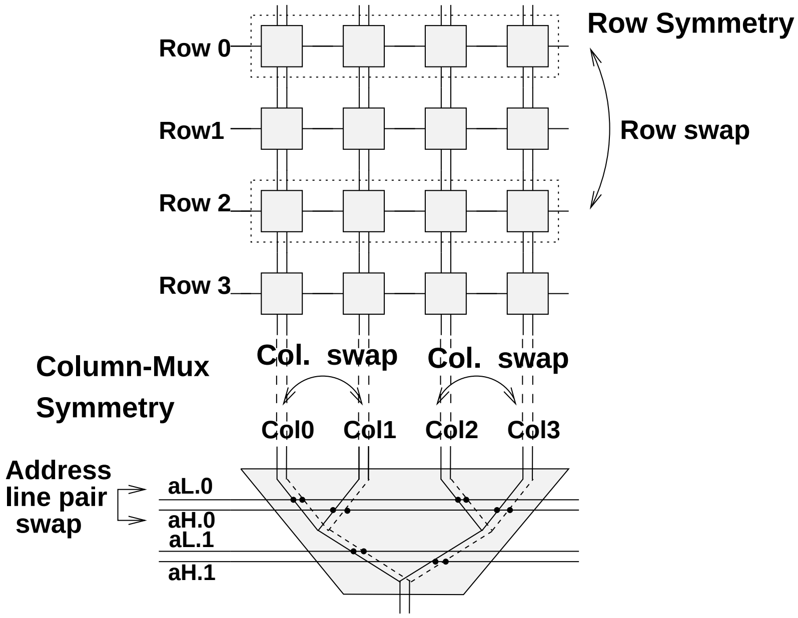

Figure 3.11: Structural symmetries of the SRAM core.

symbolically simulate the decoder with symbolic values s_0 and s_1 at the decoder inputs a.2, and a.3 in Figure 3.10. As the simulation proceeds, we check that a substitution of $\overline{s_0}$ for s_0 in the symbolic waveform for wl.0 (resp., wl.2) matches the symbolic waveform on wl.1 (resp., wl.3).

Figure 3.11 illustrates the two structural symmetries of the SRAM core and column Mux combination. The *row symmetry* arises from the invariance of the core-mux circuit structure under permutations of the rows of the core. The *column-mux symmetry* arises from the invariance of the circuit structure under a swap of column address latch output pairs accompanied by a corresponding exchange of columns. For example, in Figure 3.11, a swap of aH.0 and aL.0 accompanied by a swap of column 0 with 1, and a swap of column 2 with 3 is a symmetry of the circuit.

We verify the core-Mux symmetries in two parts. First we verify that arbitrary row and column permutations are symmetries of the core. Verification that the exchange and rotate permutation generators for rows and columns are symmetries suffices for this. This gives a total of 4 symmetry checks for the core. Next we verify the column-mux symmetry for the Mux. In the figure, the generators of the four different column

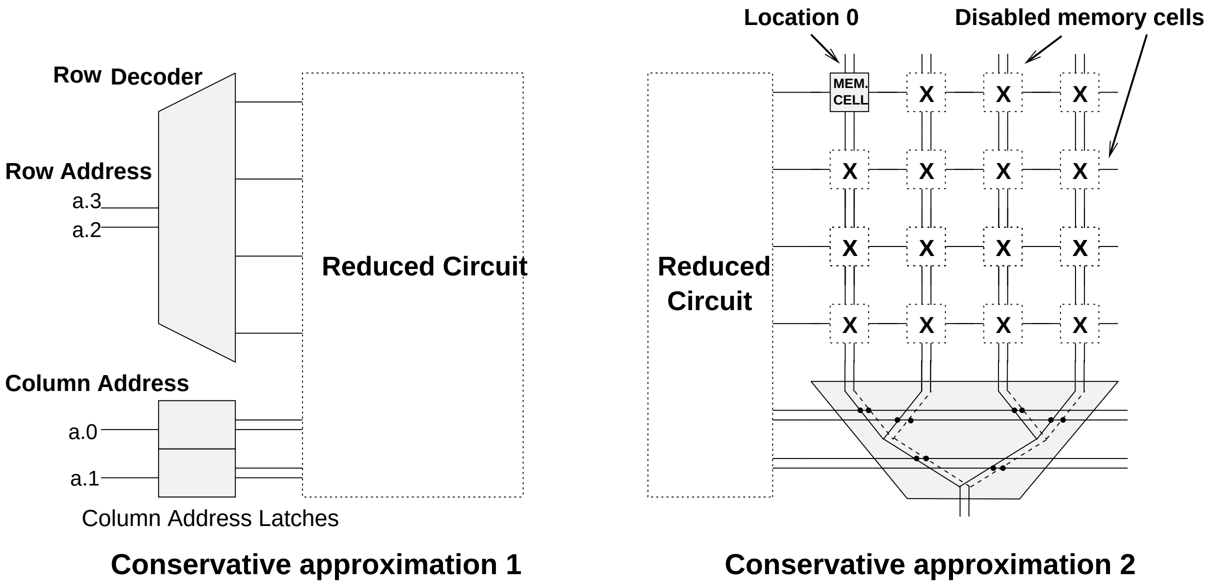

Figure 3.12: Conservative approximations of the SRAM.

address line pair permutations are the two permutations associated with each column address latch output pair. Therefore, two symmetry checks verify the column-mux symmetry. In general n symmetry checks must be done for the Mux in a SRAM with n column address line pairs.

3.5.2 Verification steps

In order to verify the SRAM circuit we go through the following sequence of steps.

1. **Circuit partitioning** — We partition the SRAM circuit into two parts. The first part consists of the decoder with the column address latches. The second part consists of the memory core and the column Mux.

2. **Symmetry verification** — Using symbolic simulation we verify the symmetries of the decoder and column latches. Using circuit graph isomorphism checks we verify the symmetries of the core and the column Mux.

3. **Conservative approximations** — We create two conservative approximations of the SRAM (Figure 3.12). In the first model, the memory core and the column

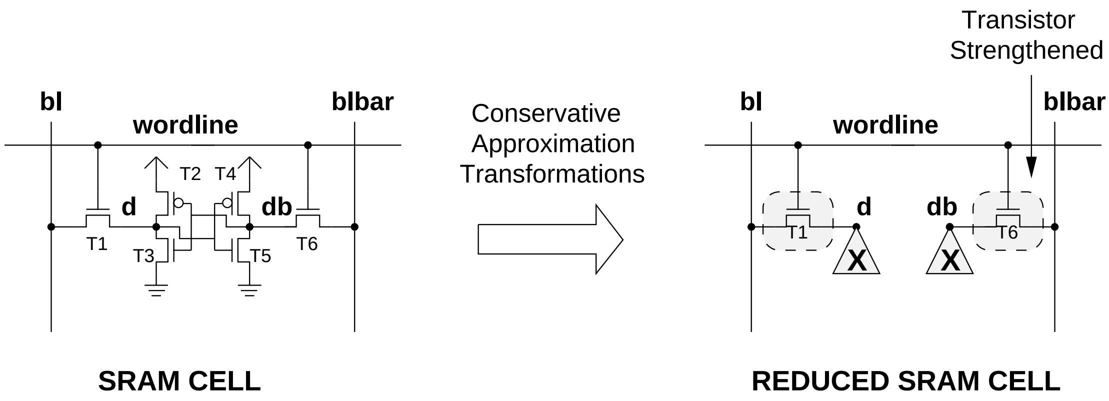

Figure 3.13: Creation of a conservative approximation of a SRAM Cell

Mux are "disabled". In the second model, the decoder, the column address latches are disabled, and all the memory cells except that for location 0 are disabled. Figure 3.13 shows how we conservatively disable a SRAM cell by attaching X-drivers, and strengthening transistors adjacent to the X-drivers. The figure also shows the optimization that chains of N or P transistors from Vdd or ground to an X-driver may be eliminated from the circuit — their presence does not alter circuit behavior.

4. **Waveform capture** — Given the assertion $[A \Longrightarrow C]$ specifying an operation for memory location 0, we use the antecedent A to symbolically simulate conservative approximation 1. During the process of symbolic simulation we record the signal waveforms on the outputs of the decoder and the column address latches. We construct a trajectory formula W, which captures the signal values on the outputs recorded above. As discussed earlier, it can be shown that $[A \Longrightarrow W]$ is true.

5. **Verification of SRAM core** — Finally, with conservative approximation 2, we show that given the waveform W, and the antecedent A, the consequent C is true, i.e., $[A \wedge W \Longrightarrow C]$. From the earlier discussion in section 3.4, if $[A \Longrightarrow W]$ and $[A \wedge W \Longrightarrow C]$ are both true, then we can conclude that $[A \Longrightarrow C]$ is true, i.e., the memory operation is verified for location 0. Given the symmetries of the circuit we can then conclude that the operation works

SRAM size (bits)	No. of Transistors	Model Size (Bool. ops)		Anamos Time (CPU Secs.)		Anamos Memory (MB)	
		Full	Reduced	Full	Reduced	Full	Reduced
1K	6690	79951	2781	120	4.1	9.6	0.9
4K	25676	307555	5462	863	14.1	36.8	2.1
16K	100566	1205239	10895	7066	43.2	144.2	6.0
64K	397642	—	21960	—	170.7	—	22.0
256K	1581494	—	44545	—	732.7	—	80.0

Table 3.1: Generation of SRAM model: Full vs. Reduced model.

correctly for every memory location.

3.6 Experiments and Results

All the time and memory figures in this section have been measured on a Sun SparcStation-20. We used the Anamos switch-level analyzer to generate switch-level models [19]. We modified Anamos to make it possible to attach *X-drivers* to circuit nodes to generate reduced models (conservative approximations) of switch-level circuits. Table 3.1 shows the results of model generation for SRAM circuits of varying sizes. The full circuit model contains about 73 operations for each memory bit. However, for circuits larger than 16K, it was not possible to generate the full circuit model within reasonable time or memory bounds (empty table entries). Conservative approximations of SRAM circuits, on the other hand, can be generated for much larger circuits for a miniscule fraction of the cost of the full model. The reduced model size grows proportional to the square root of the SRAM size, and its generation time and memory is proportional to the SRAM size.

To verify a structural symmetries, we do graph isomorphism checks as outlined in section 3.3. We have modified the isomorphism checking code [7] from Anamos for our purpose. Table 3.2 reports the running time and memory taken for converting one instance of the memory core or column mux permutation into a canonical circuit,

SRAM Size (bits)	Memory Core			Column Multiplexer			Total Isomorph. Check Time (Secs.)
	CPU Time (Secs.)	Memory (MB)	No. of checks	CPU Time (Secs.)	Memory (MB)	No. of checks	
1K	2.6	1.6	4	0.3	0.19	5	11.9
4K	11.1	6.5	4	0.5	0.38	6	47.4
16K	51.2	26.0	4	1.3	0.74	7	214.1
64K	232.1	104.0	4	3.0	1.44	8	952.4
256K	1135.6	416.0	4	6.6	3.50	9	4601.8

Table 3.2: Symmetry checks for memory core and column multiplexer.

and the total time to do all the isomorphism checks. The total time and memory requirements scale linearly with the SRAM size. Table 3.2 reports the resources required to check the structural symmetries. The memory core has four symmetry generators. We verify that each of these is a symmetry (columns 2,3,4). The column multiplexer has a number of symmetry generators equal to the number of column address lines. We also verify each of these generators (columns 5,6,7). The total time is reported in column 8. Table 3.3 shows the time and memory required to check the decoder and column address latch symmetries by symbolic simulation.

We used the Voss verification system [116] to verify the reduced SRAM circuit. Table 3.4 shows the running time and the memory required for verifying the write operation for location 0. In addition, we verify two other properties — that the read operation reads the value stored at location 0 (Table 3.5), and that operations at other addresses do not change the data in location 0 (Table 3.6). The time and memory required to verify these other operations is similar to that of the write. Much of the verification time and memory is taken to read in the reduced circuit model and representing it. So, it is not surprising that the time and memory requirements grow roughly proportional to the square root of the memory size.

The total verification time for a SRAM circuit is the sum of the times in tables 3.1, 3.2, 3.3 and 3.4. For example, to verify a 64K SRAM, 170.7 secs. are required to generate the reduced circuit model, a total of 952.4 + 3.2 secs. are required to verify the circuit symmetries, and an additional 6.0 + 6.6 + 6.1 secs. are required to verify

SRAM Size (bits)	Time (CPU Secs.)	Memory (MB)
1K	1.7	0.69
4K	2.1	0.74
16K	2.5	0.88
64K	3.2	1.10
256K	4.2	1.52

Table 3.3: Decoder and col. latch symmetry checks.

SRAM Size (bits)	Verif. Time (CPU Secs.)	Verif. Memory (MB)
1K	1.5	0.79
4K	2.0	1.05
16K	3.0	1.80
64K	6.0	2.84
256K	18.5	4.26

Table 3.4: Verification of reduced SRAM writes.

the reduced model for all the operations. This gives a total verification time of 1145.0 secs. It is interesting to note that symmetry checks dominate much of this time. In the verification process, the only time we ever work with the complete circuit is the symmetry check phase. This partially explains the reason for the relatively large time and memory requirements of this phase. However, the circuit isomorphism code we have used is a simple modification of that in Anamos. There is considerable scope for reducing time and memory by developing a specialized circuit isomorphism checker.

SRAM Size (bits)	Verif. Time (CPU Secs.)	Verif. Memory (MB)
1K	1.7	0.79
4K	2.2	1.05
16K	3.0	1.80
64K	6.2	2.84
256K	18.7	4.26

Table 3.5: Verification of reduced SRAM reads.

SRAM Size (bits)	Verif. Time (CPU Secs.)	Verif. Memory (MB)
1K	1.8	0.87
4K	2.2	1.12
16K	3.1	1.82
64K	6.6	2.90
256K	19.6	4.35

Table 3.6: Verification that unaddressed location unchanged.

3.7 Related Work

Petri nets have long been used to describe systems consisting of communicating concurrent processes. Some early work on exploiting symmetry in petri nets is by Huber et al. [67] in 1984. In [107], Starke proposes an algorithm to compute the generators of the symmetry group for petri nets. Jensen [73] describes the application of symmetry to construct condensed versions of state spaces of concurrent systems described by colored petri nets. The work shows that for reachability properties, the unreduced state space satisfies a property, if and only if the condensed state space satisfies the property. Typically the work in this area focusses on reachability analysis, rather than more general temporal properties, and does not consider the added complexity (and the concomitant payoff) of symbolic state space representations.

Work on exploiting symmetry for automated formal verification techniques is quite recent. Emerson and Sistla [50, 49] show how to exploit symmetry in model checking with the CTL* temporal logic. In a system $\mathcal{M}$ consisting of many isomorphic processes, the symmetry in the system is captured in the group of permutations of process indices defining graph automorphisms of $\mathcal{M}$. Similarly, symmetry in a specification formula f is captured by the group of permutations of process indices that leave f invariant. Given a permutation group G, which is contained in both groups, one can construct a quotient structure $\overline{\mathcal{M}}$, such that for a start state s, and the equivalent state $\overline{s}$ in $\overline{\mathcal{M}}$, $\mathcal{M}, s \models f$ iff $\overline{\mathcal{M}}, \overline{s} \models f$ holds. This is the *correspondence theorem*, which is the central result of their work. This work, however, does not address the complexities that arise from symbolic representations of the state space.

The work by Clarke et al. [38, 39] takes a slightly more general approach than that by Emerson. It views symmetry as a transition relation preserving permutation, not just permutation of indices of identical processes. Given a symmetry group G acting on a Kripke structure M, one can construct a reduced structure M_G. The *correspondence theorem* in this work shows that if there is a CTL* formula f, such that all the atomic propositions in f are invariant under G, the f is true in M if and only if it is true in M_G. This work discusses the construction of the reduced transition relation which is represented symbolically.

The distinction between our work and that by Clarke et al., or Emerson et al. comes partly from heritage. The roots of our work lie in the verification of data intensive systems using symbolic simulation of switch-level circuits. We represent the system transition by means of an excitation function. Naturally, when we consider symmetry in a system, we examine the symmetry in the excitation function. We explore structural and data symmetries in the system, which give rise to symmetries in the excitation function. We actually verify that these symmetry properties exist in the system, and then we construct a reduced model of the circuit. This approach allows us to verify huge systems, including one with 262144 state holding elements (or 2^{262144} states). Emerson or Clarke view symmetry as a permutation on a state graph with some desired properties, like preserving the transition relation, or a permutation of process indices which a transition graph automorphism. While this approach appears more general, it imposes a significant limitation – capturing phenomena like the symmetrical behavior on the outputs of a decoder is extremely tedious, if not impossible, with their process oriented point of view. Also, in both the approaches outlined above, to be effectively exploited, symmetry must be present in both the state transition graph, and the temporal logic formula being verified. In contrast, our approach does not impose the restriction of symmetry on the temporal logic formula being verified.

In [69], Ip and Dill discuss the verification of large concurrent systems, where the symmetry in the system is identified by a special scalar-set datatype in the system description language. They view symmetry as an automorphism on the state transition graph, they describe an on-the-fly construction of the reduced state transition graph. They show that simple safety properties which hold in the reduced state graph also hold in the original system. One approach contrasts with this, as we do not constrain the user to explicitly give symmetry in the system description because we can directly work with transistor netlists. Also, we can construct conservative models directly by switch-level analysis of the transistor-level system description, which is more efficient than on-the-fly construction of the reduced state transition relation.

Aggarwal, Kurshan and Sabnani have exploited symmetry for the verification of the alternating bit protocol, a standard benchmark for protocol verification techniques [1]. Applying reduction techniques specific to this problem, and some simplifying assumptions, they reduce the large state space using machine homomorphisms. They show that for verifying properties about the given state machine, if suffices to verify the desired properties on the reduced machine. This approach to a some degree approximates our notion of data symmetry by considering the data symmetry between 0 and 1 in the transmitted messages. However, the approach does not consider many important issues like detecting, and verifying symmetry, and the automated construction of symmetry reduced state space.

Most approaches to exploiting symmetry in verification start by reducing the state space using equivalent relations among different states. Partial order methods [105, 106, 122] is an important class of work which diverges from this viewpoint by considering equivalent relations among different paths. Each path consists of an interleaving sequence of actions. If a set of actions occurring in two paths are independent, and therefore their order can be permuted, then the paths are considered equivalent with respect to this set of actions. So, for verification, one needs to consider only one of these paths.

Gupta and Fisher describe linearly inductive functions (LIFs) to capture structural induction in parametrized circuit descriptions [59, 58]. They present a canonical representation for LIFs which provides a fixed size representation for all size instance of the circuit. Such an approach enables efficient verification by capturing similarity in the structure of systems.

Chapter 4

Verification of Content Addressable Memories

Content Addressable Memories (CAMs) are an important subclass of memory arrays. CAMs are widely used in applications which require fast parallel search operations. A common example is the translation-lookaside buffer (TLB) in the memory management unit of a processor which translates virtual addresses to physical. Other examples of CAMs on modern processors include branch prediction buffers, branch target buffers and cache tags. Outside the realm of processors, CAMs have been used in various applications such as data compression [85], [83], data-base accelerators [126], image processing [96], global routing [115], and Lisp machines [10].

In this chapter, we describe how we successfully leveraged STE, along with new Boolean encoding techniques, to verify CAMs. The encodings were needed to contain the exponential growth in the verification space requirements with increasing CAM sizes, as seen with a naive use of variables with STE.

This chapter begins with a short description on the structure of CAMs (section 4.1). The basic machinery to generate and manipulate symbolic ternary vectors for efficient verification of CAMs follows in section 4.2. Section 4.3 first shows the search for an efficient Boolean encoding for CAM verification, and it then shows the application of these encodings to various types of CAMs. The advantages of the encoding techniques we have developed are illustrated quantitatively in section 4.4. Chapter 6 discusses the verification of two complex CAM arrays from PowerPC microprocessors.

4.1 The structure of CAMs

Generally, CAMs employ as an identifier a bit field called a *tag*. The tag serves as a key to identify a particular data entry stored in the memory array. CAMs vary depending upon data and tag size, techniques to read and write contents and mark contents as valid, tag masking fields, etc. In spite of all this diversity, CAMs all have in common the *associative read capability*. The associative read operation consists of searching, in parallel, all tags in the CAM to determine if there is a match to a particular tag of interest, and then sending the associated data entry to an appropriate read port of the memory. In some instances CAMs also have an *associative write capability*, discussed in detail in Section 6.3.

The high-level design shown in Figure 4.1 is a very basic CAM. We implemented this design to serve as a vehicle for experiments with STE on CAMs, and, in particular, as an experimental vehicle for finding better Boolean encodings to aid in using STE on CAMs.

We have implemented this design as a transistor-level netlist. Each t-bit tag consists of t tag cells. Each tag cell contains 9 transistors and its design, which is based on the one in [125, pp.590], is shown in Figure 4.2. In the figure, a 6-transistor SRAM cell, consisting of transistors T1 through T6, resides at the core of the tag cell. Thus, one can perform reads and writes in a tag portion of a CAM as in a regular SRAM array. Transistors T7 through T9 form a comparator structure which compare the data stored in the tag cell to the data (and its complement) appearing at bl, and blbar. Before the compare operation begins, Match Line is precharged to a high. A mismatch in the value stored in the cell (at nodes d and db), and the incoming value at nodes cmp, and cmpbar causes node n to become high, and thus results in the discharge of Match Line. Typically, t tag cells share a single Match Line, to do comparisons over bit-vectors. The data portion of the CAM is organized much like a conventional SRAM, with the exception that the word lines are driven by the match lines from the corresponding tag entries.

This design has n tag entries, $T[0], T[1], ..., T[n-1]$. Corresponding to each such tag entry, $T[i]$, there is a data entry $D[i]$. By specifying the proper combination of

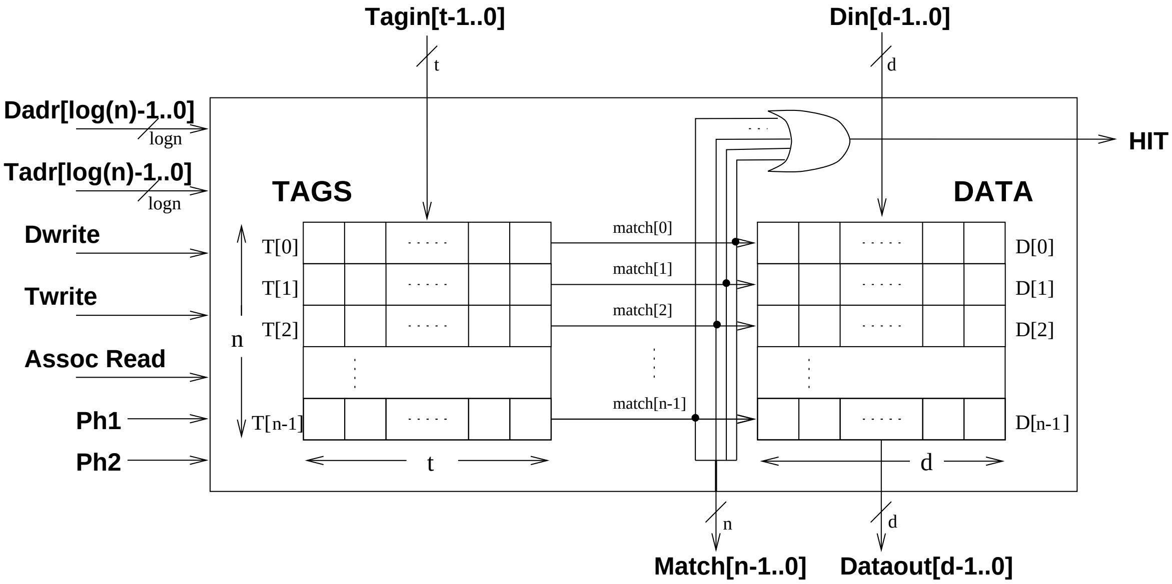

Figure 4.1: Content Addressable Memory: Tag size = t, Number of entries = n, Data size = d

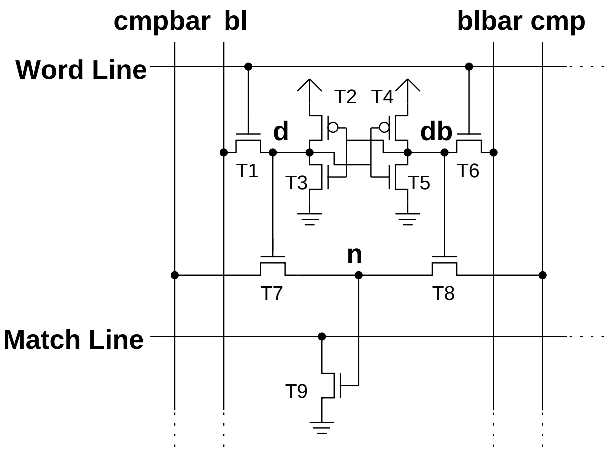

Figure 4.2: Tag cell in a CAM

address and data at Tadr, and Tagin, one can write to a desired tag entry as in a conventional random access memory. Similarly, by specifying address and data at Dadr, and Din, one may update the data entries. Also, while the extra logic has not been shown in the figure, many CAMs have the capability to perform tag and data reads based on the addresses at Dadr and Tadr. However, the most distinctive operation of this circuit is the *associative read* operation. In this operation $Tagin$ is compared in parallel with all the $T[]$ tag entries, and if there is a match on the ith tag entry, then HIT rises, and $D[i]$ appears at $Dataout$. If there is no match to $Tagin$, i.e., a *miss*, HIT remains low (and, the surrounding circuitry would ignore $Dataout$).

In the vast majority of CAMs, it is an assumption that, among valid tag entries there is *at most one* tag that matches $Tagin$. This property, the *at most one tag match property*, is an important system invariant. However, this property is usually not enforced in hardware, i.e., no special circuitry is implemented to detect or guarantee this condition. Rather, CAMs generally depend upon surrounding circuitry, or the software manipulating the entire chip, to maintain this invariant. For example, in the branch target address cache (BTAC) array (described in Chapter 6), the branch prediction logic controlling the BTAC writes ensures the at most one match invariant holds. In a CAM like the block address translator (BAT) array [112], the responsibility of maintaining the invariant is with the operating system software, which can write new tag and data values directly into the CAM.

4.1.1 Variations in CAM designs

Two important variations in the CAM design presented above are the ability to handle multiple matches, and the ability to perform content addressable writes. In the case where multiple matches are allowed, two distinct schemes have been presented in literature [54, 85]. The first scheme selects one of the multiple hits based on a preassigned priority of the different matches. The second scheme counts the number of matches that occur.

In the case of multiple matches with priority, a priority encoder is introduced between the tag and data portions of a CAM (Figure 4.3). This encoder takes as its input the match signals from the tags, and the match signal of the highest priority is sent to the data part of the CAM. All other match signals to the data part are suppressed. Usually the priority between the different match signals is assigned

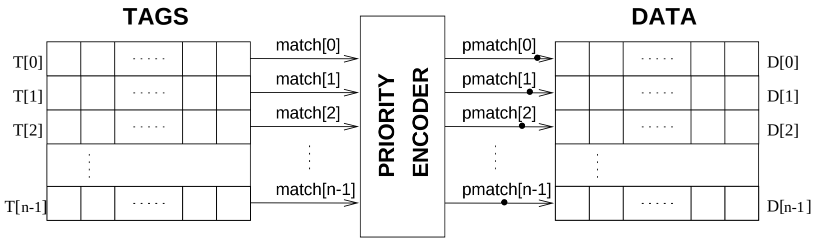

Figure 4.3: Multiple matches with priortization.

statically. In the second multi-match scheme, the number of matches is also a part of the output. Applications of this include minimum distance string matching [126]. Caxton [54, pp.79] shows an organization of the counting circuitry for minimizing propagation delay. In the multi-match counting scheme, the data portion is usually absent from the CAM.

Content addressable writes are done on the basis of the values stored in the tags. The most common scheme is to select the tag-data entry to be replaced by performing a content addressable read with a given tag input. If there is a tag entry which matches the input tag, then that particular tag and its corresponding data entry are selected for replacement. In the subsequent phase or cycle they are updated to the desired value. One refinement to this includes having a valid bit with each tag-data entry pair, so that in case of a write, the first invalid entry is written to. Another refinement includes having a least recently used type replacement policy to select the replacement entry in case the content addressable read does not yield a hit. The BTAC example in Chapter 6 illustrates some of these features in greater detail.

4.2 Symbolic vector generation and manipulation

Verification of CAMs requires the extensive use of complex ternary vectors to specify the various CAM operations, and efficiently encode the system invariant conditions. Typically, a large number of non-symbolic or *scalar* ternary vectors are represented by a single symbolic vector. In an earlier paper [103] we demonstrated a number of Boolean encoding schemes to efficiently verify CAMs. The notation we presented included many auxiliary variables which were introduced for encoding conditions such as vector mismatch position. These auxiliary variables did not contribute much to the understanding of the high-level system behavior, and they should not have been visible in the abstract specifications. To overcome this problem, and to allow simple intuitive abstract specifications, we have developed a simple consistent notation to specify the generation and manipulation of symbolic ternary vectors. This is described ahead. Following this, in section 4.3, we discuss the application of symbolic ternary vectors to efficient CAM verification.

The basic object we work with is a *symbolic ternary vector*, which is an array of symbolic ternary values. When each element of a symbolic ternary vector is a binary quantity, we refer to the vector as a *symbolic binary vector*. When a symbolic ternary (binary) vector contains no symbolic variables, then we also refer to it as a ternary (binary) vector, or a scalar ternary (binary) vector. A symbolic ternary value is represented by a pair of OBDDs using the dual-rail encoding of chapter 2. Each symbolic ternary vector implicitly represents a set of scalar ternary vectors. We formally define this object below.

Definition 12 *A symbolic ternary vector A is an object with the following four attributes:*

1. *$length(A)$ — The length of vector A.*
2. *$support(A)$ — The list of symbolic Boolean variables which appear in the support set of the functions in A.*
3. *$elements(A)$ — The subset of vectors in $\{0, 1, X\}^{length(A)}$ which are represented by A.*
4. *$compatibles(A)$ — The subset of vectors in $\{0, 1\}^{length(A)}$ such for any element $u \in compatibles(A)$, there exists an element $v \in elements(A)$ such that $v \sqsubseteq_{\mathcal{T}} u$.*

The attribute $elements(A)$ is the set of all non-symbolic vectors that A evaluates to for all possible assignments to the symbolic Boolean variables in $support(A)$. Based on the information content of the $0/1/X$ ternary values, 0, and 1 conflict with each other, and all the other pairs of ternary values are non-conflicting or *compatible*.

Given an integer k, $0 \leq k < length(A)$, $A[k]$ is the k^{th} component of vector A. Given integers j, and k, $0 \leq j, k < length(A)$, and $j \leq k$, $A[j..k]$ is the vector of length

$k-j+1$, consisting of components $A[j]$ through $A[k]$. A binary vector is compatible with a ternary vector if at every bit position the two vectors have compatible values. *compatibles*(A) refers to the set of binary vectors each of which has at least one compatible ternary vector in *elements*(A).

In the examples ahead, we use the notation (f_L, f_H) to represent a symbolic ternary value, where f_L, and f_H are Boolean expressions representing the low rail and the high rail values respectively. Of course, the set of symbolic binary vectors is included in the set of symbolic ternary vectors, and in such a vector the low-rail and the high-rail values are complementary. When it is clear from context that (f_L, f_H) is a symbolic binary value, i.e., f_L and f_H are complementary, then sometimes instead of (f_L, f_H) we write f_H, the high-rail value. (0,1), (1,0), and (1,1) represent the ternary values 1,0, and X respectively. If all the elements of a vector are scalar quantities, we write the vector as a sequence of ternary values, e.g., $100X1X0$. If the vector contains only symbolic binary values, then it may be written as comma separated sequence of symbolic values within angle brackets, e.g., $\langle a_0, a_1 \cdot a_0, a_1\rangle$. If the vector contains symbolic ternary values, then it is written as a list of comma separated symbolic ternary values enclosed within angle brackets. As an example, let $A = \langle (a+b, \overline{a}+b), (c, \overline{c})\rangle$. In this case, $length(A) = 2$, $support(A) = \{a, b, c\}$, $elements(A) = \{00, 01, 10, 11, X0, X1\}$, and $compatibles(A) = \{00, 01, 10, 11\}$.

For variable set v, let Φ_v denote the set of all 0/1 assignments to the variables in v. For $v \subseteq support(A)$, let $\phi \in \Phi_v$. We define $restrict(A, \phi)$ to be the symbolic ternary vector B which is obtained by setting the variables in the functions in A to their assignments specified in ϕ. The resultant vector B has $support(B) = support(A) - v$. When $v = support(A)$, then $restrict(A, \phi)$ denotes a scalar ternary vector $\vec{a}$.

The starting point for the creation of symbolic ternary vectors is a set of *generators*. The generators each produce a symbolic binary vector A of a specified length n, such that $element(A)$ is a a subset of $\{0,1\}^n$. To synthesize more complex symbolic ternary vectors, we have a set of *operators*. These operators act upon the symbolic vectors obtained from the generators, to yield new symbolic vectors.

4.2.1 Symbolic vector generators

Below, we describe the *bitvector*, *onehot*, *index*, and the *unary* generators. All generators other than *bitvector* also have their *anonymous* counterparts. These are described at the end of this section.

Bitvector generator

The most basic generator is the *bitvector* generator. Given an integer argument n, it returns a vector of n symbolic Boolean variables. If $A = bitvector(n)$, then $length(A) = n$, $support(n)$ contains n Boolean variables, and $elements(A) = compatibles(A) = \{0, 1\}^n$.

Onehot generator

The *onehot* generator takes two arguments, an integer n, and a vector of Boolean variables v. $onehot(n, v)$ returns a vector which symbolically combines all the one-hot vectors of length n, with the variables in v. Vector v must have a length of at least $\lceil log_2 n \rceil$. If $A = onehot(n, v)$, $support(A) = \{v[0], v[1], \ldots, v[length(v) - 1]\}$ $length(A) = n$, $support(A) = v$, and $elements(A) = \{0^{i-1}10^{n-i} \mid 1 \leq i \leq n\}$. If $\phi = (v = k)$, is an assignment, where the variables in vector v are assigned 0/1 values corresponding to the integer k, $v[0]$ being the LSB, then $Restrict(A, \phi) = 0^{k-1}10^{n-k}$.

Index generator

Often, given a set of symbolic ternary vectors $S = \{s_0, s_1, ..., s_{m-1}\}$, $S \subseteq \{0, 1\}^n$, it is necessary to index one element of S symbolically with a vector of Boolean variables v. The *index* generator takes S, and a variable vector v of length $\lceil log_2 m \rceil$ as arguments, to generate the desired symbolic ternary vector. $A = index(S, v)$ is such that, $length(A) = n$, and $elements(A) = S$. If $\phi = (v = i)$, is an assignment, where the variables in vector v are assigned Boolean values corresponding to the integer i, $v[0]$ being the LSB, then $Restrict(A, \phi) = s_i$.

Unary generator

The *unary* generator, given an integer argument n, and a vector of symbolic variables v, produces a symbolic vector of length n which symbolically combines all vectors of the form $0^i 1^{n-i}$, $0 \leq i < n$. If $A = unary(n, v)$, where v is of length $\lceil log_2 n \rceil$, then $length(A) = n$, and $elements(A) = \{0^i 1^{n-i} \mid 0 \leq i < n\}$.

Anonymous generators

The second argument of the *onehot*, *index*, and *unary* generators is a vector of Boolean variables which helps combine a number of scalar vectors into one symbolic ternary vector. Often, it is necessary to know the identity of these "combining" variables, but there are instances, where their identity is not important. In such cases, it should not be necessary to specify the "combining" variables when invoking the generators. Instead, the generators should implicitly create the necessary Boolean variables and perform their task. Each of the last three operators have their *anonymous* counterparts which are written exactly as before, with the exception that they do not have a second argument. In the examples ahead, we illustrate the use of both *anonymous*, and *explicit* generators.

4.2.2 Symbolic vector operators

An operator is a function which maps an element of the set of binary vectors to a subset of the set of ternary vectors. More formally,

Definition 13 *A symbolic vector operator is a mapping*

$$op : \{0,1\}^n \;\rightarrow\; 2^{\{0,1,X\}^m}$$

which maps a binary vector of length n *to a set of ternary vectors of length* m*. If* A *is a symbolic vector produced by a generator, such that* $length(A) = n$*, and* $support(A) = v$*, then* $op(A)$ *yields a symbolic vector* B*, with* $length(B) = m$*,* $support(B) = v' \supseteq v$*, and the following property:*

$\forall \phi \in \Phi_v.\ [\vec{a} = Restrict(A, \phi),\ and\ B' = Restrict(B, \phi)] \Rightarrow op(\vec{a}) = elements(B')$.

Intuitively, the definition above states that if op can map scalar binary vectors to a set of ternary vectors, then it can consistently map a symbolic vector A into a symbolic vector B. Below, we discuss the *ternneq*, *binneq*, and the *ternneqmask* operators.

ternneq operator

An important aspect in the operation of CAMs is comparison over bit-vectors to determine whether or not they are equal. We introduce the *ternneq* operator that maps a symbolic binary vector A of length n to $ternneq(A)$ which represents all the symbolic vectors that are "unequal" to A. For example, if $A = 0010$, then $elements(ternneq(A)) = \{1XXX, X1XX, XX0X, XXX1\}$. The scalar ternary vectors in this set represent all the different ways in which a 4-bit vector can be unequal to the bitvector 0010. If $A = \langle f_0, f_1, f_2, f_3 \rangle$, where f_0 through f_3 are symbolic binary values, then

$$\begin{aligned} elements(ternneq(A)) = {} & elements(\langle \overline{f_0}, X, X, X \rangle) \cup elements(\langle X, \overline{f_1}, X, X \rangle) \\ & \cup elements(\langle X, X, \overline{f_2}, X \rangle) \cup elements(\langle X, X, X, \overline{f_3} \rangle) \end{aligned}$$

We discuss the ternneq operator and its explicit version further in section 4.2.3.

binneq operator

The *binneq* operator is identical to the *ternneq* operator, with the difference that for a symbolic binary vector $\vec{a}$, $elements(binneq(\vec{a}))$ is a subset of $\{0,1\}^{length(\vec{a})}$. Thus *binneq* produces symbolic vectors only over the binary domain.

ternneqmask operator

Some CAMs allow selective masking of certain bit positions so that comparison over these positions is ignored. This makes it necessary to have the *ternneqmask* operator which takes into account mask values while computing a symbolic vector unequal to a given symbolic binary vector.

Let A be a symbolic binary vector of length n. Let M be a binary vector of length n which masks comparisons of vector A. To keep our discussion simple, initially we assume that M is scalar. Mask M disables comparison over all the bit positions for which it is a 1. For instance, if $M = 0^{n-1}1$, then comparison occurs over every bit position other than the last. Similarly, if $M = 0^{n/2}1^{n/2}$, then the comparison is enabled only over the first $n/2$ bits. This idea can be extended in a straightforward manner to the case where M is symbolic. Let $v_M = support(M)$, and Φ_{v_M} be the set of all assignments to variables in v_M. For all $\phi \in \Phi_{v_M}$, $Restrict(M, \phi)$ disables comparisons over A for those bit positions at which it is 1.

With this background, we make a minor extension of definition 13 to define *ternneqmask* as an operator which maps two binary vectors to a set of ternary vectors:

$$ternnegmask : \{0,1\}^n \ \times \ \{0,1\}^n \ \rightarrow \ 2^{\{0,1,X\}^n}$$

Let A, and M be symbolic binary vectors such that $length(A) = length(M) = n$, $support(A) = v_A$, and $support(M) = v_M$. Let Φ_{v_A}, and Φ_{v_M} denote the set of all assignments of variables in v_A, and v_M respectively. $B = ternnegmask(A, M)$ is a symbolic ternary vector satisfying the following two conditions:

- $\forall \phi_a \in \Phi_A . \forall \phi_m \in \Phi_M . (\vec{a} = Restrict(A, \phi_a))$ and $(\vec{m} = Restrict(M, \phi_m))$ and $(B' = Restrict(B, \phi_a \cup \phi_m)) \Longrightarrow B' = ternneqmask(\vec{a}, \vec{m})$.

  Intuitively, this condition states that the symbolic extension of *ternneqmask* is consistent.

- If $\vec{a} = Restrict(A, \phi_a)$, $\phi_a \in \Phi_A$, and $\vec{m} = Restrict(M, \phi_m)$, $\phi_m \in \Phi_M$, then $\forall \vec{u} \in ternneqmask(\vec{a}, \vec{m}) . \forall \vec{v} \in compatibles(\vec{u}) . \vec{v} \& \overline{\vec{m}} \neq \vec{a} \& \overline{\vec{m}}$, where & is the bitwise AND operator, and $\overline{\vec{m}}$ is the bitwise negation of every element of $\vec{m}$.

  This condition enforces the inequality of $\vec{a}$ over the non-masked positions indicated by $\vec{m}$.

If $A = 1010$, and $M = 0011$, then $elements(ternneqmask(A, M)) = \{0XXX, X1XX\}$. Note that the elements in this set disagree with A over the first two bit positions.

4.2.3 Implementation of symbolic vector generators and operators

The generators and the operators can all be implemented with the common Boolean operations available in any standard OBDD interface [14]. Every call to a *bitvector* generator results in the creation of a set of new Boolean variables. Other generators like *onehot(n,v)*, and *unary(n,v)* produce their results in a straightforward manner. Each of these generators produces a symbolic vector which represents a set of symbolic or scalar vectors $\{s_0, s_1, ..., s_n\}$. The task of the variables in v is to "select" one of these vectors, and the desired symbolic expression equals $(v = 0) \cdot s_0 + (v = 1) \cdot s_1 + \ldots (v = n - 1) \cdot s_{n-1}$. The $+$ operator results in the bitwise OR of its argument vectors. The $\cdot$

operator results in the conjunction of each element of a vector with a Boolean value. For example, if $n = 4$, and $v = v_1 v_0$, then variables v_0, and v_1 select one of the four different on-hot bitvectors, 0001, 0010, 0100, and 1000.

$$\begin{aligned} & onehot(n, v) \\ = & (v_1 v_0 = 00) \cdot 0001 + (v_1 v_0 = 01) \cdot 0010 + (v_1 v_0 = 10) \cdot 0100 + (v_1 v_0 = 11) \cdot 1000 \\ = & \langle v_1 v_0, \overline{v_1} v_0, v_1 \overline{v_0}, \overline{v_1 v_0} \rangle \end{aligned}$$

We discuss below the implementation of the *ternneq* operator. Consider the earlier example, where

$$elements(ternneq(0010)) = \{1XXX, X1XX, XX0X, XXX1\}$$

Intuitively, for each bit position of the binary vector we create a new ternary vector which has the opposite binary value at that position, and has a X at every other bit position. In the next step towards creating a symbolic ternary vector for $ternneq(0010)$, we must roll the four vectors in {1XXX, X1XX, XX0X, XXX1} into one symbolic vector. This can be done with a pair of Boolean variables which select one of the four ternary vectors. An alternate viewpoint is that these variables select the bit position of the inequality, and we refer to them as *position inequality variables*. In general, for vectors of length n, $log_2 n$ position inequality variables are required. These variables can be anonymous under almost all circumstances for associative reads. During associative writes it may become necessary to expose these values in some cases.

Let $A = A_0 A_1 ... A_{n-1}$ be symbolic binary vector of length n. Let $A_i.H$, and $A_i.L$ be the high and low rail values, respectively, of the ith element of A. Since A_i is binary, $A_i.H$, and $A_i.L$ are complementary. Let $R = R_0 R_1 ... R_{n-1}$ be the symbolic ternary vector equal to $ternneq(A)$. If $R_i.H$, and $R_i.L$ are the high and low rail values, respectively, of the ith element of R, and v is a vector of Boolean variables of length $log_2 n$, then the dual rail values of R_i may be constructed as follows:

$$R_i.H = (v \neq i) + A_i.H$$
$$R_i.L = (v \neq i) + A_i.L$$

ternneqmask can be similarly implemented by taking into account each of the different possible mask values, and combining the ternary vectors so generated.

4.3 CAM properties and CAM encodings

Below, we discuss our search for efficient Boolean encodings to use in STE verifications of CAMs (Section 4.3.1). We show how a well chosen encoding can dramatically reduce the number of variables, and therefore the number of OBDD nodes, required for the verification. We discuss the results of our experiments in Section 4.4.

4.3.1 CAM Encodings

We will discuss the CAM encoding problem in the context of verifying the associative read operation of CAMs. We will refer to a generic CAM modeled after that of Figure 4.1, in Section 4.1.

The most obvious approach to verifying the associative read operation is to introduce a Boolean variable for each bit of state in the $T[i]$ and $D[i]$ tag and data entries. We illustrate this below with an example trajectory assertion. We assume the number of CAM entries, n, equals 3. Let $\vec{tin}, \vec{t_0}, \vec{t_1}$ and $\vec{t_2}$ be vectors of Boolean variables of size t, the width of the $T[i]$ entries. Let $\vec{d_0}, \vec{d_1}$ and $\vec{d_2}$ be vectors of Boolean variables of size d, the width of the $D[i]$ entries. These vectors are created by the *bitvector* generator. The following assertion specifies the associative read operation under these conditions[1].

$$\begin{aligned}
\vec{t_0} &= bitvector(t)\\
\vec{t_1} &= bitvector(t)\\
\vec{t_2} &= bitvector(t)\\
\vec{d_0} &= bitvector(d)\\
\vec{d_1} &= bitvector(d)\\
\vec{d_2} &= bitvector(d)
\end{aligned}$$

$$\begin{aligned}
(op = assocread) \wedge (Tagin = \vec{tin}) \wedge (T[0] = \vec{t_0}) \wedge (T[1] = \vec{t_1}) \wedge (T[2] = \vec{t_2}) \wedge\\
(D[0] = \vec{d_0}) \wedge (D[1] = \vec{d_1}) \wedge (D[2] = \vec{d_2})
\end{aligned}$$

[1]Some parts of the assertion necessary for verification thoroughness, e.g., that the tag and data bits are unchanged on a read, have been omitted.

$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$

$$\begin{aligned}
&(when(\neg(\vec{tin} = \vec{t_0}) \wedge \neg(\vec{tin} = \vec{t_1}) \wedge \neg(\vec{tin} = \vec{t_2}))((HIT = 0))) \wedge \\
&(when((\vec{tin} = \vec{t_0}) \wedge \neg(\vec{tin} = \vec{t_1}) \wedge \neg(\vec{tin} = \vec{t_2}))((HIT = 1) \wedge (Dataout = \vec{d_0})) \wedge \\
&(when(\neg(\vec{tin} = \vec{t_0}) \wedge (\vec{tin} = \vec{t_1}) \wedge \neg(\vec{tin} = \vec{t_2}))((HIT = 1) \wedge (Dataout = \vec{d_1}))) \wedge \\
&(when(\neg(\vec{tin} = \vec{t_0}) \wedge \neg(\vec{tin} = \vec{t_1}) \wedge (\vec{tin} = \vec{t_2}))((HIT = 1) \wedge (Dataout = \vec{d_2})))
\end{aligned}$$

The first line of the antecedent specifies that an associative read is being done, that the data at the input is $\vec{tin}$, and that the three tag registers initially contain $\vec{t_0}$, $\vec{t_1}$, and $\vec{t_2}$. The three data registers are specified as initially containing $\vec{d_0}$, $\vec{d_1}$, and $\vec{d_2}$. The first line in the consequent checks for the condition when there are no matching entries in the CAM. The second consequent line checks for HIT and $Dataout$ when only the first entry matches. Note that we do not check for conditions inconsistent with the *at most one match* system invariant. For example, we do not check for what happens when $(\vec{tin} = \vec{t_0})$ and $(\vec{tin} = \vec{t_1})$ are both true. A total of $(t + d)n + t$ Boolean variables are needed for this assertion. We call this encoding, where every bit of the circuit state has a corresponding Boolean variable, as the *full encoding* technique.

We can reduce the variable count, however, by using symbolic indexing, at this point just for the data entries. We can do this by using a vector of Boolean variables, i, $\lceil log_2 n \rceil$ bits wide, to index into the data entries, thereby saving extra variables to encode the values on the data entries. To effect this, the antecedent above should contain $(D[i] = \vec{data})$ instead of $(D[0] = \vec{d_0}) \wedge (D[1] = \vec{d_1}) \wedge (D[2] = \vec{d_2})$, where $\vec{data}$ is a vector of Boolean variables d bits wide. The assertion below reflects this change:

$$\begin{aligned}
\vec{tin} &= bitvector(t) \\
\vec{t_0} &= bitvector(t) \\
\vec{t_1} &= bitvector(t) \\
\vec{t_2} &= bitvector(t) \\
\vec{data} &= bitvector(d) \\
i &= bitvector(log_2 n)
\end{aligned}$$

$$(op = assocread) \wedge (Tagin = \vec{tin}) \wedge (T[0] = \vec{t_0}) \wedge (T[1] = \vec{t_1}) \wedge (T[2] = \vec{t_2})$$

$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$

$$(when((\vec{tin} = \vec{t_0}) \wedge \neg(\vec{tin} = \vec{t_1}) \wedge \neg(\vec{tin} = \vec{t_2}) \wedge (\vec{i} = 0))((HIT = 1) \wedge (Dataout = \vec{data}))$$
$$(when(\neg(\vec{tin} = \vec{t_0}) \wedge (\vec{tin} = \vec{t_1}) \wedge \neg(\vec{tin} = \vec{t_2}) \wedge (\vec{i} = 1))((HIT = 1) \wedge (Dataout = \vec{data}))$$
$$(when(\neg(\vec{tin} = \vec{t_0}) \wedge \neg(\vec{tin} = \vec{t_1}) \wedge (\vec{tin} = \vec{t_2}) \wedge (\vec{i} = 2))((HIT = 1) \wedge (Dataout = \vec{data}))$$

Now, we need only $(n+1)\cdot t+d+\lceil log_2 n\rceil$ Boolean variables. We call this encoding, the *plain encoding*. For identical data and tag sizes, the number of variables goes down by half, as compared to the earlier technique. However, as results later show, with increasing n, memory requirements grow rapidly, even with the plain encoding scheme. So we must improve on it.

We can reduce the number of variables even further, by taking advantage of the *at most one match* CAM system invariant. Let $Tagin$ be $\vec{tin} = \langle tin_{t-1}, tin_{t-2}, ..., tin_0\rangle$. In order that a particular tag entry $T[]$ not match $\vec{tin}$, it should be equal to the ternary symbolic vector $ternneq(\vec{tin})$.

We now verify the associative read operation in two parts. First, we verify the case where no CAM entries match the input tag, and second, we verify the case where the i^{th} entry does match the input tag. For verifying the case where no hit occurs the new assertion is:

$$tin = bitvector(t)$$

$$\begin{array}{r}(op = assocread) \wedge (Tagin = \vec{tin}) \wedge \\ (T[0] = ternneq(\vec{tin})) \wedge (T[1] = ternneq(\vec{tin})) \wedge (T[2] = ternneq(\vec{tin})) \\ \stackrel{\text{LEADSTO}}{\Longrightarrow} \\ (HIT = 0),\end{array}$$

The three sets of position inequality variables associated with the three invocations to the $ternneq$ operator are not visible in the abstract specifications. For verifying the case where one entry matches the input tag, we write:

$$(op = assocread) \wedge (datain = \vec{tin}) \wedge (D[i] = \vec{data}) \wedge$$

$$
\begin{aligned}
&(when(i=0)(T[0]=\vec{tin})) \wedge (when(i \neq 0)(T[0]=ternneq(\vec{tin}))) \wedge \\
&(when(i=1)(T[1]=\vec{tin})) \wedge (when(i \neq 1)(T[1]=ternneq(\vec{tin}))) \wedge \\
&(when(i=2)(T[2]=\vec{tin})) \wedge (when(i \neq 2)(T[2]=ternneq(\vec{tin}))) \\
&\overset{\text{LEADSTO}}{\Longrightarrow} \\
&(HIT=1) \wedge (match[k]=(k=i)) \wedge (output=data)
\end{aligned}
$$

This approach requires only $(log_2 n + n \cdot log_2 t + t + d)$ Boolean variables. It results in a substantial space savings over the earlier two approaches. We refer to the Boolean variable encoding used above as the *CAM encoding*. As will be seen in section 4.4, verification of even moderate sized CAMs would have been intractable without an encoding at least as efficient as the CAM encoding.

4.3.2 Comparison of Boolean variables for different encodings

We have presented three different Boolean encoding techniques, $full$, $plain$ and CAM in the previous section. Table 1 shows the number of symbolic Boolean variables required in each approach for a number of different CAM configurations. A column has been added to show the number of variables required for representing the transition relation of the system for CTL model checking [29]. This is always twice the number required for the full encoding. In the table it is interesting to note that the CAM encoding shows a large savings in the number of Boolean variables required for verifying CAMs with high associativity.

4.4 CAM Verification Experimental Results

Experimental Setup

Using cmu-netlist, a tool to describe transistor netlists, we implemented a number of different CAMs, all based on the CAM of Figure 4.1, with varying numbers of entries, and varying tag and data sizes. We used the Voss formal verification system [116] to verify the circuits described in this section, and those in chapter 6.

CAM configuration	Encoding Type			
$n \times t \times d$	*CAM*	*plain*	*full*	*trans. reln.*
$4 \times 4 \times 4$	18	26	36	72
$16 \times 4 \times 4$	44	76	132	264
$4 \times 16 \times 4$	38	86	96	192
$4 \times 4 \times 16$	30	38	84	168
$16 \times 16 \times 16$	100	292	528	1056

Table 4.1: Total number of Boolean variables required for different encodings. Tag size = t, Number of entries = n, Data size = d

4.4.1 Results

In Figures 4.4, 4.5, and 4.6 we have plotted the experimental results for verification of CAMs of different sizes using the CAM encoding and the plain encoding. Full encoding, which is not included here, usually performs much worse than the other two encodings. We have plotted the memory taken by the OBDDs generated in the process of verifying the associative read operation for the CAM and the plain variable encodings. All other operations of the CAM take less space and they have not been included here. The Boolean variable ordering for the experiments was carefully chosen in order to avoid unfair comparison between the two techniques. We first chose a variable ordering that, from our understanding of the circuit function, would give us small OBDDs. Upon running STE with our initial variable ordering, the OBDD package reordered some of the variables. We used this reordering information to improve our understanding of the variable interaction and further tuned the variable ordering before running STE again.

Figure 4.4 shows how the OBDD sizes vary for the plain and CAM encoding for CAMs with varying associativities (tag, data sizes are constant). As the graph shows, there is a dramatic difference in the space taken by the two encoding approaches. As the number of tag entries increase, the plain encoding requires substantially more memory than the CAM encoding. Many TLBs are highly associative. For example, the PA-7300LC processor has a 96-entry fully associative TLB [61]. For such a circuit,

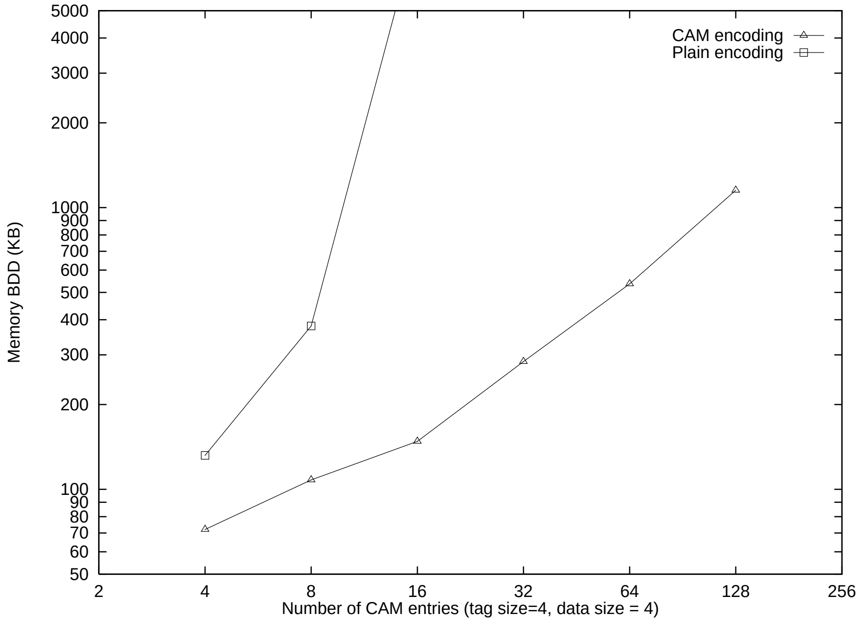

Figure 4.4: CAM: number of tag entries vs. OBDD sizes

the plain encoding approach will clearly not work. These results motivated us to use CAM encodings in all our remaining CAM verification experiments which are described in Chapter 6.

In Figures 4.5 and 4.6, we have shown the OBDD size trends for the two encodings when the tag size changes (others remaining constant), and when the data size changes. Although the results of these graphs are not as dramatic as that of Figure 4.4, they show that the use of CAM encoding still results in at least an order of magnitude savings in OBDD space, as compared to simpler encodings.

4.4.2 Discussion of CAM results

We can explain the trends in the results in the previous section in terms of the circuit structure and the interactions of the boolean functions in the circuit. Consider the 3-entry CAM postulated in Section 4.3.1, and let the tag size be k. In this design the i^{th} match line, $match[i]$ contains the result of match between the tag input and the i^{th} tag entry.

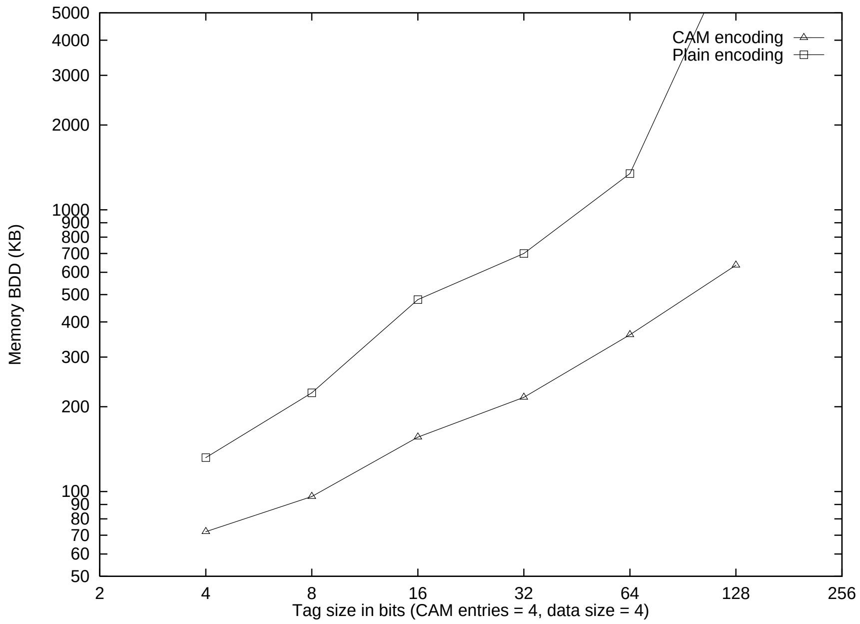

Figure 4.5: OBDD trends with varying tag size, data size and number of entries remaining constant.

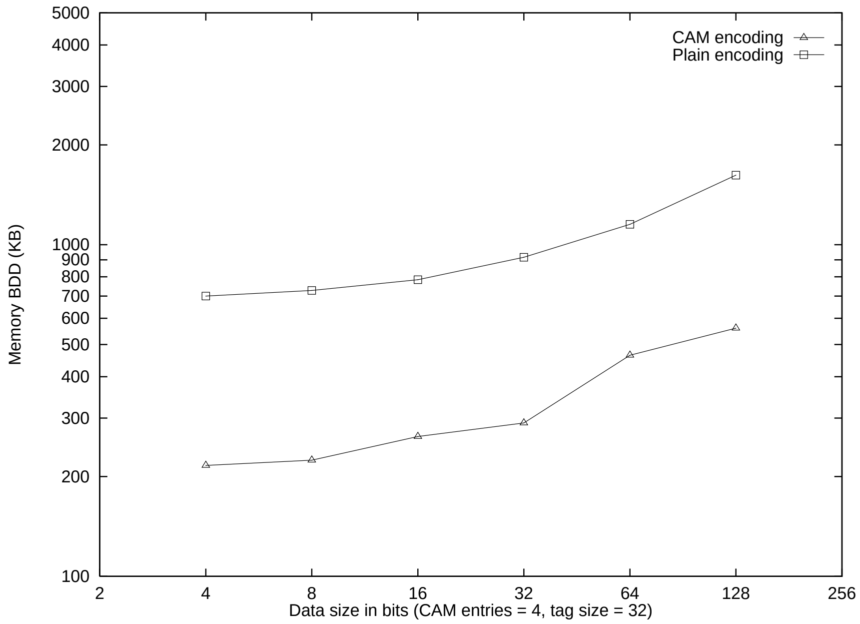

Figure 4.6: OBDD trends with varying data size, tag size and number of entries remaining constant.

When the plain encoding is used, then the i^{th} match line contains the result of the match between the input tag $\vec{tagin}$ and the i^{th} tag entry $\vec{tag_i}$. After the compare, the boolean function associated with $match[i]$ is

$$f_{match[i]} = \neg((tagin[k-1] \oplus tag_i[k-1]) \vee \ldots \vee (tagin[1] \oplus tag_i[1]) \vee (tagin[0] \oplus tag_i[0]))$$

The value on each dataout line $Dataout[j]$ is a function of the functions on all the match lines, bit $d[j]$ (used in the associative read assertion), and $\vec{i}$. So, potentially all the Boolean variables associated with the tag and data entries, and the tag input interact with each other.

When the CAM encoding is used, the antecedent fragment (Section 4.3.1) specifying the 0th tag entry is given by $(when(\vec{i} = 0)(T[0] = \vec{tin})) \wedge (when(\vec{i} \neq 0)(T[0] = ternneg(\vec{tin})))$ When the tag input is tin, then the 0th tag entry matches only if $\vec{i} = i_1 i_0$ equals 0. This information is conveyed by the Boolean function on $match[0]$. Therefore $f_{match[0]} = \overline{i_1} \cdot \overline{i_0}$. So, the functions on the dataout lines depend only on the Boolean variables in $\vec{d}$, and $\vec{i}$. Thus, the use of CAM encoding minimizes the variable interaction and this results in substantial space savings especially when the number of entries is large. We have not shown the running times of the assertions here, most of which finish in a few seconds on a RS/6000™ model 250 workstation.

4.5 Variations of associative read

Some CAM designs do allow more than one tag entry to match the incoming tag [127, 81]. Such CAMs are not usually seen on a processor. However, they can be found in applications like cryptology, data routing etc. [115]. In such CAMs, there can be two variations on the associative read operation described in Section 4.3.1. In the first variation there is a priority among the tag entries and in case of multiple matches, the highest priority match dominates [127]. In the second variation, the total number of matches is counted [54, pp. 156–167] and this number is an output of the CAM unit.

4.5.1 Multiple matches with prioritization

In this variation, from the multiple matches, the highest priority match is selected. Usually the priority among the matches is based on the address. A match at a higher address has a higher priority than a match at a lower address[2] [127].

In such a CAM, the case where HIT is 0 can be verified as described in the previous section. However, the case where there are multiple matches need a slight change. For this the antecedent must state that the i^{th} tag entry is the same as the input tag and all tag entries greater than i do not match the input tag. The consequent must then verify that only the i^{th} match line is high and all others remain low. This can be done by a simple modification of the last assertion shown in the previous section.

4.5.2 Counting matches

For counting matches, with every tag entry we include a single Boolean variable which if true ensures that the tag matches the tag input, and if false ensures that the tag does not match the tag input. In the 3-entry tag example, we can have a variable a_i, with the i^{th} tag entry, and the antecedent entry corresponding to this entry should be $(when(a_i)(T[i] = \vec{tin})) \wedge (when(\neg a_i)(T[i] = terneq(\vec{tin})))$.

Let the *count* output of the circuit give the number of tag matches. The consequent below checks that the value of *count* equals the number of a_is that are true.

$$
\begin{aligned}
(when(\neg a_0 \wedge \neg a_1 \wedge \neg a_2)(count = 0)) \wedge \\
(when((a_0 \wedge \neg a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge \neg a_1 \wedge a_2))(count = 1)) \wedge \\
(when((\neg a_0 \wedge a_1 \wedge a_2) \vee (a_0 \wedge \neg a_1 \wedge a_2) \vee (a_0 \wedge a_1 \wedge \neg a_2))(count = 2)) \wedge \\
(when(a_0 \wedge a_1 \wedge a_2)(count = 3))
\end{aligned}
$$

4.6 Related Work

While work on verification of memory arrays has been reported in [24], and [102], there has been little published on the particular needs of CAMs. In [24, pp. 102], Bryant discusses the formal verification of SRAM arrays using ternary simulation, and he states the difficulty of CAM verification as, "*...Other classes of memory designs can also be verified by simulating a linear, or nearly-linear number of patterns. ...*

[2]The other variation of a lower address having a higher priority is also possible

On the other hand, content-addressable memories do not seem to fit into this class, since it is not easy to identify where a particular datum will be stored." In [102], we reported on verification of a 4-way set associative cache tags unit. Each set of the cache tags unit may be considered a content addressable memory. However, this work, which was completed prior to our work on CAM verification, uses a variant of plain encoding to verify the correctness of each set.

Chapter 5

State node identification in circuits

Our verification methodology for arrays partitions the system specification into the set of abstract assertions, and the implementation mapping. The implementation mapping describes how the abstract state is realized in the implementation by mapping the abstract state values to values on circuit nodes The goal of state node identification is to identify the set of circuit nodes corresponding to each abstract state variable.

We start this chapter with a motivation for the problem (Section 5.1). We next describe the theory behind our *partition refinement* based approach to solve the problem (section 5.2). Section 5.3 discusses the non-deterministic Moore machine specified by a set of abstract assertions. It shows how a ternary simulation model can be derived from the set of abstract assertions. The following section describes the state node identification algorithm. Section 5.5 discusses some specific instances where we used our state node identification techniques.

5.1 Motivation

The typical design environment uses a simulation based verification methodology. In such an environment, verification is based on applying simulation patterns at the primary inputs of the design, observing the the primary outputs, and comparing them to the expected response. Seldom do these verification techniques examine internal state points in the design. Therefore, most tools such as circuit extractors, netlisters etc. do not maintain information about storage nodes or state points in a circuit, as they do for the primary inputs and outputs of a design. Exposing internal state, however, is an integral part of our methodology. This makes it necessary to develop techniques to facilitate the identification of state nodes. The ideal technique should be general enough to work on a variety of design representations which can range from transistor and gate-level netlists, to RTL and behavioral models. We have developed an approach to do so, and we describe it in the sections ahead.

Sometimes, it is possible for the designer to annotate his designs with suitable attributes to facilitate state node detection. Therefore, the need for state node identification may not seem particularly acute. However, addition of extra steps and/or new attributes and design requirements can result in changes to the existing design methodology. An automated technique such as ours has the ability to do the identification in a completely non-invasive manner. Furthermore, we should view our techniques as complementing any other available approaches to do state node identification. Sometimes, design representations we would verify are not under full control of the designers. For example, if we wished to verify a synthesized gate-level or a transistor-level netlist, the ability of the designer to add attributes to the final design representation is significantly limited. So an automated technique to identify state nodes is an important verification aid. The use of automated state node identification has been an important factor in the success we have had in verifying several memory arrays from PowerPC microprocessors [102, 103].

5.2 The partition refinement approach to state node identification

5.2.1 Ternary state machines

We use the following terminology below. Let S be a finite set. Let $|S|$ denote the cardinality of S. A partition π_S of S is a set of pairwise disjoint subsets of S, whose union is S. The elements of π_S are called *blocks*.

We use ternary symbolic simulation of circuits and specification as a part of the

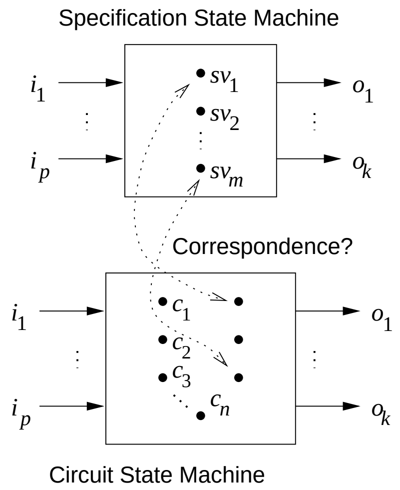

Figure 5.1: The state node identification problem

identification process. This motivates the definition of a *ternary state machine* ahead.

Definition 14 *A ternary state machine is a 3-tuple* $\langle I, S, \delta\rangle$*, where* $I = \{i_1, i_2, ..., i_p\}$ *is the set of input nodes,* $S = \{s_1, s_2, ..., s_n\}$ *is the set of state variables, and* $\delta : \mathcal{T}^{|I|} \times \mathcal{T}^{|S|} \rightarrow \mathcal{T}^{|I|}$ *is the state transition function which maps the current values on the input and state nodes to the new circuit state.*

This definition captures the behavior of switch-level and gate-level ternary simulation models. We can also construct a ternary state machine from a set of abstract assertions, which implicitly define a finite state machine [6, pp.77-78]. This construction is described in section 5.3. Note that our definition treats nodes corresponding to the primary outputs of the circuit as state nodes. Of course, values on such state nodes will not influence the new circuit state.

The *circuit state machine* (CSM) is the ternary state machine $\langle I_c, S_c, \delta_c\rangle$ defined by the unit-delay switch-level model of the circuit [17], where δ_c corresponds to the circuit excitation function. Let S_c equal $\{c_1, c_2, ..., c_n\}$.

The *specification state machine* (SSM) is the ternary state machine $\langle I_s, S_s, \delta_s\rangle$ synthesized from a set of abstract assertions of the system. Let S_s equal $\{sv_1, sv_2, ..., sv_m\}$. The goal of state node identification is to determine, for a subset $S_{ID} \subseteq S_s$ of the SSM state elements, what circuit state machine nodes correspond to each state node

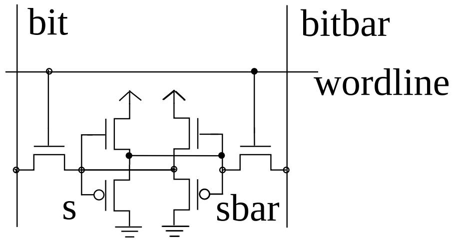

Figure 5.2: Static RAM memory cell

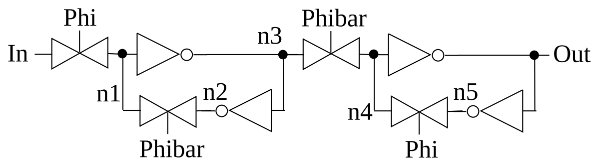

Figure 5.3: Master-Slave flip-flop

in S_{ID}. We formulate and solve the problem as that of determining a partition $\pi = \{B_1, ..., B_{|S_{ID}|}, B_{|S_{ID}|+1}\}$ of the set $(S_s \cup S_c)$ such that each block $B_i, 1 \leq i \leq |S_{ID}|$ of the partition contains contains one SSM state node from S_{ID}, and the circuit state nodes which correspond to it.

We make the assumption that corresponding to every abstract state variable which is not a primary input or output, there exists a sequential storage element of some other form. These can include storage elements like random access memory cells (Figure 5.2) or master-slave flip-flops (Figure 5.3), or any other type of static or dynamic storage element. This assumption does preclude the use of complex mappings, where the state of an abstract state variable corresponds conditionally to the states of several different latches. However, this is not a particularly onerous restriction. Under such a restriction the abstract specification corresponds closely to the RTL abstraction. It is at this abstraction level that we specify the behavior of most arrays.

In the partition refinement approach, we first start with the partition which contains only one block, namely the set, $S_s \cup S_c$. Blocks in the partition reflect our knowledge of the correspondence between the circuit nodes and the abstract state variables. Naturally, at the beginning, when nothing is known about the correspon-

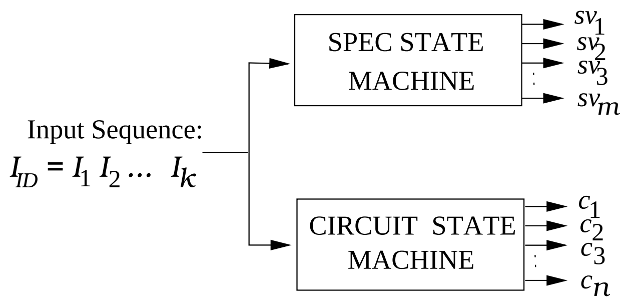

Figure 5.4: State node identification

dence, all the circuit nodes and the state variables are included in the same partition. As we refine our knowledge of the node correspondence (by simulating the two machines), the partition of the set $S_s \cup S_c$ contains smaller blocks, which indicates a more refined knowledge of the state node correspondence. At the end of the identification process, each partition block contains exactly one state variable and its corresponding circuit nodes.

5.2.2 Partition refinement by simulation

To determine the correspondence of the state nodes in SSM and the CSM, we start with both the machines in the *ternary reset state*, where all the nodes are set to X. Typically most switch-level, gate-level or RTL simulators have the ability to initialize the circuit model to all Xs. We associate the node partition $\pi_0 = \{S_s \cup S_c\}$ with this initial state where all the state nodes are indistinguishable. We apply a finite length input symbolic sequence $\mathcal{I}_{ID} = \mathcal{I}_1\mathcal{I}_2...\mathcal{I}_k$ to the SSM and the CSM (Figure 5.4). At the end of each clock cycle we observe the values on the nodes in S_s, and S_c. Below, we define the equivalence relation $\mathcal{C}_{\mathcal{I}_{ID}}$ which captures the correlation of the values on nodes in $S_s \cup S_c$ after an application of the input sequence $\mathcal{I}_{ID}$.

Definition 15 *Given nodes $n_1, n_2 \in (S_s \cup S_c)$, $n_1 \; \mathcal{C}_{\mathcal{I}_{ID}} \; n_2$ iff n_1 and n_2 acquire identical ternary values when the input sequence $\mathcal{I}_{ID}$ is applied to SSM and the CSM in their ternary reset states.*

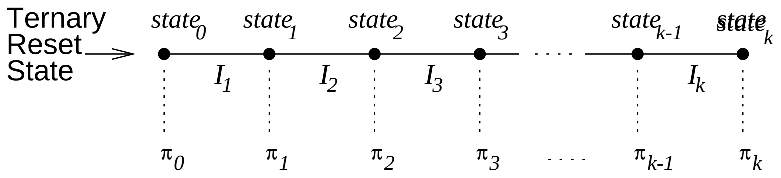

Figure 5.5: Partitioning the nodes of the SSM and the CSM.

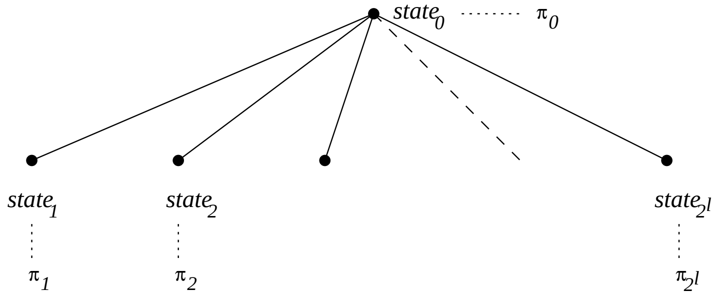

Figure 5.6: BFS node partitioning

The equivalence relation $\mathcal{C}_{\mathcal{I}_{ID}}$ partitions the set $S_s \cup S_c$. Consider Figure 5.5. It shows the application of an input sequence $\mathcal{I}_{ID} = \mathcal{I}_1\mathcal{I}_2...\mathcal{I}_k$ of length k to the circuit. Corresponding to each state $state_i$ of the CSM and the SSM circuits, there is a partition π_i of the nodes defined by the relation $\mathcal{C}_{\mathcal{I}_1\mathcal{I}_2...\mathcal{I}_k}$. We can further refine the partitions by taking their product,

$$\Pi = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_k \tag{5.1}$$

where $\pi_a \cdot \pi_b = \{x | x = y \cap z, y \in \pi_a, z \in \pi_b\}$. However, maintaining partitions $\pi_1, ...\pi_k$, and computing their product for a lengthy input sequence can be expensive.

A more efficient alternative is to use symbolic simulation to simulate multiple state sequences in parallel. If there are l symbolic Boolean variables in the input sequence, then with a single symbolic sequence we explore the equivalent of 2^l different non-symbolic paths from the initial state $state_0$ (Figure 5.6). Each path corresponds to one distinct assignment of Boolean variables. Given the 2^l partitions, we can compute their product $\Pi = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_{2^l}$ in a very simple manner — all the nodes having the same ternary symbolic value go into the same block of the partition Π.

This implies that if we can generate a symbolic simulation sequence which puts symbolic values on SSM nodes such that no two nodes in S_{ID} have identical values, and the value on any node in S_{ID} differs from the values on the nodes in $(S_s - S_{ID})$, then we can solve the state node identification problem. We call such a symbolic input sequence an *identification sequence*. After simulating such a sequence, the nodes in S_{ID} are said to be *distinguished*. For small state machines, an approach to obtain the identification sequence is to apply new symbolic values at the inputs for successive clock cycles until all the nodes in S_{ID} are distinguished. This gives the shortest possible identification sequence. However, this will not work for large state machines, where different symbolic values at the circuit inputs for each clock period over several clock cycles can lead to a blowup in the OBDD sizes. Generally, for large regular machines the construction of the identification sequence is best left to the designer, who with his detailed knowledge can easily exploit the features of the design to construct an efficient identification sequence. We illustrate this with examples in section 5.5.

5.3 Construction of the specification state machine

The specification state machine is a ternary simulation model of a system derived from a set of abstract assertions describing the system behavior. We show the construction of the simulation model by going through a sequence of graduated steps. In section 5.3.1 we discuss the non-deterministic Moore machine defined by a set of abstract assertions. The idea of a set of abstract assertions defining a non-deterministic Moore machine comes from Beatty's work [6, pp. 77-78], and we have added some notation to facilitate the discussion in the sections that follow. We have illustrated the ideas in this section with the help of the SRAM example from section 2.3.2. Next, we show the construction of a transition relation to represent the transitions of the Moore machine, and we discuss representing the transition relation in a partitioned form. The partitioned transition relation we develop naturally defines a set of excitation functions which maps a binary system state to the ternary successor state of the system. This *binary-ternary* model which directly arises from the assertions is our *reference model.* We show how a minor change to the algorithm which creates the reference model results in an algorithm which yields ternary simulation model, or the SSM. We prove that the ternary model we create accurately represents the reference model.

5.3.1 Non-deterministic Moore machine described by assertions

Let $A_M = \{[A_1 \overset{\text{LEADSTO}}{\Longrightarrow} C_1], [A_2 \overset{\text{LEADSTO}}{\Longrightarrow} C_2], \ldots, [A_k \overset{\text{LEADSTO}}{\Longrightarrow} C_k]\}$ be a set of abstract assertions. Let $\mathcal{SV} = \{sv_1, sv_2, ..., sv_m\}$ be the set of state variables of M. As shown in [6], A_M defines a non-deterministic Moore machine M over the state variables in $\mathcal{SV}$. Let the set of case variables in the assertions of A_M be $\mathcal{CV} = \{cv_1, cv_2, ..., cv_l\}$. As discussed earlier, the case variables help combine multiple non-symbolic assertions into one symbolic assertion. To simplify the discussion here, we assume that the state and the case variables are over the Boolean domain. The state space of M is $\mathcal{B}^m$, where m is the number of state variables in M. The set of transitions of M is a subset of $\mathcal{B}^m \times \mathcal{B}^m$.

To discuss the algorithm for construction of the transition relation for M from a set of assertions A_M , we first define the syntax of abstract assertions. In an abstract assertion $[A \overset{\text{LEADSTO}}{\Longrightarrow} C]$, A, and C are abstract trajectory formulas (ATFs) which have the following recursive definition:

1. (a) $(sv_i = 0)$ is an ATF, where $sv_i \in \mathcal{SV}$.
   (b) $(sv_i = 1)$ is an ATF, where $sv_i \in \mathcal{SV}$.
2. $F_1 \wedge F_2$ is an ATF, where F_1, and F_2 are ATFs.
3. $(when\ e\ F)$ is a ATF, where F is an ATF, and e is a Boolean expression over the variables in $\mathcal{CV}$.

The syntax of the ATF is similar to that of TFs in section 2.1.3 with the notable absence of the next-time operator, $\mathbf{X}$. An abstract assertion of the form $[A \overset{\text{LEADSTO}}{\Longrightarrow} C]$ may be considered a trajectory assertion of the form $[A \Rightarrow \mathbf{X}C]$ for a model structure in the abstract domain.

Beatty [6], and Jain [71] present a more elaborate syntax for abstract assertions. This includes a few additional constructs not directly included in the description of ATFs. However, the ATF syntax we have presented above captures the core features of their language. The features we have not included can be expressed in terms of the ATF primitives. For example, an expression such as $(\mathtt{M}[i] = a)$, which denotes symbolic indexing over an array of state elements $M[0..n-1]$, is apparently not a part of our language. However, this expression can be written as the ATF $(when\ (i = 0)\ (\mathtt{M}[0] = a)) \wedge \ldots (when\ (i = n-1)\ (\mathtt{M}[n-1] = a))$. Similarly, $(\mathtt{din} = a)$ can be written as the ATF $(when\ a\ (\mathtt{din} = 1)) \wedge (when\ \overline{a}\ (\mathtt{din} = 0))$.

Each assertion $[A_i \stackrel{\text{LEADSTO}}{\Longrightarrow} C_i]$ in A_M defines a superset of the transitions of M. Intuitively, the assertion states that if M starts in a state satisfying A_i, then its next state must be within the set of states specified by C_i. However, if M starts in a state not satisfying A_i, the assertion does not place any constraint on the next state of M. We denote the set of transitions specified by this assertion as $\delta_{A_iC_i}$. The set of assertions in A_M together specify, δ_{A_M}, the transitions of M, which equals $\cap_{i=1}^{n}\delta_{A_iC_i}$, the intersection of the transitions specified by the individual assertions.

SRAM non-deterministic Moore machine: An illustration

Consider the set of abstract assertions for the SRAM circuit in section 2.3.2:

$$(\mathtt{adr} = i) \wedge (\mathtt{M}[i] = a) \wedge (\mathtt{rd} = 1) \wedge (\mathtt{wr} = 0) \stackrel{\text{LEADSTO}}{\Longrightarrow} (\mathtt{M}[i] = a) \wedge (\mathtt{dataout} = a) \tag{5.2}$$

$$(\mathtt{adr} = i) \wedge (\mathtt{din} = a) \wedge (\mathtt{rd} = 0) \wedge (\mathtt{wr} = 1) \stackrel{\text{LEADSTO}}{\Longrightarrow} (\mathtt{M}[i] = a) \tag{5.3}$$

$$(\mathtt{M}[i] = a) \wedge (\mathtt{adr} = j) \stackrel{\text{LEADSTO}}{\Longrightarrow} (when(i \neq j) \rightarrow (\mathtt{M}[i] = a)) \tag{5.4}$$

The assertions described by equations 5.2 and 5.3 specify the SRAM read and write operations respectively. Equation 5.4 specifies that during any operation, unaddressed memory locations remain unchanged. The set of state variables in the SRAM abstract machine is $\mathcal{SV} = \{\mathtt{rd}, \mathtt{wr}, \mathtt{adr}[0], \mathtt{adr}[1], \mathtt{din}, \mathtt{dout}, \mathtt{M}[0], \mathtt{M}[1], \mathtt{M}[2], \mathtt{M}[3]\}$. The state space of the non-deterministic Moore machine defined by the SRAM assertions is $\mathcal{B}^{10}$, the set of all the 0/1 assignments to the SRAM state variables. The set of case variables equals $\mathcal{CV} = \{i[0], i[1], a, j[0], j[1]\}\}$. As an illustration of the utility of the case variables, consider the SRAM write assertion (equation 5.3). This assertion represents eight non-symbolic assertions, which correspond to the eight different assignments to the three case variables $i[0]$, $i[1]$, and a in the write assertion. A subset of these eight non-symbolic assertions have been shown below:

$$(\texttt{adr} = 00) \wedge (\texttt{din} = 0) \wedge (\texttt{rd} = 0) \wedge (\texttt{wr} = 1) \stackrel{\text{LEADSTO}}{\Longrightarrow} (\texttt{M[0]} = 0) \tag{5.5}$$

$$(\texttt{adr} = 00) \wedge (\texttt{din} = 1) \wedge (\texttt{rd} = 0) \wedge (\texttt{wr} = 1) \stackrel{\text{LEADSTO}}{\Longrightarrow} (\texttt{M[0]} = 1) \tag{5.6}$$

$$\ldots$$

$$(\texttt{adr} = 11) \wedge (\texttt{din} = 0) \wedge (\texttt{rd} = 0) \wedge (\texttt{wr} = 1) \stackrel{\text{LEADSTO}}{\Longrightarrow} (\texttt{M[3]} = 0) \tag{5.7}$$

$$(\texttt{adr} = 11) \wedge (\texttt{din} = 1) \wedge (\texttt{rd} = 0) \wedge (\texttt{wr} = 1) \stackrel{\text{LEADSTO}}{\Longrightarrow} (\texttt{M[3]} = 1) \tag{5.8}$$

Thus, we may view the set of SRAM assertions in equations 5.2, 5.3, and 5.4 as defining a large number of non-symbolic assertions. Each of these assertions define a set of transitions over the SRAM state space. For example, the assertion on line 5.6 describes the write operation for storing a 1 in the 0th memory location. If the initial state of the system conforms to the antecedent of the assertion, then the next state of the system should be such that $\texttt{M[0]} = 1$. The assertion specifies no other constraints on the next state. This is illustrated in Figure 5.7, where starting at state S_0, S_1, and S_2 are both valid next states. However, if the initial state does not conform to the antecedent, then no constraints are placed on the next state of the system. As the figure shows, starting at S_3, any possible system state is a valid new state.

The S_0S_1 transition states that if 1 is written to `M[0]`, then this location contains a 1 after the write is complete. Note that in the transition, the state variables corresponding to the primary inputs, and the unaddressed memory locations all change to 1. Since primary inputs like `rd` and `wr` are controlled by the environment, such state variables can change non-deterministically. However, spontaneous changes in unaddressed memory locations does not conform to our intuition of correct memory operation. It is the set of transitions specified by assertion 5.4 that prevents such behavior from occurring in the abstract machine. This assertion states that during any operation, memory locations that are not addressed retain their value. The intersection of the transitions specified by assertions 5.2, 5.3, and 5.4 is the set of valid transitions of the abstract machine specified by the SRAM assertions.

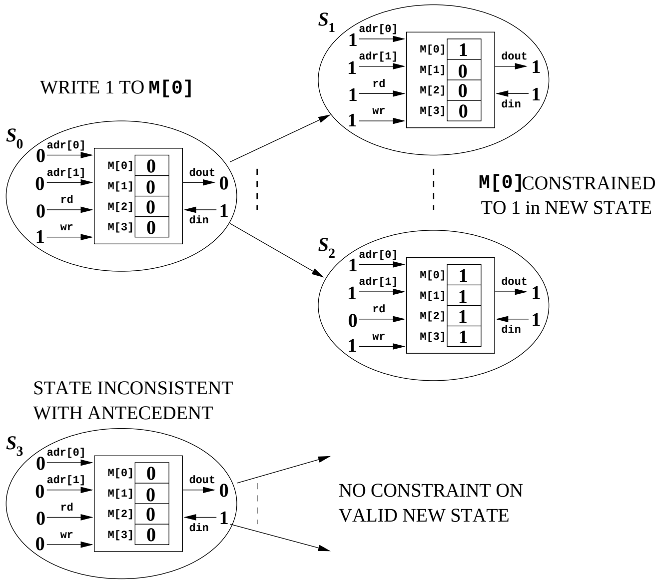

Figure 5.7: Transitions of the SRAM defined by the assertion of equation 5.6.

5.3.2 Construction of a state transition relation from assertions

To obtain the system transition relation δ_{A_M}, it is necessary to compute $\delta_{A_iC_i}$ for every assertion $[A_i \overset{\text{LEADSTO}}{\Longrightarrow} C_i]$ in A_M. We represent $\delta_{A_iC_i}$ by its characteristic function represented as an OBDD over a set of Boolean variables. δ_{A_M} is simply the conjunction of the functions representing $\delta_{A_1C_1}$, $\delta_{A_2C_2}$, ..., $\delta_{A_kC_k}$.

Below, we describe the algorithm for the construction of δ_{AC} for an assertion $[A \overset{\text{LEADSTO}}{\Longrightarrow} C]$. There are three steps in the algorithm. The first is that for each state variable, two Boolean variables are introduced, one for the "old" value, and one for the "new" value. δ_{AC} is represented as a Boolean function of all these variables, where the function yields 1 when the old and the new state are related, and 0 otherwise. Therefore, we introduce two sets of variables, $P = \{p_1, p_2, ..., p_m\}$, and $N = \{n_1, n_2, ..., n_m\}$, to symbolically represent the present state and the next state, respectively.

In the second step, we compute two Boolean functions, $A(P)$ and $C(N)$. When A does not contain any case variables, then $A(P)$ is the characteristic function representing the set of states denoted by A in terms of variables in P. Similarly, in the absence of case variables in C, $C(N)$ is the characteristic function representing the set of states denoted by C in terms of variables in N. The general procedure for computing $A(P)$ and $C(N)$ is described below.

Let F be an ATF, and let $V = \{v_1, v_2, ..., v_m\}$ be a set of Boolean variables to symbolically represent the states of M. $F(V)$ may be constructed by transforming F as follows:

1. (a) If F is $(sv_i = 0)$, $F(V)$ is the Boolean expression $\overline{v_i}$.
   (b) If F is $(sv_i = 1)$, $F(V)$ is the Boolean expression v_i.
2. If $F = F_1 \wedge F_2$, then $F(V) = F_1(V) \wedge F_2(V)$.
3. If F is $(when\ e\ F_1)$, then $F(V) = \overline{e} \vee F_1(V)$.

Each symbolic assertion actually represents a number of non-symbolic assertions. So the set of transitions of the non-symbolic assertion is the intersection of the set of transitions in all the non-symbolic assertions. This justifies the final step, where we compute δ_{AC} by universally quantifying $\overline{A(P)} \vee C(N)$ over the case variables in $\mathcal{CV}$:

$$\delta_{AC} = \forall cv_1, cv_2, ..., cv_l \in \mathcal{CV}.\ \overline{A(P)} \vee C(N) \tag{5.9}$$

In a set of assertions with a large number of case variables, we can perform a *case variable substitution* which often eliminates a substantial number of the universal quantifications necessary to construct δ_{AC}. Case variable substitution works in the following manner. If the antecedent A of an assertion $[A \overset{\text{LEADSTO}}{\Longrightarrow} C]$ is of the form $A' \wedge (sv_i = cv_j)$, then

$$\delta_{AC} = \forall cv_1, ..., cv_{j-1}, cv_{j+1}, ..., cv_l \in \mathcal{CV}.\ \overline{A'(P)[p_i/cv_j]} \vee C(N)[p_i/cv_j], \tag{5.10}$$

where $A'(P)[p_i/cv_j]$ and $C(N)[p_i/cv_j]$ are obtained by substituting p_i, the present state variable corresponding to cv_i for cv_j. Typically, by eliminating a large number of quantification operations, this optimization step substantially speeds up the construction of the transition relation.

5.3.3 Efficient transition relation construction

In a system with a large number of state variables, it may not be possible to construct or do computations with the transition relation δ_{A_M} in a reasonable amount of time and memory. In order to overcome this problem, we use the idea of representing the transition relation by a set of smaller transition relations which are implicitly conjuncted [31, 55]. So, rather than construct a monolithic transition relation δ_{A_M}, for each state variable sv_i, we construct a transition relation δ_{A_M,sv_i} which specifies how the variable sv_i gets updated. Such smaller transition relations closely resemble the description of a system in terms of next state functions of its state variables. However, unlike next state functions, these transition relations can capture non-determinism in systems. The conjunction of $\delta_{A_M,sv_1}, \ldots, \delta_{A_M,sv_m}$ equals δ_{A_M}. This is made possible because of the fact that for the purpose of computing the transition relation, the consequent of assertions can be "broken" into smaller terms, each of which contains at most one state variable. We discuss this below.

Consider the assertion $[A \overset{\text{LEADSTO}}{\Longrightarrow} C]$, where $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$. It is easy to see that $C(N) = C_1(N) \wedge \ldots \wedge C_n(N)$. So δ_{AC}, which equals $\forall cv_1, cv_2, ..., cv_l \in \mathcal{CV}.\ \overline{A(P)} \vee (C_1(N) \wedge \ldots \wedge C_n(N))$ may be written as:

$$(\forall cv_1, ..., cv_m \in \mathcal{CV}.\ \overline{A(P)} \vee C_1(N)) \wedge \ldots \wedge (\forall cv_1, ..., cv_m \in \mathcal{CV}.\ \overline{A(P)} \vee C_n(N)) \quad (5.11)$$

This justifies "breaking" an assertion like $[A \overset{\text{LEADSTO}}{\Longrightarrow} C_1 \wedge \ldots \wedge C_n]$ into a set of simpler assertions of the form $[A \overset{\text{LEADSTO}}{\Longrightarrow} C_1], \ldots, [A \overset{\text{LEADSTO}}{\Longrightarrow} C_n]$. The conjunction of the transition relation for each of these smaller assertions represents the transition relation for the original assertion.

An ATF F of the form $(when\ e\ (F_1 \wedge F_2))$ is the same as $(when\ e\ F_1) \wedge (when\ e\ F_2)$, and an ATF F of the form $(when\ f\ (when\ g\ F_1))$ may be written as $(when\ (f \wedge g)\ F_1))$. Thus any ATF F may be rewritten as $F_1 \wedge F_2 \wedge \ldots \wedge F_k$, where each F_i contains at most one abstract state variable. So, each assertion $[A_i \overset{\text{LEADSTO}}{\Longrightarrow} C_i]$ in the set A_M, may be decomposed into a set of smaller assertions $\{[A_i \overset{\text{LEADSTO}}{\Longrightarrow} C_{i,sv_1}], \ldots, [A_i \overset{\text{LEADSTO}}{\Longrightarrow} C_{i,sv_m}]\}$ such that the consequent C_{i,sv_j} contains no state variables other than sv_j. This allows us to construct δ_{A_M,sv_j}, the jth conjunct of the partitioned transition relation as:

$$\delta_{A_M,sv_j} = \wedge_{i=1}^{k} \delta_{A_i C_{i,sv_j}}$$

5.3.4 Construction of a ternary simulation model

A ternary simulation model of a system consisting of state variables $sv_1, sv_2, \ldots, sv_m$ consists of a set of m ternary excitation functions, $f_1, f_2, \ldots, f_m$, where $f_i : \mathcal{T}^m \rightarrow \mathcal{T}$ states how the new ternary state of sv_i is computed, given the current ternary state of the system. We use the dual-rail encoding to represent the value of each state variable as a pair of Boolean variables, and the excitation f_i as a pair of Boolean functions $f_i.L$, and $f_i.H$ over these Boolean variables.

First, let us consider the partitioned transition relation obtained from the previous section. δ_{A_M, sv_j}, the jth conjunct of the partitioned transition relation represents how sv_j gets updated given the previous state of the system. So, the conditions under which sv_j will be low, and high are expressed by equations 5.12 and 5.13, respectively.

$$F_j.L = \delta_{A_M, sv_j}|_{n_j=0} \tag{5.12}$$

$$F_j.H = \delta_{A_M, sv_j}|_{n_j=1} \tag{5.13}$$

The intuition is that $F_j.L$ and $F_j.H$ represent sets of previous states which make sv_j 0 and 1 respectively, in the next state. For a given state, if $F_j.L$ and $F_j.H$ are both 1, then sv_j is driven to X in the next state. Thus, non-determinism is captured by the ternary value X. However, $F_j.L$, and $F_j.H$ as constructed above, are not satisfactory as a ternary simulation model since they are not defined for ternary domains. Rather, they express how a binary state of the system maps to a new ternary state. We term these functions as the *binary-ternary* excitation, or the *binary-ternary* model of the system. Since this model arises directly from the semantics of given set of abstract assertions, we regard it as the *reference model* for the system.

A simple modification to the procedure for obtaining the binary-ternary model gives us the procedure which yields the ternary model of the system. This modification expands the set of state variables, so that in place of every original state variable sv_i, we have a pair of state variables, $sv_i.H$ and $sv_i.L$, to represent the system state encoded as dual rail values. We term these variable expansions in assertions as the *dual rail transformations*. This simplifies the construction of a set of excitation functions which can handle the case when both $sv_i.H$, and $sv_i.L$ are equal to 1 in the initial state, i.e., sv_i is X in the initial state.

Applying the dual rail assertion transformation on an ATF F yields another ATF which denoted by $F^{\mathcal{D}}$. We define this transformation below recursively:

1. (a) If $F = (V_i = 0)$, then $F^{\mathcal{D}} = (V_i.H = 0) \wedge (V_i.L = 1)$.

   (b) If $F = (V_i = 1)$, then $F^{\mathcal{D}} = (V_i.H = 1) \wedge (V_i.L = 0)$.

2. If $F = F_1 \wedge F_2$, where F_1, and F_2 are ATFs, then $F^{\mathcal{D}} = F_1^{\mathcal{D}} \wedge F_2^{\mathcal{D}}$.

3. If $F = (when\ e\ F_1)$, where F_1 is an ATF, then $F^{\mathcal{D}} = (when\ e\ F_1^{\mathcal{D}})$.

Applying this transformation to an assertion $[A \overset{\text{LEADSTO}}{\Longrightarrow} C]$ yields $[A^{\mathcal{D}} \overset{\text{LEADSTO}}{\Longrightarrow} C^{\mathcal{D}}]$. The application of this transformation to every assertion in set A_M yields $A_M^{\mathcal{D}}$.

Using the procedure of section 5.3.4, a partitioned transition relation is created from the assertions in $A_M^{\mathcal{D}}$. The transition relation uses two sets of variables, $P^{\mathcal{D}}$, and $N^{\mathcal{D}}$, to symbolically represent the present state and the next state, respectively, for the modified set of assertions. For each $i, 0 < i \leq m$, $P^{\mathcal{D}}$ contains Boolean variables $p_i.H$, and $p_i.L$, and N contains Boolean variables $n_i.H$, and $n_i.L$, which correspond to old and new values of state variables $sv_i.H$, and $sv_i.L$, respectively. The ternary excitation functions for V_j may be computed as:

$$f_j.L = \delta_{A_M^{\mathcal{D}}, sv_j.L}|_{n_j.L=1} \tag{5.14}$$

$$f_j.H = \delta_{A_M^{\mathcal{D}}, sv_j.H}|_{n_j.H=1} \tag{5.15}$$

We justify the dual rail transformation by proving that the ternary model described by equations 5.14 and 5.15 is a *consistent monotonic extension* of the reference model given by equations 5.12, and 5.13.

The ternary model is said to be *consistent* with the binary-ternary model if for identical binary vectors both yield identical next state results. The ternary model is an *extension* of the binary-ternary model if for all ternary vectors y "weaker" than a given binary vector x, the response of the ternary excitation to y is weaker than the response of the ternary-binary model to x. Thus, intuitively this concept captures the fact that the ternary model is an extended version of the ternary-binary model, and both never "conflict". The last property is that of the monotonicity of the ternary model. It states that for vectors y_1, and y_2, if $y_1 \sqsubseteq_{\mathcal{T}} y_2$, then the response of the ternary model to y_1 should be weaker that its response to y_2, i.e., the ternary model should be consistent with decreasing information. Below, we show that the ternary model we create satisfies these three important properties.

Theorem 5 *Let $f_i.H$, $f_i.L$, $1 \leq i \leq n$, and $F_i.H$, $F_i.L$, $1 \leq i \leq n$ be the ternary, and the binary-ternary excitation functions, respectively, for a system defined by a set of assertions A_M over state variables $sv_1, sv_2, ..., sv_n$.*

1. **Consistency:** *The following holds for all $x \in \mathcal{B}^n$:*

$$f_i.H(x) = F_i.H(x)$$
$$f_i.L(x) = F_i.L(x)$$

2. **Extension:** *For all $y \in \mathcal{T}^n$, $x \in \mathcal{B}^n$ such that $y \sqsubseteq_{\mathcal{T}} x$:*

$$F_i.H(x) \Rightarrow f_i.H(y)$$
$$F_i.L(x) \Rightarrow f_i.L(y)$$

3. **Monotonicity:** *For $y_1, y_2 \in \mathcal{T}^n$, if $y_1 \sqsubseteq_{\mathcal{T}} y_2$, then:*

$$f_i.H(y_2) \Rightarrow f_i.H(y_1)$$
$$f_i.L(y_2) \Rightarrow f_i.L(y_1)$$

Proof: Below, we prove the three properties, assuming all the assertions in A_M are non-symbolic. These results also carry over to symbolic assertions, since they can be rewritten as a set of non-symbolic assertions, one for each possible assignment to the case variables they contain.

1. Consistency.

Consider the assertion $[A_i \overset{\text{LEADSTO}}{\Longrightarrow} C_{i,sv_j}]$. Since the assertion is non-symbolic, $A_i(P)$ is a product term of variables in P, or their complements, and $C_{i,sv_j}(N)$ equals one of the three values: n_j, or $\overline{n_j}$, or 1. Therefore, $\delta_{A_i C_{i,sv_j}}$, which equals $\overline{A_i(P)} \vee C_{i,sv_j}(N)$, is of the form

$$p_{i_1} + \ldots + p_{i_q} + \overline{p_{i_{q+1}}} + \ldots + \overline{p_{i_r}} + C_{i,sv_j}(N) \tag{5.16}$$

where $p_{i_1}, ..., p_{i_r}$, all belong to P.

Similarly, consider the abstract assertion $[A_i^{\mathcal{D}} \overset{\text{LEADSTO}}{\Longrightarrow} C_{i,sv_j.L}^{\mathcal{D}}]$, which is obtained from the dual rail transformation of $[A_i \overset{\text{LEADSTO}}{\Longrightarrow} C_{i,sv_j}]$. $\delta_{A_i^{\mathcal{D}} C_{i,sv_j.L}^{\mathcal{D}}}$, which represents the transitions of this assertion equals

$$p_{i_1}.H + \overline{p_{i_1}.L} + \ldots + p_{i_q}.H + \overline{p_{i_q}.L} + p_{i_{q+1}}.L + \overline{p_{i_{q+1}}.H} + \ldots$$
$$+p_{i_r}.L + \overline{p_{i_r}.H} + C^{\mathcal{D}}_{i,sv_j.L}(N^{\mathcal{D}}) \quad (5.17)$$

If $C_{i,sv_j}(N)$ is equal to 1, then so is $C^{\mathcal{D}}_{i,sv_j.L}(N^{\mathcal{D}})$. If $C_{i,sv_j}(N)$ equals $n_j(\overline{n_j})$, then $C^{\mathcal{D}}_{i,sv_j.L}(N^{\mathcal{D}})$ equals $\overline{n_j.L}(n_j.L)$. Thus, for $n_j = 0$, and $n_j.L = 1$, $C_{i,sv_j}(N)$, and $C^{\mathcal{D}}_{i,sv_j.L}(N^{\mathcal{D}})$ are equal. Therefore, given $n_j = 0, n_j.L = 1$, and $x_i \in \mathcal{B}, 1 \leq i \leq m$, it is easy to see that the substitutions, $p_1 = x_1, \ldots, p_m = x_m$ and $p_1.L = \overline{x_1}, p_1.H = x_1, \ldots, p_m.L = \overline{x_m}, p_m.H = x_m$ keep the expressions in equations 5.16, and 5.17 equal. This proves that

$$(\delta_{A^{\mathcal{D}}_i C^{\mathcal{D}}_{i,sv_j.L}}|_{n_j.L=1})|_{p_1.L=\overline{x_1},p_1.H=x_1,\ldots,p_m.L=\overline{x_m},p_m.H=x_m} = (\delta_{A_i C_{i,sv_j}}|_{n_j=0})|_{p_1=x_1,\ldots,p_m=x_m} \quad (5.18)$$

Thus, $F_j.L(x)$ equals

$$\begin{aligned}
&((\wedge_{i=1}^m \delta_{A_i C_{i,sv_j}})|_{n_j=0})|_{p_1=x_1,\ldots,p_m=x_m} \\
&= \wedge_{i=1}^m (\delta_{A_i C_{i,sv_j}}|_{n_j=0,p_1=x_1,\ldots,p_m=x_m}) \\
&= \wedge_{i=1}^m (\delta_{A^{\mathcal{D}}_i C^{\mathcal{D}}_{i,sv_j.L}}|_{n_j.L=1,p_1.L=\overline{x_1},p_1.H=x_1,\ldots,p_m.L=\overline{x_m},p_m.H=x_m}) \qquad \text{(By equation 5.18)} \\
&= ((\wedge_{i=1}^m \delta_{A^{\mathcal{D}}_i C^{\mathcal{D}}_{i,sv_j.L}}|_{n_j.L=1})|_{p_1.L=\overline{x_1},p_1.H=x_1,\ldots,p_m.L=\overline{x_m},p_m.H=x_m}) \\
&= f_j.L(x)
\end{aligned}$$

$F_j.H(x)$ may be shown equal to $f_j.H(x)$ using an argument which is symmetrical to the one above for the low rail functions.

2. Extension

Let $x = x_1x_2...x_m$, be a Boolean vector of length m, $x_i \in \mathcal{B}, 1 \leq i \leq m$. Let $y = y_1y_2...y_m$ be a ternary vector of length m, $y_i \in \mathcal{T}, 1 \leq i \leq m$. Let $y_i.L$ and $y_i.H$ be the low, and high rail Boolean values for representing the ternary value y_i.

If $y \sqsubseteq_{\mathcal{T}} x$, then for $1 \leq j \leq m$, $x_j \Rightarrow y_j.H$, and $\overline{x_j} \Rightarrow y_j.L$. Using this information, with $n_j = 0$, and $n_j.L = 1$ in equations 5.16, and 5.17 one can readily show that

$$(\delta_{A_i C_{i,sv_j}}|_{n_j=0})|_{p_1=x_1,\ldots,p_m=x_m}$$
$$\Rightarrow$$
$$(\delta_{A^{\mathcal{D}}_i C^{\mathcal{D}}_{i,sv_j.L}}|_{n_j.L=1})|_{p_1.L=\overline{x_1},p_1.H=x_1,\ldots,p_m.L=\overline{x_m},p_m.H=x_m} \quad (5.19)$$

SRAM Size (bits)	Model Size (OBDD nodes)	Model Creation	
		Time (secs)	Mem. (MB)
4	142	1.7	0.95
8	346	1.9	1.02
16	818	2.4	1.15
32	1890	3.6	1.40
64	4290	5.8	1.78
128	9602	16.1	2.58
256	21250	39.3	3.91
512	46594	125.0	6.23
1024	101378	429.1	12.23

Table 5.1: Creation of a ternary model from SRAM assertions

Using $(f_1 \wedge g_1 \Rightarrow h_1) \wedge (f_2 \wedge g_2 \Rightarrow h_2) \Rightarrow (f_1 \wedge f_2 \Rightarrow g_1 \wedge g_2)$, and equation 5.19, one can show that $F_j.L(x) \Rightarrow f_j.L(y)$. That $F_j.H(x) \Rightarrow f_j.H(y)$ can be shown in a similar manner.

3. Monotonicity

The proof for monotonicity makes use of equation 5.17, $y_{2.i}.H \Rightarrow y_{1.i}.H$,and $y_{2.i}.L \Rightarrow y_{1.i}.L$ (since $y_1 \sqsubseteq_{\mathcal{T}} y_2$) to show the desired result. It is similar to the proof for extension.

□

5.3.5 Experimental Results

Table 5.1 shows the results for creation of a ternary simulation model by using our algorithm on the SRAM assertions in equations 5.2, 5.3, and 5.4. For each state variable, a pair of OBDDs representing the two rails of the excitation are created. Column 2 shows the number of OBDD nodes required to represent the low and high rail functions for all the state variables. Columns 3 and 4 show the time and the memory required to create the excitation functions. As the table shows, the model size and creation time are roughly linear with the SRAM size. One obvious optimization in the creation of the ternary model comes from the observation that generally there is no need for canonical representations of the excitation functions. By using representations like AND-OR directed acyclic graphs [18], one can possibly achieve savings in time and memory.

5.4 State node identification algorithm

From the discussion in the previous section we can see that our state node identification approach requires three key components:

- The circuit state machine.
- The specification state machine.
- An identification sequence.

For transistor-level networks, the circuit state machine is available from switch-level analysis of the circuit [19]. The construction of the specification machine is described in the previous section. The identification sequence is typically given by the user. Once we have these three components, both machines, started at the ternary reset state, are simulated with the identification sequence. After the identification sequence has been run, specification nodes in S_{ID}, and all the circuit nodes are hashed into bins based on their symbolic ternary values. The remainder of the algorithm simply consists of looking into each bin, and associating the specification node in the bin with the circuit node(s) in the bin. This gives us an algorithm which is linear in the number of specification and circuit nodes.

5.4.1 Special cases

The state node identification algorithm works well in most situations. Certain cases, however, require special modifications to the basic algorithm. These have been discussed below.

Polarity of value stored in storage element

If the value stored in the circuit storage element has the same polarity in the specification, then the above algorithm works. However, if the values are stored in the circuit with a negative polarity, then the algorithm must be modified to handle this. A change to the hashing function, so that it treats the low and the high rail symbolic values in an identical manner, takes care of this case.

Extraneous circuit nodes included as state nodes

While this is generally not encountered in memory arrays, it is possible in the case of latch and flip-flop based storage structures that nodes not typically considered storage nodes are identified as state nodes. Two enhancements to the basic algorithm handle these cases:

- Eliminate nodes without fanouts.

  Nodes without fanouts do not influence the circuit behavior. So, if such nodes have been identified as state nodes, they may be safely eliminated.

- Eliminate effects of fanouts from latches/flipflops.

  In some cases latches, or the master or slave components of flip-flops have fanouts into the adjacent circuit such that the values in the storage elements are propagated on to non-storage nodes. It is possible to trace these values to the originating logic. However, a simpler solution to this problem is to leave the appropriate clock signals inactive, and set all the other circuit inputs to X, after running the identification sequence on the circuit. By setting all nodes, other than those in latches, to X, such a step eliminates all the non-latch nodes as the possible candidates for state nodes.

5.4.2 Integrity of the verification process

The overall goal of the verification process is to verify that a given circuit implementation is a *realization* of the abstract assertions, i.e., establish a set containment relation between the IO behaviors of the two representations. Using an implementation mapping, the abstract assertions are mapped to STE assertions against which a circuit is verified. This, together with the non-triviality conditions of distinction and conformity establishes the desired set containment relation [6, Chapter 6].

Consider an implementation mapping for a circuit generated with the help of our state node identification procedure. Let us assume that this mapping satisfies the distinction and conformity conditions. The implications of this are that if we successfully verify a circuit against the set of STE assertions generated from a set of abstract assertions, and this implementation mapping, then we have proved the desired IO relationship between the abstract specification and the realization. If, however, the STE verification fails, then the error can be either in the circuit, or in the mapping, and further investigation is warranted. Given the simple timing in the implementation mappings in arrays, and that the state of each abstract state element is implemented by values on a unique set of circuit nodes, it is simple to establish the distinction and the conformity properties for the implementation mappings. So, in practice, if there is an error in the implementation mapping, then it will reveal itself during the verification process. Thus the overall verification process is conservative, and it will never pass incorrect circuits as correct.

5.5 Applications

We have applied our state node identification techniques based on the above to a number of examples [102, 103]. These include a multi-ported register file unit, data cache tags unit, and a CAM from recent PowerPC designs. Below, we first describe the state node identification process in a SRAM circuit. We follow it with a description of the identification process for the PowerPC arrays.

5.5.1 State node identification in a SRAM array

To construct the identifying sequence for a simple SRAM array, we reset the array and we do a write with symbolic values for addresses and data. Consider the 4-bit, 1 bit/word SRAM circuit shown in figure 5.8. Performing a write with the symbolic address $a1a0$, and the data value d means that location 0 will get updated to d when $a1 = 0$, and $a0 = 0$. Otherwise it remains at X. This is precisely the information conveyed by the ternary symbolic value, $(a0 + a1 + d, a0 + a1 + \overline{d})$, shown at $M[0]$ in

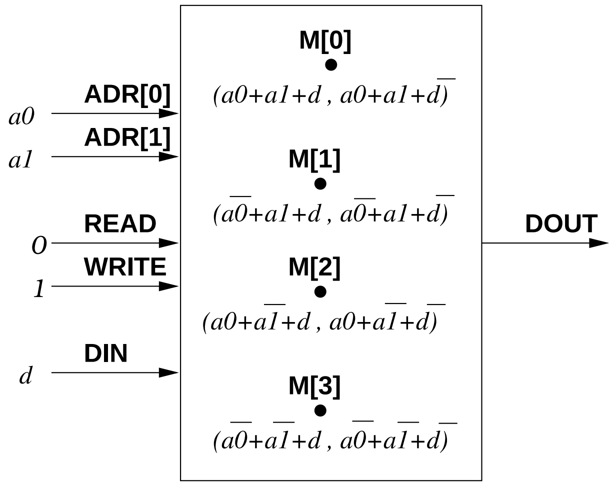

Figure 5.8: State node identification in a SRAM circuit

the figure. This symbolic ternary value evaluates to $(1, 1)$, i.e., X, for all assignments other than 00 to $a1a0$. For the assignment 00, the symbolic ternary value is the binary value $(d, \overline{d})$. In general, the symbolic value on the i^{th} location is $(a1a0 = i)?d : X$, which evaluates to d for the assignment $a1a0 = i$, and it is X for all other assignments. Such a symbolic input sequence distinguishes all the memory nodes in the array. We have made use of this sequence to identify the SRAM parts of many different memory arrays, as described below.

5.5.2 Application to PowerPC arrays

Multi-ported Register File

The register file contains two independently addressed banks of registers. Our work in [102] describes the identification of these nodes, where by doing a separate symbolic write to each of the two register banks, we identified their state nodes by using the technique for SRAM described above.

Data Cache Tags Unit

To verify the data cache tags (DTAG) unit, we identified the storage nodes corresponding to the tags, valid and modified bits for every way in every set, and the Least Recently Used (LRU) bits for every set. This has been described in [102]. The identification of the tags and the valid and modified bits was similar to the SRAM state node identification described in section 5.5.1 — these bits can be written to a location specified by the combination of a 7-bit index and two bits which specify the way in a set.

However, the SRAM identification technique does not work for the LRU state nodes, since these nodes can not be written to directly by using the regular DTAG operations. The LRU bits can be reset to all zeros or they can change as a result of a tag operation to reflect the past memory access pattern. We made use of this property to generate an identification sequence. All the LRU bits were first reset, and then we performed two DTAG operations such that a symbolic way w (encoded with Boolean variables w_1 and w_0) was accessed in the first operation, and symbolic way v (encoded with Boolean variables v_1 and v_0) was accessed in the second operation. This put unique symbolic Boolean values (Boolean functions of w_1, w_0, v_1 and v_0) on all the six LRU bits of a set. From our knowledge of how the LRU bits get updated when a DTAG operation occurs, we were able to identify all the LRU bits of all the sets.

Branch Target Address Cache

Similar to the DTAG circuit, the Branch Target Address Cache (BTAC) array does not allow direct writes to the state nodes we wish to identify. Only content-addressed writes are allowed in the array. Furthermore, we can't even reset these nodes. To overcome this problem, we constructed an identification sequence which exploits the testing circuitry in the system. We have described this in [103]. We scanned in a set of symbolic values corresponding to the outputs of a 6-to-64 decoder into the write register driver, and then performed a write with symbolic values at *wr_fadr* and *btac_data_in* (see chapter 6 for details). Since the write register drivers drive the write wordlines in the BTAC array, scanning in the symbolic values and doing a write is equivalent to having a decoder driving the wordlines. This put in ternary values on the circuit nodes identical to the ones in a SRAM circuit after a write with a symbolic address and data values.

5.6 Related work

Our work on state node identification is a special case of the much larger body of work on state-identification and fault-detection experiments on finite state machines (FSMs). We discuss some of the work in this area in section 5.6.1. However, while the approaches in the broader area of state-identification and fault-detection experiments strive for general solution to many important problems, they offer little by the way of practical algorithms. Our approach has been to work with a narrowed focus to develop workable solutions for state node identification in circuits. We discuss related work closer to our own in section 5.6.2.

5.6.1 Experiments on finite state machines

State-identification and fault-detection experiments are classical topics in the area of experimental analysis of finite state machine behavior [80]. Experiments on FSMs consist of a set of input sequences, and observing their output sequence. The machine itself is treated as a "black box", where the input and output terminals are available, but its internals are not accessible. The goal of state-identification experiments is to identify the unknown initial state of the machine (distinguishing experiments), and if that is not possible, to identify the final states of the machine (homing experiments) [80, pp. 449]. A fault-detection experiment is one which if performed on the machine, allows a decision to be reached on whether the machine is operating correctly.

Early work in this area includes that by Moore [94], and Kohavi [79], among others. A recent survey of this area by Yannkakis and Lee appears in [128].

Pixley describes work on testing sequential equivalence of two machines at the gate-level without a knowledge of their initial state [108, 109]. He presents OBDD based algorithms for testing equivalence, and experimental results on small machines. This approach, while powerful and general, does not scale up to the size of circuits we would like to work with. Furthermore, it is not applicable to switch-level design representations.

5.6.2 Identifying state nodes in circuits

A number of techniques have been proposed which appear to offer a solution to our state node identification problem. Such techniques fall into two classes. The first class of techniques work on extracting a FSM representation of a transistor-level or a gate-level netlist and in the process they identify the subset of circuit nodes which constitute the state nodes in the netlist [77], [101]. The work in [77] does not work for static storage structures, which is a serious shortcoming, for most of the circuits we would like to verify have such storage structures. The approach in [101] does not work directly on transistor netlists.

The second class of techniques attempt to identify the substructures of the circuit by examining the circuit topology. Once the circuit substructures are identified, one may identify the state nodes by examining the storage structures [97], [46]. While these approaches may work in a library-based design environment, they are not suitable for custom designs, which can contain a wide variety of customized storage cells.

The main weakness of these other techniques is their failure to exploit the dynamic properties of the state machine implemented at the transistor or the gate-level. We have used such information to develop fast, practical solutions to the state node identification problem.

Chapter 6

Case Studies - Verification of PowerPC arrays

An integral part of our thesis has been the application of the techniques we have developed to real industrial designs. We have used our techniques to verify the following memory arrays from recent PowerPC processors:

- Multi-ported register file (section 6.1).
- Data cache tags unit (section 6.2).
- Branch target address cache (section 6.3).
- Block address translation array (section 6.4).

For each of these arrays, we first describe the overall circuit organization, and the operations the array performs. Following this, we discuss the operations we verified, the time and memory required to do so. We describe bugs, if any, which we found during the course of the verification. The verification work was carried out at the joint IBM-Motorola Somerset PowerPC design center, located in Austin, Texas. The verification descriptions which follow have been reported in the 1996, and the 1997 Design Automation Conference Proceedings [102, 103].

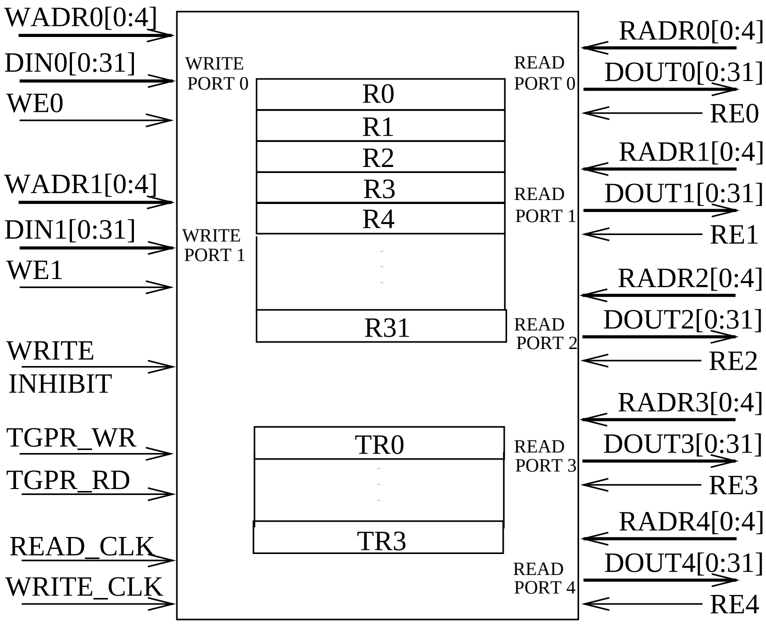

Figure 6.1: Multi-ported Register File Unit.

6.1 Multi-ported register file

Figure 6.1 shows a high-level view of the register file, which has 2 identical write ports and 5 identical read ports. When READ_CLK is high, the register file does a read operation and when WRITE_CLK is high, the register file does a write operation. The READ_CLK and the WRITE_CLK signals are mutually exclusive. The environment guarantees that the two write addresses are always different.

The register file contains 36 registers of 32 bits each, arranged in two banks, R0-R31 and TR0-TR3. During a write operation, when TGPR_WR is low, the writes go to one of R0-R31 as specified by the 5-bit address for each write port. When TGPR_WR is high, the writes go to one of TR0-TR3 based on the two least significant address bits. The environment is supposed to keep the middle address bit (bit 2) at 0, when the TGPRs are to be written. WRITE_INHIBIT when high prevents any writes from occurring. Also, each write port has a write enable signal (WE0, WE1).

The read ports also have read enable signals (RE0,..., RE4). When TGPR_RD is low, the five address bits select a register from the first bank. When TGPR_RD is high, the lowest two address bits select a register from the second bank, and bit 2 of the address must be low for the read to be successful. If a read does not occur on a port, or if bit 2 of the address is high when TGPR_RD is high, then the port's data output stays precharged high.

6.1.1 Register file

To verify the register file, we wrote six assertions. Five describe the read operation at each of the five read ports, and the sixth describes the register-file write operation. We start it with a description of the read operation, and follow it with that for the write operation.

Read operation

Since the five read ports are identical, their assertions are similar. Here we discuss the read operation at port 0 of the register file. The assertion for this operation is shown in lines 6.1 through 6.8 below. Lines 6.1 specifies that the initial state of the circuit is such that the i^{th} register in the first register bank contains the value u, and the n^{th} register in the second register bank contains the value v. The second line of the antecedent, 6.2, specifies that the read address for the port is the symbolic value j, and the control values at TGPRread, and ReadEn0 are trd, and en, respectively.

Lines 6.3 to 6.8 make up the consequent which specifies the state of the register file and its outputs as a result of performing the read operation. Lines 6.3, and6.4 state that after the read, the contents of the register file remain unchanged. Lines 6.5, and 6.6 check that, when the port is enabled, the correct data value appears at the output, and lines 6.7, and 6.8 check that when the port is not enabled, or if during a read from the second register bank bit 2 of the address is high, then the output bits are all high.

$$u = bitvector(32)$$
$$v = bitvector(32)$$
$$i = bitvector(5)$$
$$j = bitvector(5)$$

$n = bitvector(2)$
$trd = bitvector(1)$
$en = bitvector(1)$

$$(Op = Read) \wedge (R[i] = u) \wedge (TR[n] = v) \wedge \tag{6.1}$$
$$(ReadAdr0 = j) \wedge (TGPRread = trd) \wedge (ReadEn0 = en) \tag{6.2}$$
$$\stackrel{\text{LEADSTO}}{\Longrightarrow}$$
$$(R[i] = u) \wedge \tag{6.3}$$
$$(TR[n] = v) \wedge \tag{6.4}$$
$$(when(\neg trd \wedge en \wedge i = j)(Dout0 = u)) \wedge \tag{6.5}$$
$$(when(trd \wedge en \wedge n = j[4:3] \wedge \neg j[2])(Dout0 = v))) \wedge \tag{6.6}$$
$$(when(\neg en)(Dout0 = 0xFFFFFFFF)) \wedge \tag{6.7}$$
$$(when(en \wedge j[2] \wedge trd)(Dout0 = 0xFFFFFFFF)) \tag{6.8}$$

Write operation

The register file has two write ports which can update the registers in parallel. The assertion describing the write operation appears below. It shows a subset of the various possible combination of control signals for write.

$u = bitvector(32)$
$v = bitvector(32)$
$w = bitvector(32)$
$x = bitvector(32)$
$i = bitvector(5)$
$j = bitvector(5)$
$k = bitvector(5)$
$n = bitvector(2)$
$twr = bitvector(1)$

$$wen0 = bitvector(1)$$
$$wen1 = bitvector(1)$$
$$winhibit = bitvector(1)$$

$$(R[i] = u) \wedge (TR[n] = v) \wedge \tag{6.9}$$
$$(writePort0(j, x, wen0)) \wedge (writePort1(k, w, wen1)) \wedge \tag{6.10}$$
$$(TGPRwrite = twr) \wedge (writeInhibit = winhibit) \tag{6.11}$$
$$\overset{\text{LEADSTO}}{\Longrightarrow}$$
$$(when(writeNone)(R[i] = u)) \wedge \tag{6.12}$$
$$(when(writeNone)(TR[n] = v)) \wedge \tag{6.13}$$
$$(when(twr)(R[i] = u)) \wedge \tag{6.14}$$
$$(when(\neg twr)(TR[n] = v)) \wedge \tag{6.15}$$
$$\ldots$$
$$(when(i \neq j \wedge i \neq k \wedge write01\_gpr)(R[i] = u)) \wedge \tag{6.16}$$
$$(when(i = j \wedge i \neq k \wedge write01\_gpr)(R[i] = x)) \wedge \tag{6.17}$$
$$(when(i \neq j \wedge i = k \wedge write01\_gpr)(R[i] = w)) \tag{6.18}$$

$$writeNone = \neg(wen0 \vee wen1) \tag{6.19}$$
$$write01gpr = \neg winhibit \wedge wen0 \wedge wen1 \wedge \neg twr \tag{6.20}$$

Lines 6.9 through 6.11 contain the antecedent. Line 6.9 states that initially register R_i contains the symbolic value u and register TR_n contains the symbolic value v. A write operation is done at write port 0 with symbolic address j, symbolic data x and the write enable for the port set to symbolic value $wen0$ (line 6.10). Similarly, a write is done at write port 1, with address k, data w and write enable $wen1$. Finally, the symbolic value twr controls which of the two banks the writes go to and, $winhibit$ when true inhibits all writes.

To make the consequent more readable, we have used the abbreviations $writeNone$ and $write\_01gpr$. These have been described in terms of the symbolic variables in lines 6.19 and 6.20. $writeNone$ describes the condition that writes to both ports are disabled. $write01\_gpr$ describes the condition when writes through both ports are enabled, and the writes go to the first bank of the register file.

The consequent consists of lines 6.12 through 6.18. Lines 6.12 and 6.13 express the condition that when both write ports are disabled, all the registers remain unchanged. When writes are done to the second bank of registers, the first bank remains unchanged (line 6.14), and vice-versa (line 6.15). When register i contains the value u initially, and write addresses at both ports do not match i, then the value of register i remains unchanged (line 6.16). If i matches the address at the first write port ($i = j$), but not the second port ($i \neq k$), then register i gets updated to the data at the first port (line 6.17). Since, it is specified that write addresses at the two ports will not be the same, we do not check the results of write when the write addresses match, i.e., $i = j = k$.

6.1.2 Resource requirements

The verification of the register file takes a total of 267 seconds (on a IBM RS6000/580) for the complete set of assertions and generates a maximum of 8875 OBDD nodes. Voss used 31 MB of memory; 21 MB was used to represent the circuit nodes, and the excitation functions, and the rest was taken up by OBDDs and other run-time structures created by Voss's FL interpreter.

6.1.3 Bugs

In the process of verification, no bugs were found in the actual, register-file circuit. The designer did, however, test our methods by making two copies of the design, inserting a bug into one copy, and seeing if our tool could find it (it did). In addition, we translated and ran Voss on the RTL version of the register file, and found that it did not obey the specification. The "misbehavior", however, was in an underspecified area: when addressing a register in TR0-TR3, the specification states that the two most significant address bits were don't cares. However, the simulation model went into an error state if 1's were asserted on these lines, and this was detected by a failure of our assertion. The transistor netlist under the same conditions, completed the write (and the same assertion passed). This difference was detected, and showed the power of STE methods. It did not affect correct modeling of the register file inside

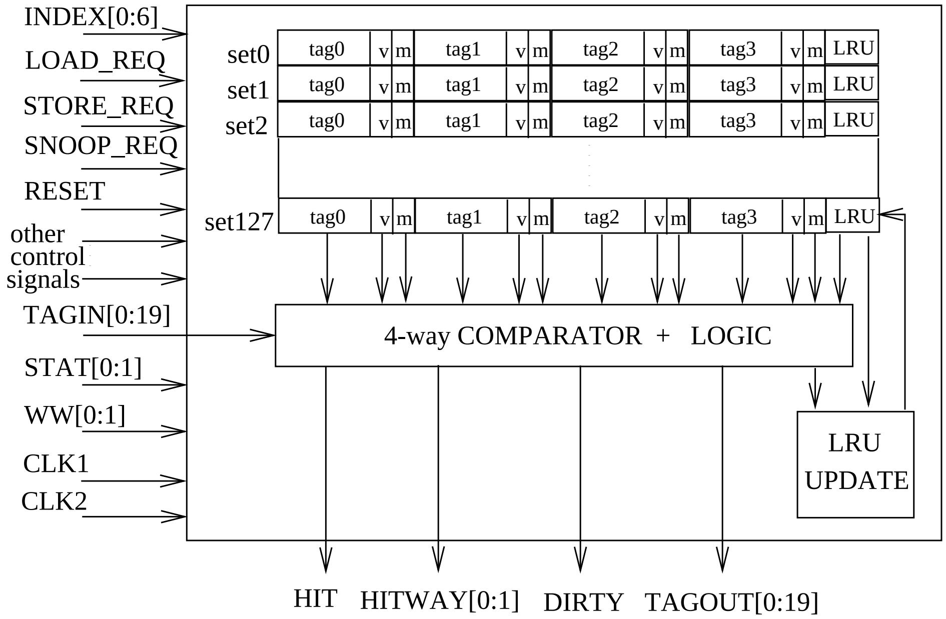

Figure 6.2: Data Cache Tags Unit

the larger chip, however, since the surrounding circuitry to the register file did, in fact, keep these bits low during writes to TR0-TR3.

6.2 Data cache tags unit

The data cache tags (DTAG) circuit, shown in figure 6.2, contains 128 4-way-associative sets. Each set contains 4 tags of 20 bits each, and each tag has one valid and one modified (dirty) bit. Also, each set contains 6 least-recently-used (LRU) bits which record the access history of its four ways.

In a typical operation, a 7-bit index at the INDEX input selects one of the 128 sets, and the 20-bit tag at TAGIN is compared in parallel with all four tags in the selected set. If a tag matches, then the HIT signal goes high and the LRU bits are updated to reflect that the matched way is most recently used. HITWAY indicates which of the four ways is hit. If none of the four tags match, the HIT signal remains low, and the least recent tag appears at TAGOUT (for cache replacement).

Other important operations are the reset and the tag write operations. In the reset operation, the RESET signal resets the DTAG unit by zeroing all valid, modified and LRU bits. In the tagwrite operation, the tag value at TAGIN and the valid and modified bit values at STAT are written into a way selected by WAYSEL of a set specified by INDEX.

6.2.1 Data cache tags operations

The data cache tags unit can do the following operations on any clock cycle – reset, load request, store request, snoop kill, snoop flush, tag refill, and status write. The assertions for some of these operations are described below.

Reset operation

The reset operation resets the tags unit by zeroing all the valid, modified and LRU bits. This can be succinctly expressed by the following assertion:

$$i = bitvector(7)$$
$$w = bitvector(2)$$

$$(op = reset) \stackrel{\text{LEADSTO}}{\Longrightarrow} (V[i][w] = 0) \wedge (M[i][w] = 0) \wedge (L[i] = 0x00)$$

Tag write operation

In this operation, tag bits and valid and modified (status) bits are written to a given way of a set. As a result of the write the LRU bits get updated to make the written way the most recent way. This operation can be specified by the assertion below.

$$t = bitvector(20)$$
$$i = bitvector(7)$$
$$wi = bitvector(7)$$
$$w = bitvector(2)$$
$$ww = bitvector(2)$$
$$m = bitvector(1)$$

$$v = bitvector(1)$$
$$l = bitvector(6)$$

$$\begin{aligned}
&(op = tagwrite) \wedge \\
&(T[i][w] = t) \wedge (V[i][w] = v) \wedge && (6.21)\\
&(M[i][w] = m) \wedge (L[i] = l) \wedge && (6.22)\\
&(writeindex = wi) \wedge (writeway = ww) \wedge && (6.23)\\
&(writetag = wtag) \wedge (writestatus = wstat) && (6.24)\\
&\overset{\text{LEADSTO}}{\Longrightarrow} \\
&(when(wi \neq i \vee ww \neq w) \\
&((T[i][w] = t) \wedge (V[i][w] = v) \wedge (M[i][w] = m))) \wedge && (6.25)\\
&(when(wi = i \wedge ww = w) \\
&((T[i][w] = wtag) \wedge (V[i][w] = wstat[0]) \wedge \\
&(M[i][w] = wstat[1]))) \wedge && (6.26)\\
&(when(wi \neq i)(L[i] = l)) \wedge && (6.27)\\
&(when(wi = i)(L[i] = update(l, ww))) && (6.28)
\end{aligned}$$

In the antecedent, lines 6.21 and 6.22 show the initial system state. They state that the tag value, valid bit and modified bit of the ith set and the wth way are t, v and m respectively. It also states that initially the LRU bits of the ith set is l. The next two lines show that tag value $wtag$ and the status bits $wstat$ are written to set wi and way ww.

As a result of the tag write, the tag, valid and modified bits of the addressed way get updated and all other ways remain unchanged. This is shown in lines 6.25 and 6.26. Line 6.28 shows that for an addressed set, the LRU bits get updated to reflect access to way ww, and they remain unchanged for a set that is not addressed (line 6.27).

The status write operation is very similar to the tag write operation — the only difference is that tags bits are not written during a status write.

Load request operation

For verifying the load request operation we wrote two assertions. The first assertion shows that if the initial machine state is
$(T[i][0] = t0) \wedge (T[i][1] = t1) \wedge (T[i][2] = t2) \wedge (T[i][3] = t3)\wedge$
$(V[i][0] = v0) \wedge (V[i][1] = v1) \wedge (V[i][2] = v2) \wedge (V[i][3] = v3)\wedge$
$(M[i][0] = m0) \wedge (M[i][1] = m1) \wedge (M[i][2] = m2)\wedge$
$(M[i][3] = m3) \wedge (LRU[i] = l)$,
and a load request is done with an index value of i and the tag input is $tagin$, then one of the following two things happen:

1. One of the four ways hit:
   For example, way 0 hits when $t0 = tagin$, and the valid bit for way 0, $v0$, is true. The LRU bits get updated to reflect that way 0 was most recently accessed, and all other state bits remain unchanged. The HIT output becomes true, HITWAY becomes 00 to reflect that way 0 has been hit, and at the dirty bit output, the value of $m0$, the dirty bit of way 0, is written out.

2. None of the ways hit: In this case all the state bits remain unchanged, and the dirty and the tag bits of the way to be replaced (least recently used way) are written out.

Certain combinations of state bit values are forbidden in this circuit. For instance, in a set no two tags can match, and it is assumed that the environment always maintains this state invariant by not writing in matching tag values Similarly, only certain combinations in the LRU bits are legal, and only these represent valid LRU information. All the DTAG actions above have been verified under these invariant conditions. A second assertion verifies that the tag, valid, modified and the LRU bits for a set that is not indexed remain unchanged in a load operation. Store request and snoop operations have not been described here, but they are very similar to the load request operation.

Resource requirements

The verification of the DTAG circuit takes about 10 minutes (on a IBM RS 6000/580) for the most complex of the assertions (e.g., the store request assertion) and generates

110,000 OBDD nodes. Voss used 150 MB of memory (of which 103 MB is to represent the circuit).

6.2.2 Bugs

Two actual bugs were discovered in the DTAG circuit. The first bug, a serious functional error, was known beforehand, but its nature was kept secret from the person running the STE verification. This bug was due to a transistor "sneak path" in the "hit detection" circuitry of the DTAG. This error was masked in regular verification process because of the assignment of directions to transistors. In addition, it is not clear if the appropriate digital vector would have been found to reveal it, had the directions not been applied. A single symbolic simulation vector, used during creation of the switch-level model, brought out this bug. This bug had already been fixed, in a later version of the circuit than the version upon which we were working.

The second bug was discovered when an assertion for what is called the status-write operation failed. Tracing the cause of the failure revealed that the LRU bits had not been updated, contrary to the specification. The LRU bits determine which line in a cache set will be replaced, in the event the set becomes full and a new line must be brought in. Faulty LRU bits merely cause discrepancies in performance (the line replaced may be the one most needed, for instance). This makes bugs in LRU bits difficult to find in digital simulation, unless one specifically monitors these bits on a cycle by cycle basis.

6.3 PowerPC Branch Target Address Cache Array

The Branch Target Address Cache (BTAC) array is part of the speculative instruction fetch mechanism on some PowerPC processors. The particular BTAC we verified, from a recent PowerPC processor, was a 64 entry content addressable memory, where each entry consists of a 30-bit tag and a 32-bit data part (Figure 6.3). The branch address is used to access the BTAC array, which contains the target address of previously executed branch instructions that are predicted to be taken.

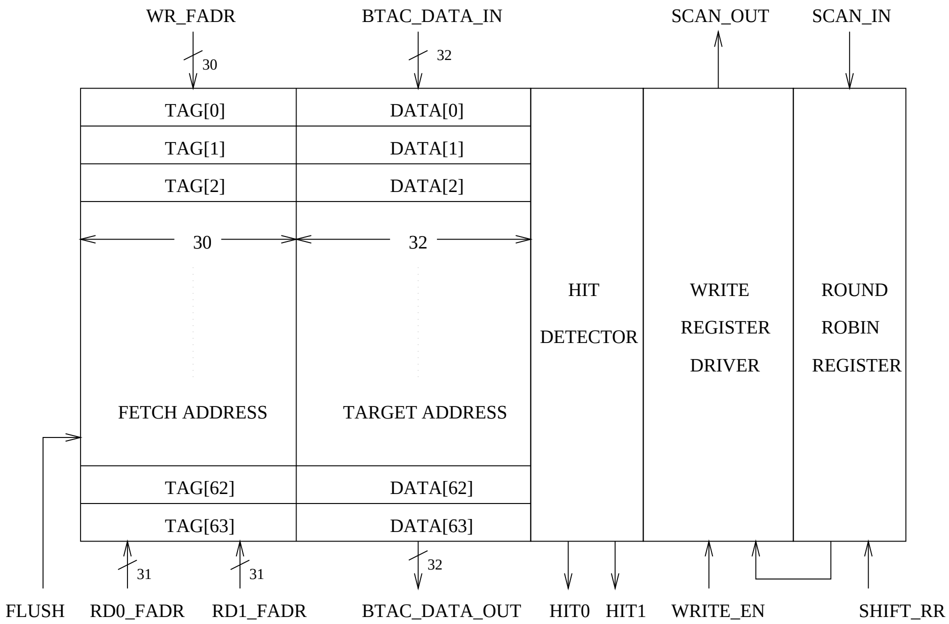

Figure 6.3: Branch Target Address Cache unit.

The primary task of this unit is, given a branch instruction address at the *rd0_fadr* input, to determine if there is a matching tag entry, and if so give out the corresponding data entry, which is the target address. The verification of this operation is in many ways similar to the CAM associative read operation verification shown in chapter 4. There are also a number of other operations this unit performs, including reset and initialize round-robin register, but for our discussion here we focus in on the most interesting one, namely the *replace* (or CAM write) operation.

6.3.1 BTAC Replace operation

In the replace operation, a TAG-DATA entry pair is updated with new values. The entry that is selected to be updated is not based on the address of the entry, rather it is based on the contents of the array, and a round-robin replacement policy. This operation is essentially a CAM write operation.

The first step in this operation is to select the entry to be replaced. For this an associative read is done with a tag value at read port 1, i.e., *rd1_fadr*. This input tag is compared with all the stored tags in parallel, and if there is a match, then *hit1* rises and the entry on which this match occurred is updated with new values appearing at *wr_fadr* and *btac_data_in* If there is no match, then a round-robin replacement policy is enforced. This replacement policy is implemented with a 64-bit round-robin register (right side of Figure 6.3) which is a one-hot encoded ring counter. The bit position in the ring counter which is 1 points to the BTAC entry to be replaced in case there is no hit on port 1. All entries which are not replaced remain unchanged.

In order to verify the replace operation, we verified a number of different cases, and many of these are similar to the memory write operation (e.g., when the compares are disabled on port 1). One of the more interesting cases is when write and compare are both enabled, but there is no hit on port 1. In this case the TAG-DATA entry pointed to by the round-robin register is written to and all other TAG-DATA entries are unchanged. This case is shown in the assertion below.

In the assertion below, we encode the TAG value unequal to the symbolic value *tag*. In order to do this, we use the CAM encoding with the help of the explicit version of the *ternneq* operator. The symbolic ternary vector generated by this operator represents the set of ternary vectors where for all $i, 0 \leq i \leq 29$, the *ith* bit position of a TAG[] entry equals $\overline{tag[i]}$, and all other bit positions of the TAG[] entry have an X. The problem with this is that if we have to show that TAG[] remains unchanged in some operation, then it is not sufficient to show that it still has its earlier value which is of the form <X,X,X,...,$\overline{tag[i]}$,...X>. The bit positions which are X can change, and we would not be able to detect it, since X simply denotes an absence of information. By using the explicit variant of the *ternneq* we can check for values at positions other than i. Essentially, in the assertion below, we have a vector of symbolic values called *val*, and with the help of this operator, encode a value of the form <val[0],val[1],...,val[i-1],$\overline{tag[i]}$,val[i+1],...,val[n-1]>. This value is unequal to *tag*, and at the same time we can verify that the value of TAG[] remains unchanged in an operation, since none of the bit positions contain X. Note that the *binneq* operator could also have been used in this case, but its use would have required new Boolean variables for each row of the BTAC array, which would be far in excess of what our current approach requires.

Lines 6.31 and 6.33 in the assertion below show our encoding for the 0th and the 63rd CAM entry. Line 6.34 states that the ith DATA entry is *data*, and line 6.35 states that the round-robin pointer points to the kth CAM-DATA entry. The consequent states that only the kth CAM-DATA entry gets changed (line 6.39), and all other entries remain unchanged (line 6.38). While the assertion below may look complex, it expresses an important system property, and it does so with a with a relatively small number of Boolean variables (406 Boolean variables), for a large system (over 4000 bits of state).

val = bitvector(30)
data = bitvector(32)
newadr = bitvector(30)
newdata = bitvector(32)
tag = bitvector(30)

i = bitvector(6)
j = bitvector(6)
k = bitvector(6)

$p0 = bitvector(5)$
$p1 = bitvector(5)$
...
$p63 = bitvector(5)$

$$(op = replace) \wedge (rd1\_fadr[0] = 1) \wedge (btac\_wr\_en = 1) \wedge (valid\_flush = 0) \wedge \quad (6.29)$$
$$(wr\_fadr = newadr) \wedge (btac\_data\_in = newdata) \wedge (rd1\_fadr = tag) \wedge \quad (6.30)$$
$$(TAG[0] = ternneqpos(tag, p0)) \wedge (when(j! = p0)(TAG[0][j] = val[j])) \wedge \quad (6.31)$$
$$... \wedge \quad (6.32)$$
$$(TAG[63] = ternneqpos(tag, p63)) \wedge (when(j! = p63)(TAG[63][j] = val[j])) \wedge \quad (6.33)$$
$$(DATA[i] = data) \wedge \quad (6.34)$$
$$(R[k] = 1) \wedge (when(l! = k)(R[l] = 0)) \quad (6.35)$$
$$\stackrel{\text{LEADSTO}}{\Longrightarrow} \quad (6.36)$$
$$(when(i! = k)(TAG[0] = ternneqpos(tag, p0)) \wedge ...$$
$$\wedge(TAG[63] = ternneqpos(tag, p63)) \wedge \quad (6.37)$$
$$(when(j! = p[0])(TAG[0][j] = val[j])) \wedge ...$$
$$\wedge(when(j! = p[63])(TAG[63][j] = val[j]))$$

$$\wedge(DATA[i] = data))) \tag{6.38}$$

$$\wedge(when(i = k)((TAG[i] = newadr) \wedge (DATA[i] = newdata))) \tag{6.39}$$

In the case where write and compare are enabled and there is a hit, the TAG-DATA entry on the line which matched the tag on $rd1\_fadr$ is changed, and all other TAG[] and DATA[] entries are unchanged. In order to verify that the tag entries which are unequal to the incoming tag remain unchanged, we have used an encoding technique similar to that in the assertion above.

6.3.2 Results

The most complex BTAC assertion takes 40MB of memory and 5 minutes to run, on a RS/6000 model 350 workstation. Of this 40MB, 24 MB is taken up by the OBDDs, and the remaining space is taken up by other run-time structures created during the verification process. The total run time for all assertions was 20 minutes. All the BTAC assertions passed, and no bugs were uncovered in this circuit.

As a comparison, if a more naive Boolean encoding were used for BTAC verification, the OBDD growth trends of Figures 4.4, 4.5 and 4.6 predict that a memory of several GB (and a 32-bit address space) would not have been sufficient for the verification!

6.4 PowerPC Block Address Translation array

The PowerPC architecture includes a block address translation (BAT) mechanism which maps ranges of effective addresses larger than a single page into contiguous areas of physical memory [112]. Such areas are used for data not subject to normal virtual memory handling, such as a memory-mapped display buffer, or an extremely large array of numerical data. This translation mechanism is implemented as an array consisting of software controlled special purpose registers. There are separate arrays, each consisting of eight registers, for the data side BAT (DBAT) and the instruction side BAT (IBAT). Here we describe the verification of the DBAT array.

The DBAT array is a CAM which contains 4 tag entries and 4 data entries. Each tag, data entry pair is organized as a pair of registers called the Upper DBAT Register

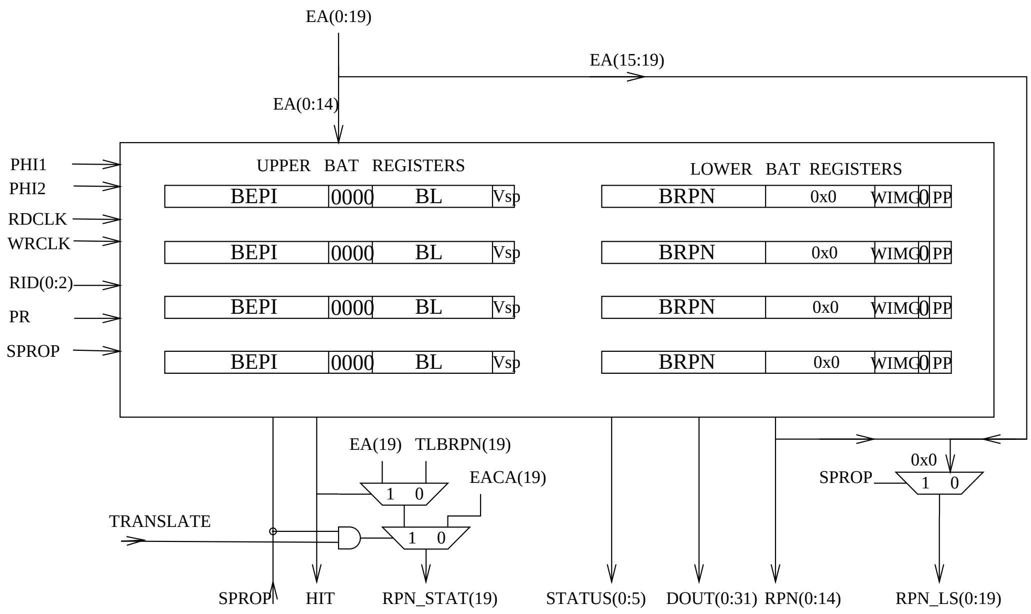

Figure 6.4: DBAT organization

and the Lower DBAT register (Figure 6.4). The two operations this array performs are the SPR operation, and the non-SPR operation. In the SPR operation, this array behaves like a register file where in a single clock cycle reads and writes are done on the Special Purpose Registers (SPRs) constituting the upper and lower DBAT registers.

In the non-SPR mode of operation, the DBAT array behaves like a CAM and it translates the 9 to 15 most significant bits of the logical address (bit 0 is the MSB) into the physical address as shown in Figures 6.4 and 6.5. The remaining bits pass unchanged. In Figure 6.4, the incoming logical address (top 15 bits, i.e., EA(0:14)) is compared to the block effective page index (BEPI) entry. The block length field (BL) contains a 11-bit mask information which decides which bits are to be compared. If the mask is all 0's, then all 15 bits are compared, and the corresponding 15-bit data entry, block real page number (BRPN) is sent out as the upper 15 bits of the physical address. If the mask entry is all 1's, then only the top most 4 bits get compared, and on a match only the top most 4 bits of the BRPN are put out as bits 0 to 3 of the

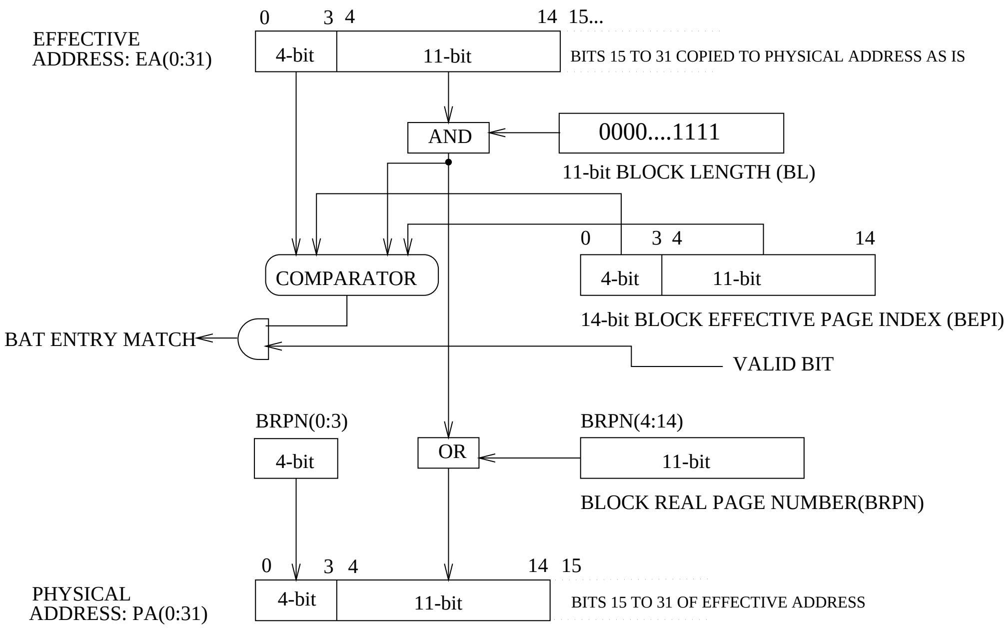

Figure 6.5: DBAT Address translation

physical address. Bits 4 to 14 of the physical address are directly copied from the logical address. The mask has a unary-style encoding. The 12 possible legal values for the mask for each tag-data entry are 00000000000, 00000000001, 00000000011, ..., 001111111111, 01111111111 and 11111111111. The lower 11 bits of the BEPI and BRPN entries should be zero corresponding to positions where the mask value is 1. Every register pair has a valid bit, Vsp, for the logical-physical address entry. This bit, when 0, indicates that the BEPI-BRPN-BL entry is invalid, and there can be no match on this entry. The translation process is shown in Figure 6.5. The system invariant is that at most one DBAT entry should match the incoming logical address. The PowerPC programming environment manual (pg. 7-25) states that it is a programming error for more than one entry to match the input [112]. We have not described the complete functionality of this complex unit here. Details can be found in [112]. While we have verified all the DBAT operations, here we have described the verification of the more interesting non-SPR mode of operation.

6.4.1 DBAT non-SPR operation

The *ternneq* operator does not work directly for expressing a mismatch on an upper DBAT register because comparison can be disabled on some selected register bits by the mask field. Furthermore, the bits masked out can be different for all the four upper DBAT registers. So, in order to express that a register contains a data value that does not match the incoming data, we need to take into account the mask bits and the legal values they can hold. To solve this problem, the *unary* generator is used in conjunction with the *terneqmask* operator.

$\vec{M} = unary(11)$ equals the symbolic vector representing all the 12 unary vectors of length 11 ranging from 00000000000, to 11111111111. $\vec{M}$ symbolically captures all the valid mask values. Let $\vec{u} = bitvector(15)$ be a Boolean variable vector of size 15. This vector cannot by itself represent the 15-bit Upper DBAT BEPI entry, as up to 11 least significant bits of the BEPI entry must be zero in positions the mask is one. Since the symbolic vector $\vec{M}$ represents the mask value, the symbolic vector $\vec{u}[0-3] \mid\mid \vec{u}[4-14]\&\neg\vec{M}$ represents all the valid values of BEPI. The BRPN entry can be described symbolically in a manner identical to BEPI.

If the valid bit for a register-pair is 0, then no match is possible. If the valid bit is 1 the comparison occurs with the incoming logical address. If an incoming tag, i.e., the 15 MSB address bits, is $\vec{tin}$, and the mask is 000000, then for the 0th register pair, a mismatch can be over any of the 15 bit positions in the BEPI entry T[0][0-14]. Similarly, if the mask is 000000000111, then comparison is done over bits 0 to 11. Consequently, the mismatch can be over these 12 bits. The *ternneqmask* operator with arguments $\vec{tin}$, and $\vec{M}$ captures all of these cases symbolically.

Since every register pair can have a different mask, we need a separate set of Boolean variables $\vec{m}$ for encoding the mask value for each pair. Also, for each register pair we need a distinct Boolean variable v to indicate whether their entry is valid Using this idea, verification of the associative read can be done in a manner similar to that of the CAM as described in chapter 4.

6.4.2 Results

We wrote two assertions for verifying the DBAT circuit, one for the SPR operation, and the other for the non-SPR operation. On a RS/6000 model 350 workstation, peak memory requirement for running all the assertions was 16.1 MB, and the total time was 15 minutes. We also wrote an assertion for the non-SPR operation using the plain encoding approach to compare against these results. This approach did not work well. Even with many control signals set to non-symbolic values, the memory required was over 100 MB!

6.4.3 Bugs

We discovered two bugs in the circuit we worked. These were that:

- In the SPR mode operation the signal, rpn_ls, should have been all 0's, and was not.
- Also in the SPR mode operation the signal, rpn_status19, was not correctly implemented, as expected from the circuit in Figure 6.4.

The significant part about discovering these bugs was that they were found after running just one assertion specifying the SPR operation. This is in contrast to running a huge number of simulation vectors, often running for days, without any certainty that such corner cases will be brought out.

6.5 Related work

While formal verification of embedded memory arrays is increasingly being recognized as an important issue for the verification of custom designs, there has been little activity in this area. Jones et al. at Rambus use symbolic switch-level simulation to validate memory array designs [75]. Their approach is to use a symbolic simulator in a manner similar to a conventional simulator, with the exception that certain control/data signal values are made symbolic. In contrast to our methodology, they do not partition the functionality of the unit into distinct operations. Rather, their approach is simply to observe the output sequence of the system, in response to an input sequence. Furthermore, for verification, they use a functional model of the memory core, rather than attempt to capture switch-level aspects of the memory core interaction [74].

Bryant discusses the verification of memory circuits with ternary simulation, and he shows that a N-bit random access memory can be verified by simulating $O(NlogN)$ patterns [23]. Of course, later work on symbolic trajectory evaluation [26, 117, 27] by Bryant and colleagues generalizes this early work. Recent work by Velev and Bryant [123] discusses a technique that allows symbolic simulation of systems with large embedded memory arrays, by replacing the arrays with a behavioral model. The memory state is represented by a list of symbolic Boolean expressions. This work, in conjunction with our work on verification of arrays at the transistor-level, has the promise of enabling hierarchical verification of systems with embedded memories.

Chapter 7

Conclusion

7.1 Summary

This thesis set out to solve the problem of verifying memory arrays which form critical components of microprocessors and many other hardware systems. Verification of these circuits has been a major weakness in the design and verification methodologies used in the industry today. Therefore, this thesis set as its goal to develop a body of techniques to verify the largest of arrays present on state of the art microprocessors.

We started with the observation that the ability of Symbolic Trajectory Evaluation (STE) to handle low-level circuit representations and detailed circuit timing made it an attractive starting point for verification of arrays. However, two fundamental problems prevented the use of STE on large arrays — the state explosion problem, and the switch-level analysis bottleneck. In chapter 3 we have developed techniques to exploit symmetry with STE to verify arrays. We show that exploiting symmetry extends by several orders of magnitude the size of designs that can be verified.

In chapter 4 we have shown the development and use of new Boolean encoding techniques to efficiently verify content addressable memories. Our encoding techniques scale up well in terms of space requirements, as compared to naive encodings. From our experimental results on verifying CAMs of different sizes, including two complex designs from microprocessors, we believe that we have solved the problem of verifying all the different types of CAMs found on a modern microprocessor. Chapter 5 develops an automated state node identification technique, which was extensively used for the verification of all the arrays we worked with.

An integral part of our thesis has been the application of the techniques we have developed to real industrial designs. Chapter 6 discusses the verification of several memory arrays from recent PowerPC processors — multi-ported register file, data cache tags unit, branch target address cache, and a block address translator array.

The touchstone of our work is if it has made a difference in the verification methodology for microprocessors. There are strong indications that at least one major microprocessor design center is adapting some of the techniques developed here as a part of its verification methodology.

7.2 Future work

7.2.1 Symmetry

An immediate extension of our work on exploiting symmetry with STE would be to develop it for the more general trajectory assertions described by Jain et al. in [72]. Most of the concepts we have developed, including symmetry properties, and their checks should be applicable in a straightforward manner. However the idea of waveform capture needs some modification, since assertions can be of the form of general graphs, rather than the "straight-line" assertions we have handled.

We have verified structural symmetries of systems by doing circuit graph isomorphism checks on essentially the entire design. While the time and space requirements for graph isomorphism checks scale up linearly with circuit size, the constant factors are large. Therefore, structural symmetry checks dominate the total verification space and time resource requirements. However, upon a little reflection, it is clear that performing structural analysis on a large flattened circuit graph is a brute force approach. In the typical design environment, hierarchical representations of the transistor netlists do exist. Checking symmetry on such a representation should be much more efficient than our approach. The details of this needs to be worked out, however.

Another direction for future work would be to extend our symmetry ideas to verify content addressable memories. In the longer run, it would be interesting to apply these ideas to verify hardware units other than memory arrays. Candidates for such an application include a processor datapath, where one can find the presence of structural symmetries because of bit-slice repetition, and data symmetries arising from the datapath operations.

The tools we use to detect and exploit symmetry do not offer much by the way of automation. We manually cut up the circuit into pieces, attach the appropriate ID circuits, reduce it to a quasi-canonical form or do symbolic simulation for symmetry checks. After the symmetry checks, we verify properties of the reduced design. While it may not be possible to offer complete automation, in that given a circuit, and an assertion, the symmetry checks and verification of the reduced circuit is are hidden from the user, significant tool building opportunities exist to make the process more appealing to the designer.

7.2.2 State node identification

Our work on the generation of a ternary simulation model from abstract assertions represents the excitation function by OBDDs. Generally, there is no need to use a canonical Boolean function representation for the excitation functions, and AND-OR directed acyclic graphs (DAGs) should work fine. Furthermore, we should be able to represent the excitation functions without using complemented versions of the dual-rail variables representing the current system state. It is important to investigate these issues so that more efficient memory models can be built. Our current models contain roughly as many OBDD nodes as there are nodes in the AND-OR DAGs of excitation functions generated by Anamos for switch-level circuits.

Our work on generation of simulation models from abstract assertions opens up the possibility of hierarchical verification, where after the verification of a circuit, it is replaced by its simulation model for verification of the system at the next level of hierarchy. Issues like interface timing, timing at two different levels of abstraction, and keeping the overall verification process conservative need to be addressed.

Another important issue is automating the generation of the identification sequence for circuits. Some open issues include if it is possible to do so efficiently for large circuits, and whether techniques to do so with ternary simulation exist.

7.2.3 Integrating array verification into conventional design flow

There are many issues that need to be looked into for seamlessly integrating our techniques into an overall processor design verification methodology. The first issue which stands out is the specification language. We use a declarative language which specifies system behavior with a set of Hoare like pre and postconditions. It concentrates on specifying the system behavior, and introduces little information about how the system is implemented [70]. However, most HDLs in use today, concentrate on describing how the system is implemented, rather than giving an abstract view of the system behavior. This operational approach to behavior specification is somewhat at odds with our methodology. However, it is unlikely that in the short term, the world will come around to our point of view. So, we should see how to best integrate our approach into a conventional design methodology. Techniques to partially automate the generation of abstract assertions and implementation mappings from HDLs should be studied. This can include annotating the HDLs with attributes like timing information for clocks and signals. One factor which makes this problem more tractable is that for arrays we can exploit common abstractions like indexing, bitvectors, and bitvector comparison operations.

Another important problem that should be looked into is that of performing FSM equivalence checking between the HDL and the switch-level implementation of arrays. Most equivalence checkers which work at the switch-level [82] do a static analysis of the transistor-level network to determine its logic functionality. To make the problem tractable for large circuits, they exploit design hierarchy to determine the equivalence of the subcells in the hierarchy, thereby determining the equivalence of the complete design. However, for arrays which typically use self timed components, and other complex forms of circuitry, static analysis does not work. Using a combination of static analysis, and symbolic simulation to abstract out the logic functionality of the design is a possible solution to the problem which should be looked into. FSM decomposition techniques should be investigated to make the problem of equivalence checking more tractable for large designs [99], [80, pp.385-437].

Bibliography

[1] S. Aggarwal, R. P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing and Verification*, volume 3, pages 91–34. IFIP, Elsevier Science Publishers B.V. (North Holland), 1983.

[2] *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, 1992.

[3] D. Bartelink. Processes of the future. *Solid State Technology*, 38(2):42–53, Feb. 1995.

[4] Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar, and G. M. Silberman. SLS — A fast switch-level simulator for verification and fault coverage analysis. In *23rd ACM/IEEE Design Automation Conference*, pages 164–170, June/July 1986.

[5] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger. Synchronous circuit verification by symbolic simulation: an illustration. In *Advanced Research in VLSI: Proceedings of the 6th MIT Conference*, pages 98–112. MIT Press, 1990.

[6] Derek L. Beatty. A methodology for formal hardware verification, with application to microprocessors. Technical Report CMU-CS-93-190, Department of Computer Science, Carnegie Mellon University, August 1993. PhD Thesis.

[7] Derek L. Beatty and Randal E. Bryant. Fast incremental circuit analysis using extracted hierarchy. In *25th ACM/IEEE Design Automation Conference*, pages 495–500, June 1988.

[8] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *31st ACM/IEEE Design Automation Conference*, June 1994.

[9] J. P. Billion and J. C. Madre. Original concepts of priam, an induatrial tool for efficient formal verification of combinational circuits. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*. North Holland, 1988.

[10] J.G. Bonar and S.P. Levitan. Real-time lisp using content addressable memory. In *Proceedings of 10th International Conference of Parallel Processing*, August 1981.

[11] Soumitra Bose and Allan L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In *Applied Formal Methods for VLSI*, pages 759–764, (Leuven, Belgium), 1989.

[12] Soumitra Bose and Allan L. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *Proceedings of the 1989 IEEE International Conference on Computer Design*, pages 217–221, Oct. 1989.

[13] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[14] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[15] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), August 1986.

[16] R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the Association for Computing Machinery*, 38(2):299–328, April 1991.

[17] Randal E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160–177, Feb. 1984.

[18] Randal E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):618–633, July 1987.

[19] Randal E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):634–649, July 1987.

[20] Randal E. Bryant. A survey of switch-level algorithms. *IEEE Design and Test*, pages 26–40, August 1987.

[21] Randal E. Bryant. Symbolic simulation—techniques and applications. In *27th ACM/IEEE Design Automation Conference*, pages 517–21, June 1990.

[22] Randal E. Bryant. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. In *IEEE International Conference on Computer-Aided Design*, pages 350–353, Nov. 1991.

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

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

[25] Randal E. Bryant, Derek L. Beatty, Karl Brace, Kyongsoon Cho, and Thomas Sheffler. Cosmos: a compiled simulator for mos circuits. In *24th ACM/IEEE Design Automation Conference*, pages 9–16, June 1987.

[26] Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, June 1991.

[27] Randal E. Bryant and Carl-Johan H. Seger. Formal verification of digital circuits using symbolic ternary system models. In Robert P. Kurshan, editor, *Computer Aided Verification*, pages 121–146, 1990.

[28] Janusz A. Brzozowski and Michael Yoeli. *Digital Networks*. Prentice-Hall Series in Automatic Computation. Prentice Hall, 1976.

[29] J. R. Burch, E. M. Clarke, D. E. Long, and D. L. Dill K. L. McMillan. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4), April 1994.

[30] J. R. Burch, E. M. Clarke, and K. McMillan. Symbolic model checking: 10sup20 states and beyond. In *The 5th annual IEEE Symposium on Logic in Computer science*, pages 428–439, 1990.

[31] J. R. Burch, E.M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 403–7, June 1991.

[32] J. R. Burch, E.M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, June 1990.

[33] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of 6th International Conference on Computer Aided Verification*, 1994.

[34] G. Cabodi, P. Camurati, F. Corno, P. Prinetto, and M. Sonza Reorda. Sequential circuit diagnosis based on formal verification techniques. In *Proceedings of the International Test Conference*, 1992.

[35] A. J. Camilleri. Simulation as an aid to verification using the HOL theorem prover. In *Design Methodologies for VLSI and Computer Architecture: Proceedings of IFIP TC10 Working Conference*, pages 148–168. North Holland, September 1988.

[36] Paul Chow, editor. *The MIPS-X RISC Microprocessor*. Kluwer Academic Publishers, 1989.

[37] E. M. Clarke, E. A. Emerson, and A. P. Sistla. automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM transactions on programming language and systems*, 8(2):244–263, 1986.

[38] Edmund M. Clarke, Robert Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9:77–104, 1996.

[39] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of 5th International Conference on Computer Aided Verification*, pages 450–462, 1993.

[40] O. Coudert, C. Berthet, and J.-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems, International Workshop Proceedings*, Grenoble, France, 1989.

[41] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, 1989.

[42] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean function vectors. In L. Claesen, editor, *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Amsterdam, 1990. North Holland.

[43] O. Coudert, J.-C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Computer Aided Verification'90*, pages 75–84, 1990.

[44] J. A. Darringer. The application of program verification techniques to hardware verification. In *Proceedings of the Design Automation Conference*, 1979.

[45] D.L.Dill, A.J.Drexler, A.J.Hu, and C.H.Yang. Protocol verification as a hardware design aid. In *Proceeding of the IEEE International Conference on Computer Design*, pages 522–525, 1992.

[46] C. Ebeling. GeminiII: A second generation layout validation program. In *Proceedings of the International Conference on Computer Aided Design*, 1992.

[47] C. Ebeling and O. Zazicek. Validating VLSI circuit layout by wirelist comparison. In *Proceedings of the International Conference on Computer Aided Design*, pages 172–173, 1982.

[48] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 169–180. ACM, 1982.

[49] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In *Proceedings of 5th International Conference on Computer Aided Verification*, pages 463–478, 1993.

[50] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

[51] Dundar Dumlugol et al. Local Relaxation Algorithms for Event-Driven Simulation of MOS Networks Including Assignable Delay Modeling. *IEEE Transactions on Computers*, C-30(3):193–202, July. 1983.

[52] R. H. Byrd et al. *Switch-Level Simulation: Models, Theory, and Algorithms*, volume 1 of *Computer-Aided Design of VLSI Circuits and Systems*. JAI Press, Greenwich, Connecticut, 1986.

[53] Stephen T. Flannagan, Perry H. Pelley, Norman Herr, Bruce E. Engles, Taisheng Feng, Scott G. Nogle, John W. Eagan, Robert J. Dunnigan, Lawrence J. Day, and Robert I. Kung. 8-ns CMOS 64k $\times$ 4 and 256k $\times$ 1 SRAMs. *IEEE Journal of Solid-State Circuits*, pages 1049–1054, October 1990.

[54] Caxton C. Foster. *Content Addressable Parallel Processors*. Van Nostrand Reinhold Company, New York, 1986.

[55] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Proceedings of Computer-Aided Verification*, pages 299–310, Berlin, Germany, June 1994. Springer-Verlag.

[56] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In MIlne and Subramanyam, editors, *Formal aspects of VLSI design*. Elsevier Scientific Publishers, 1986.

[57] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In Birtwistle and Subramanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1987.

[58] Aarti Gupta and Allan L. Fisher. Parametric circuit representation using inductive Boolean functions. In *Computer Aided Verification. 5th International Conference, CAV'93. Proceedings*, pages 15–28, June 1993.

[59] Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive Boolean functions. In *1993 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 192–9, Nov. 1993.

[60] L. Gwenapp. PPC 604 powers past Pentium. *Microprocessor Report*, pages 5–10, April 1994.

[61] L. Gwenapp. Integrated PA-7300LC powers HP midrange. *Microprocessor Report*, 9(15), November 1995.

[62] J. P. Hayes. An introduction to switch-level modeling. *IEEE Design and Test of Computers*, 4(4):18–25, Aug. 1987.

[63] Scott Hazelhurst and Carl-Johan H. Seger. A simple theorem prover based on symbolic trajectory evaluation and OBDDs. Technical Report 93-41, Department of Computer Science, University of British Columbia, 1993.

[64] Scott Hazelhurst and Carl-Johan H. Seger. Model Checking Partially Ordered State Spaces. Technical Report 95-18, Department of Computer Science, University of British Columbia, 1995.

[65] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(271-281), 1972.

[66] G. J. Holzmann and D. Peled. The state of spin. In *Computer Aided Verification. 8th International Conference, CAV'96. Proceedings*, pages 385–9, July 1996.

[67] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Reachability trees for high-level Petri nets. *Theoretical Computer Science (Netherlands)*, 45(3):94–102, 1986.

[68] W. A. Hunt. *FM8501: A Verified Microprocessor.* Lecture Notes in Artificial Intelligence. Springer Verlag, 1994.

[69] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[70] A. Jain. A case for hardware specification languages. Thesis Proposal, January 1994.

[71] Alok Jain. *Hardware Specification Language (HSL) UNIX man page.* Carnegie Mellon University, ECE Department, 1994.

[72] Alok Jain, Kyle Nelson, and Randal E. Bryant. Verifying nondeterministic implementations of deterministic systems. In *Proceedings of Formal Methods in Computer-Aided Design*, pages 109–125, Nov 1996.

[73] Kurt Jensen. Condensed state spaces for symmetrical colored petri nets. *Formal Methods in System Design*, 9:7–40, 1996.

[74] K. D. Jones. Personal communication, August 1994.

[75] K. D. Jones. Verification with the cosmos symbolic simulator. Talk at the 6th Computer Aided Verification Conference, Stanford, CA, June 1994.

[76] J. Joyce. Formal verification and implementation of a microprocessor. In Birtwistle and Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157. Kluwer Academic Publishers, Jan 1987.

[77] T. Kam and P.A.Subrahmanyam. State machine abstraction from circuit layouts using bdd's: Applications in verification and synthesis. In *Proceedings of the European Design and Test Conference*, 1992.

[78] K. Kodandapani and E. McGrath. A wirelist compare program for verifying VLSI layouts. *IEEE Design and Test of Computers*, pages 46–51, June 1986.

[79] I. Kohavi and Z. Kohavi. Variable-length distinguishing seqeunces and their application to the design of fault-detection experiments. *IEEE Transaction on computers*, C-17:792–795, August 1968.

[80] Z. Kohavi. *Switching and Finite automata theory.* McGraw-Hill, 1970.

[81] T. Kohonen. *Content Addressable Memories.* Springer Series in Information Sciences. Springer-Verlag, 1980.

[82] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin. Verity — A formal verification program for custom CMOS circuits. *IBM Journal of Research and Development*, 39(1/2), January/March 1995.

[83] C.Y. Lee and R.Y. Yang. High-throughput data compressor designs using content addressable memory. *IEE Proceedings- Circuits, Devices and Systems (UK)*, 142(1), February 1995.

[84] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM, 1985.

[85] L.Y. Liu, J.F. Wang, R.J. Wang, and J.Y. Lee. Design and hardware architectures for dynamic huffman coding. *IEE Proceedings- Computers and Digital techniques (UK)*, 142(6), November 1995.

[86] D. E. Long. Model checking and abstraction. *Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1992.

[87] Wolfgang Ludwig and Claus Falter. *Symmetries in physics: group theory applied to physical problems.* Springer-Verlag, Berlin; New York, 1988.

[88] J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988.

[89] C. H. Malley and M. Dieudonne. Logic verification methodology for PowerPC microprocessors. In *32nd ACM/IEEE Design Automation Conference*, June 1995.

[90] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, editors, *Correctness Problem in Computer Science*, pages 215–273, London, 1982. Academic Press.

[91] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 141–154. ACM, 1983.

[92] K. L. McMillan. Symbolic model checking: An approach to the state exploration problem. Technical Report CMU-CS-92-131, Carnegie Mellon University, School of Computer Science, May 1992.

[93] T.F. Melham. Abstraction mechanisms for hardware verification. In Birtwistle and Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 269–291. Kluwer Academic Publishers, Jan 1987.

[94] E. F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies, Annals of Mathematical Studies*, pages 129–153. Princeton University Press, 1956.

[95] Kyle L. Nelson, Alok Jain, and Randal E. Bryant. Formal verification of a superscalar execution unit. In *34th ACM/IEEE Design Automation Conference*, June 1997.

[96] T. Ogura, M. Nakanishi, T. Baba, and Y. Nakabayashiand R. Kasai. A 336-kbit content addressable memory for highly parallel image processing. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 273–276, May 1996.

[97] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather. Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.

[98] S. Owre, J. R. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using pvs for hardware verification. In *Theorem Provers in Circuit Design. Theory, Practice and Experience. Second International Conference, TPCD '94 Proceedings*, pages 258–279. Springer Verlag, Sept. 1994.

[99] M. Pandey. Verification of arrays. Thesis Proposal, April 1995.

[100] M. Pandey and R. E. Bryant. Exploiting symmetry in when verifying transistor-level circuits using symbolic trajectory evaluation. In *(to appear in) Proceedings of 9th International Conference on Computer Aided Verification*, 1997.

[101] M. Pandey, G. York, D. Beatty, A. Jain, S. Jain, and R.E.Bryant. Extraction of finite state machines from transistor netlists. In *Proceedings of the International Conference on Computer Design*, 1995.

[102] Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant. Formal verification of PowerPC(TM) arrays using symbolic trajectory evaluation. In *33rd ACM/IEEE Design Automation Conference*, pages 649–654, June 1996.

[103] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *34th ACM/IEEE Design Automation Conference*, June 1997.

[104] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[105] Doron Peled. All from one, one from all: on model checking using representatives. In *Proceedings of 5th International Conference on Computer Aided Verification*, pages 409–423, 1993.

[106] Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Order Methods in Verification*. American Mathematical Society, DIMACS, Princeton, NJ, 1996.

[107] P.H.Starke. Reachability analysis of Petri nets using Symmetries. *Systems Analysis - Modeling - Simulation (Germany)*, 8(4-5):293–303, 1991.

[108] C. Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In E.M. Clarke and R.P. Kurshan, editors, *Computer-Aided Verification. 2nd International Conference, CAV '90 Proceedings*, pages 54–64. Springer-Verlag, June 1990.

[109] C. Pixley. A theory and implementation of sequential hardware equivalence. *IEEE Transactions on Computer-Aided Design*, 11(12), December 1992.

[110] A. Pnueli. The temporal semantics of concurrent programs. In *The 18th symposium on Foundations of Computer Science*, 1977.

[111] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Lecture Notes in Computer Science*, volume 224, pages 510–584. Springer-Verlag, 1986.

[112] *PowerPC™ Microprocessor Family: The Programming Environments*. Motorola Inc., 1994.

[113] *PowerPC 604 RISC Microprocessor User's Manual, MPC604UM/AD Rev1.* Motorola Inc., 1994.

[114] H Ruess, N. Shankar, and M. K. Srivas. Modular verification of srt division. In *Computer Aided Verification. 8th International Conference, CAV'96. Proceedings*, pages 123–134, July 1996.

[115] M. Sato, K. Kubota, and T. Ohtsuki. A hardware implementation of gridless routing based on content addressable memory. In *27th ACM/IEEE Design Automation Conference*, pages 646–649, June 1990.

[116] Carl-Johan H. Seger. Voss—a formal hardware verification system: User's guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, 1993.

[117] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, 1995.

[118] H. P. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the Pentium processor(1994). Technical report, Intel, Nov. 30 1994. Intel Technical Report.

[119] M. Slater. Microunity lifts veil on MediaProcessor. *Microprocessor Report*, 9(14), Oct. 13 1995.

[120] M. Srivas and S.P.Miller. Formal verification of an avionics processor. Technical Report CSL-95-04, SRI International Computer Science Laboratory, 1995.

[121] M.K. Srivas and S. P. Miller. Applying formal verification to the aamp5 microprocessor: a case study in the industry use of formal methods. *Formal Methods in System Design*, 8(2):153–188, March 1996.

[122] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, 1989.

[123] M. Velev, R. E. Bryant, and A. Jain. Exploiting modeling of memory arrays in symbolic simulation. In *(to appear in) Proceedings of 9th International Conference on Computer Aided Verification*, 1997.

[124] D. Weise. Multilevel verification of MOS circuits. *IEEE Transactions on Computer-Aided Design*, 9(4):341–351, April 1990.

[125] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI design, A systems Perspective*. Addison-Wesley, 2nd edition, 1994.

[126] H. Yamada, M. Hirata, H. Nagai, and K. Takahashi. A high-speed string-search engine. *IEEE Journal of Solid-State Circuits*, SC-22(10):829–834, October 1987.

[127] H. Yamada, M. Hirata, H. Nagai, and K. Takahashi. A 288-kb fully parallel content addressable memory using a stacked-capacitor cell structure. *IEEE Journal of Solid-State Circuits*, 27(12), December 1992.

[128] M. Yannakakis and D. Lee. Principles and methods of testing finite state machines—-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.