On the Formal Verification of ATM Switches

Jianping Lu

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Sceience at

Concordia University

Montréal, Québec, Canada

1999

ABSTRACT

On the Formal Verification of ATM Switches

Jianping Lu

Because of the difficulty of adequately simulating large digital designs, there has been a surge of interest in formal verification, in which a mathematical model of the design is proved to satisfy a precise specification. *Model Checking* and *Equivalence Checking*, which have the advantage of automatic verification, are two main formal verification techniques that people are working on. The main problem of model checking and sequential equivalence checking is the state space explosion. Another drawback of model checking is lack of methods on establishing an environment and expressing a property. In this thesis, we propose *Property Division* techniques which avoid the state space explosion problem by deducing a property from several sub-properties. A number of methods on establishing an environment and expressing a property are illustrated.

Although ATM hardware is hard to design due to its high speed and various features, the applications of model checking and equivalence checking on ATM hardware verification are few. In this thesis, *Fairisle ATM switch fabric*, *Fairisle ATM null port controller*, *Input FIFO of RCMP-800* and *Concentrator of Knockout ATM switch* are developed. With the techniques we propose, all these ATM hardware designs are formally verified in the formal verification tools called *Verification Interacting with Synthesis* (VIS).

ACKOWLEDGEMENTS

I have been very fortunate to have Dr. Tahar as my advisor through out my stay at Concordia University. Dr. Tahar devotes considerable time and energy to his students, and many of my idea about verification and about what constitutes interesting research arose from talks with him. His insistence on motivation, discussion, and examples has hopefully turned this thesis from a wasteland of formality into something readable. I was fortunate to be a student of Dr. Al-khalili when he was the instructor of the course “digital design system”. His excellent teaching on digital designs helped me to build up the ATM models presented in the thesis. Special thanks are due to Dr. Mehmet-Ali and Dr. Bouguerba for their constructive criticism and comments through the course of this work.

I also wish to acknowledge all the teachers and individuals who have contributed to my knowledge during the period of my stay at Concordia.

Finally, my wife gave me her constant love and encouragement, I can never thank her enough.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

With the increasing reliance of digital systems, design errors can cause serious failures, resulting in the loss of time, money, and long design cycle. Large amounts of effort are required to correct the error, especially when the error is discovered late in the design process. For these reasons, we need approaches that enable us to discover errors and validate designs as early as possible. Conventionally, simulation has been the main debugging technique. However, due to the increasing complexity of digital systems, it is becoming impossible to simulate large designs adequately. Therefore, there has been a recent surge of interest in formal verification. In formal verification, a mathematical model of the design is compared with a formal specification describing the correctness criteria for the design. The verification is exhaustive: all possible behaviors of the model are considered [2].

Most formal verification methods fall into one of two classes: theorem proving based methods and decision graph based methods.

1.1 Theorem proving based methods

In theorem proving based methods, the designer constructs a mathematics proof, perhaps with the aid of some automated support, to prove that the model meets its specification. Because the full power of mathematics is available, such techniques are very flexible and powerful. It is possible to model systems at almost any level of detail, and to prove properties of entire classes of systems.

The theorem-proving methods have been around for over 35 years, and definitely have their staunch adherents. They have been extensively in government pilot projects, notably a popular theorem proving tool PVS [36] was used in NASA. HOL (Higher Order Logic) [19] which is another famous theorem proving tool was used in many research projects. Although there are many theorem proving tools in the world, they all prove the equivalence or implication between a specification and an implementation.

In spite of impressive demonstrations in the hardware domain and elsewhere, the theorem-proving methods have never achieved the broad level of acceptance for which their advocates had hoped. The reason undoubtedly lies in the need for expert users, and an application cycle which evolves generally slower than a normal product design cycle, so even just keeping up with the project development schedule is a problem.

1.2 Decision graph based methods

Decision graph based methods restrict the model to be finite-state and use state space search algorithms to automatically check if the specification is satisfied. Further, if the verification fails, then a counterexample trace can be produced to show the user why this is the case. The particular types of decision graph based methods that we will be considering are called *model checking* and *equivalence checking.*

- **Model Checking**

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in a propositional temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine automatically if the specifications are satisfied by the state-transition graph. The technique was originally developed in 1981 by Clarke and Emerson [12]. Quielle and Sifakis [29] independently discovered a similar verification technique shortly after thereafter. An alternative approach

based on showing inclusion between automata was later devised by Kurshan [20] at AT&T Bell Laboratory .

Model checking has several important advantages over mechanical theorem proving. The most important is that the procedure is highly automatic. Typically the user provides a high level representation of the model and the specification to be checked. The model checker will either terminate with the answer *true* indicating that the model satisfies the specification or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex reactive systems.

The first model checkers were able to find subtle errors in small circuits and protocols. However they were unable to handle very large examples due to *the state explosion problem*. The problem arises in systems composed of multiple state holding elements operating in parallel: the total number of states in the system generally grows exponentially with the number of state holding elements. Because of the limitation, many researchers in formal verification predicted that model checking would never be useful in practice.

The possibility of verifying systems with realistic complexity changed dramatically in the late 1980's with the discovery of how to represent transition relation using *ordered binary decision diagram* (OBDD) [6]. The original model checking algorithm, together with the new representation for transition relations, is called *symbolic model checking* [26]. By using the combination, it is possible to verify large reactive systems. In fact, some examples with more than 10^{120} states have been verified [9]. This is possible because the number of nodes in the OBDDs that must be constructed no longer depends on the actual number of states or the size of the transition relation. Because of this breakthrough it is now possible to verify reactive systems with realistic complexity and a number of major companies including Intel, Motorola, Fujitsu, AT&T and Nortel have started using symbolic model checkers to verify actual circuits and protocols. In several cases errors have been found that were missed by extensive simulation.

While symbolic representations have greatly increased the size of the system that can be verified, most realistic systems are still too large to be handled. Thus, it is important to find techniques that can be used in conjunction with the symbolic methods to extend the size of the systems that can be verified. Current well-known techniques are compositional reasoning and abstraction [23].

- **Equivalence Checking**

Equivalence checking is used to prove functional equivalence of two design representations modeled at different levels of abstraction [34]. Equivalence checking can be divided into two categories: one is combinational equivalence checking, and the other is sequential equivalence checking.

The main approaches to combinational equivalence checking are based on canonical representations of Boolean functions, typically binary decision diagram (BDDs) or their derivatives. The functions of the two circuits to be compared are converted into canonical forms [6] which are then structurally compared. The major advantage of BDDs is their efficiency for a wide variety of practically relevant combinational circuits. If the BDD size does not grow too large, this type of Boolean reasoning is fast and independent of the actual circuit structure. Moreover, if structural similarities of the two designs are exploited, BDDs can effectively find implications between nets even if they are farther away from the primary inputs.

Some commercial equivalence checking tools have been used in industries. For example, Chrysalis's equivalence checking tool Design Verifyer is being used in many IC design companies. Equivalence checking tools are often used to verify the equivalence between RTL and synthesized gate-level design. Also they are used to ensure the correctness of manual optimization during the fabrication process.

However, since current designs are mainly clock-driven synchronized, to perform the combinational equivalence checking between two different sequential models, we have

to divide a design into pieces, and map each register (or flip-flop) of one model into another, and compare their combinational circuits between every two consecutive registers. This will lead to a drawback: combinational equivalence checking cannot handle the equivalence checking between RTL and behavioral model because RTL model and behavioral model are developed separately and should have the same outputs at some certain clock cycles, but it is impossible to map each register in RTL model to that of behavioral one.

Sequential equivalence checking is to verify the equivalence between two sequential designs at each valid state. It is done by building the product finite state machine, and checking whether a state where the values of two corresponding outputs differ, can be reached from the set of initial states of the product machine. In other words, sequential equivalence checking only considers the behavior of two designs while ignoring their implementation detail such as latch mapping. Therefore, sequential equivalence is able to verify the equivalence between RTL and behavioral model. According to this, sequential equivalence checking is very useful in design verification, but its drawback is that it cannot handle a large design due to state space explosion problem.

1.3 Verification Interacting with Synthesis

Today, there are a lot of academic and commercial formal verification tools. One can purchase verification tools from Abstract Hardware Ltd. (CheckOff - core technology developed at Siemens), Chrysalis (Design Verifyer), Compass (Vformal-core technology developed at BULL), IBM (RuleBase-core technology developed at CMU), and Cadence (FormalCheck - core technology developed at Lecent). In addition to these, some universities develop their own academic verification tools. CMU has SMV [26], which is based on symbolic model checking. University of Montreal developed *Multiway Decision Graph* (MDG) [13], which supports property checking and equivalence checking. The most popular academic formal verification tool for both model checking and

equivalence checking is Verification Interacting with Synthesis (VIS) [5], which is developed by University of California, Berkeley.

VIS integrates the verification, simulation, and synthesis of finite-state hardware systems. It uses a Verilog front-end and supports model checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis, etc. Because of these practical features, we choose VIS as the verification tool in this thesis. In the following, we give a brief description of VIS.

- **Verilog front-end**

VIS operates on an intermediate format called BLIF-MV, which is an extension of BLIF, the intermediate format for logic synthesis accepted by SIS [33]. VIS includes a stand-alone compiler from Verilog to BLIF-MV, called VL2MV, which supports a synthesizable subset of Verilog. VL2MV extracts a set of interacting finite state machines that preserves the behavior of the source Verilog program defined in terms of simulated results. Two new features have been added to Verilog [35]:

1) *Nondeterminism*. A nondeterministic construct, $ND, has been added to specify nondeterminism on wire variables; this is the only legal way to introduce nondeterminism in VIS.
2) *Symbolic variables*. Sometimes it is desirable to specify and examine the value of variables symbolically, rather than having to explicitly encode them. VL2MV extends Verilog to allow symbolic variables using an enumerated type mechanism similar to the one available in the C programming language.

- **Hierarchy and initialization**

When a BLIF-MV description is read into VIS, it is stored hierarchically as a tree of modules, which in turn consist of sub-modules. This hierarchy can be traversed in a manner similar to traversing directories in UNIX. Simulation and verification operations can be performed at any subtree of the hierarchy. It is possible to replace the subhierarchy

rooted at the current node with a new hierarchy specified by a new BLIF-MV file, which might be a synthesized module or a manually abstracted module. VIS can also output the hierarchy below the current node to a BLIF-MV file.

- **Interaction with synthesis**

VIS can interact with SIS to optimize the existing logic by reading and writing the BLIF format, which SIS recognizes. Synthesis can be performed on any node of the hierarchy.

- **Symbolic model checking**

VIS performs symbolic model checking under Büchi fairness constraints [4] which assumes that a system will fairly go to each possible transition state and cannot miss any possible states forever. VIS reports the failure with a counterexample, (i.e., behavior seen in the system that does not satisfy the property). This is called the "debug" trace. Debug traces list a set of states that are on a path to a fair cycle and fail the CTL formula.

- **Equivalence checking**

VIS provides the capability to check the combinational equivalence of two designs. An important usage of combinational equivalence is to provide a sanity check when re-synthesizing portions of a network. VIS also provides the capability to test the sequential equivalence of two designs. Sequential verification is done by building the product finite state machine, and checking whether a state where the values of two corresponding outputs differ, can be reached from the set of initial states of the product machine. If this happens, a debug trace is provided. Both combinational and sequential equivalence verification are implemented using BDD-based routines.

- **Simulation**

VIS also provides traditional design verification in the form of a cycle-based simulator that uses BDD techniques. Since VIS performs both formal verification and simulation using the same data structures, consistency between them is ensured. VIS can

generate random input patterns or accept user-specified input patterns. Any subtree of the specified hierarchy may be simulated.

- **Algorithms**

The fundamental data structure for these algorithms is a multi-level network of latches and combinational gates that is created by flattening the hierarchy. It is assumed that there are no combinational cycles in the network.

The primary inputs and latch outputs are referred to as combinational inputs and the primary outputs and latch inputs are referred to as combinational outputs. The variables of a network are multi-valued, and logic functions over these variables are represented by multi-valued decision diagrams (MDDs) which are an extension of BDDs.

The combinational input variables and next state variables must be ordered before MDDs can be constructed. The combinational input variables are ordered by doing a depth-first traversal of the logic that generates the combinational outputs. The order in which the output logic cones are visited is determined using the algorithm of Aziz et al. [1]. This algorithm orders the latches to decrease a communication complexity bound (where backward edges are more expensive than forward edges) on the latch communication graph. The traversal of an output logic cone is done in such a way that the combinational inputs farthest from the outputs appear earlier in the ordering. Finally, each next state variable is inserted into the variable ordering immediately after the corresponding present state variable.

A good partial or total ordering on the variables can be read in to improve the performance. In addition, dynamic variable ordering is supported. Generally, a good initial ordering followed by one or two forced dynamic reorderings gives good results.

1.4 Formal Verification and Design Flow

Formal verification can be automatically used in an top-down design phase. Figure Chapter 1 .1 shows an example of employing formal verification in a digital design phase.

In this example, an RTL design is first described in Verilog [35] hardware design language, and then it is tested by a Verilog simulation tool (e.g. Cadence Verilog-XL). After that, model checking is further applied to verify the RTL design. Since model checking, which concentrates on logic relations, has a totally different mechanism from simulation, it could detect design errors which were not caught by simulation. The verified RTL design is synthesized into the gate-level netlist (referred to as Synopsys-Verilog in Figure Chapter 1 .1) in Synopsys Design Compiler which is a synthesis tool. But the format of the structural description cannot be used directly in VIS or XL-simulator, and also the primitive modules of Synopsys are different from those of VIS and XL-simulator. A program in AWK [1] automatically translates the Synopsys-Verilog format to a Verilog format that is accepted by VIS and XL-simulator (referred to as VIS/XL-Verilog in Figure Chapter 1 .1) and a file including the primitive modules of Synopsys is created so that VIS is interacted with Synopsys and XL simulator directly. In addition to simulation and model checking, equivalence checking between the RTL description and the synthesized netlist description of each submodule is employed. Equivalence checking ensures the correctness of synthesis with less human effort. An important advantage of formal verification using VIS is counterexample generation whenever equivalence or model checking fails. However, some counterexamples are difficult to analyze directly, but the counterexamples can be converted into Verilog-XL and analyzed graphically. There are some trivial differences between the syntaxes of VIS-Verilog and that of XL-simulator, but if a design is described by a subset of syntaxes which are accepted by both tools, the design description can be interacted by VIS and XL-simulator without any change.

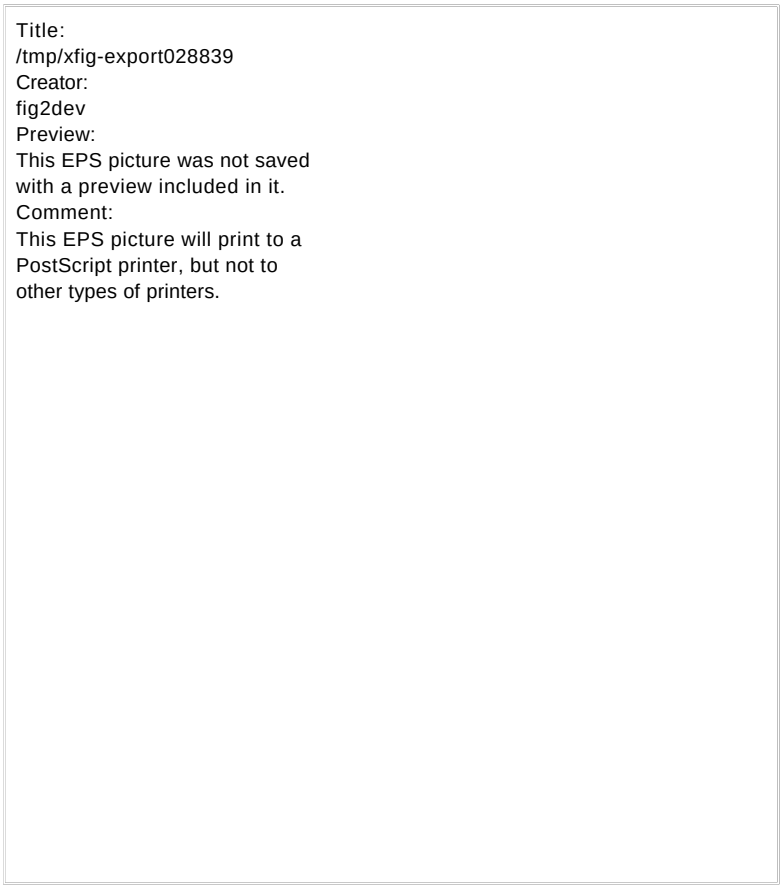


Figure Chapter 1 .1 Digital design flow using VIS

1.5 Scope of the Thesis

High-speed networks are being required for carrying all applications (voice, data, video and images) in an integrated fashion, and the most appropriate switching technique for such multimedia application is known as Asynchronous Transfer Model (ATM) [25]. Although ATM hardware is one of the most difficult designs, its formal verification examples are pitifully few. This thesis is concerned with methods for applying model checking and equivalence checking on the verification of ATM switches.

With the increasing intensive competition in ATM systems, time to market is becoming a more and more important issue in industries, so industries prefer to use decision

graph based methods than theorem proving based methods. However, even if graph based formal verification tools are used, the overhead of using formal verification is still quite high. There are four major reasons to cause this problem.

1. Many formal verification tools (especially academic formal verification tools) use some special hardware description languages instead of Verilog or VHDL [3] which are two hardware description language widely used in industry designs.
2. Due to the state space explosion problem, a design has to be abstracted in order to be verified.
3. Graph based tools have the advantage of automatic proof, but most of them must use propositional temporal logic to express properties. However, it is very difficult to express a complicated property by propositional temporal logic formulas directly.
4. Model checking can only be applied on a component without inputs. So if a component with inputs is under test, a closed environment [23], which defines the inputs, is mandatory. However, such environment is difficult to build due to the lack of rules and methods.

The thesis is aiming at the above four points. All the designs present in the thesis are described in Verilog. Using techniques introduced in the thesis, we were able to successfully verify them in VIS, and a number of design errors have been found during the verification process. Our experience proves that formal verification could be used in ATM switch designs with less overhead.

We discuss some general methods for the verification of ATM switches. Based on these methods, several different ATM switches are verified. The goal of these methods is to avoid the state explosion problem. As to model checking, we apply abstraction, reduction and compositional reasoning methods, and propose a method called *Property Division* in order to capture the property of a large system. In addition, we introduce our methods on establishing environments and describing properties in propositional temporal logic formulas. In terms of equivalence checking, we adopt modular verification to verify the

equivalence between a RTL design and a netlist design.

The principle contributions of this thesis are as follows:

1. A method for constructing environments of ATM switches using compositional reasoning, and modeling ATM switches by reduction and abstraction.
2. A method for capturing the properties of a large system and enhancing model checking.
3. Methods for describing properties using CTL[10].
4. Model checking of Fairisle ATM switch fabric, Fairisle ATM null port controller, and further verify the entire Fairisle ATM switch by property division; model checking of a commercial design block: Input FIFO of RCMP-800.
5. Equivalence checking of Fairisle ATM switch fabric and Knockout ATM switch concentrator.

1.6 Related Work

During recent years, some ATM switches were verified by various formal methods, and those ATM switches were either from academic designs or abstracted from commercial designs. The following is ATM formal verification examples we found in the literature.

- **Verification of Fairisle ATM switch fabric in HOL**

P. Curzon [14] formally verified Fairisle ATM switch fabric [14] using HOL [19]. HOL is an LCF style proof system based on higher-order logic. The verification was structured hierarchically following the implementation's module structure. The hierarchical, modular nature of the proof facilitated the management of its complexity. The structural and behavioral specifications of each module were given as relations in higher-order logic [19]. The major advantages of the verification of the switch fabric in HOL are: the excellent expressivity of the specification language; the confidence afforded in its results and the potential for scalability. However, the verification in HOL required expertise and lots of

efforts on circuit interpretation.

- **Verification of Fairisle ATM switch fabric in MDG**

Tahar et al. [37] [38] verified the Fairisle ATM switch fabric in an automatic fashion using the MDG (*Multiway Decision Graphs*) [13] by property checking and equivalence checking. MDG is used to represent sets of states as well as the transition and output relations. Based on a new technique called *abstract implicit enumeration* [13], the MDG tools are able to perform safety property checking, combinational and sequential equivalence checking. The verification in MDG has the advantage of using abstract data types and un-interpreted functions with a rewriting facility, hence allowing larger circuits to be verified. However, the current MDG tools are using a special hardware design language (MDG-HDL) which need interpretation if we verify an industry design.

- **Verification of a Commercial ATM switch in SMV**

Chen et al. [8] at Fujitsu Digital Technology Ltd, identified a design error in an ATM circuit using the tool SMV (*Symbolic Model Verifier*) [26] by verifying some properties expressed in CTL. Actually, the error was found during the chip testing process. Aiming at the specific test error, they established an ATM model by the abstraction and reduction of their original design. Using model checking in SMV, a design error was identified in the ATM model. However, the ATM model was reduced and abstracted a lot from the original design according to the specific error, and the same ATM model may not be used to verify other properties of the original design.

- **High-Level Design and Validation of ATM Switch**

Rajan et al. [30] used a combination of theorem proving [19], model checking and simulation to verify a high-level ATM model. They used model checking to verify some control components in the ATM model, and applied exhaustive simulation to verify some

operational components. Then theorem proving was applied to verify the whole ATM switch model. They discovered bugs in the high-level ATM model which was presumed correct by simulation. Combination of various formal verification methods is a trend in the formal hardware verification, but such a high-level model ignored a lot of details of a real design. In addition, since theorem proving was involved in the verification, the expertise in theorem proving was required.

- **Verification of Fairisle ATM Switching Fabric in MEPHISTO**

Schneider et. al [33] formally verified Fairisle ATM switch fabric using the verification system MEPHISTO which is based on the HOL theorem prover. They described the structure of each of the modules used in the design hierarchically and provided their behavioral specification using hardware formulas. Although they automated the verification of lower-level hardware submodules, they have not accomplished the complete verification of the implementation against the intended overall behavior of the switch fabric.

- **Verification of Fairisle ATM Switching Fabric using HSIS**

Gracez [18] has also verified some properties on the implementation of fabric using the HSIS model checking tool. The author described the netlist implementation of the ATM switch fabric using a subset of Verilog, and checked properties on submodules of the fabric using model checking. No model checking on the whole switch fabric, nor a verification against a high-level specification was reported, however. Moreover, in some cases a slightly different implementation of a module was described in order to ease the verification.

The above ATM formal verification examples demonstrate that some simple ATM switches can be verified by formal verification methods. However, these methods either use their own hardware design language or require a large changes from the original designs, so the overall overhead are pretty large. In this thesis, we will use Verilog hardware design

language to describe the design. Using the methods introduced in the thesis, we are able to verify the ATM switches with relatively small modifications on the original designs.

Chapter 2

Model Checking Methodology

In this chapter, we consider the model checking methodology of ATM switches. We will exploit a number of verification methods into the model checking of ATM switches. These methods include compositional verification, property division, abstraction and reduction, and they are jointly applied in the verification of ATM switches. Compositional reasoning gives the theory fundamental for the establishment of environments. Property division methods, which are jointly adopted with compositional reasoning, facilitates capturing properties of a large design model. Reduction and abstraction dramatically decrease the complexity of an ATM switch model without changing the relations of target signals.

Using the above methods, we have to build up an environment and express a property in temporal logic formulas (i.e. CTL). So we also introduce our methods on establishing an environment and representing a property in CTL.

2.1 Compositional Verification

The idea behind compositional reasoning is to exploit the natural decomposition of a system into communicating parallel processes. We will try to verify properties of individual components, infer that these properties hold in the complete system, and use them to deduce additional properties. The second step, inferring that local properties hold on the complete system, is the key requirement for compositional verification. Thus, we wish to examine the compositional model checking problem: how do we check that a specification is true of all systems that can be built using a given component? The theorems which support full CTL compositional model checking are very hard to develop [23]. But as for

ACTL, a subset of CTL, the problem is efficiently decidable.

2.1.1 CTL and ACTL

Temporal logic is a logic for expressing the relative ordering of events in time without mentioning time explicitly. CTL is a well-known temporal logic used in model checking. All temporal operators in CTL are interpreted relative to an implicit "current state", and each operator consists of two parts. The first is called a path quantifier and is either *A* or *E*. *A* denotes that something should be true of all "paths" starting at a current state. In contrast, *E* is used to specify the existence of a path with a certain property. The second part of a temporal operator is either *X*, *U*, or *V*. These are used to describe the ordering of events along the path or paths indicated by the *A* or *E*. The intuitive meanings of *X*, *U* and *V* are as follows:

1. *X p*: *X* is read as "next time". *X p* is true of a path if the formula *p* is true at the second state on the path. Thus, *p* is used to express properties about the immediate successors of the current state.

2. *q U p*: *U* is the "until" operator. The formula expresses that *q* is true until a point where *p* is true. Thus, *A*(true *U p*) indicates that *p* will be true in any paths in the future. *A*(true *U p*) also can be expressed as *"AFp"* where "*F*" means "future". Similarly, *E*(true *U p*) can be expressed as *EFp*.

3. *q V p*: The *V* operator is the dual of *U* and is read as "release". A path satisfies *q V p* if *p* is true at the current state, and *p* remains true up to and including the first point where *q* is true. There is no requirement that *q* ever becomes true, but when it does, it "releases" the requirement that *p* be true. *A*(false *V p*) means that *p* never releases in any path, so it has the same meaning as *AGp* where *G* is intended to express invariance. Similarly, *E*(false *V p*) has the same meaning as EG p.

   Let us now consider some example CTL formulas and their intuitive meanings.

1. *AG(req=1 -> AF ack=1)*: This formula states that for all reachable state(*AG*), if the state satisfies *req = 1* ("a request is made"), then at some later point (*AF*) we must encounter a state with *ack=1* ("an acknowledgment is received"). Note that AF is interpreted relative to the state where *req = 1*. The outer *AG* is interpreted starting with the initial states of the system.
2. *AG AF enabled = 1*: No matter what state we reach, at some later pointer we must encounter a state where *enabled* is 1, then we must reach yet another such state. In other words, *enabled* must be 1 infinitely often.
3. *AG EF restart =1*: For any reachable state, there must exist a path starting at that state that leads to a state satisfying *restart=1*. It must always be possible to "restart the system".

ACTL is a subset of CTL, which eliminates the ability to represent the existence of a path, i.e., the *E* path quantifier, and this subset is sufficiently expressive to cover almost all of the temporal formulas that are used as specification in practice. Intuitively, this is because we generally want to require that a system *must* behave correctly, rather than it *may* behave correctly.

2.1.2 Compositional Reasoning

In order to avoid state space explosion problem, compositional reasoning method [23] was proposed. It exploits the natural decomposition of a system. By using composition reasoning method, we verify components in the system with their environments, rather than a whole system. To show that the properties which are valid in individual components with the environment are still hold in the entire system, Long [23] has formally proved that "if the other components in the system guarantee the behavior of the environment, then the verified properties are true of the entire system". This is the *assume-guarantee* style of verification [28] .

The assume-guarantee style of verification was first advocated in the context of

temporal logic by Pnueli [28]. In Pnueli's system, we work with triple of the form *ágñMáfñ* . The most common reading of such a triple is "if the environment of M satisfies g, then M in this environment satisfies f". A typical chain of reasoning would be as follows:

$$\frac{\langle\rangle\ N\ \langle g\rangle \qquad \langle g\rangle\ M\ \langle f\rangle}{\langle\rangle\ \ M \mid\mid N\ \langle f\rangle}$$

Equation Chapter 2 .1 Assume-guarantee style of verification

Here, we are asserting that if:

1. N satisfies g; and
2. if the environment of M satisfies g, then M satisfies f

then the composition of M and N (i.e. $M \mid\mid N$) will satisfy f. The advantage of doing the verification in this manner is that we never have to examine the composite state space of $M \mid\mid N$. Instead, we check g using just N, and then check f using only M and the assumption g which is an environment of M (i.e. N').

Based on the above compositional reasoning method, we establish an environment of M (i.e. N') which satisfies temporal formulas g, and check the property f on the module M with N'. If the property f is satisfied by $M \mid\mid N'$, we can conclude that $M \mid\mid N$ satisfies f. Since N' is much smaller than N, the state space of $M \mid\mid N'$ is much less than that of $M \mid\mid N$. By this method, model checking can handle much larger circuits. However, Long [23] has proved that such compositional reasoning method is only applicable in a ACTL.

Once the logic can only represent behavior over all paths, we will just need to consider a single "maximal" closing environment [23]. Although composing with any other closing environment will eliminate some paths, since our formulas only express behavior overall paths, such pruning will not change a formula from true to false. Further, if the composition of the given component with its maximal closing environment satisfies the specified formula, then the formula obviously must be true of all closed systems containing the component. We will use an example to further illustrate this in Section 2.2.

2.2 Environment

Using model checking, a verification target must be a structure which has no inputs [23]. So when we verify a design which has a number of inputs, we must build up an environment to define each input signal. Because there is no input in the environment, we call it *closing environment*. In [23], Long introduced maximum closing environment in model checking, however we do find some drawbacks on maximum closing environment. We will use an example to illustrate its drawbacks first, and then introduce an actual closing environment which is more closed to the real environment circuit than an maximum closing environment, and further explain how to implement an actual environment.

A real environment for a circuit *M* is components associated with *M* in the original system, denoted as *N*. The maximal closing environment for a circuit *M*, denoted *E(N)* is defined as follows:

1. S' = F, where F is the set of all labeling functions over A_I.
2. I' = F.
3. $A'_I = \Phi$.
4. $A'_O = A_I$.
5. R'(s_o', f', s_i') is identically true.
6. L'(f, a) = f(a)

S is set of states; *I* is a nonempty set of initial state. A_I is a set of input state components, and each element *a* of A_I has a correspondent domain D_a of possible values. A_O is a set of output state components, and each element *a* of A_O has a corresponding domain D_a of possible values. *R* is a transition relation relating a starting state in *S*, a labeling function over A_I, and a ending state in *S*. For every $s_o \hat{I} S$ and labeling function *f* over A_I, there must exist some $s_i \hat{I} S$ such that $R(s_o, f, s_i)$. *L* is a function that takes a state component *a* and return an element of D_a.

In the above definition, the symbols with " ' " mean the symbols of *E(N)*, and symbols without "' " means the symbols for M.

According to the above definition, the states of a maximum environment is the set

of all labeling functions over A_I while the states of a real environment is the subset of all labeling functions over A_I. That means that if the composition of the circuit M and its maximum environment $E(N)$ (denoted as $M \,||\, E(N)$) is satisfied by an ACTL property, and the property must be valid for the composition of the circuit M and its real environment N (denoted as $M \,||\, N$). We will use the following example to illustrate this, and the example was also used in [23].

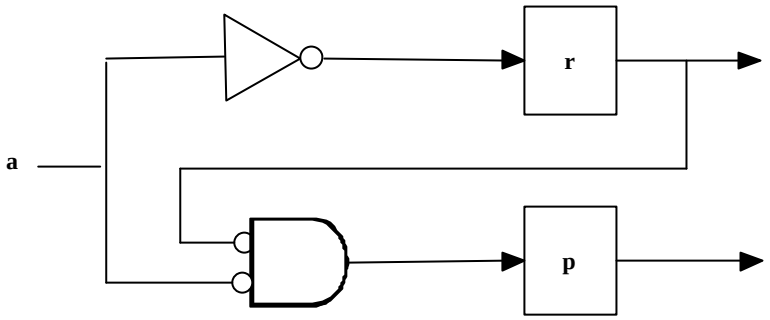


Figure Chapter 2 .1 A handshake circuit

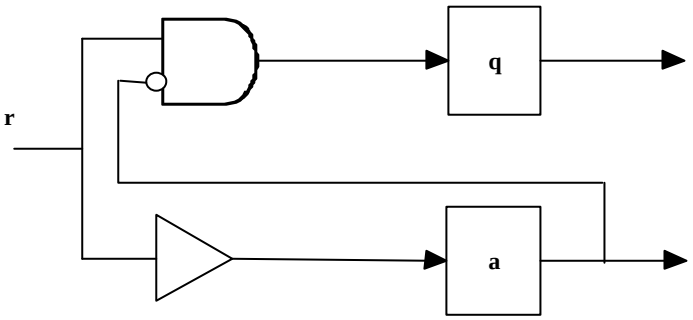


Figure Chapter 2 .2 Environment of the circuit of Figure Chapter 2 .1

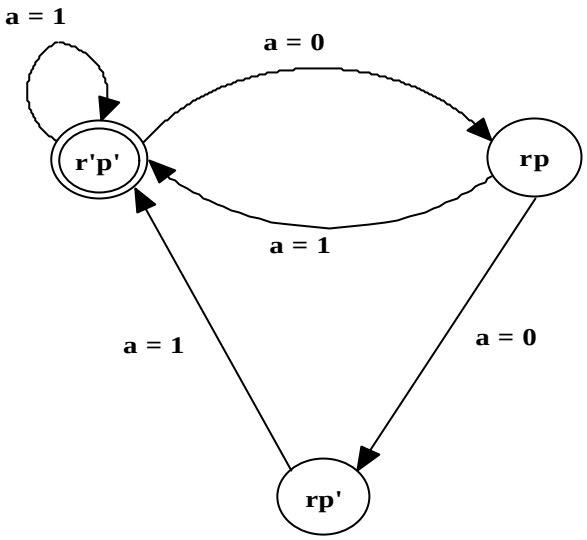


Figure Chapter 2 .3 State transition diagram for the circuit of Figure Chapter 2 .1

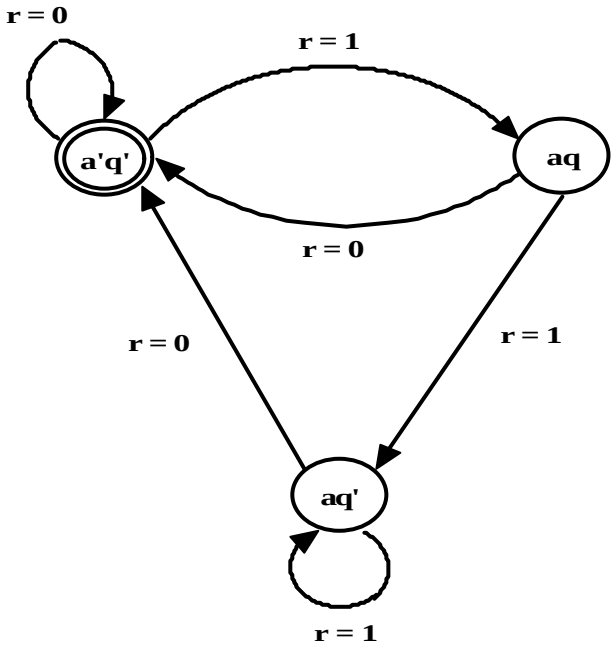


Figure Chapter 2 .4 State transition diagram for the circuit of Figure Chapter 2 .2

Consider a process that communicates with its environment via a 4-phase handshaking protocol. The process can make requests of the environment by setting one of its visible state component *r* to 1. The environment responds this request by setting the visible state component *a* to 1. When the process sees the acknowledgment, it removes its

request by setting *r* to 0. Then after the environment sets *a* back to 0, the cycle repeats. The process will also have a state component *p* that will set to 1 when it first makes request. The state transition diagram corresponding to this process is shown in Figure Chapter 2 .3, and the actual circuit is Figure Chapter 2 .1. In Figure Chapter 2 .3 and other state transition diagrams in Chapter 2, the double circle indicates an initial state, conditions on arcs are used to give the input conditions under which the transition can be taken, and " ' " denotes logic "*NOT*". The circuit shown in Figure Chapter 2 .2 is a possible environment for the circuit of Figure Chapter 2 .1. It receives requests via the input *r* and gives acknowledgments using the output *a*. It also has an output *q* that becomes 1 when it first produces an acknowledgment. When we compose the two circuits, the output *r* of the circuit in Figure Chapter 2 .1 is tied to the input *r* of Figure Chapter 2 .2. Similarly, the output *a* of Figure Chapter 2 .2 drives the input *a* of Figure Chapter 2 .1. The overall circuit is shown in Figure Chapter 2 .5. Figure Chapter 2 .4 and Figure Chapter 2 .6 are the state transition diagrams for Figure Chapter 2 .2 and Figure Chapter 2 .5, respectively.

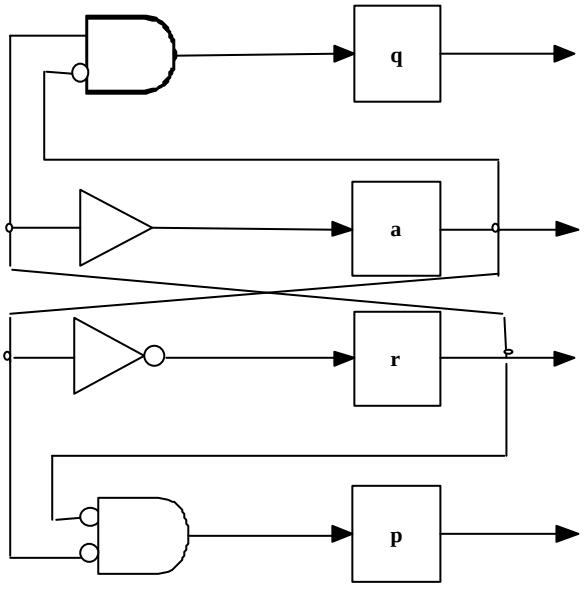


Figure Chapter 2 .5 Composed circuit

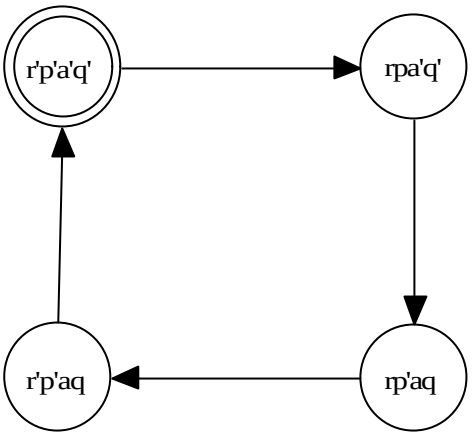


Figure Chapter 2 .6 State transition diagram representing the composite circuit

According to the above definition of a maximum environment, the maximal closing environment of Figure Chapter 2 .3 can be expressed as Figure Chapter 2 .7. (Here we use state transition diagram of Figure Chapter 2 .1 (Figure Chapter 2 .3) to represent the circuit of Figure Chapter 2 .1 because they are functionally equivalent) In Figure Chapter 2 .7, *a* is either "0" or "1", and it can be expressed as a nondeterministic variable in some formal verification tools. If we consider Figure Chapter 2 .1 and Figure Chapter 2 .7 as a circuit *M* and its maximum environment *E(N)*, *M* || *E(N)* can be obtained from Figure Chapter 2 .3 by considering the states when *a* is '1' and '0'. Figure Chapter 2 .8 is the state transition diagram of *M* || *E(N)*. Figure Chapter 2 .2 is a real closing environment of *M* (i.e. *N*), and Figure Chapter 2 .5 can be expressed as *M* || *N*. Comparing Figure Chapter 2 .6 and Figure Chapter 2 .8 for each state in *M* || *N*, we can obtain a corresponding state in *M* || *E(N)* by dropping the labeling for the state component *q*. As an example, the state *r'p'a'q'* in *M* || *N'* maps to the state *r'p'a'* in *M* || *E(N)*.

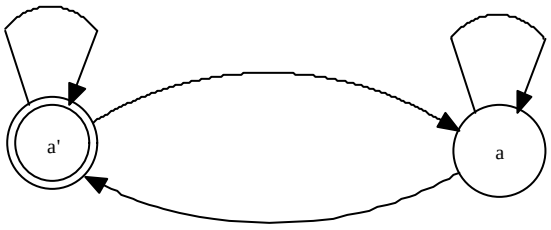


Figure Chapter 2 .7 The maximal closing environment for the structure of Figure Chapter 2 .3

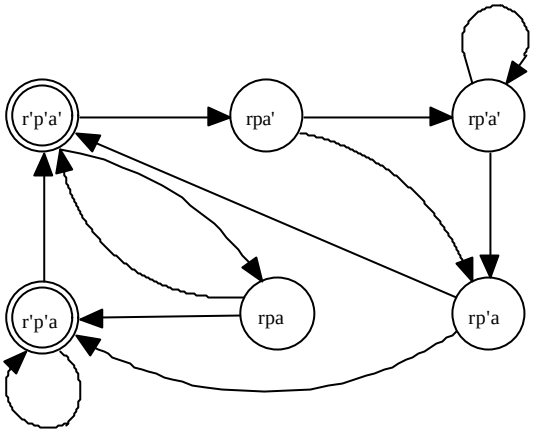


Figure Chapter 2 .8 The composition of the structure Figure Chapter 2 .3 and its maximal closing environment

In essence, the state transition of *M* || *N* can be embedded in that of *M* || *E(N)*. Now consider a formula of ACTL which describes properties of all paths from a state. If such a formula is false at some states in *M* || *N*, then we can find a path demonstrating why it is false. This path is then mapped into a corresponding path in *M* || *E(N)*, so we can prove that this path demonstrates that the corresponding state in *M* || *E(N)* does not satisfy the property either. If we verify that a formula is true *for M* || *E(N)*, then we know that the formula holds in all closed systems that contain *M*. Further, if the formula is false for *M* || *E(N)*, then obviously we could find a closed system containing *M* for which the formula is false. Therefore, to prove a property is valid on *M* || *N*, we can deduce it by proving the property is valid on *M* || *E(N)*.

But, if a property is false for *M* || *E(N)*, it may be true for *M* || *N* because every state of *N* can be mapped in *E(N)* while each state of *E(N)* may not be mapped into *N*. To explain this, we still use the example of subsection 0.2.1. Figure Chapter 2 .6 is the state transition diagram of the circuit Figure Chapter 2 .5 which can be seen as the composition of Figure Chapter 2 .1 and Figure Chapter 2 .2. If the circuits Figure Chapter 2 .1 and Figure Chapter 2 .2 are considered to be *M* and *N*, respectively, Figure Chapter 2 .5 can be expressed as *M* || *N*. Figure Chapter 2 .7 is an maximum environment of *M (i.e. E(N))*,

and *M* || *E(N)* is represented as Figure Chapter 2 .8. In Figure Chapter 2 .6 (i.e. *M* || *N*), it has the property that the next state of *rpa'q'* will always be *rp'aq*, but this property is no longer satisfied in (*M* || *E(N)*) because the next state of *rpa'* can be *rp'a'* or *rp'a* (Figure Chapter 2 .8). Therefore, a maximum closing environment is not the best environment in model checking.

In order to make the verification accurate, a environment which uses the same variables as the maximum environment and expresses its original model as much as possible is the best one. So we propose to use actual closing environment which defines some inputs as the original behaviors and some inputs as nondeterministic variables. In reality, some input signals behave in a simple manner, then we can specify its behavior in the environment. For instance, in some ATM switches, there is "*framestart*" signal which can synchronize the cell transmission, and it strobes cyclically in a certain period. So we can define this signal as its real behavior in the environment. However, some input signals have a complicated behavior, and expressing it as its original model will increase the state space dramatically. For this case, we can leave these signals as nondeterministic variables. And also, data path signals could be defined as non-deterministic variables. The following chapters include some actual environment implementations.

Since it is possible that a nondeterministic variable holds one value all the time, the behavior with other values will never come out. Therefore, we sometimes need to give fairness constraint [4] to prevent a nondeterministic variable sticking on a certain value.

2.3 Property Division

To capture the whole behavior of a system, we usually need to verify some global properties. However, the global properties are difficult to be verified due to state space explosion. Therefore, we propose property division techniques which subdivide a property to several sub-properties and check each sub-property in its correspondent submodule separately. According to the dependencies of subproperties, we classify property division into cascade property division and parallel property division.

2.3.1 Cascade Property Division

Cascade property division divides a property into several sequentially related sub-properties, and every consecutive sub-properties are related with each other.

To illustrate this, we use a simple example. Consider a target system *S* (*M* || *N*) which consist of submodule *M* and *N*. *M* and *N* are sequentially related. If we want to verify a property *P* of *S*, we can divide *P* into two subproperties *P1* and *P2* which are expected to be valid in *M* and *N*, respectively. If submodule *M* and the environment of *M* (i.e. *N'*) satisfies *P1* and submodule *N* and the environment of *N* (i.e. *M'*) satisfies *P2*, we can conclude that the target system *S* satisfies *P*. The proof strategy can also be expressed as an inference rule:

$$\frac{\langle true\rangle\ M \parallel N' \langle P1\rangle \quad \langle true\rangle\ N \parallel M' \langle P2\rangle \quad \langle true\rangle\ P1 \parallel P2\ \langle P\rangle \quad M \parallel N = S}{\text{<true>S<P>}}$$

Equation Chapter 2 .2 The proof of cascade property division

The above formula is similar to that in Section 2.1. Actually, cascade property division is based on compositional reasoning. In Equation Chapter 2 .2, the proof strategy for *átrueñ M || N' áP1ñ* and *átrueñ N || M' áP2ñ* are exactly the same as Equation Chapter 2 .1. The proof of Equation Chapter 2 .2 is straightforward. By compositional reasoning, if *M || N'* satisfies *P1*, then *M || N* satisfies *P1*. Similarly, if *M' || N* satisfies *P2*, then *M || N* satisfies *P2*. These two important assumption make the above proof valid.

Since cascade property division is based on compositional reasoning, it works only for ACTL property. The following lists a set of typical properties which can be used in cascade property division.

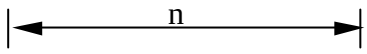
- Safety Property:

  IF **AG** (P -> **AX**^{n} Q) and **AG** (Q -> **AX**^{m} R)

  THEN **AG**(P -> **AX**$^{m+n}$R)

  where **AX**^{n} Q is an abbreviation for

  AX AX AX …AX AX AX Q

- Liveness Property:

  IF **AG (**P -> **AF** Q) and **AG**(Q -> **AF**(or **AX**^{n})R)

  THEN **AG**(P -> **AF** R)

  IF **AG** (P -> **AF**(or **AX**^{n}) Q) and **AG**(Q -> **AF** R)

  THEN **AG**(P -> **AF** R)

The proof of above equations is based on the following three theorems, and we do not give the detail proof here.

1. IF P -> Q and Q -> R, THEN P -> Q
2. **AX**^{n} **AX**^{m} R= **AX**$^{n+m}$ R
3. **(AX**^{n} **or AF) AF** R = **AF** R

The cascade property division is quite useful in the verification of an ATM switches. In any ATM switches, ATM cells carry ATM headers through a switch, and a switch processes the headers and transmit ATM cells to their destined output ports. So if we track the ATM cells through an ATM switch, it is possible to get proper break points where the properties could be appropriately divided. However, using cascade property division, the target design cannot be seen as a black box. How to divide a property depends on the understanding of a design. Sections 3.6 and 3.4.1 will give real examples of cascade property division.

2.3.2 Parallel Property Division

While cascade property division divides a property into sequentially related sub-properties, parallel property division divides a property into several independent sub-properties, and checks every sub-property by every correspondent reduced submodule that is extracted from a design.

The idea behind parallel property division is to remove the redundant structure of a design and perform model checking only on the property-related circuits. By proper partition, a design may be divided into several design units while keeping the functional equivalence with the original design. Each design unit could have some inputs of the original design, but it cannot share outputs with the other one. Also design units ought not to have any interactive signals among them. If one global property can be separated to several independent sub-properties, each sub-property may be checked in a certain design unit. Since a design unit is much smaller than the whole design, state space is saved.

Here is a virtual example of parallel property division. Consider a closed system *S*: (1) *S* is functionally equivalent to the combination of *n* submodules (M_1 || M_2 || ... || M_n). (2) The submodules have some or entire inputs of *S*, but they are independent. That means that they have different outputs and have no interactive signals with each other. We intend to check the property (I -> O_1 and O_2 and ... and O_n) which might not be verified directly due to the large state space, then if we check each sub-property (step 1 to step 4 in Equation Chapter 2 .3), we can conclude that step 5 is valid. In Equation Chapter 2 .3, "I" represents inputs expression, and "O_i"(i=1, 2, ..., n) denotes different outputs expression. "I-> O_i" means the property which "I" implies "O_i".

1. IF M_1 satisfies the property(I -> O_1); and
2. IF M_2 satisfies the property (I -> O_2); and
3. ..., and
4. IF M_n satisfies the property (I -> O_n)

5. THEN *S* satisfies Property (*I* -> O_1 and O_2 and ... and O_n)

Equation Chapter 2 .3 A virtual example of parallel property division

The proof of Equation Chapter 2 .3 is based on the following two theorems, and we do not give the detailed proof here.

1. If a property is satisfied by its related circuits of a system, the property will be also valid in the system.
2. If (I -> Q1) and (I -> Q2), THEN I -> Q1 and Q2

Since most of ATM switches are regular designs, we could divide an ATM switch into several parts, and check each sub-property in a certain design unit. In addition, as an ATM switch, any input ports can be routed to every output port, and every output ports are relatively independent, so an *n-output* ATM switch may be divided into *n* switch units. The actual example is shown in Section 3.4.2. Furthermore, the idea of parallel property division could also be used to "hide" property-unrelated circuits when verifying a property, and Chapter 5 uses the "hide" method to verify some properties of RCMP-800 Input FIFO.

2.4 Property Expression

As well known, higher level language has better expressivity. For instance, C language has better expressivity than assembly language. Among logic languages, such rule is still valid, Higher Order Logic [19] have much better representation ability than CTL which belong to temporal propositional logic. Although CTL model checking has an automatic fashion, its poor expressivity really restricts the application. To ease expressing a property in CTL, we summarize three methods: *Environment Modification*, *Internal Signal Usage and Counter Reduction*.

- **Environment Modification**

A property consists of two parts: assumptions and conclusions. However, if

several assumptions do not happen at the same state, it is impossible to express them in a CTL formula. In this case, we could use the environment to express one or more assumptions. We call this method Environment Modification. Environment Modification is really able to express a lot of properties, but the overhead is that we have to modify the environment for each property.

- **Internal Signal Usage**

A property is a logic relation between the inputs and outputs of a component. If the property is impossible to be represented by the inputs and outputs in CTL directly, we could use "Internal Signals" to express a number of sub-properties in CTL. After these sub-properties are verified, the property could be deduced from these sub-properties by property division. This method is called "Internal Signal Usage". Internal Signal Usage can be used to express any kind of properties, but the verifier must be very clear about the design to choose appropriate internal signals in CTL formulas.

- **Counter Reduction**

Counters are often used to synchronize various behaviors inside the system, and the scale of counters directly affect the possible states of the system. Needless to say, counter reduction will dramatically reduce the CPU time during model checking. In Environment Modification method, the number of CTL formulas for a property sometimes is a proportional to the scale of the counters. By using Counter Reduction and Environment Modification, we could write much less CTL formulas to represent a property. Also in Internal Signal Usage method, the scale of counters sometimes has an influence on the number of CTL formulas for a property which is related to the counter. So Internal Signal Usage method also can get benefits from Counter Reduction.

While Environment Modification and Internal Signal Usage are two methods on expressing a property in CTL, Counter Reduction reduces the number of CTL formulas for a property when it combines with either Environment Modification or Internal Signal Usage method. The practical examples on how to apply the three methods will be

demonstrated in Chapter 4.

2.5 Reduction and Abstraction

An ATM switch consists of memory, control circuits and data-path circuits. In simulation, we usually build up a memory model and simulate the whole system. However, we could not use such memory model in either model checking or equivalence checking because the memory will introduce a lot of states. Instead, we often do reduction and abstraction on a system in order to perform model checking and equivalence checking on it.

Because the advantage of model checking is to verify a control circuit with a lot of interactive signals, we usually perform reduction and abstraction on the memory and datapath. In some cases, we need to reduce the scale of control circuit as well. For example, we often reduce the scale of a cell counter in an ATM switch control circuit.

Memory is a kind of state holding element. It is impossible to verify memories by model checking because the state space increases exponentially with the capacity of a memories. For example, for 1000 byte memories, there will be 2^{8000} states while the largest example that model checking can handle is around 2^{120} states. To verify a system with memory, we usually take memories away from the system and only verify the memory read / write address, the memory data bus and read / write enable signals. In Chapter 4, we use Fairisle ATM null port controller as an example to illustrate how to use the memory peripheral signals to verify the correct memory access. Likewise, when verifying the FIFO in Chapter 5, we only check the FIFO read and write in its control circuits. This reduction is reasonable because the target of function verification is logic circuits instead of memories which are usually regular and verified by other ways.

Another component which is often reduced or abstracted is datapath circuits. Datapath circuits usually have wide data buses and operational circuitry, these circuits are not very complicated but they did occupy a lot of state space in the verification. For this reason, we usually do reduction and abstraction on it. Reduction is often used in the data

bus. For example, we could reduce the data bus from 8 bits to 1 bit as we will show in Chapter 3. On the other hand, for the operational circuits, we often use abstraction. For example, we can use three registers (a, b, c) to replace an adder (a + b = c) [23]. When register a and b have a value, (a+b) value is updated in register c. Instead of verifying a system with the adder, we only verify the system with three registers, so the state space can be decreased.

Although reduction and abstraction are very powerful methods in model checking and equivalence checking, it is not a good solution for the practical verification because it needs to modify a design. Even if we have to use them, we must try to modify our design as little as possible. A good coding style will make some reduction and abstraction easier. For example, if we define the width of a data bus as a constant variable. We can only re-define the constant variable to change the data bus width.

In this following three chapters, we apply the above model checking techniques on ATM hardware verification. In Chapter 3, we will introduce our experience on the model checking of Fairisle ATM switch fabric, and we will also adopt property division to enhance model checking and verify the entire Fairisle ATM switch which consists Fairisle ATM switch fabric and Fairisle ATM port controller model. In Chapter 4, we will illustrate three CTL property expression methods by verifying Fairisle ATM switch null port controller. A commercial ATM design block will be verified in Chapter 5 by using "hide" method, and we demonstrate that it is practical to verify an ATM commercial block using model checking.

There are a lot of CTL formulas in this thesis. The numbering of CTL formulas follows this rule: the first digit denotes the chapter number, and second digit means property number, and the last digit indicates the number of CTL formulas for a certain property. For instance, (3.3.2) means the second CTL formula of Property 3 in Chapter 3. Beside the above numbering rules, we use *a*, *b* and *c* to indicate Environment Modification, Internal Signal Usage and Counter Reduction, respectively in Chapter 4. For example, (4.3.a.1) means the first CTL formula of Property 3 using Environment Modification in Chapter 4. In

addition, in all the CTL formulas of the thesis, “!”, “->”, “*”, “+” and “^”denote logical “not”, “imply”, “and”, “or” and “xor”, respectively.

Chapter 3

Verification of Fairisle ATM Switch

In this chapter, we present our results of formally verifying an ATM switch fabric using VIS. By this example, we show how to use model checking to verify a design. In addition, we introduce how to apply property division to enhance the model checking and further verify an entire ATM switch by property division.

The device we investigated is a part of a network which carries real user data: the Fairisle ATM network, designed and in use at the Computer Laboratory of the University of Cambridge. The component we considered is the Fairisle 4 by 4 switch which consists of a Fairisle 4 by 4 switch fabric and four Fairisle ATM port controllers, performs the actual switching of data cells and forms the heart of the ATM Fairisle communication network.

3.1 The ATM Switch Fabric

The Fairisle ATM switch consists of three types of components: input port controllers, output port controllers and a switch fabric (Figure Chapter 3 .1).

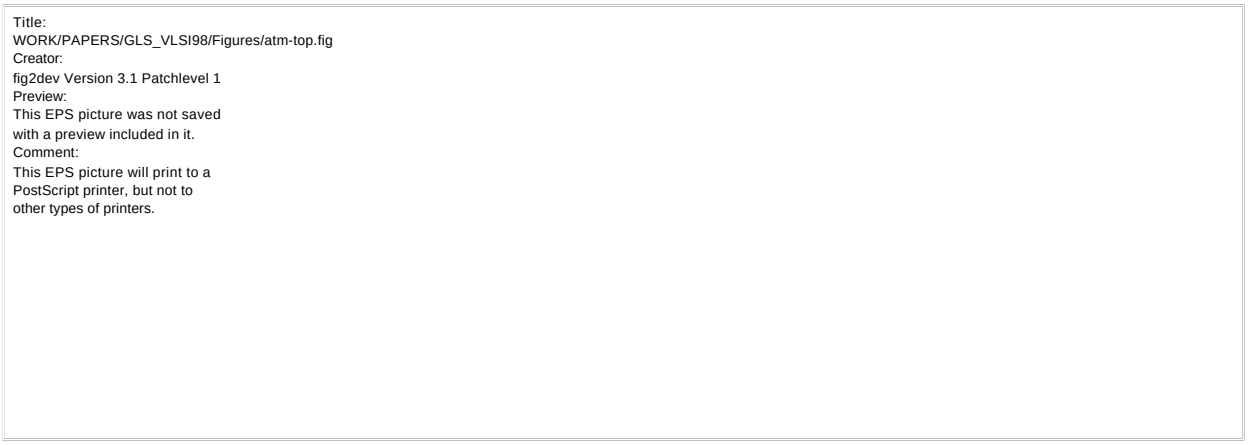


Figure Chapter 3 .1 The structure of the Fairisle ATM switch

The null port controller synchronizes incoming and outgoing data cells, appending control information in the front of the cells in a routing tag (Figure Chapter 3 .2).

Figure Chapter 3 .2 The routing tag of a Fairisle ATM cell

This tag is stripped off before the cell reaches the output stage of the fabric. The fabric switches cells, consisting of a fixed number of data bytes, from the input ports to the output ports according to the routing tag. If different port controllers inject cells destined for the same output port (which is indicated by the route bits) into the fabric at the same time, then only one will succeed, and the others must re-try later. The priority bit in the routing tag is used for arbitration, and the high priority cells are given precedence. For those with the same priority, round-robin arbitration is performed. The output controllers are informed of whether their cells were successful or not through the acknowledgments generated by the output ports. The fabric passes the acknowledgment from the requested output port to the successful input port, and does not forward the acknowledgment to unsuccessful input ports or forwards the negative acknowledgment when the output port controllers are running

short of buffer space. The port controllers and switch fabric all use the same clock, and they also use a higher-level cell frame clock: the *frameStart* signal (*fs*). It ensures that the port controllers inject data cells into the fabric synchronously so that the routing tags arrive at the same time.

3.1.1 Switch Fabric Behavior

The behavior of the switch fabric is cyclic. In each frame, the fabric waits for cells to arrive, reads them in, processes them, sends successful ones to the appropriate output ports and sends acknowledgments. It then waits for the next round of cells to arrive. The boundaries of separate cycles are determined by the *frameStart* signal. Whenever it goes high, a new cycle commences. The cells from all the input ports start when a particular bit (the active bit) of any input port goes high; the fabric does not know when this will happen. However, all the input port controllers must start sending cells at the same time within the frame. If no input port raises the active bit through the frame then the frame is inactive. Otherwise it is active. In order to initialize the fabric correctly for the forthcoming frame, the active bits must be low in the 2 cycles prior to the arrival of the *frameStart* signal. Because the decision is completed 3 clock cycles after the header time (arrival of routing tag), the fabric begins to send acknowledgment at least 3 clock cycles after that. In [21], the overall behavior of the switch fabric (including constraints from the port controllers) is expressed in form of one state machine composed of 14 states.

3.1.2 Switch Fabric Implementation

Figure Chapter 3 .3 shows a block diagram of the switch fabric implementation. It consists of an *arbitration* unit, an *acknowledgment* unit and a *dataswitch* unit. The arbitration unit is composed of a timing unit, a decoder, a priority filter and a set of arbiters. The decoder reads the routing tags of the cells and decodes the port requests with low priority and those from inactive inputs, and passes the actual request situation for each output port to the arbiters. The arbiters make arbitration decisions for each output port *i* by setting values for

the corresponding *outputDisable[i], xGrant[i], and yGrant[i]* signals. The dataswitch switches data from input ports to expected output ports according to the signals *xGrant[i]*, *yGrant[i]* and *outputDisable[i]*. The acknowledgment unit passes appropriate acknowledgment signals to the input ports according to these signals as well.

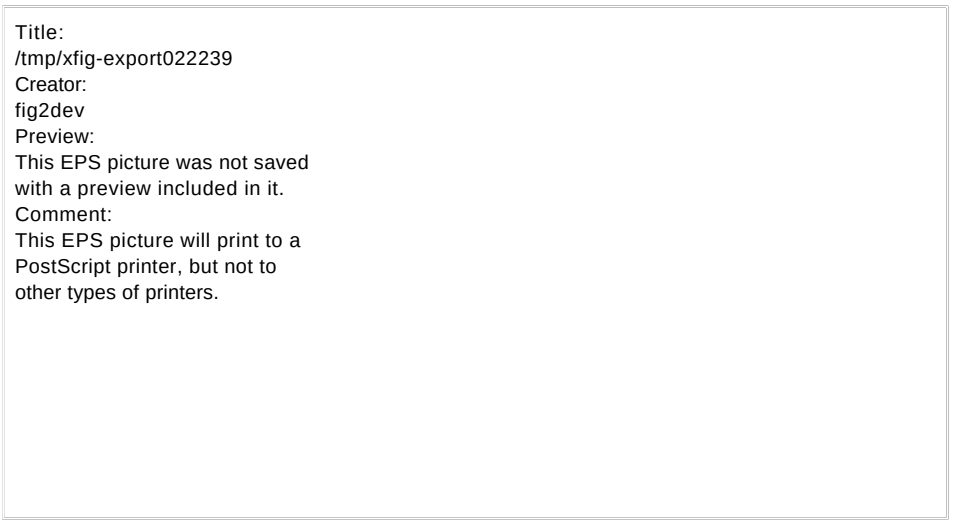


Figure Chapter 3 .3Fairisle switch fabric implementation

3.2 Model Checking

3.2.1 Environment for the switch fabric

As stated in Chapter 2, we have to give an environment to verify the properties of the fabric. The switch fabric's interface with the port controllers consists of the signals *frameStart*, 32-bit data inputs, 32-bit data output, 4-bit acknowledgment inputs and 4-bit acknowledgment outputs (Figure Chapter 3 .3). In our approach, we modeled the port controllers as a finite state machine. Since the *frameStart* signal is cyclic every 64 clock cycles, we could express the port controllers as a 68-state environment state machine (Figure Chapter 3 .4). This 68-state environment state machine is inspired from the work described in [37]. In Figure Chapter 3 .4, there are 68 states enumerated by integers. Arrows denote state transitions, and t_s, t_h and t_e denote start of a frame, start of an active

cell (header arrival) and end of a frame (which is the start of the next frame) respectively. f_s, *h* and *d* above the states mean that the *frameStart* signal, the routing tag (header) of an active cell and the data, respectively, are generated in that state. States 1 to 5 are related to the initialization of the fabric. States 6 to 68 represent the cyclic behavior of the fabric, where one cycle corresponds to one frame [37].

Figure Chapter 3 .4 68-state environment state machine

While trying model checking by this 68-state environment state machine, we noticed that it need a lot of CTL formulas for 1 property and also increases the CPU time of model checking. Thus we combined the states with the same behavior to one state; for instance, because states 13 to 64 of Figure Chapter 3 .4 have the same behavior, which is that data are transferred to the fabric, we combined them to one state that is state S3 in Figure Chapter 3 .5. Recall Counter Reduction method we introduce in Chapter 2, such states combination has the same idea. Since there is no counter inside the fabric, we do not need to change the design.

Figure Chapter 3 .5 Abstracted environment state machine with related timing diagrams

Figure Chapter 3 .5 is the abstracted environment state machine and represents the

behavior of the port controllers. t_s, t_h and t_e also correspond with states S0, S2 and S6, respectively. States S1 to S7 represent the cyclic behavior of the fabric, where one cycle corresponds to one frame. This environment state machine represents the two main features of the port controllers, namely:

1) To initialize the fabric, the active bits must be low for 2 clock cycles prior to the *frameStart* signal arriving.
2) The acknowledgment signals from output port controllers are forwarded by the fabric at least 3 clock cycles after the header arrival time. Since we fixed the state S2 as header arrival time (t_h), we give the following additional constraint which makes the environment state machine feasible.
3) If the active bit of a input port in state S2 is zero, the data input of the same input port in state S3 is also zero.

3.2.2 Property description

After establishing the environment state machine, we consider several properties of the fabric including liveness and safety properties.

Property 1: At state S2 (t_h), if input port 0 chooses output port 0 in the routing tag, eventually the data in input port 0 will be transferred to output port 0. In CTL, this liveness property is expressed as follows.

```
AG (dIn0[0]= 1 * dIn0[2]= 0 * dIn0[3]= 0 * state =
S2 -> AF (dIn0^S3 == dOut0))
(3.1.1)
```

where $dIn0^{S3}$ stores the value of din0 in state S3. Similarly, we could give other 15 liveness properties (one for each remaining 4x4 combination) to demonstrate that any input port that chooses any output port in the routing tag will eventually transfer data to that output port, but we do not express all these here. However, if there is always another cell with priority bit as '1' destining to the output port 0 and the priority bit of the cell in input port 0 is

always ‘0’, formula (3.1.1) will never be true. To prevent such unexpected scenario happening, we have to give fairness constrain. VIS supports the fairness constrains called Büchi [4] type. In order to restrict the behavior of the arbitration, we impose the following four fairness constraints. They are:

```
!(xGrant[0] = 0 * yGrant[0] = 0 )
(3.f.1)

!(xGrant[0] = 0 * yGrant[0] = 1)
(3.f.2)

!(xGrant[0] = 1 * yGrant[0] = 0)
(3.f.3)

!(xGrant[0] = 1 * yGrant[0] = 1)
(3.f.4)
```

(3.f.1) means that the data cell in the output port 0 is not always from the input port 0; likewise, (3.f.2), (3.f.3) and (3.f.4) mean that data cell in output port 0 is not always from input port 1, 2 and 3, respectively. With the fairness constrains (3.f.1) to (3.f.4), formula (3.1.1) passed model checking.

Next, we consider several safety properties. In following we present six example safety properties (Property 2 - Property 7) of the fabric along with their CTL expressions. Note that Properties 4, 5, 6 and 7 are similar to those described in [13].

Property 2: The data bytes in a cell are transferred (from t_h+5 to t_e+1, i.e. state S7) from input port 0 to output port 0 sequentially with 4 clock cycles delay.

$$AG\ (state = S7 \rightarrow (dOut0 = dIn0^{S3} + dOut0 = dIn1^{S3} + dOut0 = dIn2^{S3} + dOut0 = dIn3^{S3})$$

(3.2.1)

Property 3: The arbitration component cannot make output port 0 and output port 1 connect to the same data input port at any time.

```
AG (!(xGrant[0]==xGrant[1] * yGrant[0]==yGrant[1] *
outputDisable[0]=0 * outputDisable[1]=0))
(3.3.1)
```

Property 4: From state S3 (t_h+1) to S6 (t_h+4), the default value (zero) is put on the data output ports.

```
AG ((state=S3 + state=S4 + state=S5 + state=S6) ->
dOut0=0 * dOut1=0 * dOut2=0 * dOut3=0)
(3.4.1)
```

Property 5: Except states S5 and S6 (i.e. except the time interval t_h+3 to t_e), the default value is put on the acknowledgment output ports.

```
AG ((state = S1 + state = S7 + state = S2 + state =
S3 + state = S4) -> ackOut0 = 0 * ackOut1 = 0 *
ackOut2 = 0 * ackOut3 = 0)                    (3.5.1)
```

Property 6: In state S7 (i.e. from t_h+5 to t_e+1), if the input port 0 chooses output port 0 with the priority bit set in the header and no other input port has its priority bit set. The value on dOut0 will be $dIn0^{S3}$ which is the data input that is 4 clock cycles earlier than the data output dOut0.

```
AG (dIn0[3:0]=011* dIn1[1]=0* dIn2[1]=0 * dIn3[1]=0
* state=S2 -> AXAXAXAXAX (dOut0== dIn0^S3))
(3.6.1)
```

Property 7: In state S5 (i.e. from t_h+3 to t_e-1), if input port 0 chooses output port 0 with priority bit set in the routing tag, and no other input port has its priority bit set, the value on ackOut0 will be the input of ackIn0.

```
AG (dIn0[3:0] = 0011 * dIn1[1] = 0 * dIn2[1] = 0 *
dIn3[3] = 0 * state = S2 -> AX AX AX ( ackOut0 ==
ackIn0 ))                                     (3.7.1)
```

3.3 Abstracted fabric

We did not succeed in using the original fabric to check the properties due to state space explosion To cope with the state space explosion problem, at first, we reduced the datapath of the dataswitch unit from 8 bits to 4 bits, the property checking still consume too much CPU time. We therefore reduced the datapath further to 1 bit. Because the behavior and structure of 1-bit datapath are exactly the same as those of other 7 bits, this abstraction is valid. The arbitration and acknowledgment units remained the same as the original design. The abstracted model is shown in Figure Chapter 3 .6.

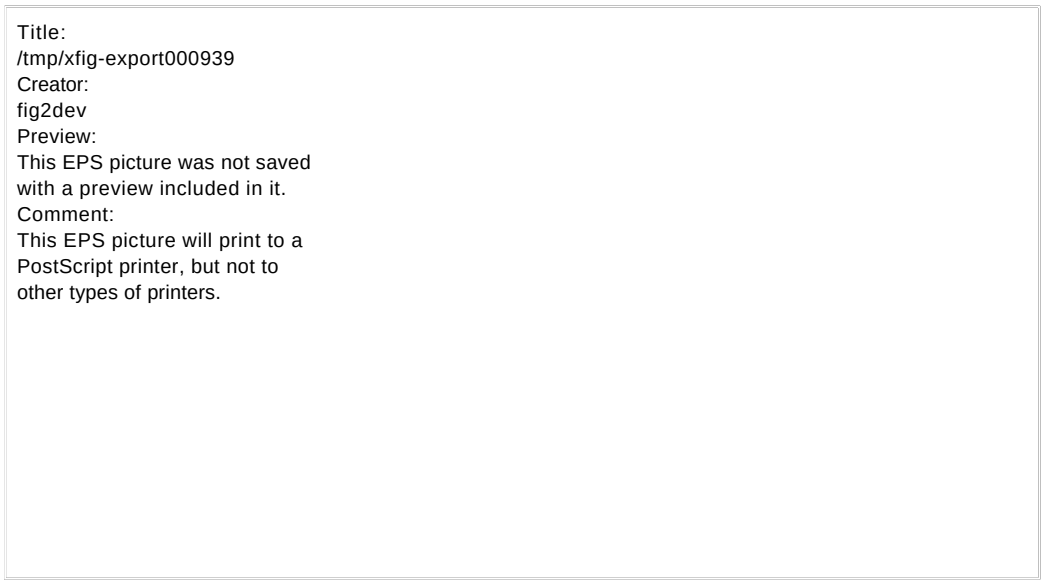


Figure Chapter 3 .6 The abstracted switch fabric

Based on this abstracted model, we checked the above seven properties which were done on SUN Sparc 20 workstation (55MHz/256MB). The CPU time (elapse time), memory usage and nodes allocated of every property checking are shown in Table Chapter 3 .1.

Properties	CPU time (seconds)	Memory (MB)	Nodes allocated (K)	# of CTL formulas

Property 1	3934	40	84, 199	1
Property 2	4551	41	90,371	1
Property 3	15	3	368	1
Property 4	3594	32	93,073	1
Property 5	833	5	28,560	1
Property 6	3679	41	79,687	1
Property 7	415	5	4,180	1

Table Chapter 3 .1 Property checking on the abstracted fabric

All the properties in Table Chapter 3 .1 were checked using the sift algorithm [5] which is a dynamic ordering algorithm. Dynamic ordering provides an optimized order that will drastically decrease the memory usage, nodes allocated, and hence decrease CPU time. We choose property 2 as an example to see the difference between the property checking with and without dynamic ordering (see Table Chapter 3 .2).

Property 2	CPU time (seconds)	Memory Usage (MB)	Nodes allocated
with dynamic ordering	4551	41	90,371
without dynamic ordering	55126	91	302,307

Table Chapter 3 .2 Dynamic ordering in property checking

3.4 Discussion on enhancement of model checking

Although we succeeded in checking all the above properties on the abstracted fabric, we found that almost all the properties are checked with unreasonable time. For example, Property 4 consumes 3594 seconds CPU time which is around 3 hours machine time. Here, we discuss several approaches we adopted to speed-up the model checking.

3.4.1 Cascade property division example

We use Property 7 as an example, and the original CTL expression of property 7 is:

```
AG (dIn0[3:0]=3 * dIn1[1] = 0 * dIn2[1] = 0 *
dIn3[1] = 0 * state = S2 -> AX AX AX (ackOut[0] ==
```

```
ackIn[0]))
```

To divide this property, we introduced the intermediate signals (variables) *xGrant[0]*, *yGrant[0]* and *outputDisable[0],* yielding the following two sub-properties:

```
sub-property1:AG (dIn0[3:0] = 3 * dIn[1] = 0 * dIn2[1] = 0 * dIn3[1]
              = 0 * state = S2 -> AX AX AX (state = S5 * xGrant[0] =
              0 * yGrant[0] = 0 * outputDisable[0] = 0))

sub-property2:AG (state = S5 * xGrant[0] = 0 * yGrant[0] *
              outputDisable[0] = 0 -> ackOut0 == ackIn0)
```

Like model checking in the abstracted fabric, we need build an environment state machine for each sub-property. For sub-property1, every input signal (variable) in its CTL expression is already in the environment of the abstracted fabric, so we can use that one directly. But for sub-property 2, some input signals (variables) like *xGrant[0]*, *yGrant[0]* and *outputDisable[0]* are not in that environment . We hence established a new environment state machine where the behaviors of the signals (variables) *xGrant[i]*, *yGrant[i]* and *outputDisable[i]* are given. Table Chapter 3 .3 gives the comparison between the property checking with cascade property division and the property checking without it. The CPU time for checking property 7 is enhanced by 41 times by cascade property division.

Property 7		CPU time (second)	Memory (MB)	Nodes allocated(K)
no cascade prop division		415	5.3	4,180
cascade property division	sub-property 1	5.1	1.9	109
	sub-property 2	4.9	1.7	79
	Total	10.0	-	-

Table Chapter 3 .3 Cascade property division in property checking

3.4.2 Parallel property division

While cascade property division introduces sequentially related intermediate variables to

divide a property, parallel property division splits a property into several parallel sub-properties without introducing any intermediate variable, and checks every sub-property by an abstracted model that is stripped from a design regularly. We use property 4 as an example.

The original CTL expression of property 4 is:

```
AG (state=S3 + state=S4 + state=S5 + state=S6 ->
dOut0[0]=0 * dOut1[0]=0 * dOut2[0]=0 * dOut3[0] = 0)
```

To verify this property, we separated it into four parallel sub-properties as follows:

```
sub-property1:AG (state = S3 + state = S4 + state = S5 + state = S6
            -> dOut0[0] = 0)

sub-property2:AG (state = S3 + state = S4 + state = S5 + state = S6
            -> dOut1[1] = 0)

sub-property3:AG (state = S3 + state = S4 + state = S5 + state = S6
            -> dOut2[2] = 0)

sub-property4:AG (state = S3 + state = S4 + state = S5 + state = S6
            -> dOut3[3] = 0)
```

For each sub-property, we established a correspondent abstracted fabric unit from the abstracted fabric. Figure Chapter 3 .7 describes the structure of this model. To verify sub-property 1, 2, 3, and 4, we used fabric unit 0, 1, 2 and 3, respectively. In Figure Chapter 3 .7, *ackOutij* denotes the acknowledgment signal in input port *i* of fabric unit *j*, so that the acknowledgment signal on input port *i* (*ackOuti*) represents the disjunction of the signals *ackOuti0*, *ackOuti1*, *ackOuti2* and *ackOuti3*. Table Chapter 3 .4 gives a comparison between model checking with parallel division and model checking without it for property 4. From this table, the CPU time for model checking has been enhanced by 73 times.

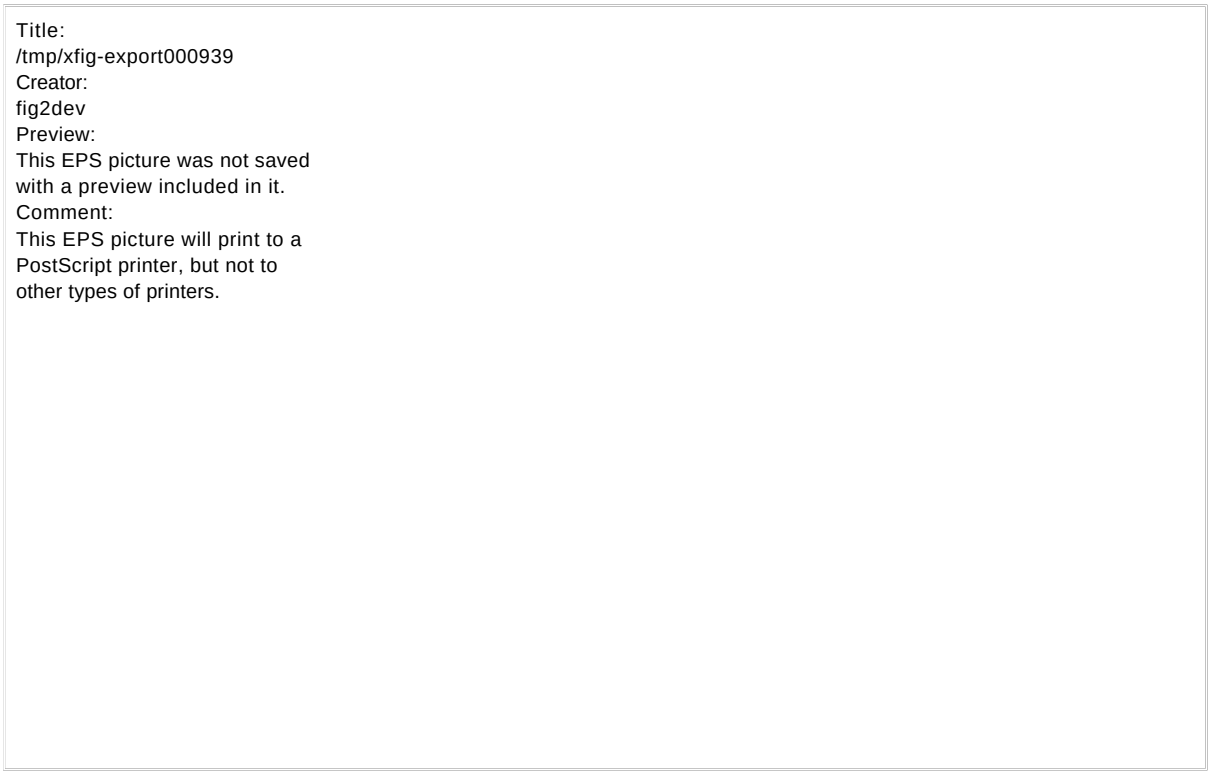


Figure Chapter 3 .7 The abstracted fabric units

Property 4		CPU time (sec.)	Memory (MB)	Nodes allocated (K)
no parallel	prop division	3593	32	93,073
parallel property division	sub-property 1	11	3	159
	sub-property 2	14	3	149
	sub-property 3	13	3	166
	sub-property 4	11	3	154
	Total	49	-	-

Table Chapter 3 .4 Parallel property division in Property 4

3.4.3 Latches Reduction

In a digital design, latches introduce states, so latches have a very obvious influence on the speed of model checking. Reducing the number of latches in a model will greatly speed-up

model checking. Table Chapter 3 .5 collects the number of latches among the original fabric, the abstracted fabric and one abstracted fabric unit (all including some latches used for the environment).

	Original fabric	Abstracted Fabric	Abstract Fabric Unit
# of latches	210	85	54

Table Chapter 3 .5 The number of latches among different models

From our experiments, we found that model checking is almost impossible using the original fabric, and it was very slow using the abstracted model as shown in Table Chapter 3 .1. However, using the abstracted fabric unit, acceptable time of model checking was achieved. Through more experiments of property checking in VIS, we found that the model which has around 50 latches can be easily verified by model checking in VIS using SUN SPARC 20 (75 MHz/256MB). This result will guide us to build abstracted models.

Many designs use latches to pause data for 1 clock cycle in its primary inputs and outputs. Since these latches are directly connected to input and output ports, ignoring these latches will not influence the state transitions within the design. However, just reducing the latches will speed-up property checking dramatically. Table Chapter 3 .6 shows that the CPU time of checking property 2 has been enhanced by nearly 100 times by using latch reduction. In this example, the data output latches that are used to delay output data for 1 clock cycle were reduced. When we ignore latches used to pause input or output data for some clock cycles, the timing behavior of a design will be changed, so we must do some corresponding updates of a property to match the model with latch reduction. We use property 2 as an example to illustrate this. The original property 2 is “The data bytes in a cell are transferred from input port 0 to output port 0 sequentially with 4 clock cycles delay”. Since the data output latches are reduced, the updated property 2 should be “the data bytes in a cell are transferred from input port 0 to output port 0 sequentially with 3 clock cycles delay”.

Property 2	CPU time (seconds)	Memory (MB)	Nodes allocated(K)
no latch reduction	4551	4	90,371
latch reduction	23	3	235

Table Chapter 3 .6 Latches reduction in model checking

3.4.4 Concluding Results on Model Checking

All model checking results in Table Chapter 3 .1 were obtained by using the abstracted fabric and dynamic ordering, and some results were not satisfactory since they took a lot of CPU time. By using cascade property division, parallel property division and latch reduction, we got satisfactory results (see Table Chapter 3 .7). For the properties listed in this chapter, one of these enhancement approaches was enough for the model checking. But for more complicated properties, we may apply the combination of several approaches.

Properties	Property1	Property2	Property3	Property4	Property5	Property6	Property7
CPU time (sec.)	28	46	15	49	73	34	10
Enhancement Approach	latch reduction	latch reduction	-	parallel division	parallel division	latch reduction	cascade division

Table Chapter 3 .7 Summary of enhanced property checking of the fabric

3.5 Error detection in model checking

No errors were discovered in the above model checking. For experimental purposes, however, we injected several design errors into the implementation: (1) We exchanged the inputs to the JK Latch that produces the *outputDisable* signal. This prevented the circuit from resetting. (2) We used the priority information of the input port 0 to control the input port 2. (3) We used an AND gate instead of an OR gate within the acknowledgment unit producing a faulty ackOut0 signal. (4) We used erroneously the same select signal to control output port 0 and 1. (5) We reduced a set of delay registers for all the data input ports. These five errors were detected by model checking and VIS generated

counterexamples that exhibit the incorrect behavior of the corresponding signals. Experimental results are reported in Table Chapter 3 .8, where the CPU time includes the time for model checking and counterexample generation.

Experiments	Property used for error detection	CPU time (seconds)	Memory usage (MB)	Nodes allocated (K)
Error 1	Property 4	83	4	1, 408
Error 2	Property 6	49	3	251
Error 3	Property 7	15	1	85
Error 4	Property 3	34	4	199
Error 5	Property 2	17	1	54

Table Chapter 3 .8 Error detection in property checking

3.6 Verification of the entire Fairisle ATM Switch

The Fairisle ATM switch consists of three types of components: input port controllers, output port controllers and a switch fabric (Figure Chapter 3 .1).

The input port controllers receive ATM cells from transmission lines, store them in queues, dequeue them, append fabric header and output header, and then transfer the cells into the switch fabric. An ATM cell consists of 48 data bytes plus a 4-byte header. The input port controllers also receive acknowledgment signals from the fabric and decide whether to send new data, to retransmit previous cell, or to stop sending data. The switch fabric transfers data cells from input port controllers to the output port controllers and passes the acknowledgment signals from the output port controllers to the input port controllers according to the fabric header. If cells on common destination clash, the switch fabric arbitrates using round-robin allocation. The output port controllers decide whether to transfer data to transmission lines or loop back data to the input port controllers according to the output header. They also detect errors in received cells, and send the acknowledgment signals to the switch fabric. The port controllers and switch fabric all use

the same clock, and they also use a higher-level cell frame clock: the *frameStart* signal (*fs*). It ensures that the port controllers inject data cells into the fabric synchronously so that the fabric headers arrive at the same time.

The switch fabric in this section is the same as the fabric we verified in the previous sections, but the port controller is an abstracted module.

The port controller consists of two input FIFOs, a receiver circuit, a network of queues containing pointers (addresses) to memory in which the incoming cells are stored before being forwarded to the switch fabric, a dispatcher-scheduler, and a transmitter circuitry. An arbitration circuit controls access to the shared memory used to store both the cells and the pointer queues.

In the FIFOs, the bytes are made of 9 bits where the 9th bit indicates the start of a cell (*frameStart* signal). The priority (high, low) of a cell is indicated by a bit in the header. This is checked by the dispatcher to insert the cell pointer to the appropriate queue. The two input FIFOs (FIFO_I, FIFO_L) are queues of bytes used to synchronize the switch with the external transmission lines, one containing bytes received and the other for loop back within the controller.

.

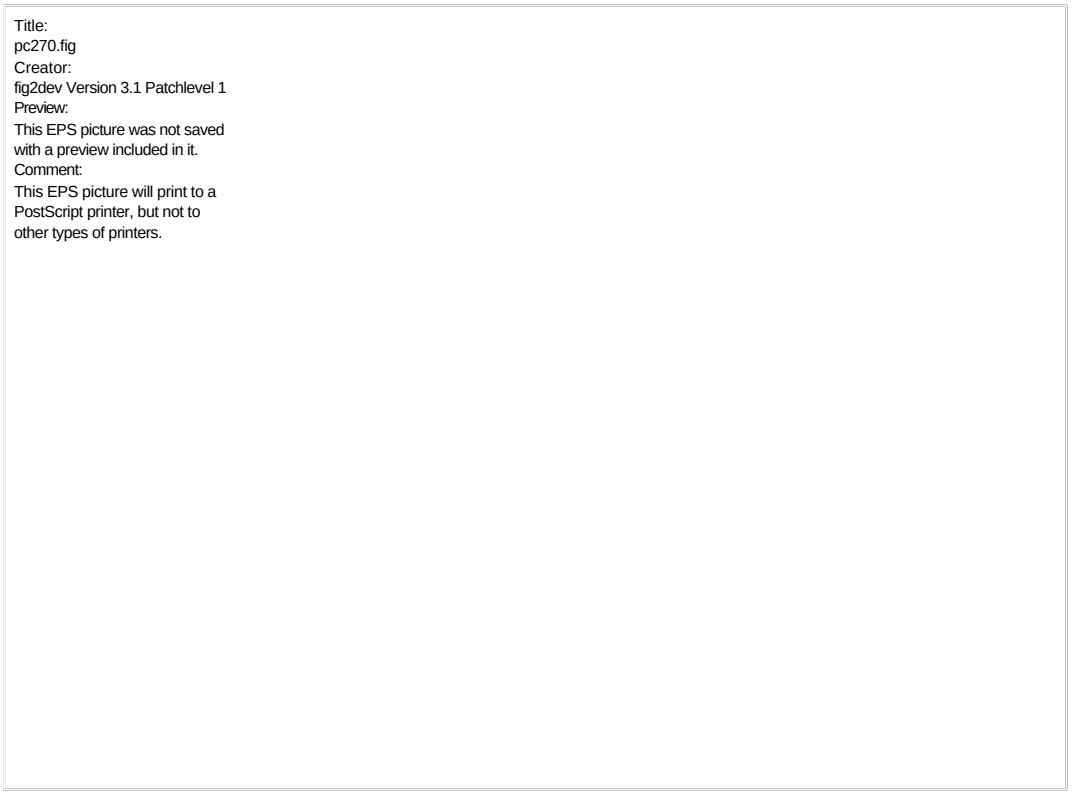


Figure Chapter 3 .8 Fairisle input and output port controller

FIFO_O contains bytes to output on the transmission line. The queues contain pointers (addresses) to memory, where ATM cells are or will be inserted. When a cell enters the port controller, a pointer from the free pointer queue (F) is allocated by the receiver. The cell is written 4 bytes at a time to the memory while the pointer is inserted into the receiver queue (R). During the extraction, if the FIFO becomes empty or the start of a new cell is detected, the current cell is dropped and the receiver starts the extraction of the newly detected cell. The dispatcher detects the presence of a cell in the receive queue, updates the cell header, adds the fabric and the output headers, and transfers the pointers to the appropriate priority queue H or L. The scheduler transfers cell pointers from the queues H, L to the transmission queue (T), giving priority to cells in the H queue. The transmitter is in charge of the transfer of the ATM cells from the transmit queue to the

switch fabric, one byte at a time. After a cell is successfully transmitted to the fabric, its pointer is returned to the free pointer queue. The arbiter controls the access to the shared memory between the receiver, the dispatcher, the scheduler and the transmitter, giving always priority to the transmitter. Although we use independent queue models, we assume that they are implemented in the shared memory, hence only one queue or cell word can be accessed at a time, under the control of the arbiter.

3.6.1 ATM Switch Modeling

The port controllers contain around 1MB of memory, 2KB FIFO buffer and 600 registers (latches) used for addressing and control. The switch fabric contains 210 latches. Both port controllers and fabric are thus too large to directly perform model checking on any of them. Therefore, we introduced a number of abstractions and reductions for both the port controllers and fabric.

In the port controller model:

1. We abstracted away the memory by using a cell memory interface while keeping pointers in separate queues represented by functionally equivalent models.
2. FIFOs were reduced to primary inputs where only the socket bits are consulted to distinguish the beginning of a cell from a cell body.
3. The queue length was abstracted for certain safety properties by using queue models with no content but only status bits that nondeterministically indicate whether the queue is full or empty, or neither full nor empty.

Based on the above abstractions and reductions, we obtained a model with 43 latches for each port controller, and thus 172 latches in total for the four port controllers.

Through the above reductions and abstractions and using the abstracted fabric of section 3.3, the total number of latches in the entire ATM switch model was reduced to about 300.

3.6.2 Property Description and Deduction

In order to capture the behavior of the whole switch including both the port controllers and the switch fabric, we established a number of global properties for the entire ATM switch. However, because of state space explosion, we did not succeed in directly verifying any of these properties on the whole switch model. We used property division to deduce a global property by several local properties. For illustration purposes, in following we present three example global properties and explain how we applied the division.

In following sections, Q_X represents a queue where X = F, R, L, H, and T denote the queues in Figure Chapter 3 .8. Furthermore, empty and full indicate the status of a queue, head expresses the fabric header and *deq* stands for dequeue. We use *i, j, x, y, z* to represent a port number which could be 0 (00), 1 (01), 2 (10) or 3 (11), where *i, x, y, z* are disjoint values (for different input port numbers). *Din* will be used to denote a data input value, while *dIn* and *dOut* will be used for data input and output signals of the fabric, respectively.

Property 8: If data bytes in a cell are allocated in an input port controller, they will eventually reach an output port of the fabric. This is expressed as:

```
AG ( Q_F.i = Din.address. i * Q_F.deq.i = 1 -> AF
(dOut.j = Din.i ) )
(3.8.1)
```

Property 8 could be subdivided into two sub-properties,

```
sub-property1:AG (Q_F.i = Din.address.i * Q_F.deq.i= 1-> AF (Q_T.i
              = Din.address.i)

sub-property2:AG ( dIn.i = Din.i * state = S3 -> AF ( state = S7 *
              dOut.j = Din.i))
```

sub-property 1 can be verified in the port controller model, and it means that if a cell in any input port controller is allocated, its address will eventually reach the output of the same input port controller through the receiver, the dispatcher-scheduler and the

transmit queue. In sub-property 1, *Din.address.i* stands for a pointer value (address of a cell) in input port controller *i*; *Q_F.deq.i = 1* represents the allocation of the data bytes in FIFO_I buffer. Sub-property 1 includes 4 CTL expressions. Since the four input port controllers are identical, we only need to check one CTL expression on one input port controller model.

While sub-property 1 can verified in port controller model, the sub-property 2 is very similar to formula (3.1.1), and can be verified in the fabric with its environment. Sub-property 2 means that the data bytes in a cell are eventually transferred from input port *i* into output port *j* with four clock cycles delay. In sub-property 2, S3 and S7 indicate the states where data cells are sent to the input port and received by output port of fabric, respectively; *dIn.i = Din.*i and *dOut.j = Din.i* means that input port *i* sends the data cell *Din.i* and the output port *j* receives *Din.i*, respectively. Since it is 4 clock cycles from state S3 to S7, the property also verifies the latency of the fabric. Since the fabric has 4 input/output ports, the CTL expressions of sub-property 2 includes 16 possible combinational CTL expressions.

When verifying sub-property 1 and sub-property 2, we gave the correspondent fairness constrain to avoid infinite idle inputs sequence, and this is similar to the fairness constraint we gave in Section 3.2.2.

Since the environment of the fabric is based on the port controller model and both of sub-properties are ACTL expression, we can deduce to the property 8 by the two sub-properties according to property division.

Property 9: If there is only one cell in the high priority queue among the four input port controllers, this cell will be transferred to the output port of the fabric with priority (8 clock cycles after it is dispatched). i.e. in CTL:

```
AG(Q_H.empty.i = 0 * Q_L.empty.i = 0 * Q_H.empty.x =
1 * Q_L.empty.x = 0 *Q_H.empty.y = 1 * Q_L.empty.y =
0 * Q_H.empty.z = 1 * Q_L.empty.z = 0 *
```

```
Din.i.H.head(3:2) = j * Dispat.state.i = D1*
Dispat.state.x = D1* Dispat.state.y = D1 *
Dispat.state.z = D1 -> AX AX AX AX AX AX AX (dOut.j
= Din.i.H);
```

This property is subdivided into two sub-properties,

```
sub-property1:AG (Q_H.empty.i = 0 * Q_L.empty.i = 0 * Dispat.state.i
              = D1 -> AX(Q_H.deq.i = 1   *   Q_L.deq.i   =   0   *
              Din.i.H.head(1) = 1)

             AG (Q_H.empty.x = 1 * Q_L.empty.x = 0 * Dispat.state.x
             =  D1  ->  AX(Q_H.deq.x  =  0  *  Q_L.deq.x  =  1  *
             Din.x.L.head(1)=0)
             AG (Q_H.empty.y = 1 * Q_L.empty.y = 0 * Dispat.state.y
             =  D1  ->  AX(Q_H.deq.y  =  0  *  Q_L.deq.y  =  1*
             Din.y.L.head(1) = 0)

             AG (Q_H.empty.z = 1 * Q_L.empty.z = 0 * Dispat.state.x
             =  D1  ->  AX(Q_H.deq.z  =  0  *  Q_L.deq.z  =  1*
             Din.z.L.head(1) = 0)

sub-property2:AG  (Din.head.i  [3:0]  =  j11  *  Din.head.x[1]=  0  *
              Din.head.y[1] = 0 * Din.head.z[1] = 0   * state = S2
              -> AX AX AX AX AX (dOut.j = Din.i))
```

In above CTL expression, *Din.i.H* means the value of the cell in high priority of input port controller *i. Din.head.i, Din.head.x, Din.head.y* and *Din.head.z* in Sub-property2 (which are the cells in the input of the fabric), are equivalent to *Din.i.H.head, Din.x.L.head, Din.y.L.head* and *Din.z.L.head* in Sub-property1 (which are the cells in the output of their correspondent input port controllers). Similarly, *Din.i* is equivalent to *Din.i.H.*

sub-property 1 can be verified in the port controller model, and it means that if there is a cell with high priority (in high priority queue) in input port controller i and there are

no cells with priority in the other three input port controllers (x, y, z), the cell with priority will be transferred to the transmit queue with priority in input port controller *i*, and its priority bit in its fabric header is "1"; the cells without priority will be transferred to their correspondent transmit queue with "0" as its priority bit in the fabric header in the other three input port controllers. In sub-property 1, *Dispat.state.k = D1* (*k = i, x, y, z*) indicates that input port controller k is in its dispatch state (D1). *Din.i.H.head[1] = 1* means that the priority bit for the cell in output of port controller *i* is '1'. *Din.x.L.head[1] = 0, Din.y.L.head[1] = 0* and *Din.z.L.head[1] = 0* mean that the priority bit of the cells in the outputs of the other three input port controllers are "0"s. Sub-property 1 includes 16 possible CTL expressions depending on the value *i, x, y* and *z*, however, since the four port controllers are identical, we only need to check 2 CTL expressions.

Sub-property 2 can be verified in the fabric with its environment, and it means that a data cell with priority at input port *i* of the fabric will transfer to its destined output port *j* with 4 clock cycles delay. In sub-property 2, *Din.head.i[3:0] = j11* represents that the cell in input port *i* of the fabric has a priority and its destination is output port *j*. *Din.head.x[1]= 0, Din.head.y[1]= 0* and *Din.head.z[1]= 0* indicate that the priority bits of the cells in the other three input ports of the fabric are "0". Since the data bytes are transferred to the output port *j* 5 clock cycles after state S2 which means 4 clock cycles after the data bytes transmit (state S3) (Figure Chapter 3 .5).

For similar reasons as the sub-properties of Property 4.8, We could deduce the property 4.9 from the two sub-properties according to compositional reasoning and property division.

Property10: An acknowledge signal associated with a cell with priority will be transferred from an output port of the fabric to the destined input port according to the fabric header; once the correspondent input port controller receives positive acknowledge signal, it will transmit data continually, otherwise it will stop sending data. We express this in CTL as:

```
Din.head.i [3:0] = j11 * Din.head.x[1] = 0 *
Din.head.y[1] = 0 * Din.head.z[1] = 0 * (state = S5
+ state = S6) * ackIn.j = 0 * transmit_state.i= send
-> AX (transmit_state.i = stop))
(3.10.1)

Din.head.i [3:0] = j11 * Din.head.x[1] = 0 *
Din.head.y[1] = 0 * Din.head.z[1] = 0 * (state = S5
+ state = S6) * ackIn.j = 1* transmit_state.i = send
-> AX (transmit_state.i = send ))
(3.10.2)
```

This property could be subdivided into two sub-properties,

```
sub-property1:AG ( Din.head.i [3:0] = j11 * Din.head.x [1] = 0 *
             Din.head.y[1] = 0 * Din.head.z[1] = 0 * ackIn.j = 0 *
             ( state = S5 + state = S6) -> ackOut.i = 0)

            AG ( Din.head.i [3:0] = j11 * Din.head.x [1] = 0 *
            Din.head.y[1] = 0 * Din.head.z[1] = 0 * ackIn.j = 1 *
            (state = S5 + state = S6) -> ackOut.i = 1)


sub-property2:AG (ackOut.i = 0 * (transmit_state.i = send) ->
               AX(transmit_state= stop))
             AG (ackOut.i= 1 * (transmit_state.i = send) ->
                AX(transmit_state= send))
```

Sub-property 1 can be verified in the switch fabric, and it means that an acknowledge signal associated with a cell with priority could be transferred from the output port of the fabric to the input port of the fabric according to the fabric header. In sub-property 1, *ackIn.j* and *ackOut.i* signals denote the acknowledgment signal into the output port *j* of the fabric and the acknowledgment signal out of the input port *i* of the fabric,

respectively. S5 and S6 are the states where acknowledgment signals are transferred. This includes 32 combinational CTL expression.

Sub-property 2 can be verified in the port controller, and it means that a negative acknowledge implies that the port controller stops sending data, and a positive acknowledge will make the port controller transmit data continually. In sub-property 2, *transmit_state.i = send* and *transmit_state.i = stop* express that the input port controller *i* is in the send and the stop state, respectively. *ackOut.i = 0* and *ackOut.i = 1* indicate the negative and positive acknowledgment, respectively..

If we use the assumption of sub-property 1 *(Din.head.i [3:0] = j11 * Din.head.x [1] = 0 * Din.head.y[1] = 0 * Din.head.z[1] = 0 * ackIn.j = 0 * (state = S5 + state = S6)* and *Din.head.i [3:0] = j11 * Din.head.x [1] = 0 * Din.head.y[1] = 0 * Din.head.z[1] = 0 * ackIn.j = 1 * (state = S5 + state = S6))* to substitute the assumption of sub-property 2 *(ackOut.i = 0 and ackOut. i = 1)*, we will get Property 10, hence Property 10 is valid.

3.7 Summary

In this chapter, we described our verification on Fairisle ATM switch fabric in detail. By this example, we present our experience on reducing the scale of the verification target and build up environments. In addition, we introduce three methods to enhance model checking, the first two methods (cascade property division and parallel property division) were introduced in Chapter 2, the other method (latch reduction) speeds up model checking by reducing the input or output latches. Furthermore, we apply property division to verify three properties on the entire Fairisle ATM switch which consists of Fairisle switching fabric and Fairisle port controller model.

Chapter 4

Verification of ATM Port Controller

The null port controller (Figure Chapter 4 .1) is a real design from Cambridge University and it is a part of Fairisle ATM switch. Since it does only VCI mapping and FIFO queuing, it is called the null port controller. In the original design, a Xilinx chip controls all its functions, and it uses triple ported DRAMs to look up the new VCI, and uses a FIFO to

do speed matching with the transmission board. The null port controller connects with Fairisle ATM switch fabric, and transmits ATM cells to the fabric and receives acknowledgment signals from the fabric. Both the null port controller and the switch fabric use the same *framestart* signal to synchronize the behavior.

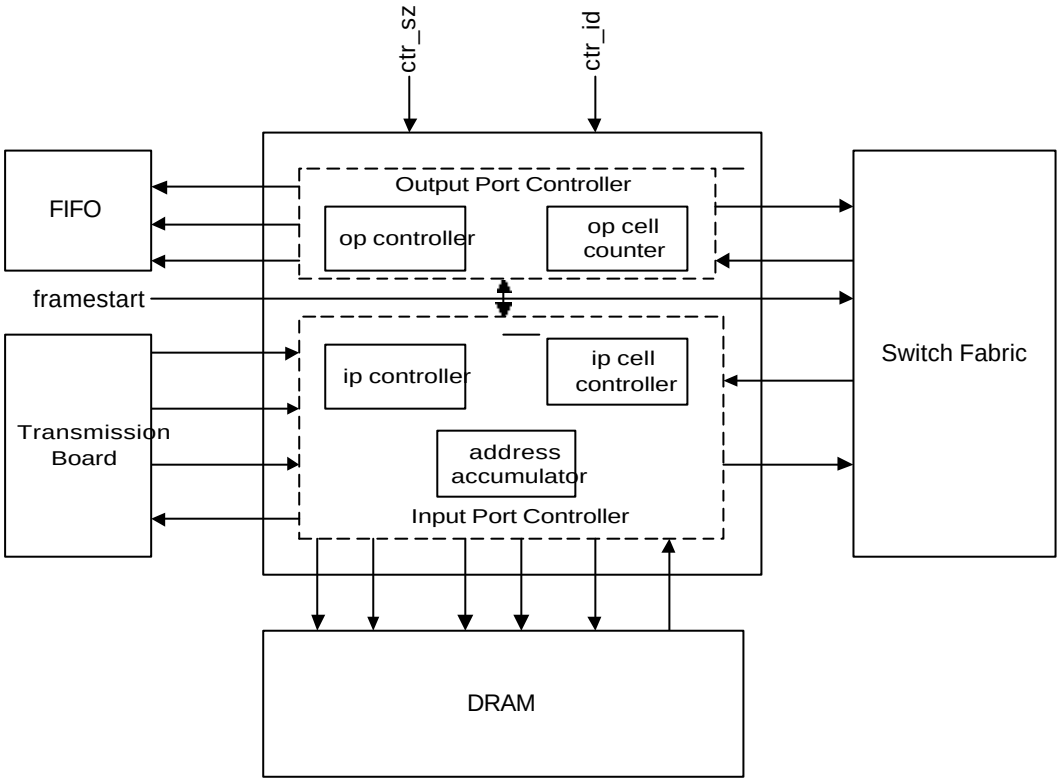


Figure Chapter 4 .1 The structure of the null port controller

The null port controller consists of an input port controller and an output port controller. It is able to transmit one cell every 128 clock cycle. If the clock frequency is 20 MHz, the maximum bit rate is 80 MHz. There are no service classification, no scheduling or traffic shaping, no monitoring and policing in this port controller, but we can give a priority to an ATM cell, and this is done by preloading the priority bit into the memory. The priority bit will be used for "arbitration" in the switch fabric.

Figure Chapter 4 .2 is the format of an ATM cell. Although the actual data is 48 bytes per cell, each cell is assigned 64 bytes in the memory. Except the 48 bytes data, the receiving cell has 4 more bytes: 2 VCI bytes and 2 FAS bytes; the transmission cell has 6 more bytes: 1 Fabric Routing Byte (FRB), 1 Port controller Routing Byte (PRB), 2 VCI bytes and 2 FAS bytes. Since each cell consumes 64 bytes memory, the memory, which is 256k x 8 bit, can contain 4096 ATM cells. This means that the port controller supports 4096 connections. To prevent the two cells with the same VCI arriving at the memory consecutively, only one cell is allowed in the memory.

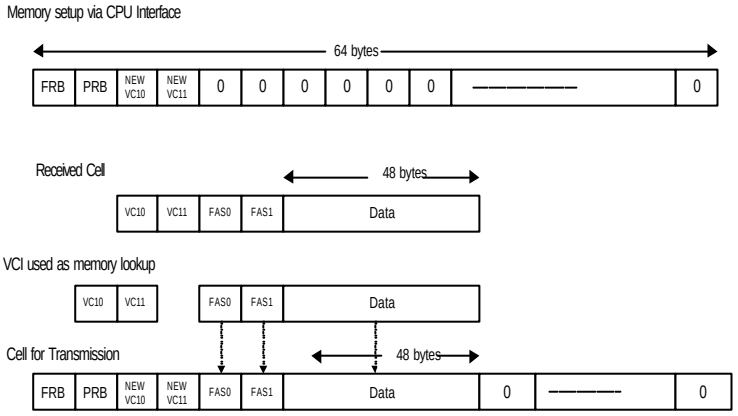


Figure Chapter 4 .2 The format of received cell and cell for transmission

We got some documents and structural codes of the null port controller from Cambridge University, but those were not complete. In this work, we implemented the null port controller RTL according to its documents. Our implementation is based on the original design, and the main difference is that we used SRAM instead of DRAM to store the cell because the current SRAM is much cheaper than before, and it is being used in ATM hardware designs. Our RTL description of the null port controller is written in Verilog. To verify the port controller in VIS, we establish an environment, and define a number of CTL properties. In following sections, we will introduce the behavior and structure of the port controller in Section 4.1 and Section 4.2. Section 4.3 describes the properties of the null

port controller. Section 4.4 illustrates the three methods on expressing a property in CTL, and Section 4.5 summarizes the chapter.

4.1 Behavior of the Null Port Controller

The null port controller consists of input port controller and output port controller. The input port controller receives data cell from the transmission board, and writes data cells into memory according to their VCIs. In addition, the input port controller reads the data cell out of the memory and transmit into the fabric. Once it receives the positive acknowledgement signal, the input port controller will continue transmitting data; otherwise, it will stop sending data. The output port controller receives a data cell from the fabric, and send an acknowledgment signal back to the fabric. If the output port controller receives a data cell, it gives a positive acknowledgment signal; otherwise, it sends a negative acknowledgment.

The input and output port controller synchronize the behavior by clock and *framestart* signal whose period is 64 clock cycles. The input port controller always monitors *framestart* signal. After *framestart* signal is asserted, if the memory is empty and the transmission board read request is asserted, the input port controller will assert a write enable signal to the transmission board. But the data are transferred into the memory only after the input port controller receives the start of cell (SOC) signal. After the SOC signal is received, the input port controller will latch the first followed two bytes which are the old VCIs of a data cell. Table Chapter 4 .1 is the conversion between the old VCI of a data cell and its memory location, where *r* denotes row address and *c* means column address. The decimal digit indicates the position in the binary address (e.g. *r4* means bit 4 of a row address). The whole VCI0 byte and bits 4 to 7 of VCI1 byte will become the row address and the most 3 significant bits of the column address, and bits 3 to 0 of VCI1 byte are unused in the conversion.

VCI 0 byte	0	1	2	3	4	5	6	7
addr_r	r1	r2	r3	r4	r5	r6	r7	r8

VCI 1 byte	0	1	2	3	4	5	6	7
addr_r	-	-	-	-	c6	c7	c8	r0

Table Chapter 4 .1 VCI to memory location conversion

On the other hand, when *framestart* signal is asserted and the input port controller has a cell to send, the input port controller will read the data cell from the memory into the fabric. After a certain clock cycles, if the input port controller receives the positive acknowledgment signal through the switch fabric, it will continue sending the data cell; otherwise, it will stop the transmission.

While the input port controller receives data from the transmission board and transmits the data into the fabric, the output port controller receives data from the switch fabric and gives the acknowledgment signal to the input port controller through th e switch fabric. After *framestart* signal is asserted, the output port controller will detect the active bit of port controller header. If the active bit is asserted, the output port controller will generate the positive acknowledgment signal which will be transmitted into the input port controller through the fabric; otherwise, the output port controller will generate the negative acknowledgment signal. If the output port controller receives a data cell, it will write the data into the output FIFO, and the first byte of the data, which is Port controller Routing Byte (PRB), will be stripped and one SOC signal will be generated and instead into the output FIFO by the output port controller.

4.2 Structure of the null port controller

Figure Chapter 4 .3 shows the structure of the null port controller. It consists of an input port controller and an output port controller. The input port controller processes the signals from the transmission board, the memory and the fabric. The output port controller interfaces with the signals from the fabric and the output FIFO.

The input port controller consists of an *ip controller*, an *ip cell counter* and an *address accumulator*. The *ip controller*, which coordinates the *ip cell counter* and the

address accumulator, controls the data reception, transmission, memory read and write. The *ip cell count* is an up counter which increments by 1 per data byte transfer, and so does the *address accumulator*. In Figure Chapter 4 .3, the signals *ip_mem_data, ip_mem_wr_en, ip_mem_addr_r, ip_mem_addr_c, ip_mem_rd_req* and *mem_ip_data* are the interface signals between the input port controller and the cell memory. The signals *ip_mem_data* and *mem_ip_data* mean the data outputs from the input port controller to the cell memory and the data inputs from the cell memory to the input port controller, respectively. Both of the signals have 8-bit bus width. The signals *ip_mem_wr_en* and *ip_mem_rd_req* are the memory write enable and memory read request signal, respectively. The memory row and column addresses are provided by *ip_mem_addr_r* and *ip_mem_addr_c*. In the interface between the input port controller and transmission board, *rx_ip_data* is an 8-bit data bus which is the data inputs from the transmission board to the input port controller. The signals *rx_rd_req* and *rx_ip_soc* indicate cell available in the transmission board and the start of a cell, respectively. The signal *ip_rx_wr_en* demonstrates whether the input port controller is able to accept a cell or not. In addition, the input port controller has an interface with the fabric. The *ip_fab_data* is a 8-bit data bus which transfers data from the input port controller to the fabric. The *fab_ip_ack* is the acknowledgment signal which indicates whether the current cell succeeds in transferring to the destined output port controller.

The output port controller consists of an *op controller* and an *op cell counter*. The *op controller* generates the acknowledgment and SOC signals, and controls *op cell counter*. The *op cell counter,* which is very similar to the *ip cell counter*, increments by one per data transfer. In Figure Chapter 4 .3, *op_fab_ack* and *fab_op_data* are the signals in the interface between the output port controller and the fabric. *fab_op_data* is an 8-bit data bus from the fabric to the output port controller. *fab_op_ack* is acknowledgment signal generated by the output port controller. In addition, there are *op_fifo_data, op_fifo_wr_en* and *op_fifo_soc* signals between the output port controller and the output FIFO. The *op_fifo_data* is an 8-bit data path from the output port

controller to the FIFO. The *op_fifo_wr_en* is the write enable signal for the output FIFO. The signal *op_fifo_soc* indicates the start of a cell, and it is asserted before the first byte data transfer.

There are two control registers (*ctr_id* and *ctr_sz*) and one status register (*ip_empty*) inside the port controller. *ctr_id* is an input disable register. When *ctr_id* asserts, all the inputs are disable. During the period of *ctr_id =1*, the microprocessor could pre-load the new VCIs, FRB and PRB into the memory. The register *ctr_sz* is for debugging purpose. When *ctr_sz* is high, the memory address of the incoming cell is not based on the old VCI values, instead, the row address of the incoming cell is 0 and the column address is from 0 to 63. The register *ip_empty* is used to indicate the status of the port controller. When it is asserted, the input port controller can accept a cell from the transmission board; otherwise, a cell can be transmitted into the fabric from the input port controller.

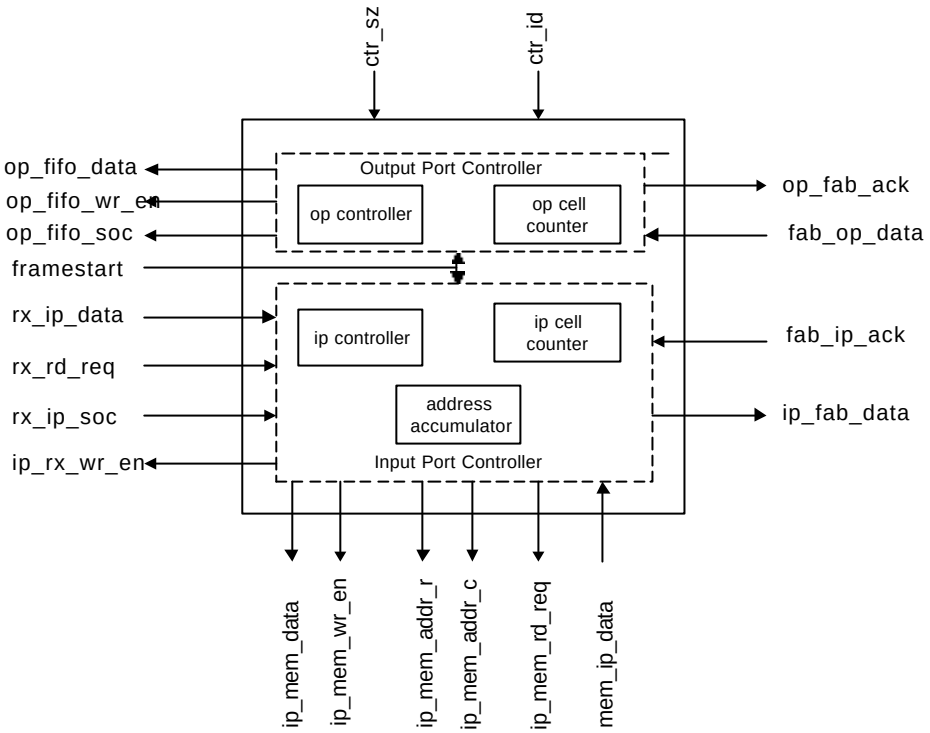


Figure Chapter 4 .3 The structure of the null port controller

4.3 Properties of the Null Port Controller

Before performing model checking, we must figure out the necessary properties of the null port controller, and this is similar to create some scenarios before the simulation. The null port controller appends the new VCIs, FRB and PRB onto ATM cells and transfers them into the fabric, so its major properties could include registers reset, memory addressing, cell counting, data and acknowledgment transfer. According to these, we give the following seven properties.

Property 1: The null port controller will be reset properly when either the null port controller reset signal (*npc_rst_n*) or the null port controller input disable

signal (*ctr_id*) is asserted.

Property 2: When the input port controller can accept a cell, the transmission board has a cell to send and the input port controller is in debugging state (*ctr_sz = 1*), the address will be set up properly, and data will be transferred correctly.

Property 3: When the input port controller can accept a cell, the transmission board has a cell to send and the input port controller is in the normal operation state (*ctr_sz=0*), the memory address will be set up and incremented properly, and data will be transferred correctly.

Property 4: When the input port controller has a cell to send, it will transfer data and set up the address properly. If the input port controller does not receive a positive acknowledgment signal, it will stop sending data, otherwise, it will send data continually.

Property 5: An ATM cell can be transferred from the transmission board to the fabric coherently.

Property 6: The cell memory cannot be read and write at the same time.

Property 7: The output port controller will send an acknowledgment signal after it detects an incoming cell.

The seven properties cover the main features of the null port controller. We will use Property 3 as an example to illustrate how to build an environment and write a property in CTL.

4.4 Examples on Property Description in CTL

In Chapter 2, we have introduced three methods on describing a property in CTL: Environment Modification, Internal Signal Usage and Counter Reduction. In this section, we will illustrate how to use these methods by verifying Property 3. From the comparison the performance of model checking using different methods, we demonstrate advantages and

disadvantages of each method.

Property 3: When the null port controller can accept a cell, the transceiver board has a cell to send and the null port controller is in normal operation state (ctr_sz=0), the memory address will be set up and incremented properly, and data will be transferred correctly.

- **Environment Modification**

Property 3 has the following five assumptions:

1) The input null port controller can accept a cell, and it can be expressed as "*ip_empty = 1*";
2) Transmission board has a cell to send, and it can be expressed as "*rx_ip_rd_req = 1*";
3) The null port controller is in normal operation state, and it can be expressed as "*ctr_sz = 0*";
4) The input null port controller will accept the data from the transmission board after *rx_ip_soc* is asserted;
5) The null port controller is not in reset or input disable operation, and it can be expressed as "*npc_rst_n = 1*" and "*ctr_id = 0*".

The input null port controller detects *ip_empty*, *rx_ip_rd_req* and *ctr_sz* signals after *framestart* is asserted. If these three signals are satisfied with the above assumptions, the null port controller will start monitoring *rx_ip_soc* in the following clock cycles. The null port controller will start transferring data from the transmission board after it detects the asserted *rx_ip_soc*. Using model checking, we need to express the logic relations between assumptions and conclusions in CTL. For example, if we express the sub-property which is "If the null port controller is in normal operation, the transmission board has a cell to send and the input port controller can accept a cell, the first byte of VCI value will become the Bit 1 to Bit 8 of the memory row address 2 clock cycles after *rx_ip_soc*

asserts.", we hope that the CTL formula can be expressed as the following.

```
AG (ip_empty=1 * rx_ip_rd_req=1 * ctr_sz=0 *
npc_rst_n = 1 *  rx_ip_soc=1  -> AX AX
ip_mem_addr_r[8:1] == rx_ip_data)
```

However, the above CTL expression is not correct because the first four assumptions (ip_empty = 1, rx_ip_rd_req = 1, ctr_sz = 0 and npc_rst_n = 1) do not happen at the same state as the fourth assumption (rx_ip_soc=1). Actually, we cannot express such property directly in CTL. Therefore, we apply Environment Modification method which is to put the assumptions into the environment.

As to the environment, because the null port controller has a cyclic period synchronized by *framestart* signal whose period is 64 clock cycles, we could establish our environment which has 64 states.

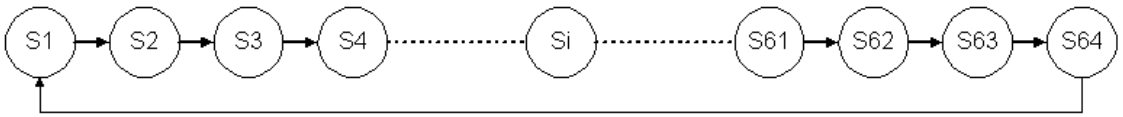


Figure Chapter 4 .4 The state transition diagram in the environment of the null port controller

Figure Chapter 4 .4 is the state transition diagram of the null port controller environment. The null port controller has some cyclic behavior so that we can use these specific states to express its specification. In Figure Chapter 4 .4, S1 denotes the cycle that *framestart* is asserted. The behavior of the null port controller can be divided into two parts: one is the data transmission which includes that the data is transferred from the input port controller to the fabric and from output port controller to the output FIFO, and the acknowledgment signal processing is also included in the data transmission process; the other is that the data is transferred from the transmission board to the memory of the null port controller. So the null port controller will have different behavior at each environment state between data transmission and data reception process.

If a cell is waiting to be transmitted in the input port controller memory, S2

denotes that the input port controller is going to provide the address to the memory. In S3, the memory address and memory read enable signals are given to the memory. S4, S5, S6, S7 and S8 denotes that the first five bytes are transferred from the input port controller to the fabric. In S9, the input port controller will detect the acknowledge signal (*fab_ip_ack*) it receives. If *fab_ip_ack* is asserted, S10 to S57 will be the states that the input port controller transfers the rest of 48 bytes, and the input port controller will be in "idle" state from S58 to S64; otherwise, the input port controller will stop sending data, S10 to S64 will be in "idle" state. At the mean time, the output port controller will detect the active bit of PRB at S9, if the active bit is asserted, the output port controller will send an positive acknowledgement signal to the fabric, and the fabric will pass it to the input port controller immediately. S10 to S61 will be the states that the input port controller forwards the rest of 52 bytes data cell to the output FIFO. If the active bit is de-asserted, the output port controller will always be in "idle" state in S1 to S64.

On the other hand, if the memory is empty, the null port controller will detect *rx_ip_soc* signal. Once *rx_ip_soc* signal is asserted, the next state will be the state that the transmission board transfers the first data byte (VCI) to the input port controller. Let's assume *rx_ip_soc* asserted in S3, *ip_rx_wr_en* will be asserted at S4, the first two bytes (two VCI bytes) are transferred to *ip_mem_addr* in S5 and S6. S7 to S56 will be the states that the null port controller transfers the rest of 50 bytes ATM cell to the memory. During these periods, the *ip_mem_wr_en* signal will be asserted.

The 64-state environment looks like the following:

```
1.    typedef num {S1, S2, S3, …, Si, … , S64} state;
2.    assign rx_ip_data_ran = $ND(0, 1, 2, …, 255);
3.    always @ (posedge clock) begin
4.    case (state)
5.          S64: state = S1;
6.          S1: state = S2;
7.          S2: state = S3;
8.          S3: state = S4;
                  ……
```

```
9.          Si: state = Si+1;
                  ……
10.         S63: state = S64;
11.   endcase;
12.    if (state== S1)
13.         framestart = 1;
14.    else
15.         framestart = 0;
16.    if (state == S3)
17.         rx_ip_soc = 1;
18.    else
19.         rx_ip_soc = 0;
20.      ip_empty = 1;
21.      rx_ip_rd_req=1 ;
22.      ctr_sz = 0;
23.      ctr_id = 0;
24.      npc_rst_n = 1;
25.      rx_ip_data = rx_ip_data_ran;
26.    if (state = S4)  rx_ip_data_s4 = rx_ip_data_ran;
27.    else if (state=S5) rx_ip_data_s5 = rx_ip_data_ran;
28.    else if (state=S6) rx_ip_data_s6 = rx_ip_data_ran;
                              ……
29.    end
```

Figure Chapter 4 .5Environment of the null port controller for Property 3 using EM

In Figure Chapter 4 .5, line 1 enumerates the 64 states of the null port controller, and line 4 to 11 lists the transfer of the 64 states. Since one state is correspondent to a clock cycle, states will be transferred from S1 to S64 consecutively. Line12 to 15 defines that the *framestart* signal which is asserted at S1 and de-asserted at other states. Line 20 to 24 represent the above 1 to 5 assumptions, respectively. Line 25 assigns the input signal *rx_ip_data* as 8-bit random value. Line 26 stores the value of *rx_ip_data* at state S4 as *rx_ip_data_s4*; Likewise, line 27 to 28 store the values of *rx_ip_data* at state S5 and S6 as *rx_ip_data_s5* and *rx_ip_data_r6*, respectively. Using the similar method, we could store the value of *rx_ip_data* at any states. The stored *rx_ip_data* values will be

applied to verify if the data are transferred coherently.

By the above environment, Property 3 is expressible. We divide the verification into the two steps. The first step is to verify the assumptions to see if the environment represents the assumptions, and the second step is to check if conclusions are valid.

Step 1. Verify the above 5 assumptions.

The five assumptions can be expressed as the following CTL expression:

```
AG( npc_rst_n = 1)
(4.3.a.1)

AG(state= S1 -> ctr_id = 0 * ctr_sz = 0 * ip_empty =
1 *   rx_ip_rd_req = 1)
(4.3.a.2)

AG(state = S1  -> framestart = 1)
(4.3.a.3)

AG(state=S2 + state=S3 + … + state=S64 -> framestart
= 0)
(4.3.a.4)

AG(state = S3 -> rx_ip_soc = 1)
(4.3.a.5)
```

Formulas (4.3.a.3) and (4.3.a.4) is to test the behavior *framestart* signal. formula (4.3.a.2) expresses that the null port controller is not in program or debugging mode and is going to receive an ATM cell, and also the transmission board has a cell available to send. Please note the following two points on formulas (4.3.a.5) and (4.3.a.1):

1. In formula (4.3.a.5), we define "*rx_ip_soc=1*" at state S3, but the actual "*rx_ip_soc*" signal can be asserted at 1 ~11 clock cycles after S1. Because such assumption does not affect the behavior of the null port controller, the assumption is valid.

2. Since we use “AG” which means that the formula will be valid in any states and any paths, we must be careful of the initial state. For example, if we give the initial state as *npc_rst_n = 0*, formula (4.3.a.1) will not be valid.

Step2. Verify the conclusions.

In property 3, we have to verify two aspects: one is to ensure that two bytes of VCI are transferred to be memory address and further the memory address is incremented by 1 per byte data transfer, and the other is to verify that the data are transferred from transmission board to the memory with one clock cycle delay and the memory write enable signal is asserted during the data transfer.

At first, we verify the first aspect: the two bytes of VCI are transferred to be memory address and further the memory address is incremented by 1 per clock cycle.

Formulas (4.3.a.6) and (4.3.a.7) verify that the two bytes of VCI are transferred to be memory address correctly.

```
AG (state = S5 -> ip_mem_addr_r[8:1]==
rx_ip_data_s4)


(4.3.a.6)

AG (state = S6 -> ip_mem_addr_r[8:1] ==
rx_ip_data_s4 * ip_mem_addr_r[0] == rx_ip_data_s5[7]
* ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4]  *
ip_mem_addr_c[5:0] = 6'b000100)          (4.3.a.7)
```

The correct memory addresses increment can be verified by (4.3.a.8) to (4.3.a.11). (4.3.a.8) to (4.3.a.10) express that the memory row address is remained, but the memory column address is incremented by 1 per clock cycle until the total 50 bytes data (except two bytes VCIs) have been transferred. Due to the space limitation, we omit the CTL expressions when state = S9 to S55 which is very similar to Formulas (4.3.a.8) to (4.3.a.10) except that we have to give the correspondent values for ip_mem_addr_c[5:0].

Formula (4.3.a.11) represents that the memory address will be pointed to the first byte of a new VCI ATM cell so that the ATM cell will be transferred immediately after the next asserted *framestart* signal.

```
AG(state = S7 -> ip_mem_addr_r[8:1] == rx_ip_data_s4
*  ip_mem_addr_r[0] == rx_ip_data_s5[7]  *
ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4]  *
ip_mem_addr_c[5:0] = 6'b000100)          (4.3.a.8)

AG(state = S8 -> ip_mem_addr_r[8:1] == rx_ip_data_s4
* ip_mem_addr_r[0]==rx_ip_data_s5[7]  *
ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] *
ip_mem_addr_c[5:0] = 6'b000101)          (4.3.a.9)

AG(state = S56 -> ip_mem_addr_r[8:1] ==
rx_ip_data_s4 * ip_mem_addr_r[0]==rx_ip_data_s5[7]
*ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4]
*ip_mem_addr_c[5:0] = 6'b110101)
(4.3.a.10)

AG(state = S57 + …… +  state = S64 ->
ip_mem_addr_r[8:1] == rx_ip_data_s4 *
ip_mem_addr_r[0] == rx_ip_data_s5[7] *
ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] *
ip_mem_addr_c[5:0] = 6'b000000)
(4.3.a.11)
```

Next, we verify that the data are transferred from the transmission board to the memory with one clock cycle delay and the memory write enable signal is asserted during data transfer process. This sub-property involves two signals. One is the memory write enable signal (*ip_mem_wr_en*) and the other is the data output signals (*ip_mem_data*). *ip_mem_wr_en* signal should be asserted during the data transfer period (i.e. S7 to S56),

and this is expressed by (4.3.a.12) and (4.3.a.13). And also during the data transfer period, *ip_mem_data* should be the value of *rx_ip_data* with one clock cycle delay. The first and last byte data transfer are represented by (4.3.a.14) and (4.3.a.15). Due to the space limitation, we do not enumerate all the CTL expressions here, and the rest CTL expression will be very similar to (4.3.a.14) and (4.3.a.15).

```
AG (state = S1 + state = S2 + … + state = S6 + state
= S57 + … + state = S64 -> ip_mem_wr_en = 0)
```

(4.3.a.12)

```
AG (state = S7 + state = S8 + … + state = S56 ->
ip_mem_wr_en = 1)
```

(4.3.a.13)

```
AG (state = S7 -> ip_mem_data == rx_ip_data_s6)
```

(4.3.a.14)

```
AG (state = S56 -> ip_mem_data == rx_ip_data_s55)
```

(4.3.a.15)

So far, the assumptions and conclusions of Property 3 are satisfied by the combination of null port controller and environment, then we conclude that the assumptions imply the conclusion. (The proof is fairly simple, a proposition A is true and a proposition B is true, then the proposition A->B is true.) Environment Modification solves the problems of CTL expression limitation by adding some assumptions in the environment. There are two steps to verify a property by Environment Modification. The first step is to verify the assumptions of a property, and the second step is to verify the conclusions. Because ATM has a cyclic behavior, its behavior can be specified in an environment with some certain

states. By this method, all the above seven properties of the null port controller can be verified. However, we have to do minor modification on the environment when we verify a different property. And also, using this method, there are a lot of CTL formulas for a property because we have to verify the behavior at each environment state.

4.4.2 Internal Signal Usage

The idea of the method is to introduce some internal signals to help us to express a property. We still use property 3 as an example to illustrate this method.

At first, we have to establish an environment which is similar to Figure Chapter 4 .5, but we do not need specify the value of *ip_empty*, *npc_rst_n*, *rx_ip_soc*, *rx_ip_rd_req*, *ctr_id* and *ctr_sz* signals, instead, they are assigned as nondeterministic variables. Figure Chapter 4 .6 is the actual environment.

```
1.     typedef num {S1, S2, S3, …, Si, … , S64} state;
2.     assign rx_ip_data_ran = $ND(0, 1, 2, …, 255);
3.     always @ (posedge clock) begin
4.     case (state)
5.          S64: state = S1;
6.          S1: state = S2;
7.          S2: state = S3;
8.          S3: state = S4;
                              ……
9.          Si: state = Si+1;
10.         S63: state = S64;
11.     endcase;
12.     if (state== S1)
13.         framestart = 1;
14.     else
15.         framestart = 0;
16.         rx_ip_soc = rx_ip_soc_ran;
17.         ip_empty = ip_empty_ran;
18.         rx_ip_rd_req =rx_ip_rd_req_ran;
19.         ctr_sz = ctr_sz_ran;
20.         ctr_id = ctr_id_ran;
```

```
21.         npc_rst_n = npc_rst_n_ran;
22.         rx_ip_data = rx_ip_data_ran;
23.     if (state = S4)  rx_ip_data_s4 = rx_ip_data_ran;
24.     else if (state=S5) rx_ip_data_s5 = rx_ip_data_ran;
25.     else if (state=S6) rx_ip_data_s6 = rx_ip_data_ran;
                                ……
26.     always @(posedge clock) begin
27.          ip_mem_addr_c_r1[5 : 0] = ip_mem_addr_c[5:0];
28.     end
29.     always @(posedge clock) begin
30.          ip_mem_addr_c_plus1 = ip_mem_addr_c_r1[5:0] + 1;
31.     end
32.     always @(posedge clock) begin
33.          rx_ip_data_r1 = rx_ip_data;
34.     end
```

Figure Chapter 4 .6 Environment of null port controller for the internal signals involved CTL

Since we only can compare the equivalence between two signals or between one signal and a certain value in model checking, we propose to build some assistant signals (variables) to ease property expression in model checking. For instance, in Figure Chapter 4 .6, line 29 to 31 is to create the assistant signal *ip_mem_addr_c_plus1* which is always equal to "ip_mem_addr_c[5:0] + 1" with one clock cycle delay. This signal will be used in the CTL formulas.

Similar to Section 4.4.1, we verify the proper address transfer and increment first, and then verify the correct data transfer. To verify the proper address transfer and increment, we need prove the following three consecutive sub-properties:

sub-property1: The null port controller uses the two bytes VCI as the initial memory address two clock cycle after *rx_ip_soc* asserts;

sub-property2: After sub-property1, the memory address is incremented by 1 per clock cycle until the 50 bytes data have been completely transferred;

sub-property3: After sub-property2, the memory address will point to the first byte of the ATM cell.

The following formulas (4.3.b.1 ~ 4.3.b.8) are CTL expression of the three sub-properties:

```
AG (framestart = 1 -> ip_state_i = idle)
(4.3.b.1)

AG(framestart = 1 * npc_rst_n = 1 * ip_state_i =
idle * ip_empty = 1 * rx_ip_rd_req = 1 * ctr_id = 0
-> AX (ip_state_i = rx_wait))
(4.3.b.2)

AG(ip_state_i = rx_wait * rx_ip_soc = 1  ->
AX(ip_state_i = rx_store1))
(4.3.b.3)

AG (ip_state_i = rx_store1 * ctr_sz = 0 ->
AX(ip_state_i = rx_store_2 * ip_mem_addr_r[8:1] ==
rx_ip_data_s4))                              (4.3.b.4)

AG(ip_state_i = rx_store2 *  ctr_sz = 0 -> AX
(ip_state_i = rx_data *
ip_mem_addr_r[8:1]==rx_ip_data_s4 * ip_mem_addr_r[0]
== rx_ip_data_s5[7] * ip_mem_addr_c[8:6] ==
rx_ip_data_s5[6:4] * ip_mem_addr_c[5:0] = 6'b000100
* ip_cell_cnt = 50 )
(4.3.b.5)

AG(ip_state_i = rx_data ->
(ip_mem_addr_r[8:1]==rx_ip_data_s4 *
ip_mem_addr_r[0] == rx_ip_data_s5[7] *
ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] *
ip_mem_addr_c[5:0] == ip_mem_addr_c_plus1 *
ip_cell_cnt == cell_cnt_minus1))
```

(4.3.b.6)

```
AG(ip_state_i = rx_data * ip_cell_cnt = 1 -> AX
(ip_state_i = ip_idle *
ip_mem_addr_r[8:1]==rx_ip_data_s4 * ip_mem_addr_r[0]
== rx_ip_data_s5[7] * ip_mem_addr_c[8:6] ==
rx_ip_data_s5[6:4] * ip_mem_addr_c[5:0] = 53)
```

(4.3.b.7)

```
AG(ip_state_i = rx_data * ip_cell_cnt = 1 -> AX
AX(ip_state_i = ip_idle * ip_mem_addr_r[8:1] ==
rx_ip_data_s4 * ip_mem_addr_r[0] == rx_ip_data_s5[7]
* ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] *
ip_mem_addr_c[5:0] = 0)                    (4.3.b.8)
```

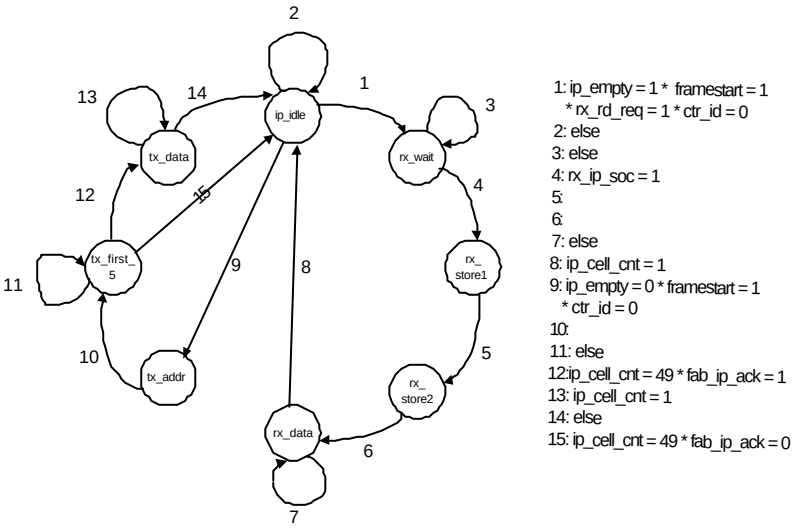


Figure Chapter 4 .7 State diagram of the input port controller

We use the internal signal *ip_state_i* to help us express subproperties in CTL. *ip_state_i* is a state variable which has 8 states: *ip_idle, rx_wait, rx_store1*, *rx_store2*,

rx_data, *tx_addr*, *tx_first_5* and *tx_data* (Figure Chapter **4 .7**). The state transition is shown as Figure Chapter **4 .7**, basically, *rx_idle* is "idle" state; *rx_wait* means the state of waiting for rx_ip_soc asserted; *rx_store1* and *rx_store2* indicate the states that the input port controller stores the first and second VCI byte, respectively; *rx_data* is the state of data transfer from transmission board to the input port controller; *tx_addr* is the state of setting the memory address; *tx_first_5* means the state of transmitting the first 5 bytes data to the fabric; *tx_data* indicates the state of transmitting the rest of data into the fabric. Sub-property 1 can be deduced by (4.3.b.1) to (4.3.b.5), and the deduction is based on property division. By (4.3.b.1) and (4.3.b.2), we obtain (4.3.b.9), and by (4.3.b.3), (4.3.b.4) and (4.3.b.5), we obtain (4.3.b.10). The combination of (4.3.b.9) and (4.3.b.10) is the sub-property1.

```
AG(framestart = 1 * npc_rst_n = 1 * ip_empty = 1 *
rx_ip_rd_req = 1 * ctr_id = 0 -> AX (ip_state_i =
rx_wait))                                    (4.3.b.9)

AG(npc0.ip_state_i = rx_wait *  rx_ip_soc = 1 *
ctr_sz = 0 -> AX AX(ip_state_i = rx_data *
ip_mem_addr_r[8:1] ==rx_ip_data_s4 *
ip_mem_addr_r[0] == rx_ip_data_s5[7] *
ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] *
ip_mem_addr_c[5:0] = 6'b000100 * ip_cell_cnt = 50 ))
(4.3.b.10)
```

In formula (4.3.b.6), we use two assistant signals: *ip_mem_addr_c_plus1* and *cell_cnt_minus1*. As shown in Figure Chapter 4 .6, *ip_mem_addr_c_plus1* is always equal to *ip_mem_addr_c[5 :0] + 1* with one clock cycle delay, so *ip_mem_addr_c[5 : 0] = ip_mem_addr_c_plus1* represents that *ip_mem_addr_c[5:0]* will increment by one each clock cycle. Similarly, we can build up the signal *cell_cnt_minus1*. Since *cell_cnt_minus1* is an internal signal, *cell_cnt_minus1* has to be defined inside the null

port controller.

Sub-property2 can be deduced by (4.3.b.5), (4.3.b.6) and (4.3.b.7). (4.3.b.5) implies that *ip_cell_cnt = 50* and the memory address points to the first data byte in the first clock cycle of *rx_data* state; (4.3.b.6) means that *ip_cell_cnt* decrements by 1 and *ip_mem_addr_c* increments by 1 per clock cycle during *rx_data* state; (4.3.b.7) indicates that when *ip_cell_cnt* reaches 1, *ip_state_i* will become "idle", and the least significant bits of *ip_mem_addr_c* will be 54 which points to the last data byte of an ATM cell. The deduction is still based on property division. Formula (4.3.b.6) represents the general behavior of *ip_mem_addr_c*, *ip_mem_addr_r* and *ip_cell_cnt*. Formula (4.3.b.5) and (4.3.b.7) give the lower and upper boundary of *ip_mem_addr_c* and *ip_mem_cnt*. Since the three CTL expressions are relatively complicated, we use the following simple example to illustrate how the deduction works.

Supposed we have the following three valid CTL expressions (4.e.1, 4.e.2 and 4.e.3), T1, T2, T3 expresses three different environment states, *addr* is a variable, and *addr_plus1* is the variable which is always greater than *addr* of the previous clock cycle by 1. Formula (4.e.1) and (4.e.3) have the similar formats. Formula (4.e.2) is a general expression, so we could convert (4.e.2) into (4.e.4) which includes 49 CTL expressions which have the same style as formula (4.e.1) or (4.e.3). By (4.e.1) (4.e.2) and (4.e.4), we can infer that *addr* will be incremented by 1 per clock cycle during state T2 and T2 state will last for 50 clock cycles to allow *addr* increment from 4 to 53. In this example, a simple infer rule has been applied.

```
AG (state = T1 -> AX (state = T2 * addr = 4)          (4.e.1)

AG (state = T2 -> AX (addr == addr_plus1)             (4.e.2)

AG (state = T2 * addr = 53 -> AX (state = T3 * addr = 54))
                                                      (4.e.3)

AG(state = T2 * addr = 4 -> AX (addr = 5))
```

```
AG(state = T2 * addr = 5 -> AX (addr = 6))
      ……
AG(state = T2 * addr = 52 -> AX(addr = 53))          (4.e.4)
```

Formula (4.3.b.5), (4.3.b.6) and (4.3.b.7) are very similar to formula (4.e.1) (4.e.2) and (4.e.3), so we could use the similar conversion to infer sub-property 2.

(4.3.b.8) indicates after the 50 bytes data transfer, the memory address will point to the first byte of an ATM cell with new VCI and header, and this is sub-property 3.

By the above formula, we see that *rx_data* state will keep for 50 clock cycles which allow to transfer 50 bytes data cell to the memory. So the data transfer state can be easily expressed by *ip_state_i = rx_data*. To prove the correct data transfer, we only need to prove that *ip_mem_wr_en* is asserted and the data are transferred from the transmission board to the memory only during *rx_data* state. (4.3.b.11)(4.3.b.12) and (4.3.b.13) express this property. In (4.3.b.13), *rx_ip_data_r1*, which is also an assistant signal, denotes the value of *rx_ip_data* with one clock cycle delay (Figure Chapter 4 .6).

```
AG(ip_state_i = rx_data -> AX(ip_mem_wr_en = 1))

(4.3.b.11)

AG(!(ip_state_i = rx_data)-> AX(ip_mem_wr_en = 0))

(4.3.b.12)

AG (ip_state_i = rx_data -> AX (ip_mem_data
==rx_ip_data_r1))
(4.3.b.13)
```

So far, all the sub-properties are expressed in CTL, and property 3 is the

combination of these sub-properties. By Internal Signal Usage, we can express all the properties in the null port controller without any environment modification, but we have to be very clear about the implementation of the null port controller. In addition, we need to adopt property division to deduce a property by several sub-properties. Between Environment Modification and Internal Signal Usage, it is hard to say which one is better. Environment Modification can express the behavior clearly with the knowledge of the design specification. Internal CTL Usage saves a lot of time to modify the environments for different properties, but verifier must have in-depth knowledge on the design implementation. Since Internal Signal Usage uses some internal signals in CTL expression, the CPU time of the model checking will be less than that using Environment Modification. Although both of the methods can succeed in verifying properties, we find it is still hard to apply them in a large design and also too many explicit CTL formulas corresponding to environment states are required for a property. Therefore, we propose Counter Reduction method. Counter reduction, which concentrates on the main behavior of a design, reduces the scale of counters and the correspondent environment states so that CTL expressions are simplified and the model checking gets enhanced.

4.4.3 Counter Reduction

In property 3, 50 bytes data are transferred from transmission board to the input port controller. Because a byte of data transfer has the same behavior as the data transfer of other 49 bytes, we could reduce the number of data transfers in the verification. In the null port controller, the number of data transfer is controlled by counters, so we propose to reduce the scale of the counter to simplify our verification. This method could be applied to many ATM hardware verifications because counters are often consisted of by ATM hardware designs.

In the null port controller, the acknowledge signal should be available at 5 clock cycles after the input port controller sends the first byte data to the fabric, so we could apply 12 bytes data in a cell which includes 2 bytes VCIs, 2 bytes FAS and 8 bytes data).

Accordingly, the counter size should be reduced by 40 (52-12). Then we have to change our environment machine from 64 state to 15 states (10 states for data transfers and 5 states for handshaking).

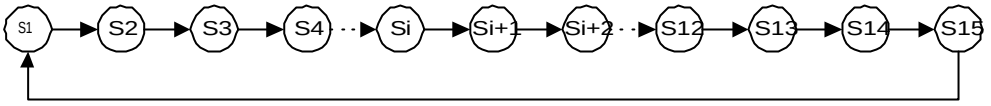


Figure Chapter 4 .8 Environment of null port controller using counter reduction method

Figure Chapter 4 .8 is the new environment. If we consider the receiving process of the null port controller, S1 and S2 are correspondent to *framestart* and *rx_ip_soc* assertions, S3, S4 are corresponding to the state *rx_store1* and *rx_store2*, respectively, S5 to S14 are for *rx_data* state, S15 is for *ip_idle*. After reducing the environment states and the counter sizes, we could use either Environment Modification or Internal Signal Usage methods to verify it. In this work, we apply both Environment Modification and Internal Signal Usage to verify Property 3 with the reduced model. The experiment results are present in the next subsection, and their environments and CTL formulas are described in Appendix B.

4.4.4 Experimental Results and Summary on the Three Methods

We perform the model checking in VIS-1.3 which is installed in SUN ULTRA SPARC (300MHz/500 MB).

Methods	Environment Modification	Internal Signal Usage	Counter Reduction	
			with EM	with ISU
# of CTL Expressions	110	11	30	11
CPU time (seconds)	2745	150	209	71
Memory Usage (MB)	398	235	156	108
Node Allocated (K)	734, 475	453, 598	293, 354	183, 254
Design Modification	No	No	Minor	Minor

Environment Modification	Minor	No	Minor	No
Knowledge on the Design Implementation	Black box	White box	No	Deep

Table Chapter 4 .2 Experimental results and summary on the three methods

Table Chapter 4 .2 lists the experimental results and summaries the advantages and drawbacks of the three methods. Environment Modification (EM) need more CTL expressions and minor environment modification, but a verifier is only required to understand the specification of a design. On the contrary, Internal Signal Usage (ISU) has less CTL expressions and the environment is fixed for all the properties, but the verifier must know very well about the design. Counter Reduction can enhance the model checking, and it becomes extremely necessary for a large design.

4.5 Experimental results and error detection:

We applied the combination of Environment Modification and Counter Reduction methods to verify the rest of six properties. Since their environments and CTL expressions are similar to that of Property 3, we put them in Appendix B. The model checking was performed in SUN ULTRA SPARC (300MHz/500 MB). Table Chapter 4 .3 is their experimental results.

Properties	# of CTL expression	CPU time (seconds)	Memory Usage (MB)	Nodes Allocated (K)
Property 1	1	52	92	203,493
Property 2	30	256	198	284,563
Property 3	30	209	156	293, 354
Property 4	25	378	201	304,731
Property 5	-	-	-	-
Property 6	2	34	77	153,980
Property 7	2	76	89	197,091

Table Chapter 4 .3 Experimental results on the model checking of null port controller

In Table Chapter 4 .3, we see that all the properties pass the model checking by reasonable CTU time. Property 4 took the longest CPU time (378 seconds) which was around half a hour of machine time. So in terms of machine time, model checking on the null port controller is also acceptable.

We did simulation on the null port controller before the model checking. By the simulation, we detected a number of syntax errors, mistaken variable names and wrong counter numbers. By the model checking, we detected some logic errors such as the memory write and read incoherence, the signals *ctr_sz* and *ctr_id* misfunction. The errors were traced by the counterexamples generated by VIS.

4.6 Summary

In this chapter, we verified Fairisle ATM null port controller by model checking. Compared with the Fairisle ATM switch fabric, the null port controller is more difficult to be verified because its state transitions are more complicated and both environments and CTL formulas are harder to handle. However, we succeeded in verifying it by applying several techniques on CTL expression and environment establishment. The techniques include the three methods of describing a property in CTL, the creation of assistant variables and property division. These techniques were illustrated by verifying Property 3 in different ways. By these techniques, all the properties were verified in a reasonable time and we also detected several designs errors.

Chapter 5

Model checking of Input FIFO

Chapter 3 and Chapter 4 give the verification examples on the academic ATM switch designs. In this chapter, we will apply model checking in a commercial one. Our example is to adopt model checking to verify a block of RCMP-800 [27] chip which is a product of PMC-sierra Inc..

5.1 Introduction on RCMP-800

The Routing Control, Monitoring and Policing 800 Mbps (RCMP-800) device is a monolithic integrated circuit that implements ATM layer functions that include fault and performance monitoring, header translation and cell rate policing. The RCMP-800 is intended to be situated between a switch core and the physical layer devices in the ingress direction. The RCMP-800 supports a sustained aggregate throughput of 1.42 x 10^6 cell/s. The RCMP-800 uses external SRAM to store per-VPI/VCI data structures. The device is capable of supporting up to 65536 connections. The input cell interface can be connected to up to 32 physical layer devices through a SCI-PHY compatible bus. The 53 byte ATM cell is encapsulated in a data structure which can contain pre-pended or post-pended routing information. Receiving cells are buffered in a four cell deep FIFO. All physical layer and unassigned cells are discarded. For the remaining cells, a subset of ATM header and appended bits is used as a search key to find the VC Table Record for the virtual translation. If a connection is not provisioned and the search terminates unsuccessfully as a result, the cell is discarded and a count of invalid cells is incremented. If the search is successful, subsequent processing of the cell is dependent on contents of the cell and configuration fields in the VC Table Record.

The RCMP-800 performs header translation if configured. The ATM header is replaced by contents of fields in the VC Table Record for the connection. The VCI contents are passed through transparently for VPCs. Appended bytes can be replaced, added or removed. If the RCMP-800 is the end point for an F4 or F5 OAM stream, the OAM cells are dropped and processed. If the RCMP-800 is not the end point, the OAM cells are passed to the Output Cell Interface with an optional copy passed to the Microprocessor Cell Buffer for external processing. Continually Check cells can be generated if no user cells have been received in the latest 1.5 +/- 0.5 or 2.5 +/- 0.5 (default) seconds.

Cell rate policing is supported through two instances of the Generic Cell Rate

Algorithm (GCRA) for each connection. Each cell that violates the traffic contract can be tagged (CLP bit set high) or discarded. To allow full flexibility, each GCRA instance can be programmed to police any combination of user cells, OAM cells, Resource Management, high priority cells or low priority cells.

The RCMP-800 supports multicasting. A single received cell can result in an arbitrary number of cells presented on the Output Cell Interface, each with its own unique VPI/VCI value and appended bytes. The ATM cell payload is duplicated without modification.

The Output Cell Interface can be connected to the switch core through an extended cell format SCI-PHY compatible bus. Cells are stored in a four cell deep FIFO until the downstream devices are ready to accept them. The details of how cells are handled in this FIFO depends on the particular application of the RCMP-800.

The Microprocessor Interface is provided for device configuration, control and monitoring by an external microprocessor. This interface provides access to the external SRAM of the data structure, configuration of individual connections and monitoring of the connections. The Microprocessor Cell Buffer gives access to the cell stream, either directly or through intervention by a DMA controller. Programmed cell types can be routed to a microprocessor readable sixteen cell FIFO. The microprocessor can send cells over the Output Cell Interface.

The structure of the chip is shown as Figure Chapter 5 .1, and it consists of an input FIFO, output FIFO, Microprocessor interface, Micro Cell buffer, Cell Processor, Microprocessor RAM arbitration, External RAM address Look up and JTAG Test access port. Both Input and output FIFO are four cell depth FIFO. The basic function of the input FIFO is to receive the data cells into the RCMP, and that of the output FIFO is to transmit the data cells to the fabric. ATM cells are transferred to Micro Cell buffer from the input FIFO, and the microprocessor will read the ATM cells in the Micro Cell, and check the cell types, cell header, VCI /VPI and CI (connection identifier). Depending on such information, the microprocessor looks for the VC table, and determines the prepend and

postpend bytes, or tags CLP bit or discards the cell if GCRA is violated, or inserts RM or OAM cell which will be written in Micro Cell Buffer. Cell Processor and Microprocessor RAM arbitration, External RAM address look-up are used to look up VC routing table, so this part involves both hardware and software. The pure hardware parts for the RCMP-800 are the input and output FIFO. The whole chip verification which involves hardware and software verification is not possible for model checking due to the limitation of the current formal verification tools. However, model checking can be used to verify an individual block of RCMP-800, here we use the input FIFO as an example.

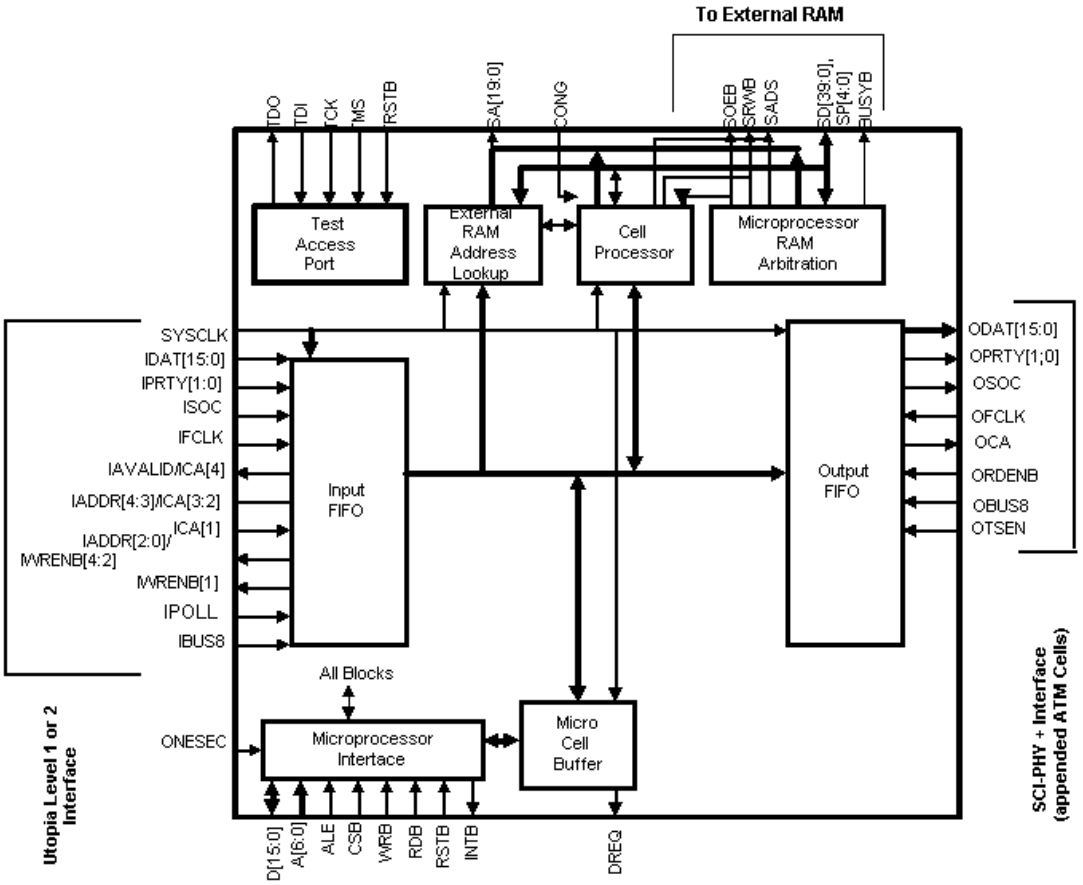


Figure Chapter 5 .1 Structure of RCMP-800

5.1.1 Behavior of Input FIFO

Cells received on the extended cell format SCI-PHY compatible Input Cell Interface are buffered in a 4 cell deep FIFO. The input buffer provides for the separation of internal timing from asynchronous external devices.

The SCI-PHY cell interface operates at clock rates up to 52 MHz and supports 16 bit and 8 bit wide data structures with programmable lengths. The 16 bit data structure contains 26 (HEC and User Defined Field excluded) or 27 words allocated to carrying an ATM cell and up to 5 appended words. The 8 bit data structure contains a 52 bytes (HEC excluded) or 53 bytes ATM cell and up to 10 appended bytes. The start of the data structure is indicated by the ISOC input. The data bus is protected by the IPRTY[1:0] inputs. The parity can be configured to be odd or even. Each parity input can cover a byte or IPTY[1] can cover all the sixteen bits data inputs.

The input FIFO filters all unassigned cells and cells reserved for the use of the Physical Layer. Unassigned cells are identified by an all zero VPI/VCI value and CLP=0. They are filtered without notification. Physical layer cells are identified by an all zero VPI/VCI value and CLP = 1. They are filtered with a resulting maskable interrupt indication and a Physical Layer cell count increment. By default, the cell coding is assumed to be for a Network-Network interface (NNI); therefore the VPI is taken to be twelve bits. If one of the PHY links is a User-Network Interface(UNI) and the GFC field is non-zero, the cell will be filtered by the Input Cell Interface (UNI) and the GFC field is non-zero, the cell will not be filtered by the input Cell Interface, but will be discarded by the VC Identification circuit. As an option, all cells can be interpreted as UNI cells.

The RCMP-800 is a bus master and services the PHY devices as one of two ways: direct status arbitration or address line polling. For direct status arbitration, the RCMP-800 monitors cell available signals (*ICA[4:1]*) from up to four physical (PHY) layer devices and generates write enables (*IWRENB[4:1]*) in response. For address line polling, *ICA[1]* and

IWRENB[1] are shared between up to 32 PHY devices and signals *IADDR[4:0]* and *IAVALID* are used to address the latter individually. The RCMP-800 performs round-robin polling of the PHY devices to determine which have available cells. The RCMP-800 will read an entire cell from one PHY device before accessing the next PHY device. No fixed cell slots exist, but instead the RCMP-800 maximizes throughput by servicing a PHY devices as soon as the bus is free and PHY device's cell available signal is asserted.

5.1.2 Functions of the input FIFO

The input FIFO receives ATM cells from the transceiver board, and stores the ATM cells in the input FIFO which has four-cell depth. The main function of the input FIFO is as the following:

1) Store ATM cells in the input FIFO. There are three counters: read counter, write counter and FIFO counter. Read counter and write counter are used to control the read and write of the FIFO, and FIFO counter indicates the depth of the FIFO.
2) Parity checking for the input data, and the parity check result is stored in the register.
3) Discard the unsigned cells.
4) Discard physical layer OAM cell with a notification
5) Two modes of the PHY devices services:
    - One is direct status arbitration. Only four physical devices are on the transceiver board, and each physical device has an individual "ICA" (input cell available) and "IWREN"(write enable signal).
    - The other is address line polling. There are 32 physical devices which are accessed by 5-bit physical address and share the same "ICA" and "IWREN" signals.

5.2 Verification of the input FIFO

Similar to the previous verification, we wrote the RTL description of the input FIFO in Verilog, but we did minor changes on the model. The difference between the original model and our verification model was that we only used 16-bit data path while the original design used either 16-bit or 8-bit data path. In addition, the input FIFO had 128 x 16 bit memory which could contain 4 ATM cells, but VIS could not handle such big memory verification. Therefore, we verified the control circuits of the input FIFO excluding the memory. Such reduction is practical because the memory model is relatively mature and the control circuit is problematic part in the verification. To simplify our verification, we apply Internal Signal Usage to describe properties.

5.2.1 The environment of the input FIFO

Similar to the previous chapter, we establish an environment for the input FIFO, and the environment gives the inputs as random variables and defines registers as a default value. Figure Chapter 5 .2 is the environment.

```
1.      always @(posedge clock) begin
2.      idat = idat_ran;
3.      prty = prty_ran;
4.      isoc = isoc_ran;
5.      ica4 = ica4_ran;
6.      ica1 = ica1_ran;
7.      ica32 = ica32_ran;
8.      ipoll = ipoll_ran;
9.      ifrdb = ifrdb_ran;
10.     hecudf = 1;
11.     icainv = 0;
12.     cellpost = 0;
13.     celllen = 0;
14.     ibyteprty = 0;
15.     icalevel0 = 1;
16.     ifrst = 0;
```

```
17.     iptyp = 0;
18.     end
```

Figure Chapter 5 .2 Environment of the input FIFO

In Figure Chapter 5 .2, line 2 to 9 are inputs from transceiver board and the block of Micro cell buffer, so we define them as nondeterministic variables. Line 10 to 14 are registers, so we give them as their default values. Because our verification is to focus on the critical behavior of the block, the constant register values will not have an influence on the verification. However, to further verify the block, we could apply other constant values for these registers.

Before we give any property description, we briefly introduce the signals in Figure Chapter 5 .2. *idat* is a 16-bit data input from transceiver board. *prty* is a 2-bit parity inputs from transceiver board. *isoc* is the "start of a cell" signal which indicates the first byte of a cell from the transceiver board. *ica4* represents the cell available for PHY device 4. when *ipoll* is low. ica32 indicates cell available for PHY device 2 and 3. *ipoll* determines the method used to poll PHY devices. If *ipoll* is high, address lines polling is applied, and it will support the maximum 32 input devices; otherwise, the input FIFO connects to four PHY entities. *ifrdb* means input FIFO read enable. In Figure Chapter 5 .2, *hecudf, icainv, cellpost, celllen, ibyteprty, icalevel0, ifrst* are such registers. *Hecudf* determines whether or not the HEC/UDF octets are included in cells transferred over the input interface. When set to logic 1 (default), the *HEC* and *UDF* octets are included; otherwise, they are omitted. The *icainv* bit selects the active polarity of the *ICA[4:1]* signals. The default configuration (*icainv = 0*) selects *ICA[4:1]* to be active high; when *icainv* is set to logic one, the *ica[4:1]* become active low. The *cellpost[3:0]* bits determine the number of postpend words in an input cell. The *celllen[3:0]* bits determine the number of appended words to the input cell. The *ibyteprty* bit selects between byte parity and word parity; if *ibyteprty* is set high, *iprty[1]* is expected to be the parity over *idat[15:8*] and *iprty[0]* is expected to be the parity over *idat[7:0]*. If *ibyteprty* is set low, *iprty[1]* is expected to be

the parity over *idat[15:0*] and *iprty[0]* is ignored. The *icalevel0* bit determines how the *ICA[4:1]* are interpreted. If *icalevel0* is high, the RCMP-800 checks for close compliance to the SCI-PHY cell transfer handshake. In this case, if the ICA signal for the PHY whose cell is currently being transferred is deasserted before the end of the cell, the cell will be discarded. If *icalevel0* is logic 0, the *ica* signal may be deasserted early without the loss of the cell. Once a cell transfer is initiated, the entire cell will be read contiguously regardless of the state of the ICA signal. The *ifrst* bit determines whether the input FIFO is in a reset state. *iptyp* determines the type of parity checking, *iptyp = 1* indicates the even parity checking, otherwise, it is odd parity checking.

5.2.2 Model Checking

According to the functions of the input FIFO described in 5.1.2, we give 8 properties.

Property 1: In normal operation (not in *discard* operation), write counter will increment by 1 whenever *writeB* is deasserted. The CTL expression is the following:

```
AG (discard = 0 * writeB=0 -> AX (wr_ptr ==
wr_ptr_plus1))
(5.1.1)
```

Where *discard* is an internal signal which determines whether the FIFO is in normal operation or discard operation, *writeB* is a write enable signal, and *wr_ptr* is a write pointer. Similar to Section 4.4.2, we introduce the assistant variable *wr_ptr_plus1* in the design module.wr_ptr_plus1 will always be *wr_ptr + 1* with one clock cycle delay.

Property 2: Whenever *ifrdb* is deasserted, read counter will incremented by 1 every clock cycle. The property can be expressed as the following:

```
AG(ifrdb = 0  -> AX(rd_ptr == rd_ptr_plus1))
(5.2.1)
```

Where *ifrdb* indicates Micro cell FIFO has enough space to receive a cell, and *rd_ptr* is read pointer of the input FIFO. Similar to *wr_ptr_plus1*, *rd_ptr_plus1* is

introduced to have the value of *rd_ptr + 1* with one clock cycle delay.

Property 3: In a normal operation, *ifcounter* will increment by 1 whenever *writeB* is deasserted and *ifrdb* is asserted; and *ifcounter* will decrement by 1 whenever *writeB* is asserted and *ifrdb* is deasserted.

```
AG (discard=0 * writeB =0 * ifrdb = 1 ->
AX(ifcounter ==ifcounter_plus1))
(5.3.1)

AG(discard=0 * writeB = 1 * ifrdb = 0  ->
AX(ifcounter == ifcounter_minus1)            (5.3.2)
```

discard, *writeB* and *ifrdb* have the same meaning as the property 2. *Ifcounter* indicates the depth of the input FIFO. Similarly, *ifcounter_plus1* and *ifcounter_minus1* are introduced to represent the values of *ifcounter + 1* and *ifcounter - 1* with one clock cycle delay, respectively.

Property 4: Parity check functions correctly, and result will be stored in the register. Since we define *ibyteprty* as default value "0" and *iptyp* as a default value "0", it is word-basis, odd parity checking.

```
AG (prtychkl == idat[0] ^ idat[1] ^ idat[2] ^
idat[3] ^ idat[4] ^ idat[5] ^ idat[6] ^ idat[7])
(5.4.1)

AG (prtychk2 ==idat[8] ^idat[9] ^ idat[10] ^
idat[11] ^ idat[12] ^ idat[13] ^ idat[14] ^
idat[15])  (5.4.2)

AG(iprty[1]= !(prtychkl ^prtychk2 ^ prty[1]))
(5.4.3)
```

iprty[1] = 1 indicates the parity error over the IDAT[15:0] bus, so our specification of the property is *AG(ibyteprty = 0 * iptyp=0 -> (iprty[1] = not (prty[1]*

^ *idat[0]* ^ *idat[1]* ^ *idat[2]* ^ … ^ *idat[15])).* According to property division, this specification is easily proved by (5.4.1), (5.4.2) and (5.4.3).

Property 5: Any unassigned cells will be discarded by the input FIFO.

The ATM header determines whether a cell is unassigned cell or not. If all the bits of VPI and VCI and CLP bit are zero, then the cell is unassigned. Since we consider NNI here, the format of an unsigned cell is like Table Chapter 5 .1.

ATM header	Octet 1	Octet 2	Octet 3	Octet 4	Octet 5
unassigned cell	00000000	00000000	00000000	0000xxx0	HEC byte

Table Chapter 5 .1 The format of an unassigned cell

```
AG (idat[15:0]=0  * cellcount = 0 * writeB = 0 -> AX
(vpi_vci[27:12] = 0 * cellcount = 1))
(5.5.1)

AG (idat[15:0]=0  * cellcount = 0 * writeB = 0 -> AX
AX (vpi_vci[27:12] = 0))
(5.5.2)

AG(idat[11:0]=0 * idat[15]=0 * cellcount= 1 * writeB
= 0 -> AX (vpi_vci[11:0] =0 * clp = 0))
(5.5.3)

AG(vpi_vci[27:0] * clp = 0 * cellcount=2 -> AX
discard = 1)                                    (5.5.4)
```

Where *idat* is a 16 bit data path which receives data from transceiver board, *vpi_vci* is a 28-bit registers which store VPI and VCI value for each cell, and *cellcount* indicates how many data bytes have been transferred into RCMP for each cell. Property 3 can be deduced by formulas 5.5.1 to 5.5.4.

Property 6: Any physical cells will be discarded by the input FIFO with a notification.

Similar to unassigned cell, physical cell is determined by its ATM header, and the

format of a physical cell is in Table Chapter 5 .2.

ATM header	Octet 1	Octet 2	Octet 3	Octet 4	Octet 5
Physical cell	xxxx0000	00000000	00000000	0000xxx1	HEC byte

Table Chapter 5 .2 The format of a physical cell

Like Property 5, Property 6 can be deduced by (5.6.1) (5.6.2) (5.6.3) and (5.6.4).

```
AG(idat[15:4] = 0 * cellcount = 0 * writeB = 0-> AX
vpi_vci[23:12] = 0 * cellcount = 1)          (5.6.1)

AG(idat[15:4] = 0 * cellcount = 0 * writeB = 0 -> AX
AX vpi_vci[23:12] = 0 )                      (5.6.2)

AG(idat[11:0]=0 *idat[15]=1 * cellcount=1 * writeB=
0 ->AX(vpi_vci[11:0]=0 * clp=1))            (5.6.3)

AG(vpi_vci[23:0] * cellcount = 2 * clp = 1 ->
discard = 1 * phycell = 1)
(5.6.4)
```

Property 7: If ipoll is low, the RCMP is receiving data from PHY device 1 and PHY device 2 has a cell available, PHY device 2 will transmit a cell to RCMP next.

Switching a receiving PHY device from one to another only happens at the state *cellcount = 0*. When a PHY device gets permission to transfer a cell into RCMP, the write enable signal (*iwren*) will be asserted. And also the variable *p_state* stores the number of PHY device. Therefore, the CTL expression is as the following:

```
AG ( ipoll = 0 * cellcount = 0 * p_state = 1 * ica2
= 1  -> AX (iwren2 = 1))
(5.7)
```

In (5.7), *p_state* stores the number of current receiving PHY device, *ica2 = 1* means that PHY device 2 has a cell to send, *iwren2 = 1* means PHY device 2 get the permission to send.

Property 8: If *ipoll* is high, the RCMP is receiving data from the PHY device of address 10 and the PHY device of address 11 has a cell to send, RCMP will transmit the cell from PHY device of address 11 next.

```
AG (ipoll = 1 * cellcount = 0 * iaddr = 10 * ica = 1
-> AX AX (iaddr = 11))                                    (5.8)
```

In (5.8), *iaddr = 10* expresses that the address of the current receiving PHY device is 10. Because RCMP has a pipeline searching process, *iaddr* will be equal to 11 in two clock cycles.

The above properties do not cover all the properties of the functions listed in 0.1.2, but other properties can be described in a similar way.

5.2.3 Experimental Results and Error Detection

We did meet state explosion problem when verifying these properties, and the machine gave the error indicating that the memory cannot be allocated. So we applied "hide" method, and this is to "hide" unrelated design when verifying a property. Because a hardware design is "process-based", we could simply comment unrelated process when verifying a property, and it is also possible to write a program to search unrelated processes and comment them automatically. Although the method seems very obvious, it is very effective. By this method, all the properties were verified in VIS with a reasonable CPU time, Table Chapter 5 .3 is the CPU time and number of CTL expression for each property.

Properties	Number of	CPU time	Memory Usage	Nodes Allocated

	CTL	(seconds)	(MB)	(K)
Property 1	1	75	97	103, 907
Property 2	1	57	68	87,103
Property 3	2	63	59	91,034
Property 4	3	56	87	79,308
Property 5	3	62	61	71,805
Property 6	3	74	102	174,830
Property 7	1	23	42	34,049
Property 8	1	12	34	20,911

Table Chapter 5 .3 Experimental results of input FIFO model checking

We did simulation before performing the model checking, but we still found a number errors in " address line polling circuitry" when testing Property 7 and 8, finally, we re-write those circuits.

5.3 Summary

In this chapter, we applied model checking to verify a block of an ATM commercial product. Using Internal Signal Usage method, we described the properties in CTL. To save state space, we defined register variables as their default values in the environment. But we still encountered state space explosion problem, and we solved the problem by adopting "hide" method which is to comment out of the property unrelated design description when verifying a certain property. In this work, model checking of all the properties are done with a reasonable time, and we did detect a set of design errors by the model checking.

Chapter 6

Equivalence Checking

With the rapid development of EDA tools, a top-down design phase has been used in industries. A digital designer usually describes a design as RTL code, and synthesizes RTL into gate-level automatically by synthesis tools. The gate level design will be further converted into transistor-level design by using standard cells. Since the such conversions are existing in a digital design phase, it is mandatory to ensure the correctness of each conversion. In addition, to verify an RTL code, a verifier often writes the behavioral code of a design, and gives the same inputs to the RTL and behavioral code and check whether the outputs of RTL and behavioral model are the same. For a large design, the exhaustive simulation is impossible, so the practical way to do this is the random simulation. However, some special scenarios cannot be detected by the random testing. If equivalence checking replaces random simulation, such comparison will be more reliable. Because of the above reasons, equivalence checking becomes quite useful in hardware verification.

Equivalence checking can be divided by two categories: one is combinational

equivalence checking, the other is sequential equivalence checking.

Combinational equivalence checking is based on Bryant's ROBDD [6] which represents a circuit as a binary decision diagram. Bryant proved that a circuit can be described as a reduced binary diagram which is a canonical form, i.e. every circuit (function) has a unique representation. Hence, equivalence checking simply involves testing whether the two binary diagrams match exactly. This method has been efficiently applied in almost every equivalence checking tool. Some commercial tools have also been used in industries to verify the equivalence between RTL and gate-level circuits. However, since the current design are mainly clock-driven synchronized design, to perform the combinational equivalence checking between two different level sequential circuits, we have to cut the designs into pieces, and map each latch (i.e. register or flip-flop) of one level design into another, and compare their combinational circuits between every two consecutive latches. Therefore, combinational equivalence cannot be applied to compare a RTL design with its behavioral one because their internal latches cannot be mapped correspondingly.

Instead, sequential equivalence checking can be used to verify the equivalence between two level designs without latch mapping, that means it can be applied to verify the equivalence between a RTL design and its behavioral model. But the drawback of sequential equivalence checking is the state space explosion problem so that it is hard to be applied in a large design.

To make use of sequential equivalence checking, we apply modular sequential equivalence checking on the verification of Fairisle ATM switch. Modular sequential equivalence checking is to verify the equivalence between behavioral submodule and RTL submodule or the RTL submodule and synthesized gate-level submodule. Although such equivalence checking cannot ensure the overall behavior of a whole system, a reliable submodule will reduce the effort in a higher-level module verification. The overall behavior can be further verified by model checking or simulation.

In this chapter, we will use sequential equivalence checking to verify each

submodule of Fairisle ATM switch fabric first. In addition, we will apply both combinational equivalence checking and sequential equivalence checking to verify a famous implementation: the concentrator block of Knockout ATM switch.

6.1 Equivalence checking of Fairisle ATM switch fabric

Since we did post-design verification of the switch fabric, we attempted sequential equivalence verification between the Verilog structural description (which we translated from the original Qudos HDL implementation), and the Verilog RTL description of the fabric based on its FSM behavior specification. If both descriptions are equivalent, the correctness of the fabric is proved. We first provided a complete behavior description of the whole switch fabric as one module and tried to verify its equivalence against the implementation of the whole fabric including all connections of submodules. However, we could not succeed in verifying it in VIS after three days continuous run on SUN SPARC 20 workstation (256M / 75MHz) due to state space explosion, even though we used dynamic ordering. We hence followed a second approach that modularizes the fabric to several parts that were similar to the hierarchical modules of the structural description, where each module would be, in addition, described in terms of its behavior specification. This second approach has the shortcoming that while we are able to check the correctness of separate submodules of the fabric structure, it is difficult to ensure the correctness of the network connecting all the submodules.

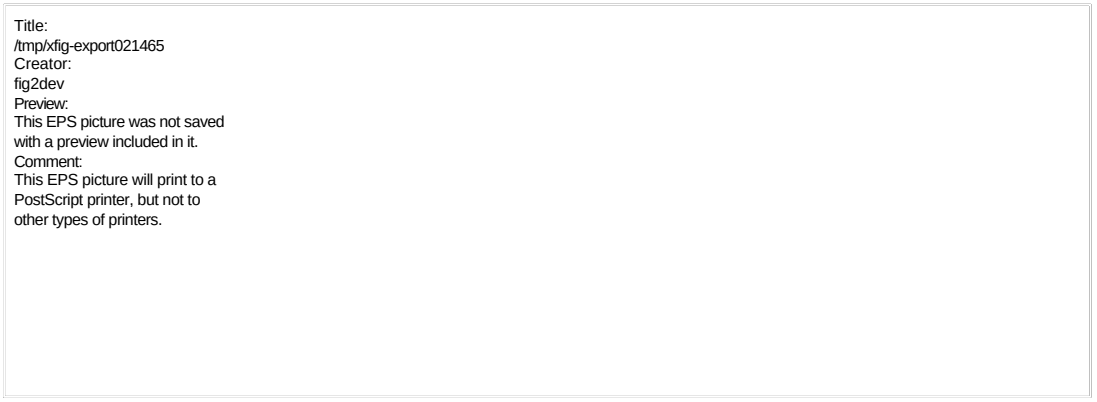


Figure Chapter 6 .1 The modular structure of the switch fabric

Figure Chapter 6 .1 represents the hierarchical structure of the fabric for which we provided RTL description for each submodule. Before we go through the modular verification, we use the timing module, which is submodule in Fairisle ATM switch fabric, to demonstrate how the equivalence checking works.

6.1.1 The Timing module: An example

The timing module determines when the arbitration unit is triggered. This module can be described as a finite state machine shown in Figure Chapter 6 .2. The *routeEnable* signal, which enables the arbitration unit, is normally low. After the *frameStart* signal is asserted, the *routeEnable* signal will keep low until any of active bits (*anyActive* signal), which indicates start of a cell transfer, goes high. Once *anyActive* is asserted, the *routeEnable* will have a strobe at the next clock cycle, and then remain low until the end of frame.

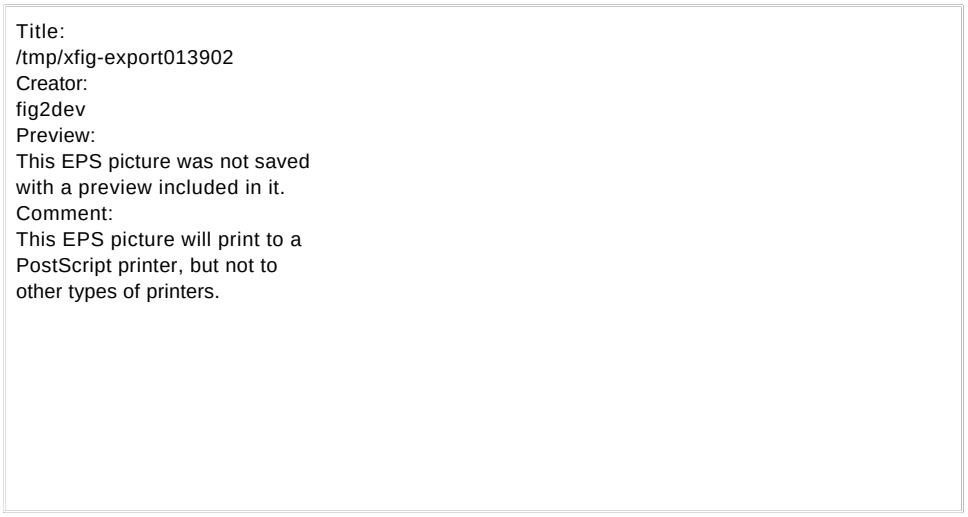


Figure Chapter 6 .2 State transitions of the timing module

According to Figure Chapter 6 .2, we wrote the following Verilog RTL description for the timing module:

```
1.    typedef enum {RUN, WAIT, ROUTE} timing_state;
```

```
2.    module TIMING(frameStart,clock,active0,active1,
          active2, active3,routeEnable);
3.    input frameStart, clock;
4.    input active0, active1, active2, active3;
5.    output routeEnable;
6.    timing_state reg state;
7.    wire anyActive;
8.    assign anyActive = active0 ||  active1 || active2
                      || active3;
9.    assign routeEnable = (state == ROUTE) ;
10.   initial state = RUN;
11.   always @(posedge clock)  begin case (state)
12.   RUN: if ( frameStart == 1)  state = WAIT;
13.   WAIT: if ( ( frameStart == 0) && (anyActive == 1))
                state = ROUTE;
14.   ROUTE: begin if ( frameStart == 0) state = RUN;
15.   else state = WAIT; end
16.   endcase end
17.   endmodule
```

On the other hand, the Verilog translation of the Qudos structural description of the timing module looks like follows:

```
1.    module TIMING(frameStart,clock,active0,active1,active2,
          active3,routeEnable);
1.    input frameStart, clock;
2.    input active0, active1, active2, active3;
3.    output routeEnable;
4.    wire xBar, dx, dy, yterm, anyActive, frameStartBar;
5.    reg routeEnable, y;
6.    initial begin routeEnable = 0; y = 0; end
7.    not InvX (xBar, routeEnable);
8.    or OrAnyActive (anyActive, active0, active1, active2,
                  active3);
9.    not InvFS (frameStartBar, frameStart);
10.   and AnyTerm (yterm, xBar, y);
11.   and AndDx (dx, xBar, y, anyActive, frameStartBar);
```

```
12.   or OrDy (dy, yterm, frameStart);
13.   always @(posedge clock)
14.   begin routeEnable = dx; y = dy; end
15.   endmodule
```

The equivalence checking between the above structural and RTL descriptions was done automatically in VIS. We have been able to prove that the two networks (descriptions) are sequentially equivalent.

6.1.2 Experimental Results on Equivalence Checking

The equivalence checking of Fairisle switch fabric is done by comparing RTL module with structural module. We were able to perform sequential equivalence on module *In_latches*, *Out_latches*, *Pause*, *Timing*, *Priority_decode* and *Arbiters*. And also we can verify the combinational equivalence of module *acknowledgment* (see Table Chapter 6 .1). But we failed in verifying in modules *Dataswitch*, *Pause_dataswitch* and *Switch_fabric* in VIS even if we applied dynamic ordering. Table Chapter 6 .1 gives the CPU time through equivalence checking, and the number of latches in the modules is indicated as well.

Besides dynamic ordering, we tried to build an environment state machine to restrict the behavior of input variables in order to enhance equivalence checking. We also used different partition and image computation methods, but all these methods did not speed-up the equivalence checking much.

Component	CPU time (seconds)	Number of latches
Acknowledgment	1	0
In_latches	2	32
Out_latches	2	32
Pause	2	32
Arbiter_I	1	3
Arbiters	13	12

Priority_decode	27	16
Timing	1	2
Dataswitch_I	1855	16
Arbitration	67860	30
Dataswitch	-	64
Pause_dataswitch	-	96
Switch_fabric	-	190

Table Chapter 6 .1 equivalence checking of each submodule

6.1.3 Analysis on sequential equivalence checking

In general, the more latches a module has, the more expensive is the sequential equivalence checking. When the number of latches in one module reaches a certain value, the CPU time of the sequential equivalence checking rises a lot. The following experiment shows this. Module *Dataswitch_i* consists of four DMUX units. Table Chapter 6 .2 compares the CPU time of equivalence checking among four similar modules with different DMUX units. The *Dataswitch_i* module, which consists of four DMUX units, is composed of only 4 latches more than the module with three DMUX units. However, the CPU time of equivalence checking is increased from 9.7 seconds to 1855.8 seconds. Therefore, it is almost impossible to check equivalence for module *Dataswitch*, *Pause_dataswitch* and *Switch_ fabric* because they contain much more latches than module *Dataswitch_i*.

Learning from the above experiment, we think that the sequential equivalence should be only applied on small modules. Through more experiments, we found that sequential equivalence checking could be done with acceptable time if the number of latches in a model is less than 20 in SUN Sparc 20 (75 MHz/256 MB).

Components	CPU time (seconds)	Number of latches

Module with 1 DMUX	1	4
Module with 2 DMUX	7	8
Module with 3 DMUX	10	12
Module with 4 DMUX	1856	16

Table Chapter 6 .2 Equivalence checking among modules with different DMUX units

6.1.4 Error detection with equivalence checking

No errors were discovered after equivalence checking on each submodule. For experimental purposes, however, we detected the same five errors described in Section 4.6 by equivalence checking. VIS generated counterexamples that exhibited the incorrect behavior of the corresponding signals. Experimental results are reported in Table 9.

Experiments	Affected submodules	CPU time (seconds)
Error 1	Arbiters	21
Error 2	Priority_decode	24
Error 3	Acknowledgment	2
Error 4	Priority_decode	55
Error 5	In_latches	1

Table Chapter 6 .3 Error detection in equivalence checking of submodules

6.2 Equivalence checking of Knockout switch concentrator

The Knockout switch is an N-input N-output packet switch with all inputs and outputs operating at the same bit rate. Fixed-length packets arrive on the N-input in a time-slotted fashion, with each packet containing the address of the output port. Knockout switch has application in both datagram and virtual circuit packet networks.

Aside from having control over the average number of packet arrivals destined for a given output, we assume no control over the specific arrival time of packets on the inputs

and their associated output addresses. In other words, there is no time-slot specific scheduling that prevents two or more packets from arriving on different inputs in the same time slot destined for the same output. Hence, to avoid (or at least provide a sufficiently small probability of) lost packets, at a minimum, packet buffering must be provided in the switch to smooth fluctuations in packet arrivals destined for the same output.

The interconnection fabric for the Knockout Switch has two basic characteristics: 1) each input has a separate broadcast bus, and 2) each output has access to the packets arriving on all inputs. Figure Chapter 6 .3 (a) illustrates these two characteristics where each of the N inputs is placed directly on a separate broadcast bus and each output passively interfaces to the complete set of N buses. This simple structure provides us with several important features.

First, with each input having a direct path to every output, no switch blocking occurs where packets destined for one output interfere with (i.e., delay or block) packets going to other outputs. The only congestion in the switch takes place at the interface to each output where, as mentioned, packets can arrive simultaneously on different inputs destined for the same output. Without a priori scheduling of packet arrivals, this type of congestion is unavoidable, and dealing with it typically represents the greatest source of complexity within the packet switch. The focus of the Knockout Switch architecture is one of minimizing this complexity.

In addition to the above property, the switch architecture is modular: the N broadcast buses can reside on an equipment backplane with the circuitry for each of the N input/output pairs placed on a single plug-in circuit card. Hence, the switch can grow modularly from 2 x 2 up to N x N by adding additional circuit cards.

6.2.1 Architecture of Knockout ATM Switch

Figure Chapter 6 .3 b illustrates the architecture of the bus interface associated with each output of the switch. The bus interface has three major components: packet filter, concentrator and shifter buffer. At the top of Figure Chapter 6 .3 b, there are a row of N

packet filters. Here the address of every packet broadcast on each of the N buses is destined, with packets addressed to the output allowed to pass on, to the concentrator and all other blocked. The concentrator then achieves an N to L (L << N) concentration of input lines, wherein up to L packets making it through the packet filters in each time slot will emerge at the outputs of the concentrator. These L concentration outputs then enter a shared buffer composed of a shifter and L separate FIFO buffers. The shared buffer allows complete sharing of the L FIFO buffers and provides an equivalent of a single queue with L inputs and one output. Operating under a first-in first-out queuing discipline.

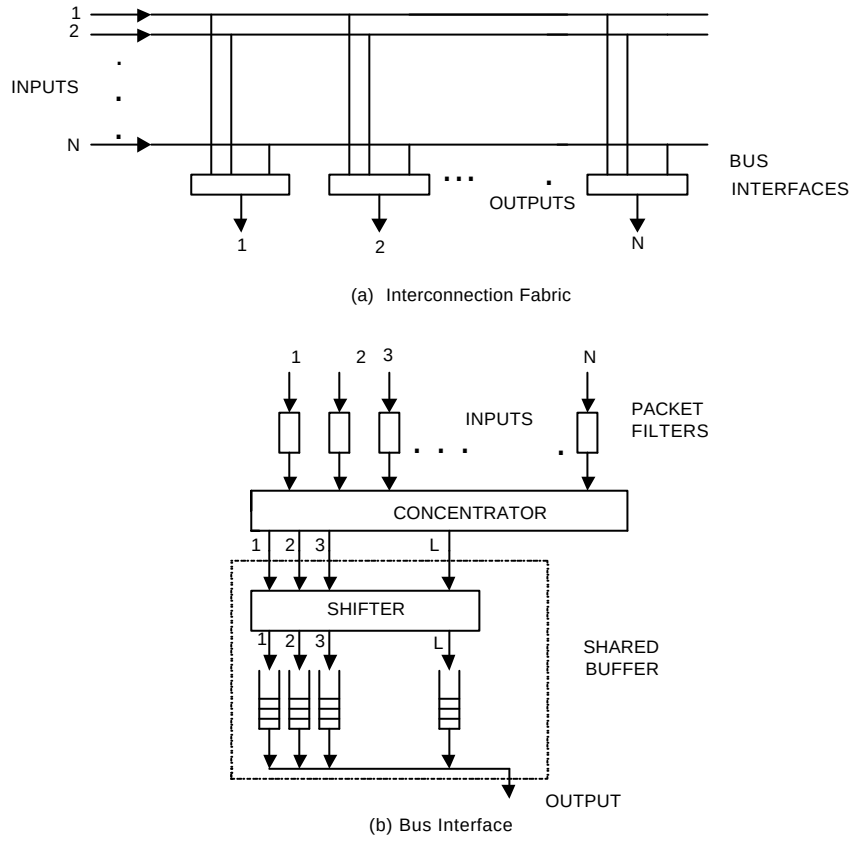


Figure Chapter 6 .3 Structure of Knockout ATM Switch

In Figure Chapter 6 .3 (b), the circuits for packet filter and shifter are very simple and common, and they are quite similar to other ATM switches. However, the concentrator circuit is very special. Specifically, the function of the concentrator is : if there are k packets

arriving at a time slot for the same output address, these k packets, after passing through the concentrator, will emerge from the concentrator on outputs 1 to k, when $k \leq L$. If $k > L$, all L outputs of the concentrator will have packets, and k-L packets will be dropped (i.e. lost) within the concentrator.

The basic building block of the concentrator is a simple 2 x 2 contention switch shown in Figure Chapter 6 .4(a). The two inputs contend for the "winner" output according to their activity bits. If only one input has an arriving packet (indicated by the active bit = 1), it is routed to the winner (left) output. If both inputs have arriving packets, one input is routed to the winner output and the other input is routed to the loser output. If both inputs have no arriving packets, we do not care except that the active bit for both should remain at logic 0 at the switch outputs.

The above requirements are met by a switch with the two states shown in Figure Chapter 6 .4(b). Here, the switch examines the active bit for only the left input. If the active bit is a 1, the left input is routed to the winner output and the right input is routed to the loser output. If the active bit is a 0, the right input is routed to the winner output, and no path is provided through the switch for the left input.

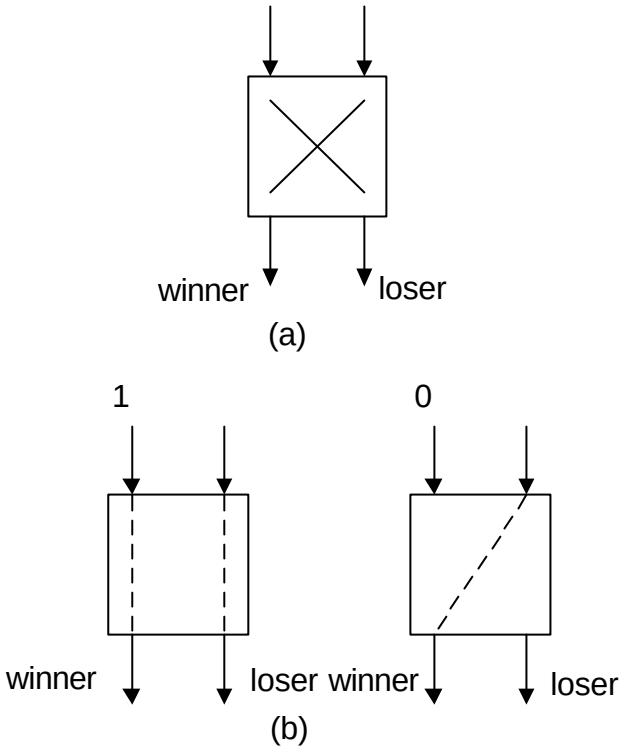


Figure Chapter 6 .4 (a) The 2 X 2 contention switch (b) State of a x 2 contention switch

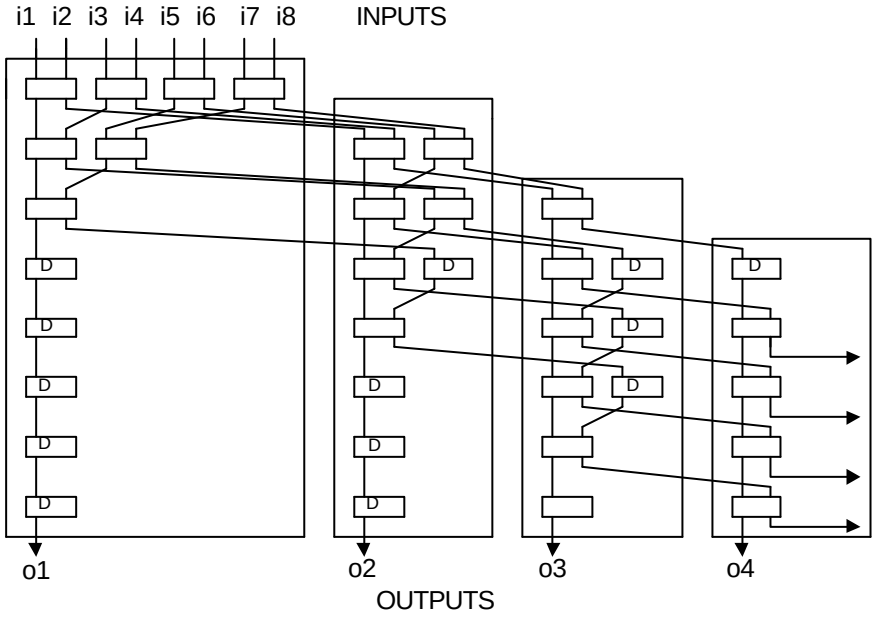


Figure Chapter 6 .5 The 8-input / 4-output concentrator

Figure Chapter 6 .5 shows the design of an 8-input/4-output concentrator composed of these simple 2 x 2 switch element and single-input/single-output 1-bit delay element (marked by "D"), and it can be easily expanded to N-input/L-output concentrator with the same two elements, so it has the advantage of easy implementation. However, the output function with inputs is not easily deduced from the circuits, and also the connection is complex so that the mistakes tend to hide in the implementation. For these two reasons, we need to verify the above structure. In the following section, we give our methods to verify the circuit by the equivalence checking.

6.2.2 Equivalence Checking of the Concentrator

To verify the structure of the concentrator, we use equivalence checking in VIS. Although the structure of Knockout concentrator (Figure Chapter 6 .5) seems complicated, its specification is clear: if there is only one active cell in the input ports, it will be transmitted to the most left output port; if there are only two active cells in the input ports, they will be transmitted to the two most left output ports; if there are three active cells in the input ports, they will be transmitted to the three most left output ports; if there are four or more than four, they will be transmitted to all the four output ports. The specification can be expressed as Figure Chapter 6 .6.

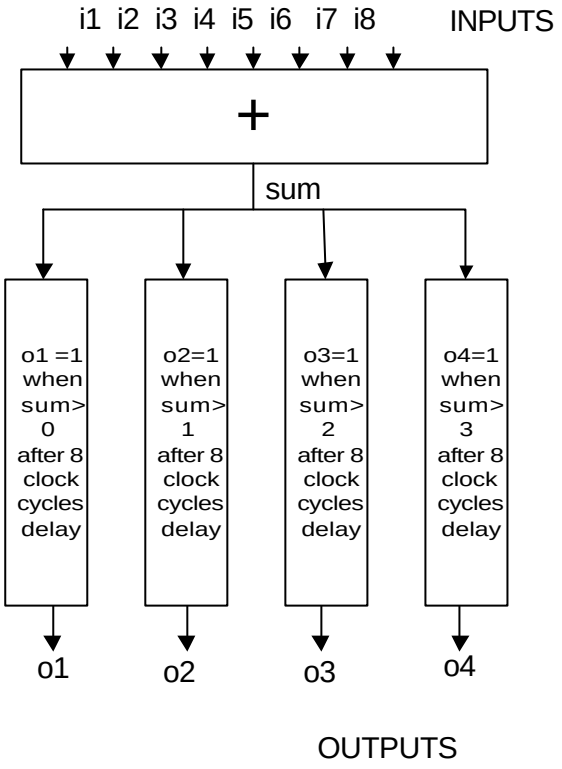


Figure Chapter 6 .6 The specification of the concentrator specification

In Figure Chapter 6 .6, *i1, i2, …, i8* express eight inputs of the concentrator. *ix = 1* means there is a cell at input *x (x=1, 2, …, 8)*, and *ix = 0* indicates there is no cell at input *x*. The signal *sum* means the sum of the 8 inputs, so the number of *sum* indicates the number of inputs which have cells to send. Because there are 8 clock cycles delay in the structural model of Knockout concentrator (Figure Chapter 6 .3), we also give 8 clock cycles delays in the specification model in order to make structural model and the specification model comparable.

We described both structural and specification Knockout concentrator model in Verilog. Because there were 64 latches inside the Knockout concentrator, we adopted sequential equivalence checking on them. We did sequential equivalence checking in Sun UltraSparc (300MHz/512MB) workstation. However, we could not get the results because the verification was crash with failing in allocating memories.

We noticed that the latches in the structural Knockout concentrator are only for distributing the operation which could be done with one clock cycle into eight clock cycles to complete so that the clock frequency of the whole system could be higher. In other words, these latches do not have the function of state transition. Therefore, reducing these latches will not have an influence on the function of the design.

After the latches are taken away from the Knockout concentrator, it becomes a combinational circuit. So combinational equivalence checking is possible to be applied. Practically, it is fairly easy to reduce the latches in its Verilog module. In the structural Knockout concentrator description, we only need to modify the two submodules: 2x2 switch element and D flip-flop while keeping the same concentrator structure. In the specification Verilog module, we only need remove the eight clock cycles delay.

The combinational equivalence checking on Knockout concentrator was very efficient, and we got the result immediately after we submitted the verification job in VIS.

6.2.3 Experimental results and discussion

The knockout concentrator is composed of 64 latches. Table Chapter 6 .4 shows the experimental results for the equivalence checking in combinational equivalence checking and sequential equivalence checking.

Method	combination equi. chk.	sequential equi. chk.
CPU time (sec.)	32	-
Mem. usage (MB)	45	-
Nodes allocated(K)	32,476	-

Table Chapter 6 .4 Experimental results of the equivalence checking on the concentrator

In Table Chapter 6 .4, combinational equivalence checking used little CPU time and memory usage to verify the Knockout concentrator which sequential equivalence checking was not able to verify. This is the reason that sequential equivalence checking has not been used in the industries while combinational equivalence checking is used in industrial ASIC design verification such as the correctness of synthesis.

6.3 Summary

This chapter introduced combinational equivalence checking and sequential equivalence checking. Combinational equivalence checking can handle a relatively large design, and with the help of some techniques such as latch mapping, it also can partially verify a sequential circuit. Sequential equivalence checking is to verify the equivalence of a sequential circuit between behavioral model and RTL model which is almost impossible to be done in combinational circuit.

By using the combination of combinational equivalence checking and sequential equivalence checking, we succeeded in verifying the equivalence of each submodule in Fairisle ATM switch fabric between its structural model and RTL model. To make good use of sequential equivalence checking, we proposed modular verification in the verification of Fairisle ATM switch fabric. In addition, we used Knockout concentrator as an example to illustrate how to build a specification model to verify a special structure. Furthermore, in order to make advantage of combinational equivalence checking, we demonstrate how to transfer a sequential equivalence checking problem into a combinational equivalence checking problem.

Chapter 7

Conclusions

In this thesis, we have described methods for formally verifying ATM hardware by model checking and equivalence checking. We proposed property division which is based on compositional reasoning. We showed how to establish an environment, and how to use CTL to express a property. We also proposed modular verification in terms of sequential equivalence checking, and demonstrate how to build a specification model and transfer a sequential verification problem into a combinational verification problem. To demonstrate that our techniques were practical, we used our methods to verify several academic and commercial ATM hardware designs. During the verification process, we discovered errors in these designs.

While we have considered a number of real ATM designs, we gain some experience in applying formal methods on ATM hardware. We feel that it is particularly important to look at a single system across several levels of abstraction. Recall that in the model checking of the Fairisle ATM switch fabric, we constructed the environment which is an abstraction or reduction model of the port controller. As such, we were able to verify the entire ATM switch by property division method. However, to correctly partition a design and build a property environment, a verifier has to have in-depth knowledge on the design.

The most significant drawback of automatic formal verification is that it cannot handle a large design, so simulation is still a very important method for hardware verification. However, we usually use a hierarchical-verification style in industry. If we look at a chip, it is divided into several modules, and each module is further divided into many blocks. So the design verification will include block-level verification, module-level

verification and top-level verification. The more adequate the lower level verification is, the easier the upper verification will be. Therefore, if we can apply formal verification into the module-level or block-level verification, it will efficiently improve the verification quality because it can verify a small (or medium) design rapidly and exhaustively. In our work, we could efficiently verify a medium size design by the combination of equivalence checking and model checking with a set of methods we proposed in the thesis.

There are also a number of questions that we would like to address. One question concerns the design modification we have done during the formal verification process. Once a design is done, we do not want to manually modify it for the verification purpose because it is possible to introduce another error during the modification. To avoid the manually modification, we hope to develop some tools to do the abstraction, reduction and create some verification assistant variables.

Another issue is to make environment and CTL formulas generation automatically, and a verifier only needs to give assumptions and conclusions. In this work, we created all the environments and CTL formulas manually. But, we believe that an automatic tools could be developed to do these because we found there were some rules existing in their generation. Also, either Environment Modification or Internal Signal Usage could be embed in such automatic tools.

Since a design is getting bigger and bigger, and the verification is extremely difficult. How to make a design to be easily verified becomes another issue. Recall section 0.4.2, we use fabric unit in the verification instead of abstracted fabric, the model checking becomes more efficient. So we could develop a set of proper rules for a certain design, the design based on such rules will be easy to verify using formal methods.

Finally, state space explosion is the most important problem in model checking and sequential equivalence checking. We hope good algorithms will be developed so that formal verification tools can handle much larger designs in the future. With the efforts in the above aspects, we believe that formal verification will play a very important role in hardware verification in the future.

Bibliography

[1] Aziz A., Tasiran S., and Brayton R.: BDD Variable Ordering for Interacting Finite State Machines. In Proc. of the Design Automation Conf., pages 283-288, SanDiago, CA, June 1994.

[2] A. Gupta, "Formal Hardware Verification Methods: A Survey", Formal Methods in System Design, Vol. 1, pp. 151-238, 1992.

[3] Ashenden, P. : The Designer's Guide to VHDL, Morgan Kaufmann Publishers, San Francisco, California, 1994.

[4] Büchi J.: On a decision method in restricted second order arithmatic. In Proceeding of the 1960 International Congress on Logic, Methodology, and Philosophy of Science, Standford University Press, 1962.

[5] Brayton R. et.al.: VIS: A System for Verification and Synthesis, Technical Report UCB/ERL M95, Electronics Research Laboratory, University of California, Berkeley, December 1995.

[6] Bryant R.: Graph-Based Algorithms for Boolean Function Manipulation; IEEE Transactions on Computers, Vol. C-35, No. 8, pp. 677-691, August 1986,.

[7] Browne M., et. al: Automatic verification of sequential circuits using temporal logic IEEE transactions on Computers, Vol. C-35, No. 12, pp. 1035-1044, 1986.

[8] Chen, B.; Yamazaki, M.; Fujita, M.: Bug Identification of a Real Chip Design by Symbolic Model Checking; Proc. International Conference on Circuits And Systems

(ISCAS'94), London, UK, pp. 132-136, June 1994.

[9] Burch J.R., Clark E.M., et.al: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits. Vol. 13, No. 4, pp. 401-424, April 1994.

[10] Clarke E., et.al: Synthesis of synchronization skeletons for branching time temporal logic, In Logic of Programs: Workshop, Yorktown Heights, NY, Vol. 131 of Lecture Notes in Computer Science, May 1981.

[11] Clarke E. M., Grumberg O., McMillan K. L., and Zhao X: Efficient generation of counterexamples and witnesses in symbolic model checking In Proc. 32nd Design Automat. Conf., pp. 427-432, June 1995.

[12] Clarke E.M., Emerson E.A., and Sistla A. P: Automatic verification of finite-state concurrent system using temporal logic specifications. ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, pp. 244-264, 1986.

[13] Corella, F.; Zhou, Z.; Song, X.; Langevin, M.; Cerny, E: Multiway Decision Graphs for Automated Hardware Verification; Formal Methods in System Design, Vol. 10, No. 1, pp. 7-46, 1997.

[14] Curzon, P.: The Formal Verification of the Fairisle ATM Switching Element; Technical Reports No. 328 & No. 329, University of Cambridge, Computer Laboratory, March 1994.

[15] Dan, V. ; Cerny, E.; Song, X. : The formal verification a Fairisle ATM switch port controller; Technical Report, University of Montreal, 1998.

[16] Edgcombe, K.: The Qudos Quick Chip User Guide; Qudos Limited.

[17] Fujii, Ootomo G., and Hori G.: Interleaving based variable ordering methods for ordered binary decision diagrams. In Proc. Intl. Conf. On Computer-Aided Design, pp. 38-41, Nov. 1993.

[18] Garcez E.: "The Verification of an ATM Switching Fabric using the HSIS Tool", Technical Report, WSI-95-13, Tübingen University, Germany, 1995

[19] Gordon, M.; Melham, T.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic; Cambridge, University Press, 1993.

[20] Kurshan R. P., Analysis of discrete event coordination. Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems. Models. Formalisms. Correctness. Vol. 430 of Lecture Notes in Computer Science. Springer-Verlag. May 1989.

[21] Langevin, M.; Tahar, S.; Zhou, Z.; Song, X.; Cerny E.: Behavior Verification of an ATM Switch Fabric using Implicit Abstract State numeration; Proc. IEEE International Conference on Computer Design (ICCD'96), Austin, Texas, USA; pp. 20 - 26, October 1996.

[22] Leslie, I. and McAuley, D.: Fairisle: An ATM Network for the Local Area; ACM Communication Review, Vol. 19, No. 4, pp. 327-336 September 1991,.

[23] Long, D.: Model Checking, Abstraction, and Compositional Verification; PH.D thesis, July 1993.

[24] Lu, J. and Tahar, S.: On the Formal Verification and Reimplementation of

an ATM Switch Fabric Using VIS; Technical Report No. 401, Concordia University, Dept. of ECE, September 1997.

[25] McDysan, D., Spohn, D.: ATM Theory and Application, McGraw-Hill series on computer communications, New York, 1994.

[26] McMillan, M.: Symbolic Model Checking; Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[27] PMC-sierra, Inc: ATM layer routing control, monitoring and policing 800 Mbps, PMC-940904, August 1997.

[28] Pnueli A.: In transition for global to modular temporal reasoning about programs. In K.R. Apt, editor, Logics and Models of Concurrent Systems, Vol. 13 of NATO ASI series. Series F, Computer and system science. Springer-Verlag, 1984.

[29] Quielle J.P. and Sifakis J: Specification and verification of concurrent systems in CESAR. In Proceedings of the Fifth International Symposium in Programming, 1981.

[30] Rajan S., Fujita M., Yuan K., Lee M.: High-Level Design and Validation of ATM Switch; Proc. IEEE International High Level Design Validation and Test Workshop (HLDVT'97), Oakland, California, USA, November 1997.

[31] Ranjan R., Aziz A., Plessier B., Pixley C., and Brayton R.: Efficient Formal Design Verification: Data Structure + Algorithms. Technical Report UCB/ERL M94/100, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Oct. 1994.

[32] Schneider K. and Kropf T.: Verifying Hardware Correctness by Combining

Theorem Proving and Model Checking, In: J. Alves-Foss(editor), International Workshop on Higher Order Logic Theorem Proving and Its Applications (B-Track), pp. 89-104, August 1995.

[33] Sentovich E., Singh E., Lavagno E., Moon E., Murgai R., Saldanha A. Savoj H., Stephan P., R Brayton R., and Sangiovanni-Vincentelli A: SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[34] Smith G., Bahnsen R., and Halliwell H: "Boolean comparison of hardware and flowcharts", IBM Journal of Research and Development, Vol. 26, pp. 106-116, January 1982.

[35] Smith, Michael J. S.: "Application-specific integrated circuits", published by Addison Wesley Longman, Inc., ISBN 0-201-50022-1, 1997.

[36] S. Owre, N. Shankar, aand J. Rushby, "User Guide for the PVS Specificationand Verification System, Language, and Proof Checker", Computer Science Laboratory, SRI International, Melno Park, California, February 1993.

[37] Tahar, S.; Zhou, Z.; Song, X.; Cerny, E.; Langevin, M.: Formal Verification of an ATM Switch Fabric using Multiway Decision Graphs; Proc. IEEE Sixth Great Lakes Symposium on VLSI (GLS-VLSI'96), Ames, Iowa, USA, IEEE Computer Society Press, pp. 106-111, March 1996.

[38] Tahar, S.; Curzon, P.; and Lu, J.: "Three approaches to Hardware Verification: HOL, MDG and VIS Compared; In: Gopalakrishnan, G. and Windley, P. and

Windley, P. (Eds.), Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science 1522, Springer Verlag, 1998, pp. 433-450. Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD'98), Palo Alto, California, USA, November 1998.

[39] Yeh Y.S., Hiuchyj M., and Acampora A.: "The knockout switch: a simple modular architecture for high performance packet switching." IEEE J. Selected Areas in Communications, Vol. SAC-5, No. 8, pp. 1274-1283, Oct. 1987.

Appendix A

Overview of ATM Switches

In this section, we will review and highlight the key features of each switch architecture. We begin by dividing the different fabrics into two families according to the physical connections between input and output ports of the switch fabric, namely, time and space switching. The two families are further divided into separate classes, whereby each structure can be presented systematically.

A.1 Time Division Switches

As shown in Figure Appendix A .1, a switch can be regarded as a communication resource which is shared by all input and output ports. In the time division switches, the access to this resource, which can be a shared memory or a shared medium (such as a bus or a ring), is via a time division multiplexing scheme, where all ports transmit according to a common time reference. There are two interesting features in the time division switches. The first is the existence of a linear relation between cost and complexity of the switches. The second is multicasting/broadcasting function which can be easily incorporated due to the shared element.

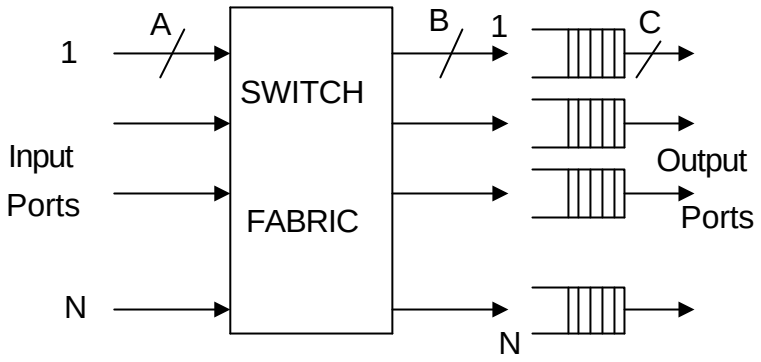


Figure Appendix A .1 Structure of ATM Switch

- **Shared Memory**

The switch architectures belonging to this family consist of a memory shared by all input and output lines (an example of a shared memory switch is shown in Figure Appendix A .2). The packets arriving on all input lines are multiplexed into a single stream which is fed to the common memory. There are two possibilities to implement the common memory. In the first method, the packets are randomly written to the memory and sequentially are read out to the output port. The second method consists of sequentially writing the packets into the memory and randomly read them out to the output port (This is the method commonly used with RAMs). Finally, the output streams de-multiplexed, and the packets are transmitted on the output lines.

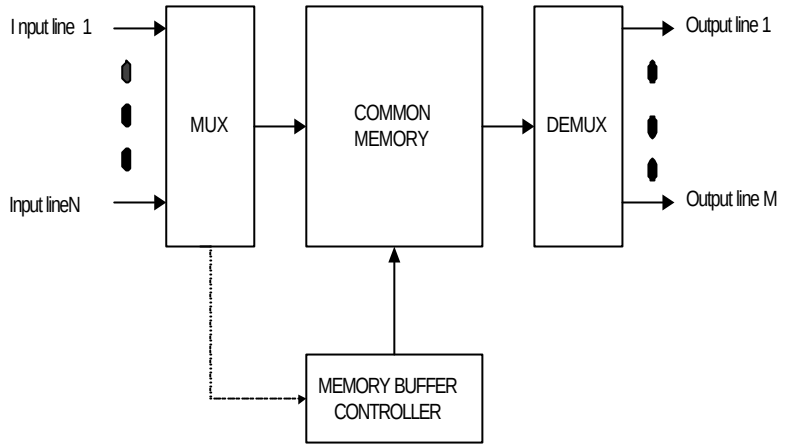


Figure Appendix A .2 The structure of shared memory ATM switch

Share memory family is further divided into two classes: complete partitioning of memory and full sharing of memory. In the complete partitioning strategy, the memory is divided into N separate sections, and each forms a queue to an output port. A packet is lost when it arrives to its destination output queue section of the memory and finds the correspondent queue section of the memory full. In the full sharing scheme, the entire memory is shared among all the output ports for queuing purposes. A packet is lost only when it arrives to the memory and finds the memory completely full.

The two main concerns when designing a switch architecture of the shared memory family are processing time and memory bandwidth and size.

- **Shared Medium**

In the shared medium switches (Figure Appendix A .3), all arriving packets on the N input lines are synchronously multiplexed onto a common high-speed medium (which can be a bus or a ring) of bandwidth equal to N times the rate of a single input line. Each output line is connected to the shared medium through an address filter capable of receiving all packets transmitted on the medium by an output buffer. It is the address filter that decides if a packet served on the medium should be written into the corresponding output buffer.

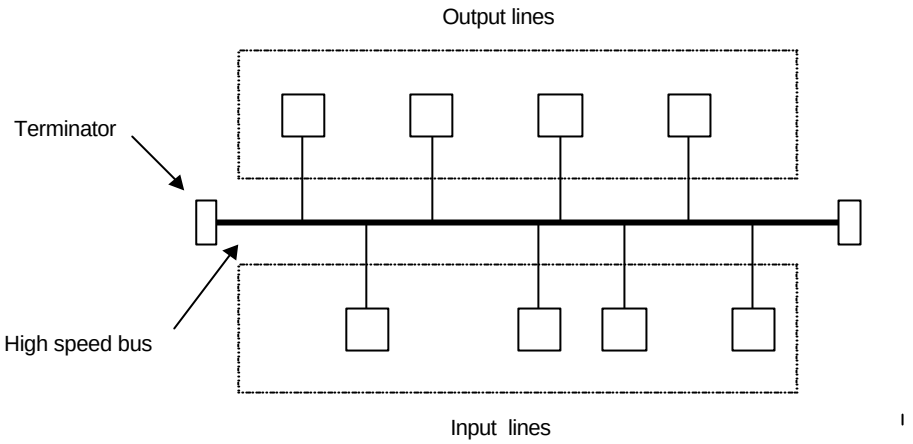


Figure Appendix A .3 The structure of shared bus ATM switch

Shared medium family is further divided into two classes: shared bus and shared ring. In the shared bus architecture, several lines are connected to a high speed bus. Since the bus is a broadcast medium, all the lines can receive the cells, keeping only those that are addressed to it and discarding the rest. In the shared ring architecture, each line is connected to a ring, and the interface modules are interconnected one to anther in a point-to-point fashion that happen to form a circle. Upon reception of a cell, the interface module can copy and remove the cell from circulation (single-casting), copy and retransmit the cell to its neighbor (multicasting/broadcasting) or simply retransmit the cell to its neighbor.

A.2 Space-Division Switch

Space-division switches may be classified into three categories: (i) crossbar fabrics, (ii) banyan-based fabrics, and (iii) fabrics with N^2 disjoint paths. But any space-division switches can be described as a common abstract model (Figure Appendix A .4). The model prescribes that for each input line i there is a router (that is, de-multiplexor) which routes its packets to N separate bins, numbered $(i, 1)$ through (i, N), one bin for each output port. At the output side, we consider that for each output line j there is a concentrator (multiplexor) which connects all bins (i, j), i =1, 2, ..., N, to output line j and

which, in each time slot, selects one packet, if any, from these output bins for transmission on output line j. The various fabrics proposed differ by the ways the routers and concentrators are implemented, and by the locations of buffers.

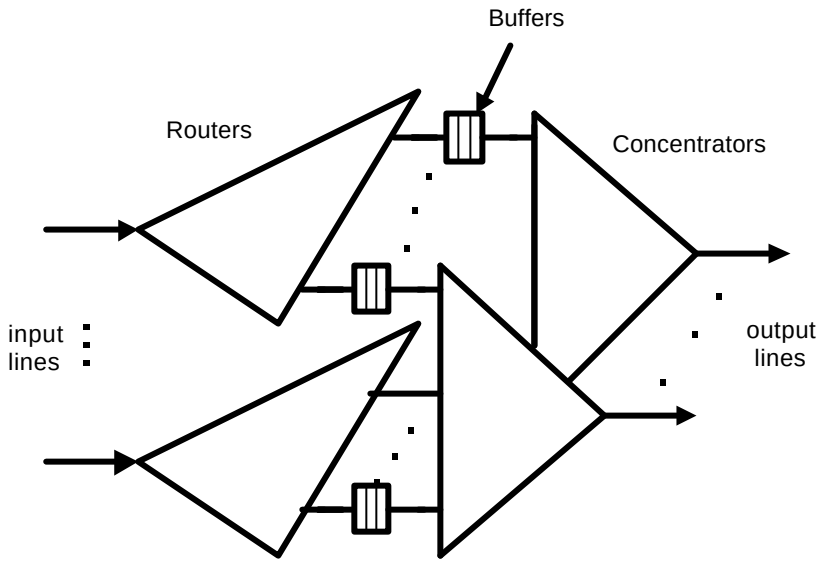


Figure Appendix A .4 Common abstract ATM switch model

- ## Crossbar Fabric

The first electronic space-division switch that came into existence is the one known as the crossbar switch (Figure Appendix A .5), originally introduced for circuit switching. Basically, a crossbar fabric consists of a square array of N^2 cross-point switches, one for each input-output pair.

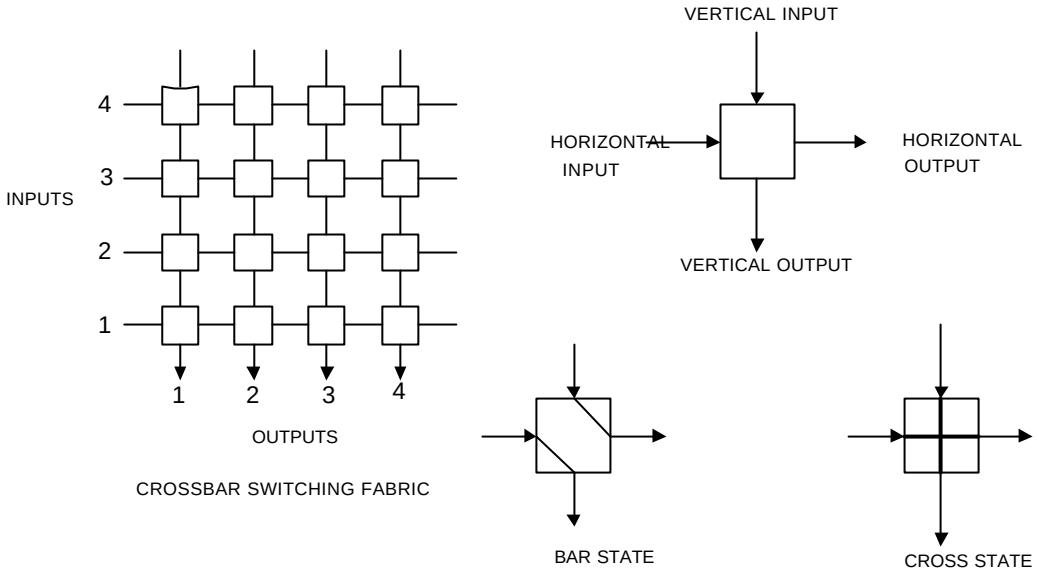


Figure Appendix A.5 Structure of crossbar ATM switch

Closing the switch at the *(i,j)th* cross-point establishes a physical connection between input line i and output line j. That is, as long as input i and output j are free, a connection between them is achieved simply by closing the *(i, j)th* switch. It is easy to see that the existence of N^2 cross-point switches in the crossbar fabric permits N pairs of input/output lines to be connected simultaneously, provided that these pairs are disjoint. If, on the other hand, there is more than one packet in the same slot destined to the same output, then one and only one of the packets can reach the output; the remaining packets will have to be either dropped or buffered somewhere. There are two possibilities for the location of buffers in a crossbar switch: (a) at the cross-points of the switching array, or (b) at the input of the switching array. Each of the two possibilities has its advantages and drawbacks. Placing the buffers at the cross-points, the switches then is work-conversing, and does not suffer the throughput limitation incurred with either dropping packets or input buffering.

However, two drawbacks to this approach. First, the total memory required for a given loss rate is greater than that required for output queuing with complete partitioning,

because the output queue is distributed over N buffers and there can be no sharing among these. The second and more important drawback is the fact that from a hardware layout point of view, the buffer memory typically requires a much larger real-estate than the switching array itself, and combining them both on the same circuit would severely limit the size of the switching fabric implementable on a chip.

Input queuing consists of placing a separate buffer at each input to the switch. A packet arriving at an input line first enters the buffer, and if it cannot proceed due to a conflict, it remains in the buffer waiting to be switched at a later time. Placing the buffers at the input of the switch separates the buffering and switching functions, a very desirable outcome from the point of view of layout and circuit compactness. With this configuration, the implementation of the switching fabric may take different forms. One implementation is to distribute the switching function over all. When a packet reaches a crosspoint that has already been set by an earlier packet, or alternatively loses in its contention for a cross-point to another contending packet, then a "blocking" signal is returned on a reverse path to the input port, the result of which is to block the transmission of the packet and keep the packet in the input buffer for later tries. Another implementation is to centralize the contention resolution function for each output port by providing an arbiter for each port. Each such arbiter sees all packets that are proceeding to the corresponding output, selects one of them according to some rule, and blocks all others by the means of return control signals. By replicating the address decoder at each crosspoint, a single "horizontal" bus is needed per input line on which the packet and its requested output port address are broadcast. Similarly, since only one arbiter may be sending a blocking signal to a given input line at a time, a single reverse control line is required per input line. As we will see in Chapter 4, Fairisle ATM switch fabric belong to the latter implementation.

We now address a number of issues pertaining to the performance and control of input-buffered crossbar switches. The first question concerns that service discipline used in admitting packets queued at the input buffers. The simplest discipline from a control and implementation point of view is first-come-first-served(FCFS) in which only the head of the

line (HOL) of each input queue may content to the switch array. If a packet is successfully switched , then it is removed from the input buffer, and the next in line is considered in the following slot. Otherwise the packet remains at the head of the queue, and contends again in the following slot. Although very simple, the main problem with FCFS is HOL blocking. If there are k packets contending for the same output, only one packet is served in a timeslot, and the remaining $k-1$ packets wait for the next time slot. This also implies that $k-1$ output line would have remained idle in that time slot, despite the fact that there may be packets, queued behind the unsuccessful packets, destined to idle output lines, but which are prevented from reaching their destinations. This situation renders the switch non-work-conserving, driving its throughput below its nominal value. Fairisle null input port controller which will be introduced in Chapter 3 belongs to such FCFS input buffer.

The crossbar fabric has one major intrinsic drawback: it requires N^2 crosspoints, and therefore the size of realizable switches tends to be limited. In addition, it has two undesirable features: (i)when self-routing is used, the processing performed at each crosspoint requires knowledge of the complete output port address; and (ii) the transit time is not constant over all input/output pairs unless artificial delays are introduced at the inputs and outputs of the switch.

- **Banyan-Based Space-Division Switches**

While crossbar fabric has the above drawbacks, banyan based fabrics (Figure Appendix A .6) are multistage interconnection network which have less switching elements than that found in crossbar based fabrics with self routing property.

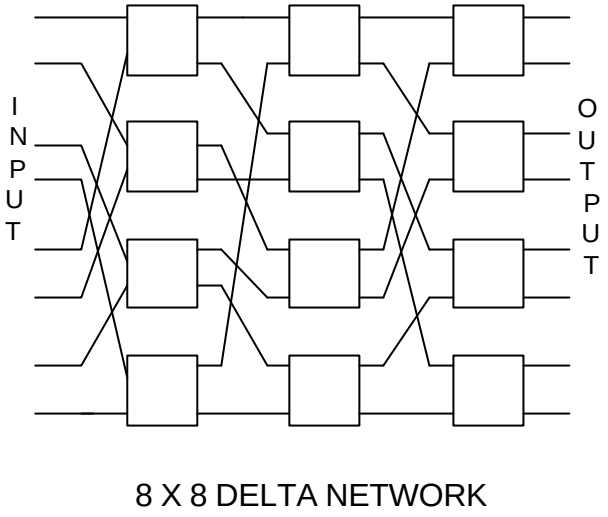


Figure Appendix A .6 Structure of Banyan ATM switch

The main idea of banyan ATM switch was to make more than one input line share the same switching element at the same time, and use multistage interconnection to let each input have at least a path to each output. Through the use of multiple stages, it was possible to reduce the complexity of the switch as compared to crossbar based fabric.

Banyan switching fabrics have some interesting characteristics such as modularity and self-routing. However, banyan fabrics has the major drawbacks such as internal blocking and performance degradation. The first solution consists of placing buffers at the points of conflicts, leading to what is known as the buffered-banyan switching fabric. The second solution consists of using input buffering and of blocking packets at the input by the means of upstream, control signals issued when conflicts occur. Given the extremely low throughput attained in a banyan network, this solution is not very desirable. The third also consists of using input buffering, but also includes the means to improve the throughput of the banyan self-routing network. This consists of sorting the input packets in order to remove output conflict and to present to the banyan (sub) permutations that are guaranteed to pass without conflicts. Packets that are not selected are buffered for later tries. Sorting is accomplished by the means of a Batcher sorter, and thus the resulting fabric is referred to

as the Batcher-banyan switching fabric. Finally, a fourth solution consists of using multiple copies of the banyan interconnection network in parallel or in tandem, thus increasing the number of possible paths between inputs and outputs, and achieving output buffering.

- **Switching Fabric with N^2 Disjoint Paths**

The final set of space-division switching fabrics that we describe in this section consists of fabrics with sufficient hardware resources to permit the establishment of N^2 disjoint paths among the inputs and outputs, and thus achieve output buffering. The most obvious example is the bus-matrix switching architecture. It uses N broadcast input buses, N multi-access output buses, and N^2 crosspoint buffer memories; each crosspoint memory component contains an address filter corresponding to the output bus to which it is connected. Referring back to the abstract model (Figure Appendix A .4), the router for an input line consists here of the input bus along with the N address filters connected to that bus, and the concentrator for an output line is the corresponding multi-access bus.

The other fabric with great similarity to the bus matrix fabric but which achieve output buffering have been used in some industry designs: the knockout switch (Figure Appendix A .7). In the knockout switch, each input port transmits its packets on a broadcast bus to which all output ports are tapped. That is, each output line has a bus interface connecting to all input buses. Such an interface contains N address filters, one for each input line, which recognize packets addressed to the corresponding output line. With the N filters operating parallel, a bus interface is capable of receiving N packets per slot. The outputs of the filters are connected to an N x L concentrator which selects up to L packets out of those accepted by the filters. If more than L packets are destined to the same output line in a given slot, only L are received into the buffer and remaining ones are lost.

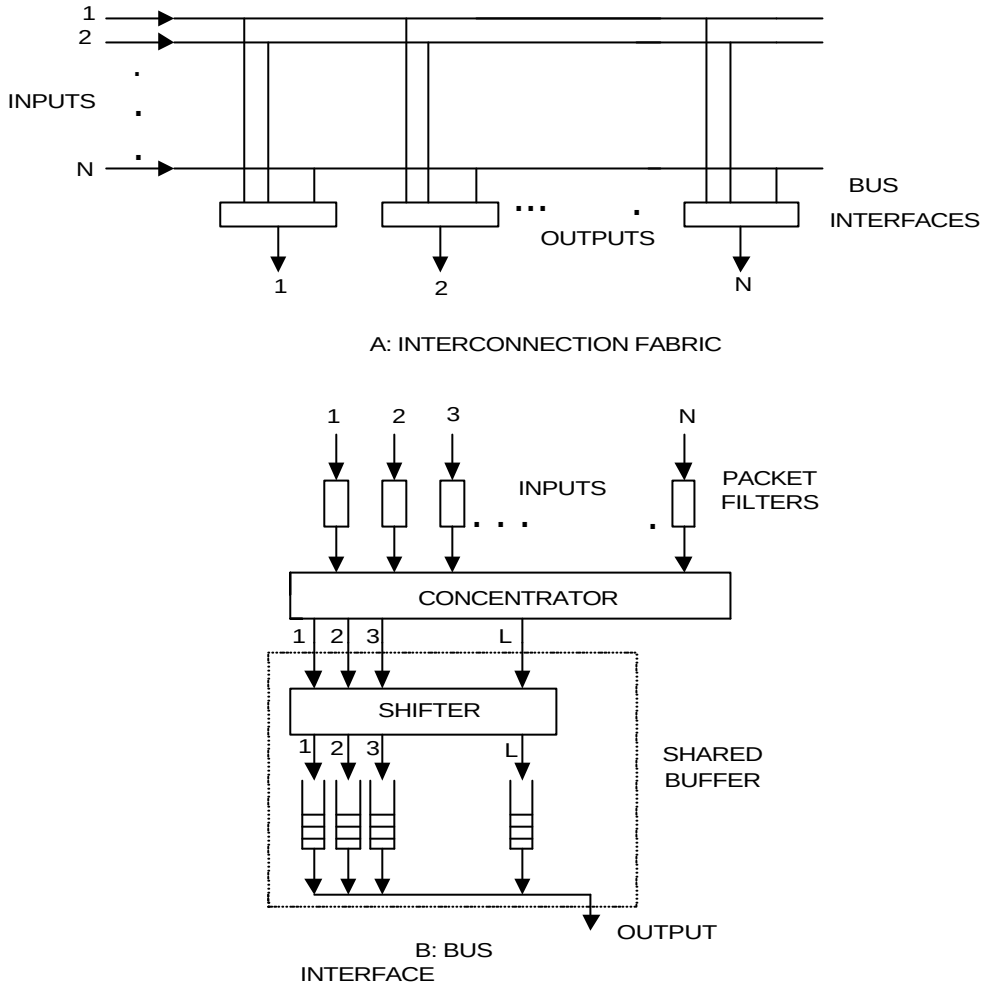


Figure Appendix A .7 Structure of knockout ATM switch

A hardware implementation is proposed for the concentrators following the simple knockout algorithm used in tournaments. Referring back to the abstract model, the router for an input line consists here again of the input bus along with the N address filters connected to that bus; no intermediate buffers are employed; instead output buffering is achieved by the means of the concentrators with a fixed output width. This design is also very similar to the shared bus architecture because both of them have address filters and buffers in each output port.

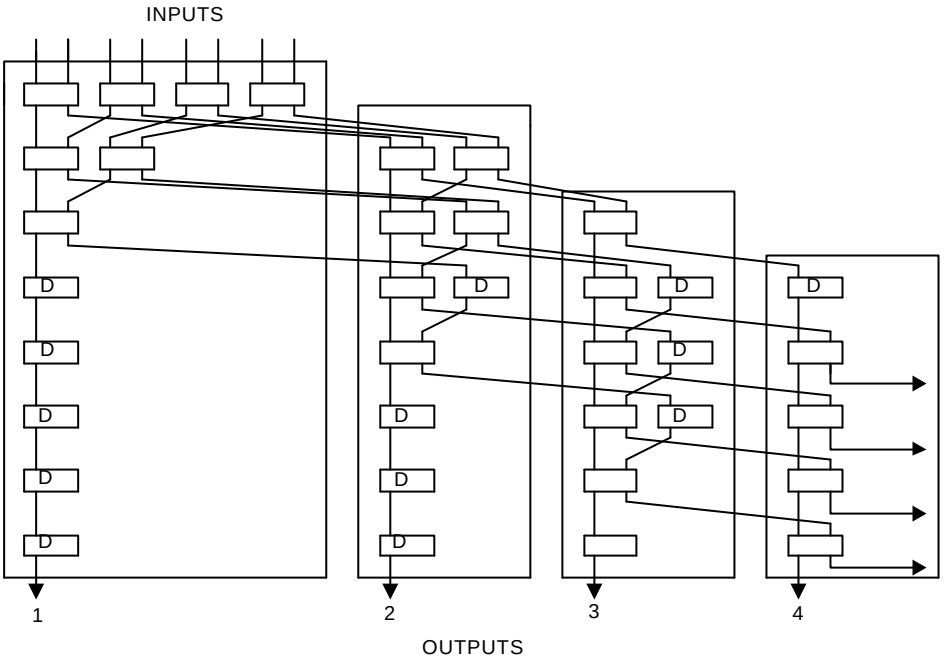


Figure Appendix A .8 Structure of Knockout concentrator

For each output port, N separate filters are used, one for each input bus, operating at the speed of a single line. In the Knockout Switch, all N packets arriving in a slot cannot be received by an output buffer, while only L may be received. The rationale behind the latter is that L need not be large to achieve low packet-loss rates. For example, under a uniform input pattern, a loss rate of 10^{-6} is achieved with L as small as 8, regardless of the load and switch size.

Among the above various classes of ATM switch, we choose a crossbar ATM switch and one N^2 disjoint path ATM switch (Knockout ATM switch) as our verification examples in this thesis. This is because we are only able to get the design examples on these two kinds of ATM switches. In addition, we apply model checking on a simple port controller and a commercial RCMP chip which is used for ATM ingress traffic management. These two designs can be adopted to be a component in almost any kinds of ATM switches.

Appendix B

Model Checking of Null Port Controller

B.1 Counter Reduction in Property 3

Here we introduce how to use Counter Reduction to verify Property 3. Basically, Counter Reduction must be jointly applied with Environment Modification or Internal Signal Usage method. We will introduce the verification of Property 3 using the combination of Counter Reduction and Environment Modification method first.

The following is the environment of the null port controller for property 3 jointly using Counter Reduction and Environment Modification method.

```
1.    typedef num {S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11,
      S12, S13, S14, S15} state;
2.    assign rx_ip_data_ran = $ND(0, 1, 2, …, 255);
3.    always @ (posedge clock) begin
4.    case (state)
5.          S15: state = S1;
6.          S1: state = S2;
7.          S2: state = S3;
8.          S3: state = S4;
9.          S4: state = S5;
10.         S5: state = S6;
11.         S6: state = S7;
12.         S7: state = S8;
13.         S8: state = S9;
14.         S9: state = S10;
```

……

```
15.         S14: state = S15;
16.    endcase;
17.    if (state== S1)
18.         framestart = 1;
19.    else
20.         framestart = 0;
21.    if (state == S2)
22.         rx_ip_soc = 1;
23.    else
24.         rx_ip_soc = 0;
25.         ip_empty = 1;
26.         rx_ip_rd_req = 1 ;
27.         ctr_sz = 0;
28.         ctr_id = 0;
29.         npc_rst_n = 1;
30.         rx_ip_data = rx_ip_data_ran;
31.    if (state = S4)  rx_ip_data_s4 = rx_ip_data_ran;
32.    else if (state=S5) rx_ip_data_s5 = rx_ip_data_ran;
33.    else if (state=S6) rx_ip_data_s6 = rx_ip_data_ran;
                              ……
34.   end
```

Figure Appendix B .9 Environment for property 3 using counter reduction and Environment Modification

The following lists all the CTL expression for property 3 using Environment Modification.

```
AG( npc_rst_n = 1)
(A.3.c.a.1)

AG(state= S1 -> ctr_id = 0 * ctr_sz = 0 * ip_empty = 1 *
rx_ip_rd_req = 1)
(A.3.c.a.2)

AG(state = S1  -> framestart = 1)                    (A.3.c.a.3)

AG(state = S2 + state = S3 + … + state = S15 -> framestart =
```

```
0);
(A.3.c.a.4)
AG(state = S2 -> rx_ip_soc = 1)
(A.3.c.a.5)
AG (state = S4 -> ip_mem_addr_r[8:1] == rx_ip_data_s3)

(A.3.c.a.6)
AG(state = S5 -> ip_mem_addr_r[8:1] == rx_ip_data_s3 *
ip_mem_addr_r[0] == rx_ip_data_s4[7]  * ip_mem_addr_c[8:6] ==
rx_ip_data_s5[6:4] * ip_mem_addr_c[5:0] = 6'b000100);

(A.3.c.a.7)
AG(state = S6 -> ip_mem_addr_r[8:1] == rx_ip_data_s3 *
ip_mem_addr_r[0] == rx_ip_data_s4[7]  * ip_mem_addr_c[8:6] ==
rx_ip_data_s5[6:4]  *  ip_mem_addr_c[5:0] = 6'b000101);
                                                    (A.3.c.a.8)
AG(state = S7 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0] == rx_ip_data_ran_s4[7]  *
ip_mem_addr_c[8:6] == rx_ip_data_ran_s4[6:4]
* ip_mem_addr_c[5:0] = 6'b000110);                (A.3.c.a.9)
AG(state = S8 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0] == rx_ip_data_ran_s4[7]  *
ip_mem_addr_c[8:6] == rx_ip_data_ran_s4[6:4] *
ip_mem_addr_c[5:0] = 6'b000111);
(A.3.c.a.10)
AG(state = S9 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
```

```
* ip_mem_addr_r[0]==rx_ip_data_ran_s4[7] * ip_mem_addr_c[8:6]
== rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b001000);
                                                    (A.3.c.a.11)
AG(state = S10 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0]==rx_ip_data_ran_s4[7] * ip_mem_addr_c[8:6]
== rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b001001);

(A.3.c.a.12)
AG(state = S11 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0]==rx_ip_data_ran_s4[7] *ip_mem_addr_c[8:6]
== rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b001010);

(A.3.c.a.13)
AG(state = S12 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0]==rx_ip_data_ran_s4[7] * ip_mem_addr_c[8:6]
== rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b001011);

(A.3.c.a.14)
AG(state = S13 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0]==rx_ip_data_ran_s4[7] *ip_mem_addr_c[8:6]
== rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b001100);

(A.3.c.a.15)
AG(state = S14 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0]==rx_ip_data_ran_s4[7] * ip_mem_addr_c[8:6]
== rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b001101);
```

(A.3.c.a.16)

```
AG(state = S15 -> ip_mem_addr_r[8:1] == rx_ip_data_ran_s3
* ip_mem_addr_r[0]==rx_ip_data_ran_s4[7] *ip_mem_addr_c[8:6]
== rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b000000);
```

(A.3.c.a.17)

```
AG(state = S1 + state = S2 + state = S3 -> ip_mem_addr_r[8:1]
== rx_ip_data_ran_s3 * ip_mem_addr_r[0] ==
rx_ip_data_ran_s4[7] * ip_mem_addr_c[8:6] ==
rx_ip_data_ran_s4[6:4] * ip_mem_addr_c[5:0] = 6'b000000);
```

(A.3.c.a.18) AG (state = S1 + state = S2 + state=S3 + state=S4 + state=S5 -> ip_mem_wr_en = 0)

(A.3.c.a.19)

```
AG (state=S6 + state=S7 + state=S8 + state=S9 + state=S10 +
state=S11 + state=S12 + state=S13 + state=S14 + state = S15 -
> ip_mem_wr_en = 1)                                (A.3.c.a.20)
```

```
AG (state = S6 -> ip_mem_data == rx_ip_data_s5)
```

(A.3.c.a.21)

```
AG (state = S7 -> ip_mem_data == rx_ip_data_s6)
```

(A.3.c.a.22)

```
AG (state = S8 -> ip_mem_data == rx_ip_data_s7)
```

(A.3.c.a.23)

```
AG (state = S9 -> ip_mem_data == rx_ip_data_s8)
```

(A.3.c.a.24)

```
AG (state = S10 -> ip_mem_data == rx_ip_data_s9)
(A.3.c.a.25)

AG (state = S11 -> ip_mem_data == rx_ip_data_s10)
(A.3.c.a.26)

AG (state = S12 -> ip_mem_data == rx_ip_data_s11)
(A.3.c.a.27)

AG (state = S13 -> ip_mem_data == rx_ip_data_s12)
(A.3.c.a.28)

AG (state = S14 -> ip_mem_data == rx_ip_data_s13)
(A.3.c.a.29)

AG (state = S15 -> ip_mem_data == rx_ip_data_s14)
(A.3.c.a.30)
```

(A.3.c.a.1) to (A.3.c.a.5) express the five assumptions. (A.3.c.a.6) to (A.3.c.a.18) describe the address initialization and incrementation. (A.3.c.a.19) to (A.3.c.a.30) represent that the data bytes can be transferred properly from transceiver board to the memory with a clock cycle delay. Since the meaning of each CTL expression is very similar to that in Section 4.4, we do not explain them in detail here.

Using Internal Signal Involved CTL, the environment looks like the following:

```
1.    typedef num {S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11,
      S12, S13, S14, S15} state;
2.    assign rx_ip_data_ran = $ND(0, 1, 2, …, 255);
3.    always @ (posedge clock) begin
4.    case (state)
5.          S15: state = S1;
6.          S1: state = S2;
7.          S2: state = S3;
8.          S3: state = S4;
9.          S4: state = S5;
10.         S5: state = S6;
11.         S6: state = S7;
```

```
12.         S7: state = S8;
                  …….
13.         S12: state = S13;
14.         S13: state = S14;
15.         S14: state = S15;
16.     endcase;
17.    if (state== S1)
18.         framestart = 1;
19.    else
20.         framestart = 0;
21.         rx_ip_soc = rx_ip_soc_ran;
22.         ip_empty = ip_empty_ran;
23.         rx_ip_rd_req =rx_ip_rd_req_ran;
24.         ctr_sz = ctr_sz_ran;
25.         ctr_id = ctr_id_ran;
26.         npc_rst_n = npc_rst_n_ran;
27.         rx_ip_data = rx_ip_data_ran;
28.    if (state = S4)  rx_ip_data_s4 = rx_ip_data_ran;
29.    else if (state=S5) rx_ip_data_s5 = rx_ip_data_ran;
30.    else if (state=S6) rx_ip_data_s6 = rx_ip_data_ran;
31.    always @(posedge clock) begin
32.          ip_mem_addr_c_r1[5 : 0] = ip_mem_addr_c[5:0];
33.          rx_ip_data_r1 = rx_ip_data;
34.     end
35.     always @(posedge clock) begin
36.          ip_mem_addr_c_plus1 = ip_mem_addr_c_r1[5:0] + 1;
37.     end
38.   end
```

Figure Appendix B .10 Environment for Property 3 using Counter Reduction and Internal Signal Usage

The following (A.3.c.b.1) to (A.3.c.b.11) is the CTL expression of Property 3 using Counter Reduction and Internal Signal Usage methods, and these are exactly the same as (4.3.b.1) to (4.3.b.11), respectively, except (A.3.c.b.5) and (A.3.c.b.7). The difference between (A.3.c.b.5) and (4.3.b.5) is ip_cell_cnt = 10 in (A.3.c.b.5) and ip_cell_cnt = 50 in (4.3.b.5). Likewise, the difference between (A.3.c.b.7) and (4.3.b.7) is ip_mem_addr_c[5:0] = 14 in (A.3.b.c.7) and ip_mem_addr_c[5:0] = 54 in (A.3.b.7).

AG (framestart = 1 -> ip_state_i = idle) (A.3.c.b.1)

AG(framestart = 1 * npc_rst_n = 1 * ip_state_i = idle * ip_empty = 1 * rx_ip_rd_req = 1 * ctr_id = 0 -> AX (ip_state_i = rx_wait))
(A.3.c.b.2)

AG(ip_state_i = rx_wait * rx_ip_soc = 1 -> AX(ip_state_i = rx_store1))
(A.3.c.b.3)

AG (ip_state_i = rx_store1 * ctr_sz = 0 -> AX(ip_state_i = rx_store_2 * ip_mem_addr_r[8:1] == rx_ip_data_s4))

(A.3.c.b.4)

AG(ip_state_i = rx_store2 * ctr_sz = 0 -> AX (ip_state_i = rx_data * ip_mem_addr_r[8:1]==rx_ip_data_ran_s4 * ip_mem_addr_r[0] == rx_ip_data_s5[7] * ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] * ip_mem_addr_c[5:0] = 6’b000100 * ip_cell_cnt = 10) (A.3.c.b.5)

AG(ip_state_i = rx_data -> AX (ip_mem_addr_r[8:1]== rx_ip_data_s4 * ip_mem_addr_r[0] == rx_ip_data_s5[7] * ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] * ip_mem_addr_c[5:0] = ip_mem_addr_c_plus1 * ip_cell_cnt == cell_cnt_minus1)
(4.3.c.b.6)

AG(ip_state_i = rx_data * ip_cell_cnt = 1 -> AX (ip_state_i = ip_idle * ip_mem_addr_r[8:1]==rx_ip_data_s4 * ip_mem_addr_r[0] == rx_ip_data_s5[7] * ip_mem_addr_c[8:6] == rx_ip_data_s5[6:4] * i_mem_addr_c[5:0]=14)
(A.3.c.b.7)

```
AG(ip_state_i = rx_data * ip_cell_cnt = 1 -> AXAX(ip_state_i
= ip_idle * ip_mem_addr_r[8:1]==rx_ip_data_s4 *
ip_mem_addr_r[0] == rx_ip_data_s5[7] * ip_mem_addr_c[8:6] ==
rx_ip_data_s5[6:4] * ip_mem_addr_c[5:0] = 0)(4.3.c.b.8)

AG(ip_state_i = rx_data -> AX(ip_mem_wr_en = 1))
(A.3.c.b.9)

AG(!(ip_state_i = rx_data) -> AX(ip_mem_wr_en = 0))


(A.3.c.b.10)

AG (ip_state_i = rx_data -> AX (ip_mem_data ==rx_ip_data_r1))


(A.3.c.b.11)
```

B.2 Model Checking of the null port controller

By the three methods that we introduce in Section 4.4, we are able to verify any properties in null port controller. In this section, we explain how to establish environments and CTL expressions to verify the properties listed in section 4.3.1. To simplify our model checking without losing the properties of the design, we choose Counter Reduction with Environment Modification. The environments and CTL expressions for other methods can be developed in a similar way. The detailed example has been illustrated in Section 4.4, for the space limitation, we will not explain them in detail here.

Property 1: The null port controller will be reset properly when either null port controller reset signal (npc_rst_n) or null port controller disable signal (ctr_id) asserts.

The environment of this property can use the environment in Figure Appendix B .9. The CTL expression is the following:

```
AG (npc_rst_n = 0 or ctr_id = 1 -> ip_mem_data = 0 *
```

```
ip_mem_wr_en = 0 * ip_mem_addr_c = 0 * ip_mem_addr_r = 0 *
ip_fab_data = 0 * op_fifo_data = 0 * op_fab_ack = 0)
                                                    (4.1.c.a.1)
```

Property 2: When input null port controller can accept a cell, transceiver board has a cell to send and null port controller is in debugging state (*ctr_sz* = 1), address will be set up and increment properly, and data will be transfer correctly.

Property 2 can use the environment of Property 3 (Figure Appendix B .9) except that we give ctr_sz as "1". The CTL expressions are very similar to (A.3.c.a.1) to (A.3.c.a.30) except that ip_mem_addr_r[8:0] = 0 and ip_mem_addr_c[8:6] = 0, so we do not list all the CTL expressions here.

Property 4: When input null port controller has a cell to send and it will transfer data and increment address properly. If it does not receive positive acknowledgment signal, it will stop sending data, otherwise, it will send data continually.

The environment for this property is as the following:

```
1.    typedef num {S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11,
      S12, S13, S14, S15} state;
2.    assign rx_ip_data_ran = $ND(0, 1, 2, …, 255);
3.    always @ (posedge clock) begin
4.    case (state)
5.          S15: state = S1;
6.          S1: state = S2;
7.          S2: state = S3;
8.          S3: state = S4;
9.          S4: state = S5;
10.         S5: state = S6;
                                        ……
11.         S10: state = S11;
12.         S11: state = S12;
13.         S12: state = S13;
14.         S13: state = S14;
```

```
15.         S14: state = S15;
16.     endcase;
17.     if (state== S1)
18.         framestart = 1;
19.    else
20.         framestart = 0;
21.         rx_ip_soc = rx_ip_soc_ran;
22.         ip_empty = 0;
23.         rx_ip_rd_req =rx_ip_rd_req_ran;
24.         ctr_sz = 0;
25.         ctr_id = 0;
26.         npc_rst_n = 1;
27.         rx_ip_data = rx_ip_data_ran;
28.         mem_ip_data = mem_ip_data_ran;
29.         fab_ip_ack = fab_ip_ack_ran;
30.    always @(posedge clock) begin
31.        if (state = S3) mem_ip_data_s3 = mem_ip_data_ran;
32.        else if (state = S4) mem_ip_data_s4 = mem_ip_data_ran;
33.           ……
34.    end
35.    always @(posedge clock) begin
36.        if (state == S1) begin
37.           ip_mem_addr_r_s1 = ip_mem_addr_r;
38.           ip_mem_addr_c_s1= ip_mem_addr_c;
39.        end
40.    end
41.    always @(posedge clock) begin
42.         ip_mem_addr_c_r1 = ip_mem_addr_c;
43.    end
44.    always @(posedge clock) begin
40.         ip_mem_addr_c_plus1 = ip_mem_addr_c_r1 + 1;
41.    end
42.    end
```

Figure Appendix B .11 Environment of null port controller for Property 4 using Counter Reduction and Environment Modification

In the environment, we define ip_empty = 0, ctr_sz=0, ctr_id=0, npc_rst_n = 1, so the input null port controller will transmit data bytes to the fabric. When the input null port

controller transmits a data cell to the fabric, in the state S3, the first byte memory address will be ready, and the in state S4, the first byte will be transferred to the fabric. Then after that, the data bytes are transferred every byte per clock cycle. In state S8, the fifth byte data will be transferred to the input port of the fabric. Since the first byte data is the fabric header, it will be stripped off in the fabric. The second byte (the output null port controller header) is in the input port of the fabric at S5, and will be transferred to the output of the fabric at S9 because of four clock cycle delay inside the fabric. The output null port controller detects the second data byte at S9, and give the acknowledge signal at S11. The acknowledgment signal will be passed to the input null port controller immediately. The input null port controller get the acknowledgment signal at S11. If the input null port controller get the positive acknowledgment signal, it will continually transfer data at S13, otherwise, the inputs of the fabric will be 0 at S13. To test this, we should have a 15 state environment (S1 to S15) and 10 bytes data to send.

First of all, we have to verify that the memory address are set up and incremented correctly. The specification of the memory address behavior includes the following four sub-properties:

sub-property 1. During state S1, S2 and S3, the memory address will not change;

sub-property 2. From state S4 to S12, the memory column address will increment by 1 per clock cycle;

sub-property 3. At S13, it will be pointed to the first data byte if the input null port controller received a negative acknowledgment signal, and it will continue to increment at S13 if it received a positive acknowledgment signal.

Sub-property 4. At S14, the memory address will point to the first data byte because the input port controller has finished transferred a complete cell.

Sub-property 1 can be expressed by (A.4.c.a.1)(A.4.c.a.2) and (A.4.c.a.3)

```
AG(state = S1 -> ip_mem_addr_r == ip_mem_addr_r_s1 *
ip_mem_addr_c == ip_mem_addr_c_s1)
```

(A.4.c.a.1)

```
 AG(state = S1  ->  AX(ip_mem_addr_r == ip_mem_addr_r_s1 *
ip_mem_addr_c = ip_mem_addr_c_s1))
```

(A.4.c.a.2)

```
AG(state = S1 -> AX AX (ip_mem_addr_r == ip_mem_addr_r_s1 *
ip_mem_addr_c = ip_mem_addr_c_s1))
```

(A.4.c.a.3)

Sub-property 2 can be expressed by

```
AG(state = S4 + state = S5 + state = S6 + state = S7 + state
= S8 + state = S9 + state = S10 + state = S11 + state = S12 -
> ip_mem_addr_r == ip_mem_addr_r_s1 * ip_mem_addr_c ==
ip_mem_addr_c_plus1)
```

(A.4.c.a.4)

Sub-property 3 can be expressed by (A.4.c.a.5) and (A.4.c.a.6).

```
AG(state = S11 * fab_ip_ack = 1  -> AX AX(ip_mem_addr_r ==
ip_mem_addr_r_s1* ip_mem_addr_c = ip_mem_addr_c_plus1))
```

(A.4.c.a.5)

```
AG(state = S11 * fab_ip_ack = 0  ->(ip_mem_addr_r ==
ip_mem_addr_r_s1* ip_mem_addr_c = ip_mem_addr_c_s1))
```

(A.4.c.a.6)

Sub-property 4 can be expressed by (A.4.c.a.7)

```
AG(state = S14  -> ip_mem_addr_r == ip_mem_addr_r_s1 *
ip_mem_addr_c = ip_mem_addr_c_s1)
```

(A.4.c.a.7)

Next, we verify the data bytes are transferred correctly. The specification of the data cell transfer can be described as the following four sub-properties:

sub-property 5. At state S1, S2, S3, *ip_fab_data* = 0;

sub-property 6. At state S4 to S12, *ip_fab_data* is transferred from the memory to the inputs of the fabric with one clock cycle delay.

sub-property 7. At S13, if the input null port controller receives a positive acknowledgment signal at state S11, it will continually transfer a data byte, otherwise, it will stop sending a data

sub-property 8. At state S14 and S15, the *ip_fab_data* is equal to zero which means that there is no data transfer occurred at these two states.

Sub-property 5 can be expressed by (A.4.c.a.8):

```
AG(state = S1 or state = S2 or state = S3 -> ip_fab_data = 0)
```

(A.4.c.a.8)

Sub-property 6 can be expressed by (A.4.c.a.9) to (A.4.c.a.17).

```
AG(state = S3 * mem_ip_data==mem_ip_data_s3 ->
AX(ip_fab_data== mem_ip_data_s3)
```

(A.4.c.a.9)

```
AG(state = S4 * mem_ip_data==mem_ip_data_s4 -> AX(ip_fab_data
== mem_ip_data_s4))
```

(A.4.c.a.10)

```
AG(state = S5 *mem_ip_data==mem_ip_data_s5 -> AX(ip_fab_data
== mem_ip_data_s5))
```

(A.4.c.a.11)

```
AG(state = S6 *mem_ip_data == mem_ip_data_s6 -> AX
```

```
(ip_fab_data==mem_ip_data_s6))
(A.4.c.a.12)

AG(state = S7 * mem_ip_data==mem_ip_data_s7 ->AX (ip_fab_data
== mem_ip_data_s7))
(A.4.c.a.13)

AG(state = S8 * mem_ip_data == mem_ip_data_s8 -> AX
(ip_fab_data == mem_ip_data_s8))
(A.4.c.a.14)

AG(state = S9 * mem_ip_data == mem_ip_data_s9 -> AX
(ip_fab_data == mem_ip_data_s9))
(A.4.c.a.15)

AG(state = S10 * mem_ip_data == mem_ip_data_s10 -> AX
(ip_fab_data == mem_ip_data_s10))
(A.4.c.a.16)

AG (state = S11 * mem_ip_data == mem_ip_data_s11 -> AX
(ip_fab_data == mem_ip_data_s11))
(A.4.c.a.17)
```

Another way to express Sub-property 6 is to set up a assistant variable mem_ip_data_r1 which is mem_ip_data with one clock cycle delay, and then sub-property 6 can be represented as the following:

```
AG (state = S4 + state = S5 + state = S6 + state = S7 + state
= S8 + state = S9 + state = S10 + state = S11 + state = S12 -
> ip_fab_data == mem_ip_data_r1)
```

Sub-property 7 can be expressed by (A.4.c.a.18) and (A.4.c.a.19).

```
AG(state = S11 * fab_ip_ack = 1  -> AX AX (ip_fab_data ==
mem_ip_data_s12))
(A.4.c.a.18)
```

```
AG(state = S11 * fab_ip_ack = 0 -> AX AX(ip_fab_data = 0))
```

(A.4.c.a.19)

Sub-property 8 can be expressed by (A.4.c.a.20)

```
AG(state = S14 ∨ state = S15 -> ip_fab_data = 0)
```

(A.4.c.a.20)

In addition, we have to verify that the memory read request signal perform properly. The expected behavior of *ip_mem_rd_req* signal are as the following :

sub-property 9. At state S1, S2, *ip_mem_rd_req* will be de-asserted.

sub-property 10. At state S3 to S12, *ip_mem_rd_req* will be asserted.

sub-property 11. If *fab_ip_ack* is asserted at S11, it will be asserted at state S13; otherwise, it will be deasserted.

sub-property 12. At state S14 and S15, *ip_mem_rd_req* will be deasserted.

(A.4.c.a.21) represents sub-property 9, and (A.4.c.a.22) expresses sub-property 10. Sub-property 11 is represented by (A.4.c.a.23) and (A.4.c.a.24), and sub-property 12 is expressed by (A.4.c.a.25).

```
AG(state = S1 + state=S2 -> ip_mem_rd_req = 0)   (A.4.c.a.21)
AG(state = S3 + state = S4 + state = S5 + state = S6 + state
= S7 + state = S8 + state = S9 + state = S10 + state = S11 +
state = S12 -> ip_mem_rd_req = 1)                (A.4.c.a.22)
AG(state = S11 * fab_ip_ack = 1 -> AXAX (ip_mem_rd_req = 1))
                                                 (A.4.c.a.23)
AG(state = S11 * fab_ip_ack = 0 -> AXAX (ip_mem_rd_req = 0)
                                                 (A.4.c.a.24)
AG(state = S14 or state = S15 -> ip_mem_rd_req = 0)
```

(A.4.c.a.25)

Property 5: An ATM cell can be transferred from transceiver board to the fabric coherently.

Property 5 can be deduced from property 3 and property 4. In property 3, we have proved that the memory address will point at the first data byte after data bytes transfer (i.e. (A.3.c.a.17)), and property 4 did not change the memory address the first several states (i.e. (A.4.c.a.1)(A.4.c.a.2) and (A.4.c.a.3)) which means that the memory address still points at the first data byte.

Property 6: Memory cannot be read or written at the same time.

This property are checked by using the environments of both Property 3 and Property 4. The environment of Property 3 (Figure Appendix B .9) expresses that the input null port controller receives the data bytes the transceiver board, so the memory read request signal (*ip_mem_rd_req*) will be always de-asserted. On the other hand, the environment of Property 4 (Figure Appendix B .11) express that the input null port controller transmits data bytes from the memory to the fabric, so the memory write enable signal will be always de-asserted. CTL expression can be written by the following, but they are executed in different environments.

```
AG (ip_mem_wr_en = 0)                    (A.6.c.a.1)
AG(ip_mem_rd_req = 0)                    (A.6.c.a.2)
```

Property 7: Output null port controller will send an acknowledgment signal and transfer the cell to the FIFO properly after it detects that a cell is coming.

The property can be checked using the same environment as that of property 4 (Figure Appendix B .11). As the analysis in property 4, the second data byte transmitted by the input null port controller at state S5 will reach the output null port controller at state S9, and the output null port controller will detect the port controller routing byte at S9 and give

an acknowledgment signal at state S10. Whether the output null port controller give a positive acknowledgment or a negative one depends on the last significant bit of the first byte the output null port controller receive. If the least significant bit is “1”, the acknowledgment signal is positive; otherwise, it is negative. So we could use the environment of property 4, and the CTL expression can be written as the following:

```
AG (fab_op_data[0] = 1 * state = S9 -> AX (op_fab_ack = 1))
```

(A.7.c.a.1)

```
AG(fab_op_data[0] = 0 * state = S9 -> AX(op_fab_ack = 0))
```

(A.7.c.a.2)

So far, we have proved the seven properties of the null port controller, and these properties examine the main feature of the null port controller. The experimental results are reported in the next section.