Modeling and Verification of the Fairisle ATM Null Port Controller in VIS

Jianping Lu and Sofiène Tahar

Electrical & Computer Engineering Department, Concordia University
Montreal, Quebec, H3G 1M8 Canada
{jianping, tahar}@ece.concordia.ca

Technical Report

March 2003

Abstract. *In this report, we present the practical formal verification of Fairisle ATM (Asynchronous Transfer Mode) switch port controller using model checking. The ATM port controller is part of the Cambridge Fairisle ATM network and plays a key role in the ATM switching process. In particular, we present our experience on the model checking of the ATM port controller using the VIS tool from UC Berkeley. To this end, we successfully modeled the port controller behavior and structure in Verilog HDL, established the necessary verification environments and verified a number of relevant temporal properties on the port controller.*

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 design bugs, 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 [3].

One very successful formal verification approach is model checking [3] which enables to check a design model against temporal logic properties. 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. However, the specifications are not always easy to be expressed in the given temporal logic. In this paper, we present our experience in the formal specification using temporal logic and verification using model checking of the Fairisle [4] ATM (Asynchronous Transfer Mode [2]) switch port controller using the VIS (Verification Interacting with Synthesis) [1] tool from UC Berkeley. The Fairisle port controller (Figure 1) is a real design

from Cambridge University. It is at the heart of Fairisle ATM network switch [4]. In the ingress [2], the port controller receives ATM cells from the transmission board and performs the ATM switching on the received cells. It also sends the ATM cells to the switch fabric [5]. In the egress [2], the port controller receives ATM cells from the fabric and sends the acknowledgment signals to the switch fabric. The port controller assigns priorities to ATM cells, by preloading priority bits into the memory. The priority bit will be used for arbitration in the switch fabric.

In this work, we modeled the port controller at the RTL (Register Transfer Level) by following some documentation and incomplete structural code we have obtained from Cambridge. The RTL description of the port controller is written in Verilog HDL (Hardware Description Language). To verify the port controller in VIS, we established a proper environment, and defined a number of related CTL (Computation Tree Logic [3]) properties.

In following sections, we will introduce the behavior and structure of the port controller in Section 2. Section 3 describes the properties we established on the port controller. Section 4 illustrates a practical method on verifying CTL properties using model checking, and Section 5 summarizes the paper.

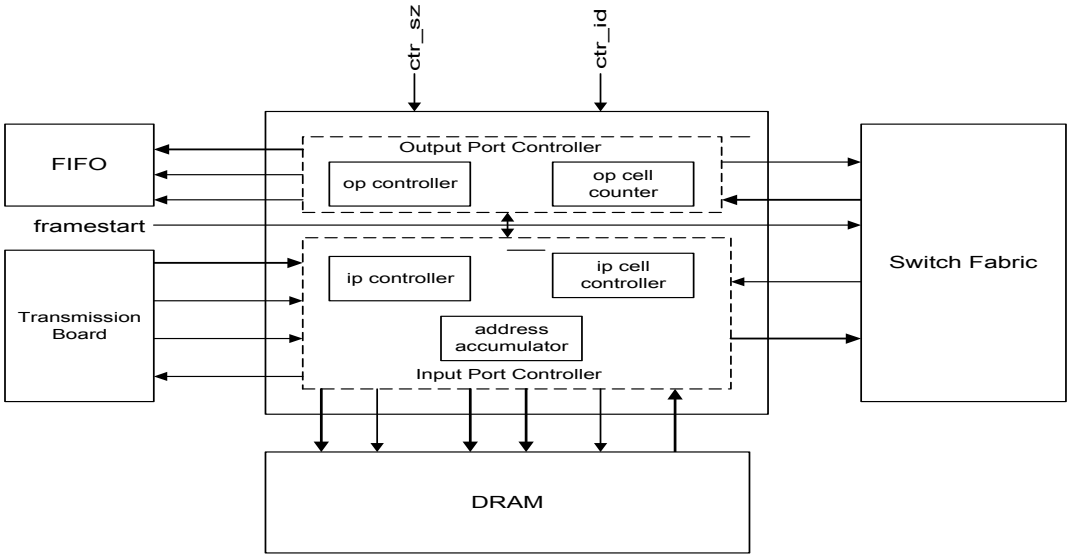

Figure 1. The Fairisle null port controller.

2. The Fairisle Null Port Controller

The Fairisle ATM switch [4] is a real ATM switch which was developed and used by Cambridge University. The Fairisle ATM switch consists mainly of port controllers and a switching fabric. The null port controller is a part of Fairisle ATM switch. Since it does only VCI (Virtual Channel Identifier) mapping and FIFO (First In First Out) 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. It also uses a FIFO to do speed matching with the transmission board. As shown in Figure 1, the null port controller is connected to the Fairisle ATM switch fabric, transmits ATM cells to the fabric and receives acknowledgment signals from it. Both the null port controller and the switch fabric use the same *framestart* signal (Figure 1) to synchronize the overall behavior.

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. With a clock frequency of 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 2 shows the format of an ATM cell. Received cells have 52 bytes: 48 data bytes, 2 VCI (Virtual Channel Identifier) bytes and 2 FAS (Frame Assignment Sequence) bytes. Transmitted cells have 54 bytes: 48 data bytes, 1 Fabric Routing Byte (FRB), 1 Port controller Routing Byte (PRB), 2 VCI bytes and 2 FA 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

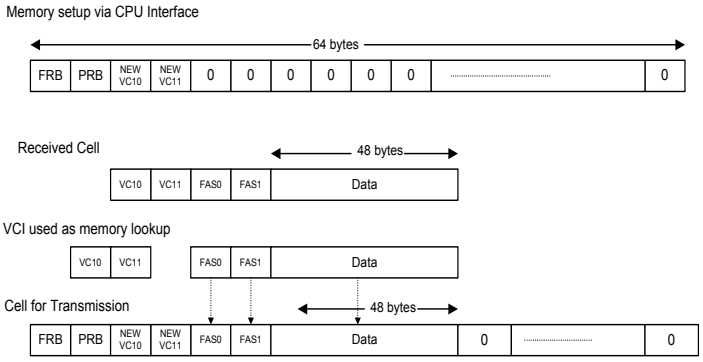

Figure 2. Format of received and transmitted cells.

For this project, we obtained 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 at the RTL (Register Transfer Level) according to its documents. The RTL description of the null port controller is written in Verilog HDL (Hardware Description Language). 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 currently SRAM are being used in ATM hardware designs. The only difference for using SRAM instead of DRAM was the memory interface, so it did not affect the internal logic of the null port controller which was our main focus of the verification. In following subsections, we will describe the behavior and structure of the port controller in more detail.

2.1. Behavior of the Fairisle Port Controller

The Fairisle null port controller consists of input port controller and output port controller. The input port controller receives ATM cells from the transmission board, and writes them into the memory at an address based on the value of the VCI [2]. In addition, the input port controller reads ATM cells out of the memory and transmits them into the switch fabric. Once it receives positive acknowledgement signals, the input port controller will continue transmitting data; otherwise, it will stop sending data. The output port controller receives data cells from the fabric, and

sends acknowledgment signals 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 port controller always monitors the *framestart* signal (Figure 1). On an active *framestart* signal, the input port controller will assert a write enable signal to the memory. After the *framestart* signal is received, 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 bytes transferred into the memory only after the input port controller receives the start of cell (SOC) signal. Once the SOC signal is received, the input port controller will latch the first two bytes which build the VCI field of the receiving cell. Table 1 is the conversion between the VCI of the receiving cell and its memory location, where *c* means the column address and *r* means the row address. The decimal digit indicates the position in the binary address (e.g., *r4* means bit 4 of the row address). The whole VCI 0 byte and bits 4 to 7 of VCI 1 byte will become the row address and the most 3 significant bits of the column address, and bits 3 to 0 of VCI 1 byte are unused in the conversion. On the other hand, when the *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 number of clock cycles, if the input port controller receives the positive acknowledgment signal through the switch fabric, it will continue sending the ATM cell; otherwise, it will stop the transmission.

Table 1. VCI to memory location 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

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 the switch fabric. After the *framestart* signal is asserted, the output port controller will detect the active bit in the 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 the Port controller Routing Byte (PRB), will be stripped.

The state transition of the input port controller with 8 states (*ip_idle*, *rx_wait*, *rx_store1*, *rx_store2*, *rx_data*, *tx_addr*, *tx_first_5* and *tx_data*) is shown in Figure 3. 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.

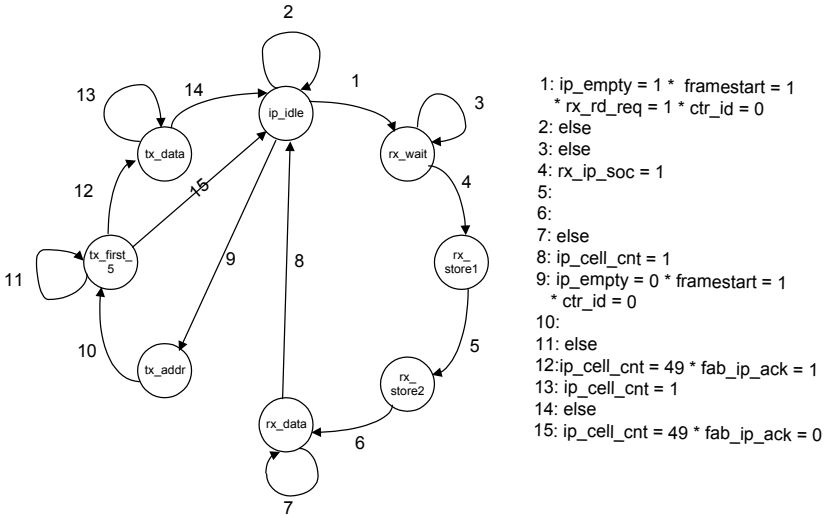

Figure 3. State transition diagram of the input port controller.

2.2. Structure of the Fairisle Port Controller

Figure 4 shows the structure of the 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* and *address accumulator* are up counters that increment by 1 per data byte transfer. In Figure 4, 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 to the cell memory and the data inputs from the cell memory, respectively. Both 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 signals, respectively. The memory row and column addresses are provided by *ip_mem_addr_r* and *ip_mem_addr_c*, respectively. The *rx_ip_data* is an 8-bit data bus which is the data inputs from the transmission board. The signals *rx_rd_req* and *rx_ip_soc* indicate cell availability in the transmission board and the start of a cell, respectively. The *rx_ip_soc* signal corresponds to the framestart mentioned above. The signal *ip_rx_wr_en* demonstrates whether the input port controller is able to accept a cell or not. The *ip_fab_data* is an 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 succeeded the transfer to the destined output port controller. The *fab_op_data* is an 8-bit data input from the fabric to the port controller and *op_fab_ack* is the acknowledgment signal generated by this latter.

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 4, *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 an 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 datapath 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. The *npc_rst_n* is the reset signal.

Inside the port controller, there are two control registers (*ctr_id* and *ctr_sz*) and one status register (*ip_empty*). The *ctr_id* disables the inputs when it is asserted. 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 purposes. 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.

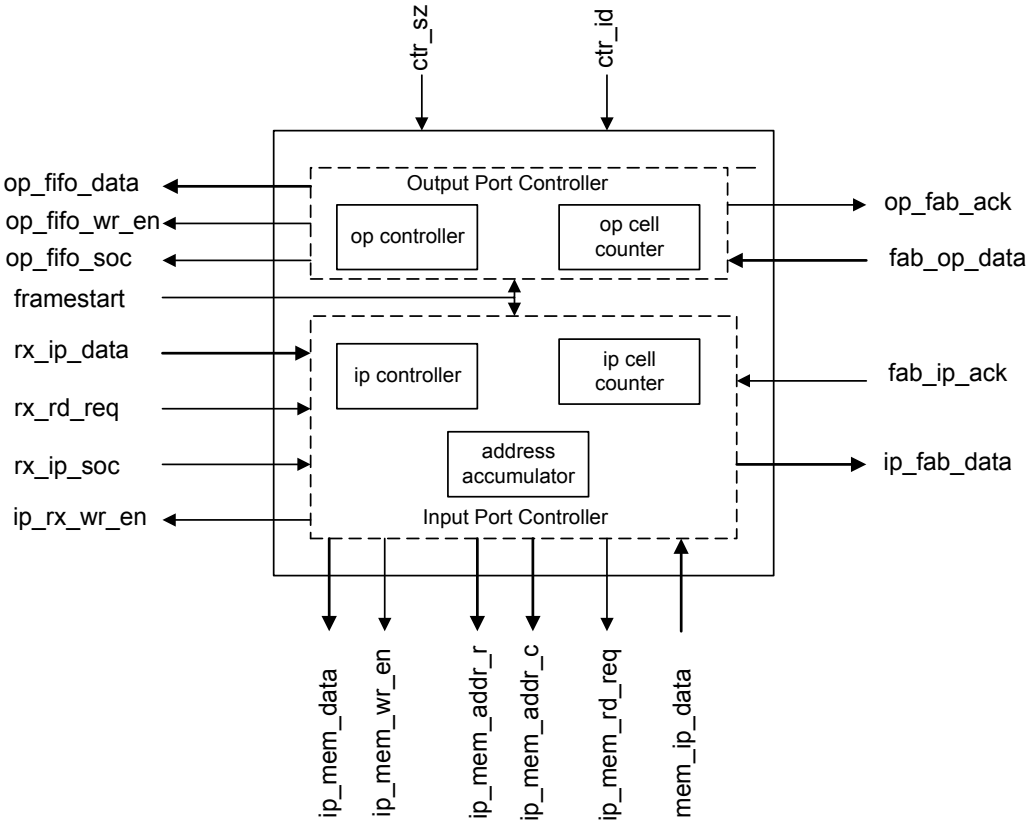

Figure 4. Structure of the null port controller.

3. Properties of the Null Port Controller

Before performing model checking, we must figure out the necessary properties of the null port controller. This is similar to creating some scenarios before any simulation. The Fairisle 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. Accordingly, we defined the following six major properties.

Property 1: The port controller will be reset properly when either the reset signal (*npc_rst_n*) is zero 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*), then the cell will be transferred to the input port controller and stored in the memory at the right location.

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*), then the cell will be transferred to the input port controller and stored in the memory at the right location.

Property 4: When the input port controller has a cell to send, it will send the cell to the fabric. If the input port controller does not receive a positive acknowledgment signal, it will stop sending the cell; otherwise, it will send the data cell completely.

Property 5: The memory cannot be read and write at the same time.

Property 6: The output port controller will send an acknowledgment signal after it detects an incoming cell.

Each of the above properties will be described formally in CTL. In the next section, we will report in detail the formal specification and verification of one sample property, *Property 3*. The specification of the other properties can be found in [6].

4. Formal Specification and Verification Approach

In this section we will demonstrate by example (using *Property 3*) how the specification and verification is processed in a practical way. *Property 3* states that

"*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*".

This property has the following five assumptions:

1. The input port controller can accept a cell, expressed as "*ip_empty* = 1";
2. The transmission board has a cell to send, expressed as "*rx_ip_rd_req* = 1";
3. The port controller is in normal operation state, expressed as "*ctr_sz* = 0" and "*ctr_id* = 0";
4. The port controller receives the *framestart* signal, expressed as "*rx_ip_soc* = 1";
5. The input port controller is not in reset state, and it can be expressed as "*npc_rst_n* = 1".

The input port controller first detects the signals *ip_empty*, *rx_ip_rd_req*, *ctr_sz, npc_rst_n,* and *ctr_id*. If these signals are satisfied with the above assumptions, the null port controller will start

monitoring *rx_ip_soc* in the following clock cycles. If the *rx_ip_soc* is asserted as well, the cell is transferred to the port controller. This behavior can be expressed formally in CTL as the follows[1]

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

The above CTL expression is not fully correct because the assumptions (*ip_empty*=1 and *rx_ip_rd_req*=1) do not happen at the same state as the fourth assumption (*rx_ip_soc*=1). Therefore, we need to put the assumptions into the environment. In fact, because the port controller has a cyclic period synchronized by the *rx_ip_soc* signal whose period is 64 clock cycles, we could establish an environment state machine with 64 states (Figure 5).

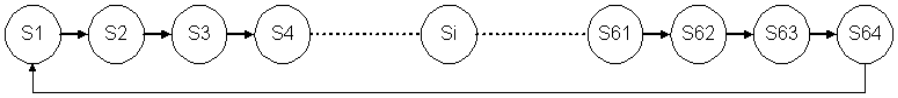

Figure 5. Environment state machine for the port controller.

4.1. Property Environment

In Figure 5, 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 denote 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 where the input port controller transfers the rest of the 48 bytes, then go into the "idle" state from S58 to S64; Otherwise, the input port controller will stop sending data, and S10 to S64 will be in "idle" state. In the meantime, 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 the *rx_ip_soc* signal. Once this latter is asserted, the next state will be the one where the transmission board transfers the first data byte (VCI) to the input port controller. For instance, assuming *rx_ip_soc* is asserted in S3, then *ip_rx_wr_en* will be asserted in 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.

1. The symbols "*", "+", "->" represent logical "and", "or" and "implication", respectively, and the symbols "AG" and "AX" are temporal operators meaning for all paths in all states and for all paths in next state, respectively.

The Verilog code for the above 64-state environment of the null port controller for *Property 3* is shown in Figure 6. In this environment, the correct assumption *npc_rst_n*=1, *ctl_id=0, ctr_sz*=0*, ip_empty*=1, *rx_ip_rd_req*=1, and *rx_ip_soc*=1 are set properly.

```
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.          ……
10.         Si: state = Si+1;
11.         …
12.         S63: state = S64;
13.    endcase;
14.     if (state== S1)
15.         framestart = 1;
16.     else
17.         framestart = 0;
18.     if (state == S3)
19.         rx_ip_soc = 1;
20.     else
21.         rx_ip_soc = 0;
22.       ip_empty = 1;
23.       rx_ip_rd_req=1 ;
24.       ctr_sz = 0;
25.       ctr_id = 0;
26.       npc_rst_n = 1;
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.    end
```

Figure 6. Verilog code of the environment of the port controller for Property 3.

In Figure 6, 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 in 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* in state S4 as *rx_ip_data_s4*; Likewise, line 27 to 28 store the values of *rx_ip_data* in state S5 and S6 as *rx_ip_data_s5* and *rx_ip_data_r6*, respectively. Similarly, we could store the value of *rx_ip_data* in any states. The stored *rx_ip_data* values will be applied to verify if the data are transferred coherently.

4.2. Property Assumptions and Conclusions

Using the above environment, *Property 3* is now more accurately expressible. We divide the verification into the two steps. The first step is to verify whether the environment represents the assumption, and the second step is to check if the conclusion is valid.

Step 1. Verify the assumptions

The above five assumptions are expressed using the following CTL expressions:

```
AG(npc_rst_n=1 * ctr_id=0 * ctr_sz=0)                                    (1)
```

```
AG(state=S1 -> ip_empty =1 * rx_ip_rd_req=1)                             (2)
```

```
AG(state=S3 -> rx_ip_soc=1)                                              (3)
```

Formulae (1) and (2) express that the null port controller is not in program or debugging mode and is going to receive an ATM cell. It also ensures that the transmission board has a cell available to send. In formula (3), we define "*rx_ip_soc=1*" at state S3, but the actual "*rx_ip_soc*" signal can be asserted 1 to 11 clock cycles after S1. Because such assumption does not affect the behavior of the null port controller, the assumption is valid. Finally, 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 (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 become the memory address and the memory address is incremented by 1 per byte data transfer. And the other is to verify that the data is transferred from transmission board to the memory with one clock cycle delay and the memory write enable signal is asserted during the data transfer. The formulae (4) and (5) below specify 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)
```

```
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)                                          (5)
```

The correct memory addresses increment can be specified by formulae (6), (7), (8) and (9) below. Formulae (6) to (8) 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. The CTL properties for the address increment between S8 and S56 are not listed in here, but they are very similar to (7) and (8), except that we have to give the correspondent values for *ip_mem_addr_c[5:0*]. Formula (9) 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)                                          (6)
```

```
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)                                          (7)
```

```
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)                                          (8)
```

```
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)                                       (9)
```

Next, we verify that the data is 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 signal (ip_mem_data). The ip_mem_wr_en signal, which should be asserted during the data transfer period (S7 to S56), is expressed by formulae (10) and (11) below. Also during the data transfer period, the ip_mem_data should equal the value of rx_ip_data with one clock cycle delay. The first and last byte data transfers are represented by formulae (12) and (13) below. The CTL properties for the rest of data transfer are not enumerated in here, but they are very similar to (12) and (13).

```
AG(state=S1 + state=S2 + … + state=S6 + state=S57 + … + state=S64 -> ip_mem_wr_en=0)(10)
```

```
AG(state=S7 + state=S8 + … + state=S56 -> ip_mem_wr_en=1)                (11)
```

```
AG(state=S7 -> ip_mem_data==rx_ip_data_s6)                              (12)
```

```
AG(state=S56 -> ip_mem_data==rx_ip_data_s55)                            (13)
```

Through the combination of the established environment with the null port controller model, the assumptions and conclusion of *Property 3* are successfully verified through model checking in VIS. We hence conclude formally that the assumptions imply the conclusion[2]. Following the above method, all other five properties of the port controller have been similarly specified and verified. However, we had to do minor modification to the environment when verifying each property. Besides, using this method, a lot of CTL formulae needed to be specified for every property because we have to verify the behavior at each environment state.

4.3. Property Division

To improve/ease the formal verification (model checking) of the above CTL expressions, we propose to use the idea of property division [6], which is based on using internal signals of the design to derive sub-properties (expressions) that can be combined to form back the original property (expression). At first, we have to establish an environment which is similar to Figure 6, 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 non-deterministic variables [1]. Figure 7 is the proposed modified environment of the null port controller for *Property 3*.

2. The proof is fairly simple, a proposition A is true and a proposition B is true, then the proposition $A \rightarrow B$ is true.

```
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 7. Verilog code of the modified environment for property division

Since we can only 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 7, 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 formulae to follow. Similar to Section 4.2., we first verify the proper address transfer and increment, and then verify the correct data transfer. To verify the proper address transfer and increment, we need to prove the following three consecutive sub-properties:

Sub-property 3a: The null port controller uses the two bytes VCI as the initial memory address two clock cycle after *rx_ip_soc* asserts;

Sub-property 3b: After *Sub-property 3a*, the memory address is incremented by 1 per clock cycle until the 50 bytes data have been completely transferred;

Sub-property 3c: After *Sub-property 3b*, the memory address will point to the first byte of the ATM cell.

The following formulae (14) to (21) are the CTL expressions of the above three sub-properties:

```
AG (framestart = 1 -> ip_state_i = idle)                                    (14)

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))                 (15)

AG (ip_state_i = rx_wait * rx_ip_soc = 1  -> AX(ip_state_i = rx_store1))    (16)

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))                                (   (17)

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 )                         (18)

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))                                            (19)

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)                                                    (20)

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)                                                     (21)
```

We use the internal signal *ip_state_i* to help us express sub-properties in CTL. *ip_state_i* is a state variable which has. The state transition of the input port controller with 8 states (*ip_idle*, *rx_wait*, *rx_store1*, *rx_store2*, *rx_data*, *tx_addr*, *tx_first_5* and *tx_data*) is shown in Figure 3. Accordingly, *Sub-property 3a* can be deduced by (14) to (18), where the deduction is based on property division [6]. In fact, *Sub-property 3a* is obtained through the combination of the following formulae (22) and (23), where (22) is deduced from (14) and (15), and (23) from (16), (17) and (18), respectively.

```
AG(framestart = 1 * npc_rst_n = 1 * ip_empty = 1 * rx_ip_rd_req = 1 *
ctr_id = 0 -> AX (ip_state_i = rx_wait))                                    (22)

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                                                                          (23)
```

In formula (19), we use two assistant signals: *ip_mem_addr_c_plus1* and *cell_cnt_minus1*. As shown in Figure 7, *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-property 3b can be deduced from expressions (18), (19) and (20). (18) 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; (19) means that *ip_cell_cnt* decrements by 1 and *ip_mem_addr_c* increments by 1 per clock cycle during *rx_data* state; (20) 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 also based on property division [6]. Formula (19) represents the general behavior of *ip_mem_addr_c*, *ip_mem_addr_r* and *ip_cell_cnt*, while (18) and (20) 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 (24), (25) and (26), *T1*, *T2*, *T3* expresses three different environment states, *addr* is a variable, and *addr_plus1* is the variable which is always greater than *addr* by 1 at the previous clock cycle. Formula (24) and (26) have a similar formats. Formula (25) is a general expression, however. Hence, we could convert (25) into (27) which includes 49 CTL expressions which have the same style as formula (24) or (26). By (24), (25) and (27), 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 inference rule is applied.

```
AG (state = T1 -> AX (state = T2 * addr = 4)                          (24)

AG (state = T2 -> AX (addr == addr_plus1)                             (25)

AG (state = T2 * addr = 53 -> AX (state = T3 * addr = 54))            (26)

AG (state = T2 * addr = 4 -> AX (addr = 5))
AG (state = T2 * addr = 5 -> AX (addr = 6))
            ……
AG (state = T2 * addr = 52 -> AX (addr = 53))                         (27)
```

Obviously, the formulae (18), (19) and (20) are very similar to (24), (25) and (26), respectively. We hence use a similar conversion to infer *Sub-property 3b*.

Finally, expression (21) indicates that after 50 bytes data are transferred, the memory address will point to the first byte of an ATM cell with a new VCI and header. This is actually *Sub-property 3c*.

Using the above formulae, we realize 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 i*p_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. (28), (29) and (30) express this property. In (30), *rx_ip_data_r1*, which is also an assistant signal, denotes the value of *rx_ip_data* with one clock cycle delay.

```
AG (ip_state_i = rx_data -> AX(ip_mem_wr_en = 1))                     (28)

AG (!(ip_state_i = rx_data)-> AX(ip_mem_wr_en = 0))                   (29)

AG (ip_state_i = rx_data -> AX (ip_mem_data ==rx_ip_data_r1))         (30)
```

4.4. Datapath 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. 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).

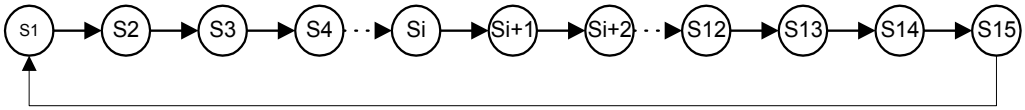

Figure 8. Modified environment of null port controller for datapath reduction

Figure 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 of the methods described in the previous sections. We have used the combination of all approaches to maximize performance. The experiment results are present in the next subsection.

4.5. Experimental Results

Experimental results on the model checking of all 6 properties are shown in Table 2, including CPU time, memory usage and number of BDD nodes generated. The experiments were performed using VIS-3.1 on a Sun Ultra Sparc (300MHz/500 MB) machine and using the verification approaches described above for *Property 3*. The environments and CTL formulae of all properties are described in [6]. Generally, in terms of machine time, model checking on the null port controller gives acceptable verification performance compared to simulation. In fact, we did run simulation on the null port controller before the model checking, where we have been able to detect a number of syntax errors, mistaken variable names and wrong counter numbers. By model checking, we detected some logic errors such as the memory write and read incoherence, and a misbehavior of the signals *ctr_sz* and *ctr_id*. The errors were traced by the counterexamples generated by VIS.

Table 2. Model checking experimental results

Properties	CPU time (sec.)	Memory (MB)	BDD Nodes (K)
Property 1	52	92	203,493
Property 2	256	198	284,563
Property 3	209	156	293, 354
Property 4	378	201	304,731
Property 5	34	77	153,980
Property 6	76	89	197,091

5. Conclusions

In this report, we have presented the modeling and formal verification by model checking of an ATM switch port controller. This is a real design of a telecommunications component used in the Cambridge Fairisle ATM network. While some specification properties cannot be concisely expressed using single temporal logic formulae, we have shown how we make use of an environment state machine to enable a proper specification. To enable the model checking process, properties are further subdivided into a set of assumption and conclusion sub-formulae which are combined by conjunction. Using such an approach, we succeeded the model checking of all specification properties of the port controller within the reasonable time. The method presented could be enough in order to verify larger designs. To this end, we may have to apply some more advanced techniques, such as symmetry reduction or compositional verification [5].

References

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

[2] H.D. Ginsburg, "ATM Solutions for Enterprise Internetworking", Addison-Wesley, 1996.

[3] C. Kern and M. Greenstreet, "Formal Verification in Hardware Design: A Survey", ACM Trans. on Design Automation of Electronic Systems, Vol. 4, April 1999, pp. 123-193.

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

[5] J. Lu and S. Tahar, "Practical Approaches to the Automatic Verification of an ATM Switch Fabric using VIS", Proc. IEEE 8th Great Lakes Symposium on VLSI, Lafayette, Louisiana, USA, Feb. 1998, pp. 368-373.

[6] J. Lu, "On the formal Verification of ATM Switches", M.A.Sc. Thesis, Department of Electrical and Computer Engineering, Concordia University, Canada, May 1999.