CHAPTER 5: IMPLEMENTATION AND TESTING

Chapter 5 discusses the implementation and testing phases that need to be done for the simulator. During the implementation phase, all the classes with important attributes will be shown together with the explanation of these attributes as well as methods contained within the classes.

Simulator testing is done in three parts. Component testing will test the resource classes and module testing focuses on cell switching testing and cell rate testing. Finally, the test driver is used to test the entire system.

5.1 Implementation

This ATM simulator consists of two main packages: *swictchpackage* and *simulatorglobal*. Followings will describe attributes within classes for these two packages.

5.1.1 Package: simulatorglobal

simulatorglobal consists of necessary classes for the general ATM network. Certain important classes which is useful for this simulator are *Queue.java, QueueNode.java, Indicator.java and GlobalClock.java*

QueueNode.java and Queue.java

```
public class QueueNode {
    private Object data;                    // data
    private QueueNode next;                 // pointer to next object
}

public class Queue {
    private QueueNode firstQueueNode;       // first node in  the linked list
    private QueueNode lastQueueNode;        // last node in the linked list
    private int numberOfNodes;              // number of nodes
}
```

QueueNode and *Queue* work together to provide the abstract data type of linked list. Interface methods involved in *Queue* are inserting and removing *QueueNode*, also from any position in *Queue*, retrieving and updating *QueueNode's* information in any position in *Queue*. The other two methods are getting *Queue* size and checking if it is an empty *Queue*.

Indicator.java

```
public class Indicator {
    protected int numObjects=0;
    protected int numObjectsReady=0;
}
```

The indicator class maintains two attributes. *NumObject* indicates number of objects registered into it and *numObjectsReady* indicates number of objects that have signalled to it. Therefore, Initial values for both attributes are 0.

The interface methods of this class allow for registration, signalling of objects and resetting of the signals for all objects.

GlobalClock.java

```
public class GlobalClock {
    //time atrributes
    protected long tick;
    protected static float usecsPerTick = 0.01f;
    protected long maxTicks = Long.MAX_VALUE;

    //control attributes
    protected boolean clockStop = true;
    protected boolean tickHappens = false;

    //object attributes
    protected CheckList signalledThisTick = new CheckList();
}
```

GlobalClock is used to synchronise the activities of all classes across the simulator. The attribute *tick* is used as the time unit in *GlobalClock*. This *tick* value is related to the real world time microsecond by *usecsPerTick:* how many microseconds for running one tick. The default value is 0.01 microseconds per tick. The control attributes determine the clock to proceed to next tick time or stop moving. Finally the object checklist maintains a list of objects bound to the *GlobalClock* and therefore dependant upon it for synchronising.

Interface methods provided in the clock including registration of objects with the clock, clock signalling for moving to the next tick, clock polling for the permission to move to the next tick, and clock querying for the conversion of tick so real time measure and vice versa.

AtmCell.java

```
public class AtmCell {
    protected int VPI;                  // virtual path indicator
    protected int VCI;                  // virtual channel indicator
    protected int CLP;                  // RESERVED
    protected int PT;                   // RESERVED

    //following attributes for RM cells
    protected float MCR;                // minimum cell rate, set by user
    protected float ACR;                // RESERVED
    protected float ER=0;               // RESERVED
    protected boolean CI=false;         // RESERVED
    protected boolean NI=false;         // RESERVED
}
```

AtmCell is the fundamental object in ATM switching. *VPI* and *VCI* values are used to provide switching information and *MCR* is used for controlling cell transfer rate. Others are reserved.

Interface methods allow the retrieving and updating for the values of *VPI, VCI*, and *MCR*.

5.1.2 Package: switchpackage

switchpackage consists of the necessary classes for ATM switch, i.e *InjectedHeader.java*, *Buffer.java* *Switch.java*, *BanyanSwitch.java*, and *RoutingTable.java*,.

InjectedHeader.java

```
public class InjectedHeader {
    private boolean activity;   // TRUE if the corresponding buffer contains
                                // cell
    private int destination;    // output port destination
    private int vpi, vci;       // both values identify an ATM application
}
```

InjectedHeader is the object for injected header cell. This object copies some data from *buffer* for switching purpose. *activity* is a boolean type which return true when there is an ATM cell inside the buffer. *destination* refers to the output buffer for which the cell should be switched to. In case that *activity* is false, *destination* is set to –1. Lastly, both *vpi* and *vci* work together to identify an ATM application.

Interface methods allow the retrieving of destination, activity, vpi, and vci. The other two methods insert all these information into injected header as well as clear the injected header cell.

Buffer.java

```
public class Buffer {
    protected int max_buff_size;       // user input maximum buffer size in cells
    protected int curr_buff_size;      // buffer size occupied so far in cells
    protected int total_cells_lost;    // total cells discard in this buffer in cells
    protected Queue bufferQueue;       // cells contain in this buffer
    protected Link connectedLink;      // a link at the side of buffer
}
```

Buffer contains attributes for the buffer size, current buffer size occupied by ATM cells, a queue for storing ATM cells, and the connected link. Connected link is only applied to input and output buffer of the ATM switch.

Interface method, *getBufferList* returns all of the ATM cells in queue structure, which allows the insertion, deletion, modification, and retrieving of any ATM cell in the queue.

Switch.java

```
public class Switch extends Thread {
    protected Indicator indicator;
    protected int objectId;                    // object ID assigned to this
                                               // thread
    protected String name="";                  // object's name
    protected Graphics g;                      // inherited graphics property
    protected GlobalClock clock;

    protected float switchingCellCredit = 0;
    protected int cellCredit = 0;              // number of time that the internal
                                               // switching can be performed
    protected int switchSize;                  // equals to number of ports
    protected float switchingRate;             // number of cells to be switched
                                               // per tick
    protected RoutingTable routingTable;       // routing table for this switch
}
```

Switch is an inheritance of *Thread* to allow this *Switch* to operate concurrently with other *Thread* objects in the simulator. General attributes of this class consist of the object name and object ID. Object ID is assigned by the *GlobalClock* during registration. Other attributes are reference to the global clock, indicator, as well as Java build-in Graphics object. Both *switchingCellCredit* and *cellCredit* are initialised with 0. These values are reset to 0 with every completion of one second. Finally, *switchSize, switchingRate,* and *routingTable* store the information for the size of ATM switch, switching rate (in unit of number of cells per second), and routing table respectively.

ATM switch maintains two internal methods. It allows the conversion from cells per second to cells per tick for the use in simulator environment and returns a true value for the completion of one second. Among the interfaces methods are retrieving ATM switch size and routing table.

BanyanSwitch.java

```
public class BanyanSwitch extends Switch {
    protected int totalStages;                      // number of stages in Banyan
                                                    // switch
    protected Buffer buffer[][];                    // number of buffers in this
                                                    // switch
    protected InjectedHeader inHeader[][]; // contain the information for the
                                                    // head-of-line ATM cell
}
```

BanyanSwitch is an inheritance of *Switch* class. Hence, it is not only behaved as ATM switch but also as a thread. *BanyanSwitch* makes use of *totalStage* to store a number of stages. *buffer[][]* is a two dimensional arrays for all the buffers contained in this object and *inHeader* is a two dimensional injected header cell for the corresponding input buffers.

When this object is instantiated, both objects *Buffer* and *InjectedHeader* are also instantiated through the *BanyanSwitch* constructor.

```
Buffer buff[][] = new Buffer[ switchSize ][ totalStages + 1 ];
inHeader = new InjectedHeader[ switchSize ][ totalStages ];
```

The number of buffers, i.e. *SwitchSize* x (*TotalStages* + 1) but not *SwitchSize* x *TotalStages* has been explained in chapter 4, section 4.2. Due to Java array range for size N is from 0 to N-1 but not 1 to N, as a result, first buffer in every stage must be assigned with [0][which stage] and so forth. Same concept is applied to injected header cell too.

Five important internal methods are used in *BanyanSwitch,* i.e. *switchCell, getSelectedInlet, banyan2x2, banyan4x4, and banyan8x8.* These methods are declared internally because they need not accessed by other objects. *switchCell* simply switches an ATM cell from input buffer to its output buffer if the output buffer is not full. *getSelectedInlet* return the inlet that should be selected (or neither one is selected) when internal blocking occurs. *Banyan2x2* performs switching within a switching element. Both *Banyan4x4* and *Banyan8x8* determine the entire routing path within a Banyan 4x4 and Banyan 8x8 respectively. They will be further explained in the following sub-sections.

Banyan 4x4 Implementation

The logic for Implementation of Banyan 4x4 is firstly, Perform first stage switching by inserting information of cell into *InjectedHeader*, this is followed by perform switching for stage 1's first switching element and second element. Secondly, perform second stage switching by inserting information of cell into *InjectedHeader* and followed by perform switching for stage 2's first switching element and second element.

```
// Banyan 4x 4 switching
Private void banyan4x4() {
    // 1.Insert the first cell from each input buffer (stage 1) into injected
    // header level 1
    setInjectedHeader( 0 );

    // 2.Perform the 1st stage switching
    banyan2x2( getInjectedHeader( 0, 0 ), getInjectedHeader( 1, 0 ),
        getBuffer( 0, 0 ), getBuffer( 1, 0 ), getBuffer( 0, 1 ), getBuffer( 2, 1 ), 1 );

    banyan2x2( getInjectedHeader( 2, 0 ), getInjectedHeader( 3, 0 ),
        getBuffer( 2, 0 ), getBuffer( 3, 0 ), getBuffer( 1, 1 ), getBuffer( 3, 1 ), 1 );

    setInjectedHeader(1 );

    // 4.Perform the 2nd stage switching
    banyan2x2( getInjectedHeader( 0, 1 ), getInjectedHeader( 1, 1 ),
```

```
        getBuffer( 0, 1 ), getBuffer( 1, 1 ),getBuffer( 0, 2 ), getBuffer( 1, 2 ), 2 );

    banyan2x2( getInjectedHeader( 2, 1 ), getInjectedHeader( 3, 1 ),
        getBuffer( 2, 1 ), getBuffer( 3, 1 ),getBuffer( 2, 2 ), getBuffer( 3, 2 ), 2 );
}
```

Banyan 8x8 Switching Implementation

The logic for Implementation of Banyan 8x8 is similar to Banyan 4x4 where stage 1 switching perform before stage 2 switching and followed by stage 3 switching.

```
// Banyan 8x8 ports switching
private void banyan8x8() {
    // 1.Insert the first cell from each input buffer (stage 1) into injected
    //  header level 1
    setInjectedHeader( 0 );

    // 2.Perform the 1st stage switching
    banyan2x2( getInjectedHeader( 0, 0 ), getInjectedHeader( 1, 0 ),
        getBuffer( 0, 0 ), getBuffer( 1, 0 ), getBuffer( 0, 1 ), getBuffer( 4, 1 ), 1 );

    banyan2x2( getInjectedHeader( 2, 0 ), getInjectedHeader( 3, 0 ),
        getBuffer( 2, 0 ), getBuffer( 3, 0 ), getBuffer( 2, 1 ), getBuffer( 6, 1 ), 1 );

    banyan2x2( getInjectedHeader( 4, 0 ), getInjectedHeader( 5, 0 ),
        getBuffer( 4, 0 ), getBuffer( 5, 0 ), getBuffer( 1, 1 ), getBuffer( 5, 1 ), 1 );

    banyan2x2( getInjectedHeader( 6, 0 ), getInjectedHeader( 7, 0 ),
        getBuffer( 6, 0 ), getBuffer( 7, 0 ), getBuffer( 3, 1 ), getBuffer( 7, 1 ), 1 );

    // 3.Insert the first cell from each input buffer (stage 1) into injected
    //  header level 2
    setInjectedHeader( 1 );

    // 4.Perform the 2nd stage switching
    banyan2x2( getInjectedHeader( 0, 1 ), getInjectedHeader( 1, 1 ),
        getBuffer( 0, 1 ), getBuffer( 1, 1 ), getBuffer( 0, 2 ), getBuffer( 2, 2 ), 2 );

    banyan2x2( getInjectedHeader( 2, 1 ), getInjectedHeader( 3, 1 ),
        getBuffer( 2, 1 ), getBuffer( 3, 1 ), getBuffer( 1, 2 ), getBuffer( 3, 2 ), 2 );

    banyan2x2( getInjectedHeader( 4, 1 ), getInjectedHeader( 5, 1 ),
        getBuffer( 4, 1 ), getBuffer( 5, 1 ), getBuffer( 4, 2 ), getBuffer( 6, 2 ), 2 );
```

```
        getBuffer( 0, 1 ), getBuffer( 1, 1 ),getBuffer( 0, 2 ), getBuffer( 1, 2 ), 2 );

    banyan2x2( getInjectedHeader( 2, 1 ), getInjectedHeader( 3, 1 ),
        getBuffer( 2, 1 ), getBuffer( 3, 1 ),getBuffer( 2, 2 ), getBuffer( 3, 2 ), 2 );
}
```

Banyan 8x8 Switching Implementation

The logic for Implementation of Banyan 8x8 is similar to Banyan 4x4 where stage 1 switching perform before stage 2 switching and followed by stage 3 switching.

```
// Banyan 8x8 ports switching
private void banyan8x8() {
    // 1.Insert the first cell from each input buffer (stage 1) into injected
    //  header level 1
    setInjectedHeader( 0 );

    // 2.Perform the 1st stage switching
    banyan2x2( getInjectedHeader( 0, 0 ), getInjectedHeader( 1, 0 ),
        getBuffer( 0, 0 ), getBuffer( 1, 0 ), getBuffer( 0, 1 ), getBuffer( 4, 1 ), 1 );

    banyan2x2( getInjectedHeader( 2, 0 ), getInjectedHeader( 3, 0 ),
        getBuffer( 2, 0 ), getBuffer( 3, 0 ), getBuffer( 2, 1 ), getBuffer( 6, 1 ), 1 );

    banyan2x2( getInjectedHeader( 4, 0 ), getInjectedHeader( 5, 0 ),
        getBuffer( 4, 0 ), getBuffer( 5, 0 ), getBuffer( 1, 1 ), getBuffer( 5, 1 ), 1 );

    banyan2x2( getInjectedHeader( 6, 0 ), getInjectedHeader( 7, 0 ),
        getBuffer( 6, 0 ), getBuffer( 7, 0 ), getBuffer( 3, 1 ), getBuffer( 7, 1 ), 1 );

    // 3.Insert the first cell from each input buffer (stage 1) into injected
    //  header level 2
    setInjectedHeader( 1 );

    // 4.Perform the 2nd stage switching
    banyan2x2( getInjectedHeader( 0, 1 ), getInjectedHeader( 1, 1 ),
        getBuffer( 0, 1 ), getBuffer( 1, 1 ), getBuffer( 0, 2 ), getBuffer( 2, 2 ), 2 );

    banyan2x2( getInjectedHeader( 2, 1 ), getInjectedHeader( 3, 1 ),
        getBuffer( 2, 1 ), getBuffer( 3, 1 ), getBuffer( 1, 2 ), getBuffer( 3, 2 ), 2 );

    banyan2x2( getInjectedHeader( 4, 1 ), getInjectedHeader( 5, 1 ),
        getBuffer( 4, 1 ), getBuffer( 5, 1 ), getBuffer( 4, 2 ), getBuffer( 6, 2 ), 2 );
```

```
banyan2x2( getInjectedHeader( 6, 1 ), getInjectedHeader( 7, 1 ),
    getBuffer( 6, 1 ), getBuffer( 7, 1 ), getBuffer( 5, 2 ), getBuffer( 7, 2 ), 2 );

// 5.Insert the first cell from each input buffer (stage 1) into injected
//  header level 3
setInjectedHeader( 2 );

// 6.Perform the 2nd stage switching
banyan2x2( getInjectedHeader( 0, 2 ), getInjectedHeader( 1, 2 ),
    getBuffer( 0, 2 ), getBuffer( 1, 2 ), getBuffer( 0, 3 ), getBuffer( 1, 3 ), 3 );

banyan2x2( getInjectedHeader( 2, 2 ), getInjectedHeader( 3, 2 ),
    getBuffer( 2, 2 ), getBuffer( 3, 2 ), getBuffer( 2, 3 ), getBuffer( 3, 3 ), 3 );

banyan2x2( getInjectedHeader( 4, 2 ), getInjectedHeader( 5, 2 ),
    getBuffer( 4, 2 ), getBuffer( 5, 2 ), getBuffer( 4, 3 ), getBuffer( 5, 3 ), 3 );

banyan2x2( getInjectedHeader( 6, 2 ), getInjectedHeader( 7, 2 ),
    getBuffer( 6, 2 ), getBuffer( 7, 2 ), getBuffer( 6, 3 ), getBuffer( 7, 3 ), 3 );
```

Banyan switching execution is performed by the interface method *run.* Whenever the clock of *BanyanSwitch* is running, the following tasks will be executed:

- If it is the completion of one second, resets TCT values for all the records in routing table.
- Calculates *switchingCellCredit* and *cellCredit.*
- Performs *cellCredit* times of switching.
- Indicates to the global clock that it is ready to move to the next tick.

Below is the coding for method *run.*

```
// When the clock is still running
while (!clock.isClockStop()) {
    // If it is one second, then reset the tct value in routing table
    if ( isOneSec( tick ) ) {
        routingTable.resetTct();
    }

    // Calculate the credit for performing the internal switching
    switchingCellCredit += switchingRate;
```

```
        cellCredit = ( int )switchingCellCredit;
        switchingCellCredit -= cellCredit;

        // Perform cellCredit times of switching
        while( cellCredit > 0 ) {
            if( switchSize == 8 ) banyan8x8();
            else banyan4x4();
            cellCredit--;
        }

        // Give chance to other thread object to execute
        while (!clock.isAllowedToGo( objectId ) ) {
            clock.tick(objectId);
            try {
                Thread.sleep(1);
            }
            catch (InterruptedException e) {
                System.err.println("Exception Occured");
            }
        }
    }
```

RoutingTable.java

```
public class RoutingTable {
    private Queue ipPortQueue = new Queue();// input port number
    private Queue ipVpiQueue = new Queue(); // input VPI
    private Queue ipVciQueue = new Queue(); // input VCI
    private Queue opPortQueue = new Queue();// output port number
    private Queue opVpiQueue = new Queue(); // output VPI
    private Queue opVciQueue = new Queue(); // output VCI
    private Queue pcrQueue = new Queue();   // peak cell rate ( cells/sec )
    private Queue mcrQueue = new Queue();   // minimum cell rate ( cells/sec )
    private Queue tctQueue = new Queue();   // number of cells transferred
                                            // within current second
    private int rows = 0;                   // number of records in routing
                                            // table
}
```

Routing table maintains nine attributes. Eight of these attributes are the queue object for input port, input VPI, input VCI, output port, output VPI, output VCI, PCR, MCR, and TCT. The other one is used to store current record for the routing table. All of the queues in routing table have the same length. Each of these values in the same position within queue represents a single record for ATM application; i.e. a record in routing table consists of these eight values. When inserting a record into routing table, all of these queues must be inserted and number of rows is increased by one and vice versa.

Interface methods included in routing table is inserting and removing record, increase TCT value and retrieving any of the eight values above based on queue position or input VPI/VCI values.

5.2 Component Testing

Component testing is done in several classes like *QueueNode.java*, *Queue.java*, *RoutingTable.java*, *Buffer.java*, *Switch.java*, and *BanyanSwitch.java.*

Queue.java and QueueNode.java

Both Queue and QueueNode are tested by instantiate a Queue object and three different types of other objects. These objects are then inserted into the Queue object. Finally, the content of the Queue object is printed out to ensure the Queue object is working properly. Since QueueNode is instantiated by Queue, it means that *QueueNode* is working properly too. Example below shows that three Integer, Boolean and String objects are inserted into Queue object by using the method *insertAtFront* and *insertAtBack,* The output shows the Queue is working properly.

1. Instantiate object.

```
Queue a = new Queue();
Queue queue = new Queue();
Integer a = new Integer(3);
Boolean b = new Boolean( true );
String c = new String("Test");
```

2. Input data into queue object.

```
queue.insertAtFront( a );
```

```
queue.insertAtFront( b );
queue.insertAtBack( c );
```

3. Display the output.

```
print( Queue q ) {
    for( int i = 0; i < q. getNumberOfNodes(); i++ )
        System.out.println( q.getMiddleObject( i ) );
}
```

Output:

```
true
3
Test
```

AtmCell.java

Testing on AtmCell java is easy. A new AtmCell object is instantiated and assigned VPI/VCI value. At the end, the VPI/VCI value is printed out.

1. Instantiate object and insert VPI/VCI value.

```
AtmCell cell = new AtmCell();
cell.setVPI( 10 );
cell.setVCI(50 );
```

2. Display output.

```
System.out.println( "Value for VPI:VCI = " + cellsgetVPI() + ":" +
cellsgetVPI( ) );
```

Output:

```
Value for VPI:VCI =10:5
```

RoutingTable.java

RoutingTable is tested by instantiating RoutingTable object and insert several records into this object. The output shown is exactly same as the input data.

1. Instantiate object of *RoutingTable* and insert data into it.

```
RoutingTable rt = new RoutingTable();
rt.insertData(0, 0, 0, 0, 0, 0, 2000, 1000 );
rt.insertData(0, 1, 0, 0, 1, 0, 2000, 1000 );
rt.insertData(1, 0, 1, 0, 0, 1, 2000, 1000 );
rt.insertData(1, 1, 1, 2, 1, 1, 100000,2000 );
```

2. Display output.

```
for( int i=0; i < rt.getRows(); i++ ) {
    Sysem.out.println( rt.getInptPort( i ) + rt.getInputVPI( i ) + " " +
    rt.getInputVCI( i )  + " "rt.getOutputPort( i ) + " "+ rt.getOutputVPI( i )
    + " " + rt.getOutputVCI( i ) + " " + rt.getPcr( i ) + rt.getMcr( i ) +
    rt.getTct( i );
}
```

Output:

```
0 0 0 0 0 0 2000 1000 0
0 1 0 0 1 0 2000 1000 0
1 0 1 0 0 1, 2000 1000 0
1 1 1 2 1 1 100000 2000 0
```

Buffer.java

Testing for Buffer class focuses on attributes bufferQueue, max_buff_size, and current_buff_size. bufferQueue is a link list of ATM cell, therefore testing on retrieving the ATM cell from bufferQueue is important to make sure that the cell can be inserted, removed and retrieved correctly. A simple test can be done on max_buff_size and current_buff_size to make sure these two attributes work correctly.

Switch.java and BanyanSwitch.java

Both Switch and BanyanSwitch can be tested together by instantiating a BanyanSwitch object. Since BanyanSwitch consists of other objects like Buffer and RoutingTable (which have been tested), testing for BanyanSwitch is focuses on the algorithm for cell routes within switching elements.

5.3 Module Testing

The major purposes of the simulator module testing are switching testing and ATM application cell rate testing. The following two sub-sections describe about these two test.

5.3.1 Switching Testing

Switching testing is carried out with the purpose to make sure that internal switching for ATM cells is correctly and successfully reaching desired output port. Switching testing is further splits into Banyan 4x4 and Banyan 8x8 testing. The test sequence for both are firstly, initialises routing table. Secondly, pump in ATM cells into input buffers. Finally, perform switching and get the result.

Banyan 4x4

The routing table for Banyan 4x4 is initialised with the following data.

Table 5.1: Routing Table for Banyan 4x4 Testing

Input port	Input VPI	Input VCI	Output Port	Output VPI	Output VCI
0	0	0	0	0	0
0	1	0	1	1	0
1	0	1	0	0	1
1	1	1	2	1	1
3	2	3	3	2	3

From routing table, two ATM applications that pass through input port 0 are identified as VPI/VCI 0/0 and VPI/VCI 1/0. ATM application with VPI/VCI 0/0 is addressed to output port 0 (i.e. it is routed back to the same port) and ATM application with VPI/VCI 1/0 is addressed to output port 1. Input port 1 has two applications too and they are identified as VPI/VCI 0/1 and 1/1, the output ports are 0 and 2 respectively. Input port 2 does not have any incoming cell and, for input port 3, only one incoming application (VPI/VCI 2/2) which is addressed to the same output port. The output VPI/VCI does not need to be changed since there is no conflict for all the ATM applications that pass through this switch.

After establishing the routing table, the ATM cells are pumped into the input buffers; only two cells per ATM application are pumped in. Figure 5.1 illustrates the sequence of ATM cells in input buffer from port 0 to port 3.

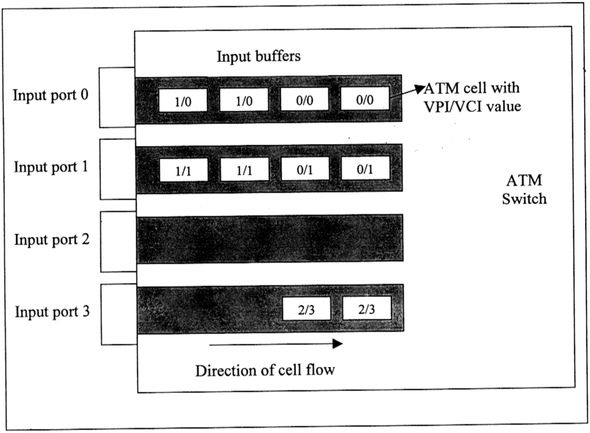

Figure 5.1: Cells Pumped into Input Buffer

The output below shows that all the cells are switched properly

Output buffer at port 0: 0/0, 0/1, 0/0, 0/1

Output buffer at port 1: 1/0, 1/0

Output buffer at port 2: 1/1, 1/1

Output buffer at port 3: 2/3, 2/3

Banyan 8x8

Table 5.2: Routing Table For Banyan 8x8 Testing

Input port	Input VPI	Input VCI	Output Port	Output VPI	Output VCI
0	0	0	0	0	0
0	1	0	1	1	0
1	0	1	0	0	1
1	1	1	2	1	1
3	2	3	3	2	3
4	4	3	3	4	3
5	5	3	6	5	3
6	6	3	7	6	3

Again, only two cells per ATM application is pumped into input buffers according to the routing table and the output below shows that all the cells are switched properly.

Output buffer at port 0: 0/0, 0/1, 0/0, 0/1

Output buffer at port 1: 1/0, 1/0

Output buffer at port 2: 1/1, 1/1

Output buffer at port 3: 4/3, 2/3, 4/3, 2/3

Output buffer at port 6: 5/3, 5/3

Output buffer at port 7: 6/3, 6/3

5.3.2 Cell Rate Testing

Cell rate testing is done with the objective to ensure that the switching rate for ABR application must in between MCR and PCR value, i.e. the TCT value must greater or equal to MCR and less or equal to PCR within one-second period. Testing is done by adding in a line of JAVA code into the program. This is to prompt the number of TCT when there is an ATM cell switched within the switch. Output shows that TCT value will increase by one when the ATM cell is switched. Within one-second time, the value of TCT is increase but never grows higher than PCR value.

5.4 System Testing

System testing is done by building in a new testing package – *switchtester* package. Two new classes built are *Operation.java* and *Tester.java. Operation.java* consists of several Java static function like initialise routing table, insert ATM cell into buffers, and print the content of output buffer to a temporary result file.

Tester.java is a driver to perform the system testing. It accepts input from user through Java applet. Among the important inputs are switch size (or the number of ports), ATM switch switching rate, buffer size (all the buffers will have same size), and the resulted file name.

Tester class creates the routing table and insert ATM cells into input buffers. Once user click the start button, simulation start by creating the ATM Switch thread, follow by perform the internal switching. At the same time, the value of current tick time, *switchingCellCredit*, and *cellCredit* is printed to result file.

This switching process is carrying on until user press end button. Finally, the content of output buffers will be printed to the result file. Testing is successful if the result file compared to desired output values is same.

5.5 Summary

This chapter gives an idea on how the implementation and testing processes on the switching simulator were carried out. Class implementation explains the attributes of each class, which are declared together with their data types. The class implementation also explains the methods in each class. This section is followed by a description on initialisation in Banyan Switch and the execution of Banyan 4x4 and Banyan 8x8. The driver for executing Banyan Switch is also shown.

Testing of ATM switching stimulator begins with component testing, followed by module testing and system testing. Component testing focuses on the individual testing of each class. Meanwhile, module testing focuses on switching testing and cell rate testing. The system testing tests the whole simulator to ensure that it runs on the actual environment. The testing result obtained shows that the simulator is working correctness.