Path Allocation in a Three-Stage Broadband Switch with Intermediate Channel Grouping

Martin Collier and Tommy Curran

School of Electronic Engineering,
Dublin City University, Glasnevin, Dublin 9, Ireland.

Abstract

A method for path allocation is described for use with three-stage ATM switches which feature multiple channels between the switch modules in adjacent stages. The method is suited to hardware implementation using parallelism to achieve a very short execution time. This allows path allocation to be performed anew in each time slot. A detailed description of the necessary hardware is presented. This hardware counts the number of cells requesting each output module, allocates a path through the intermediate stage of the switch to each cell, and generates a routing tag for each cell, indicating the path assigned to it.

1: Introduction.

A range of designs has been proposed for broadband switching (e.g., [1-4]). Many of these proposals are only practical in the design of small switches. For example, the number of switch elements in the Sunshine switch [3] becomes excessive as the switch size increases [5]. A different approach must be taken to the design of large switches.

An obvious method of implementing a large switch, given these constraints on switch size, is to design the switch with multiple stages, where each stage consists of smaller switch modules. The Clos network [6] exemplifies a switch of this type. This solution typically introduces a new problem whereby multiple paths from source to destination become available. Thus, even if the individual switch modules possess the self-routing feature, this feature is not retained by the overall switch. Some method of path allocation is then necessary, to select among the available paths from source to destination.

We distinguish between two time scales over which paths may be allocated. In one approach, all cells belonging to a virtual circuit are allocated the same path. Thus path allocation is performed at call setup time, and this path is allocated for the duration of the call. In the second approach, path allocation is performed independently in each time slot, and so the path is allocated for the duration of one time slot only. We refer to these two approaches as path allocation at *call* level, and path allocation at *cell* level, respectively.

We consider below the problem of implementing a cell-level algorithm for path allocation in the channel-grouped three stage network of Fig. 1. The hardware implementation must be such that the resulting circuitry is not required to operate at a prohibitively high speed. In practice, this means that the parallelism in the hardware must be maximised. Our motive for adopting a channel-grouped architecture is that it reduces the execution speed required of the path allocation hardware. The use of channel grouping can also improve performance in ATM switches [7,8]. The path allocation algorithm and the hardware necessary to implement it are described below.

2: An algorithm for path allocation at cell level.

2.1: The objectives of a path allocation algorithm.

There are S_1 routes from each input module to each intermediate switch module. There are S_2 routes from each intermediate switch module to each output module. We must choose, for every input cell (if possible) an intermediate switch module through which to pass on the way to the selected destination, such that

0743-166X/93 $03.00 © 1993 IEEE

no input module attempts to route more than S_1 cells via any intermediate switch module, and no intermediate switch module attempts to route more than S_2 cells to any output module, in any one time slot.

Note that, in this problem, an attempt is made to reserve bandwidth for each input cell, such that it can pass through the intermediate stage without blocking. An alternative, and simpler, strategy would not test for the availability of a path from the intermediate stage to the output stage, and would select intermediate stage modules based on some simpler criterion (e.g. random selection). The former strategy is preferred for a number of reasons:

(i) no queueing occurs in the intermediate stage; thus the delay through the intermediate stage is uniform, regardless of the path taken; this makes it possible to preserve cell sequence on a virtual circuit;
(ii) the intermediate stage can never be congested;
(iii) intermediate stage modules can be of simple design, since contention cannot occur.

2.2: Existing algorithms for cell-level path allocation.

A number of solutions to this problem have been proposed, in the special case where $S_1 = S_2 = 1$ [9-11]. These algorithms typically use one bit to represent each channel in the switch. Thus the number of bits being processed by the path allocation algorithm is large for a large switch. The algorithm in [9] was adapted to handle architectures with intermediate channel grouping in [8]. This adaptation increases the complexity, and thus the execution time, of the algorithm.

It is possible to *reduce* the execution time of the path allocation algorithm, for a given intermediate stage bandwidth, through the use of intermediate channel grouping, as demonstrated below.

3: A new algorithm for path allocation.

3.1: Basic principles.

A new and efficient algorithm will now be described. It is suitable for use in a channel-grouped three-stage switch and requires only knowledge obtainable at the input side of the switch. The key to its high performance is the encoding of data concerning the availability of paths into binary words (thereby reducing the number of bits to be processed by the algorithm for a given switch size) and the extensive use of parallelism. It operates on the following quantities:

A_{ir} : the number of channels available from input module i to intermediate switch module r;

B_{rj} : the number of channels available from intermediate switch module r to output module j;

K_{ij} : the number of requests from input module i for output module j.

Note that A_{ir} and K_{ij} need only be local to the input module. The B_{ir}'s must be forwarded to each input module in turn. This is performed by a ring structure connecting each input module. Such an arrangement is shown in Fig. 2, where each row is located at an input module. Let R_{irj} be the number of cells to be routed from input module i to output module j via intermediate switch module r. The values of A_{ir}, B_{rj} and K_{ij} are updated using the procedure *atomic*(i,r,j) described below:

$$\left.\begin{aligned} R_{irj} &= \min(K_{ij}, B_{rj}, A_{ir}) \\ K_{ij} &\leftarrow K_{ij} - R_{irj} \\ B_{rj} &\leftarrow B_{rj} - R_{irj} \\ A_{ir} &\leftarrow A_{ir} - R_{irj} \end{aligned}\right\} \quad atomic(i,r,j)$$

This procedure is 'atomic' in the sense that it is the basic building block from which the path allocation algorithm is constructed. The procedure determines the capacity available from input module i to output module j via intermediate switch module r (i.e. the minimum of A_{ir} and B_{rj}). The number of requests which can be satisfied is equal to the minimum of the number of requests outstanding (K_{ij}) and the available capacity.

A sequential implementation of the path allocation algorithm requires the repeated execution of *atomic*(i,r,j) on a single processor for all possible values of i,r and j. Initially K_{ij} is set equal to the total number of requests from input module i for output module j (which number is obtained by examining the requests at the switch module inputs), A_{ir} is set equal to S_1 and B_{rj} is set equal to S_2.

A parallel implementation requires multiple processors, each executing the *atomic()* procedure for a different set of procedure parameters, subject to the following constraints:

- No two processors shall simultaneously require access to the same quantity. For example, *atomic*(2,0,0) uses A_{20}, B_{00} and K_{20}, so that *atomic*(2,0,X), *atomic*(2,X,0) and *atomic*(X,0,0) cannot be executed concurrently with *atomic*(2,0,0) for any X.

- The data required by a processor for the next iteration of the algorithm should be available locally, or from adjacent processors.

An implementation satisfying these two constraints will now be described.

3.2: Implementation of the algorithm.

The algorithm requires a total of $L_1.L_2$ processors. The detailed operation of the algorithm depends on the number of switch modules in each stage. The simplest case, where $L_1 = L_2 = m$, is described here, but only minor modifications to this algorithm will be required if the values of L_1, L_2, and m are not equal.

Processor X_{ij} is initialised by loading the following three values:

(i) the initial value of K_{ij};

(ii) the initial value of $A_{i,(i+j) \bmod m}$ (i.e., S_1);

(iii) the initial value of $B_{(i+j) \bmod m,j}$ (i.e., S_2).

The algorithm then requires m iterations (iterations zero through m-1). Processor X_{ij} executes *atomic*(i, (i+j-k) mod m, j) during iteration k . After each iteration X_{ij} forwards the updated value of B_{rj} to $X_{(i+1) \bmod m, j}$ and of A_{ir} to $X_{i, (j+1) \bmod m}$, and retains K_{ij}.

The mechanism for passing information to an input cell concerning the path allocated to it will be described in section 3.5. The hardware layout for the case where m = 4 is shown in Fig. 2, which illustrates the array of sixteen processors required, and the contents of their registers during iteration zero of the algorithm. Each row in Fig. 2 contains four processors, which are co-located with the corresponding input module. Each column in Fig. 2 processes requests for a single output module. Thus, for example, the processor in row one and column two of the array handles requests for cells to be routed from input module one to output module two. A total of 64 paths is available through the switch (four for each input-output module pair). The sixteen processors attempt to allocate cells to sixteen of these paths during each iteration. After each iteration, the updated value of A_{ir} is passed to the adjacent processor in the same row, and the updated value of B_{rj} is passed to the adjacent processor in the same column. The directions of data flow are indicated by arrows in Fig. 2. No two processors can allocate a path sharing a channel in the same iteration. Nevertheless, after four iterations, all possible paths have been allocated.

3.3: The processing element.

The processor must execute the *atomic*() procedure, and thus must perform two types of operation:

1. Find the minimum of three numbers.
2. Perform three subtractions.

Fig. 3 shows a possible implementation of the processing element, which uses bit-serial arithmetic. Determination of the minimum requires values to be presented most significant bit first, while bit-serial subtraction requires values which are presented least significant bit first. Hence the processor must be able to perform bit reversal on the quantities processed. A bit-parallel implementation avoids this difficulty, at the cost of increased complexity.

The processor design will involve a trade-off of circuit complexity against operating speed, since there is an upper bound on the permissible execution time.

The time within which the algorithm is required to execute depends on whether cells losing contention are discarded or are queued until the next time slot.

Consider the case where cells losing contention join an input queue. The queue controller, when it submits a cell to the path allocation process, retains a copy in the input buffer. It then awaits an acknowledgement signal from the path allocation hardware, indicating whether the cell has been allocated a path through the switch, or has been discarded. It then submits the copy cell (if the original cell was discarded) or it purges the copy cell and submits the next cell in the input buffer (if the first cell was successfully routed). The acknowledgement must be returned within the duration of one time slot so that cells can be submitted to the switch in successive time slots. Hence the time taken for the path allocation process to execute should be less than the duration of one time slot. This stringent requirement could be relaxed if preservation of cell sequence was not mandatory, since a cell losing contention could then rejoin the queue in a later time slot.

No acknowledgements are required if there is no input queueing. Hence the path allocation process need not execute within one time slot. However, it must still be possible to submit cells to the switch in successive time slots. Additional copies of the path allocation hardware are required to ensure this, equal in number to the execution time of the path allocation process in time-slots. For example, if the path allocation process has an execution time of two time slots, cells arriving during even-numbered time slots will be processed by one copy of the hardware, and cells arriving during odd-numbered time slots will be processed by another. Hence a tradeoff may be performed during switch

design between processor speed and the number of processors required. Omitting the input queues has the additional advantage that no hardware is required to generate the acknowledgements.

3.4: Counting requests

Hardware is also needed in each input module to perform the following tasks before and during the path allocation process :

- to count the number of requests for each output module so as to obtain the initial values of the K_{ij}'s;
- to forward a routing tag based on the results of path allocation to each input cell.

The counting of requests can be performed by the hardware of Fig. 4. This merges the input cells with a set of control packets, one for each input module, in a Batcher sorter (with n_1+m inputs and outputs), in the manner described in [12]. Idle inputs submit an inactive packet to the sorter. The sorter output contains (starting at the *lowest*-numbered output in Fig. 4) the control packet for output module 0, followed by all the data cells intended for output module 0, followed by the control packet for output module 1, etc. The inactive packets are sorted to the highest-numbered outputs.

The address generators in Fig. 4 serve different purposes during the counting of requests and in routing tag assignment. When counting requests, they determine the type of packet which is present at the corresponding output of the Batcher network, and generate a bit (the *identity* bit) which is 1 for a control packet or an inactive packet and 0 for a data packet (cell). A copy of the identity bit is stored in a one-bit register which is connected to its neighbours in adjacent address generators in such a way as to form a shift register.

These bits are shifted n_1+m times (in the direction shown in Fig. 4) from address generator to address generator, and hence into a counter of wordlength $\lceil \log_2(m) \rceil$ which is reset upon receiving a 1 (a control packet or inactive packet) and incremented on receipt of a 0 (data cell). Each time a 1 is received the counter contents are rotated into $K_{i,0}$, $K_{i,0}$ is rotated into $K_{i,1}$, etc. After n_1+m shifts the appropriate values are stored in the K_{ij} registers.

3.5: Routing tag assignment

The algorithm used for routing tag assignment is best described by means of an example. We assume that there are four intermediate switch modules, labelled ISM_0 to ISM_3, and consider how to assign routing tags to the cells from input module zero (IM_0) which have requested output module one (OM_1).

Let us assume that four cells from IM_0 have requested output module zero (OM_0), and that seven cells from IM_0 have requested OM_1. Hence, the first thirteen outputs of the Batcher sorter are as shown in Fig. 5, after the data cells have been merged with control packets, as discussed in section 3.4.2.

The results of the path allocation process for cells from IM_0 requesting OM_1 are produced by processor X_{01}. Some means must be found to forward these results to the relevant cells, which appear at sorter outputs six through twelve. We assume that paths are allocated in the order shown in Table I. Thus one cell loses contention.

Note that a connection has already been established from processor X_{01} (via the routing packet generator for OM_1) to sorter output five, as shown in Fig. 5. Thus the relevant routing information may be easily forwarded to sorter output five. The problem remains of how to relay this information to the data cells at sorter outputs six through twelve.

The solution to this problem would be trivial if all seven cells were to be allocated a route via the same intermediate switch module. The address generator at sorter output five could generate seven tokens, each granting access to that intermediate switch module. These seven tokens would be forwarded to sorter output six, where one would be seized. The remaining six tokens would be forwarded to sorter output seven. After seven iterations of this procedure, all seven data cells would possess the necessary token, stored in the associated address generator.

This token passing algorithm is easily implemented in hardware. Each address generator stores data concerning tokens as a routing packet containing two fields, which are the token address (indicating the address of the intermediate switch module to which access is being granted) and the token count (indicating the number of such tokens). Routing packets are received by the address generators associated with control packets (such as that at output five in our example), from the appropriate routing packet generator. Other address generators receive a routing packet from their neighbours at the start of each iteration of the algorithm. If the token count is zero, this routing packet is discarded. Otherwise the token count is decremented, and the routing packet is stored in the address generator, and is forwarded to the adjacent generator at the start of the next iteration.

The hardware required is thus very simple, with a

unidirectional flow of data from address generator to address generator. Once all the tokens have been distributed (after seven iterations), subsequent iterations of the algorithm cause no changes in the route assignments, so that the number of iterations performed is not critical, provided it is not too low.

The same hardware may be used to solve the problem of assigning routing tags to the data cells when paths through more than one intermediate switch module have been allocated. The technique is to execute multiple passes of the above algorithm, each for a different token address, and each with an appropriately chosen value for the token count. An address generator may then receive a succession of routing packets, each with a different token address. However, only the last such packet received will contain the correct token.

Five passes of the algorithm suffice to supply all the data cells with the correct token in the example of Fig. 5, as shown below.

Pass One: Seven tokens are received for ISM_1. At the end of this pass, all seven cells have been assigned a route through ISM_1.

Pass Two: Four (i.e., 7-3) tokens are received for ISM_0. At the end of this pass, four cells have been assigned a route through ISM_0. Three cells (i.e., those at sorter outputs ten through twelve) have retained their tokens for ISM_1.

Thus, at the end of Pass Two, the correct number of cells has been assigned a route through ISM_1.

Pass Three: Three (i.e., 7-3-1) tokens are received for ISM_3.

Pass Four: Three (i.e., 7-3-1-0) tokens are received for ISM_2.

Pass Five: One (i.e., 7-3-1-0-2) null token is received, indicating that one cell has lost contention.

At the end of Pass Five, a route through each intermediate switch module has been assigned to the correct number of cells, as shown in Fig. 5. In particular, no route is assigned via ISM_3. The key to achieving the required result was the correct choice of token count for each pass of the algorithm. The sequence chosen was {7,4,3,3,1}. This is identical to the sequence of K_{01} values generated by processor X_{01} during the path allocation process, i.e., it is equal to the number of cells not yet allocated a path after each cycle of path allocation. Thus, the necessary sequence of token counts may be obtained from the TC_{out} output of the processor, shown in Fig. 3.

Each pass of the algorithm can commence after the first iteration of the preceding pass. Hence the execution time increases only slowly with the number of passes. Routing tag assignment can thus be carried out as follows.

The routing packet generator associated with processor X_{ij} generates a routing packet, concurrently with each iteration of the path allocation algorithm. The token address is the value of r, the intermediate switch module through which the processor is attempting to route cells. This value can be easily generated by a counter decremented after every iteration of the path allocation algorithm. The token count is set to the value of K_{ij}. This routing packet is forwarded to the address generator associated with control packet j through the Batcher network in Fig. 4.

Each address generator performs the actions illustrated in Fig. 6 after every iteration of the path allocation algorithm. Upon completion of the path allocation algorithm, the value of K_{ij} is equal to the number of cells which have lost contention. This is forwarded to the relevant cells in a special routing packet, whose token address field indicates that these are null tokens, which flag the corresponding cells as having lost contention.

Upon completion of this process, the token address and token count values stored by each address generator comprise a unique routing tag, which is then prefixed to the associated data cell. The cell is submitted to the first stage of the switch, and is thereby routed to the appropriate intermediate switch module.

The operation of this algorithm for the example considered earlier is shown in Table II. The data stored in each address generator at the end of each iteration of the algorithm are shown. After nine iterations, the routing tags have been successfully assigned.

This routing assignment algorithm has the benefit of simplicity but its execution time is quite long. However, it operates in parallel with the path allocation process, although it takes longer to execute, because of the delay in propagating routing packets through the address generators.

4: A design example.

A 3072 x 8192 switch can be constructed by choosing $L_1 = L_2 = m = 32$, $n_1 = 96$, $n_2 = 256$, $S_1 = 4$ and $S_2 = 8$ in the switch shown in Fig. 1. The probability of cell loss due to non-allocation of paths through this switch has been obtained by simulation. The simulation model assumes that all switch inputs have a 100% load, that traffic is uniformly distributed among the output modules, and that cells not allocated paths on the first attempt are discarded. The resulting figure for cell loss probability is below 10^{-10}. The

performance of this type of switch will be considered in greater detail in a future paper.

The input module dimensions are 128 x 128. At most 96 of the 128 inputs carry active data. The dimensions of the intermediate and output stage modules are 128 x 256, and 256 x 256 respectively. The input and intermediate switch modules can be of simple design, since they are contention-free. Thus the only stage of the switch which represents a major design challenge is the output stage, where the 256 x 256 switch modules should also introduce a low cell loss probability.

We now estimate the speed required of the path allocation hardware. The counting of requests requires a Batcher network with 128 inputs, and requires 128 (i.e., $n_1 + m$) clock cycles to execute. One execution of the *atomic*() procedure, if implemented using the technique of Fig. 3, requires approximately 15 clock cycles, depending on the implementation. Thus perhaps 480 (i.e., 15.32) cycles will be needed to test all possible paths. The number of processors required is 1024 (i.e., 32.32), but the IC count should be relatively low because of the simplicity of the processor design. The route allocation process will add at most (96-4).2 = 184 cycles to the execution time, assuming that two cycles are required to propagate routing packets through the address generators (at least 4 cells will be allocated a path). This represents a total of 792 clock cycles. Thus the clock rate required for the complete algorithm to execute within one time slot is approx. 290 MHz (neglecting any speed-up required to make good propagation delays). The required clock speed could be reduced using a more complex processor design (e.g., using bit-parallel arithmetic). An alternative is to construct two copies of the hardware, which process requests in alternate time-slots. The clock rate required in this case should be below 150 MHz.

5: Conclusions.

A new algorithm for path allocation in three-stage broadband networks has been described. A complete hardware implementation of this algorithm has been presented, including a method for generating the initial data required by the algorithm, and for forwarding the results to each cell at the input side of the switch, in the form of a routing tag. The operating speed required of the design ($\cong$ 150 MHz) appears within the capabilities of VLSI technology in the short term, allowing a switch with a throughput of 400 Gb/s to be constructed.

References

[1] J.S. Turner, "Design of a broadcast packet switching network", *IEEE Trans. Commun.*, Vol. COM-36, no. 6, pp. 734-743, June 1988.

[2] A. Huang and S. Knauer, "Starlite: a wideband digital switch", *Globecom '84 Conference Record*, pp. 121-125, Nov. 1984

[3] J.N. Giacopelli, W.D. Sincoskie and M. Littlewood, "Sunshine: a high performance self-routing broadband packet switch architecture", *Proc. of the International Switching Symposium*, Stockholm, 1990, vol. III, pp. 123-129.

[4] H. Kuwahara, N. Endo, M. Ogino and T. Kozaki, "A shared buffer memory switch for an ATM exchange", *Proc. ICC '89*, pp. 118-122.

[5] T.T. Lee, "A modular architecture for very large packet switches", *IEEE Trans. Commun.*, Vol. COM-38, no. 7, pp. 1097-1106, July 1990.

[6] C. Clos, "A study of non-blocking switching networks", *Bell Systems Tech. Journal*, vol. 32, no. 2, pp. 406-424, Mar. 1953.

[7] A. Pattavina, "Multichannel bandwidth allocation in a broadband packet switch", *IEEE J. Select. Areas Commun.*, Vol. SAC-6, no. 9, pp. 1489-1499, Dec. 1988.

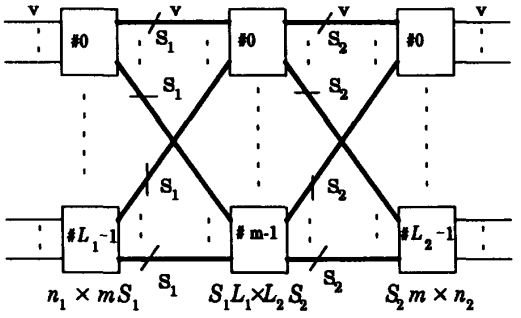
[8] K.Y. Eng and C.-L. I, "Performance analysis of a growable architecture for broadband packet (ATM) switching", *Globecom '89 Conference Record*, pp. 1173-1180.

[9] K.Y. Eng, M.J. Karol and Y.S. Yeh, "A growable packet (ATM) switch architecture: design principles and applications", *Globecom '89 Conference Record*, pp. 1159-1165.

[10] A. Cisneros, "Large packet switch and contention resolution device", *Proc. of the International Switching Symposium*, Stockholm, 1990, vol. III, pp. 77-83.

[11] R. Proctor and T. Maddern, "Synchronous ATM switching fabrics", *Proc. of the International Switching Symposium*, Stockholm, 1990, vol. IV, pp. 109-114.

[12] C. Day, J. Giacopelli and J. Hickey, "Applications of self-routing switches to LATA fiber optic networks", *Proc. ISS '87*, pp. 519-523.



V: channel rate (155 Mb/s)
L_1 (L_2) : the number of input (output) modules
n_1 (n_2) : the number of input (output) ports per input (output) module
m: the number of intermediate switch modules
S_1 (S_2) : the number of channels in the channel group connecting each input (output) module to each intermediate switch module

Fig. 1: A three-stage switch with intermediate channel grouping.

iteration	0	1	2	3
routing via	ISM_1	ISM_0	ISM_3	ISM_2
no. of paths	3	1	0	2

Table I: An example of path allocation.

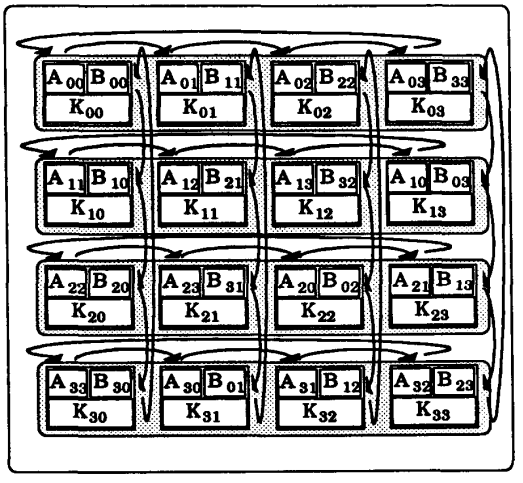
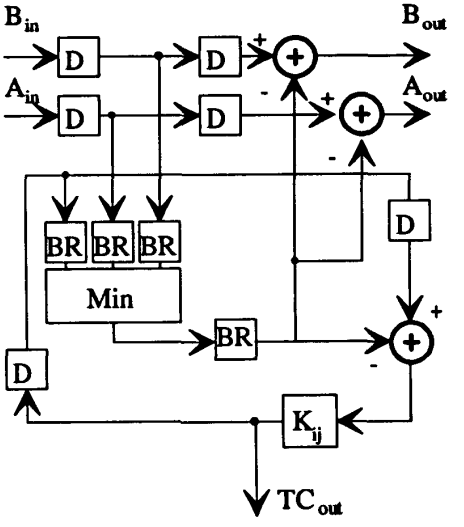
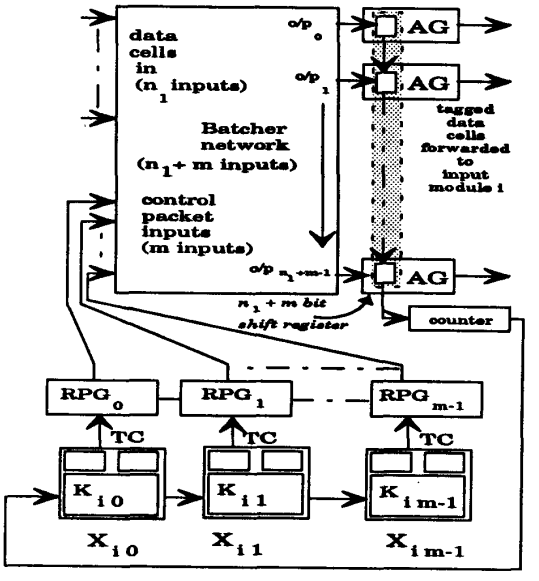


Fig. 2: Processor contents during iteration 0.



MUX : Multiplexor
D: Synchronisation Delays
BR: Bit Reversal
TC: Token Count

Fig. 3: The *atomic*() processor implementation.



X_{ij}: Processor (input module i, output module j)
RPG: Routing Packet Generator
AG: Address Generator

Fig. 4: Request count, routing tag generation.

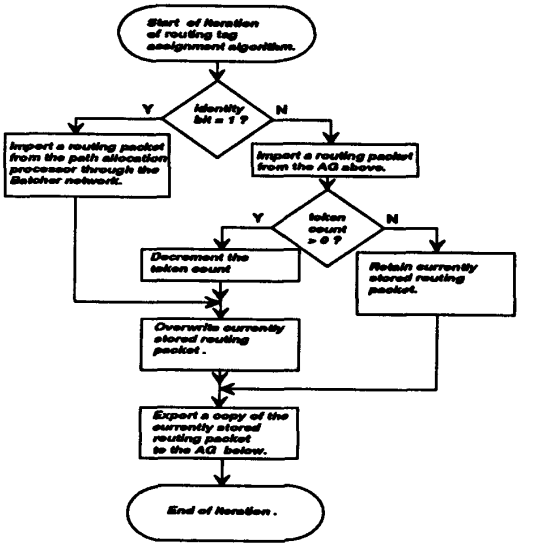


Fig. 6: Address generator (AG) operations during routing tag assignment.

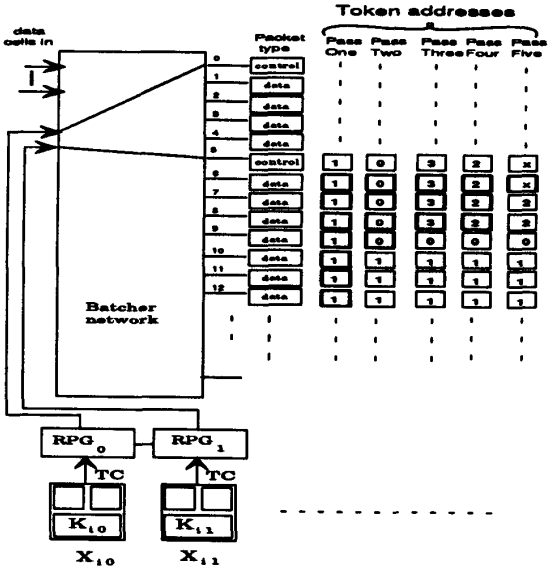


Fig. 5: An example of route assignment.

sort o/p no.	packet type	iteration 0	1	2	3	4	5	6	7
		routing packet contents (*token address, token count*)							
5	OM_i control	1,7	0,4	3,3	2,3	x,1	x,0	x,0	x,0
6	OM_i *data*	-	1,6	0,3	3,2	2,2	x,0	x,0	x,0
7	OM_i *data*	-	-	1,5	0,2	3,1	2,1	2,1	2,1
8	OM_i *data*	-	-	-	1,4	0,1	3,0	2,0	2,0
9	OM_i *data*	-	-	-	-	1,3	0,0	0,0	0,0
10	OM_i *data*	-	-	-	-	-	1,2	1,2	1,2
11	OM_i *data*	-	-	-	-	-	-	1,1	1,1
12	OM_i *data*	-	-	-	-	-	-	-	1,0

Each column represents an iteration of the algorithm. Each row represents the contents of an address generator after it has seized a token. An 'x' indicates a null token.

Table II : An example of routing tag assignment