Engineering Issues, Arbiters and Allocators

Lecture #11: Monday, May 14th, 2001
Lecturer: Professor Bill Dally
Scribes: David Wang and Milind Kopikare
Reviewer: Kelly Shaw

Engineering Issues

The two core design issues discussed in lecture were buffers and switches. The tradeoffs involved in the choice of implementation for each have a significant affect upon router latency, throughput, and cost.

1 Buffers

Buffers are required to store data elements (flits or packets) when a needed resource(s) is not available. Common scenarios include:

- *Blocked Output Port*: If the packet's desired output port is blocked, the packet needs to be buffered.
- *Output Port Contention*: If multiple incoming packets from different ports wish to exit the same output port, the loser of the allocation will need to be buffered.
- *Routing Contention*: If a structural hazard for the routing unit occurs between multiple arriving data elements, the losers will need to be buffered.

Buffering can be partitioned along a centralized or distributed approach.

1.1 Centralized Buffering

In the case of centralized buffering, a single buffer services all the inputs and outputs.

- *Advantages:* This central buffer serves as a buffer as well as a switch, allowing the buffer to share the capacity across all the inputs ports.
- *Disadvantages:* It is very hard to build a single shared memory with the amount of bandwidth needed to support this. When the granularity of the data elements becomes very small, it is difficult to support the large number of reads and writes required for high throughput.

1.2 Distributed Buffering

Distributed buffering essentially dedicates separate buffers for each channel. Buffering can be analyzed from the perspective of buffer placement (switch vs. inputs) and buffer structure (static vs. dynamic).

1.2.1 Buffer Placement

The two possible locations to distribute the buffers are at the inputs or within the switches themselves as shown in figure 1.

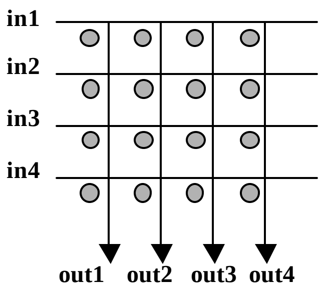


Figure 1: Buffers within the Crosspoints of a Switch

Switch Placement

Given a crossbar in figure 1, buffers are placed at each cross-point. For example, if in1 is routed to out1, in1 can store the data elements into the buffer located at the cross-point between in1 and out1. This structure is equivalent to an output queued switch which can connect any input to any output without interference. This non-interference comes at a high cost in terms of buffer count. As the number of inputs and outputs grow, the number of intermediate buffers needed grows on the order of $O(N^2)$.

Input Placement

A more cost effective solution than distributing the buffers within the switch would be to place the buffers for each channel at the input. This implementation choice results in potential blocking at the inputs.

1.2.2 Buffer Allocation

The allocation of the buffers may either be static or dynamic. In a statically allocated buffer, the location and quantity of memory dedicated to each channel is fixed, while in a dynamically allocated implementation, the locations and quantity of memory dedicated to each channel may vary.

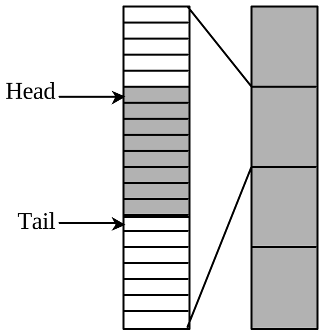


Figure 2: Statically Allocated Buffers

Statically Allocated Buffers

An example of a RAM with four statically allocated first in first out (FIFO) buffers is illustrated in figure 2. Each buffer occupies ¼ of the memory, with a head and a tail pointer. The head points to the next element in the buffer to leave the buffer, and the tail points to where the next element should enter the buffer. The head and tail wrap around when they reach the boundary of the buffer. This is known as a circular queue. When the head catches up to the tail, the buffer is empty, and when the tail catches up to the head, the buffer is full. This works particularly well if the length of the buffer is a power of two, since the MSBs of the RAM addresses are the address of the first word and the Head and Tail represent the LSBs.

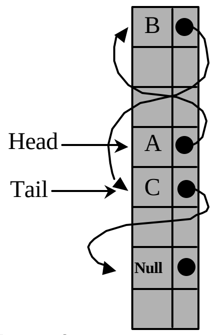


Figure 3: Linked List for a Dynamically Allocated Buffer

Dynamically Allocated Buffers

An example of a RAM with a dynamically allocated buffer is illustrated in figure 3. Each buffer has some fixed portion of the memory allocated to it, and the rest of the memory is shared. The most common implementation of this scheme is with a linked list structure. Each location in memory has a pointer associated with it which points to the next element in the buffer. Due to their small relative size (1-2bytes), the pointers do not introduce significant overhead.

Like the statically allocated structure, the dynamically allocated buffers utilize head and tail pointers which preserve the FIFO nature of the buffer. There is also a free pointer which points to a linked list of free memory elements in the RAM. Note that each logical queue in memory will have its own head, tail, and free pointer associated with it. In addition, since these pointers are stored in external registers, they can each be updated in a single clock cycle.

To support the addition of an element into the buffer in a single clock cycle, the pointer and element memory can be split up into two different RAMs. This way, both the pointer and data can be updated to the appropriate values simultaneously. Another approach would be to modify the tail pointer, so it always points to the next free element which contains NULL, illustrated in figure 3. Thus, to add element D, data would be added to the Tail pointer entry, and both the tail and free pointer would be updated, requiring only a single memory write.

2 Switches

Two methods were discussed in switch implementation:

- Bus
- Crossbar

These two implementations represent degenerate cases in a continuum of possibilities, as discussed later in section 2.3.

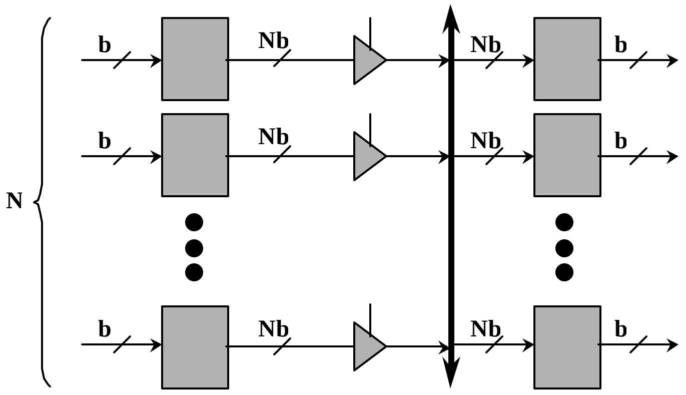


Figure 4: Bus Switch Datapath

2.1 Bus Switch

A bus structure is one of the simplest methods to realize a switch. In figure 7, a bus with N inputs, ach with bandwidth b,is shown. To operate at full rate, de-serializing buffers are placed at the inputs with tri-state buffers driving the bus. The de-serializing buffers drive data on the N*b bandwidth bus 1/Nth of the time. On the receiving side, N serializing buffers are placed to read in bursts of N*b, and stream them to the outputs at bandwidth b.

Advantage: Simple allocation. During the cycle that an output gets the bus, it can reach back and read *any* input. The desired allocation is always provided. In other words, the output can read an input even if every other output want to access the same input. This is not possible with a simple crossbar.

Disadvantages: The key disadvantage to a bus implementation is granularity. Imagine the case when the switch allocator decides to allocate different consecutive flits to different output ports. If the switch allocation occurs at the flit level, the bus width cannot exceed the number of bits in a flit. Thus the following relationship must hold:

$$N \times b \leq flit\_length$$

Another disadvantage is the latency which the de-serialization and serialization buffers introduce.

2.2 Crossbar Switch

Unlike the bus approach, a crossbar switch requires allocation since the output buffers cannot independently make choices on which inputs to be connected to. Figure 5.1 exhibits a possible scenario where the blocking of input ports will occur. The inputs have the following output mapping:

Input Port Number	Desired Output Port Connection
1	1 & 2
2	2 & 3
3	3 & 4
4	4 & 4

Figure 5.1: Table Illustrating blocking for a Crossbar

Thus, a poor allocation may assign output ports to the input ports shown in figure 5.2, resulting in input ports in2 and in4 being completely blocked. (This can be seen if the allocator assigns 1→1, 1→2, 3→3, 3→4).

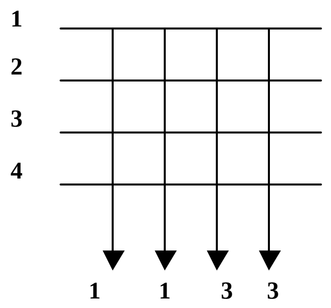


Figure 5.2: Illustration of a Possible Poor Allocation from Figure 5.

2.3 Taking the Middle Road.

A continuum exists between a simple crossbar implementation and the bus example. In figure 6, each input has two input port connections into the crossbar, allowing a tradeoff between how much logic is put into the switch vs. the switch allocator. There is a continuum between the two extremes of the crossbar and bus approaches. For example, instead of having only one path from an input to an output, figure 6 illustrates how with two input points into the crossbar, two outputs 1 and 2 may read from the same input 1. Typically there is a nice middle point which will resolve 99% of the simple allocations without the granularity problems which a bus switch will experience. In addition, observe how increasing the number of input point into the crossbar results in making it easier to get a maximum assignment.

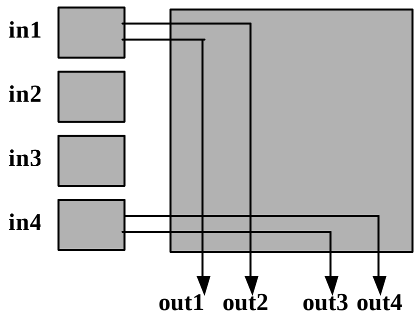


Figure 6: Easing the Allocation task by adding additional input ports

3 Arbitration

One can think of arbitration as a box where at most *N* inputs send requests to the box and there is just one resource sitting at the other end. The problem of arbitration lies in resolving the issue of deciding which requestor gets the resource. In the switching module of a router, all *N* inputs send a request to each of the outputs for whom they have a packet to send. Each output arbitrates among the requests and sends one grant. The grants are one hot and we grants are given to only those inputs who have sent a request.

One metric to measure the performance of an arbiter is its *Fairness*. There are various notions of Fairness. Some of them are:

- **Weak Fairness**: This means that no input that has packets to send will be starved of bandwidth. Intuitively, it means that if an input is patient enough, it will eventually be served. This however is not necessarily desirable, especially in delay sensitive applications where the application's packets need to be sent with a certain delay bound.
- **Strong Fairness**: This implies that there is a rate bound on the number of grants. This notion of fairness is more accurate in the sense that it strives to ensure that all flows end up getting the same amount of bandwidth all the time.
- **Weighted Fairness**: Even strong fairness is not always desirable. There might be some packets that would want to be sent immediately and a few others that could afford to be detained longer. This gives rise to the idea of weighted fairness. Here each flow of packets has a weight associated with it and the rate of service to a flow of packets is in proportion to the weight associated with it.
- **FCFS fairness**: FCFS stands for First Come First Served. It means that if packet *A* arrived before packet *B*, FIFS fairness demands that *A* be served before *B*. If *A* never gets

served, then *B* never gets served either. This, when combined with Weak Fairness, gives us a good sense of Fairness among all the packets. However, it should be noted that even this is not perfect and is in fact also weaker than the notion of Strong Fairness.

- **LULS Fairness**: LULS stands for Last Used Last Served. This means that is an input was served last, it will have last dibs at the resource the next time around.

3.1 Global vs. Local Fairness

Arbitrators only have freedom over deciding on the packets at their inputs and do not have much knowledge of the complete interconnect network as a whole. Thus, while a packet might get vary fair service at one router, it might have had suffered terribly at a router upstream. One instance of this problem is the famous *Parking Lot Problem*.

3.2 Parking Lot Problem

Imagine you are parked at a sport arena and after the game is over you are trying to get out of the parking lot. Now even if you reached your car before everyone else, and even though courtesy demands that you get out of the parking lot first, the cars parked nearest the exit get to exit before you do. In fact, if your car is parked, say, 8 rows from the exit, at each row, you have to contend with a car for getting ahead. The net result is that you end up coming out of the parking lot way later than the others. Figure 7 below shows how this problem applies to a stream of packets as well which have to pass through a row of routers. Here packets '*c*' and '*d*' suffer the most getting only 1/8th of the bandwidth while packet '*a*' gets ½of the bandwidth. Note that each router in this row is a fair arbitrating router, except that it arbitrates fairly only on the local stream of packets that it sees and does not take a global view of the entire system. One way to get over this problem is to have time stamps on each packet. Thus, if two packets are contending for the output slot at a router, the arbiter would give preference to the arbiter that has "aged" more. This would solve the problem for the figure 7 below with '*c*' and '*d*' packets also getting equal (i.e 1/4th of the bandwidth).

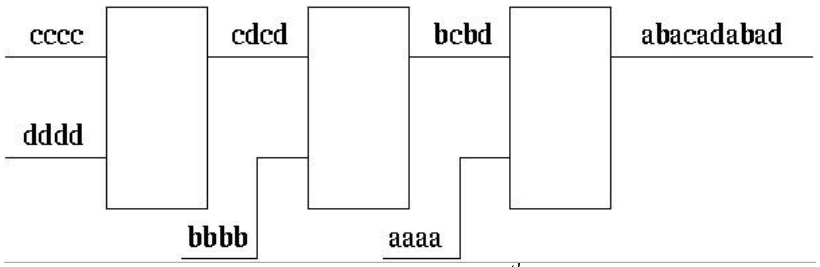


Figure 7: Packets c and d get just 1/8th of the bandwidth.

Another solution towards solving this Parking Lot Problem is to have weights associated with each flow of packets. These weights are stored at each router and depending on the weights, the arbiter gives preference to one flow of packets as compared to the other. This method, although it works well in principle, requires that we maintain state of which flows of packets have what weights and this can grow to be a huge list. Maintaining this list of weights and dynamically updating them at each router is a very time consuming and inefficient process.

3.3 Priority Arbiter:

Getting back to arbitration at a local router, one of the simplest ways to ensure arbitration is the *Priority Arbiter*. This is not really fair in any of the ways we have talked about so far but works fairly well for uniform loads. It's advantage is its simplicity. A hardware block diagram of a priority encoder that is used to implement a priority arbiter is shown in figure 8. It can be viewed like an iterative circuit. A carry-in to an input *j* is 1 if no *input* < *j* has requested the output in that round. If even input *j* does not request for that resource, there is a carry-out to the next requesting input. Notice that with this setup, the carry-out to the next input does not have to depend on the grant given to the previous input. This helps to avoid delay.

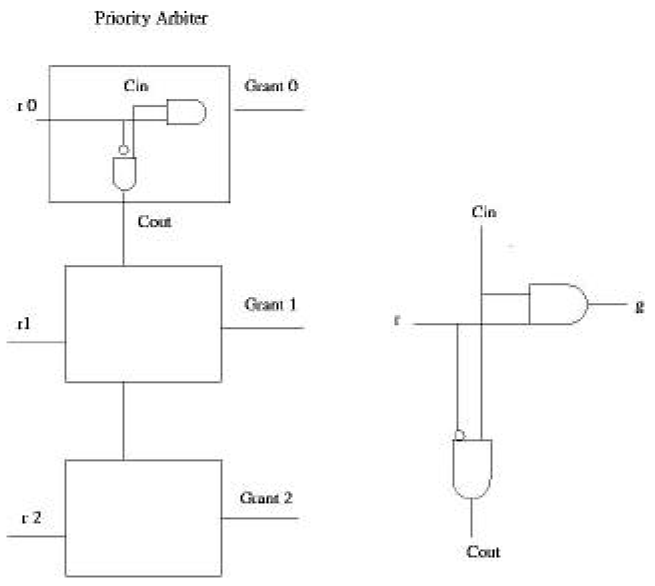


Figure 8: A block Diagram of the Priority Encoder

3.4 Round Robin Arbiter (RRA):

Now it is easy to convert a Fixed Priority Arbiter into a Weakly Fair one. This is what is called as the *Round Robin Arbiter*. Here we wrap the linear stack of input requests around. There exist a Priority Mechanism exhibited in figure 9. This Priority has to be one hot. This priority mechanism can be thought of as a token that gets rotated among all the inputs. Thus when requestor *j* has the priority token and it also has a packet to send, *j* gets the grant and all other requestors have a 0 in their Carry-In. After the first arbitration, requestor (*j*+1) gets the token for priority and the cycle continues. The RRA is weakly fair, because every input gets the resource at least every N cycles.

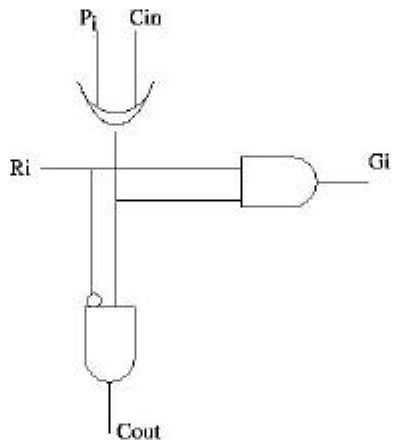


Figure 9: A Logic Diagram of the Round Robin Arbiter.

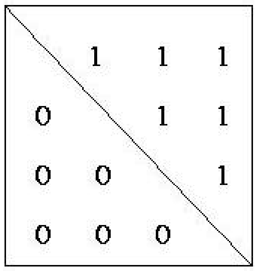
3.5 Weighted RRA:

An extension to RRA is the weighted version of RRA. Here if we want to process Inputs 1's packets twice as often as Input 2's packets we do it in one of two ways.

- We place two request of Input 1 in the Request Stack. In such a case we place the second request somewhere after the first and not immediately below it. This is just to ensure a good distribution of Fairness among all inputs.
- Another way is to have a counter for each input. The counter represents the number of requests the input should be granted over a given period of time. The higher the priority, the higher the original count. Thus Input 1 could have a counter with a value of 256 and Input 2 could have a value of 128. Each time a queue gets served, its counter gets decremented. When a counter becomes zero, its input can no longer be served till all other inputs have exhausted their counters as well. It is clear that by this scheme we ensure that Input 1 gets served twice as much as Input 2 over a large enough time. Note however, that this scheme ensures weighted fairness only over a certain time constant. It is not possible to ensure weighted fairness over a smaller time window..

3.6 Matrix Arbiter:

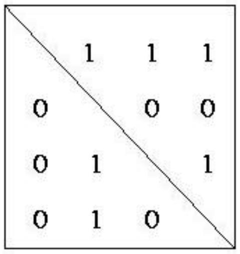
The Matrix Arbiter gets over the problem of *LULS* fairness. The Matrix Arbiter keeps a complete order of request grants. This information is maintained in a matrix form where each row corresponds to an input and each column corresponds to an output. There are four Requestors 0, 1, 2, 3. A 1 at the a^{th} row and the b^{th} column means requestor a beats requestor b in acquiring the resource. The implementation is extremely simple with it being possible to store each element in a flip flop. The Matrix Arbiter logic for arbitration can be better understood by figure 10 below.



Priority Sequence = {0, 1, 2, 3}

Figure 10: Initial Priority Sequence of the Matrix Arbiter.

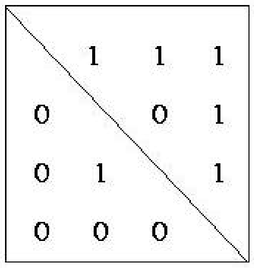
In this case we only need 6 flip-flops. The relative priorities among the inputs fighting for the sole resource are {0,1,2,3}. Now lets say input one gets processed. The new Matrix will now look like figure 11 below:



Priority Sequence = {0, 2, 3, 1}

Figure 11: The Matrix Arbiter after Input 1 gets processed.

Notice that all the elements in column 1 become One's (means that all inputs now beat Input 1) and all elements at Row 1 become zero (which means that Input 1 cannot beat any other input). Thus the new priority list is {0,2,3,1}. If now we process Input 3, the new matrix will look like the figure below, resulting in a new priority list of {0,2,1,3}.



Priority Sequence {0, 2, 1, 3}

Figure 12: The Matrix Arbiter after input 1 and input 3 get processed.

Notice that Matrix Arbiter emulates the LULS notion of fairness. The reason this is able to do that and RRA is not is because we are now keeping additional state information and hence know how much in the past a particular input was served. Also another interesting thing to note is that we only need to maintain the triangular portion of the Matrix, which effectively means that we need to store only half the bits. This is a good improvement when we consider large switches with a large number of inputs.

3.7 FCFS Queuing Arbiter:

Another possible notion of fairness is to serve in a FCFS manner as explained earlier. Notice that we can reuse the design of a Matrix Arbiter for ensuring FCFS service as well. Basically, the Matrix Arbiter is a Queue structure with inputs that were processed most recently pushed back to the very end of the queue. The FCFS arbiter is similar in that it picks the input that had a packet for that output and which arrived to the queue the earliest. An analogy to this is that of going to a bakery. You get a coupon with a number marked on it. You then get served when your number gets called. Similarly each packet will have a time stamp associated with it on arrival and the arbitrator will pick the input with the largest time stamp (as it has been the longest in the queue). The advantage here is that the arbitrators do not need to maintain any state information about any of the input queues. This type of arbitrator also ensures global fairness, a problem that was discussed when we discussed the Parking Lot Problem. Figure 13 below shows a FIFS (Queuing) Arbiter.

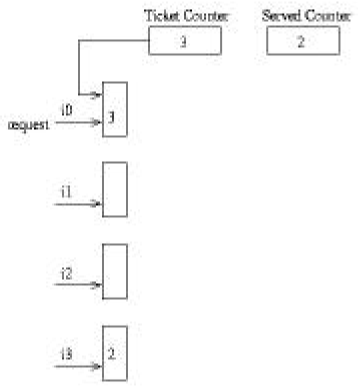


Figure 13: A FCFS (Queuing) Arbiter

3.8 Priority (Weighted) Fairness:

In addition to the two ways of ensuring a weighted form of Fairness that have been discussed earlier, another obvious and probably more simpler mechanism is to have different classes of priority. For each priority class, you will have an arbiter. The arbiters are then connected in a sequence. If a High Priority (HP) arbiter grants a request, the Low Priority (LP) arbiters are inhibited. Otherwise, LP arbiters may grant a request. Thus if are using RRA as the arbitration scheme, we simply have a High Priority RRA, and if none of the inputs belonging to the HP RRA request a grant, only then does control shift to the LP RRA. Note however that this scheme could potentially result in permanent starvation of the inputs belonging to the low priority stage. In other words, the LP stage inputs get the scraps after the HP inputs are done with the main course. One way to avoid this would be to ensure that every once in a while, the LP inputs get to steal one cycle from the HP inputs.

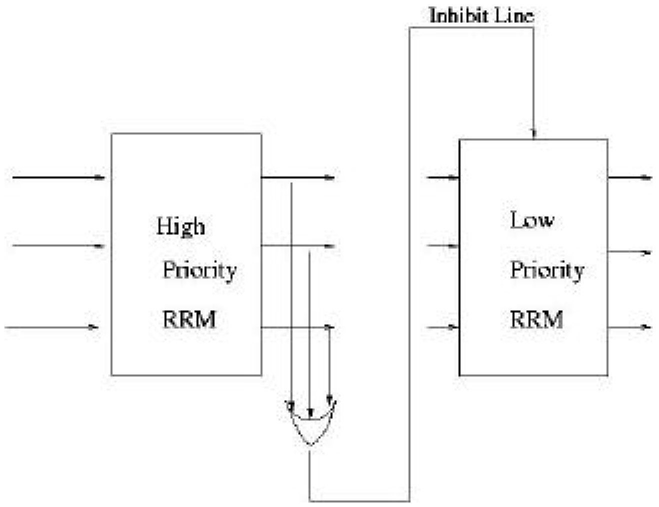


Figure 14: A sequence of HP and LP arbiters.

4 Allocators

An allocator is a box that takes in bids from *N* inputs and picks at most one input to connect to each of *N* outputs as depicted below in figure 15. The general problem of allocation is an instance of bipartite matching problem illustrated in figure 16. You have a bunch of inputs and outputs and you need to determine how to match the inputs to the outputs. One can view the problem as a matrix where rows are the inputs and columns denote the outputs and a number in the *(i,j)*th entry in the Matrix means that input *i* has a request for output *j*. Now output *j* can chose to grant to only one of the inputs in it's column. Also, among all the grants that arrive at an input, an input can choose only one among them. Thus finally the matrix, created from our request example looks as shown in the figure 17 with each row and each column having not more than one entry circled (matched).

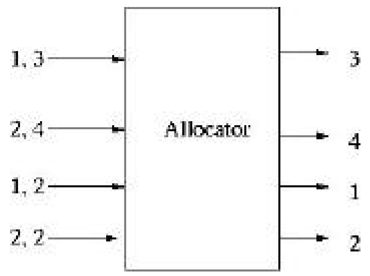


Figure 15: An Allocator

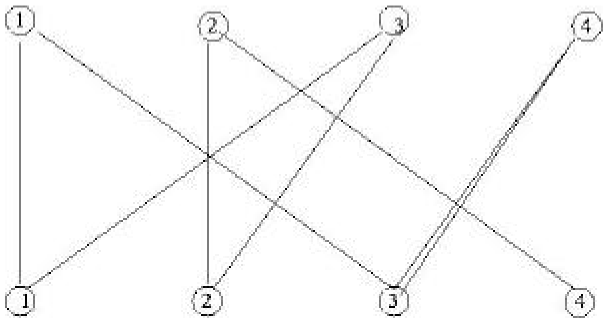


Figure 16: A Bipartite Matching of inputs to the outputs.

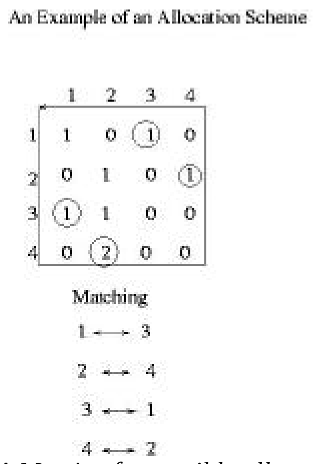


Figure 17: A Matrix of a possible allocation strategy.

Figure 17 shown above is actually a very good allocation strategy as each input has an output matched to it. But this may not always be the case. Consider for example the case of a greedy algorithm that goes through each input sequentially and matches the first available output for whom it has a request. As shown in the figure 18, only Input 1 and Input 2 are matched and the throughput is only half of what it could have been.

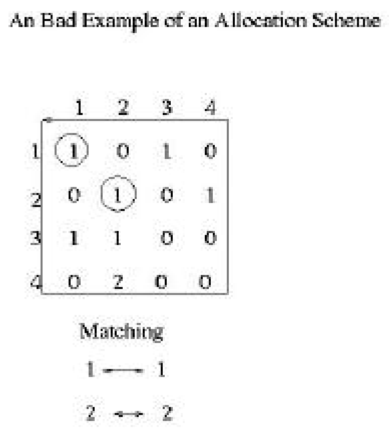


Figure 18: A bad example of an allocation strategy.

An issue of allocation is how to place your earlier assignments so that they don't block your later assignments. One way of getting over this problem is to *overprovision* a switch. For example, if each input has two wires extending from it, we could potentially connect each input to 2 outputs. Thus the new Matrix even for a greedy algorithm would look as shown. From the figure 19, we notice that all 4 outputs have a packet destined for them and hence it has 100% throughput.

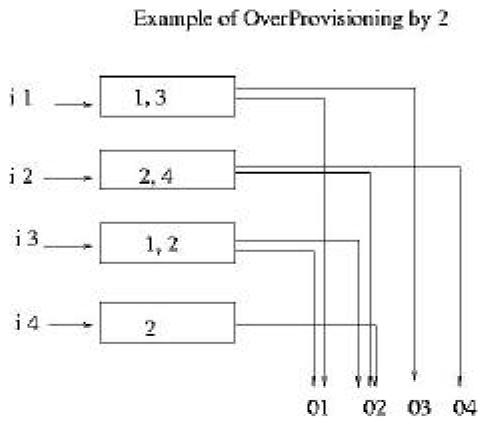
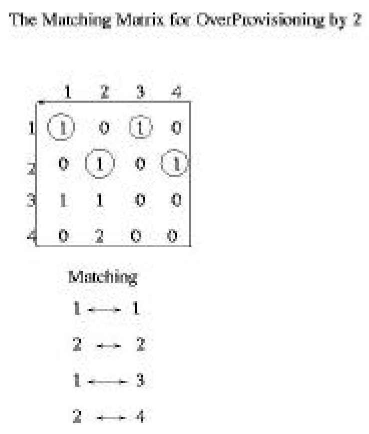


Figure 19



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & \textcircled{1} & 0 & \textcircled{1} & 0 \\ 2 & 0 & \textcircled{1} & 0 & \textcircled{1} \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 2 & 0 & 0 \end{array}$$

$1 \longleftrightarrow 1$
$2 \longleftrightarrow 2$
$1 \longleftrightarrow 3$
$2 \longleftrightarrow 4$

Figure 20

Getting a perfect matching is very time consuming and hard and involves tricks like backtracking even after a match has been made etc. This is very complicated. A compromise to the complexity and the performance is the class of algorithms called *Separable Allocation.*

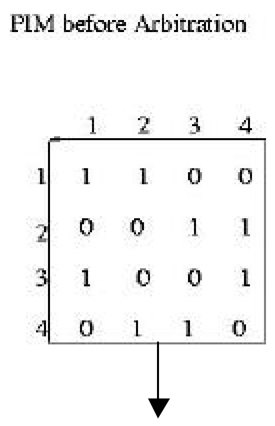
4.1 Separable Allocation

This involves separating the input stage and the output stage of allocation. If say we decide that the output stage gets a chance to decide first, each output looks at all input requests that arrive at it and decides which input to grant by picking a winner based on some policy (There is a winner chosen in each column). These grants are then carried over to the input stage and it is now the inputs' turn to decide on which outputs' request to accept (There is a winner chosen in each row). Note that an input could potentially get a grant from more than one output.

It is possible that there are a pair of inputs and outputs which have been unmatched after the end of this process. In such a case, we simply repeat the whole process again. It has been proven that in the worst case we would have to do at most *N* such iterations.

4.2 **PIM** (parallel iterative matching)

PIM is one such Separable Allocator. We do output arbitration first followed by input arbitration. Here each Output (Input) picks a winner among the possible requests (grants) in a random fashion. Multiple iterations of this process can improve the number of grants. Figure 21 below shows how PIM actually gets better in each iteration. In first iteration, only Input 1 and Input 2 get matched. In the second iteration , even Input 3 and Input 4 get matched.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 1 & 0 & 0 & 1 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$

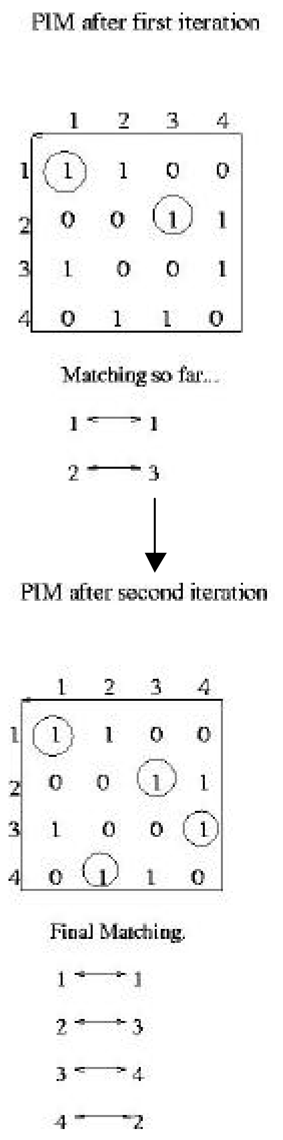


Figure 21: Illustration of Iterative Steps in PIM

4.3 SLIP

This is an improvement over the PIM allocator in the sense that unlike PIM, the outputs and inputs do not pick the winners randomly, but rather in a deterministic round robin fashion. Additionally, these Round Robin Arbiters move the priority a step ahead in their circular stack only if in that cycle, the winner it picked was actually able to get matched. SLIP performs very well as compared to PIM if the matrix is fairly full and we get fairly high throughput on the very first iteration.

However, even SLIP could end up making a bad match and hence result in a low throughput. In the next lecture, we will talk about a Heuristic Allocation scheme that tries to get over the limitations imposed by SLIP and PIM.