韓國 軍事運營分析 學會誌
第 13 卷 第 2 號, 1987.12.

A Pipelined Architecture for Maze Routing

Youn Ju Won *
Sartaj K, Sahni**

Abstract

This paper presents a hardware accelerator for the maze routing problem. This accelerator consists of three 3 stage pipelines. Banked memory is used to avoid memory read/write conflicts and obtain maximum efficiency.

1. Introduction

Hardware accelerator for Lee's maze routing algorithm have been studied earlier in [BLAN81], [MUDG82], [MUDG82], and [NAIR82]. A cellular mesh connected array of processors is proposed in [BLAN81]. [IOSU86] recently improved the modular architecture of the iterative array of processors. [MUDG82] proposes a pipeline of processors and a window scan algorithm. A mesh connected microcomputer system is proposed in [NAIR82]. In this paper, we reexamine the problem of developing a suitable hardware accelerator for Lee's maze route.

2.· Lee's Router

Despite its many short comings, Lee's router and its iimprovement versions play an important role in routing. Typically a Lee type router is used in the final stages of routing when other router

* Korea Military Academy
** University of Minnesota

have failed to complete the route. Lee's router performs a breadth first search from one wire end point (called the source) to the other (called the target). It not only guarantees to find a wire path between source and target (whenever such a path exists), it guarantees to find a shortest such path.

Figure 2.1(a) shows a possible initial configuration. S and T denote the source and the target cells, respectively. The hatched cells indicate cells with obstructions. Lee's router finds a shortest path from S to T by first labeling all cells one unit from S, then all that are 2 units from S are labeled, then those that are 3 units from S are labeled, and so on. This labeling continues until the target cell T is reached. Cells with obstructions are not labeled. This labeling phase is called frontwave expansion. Figure 2.1(b) shows the effect of the front wave expansion phase on the example of the Figure 2.1(a). We use four lables: → , ← , ↓ , and ↑. These point to the cell from which we reached the current cell. Thus all cells adjacent to S have a label that points to S.

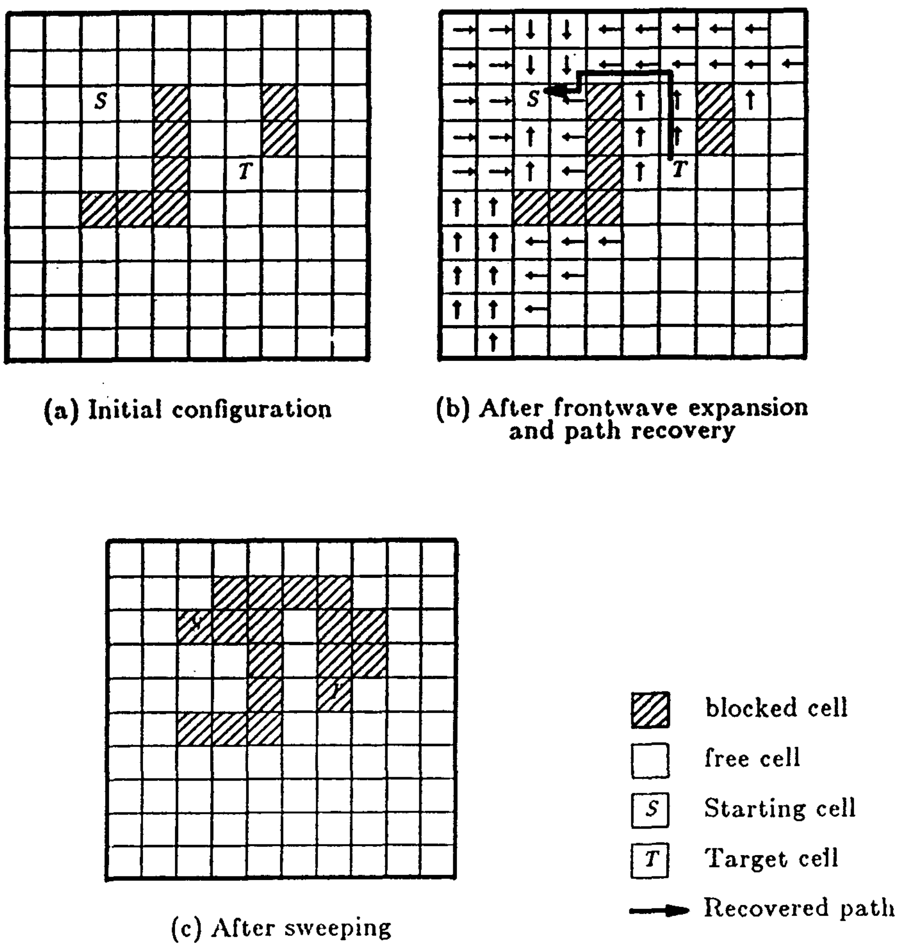


Figure 2.1 Maze routing phases

If the front wave expansion reaches the target cell T, then the path recovery phase is entered. This is a traceback from T to S and is done by simply following the arrow labels back from T to S (see Figure 2.1(b)). Now the wire path has been identified. Before the next wire can be routed, this wire path must be blocked and all arrow labels cleared from the grid. This is done in the sweeping phase. This sweeping phase is similar to the front wave expansion phase. Figure 2.1(c) shows the configuration after sweeping.

The complexity of Lee's router is due to the front wave expansion and sweeping phases. Since these are quite similar, we explicitly discuss the former only.

3. Strategies

The general approach for finding a suitable special purpose parallel architecture is to start with finding the source of parallelism from the nature of the given algorithm. A very straightforward idea is to have one processor for each grid cell and expand all cells of the front wave at the same time. However, this requires an impractically large number of processors. Thus we have to find another way to parallize the algorithm. There are three significant source of parallelism in Lee's algorithm that are directly applied to our parallel architecture.

[PARALLELISM 1]

In the front wave expansion phase of Lee's algorithm, the four neighbor cells of each front wave cell need to be examined. Examining each cell consists of the following steps; fetch cell information from the memory first; some computational work is necessary to see if it is blocked, target or free cell; finally the next front wave just generated, if any, is to be stored in the front wave queue. Thus, a popular type of parallelism, called pipelining, can be used by simply assigning these functions separately into three stages as in the Figure 3.1. This gives a potential of 3X speed up over a single processor implementation.

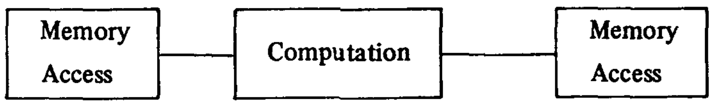


Figure 3.1 Pipelining

[PARALLELISM 2]

Observing that (up to) four neighbor cells of each front wave cell examined, it is desirable to examine these in parallel. To do so, we must be able to handle four neighbor cells at the same time with four different processors. Memory partitioning, as will be described later, is used to make parallel memory fetching possible. This gives 4X speed up.

[PARALLELISM 3]

The concept of front waves gives the third source of parallelism. Front waves are maintained as a set of cells which are the same distance from the source cell. Consider the grid plane masked with chess shape as in Figure 3.2. Given any starting cell s which is colored white(black), all front waves of distance $2i$, $i > 0$ belong to the white(black) region and all front waves of distance $2i - 1$, belong to the black(white) region Since front waves are expanded by distance 1, and two consecutive front waves are in different regions, we can separate the grid plane into two parts and two processors can be built in such a way that processor A handles only white cells and Processor B handles only black cells (see Figure 3.3). This has a potential of 2X speed up.

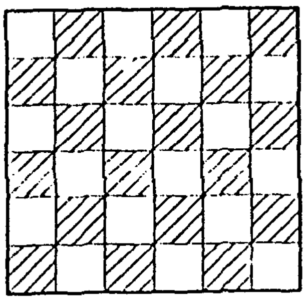


Figure 3.2 Divide into 2 regions

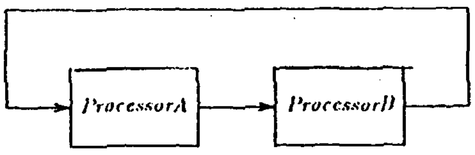


Figure 3.3 Pipeline of 2 processors

Combining the above sources of parallelism together yields potential sources of speed up of up to 32X. Section 4 exploits the first two sources of parallelism and the third source is considered in a later section. Our primary design is comprised of three 3 stage pipelines and a banked memory. The banked memory permits read/write to occur with no wait and no conflicts. Finally we discuss extensions to global routing and to more complex cost functions.

4. Pipelined Architecture

4.1. System Overview

A very high level block diagram of our maze router is given in Figure 4.1. There are three significant components in our system: Banked Queue Memory (BQM), Banked Cell Memory (BCM), and Pipelined Processor (PP). The BCM is used to store the routing grid, the BQM stores cells that have to be expanded in the front wave expansion phase, and the PP does the actual cell expansion. The BQM operates as a FIFO queue. This queue is initialized to contain the unblocked neighbors of the cell S (there may be up to 4 such cells). Each of these cells has their direction label set to point to S. Following this initialization, the maze router of Figure 4.1 takes over and performs the front wave expansion phase of Lee's router. A cell is extracted from the BQM and fed to the PP (pipelined processor). This box consists of three 3-stage pipelines (Figure 4.2).

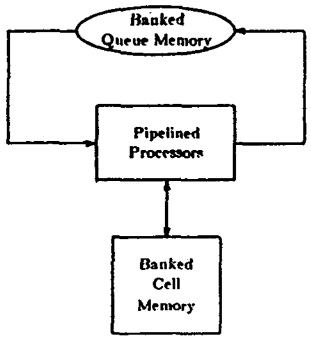


Figure 4.1 Maze router

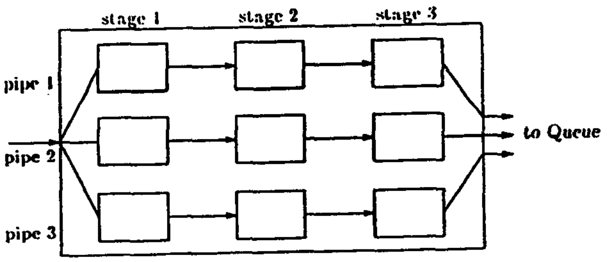


Figure 4.2 Pipelined processors

The cell extracted from the BQM is broadcast to stage 1 of the three pipelines. Each pipeline is assigned the task of examining one of the neighbors of the incoming cell. Note that an incoming cell has only three neighbors to be examined (the fourth neighbor is on the path from S to the incoming cell – it is the predecessor of this cell – and need not be examined). Each pipeline autonomously determines which one of incoming cell's neighbors it should examine and fetches it's neighbor from the BCM. Because of the way this memory is banked, it is possible to fetch all 3 neighbors in a single cycle. Neighbor fetching and some bookkeeping is done in stage 1 of each pipeline. Stage 2 determines where in the queue this neighbor is to be added. Note that in each cycle, up to three (and as few as zero) neighbors may need to be added to the queue. In stage 3, the neighbors to be added to the queue are properly labeled (i.e., labels point to their predecessor cell) and their addresses are added to the queue. The queue is banked so as to allow up to three writes to occur simultaneously.

Front wave expansion continues until either the target cell is reached or on four consecutive cycles, no cell has been fed from the BQM to the PP. If the latter occurs, the BQM and PP are both empty and there is no path from source to target.

4.2. Banked Cell Memory (BCM)

The BCM is used to store the routing grid. Each grid cell requires 5 bits of the BCM. At the top level the BCM is partitioned into two blocks: BCMA and BCMB. BCMA consists of 3 bit cells while BCMB consists of 2 bit cells. Each cell of the routing grid is represented by a 3 bit cell in BCMA and a 2 bit cell in BCMB. BCMA cells are accessible only to stage 1 processors and BCMB cells to stage 3 processors. So, read/write performed by stage 1 processors do not conflict with read/write performed by stage 3 processors. The usage of a cell in each of the banks BCMA and BCMB is described in Figure 4.3. The number inside a field is the number of bits used by that field.

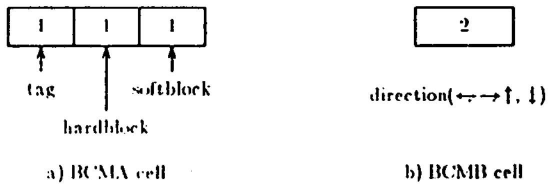


Figure 4.3 BCM cell

Each of the two banks BCMA and BCMB is itself composed of four banks of memory. Figure 4.4 shows an 8X8 grid mapped into 4 banks. With this banking scheme, no grid has two of its neighbors in the same memory bank.

Consider a routing grid with 2^n, $n > 1$, columns. Assume that the rows and columns are numbered beginning at 0. Cell (i, j) of the grid is represnted in bank j mod 4 if i mod 4 $\in \{0,3\}$ and in bank (j+2) mod 4 if i mod 4 $\in \{1,2\}$. Within a bank, cell (i,j) is in row i and column $\lfloor j/4 \rfloor$.

It is more convenient to view each memory bank as one dimensional. This is done using the standard row major interpretation of a two dimensional array. If the position in this one dimensional memory are numbered from zero, grid cell (i,j) is represented in address $i2^{n-2}+\lfloor j/4 \rfloor$ of the appropriate bank. This formula needs to be used only to obtain the addresses of the source and target cells. The bank and bank address of a cell adjacent to another is more easily obtained by using a table lookup scheme. For this, we need to define the following for each grid cell (i,j):

PowParity = 0 if i is even
1 otherwise
ColumnParity = j mod 4

If we know the RowParity, ColumnParity, bank and address of a cell (i,j) then these quantities for its south, north, east, and west neighbors may be obtained using the table of Figure 4.5. Stage 1 of each of the three pipelines in PP needs access to these tables. However, no two stage 1's access the same table simultaneously. So, there are no access conflicts.

4.3. Pipelined Processors (PP)

Each cell in the banked queue memory (BQM) has the configuration shown in Figure 4.6. The numbers within a field give the number of bits needed for that field. The direction to the predecessor cell is encoded as arrows.

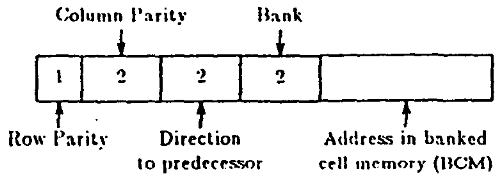


Figure 4.6 Cell description in Queue

There is a queue read processor associated with the BQM. On each cycle of the hardware router, this processor attempts to extract the front element from the queue in the BQM. The detailed functioning of this read processor is described later. If the queue remaines empty for four consecutive cycles, the queue read processor terminates the search for a path from source to target. In this case, there is no path from source to target. If the queue is not empty, then the front element is extracted and broadcast to the stage 1 processors of each of the three pipelined processors. If the queue is empty, then a special element ϕ is broadcast. The detailed working of each stage of the pipelined processors is provided below.

4.3.1. Stage 1

In each cycle, the stage 1 processor of the ith pipeline, executes the program fragment given in Figure 4.7. If the Current Cell descriptor received is ϕ, then the queue was empty and path expansion to a neighbor cell cannot be done on this cycle. In this case, the next cell to expand to, Next Cell, is

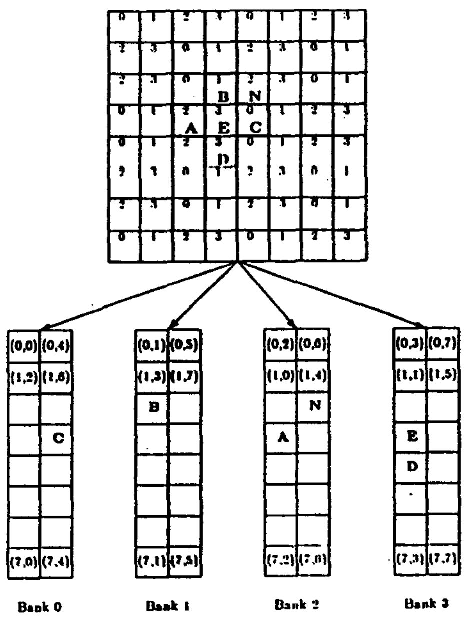
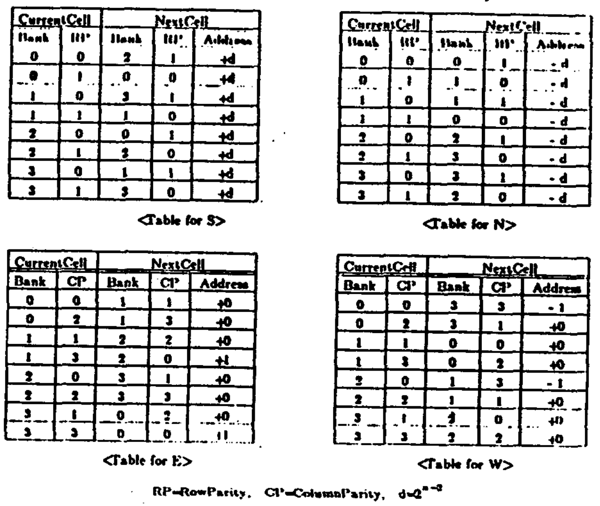


Figure 4 .4 Memory partitioning



CurrentCell		NextCell		
Bank	RP	Bank	RP	Address
0	0	2	1	+d
0	1	0	0	+d
1	0	3	1	+d
1	1	1	0	+d
2	0	0	1	+d
2	1	2	0	+d
3	0	1	1	+d
3	1	3	0	+d

<Table for S>

CurrentCell		NextCell		
Bank	RP	Bank	RP	Address
0	0	0	1	- d
0	1	1	0	- d
1	0	1	1	- d
1	1	0	0	- d
2	0	2	1	- d
2	1	3	0	- d
3	0	3	1	- d
3	1	2	0	- d

<Table for N>

CurrentCell		NextCell		
Bank	CP	Bank	CP	Address
0	0	1	1	+0
0	2	1	3	+0
1	1	2	2	+0
1	3	2	0	+1
2	0	3	1	+0
2	2	3	3	+0
3	1	0	2	+0
3	3	0	0	+1

<Table for E>

CurrentCell		NextCell		
Bank	CP	Bank	CP	Address
0	0	3	3	- 1
0	2	3	1	+0
1	1	0	0	+0
1	3	0	2	+0
2	0	1	3	- 1
2	2	1	1	+0
3	1	2	0	+0
3	3	2	2	+0

<Table for W>

RP=RowParity, CP=ColumnParity, $d=2^{n-2}$

Figure 4.5 Neighbor address tables

set to ϕ. If Current Cell $<>\phi$, then three neighbor cells are located in the directions ($\leftarrow, \rightarrow, \downarrow, \uparrow$) – <direction to predecessor cell>. The computation of line 4 enables each of the 3 stage 1 processor to pick one of these three directions without consulting the other stage 1 processor. No two stage 1 processor pick the same direction.

Each of stage 1 processor accesses one of the tables of Figure 4.5. Since no two have the same value for dir, there is no table access conflict. Using the Row Parity, Column Parity and bank information of the Current Cell, this information for the appropriate neighbor cell is obtained from the table. Further, the address of the neighbor cell is readily computed using information from the table and the address of the current cell. The address and bank of the neighbor cell is used in line 6 to access the three bits (tag, hardblock, softblock) associated with the neighbor cell. After a softblock bit is read, it is set to 1 (regardless of its initial value). This prevents a subsequent reexamination of the neighbor cell.

```
1. Receive CurrentCell descriptor from queue read processor;
2. if CurrentCell $\neq \phi$
3. then begin
4.     dir :=(direction to predecessor + i) mode 4;
5.     Use the CurrentCell descriptor and the table from Figure 4.3 for the direction dir to
       obtain the descriptor for the neighbor cell in the direction dir;
6.     Fetch the 3 bits for this neighbor cell from the banked memory BCMA and update
       the softblock bit of this memory cell to 1;
7.     If the hardblock bit is 1 or the softblock bit was already 1 before the update of line 6,
           then NextCell = $\phi$
           else NextCell = neighbor cell
       endif
   end
8. else NextCell = $\phi$;
9. Transmit NextCell descriptor, dirand direction to predecessor of CurrentCell to stage 2;
```

Figure 4.7 **Stage 1 of *i*'th pipeline, $i \in 1,2,3$**

```
1. Receiver NextCell descriptor, dir, and direction to predecessor of CurrentCell from
   stage 1 processor;
2. if NextCell = target cell then terminate front wave expansion;
3. if NextCell $\neq \phi$ then flag(dir): =1;
4. Synchronize;
5. Use flag(        ), dir, direction to predecessor of CurrentCell and the table of Figure 5.4
   to obtain Priority and Num;
6. Positon : = Next + Priority; Bank: = Position mod 3;
7. if processor = 1 then Next: = Next + Num;
8. flag(dir): =0,
9. Transmit Position, bank, dir, and NextCell descriptor to stage 3;
```

Figure 4.8 **Stage 2**

4.3.2 Stage 2

The code executed by each of the stage 2 processors is given in Figure 4.8. After receiving the information transmitted by the stage 1 processors, each stage 2 processor determines if its NextCell address bank corresponds to that of the target cell. If it does, then front wave expansion is terminated. If none of the NextCell descriptors corresponds to the target cell, the stage 2 processors cooperate to determine how many NextCells are not ϕ. Let this number be Num. Clearly, $0 \leqslant \text{Num} \leqslant 3$. To determine the value of Num, each stage 2 processor sets a flag. There are four flags: flag(↑), flag(↓), flag(←), flag(→). At the start of each cycle, each flag is 0. In line 3, at most three of these flags get changed to 1. In line 4, the three stage 2 processors synchronize. Essentially, line 5 cannot be exexuted until all flags have been appropriately set. The four flags form global data that is shared by the processors.

Once the processors have synchronized in line 4, they perform a simultaneous read into the table of Figure 4.9. This table is divided into 4 segments depending on the direction to the predecessor of CurrentCell. Only the segment for the case when this direction is ↓ = south is shown. Each stage 2 processor needs to read the value of Num and Priority(dir) from the row corresponding to flag(↑), flag(↓), flag(←), flag(→). This can be accomplished by providing a three way fan out for the read or by replicating this table.

	flag				Priority				
Pre dir	→	↓	←	↑	→	↓	←	↑	Num
S	0	-	0	0	1	-	2	0	0
	0	-	0	1	1	-	2	0	1
	0	-	1	0	2	-	0	1	1
	0	-	1	1	2	-	1	0	2
	1	-	0	0	0	-	1	2	1
	1	-	0	1	1	-	2	0	2
	1	-	1	0	0	-	1	2	2
	1	-	1	1	1	-	2	0	3
N		..	..			...	...		..
E		..	..			...	...		..
W		...	..			...	...		..

Fig 4.9 Priority table for stage 2

As can be seen, Num is simply the number of flags that are 1. Priorities for front wave expansion are assigned in such a way that the routed wires will have few jogs and will nest properly. These are two attributes of good wire layout. Traditionally, one attempts to minimize jogs and get good routing during the path recovery phase. This is accomplished by not making a jog unless necessary and when a jog is necessary, it is made by assigning a predetermined priority order for the jogs. In our scheme, we attempt to obtain good routes by monitoring the front wave expansion. Priority is given to continue expansion in the direction of the current wire path.

```
1. Receive Position, Bank, dir, and NextCell descriptor;
2. if $NextCell = \phi$ then
3. begin
4. Let direction field of NextCell descriptor to $dir \oplus 1$ (i.e., complement bit 0 of dir);
5. Write NextCell descriptor into position Position and bank Bank of BQM;
6. Set direction field of NextCell in BCMB to $dir \oplus 1$;
7. end;
```

Figure 4.10 Stage 3

In line 6, the position and bank of the BQM into which the NextCell descriptor is to be queued is computed. Next gives the next available space in this memory. This is updated in line 7 by the stage 2 processor of pipeline 1 only. Hence, there are no write conflicts in line 6), then each stage 2 processor can update its copy of Next. Line 8 resets flag to 0.

4.3.3. Stage 3

Figure 4.10 gives the program segment executed by each of the three stage 3 processors. This is quite self explanatory. Note that by complementing bit 0 of dir (or taking the exclusive or, e, of dir with 1), we get the code of the opposite direction of dir (north and south are opposites, east and west are opposites). There are no memory access conflicts in lines 5 and 6 as each stage 3 processor accesses a diferent memory bank. Also, note that line 4 and 5 can be executed in parallel with line 6.

5. Banked Queue Memory (BQM)

The configuration of each cell of the queue memory is given in Figure 4.6. Since, in any cycle, up to three cells may be written into this memory, this memory is partitioned into three banks: 0, 1 and 2. The partitioning scheme used is straightforward. Cell i is in bank i mod 3. In order to handle and nXn grid, at most 4n queue memory cells are needed. However, one can make it do with a BQM of a fixed size. For this, the BQM is partitioned into four buffers and each buffer is comprised of three banks (cf. Figure 5.1). A disk is used as intermediate storage.

Assume that the four buffers of queue memory are numbered 1, 2, 3, and 4. The maze router starts with a queue configuration which the unblocked neighbors of the source cell (there are up to 4 such neighbors) are in buffer 1. This is the initial read and write buffer (we assume that the buffer capacity is $\gg$4). There are several variables associated with the queue buffers. These are

RB	:	current read buffer
WB	:	current write buffer
NB	:	next buffer to read
NNB	:	buffer to read after next buffer
#Disk	:	number of buffer loads on Disk
Front	:	position in RB of front element of queue
Last	:	position in WB of last element of queue

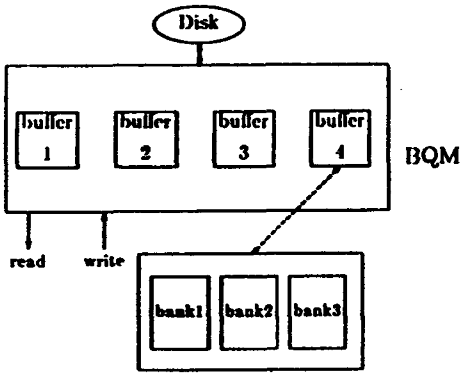


Figure 5.1 Banked Queue Configuration

```
if WB is full (i.e., Buffer Size – Last < 3) then [
   wait for buffer output (if any) to complete
   if there was a buffer whose output has just completed then add this to the free list;
   mutexbegin {mutually exclusive with line 11-17 of Figure 6.2}
      case
         (#disk > 0) or ( # of free buffers = 1):
            Initiate write of WB to disk;
            # disk = # disk + 1;
            WB = a buffer from free buffer list;
         # of free buffers = 2:  {WB ≠ RB}
            WB = a buffer from free buffer list;
            NNB = WB;
      otherwise :  { # of free buffers = 3, WB = RB}
            WB = a buffer from free buffer list;
            NB = WB;
   endcase
mutexend;
Last = 0;]
```

Figure 5.2 Queue update following a write

Initially, RB = WB = 1; NB = NNB = #Disk = 0; Front = 1 and Last is the number of unblocked neighbors of the source cell, $1 \leqslant Last \leqslant 4$. The buffers 2,3, and 4 are initially free. These are kept on a list of free buffers. This list operates as a stack. Let Buffersize denote the number of cells a buffer may hold. The current wirte buffer, WB, is said to be full when Buffersize $-$ Last $\leqslant 3$. In other words, a buffer that is not full can hold at least three additional cells. Hence, whenever the stage three processors write into the BAM, they write into the same buffer. Following the writing of the new cells into the current write buffer, WB, the code of Figure 5.2 is executed by stage 3 of processor 1. This code updates the queue variables as needed.

In case the current write buffer has enough space to accomodate another stage 3 write, then the queue variables need no update. However, when this is not the case, lines 2-18 are executed. In line 2, we need to be sure that the last disk read/write operation has completed. In case this was a write operation, a buffer just written out becomes free and is added to the list of free buffers. The rest of the code (encoded in the mutexbegin and mutexend segment) is to be executed in a different time slice from the code of lines 9-17 of the queue read procedure (Figure 5.3). This is necessary as both codes read/change the same queue variables.

```
 1.  case
 2.     queue is empty:
 3.        if queue has been empty fout times in a row
 4.        then terminate
 5.        else broadcast nil cell;
 6.     RB not empty:
 7.        broadcast Front cell from buffer RB;
 8.        Front: = Front + 1;
 9.     otherwise:
10.        OldRB: = RB;
11.        mutexbegin
12.           if NB ≠ 0 then [RB=NB; NB=NNB; NNB=0]
13.           else [wait for disk read to complete; RB = newly read buffer];
14.           if # disk > 0
15.              then [initiate a read into buffer OldRB; # disk = # disk – 1]
16.              else [put buffer OldRB on free buffer list];
17.        mutexend
18.        broadcast cell 1 from bufer RB;
19.        Front = 2;
20.  endcase;
```

Figure 5.3 Reading from the buffered queue

The procedure to read from the queue is given in Figure 5.3. At the top level, the queue read operation breaks down into three cases. If the queue has been empty for four consecutive cycles of the hardware router, then the three pipeline stages are empty and there is no path to the target cell and front wave expansion is terminated. Otherwise, cells are being processed in the pipeline and a special nil cell ϕ is sent to the stage 1 processors. This cell has the property that all its neighbors are blocked. Hence stages 2 and 3 essentially perform no work when processing these neighbors.

6. System Cycle Time

The system cycle time needs to be the largest of the following times:

1. Queue read time (Figure 5.3)
2. Stage 1 time (Figure 4.6)
3. Stage 2 time (Figure 4.7)
4. Stage 3 time (Figure 4.9 + Figure 5.2)

Because of the mutually exclusive code segments in Figure 5.2 and 5.3, this maximum time could be the sum of the times for these two segments. Note that the two mutually exclusive segments interfere with each other only on those cycles in which both the current read buffer, RB, becomes empty and the current write buffer, WB, becomes full. We can avoid the mutual exclusion by giving priority to the write operation. In other words, if an attempt is made to execute both mutually exclusive code segments in the same cycle, then the write succeeds while the read broadcast a nit cell. The read succeeds on the next cycle.

In order to avoid delay caused by waiting for a disk I/O to complete, it is necessary to choose the buffer size so that the time to fill a buffer is at least as much as the time input one bufer plus that to output one buffer (note that the buffer fill time will generally be less than the buffer empty time).

Also, note that while our discussion of the buffered queue has been in terms of a random access memory and a disk, it applies just as well to the case o expensive high speed RAMs for the buffers and less expensive slow speed RAMs for the intermediate storage and also when the buffers are hardware queues and the disk is replaced a RAM.

7. Path Recovery and Sweeping

If the target cell is reached, then path recovery is done by following the backward arrows from the target cell to the source cell. The tag bit of all cells encountered on this path is set to 1. The seeeping phase is similar to front wave expansion except that the roles of softblock = 1 and softblock = 0 are interchanged. As a result all softblock bits are reset to 0. Further, cells with tag = 1 that are encountered during this sweeping have their tag bit reset to 0 and their hardblock bit set to 1.

8. Performance

We expect our hardware router to provide a significant performance improvement over a conventional implementation of Lee's algorithm on a uniprocessor computer. This is because very little overhead has been introduced in the hardware implementation. Both the hardware router and the uniprocessor implementation of Lee's algorithm perform essentially the same computations (remove a cell from the queue, examine its neighbor, add unblocked neighbors to the queue). We can compare the performance of our system to that of the architectures proposed in [BLAM81], [IOSU86], [NAIR82], and [MUDG82]. The detailed evaluation of their performances is omitted in this apper due to the space limit. However, we can readily see the effectiveness of our system over others in terms of processor utilization.

9. Extensions

As mentioned at the beginning of this chapter, there are three possible sources of parallelism. When the third parallelism is accounted for, the system block diagram takes the form given in Figure 9.1. There are two identical parts in the system each of which is exactly the same as the one described at Section 4. The left half of the system (Pipeline A) is for expanding the white region and the right

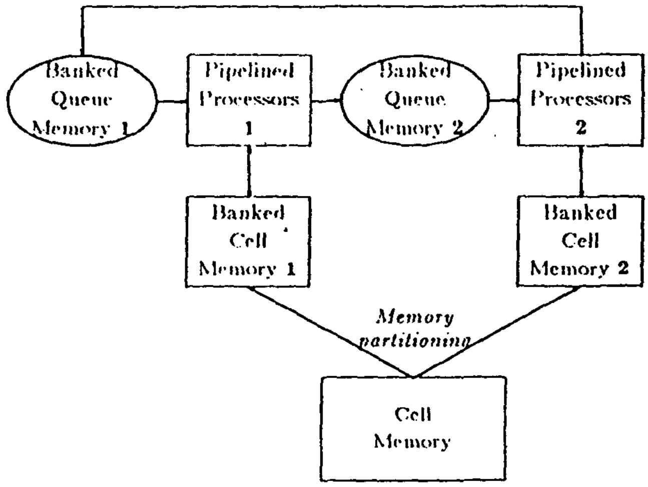


Figure 9.1 Block diagram

half (Pipeline B) is for the black region. The corresponding memory partitioning for Figure 9.1 is given in Figure 9.2. There are eight banks of memory; four of them are assigned to Pipeline A and the other four banks are assigned to Pipeline B.

The other extension is to allow complicated cost functions (not unit distance cost function). This is very useful since in practice paths other than the shortest may be more desirable (e.g., the path with less jogs may be preferred to a shorter path with many jogs). In this case, we need 4 pipelines instead of 3 because certain cells can be expanded more than twice and we have to check all four neighbors even though some of then are known to be expanded already. Basically the global routing has the same nature as the detailed routing with complicated cost functions. Also the muti layer problem can be implemented simply by adding two more pipelines; one is for going up and the other is for going down (i.e., there are 6 pipelines).

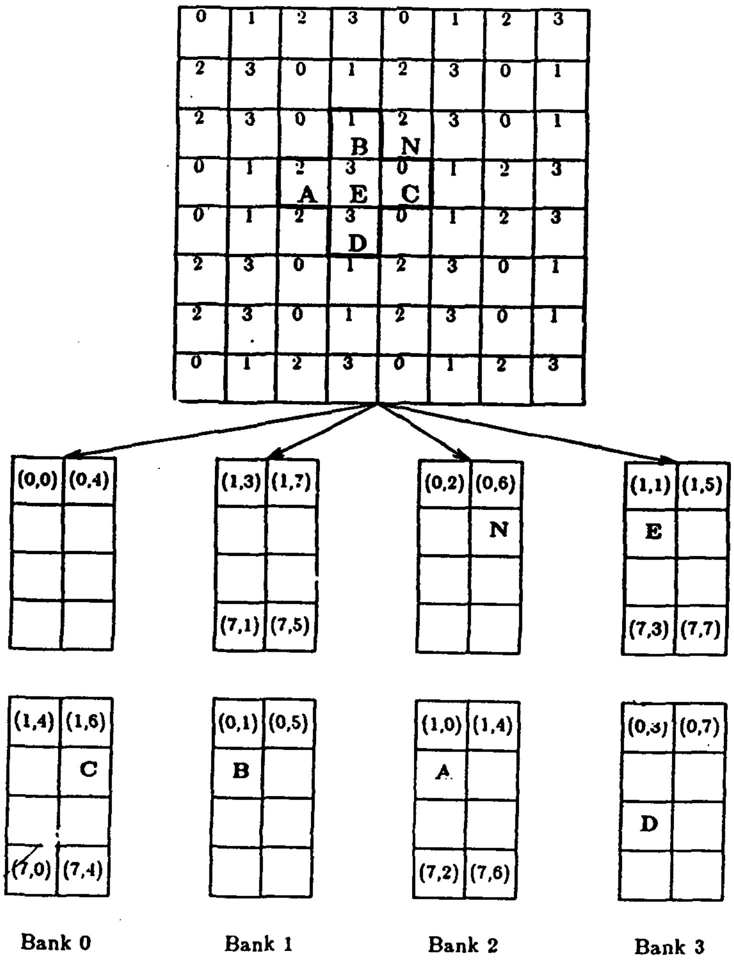


Figure 9.2 Memory partitioning

10. Conclusions

We have designed a hardware accelerator for the maze routing problem that provides good hardware utilization and good speed up. It can be used to perform grid routing on grids that fit into the available memory. Unlike other proposed hardware solutions for this problem, our design does not require an increase in the number of processors as the problem size increases. Also, our design many be extended to handle mutilayer routing by increasing the number of pipelined processors and by extending the memory partitioning scheme.

References

[BLAN81] Tom Blank, Mark Stefik, and William vanCleemput, "A parallel Bit Map Processor Architecture for DA Algorithms", ACM/IEEE 18th DA Conference Proceedings, pp. 837-845.

[BLAN84] Tom Blank, "A survey of hardware accelerators used in computer-aided design,", IEEE Design and Test of Computers, vol. 1, Aug. 1984, pp. 21-39.

[IOSU83] A. Iosupovici, C. King, and M. Breuer, "A module interchange placement machine,", ACM/IEEE 20th DA Conference Proceedings, pp. 171-174.

[IOSU86] A. Iosupovici, "A class of array architecture for Hardware Grid Router", IEEE Trans. on CAD, vol. CAD-5, No. 2, Apr 1986, pp. 245-255.

[LEE 61] C.Y. Lee, "An algorithm for path connections and its applications", IRE Trans. Electronic Computers, vol. EC-10, Sept. 1961, pp. 346-365.

[MUDG82] T.N. Mudge, R.A. Ratenbar, R.M. Lougheed, and D.E. Atkins, "Cellular Image Processing Techniques for VLSI Circuit Layout Validation and Routing", ACM/IEEE 19th DA Conference Proceedings, pp. 537-543.

[NAIR82] R. Nair, S.J. Hong, S. Liles, and R. Villani, "Global wiring on a wire routing machine," ACM/IEEE 19th DA Conference Proceedings, pp. 224-231.