# SYNCHRONOUS AGGREGATE COMMUNICATION ARCHITECTURE FOR MIMD PARALLEL PROCESSING

A Thesis

Submitted to the Faculty

of

Purdue University

by

Timothy I. Mattox

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

August 1997

## ACKNOWLEDGMENTS

I thank Jeff Sponaugle for his partnership in the development of the PAPERS1 prototype. Without his extensive help, PAPERS1 would not have been built, and thus the core of this thesis would be missing. I thank Tariq Muhammad for building the PAPERS0 prototype which inspired the development of PAPERS1. I thank my advisor, Hank, for his input, and for bringing the PAPERS/CARDboard group together. I also thank him for his wonderful ability to come up with and write wacky algorithms that turned simple NAND gates into an almost universal parallel processing communications medium. I thank my EE 559 project partners Alex Andrews, Rebecca Boyd, and David Ward for their amazing strength to follow me into the huge undertaking that became the DBM16 VLSI design project. That project lead to many insights on where this PAPERS research should continue. Lastly, I thank my family for their support and assistance throughout this long process. I especially thank my sister Holly for her insights on writing a Thesis.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# GLOSSARY AND LIST OF ACRONYMS

| | |
|---|---|
| AFAPI | Aggregate Function Application Program Interface |
| API | Application Program Interface |
| ASIC | Application Specific Integrated Circuit |
| bitwise | Computations on $A_i$ and $B_i$ for all *i* in a pair of bit vectors A and B. |
| BPU | Barrier Processing Unit |
| CE22V10 | An industry standard reprogrammable 24 pin PAL chip. |
| coordinality | A term coined in this thesis that combines the meanings of coordination and cardinality, and is a measure of communication complexity. |
| COTS | Commercial Off The Shelf |
| DBM | Dynamic Barrier MIMD |
| DOALL | A parallel loop where each iteration is independent of all the others. |
| FDDI | Fiber Distributed Data Interface, a high bandwidth networking system |
| FLOP | FLoating-Point OPeration |
| GCC | Gnu-C Compiler |
| IPROC | The identification number of this PE (processing element). |
| IXPROC | The X coordinate of this PE on a grid. |
| IYPROC | The Y coordinate of this PE on a grid. |
| MFLOPS | Millions of FLoating-Point Operations Per Second |
| MIMD | Multiple Instruction streams, Multiple Data streams |
| multiformity | A term coined in this thesis that is a measure of the differences in the grains of execution of a parallel program, i.e. how many different forms do the grains take. |
| NAND | A logic and with the result inverted. |
| NPROC | The total number of processing elements in a parallel computer. |
| NXPROC | The number of processing elements in the X dimension of a grid. |
| NYPROC | The number of processing elements in the Y dimension of a grid. |

| | |
|---|---|
| PAL | Programmable Array Logic |
| PAPERS | Purdue's Adapter for Parallel Execution and Rapid Synchronization |
| PAPERS0 | The first PAPERS prototype, built in February 1994. |
| PAPERS1 | The "second" PAPERS prototype, built in the summer of 1994 by the author and Jeff Sponaugle. The design was started as the second PAPERS prototype, but the construction of two other simpler prototypes were completed before PAPERS1. |
| PARBEGIN | The beginning of a parallel section of code where each statement can be executed in an arbitrary order, including in parallel. |
| PAREND | The end of a parallel section of code where each statement can be executed in an arbitrary order, including in parallel. |
| PASM | PArtitionable Simd/Mimd, a prototype parallel supercomputer that can switch between SIMD and MIMD execution modes at an instruction level granularity, and can partition into submachines. |
| PE | Processing Element |
| SBM | Static Barrier MIMD |
| SIMD | Single Instruction stream, Multiple Data streams |
| SMP | Symmetric MultiProcessing |
| SPMD | Single Program, Multiple Data streams |
| TTL_PAPERS | A class of simplified PAPERS prototypes that use globalNAND as the communication primitive, and are usually constructed from simple/inexpensive digital chips. |
| VLIW | Very Long Instruction Word |
| VLSI | Very Large Scale Integration |

# ABSTRACT

Mattox, Timothy I. M.S.E.E., Purdue University, August 1997. Synchronous Aggregate Communication Architecture for MIMD Parallel Processing. Major Professor: Henry G. Dietz.

A multitude of different parallel architectures have been proposed, and each works well for applications with the appropriate types of parallelism. However, to achieve the best possible speedup for a wide range of parallel applications, a parallel computer's hardware must be able to make effective use of most types of parallelism.

This thesis suggests that the most fundamental flaws of MIMD architecture can be corrected by the addition of a simple synchronous aggregate communication system. After reviewing the relationship between some basic architectural characteristics and the types of parallelism that they can support, a small set of synchronous aggregate communication operations are defined. The implementation and performance of the PAPERS1 prototype hardware, a very simple synchronous aggregate communication system for a MIMD cluster of PCs, is discussed in detail.

# 1. INTRODUCTION

Conceptually, there is a universe of problems, or programs, that could benefit from parallel processing. Unfortunately, no single existing architecture is capable of efficiently executing any arbitrary program from that universe.

Parallel processing architectures have most commonly been classified into two categories, SIMD and MIMD [1][2]. To run an arbitrary program efficiently, a machine must have hardware support for the specific characteristics of that program. It has been shown that for some applications SIMD execution is most efficient, while for some other programs MIMD execution is better. Likewise, individual phases of a program may favor one mode or the other; thus a hybrid machine that supports both SIMD and MIMD execution modes will execute programs more efficiently. Yet there are applications which are difficult to run efficiently on any of the current SIMD, MIMD, or hybrid machines.

While testing a hardware add-on designed to support compiler research for MIMD architectures, an astonishing level of performance was achieved. This hardware, known as PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) [3], added fast barrier synchronization to a cluster of PCs. We explored the characteristics of this experimental MIMD machine to discover why it performed so well. Several important properties of both parallel machines and parallel programs were found that greatly affected achievable speedup. Using these empirically derived properties, we then developed an analysis scheme that we could use to roughly predict the achievable speedup of parallel programs on various architectures.

Chapter 2 presents this analysis scheme which helps develop an understanding of what properties of a program have a significant impact on the success of parallelization. The last two sections of Chapter 2 go through a "laundry list" of features and approaches for alleviating the bottlenecks for running parallel programs efficiently. In that laundry list, one concept rises to the surface that most MIMD machines deal with very poorly. This concept is "synchronous aggregate communications" which is discussed in greater detail in Chapter 3. The approaches for improving a parallel processing machine presented in Chapters 2 and 3, are explored with an actual hardware implementation, PAPERS1, that is described in Chapter 4. The results of this implementation are evaluated in Chapter 5.

# 2. MOTIVATION

When viewed in an abstract sense, there are some subtasks in a program that, given enough (and the correct) resources, could be accomplished concurrently. The type of parallelism available may be different at various points of the program. At times, the subtasks may be small but numerous, and at other times there may be only one subtask. In this discussion, a *subtask* is defined as the smallest bit of execution of an instruction sequence on one processing element that only communicates with other subtasks at its beginning and end. The smallest bit of execution is chosen for a subtask so that available parallelism is maximized. As needed, subtasks can be combined to form larger, but fewer, subtasks. Many practical issues prevent any one machine from exploiting all the parallelism that this abstract representation reveals.

How long does it take for the hardware to accommodate a new pattern of parallel subtasks? With today's programming languages, how well can these abstract subtasks be expressed? How good is the compiler at recognizing potential concurrent subtask execution? How should the data for each subtask be distributed? How often, and how much, data must be re-distributed? Each of these are very difficult questions. To make the problem tractable, this thesis will only consider architecture and system level issues, leaving language and compiler issues to be dealt with by others.

Rather than wait for these questions to be answered, many companies have entered (and left) the parallel processing market with a wide variety of architectures. With a plethora of machines to examine, many people have come up with a multitude of taxonomies for categorizing parallel machines. However, we find that categorizing the parallel programs is helpful as well. Section 2.1 presents a taxonomy for parallel programs. Section 2.2 then examines how well common architectures execute programs based on their classification in this taxonomy. From our experience with PAPERS, sections 2.3 and 2.4 presents our recommendations for improving parallel systems.

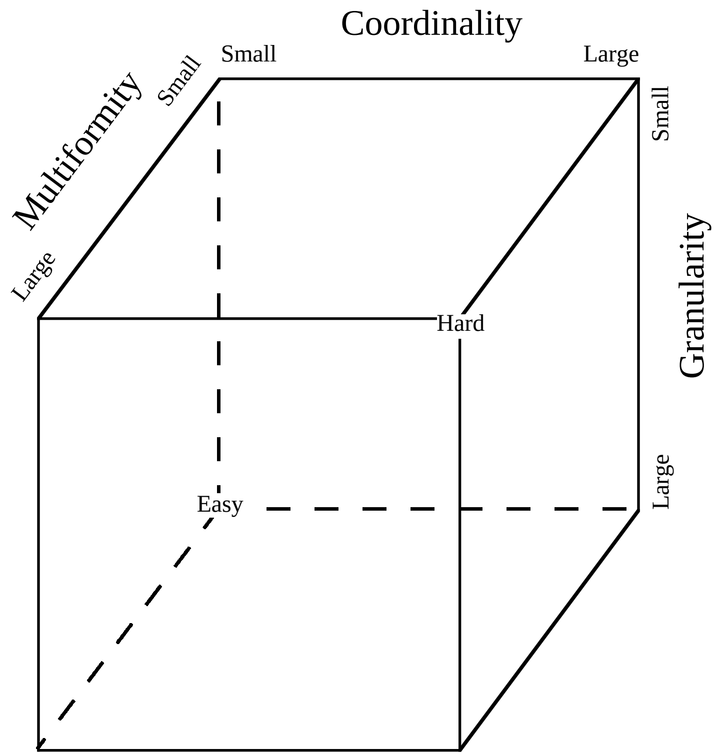


Fig. 2.1 The Parallel Processing Space.

**2.1 Our View of the Parallel Processing Space**

We find it convenient to specify three independent properties of groups of parallel subtasks within a program. The properties of a group can be represented as a point in a 3D space that we call the PPS (Parallel Processing Space), see Figure 2.1. The axes of this space are:

1) *Granularity* - the execution time of individual subtasks.
2) *Coordinality* - the complexity of subtask communications.
   This term has been specifically coined by the author to combine the meanings from its constituent words – coordination and cardinality.
3) *Multiformity* - the number of different forms of the subtasks.
   This term has been coined by the author to represent that the subtasks of a program can come in multiple forms, and this variance is an important characterization of parallel programs.

It is easy to build a machine that can efficiently execute programs that fall in the back lower left corner of the PPS, since that region corresponds to programs with large uniform grains that communicate in simple patterns. In contrast, it is much harder to build a machine that can efficiently execute programs that fall in the front upper right corner of the PPS, since that region corresponds to programs with small non-uniform grains that communicate in complex patterns.

The PPS could be expanded into additional dimensions, with axes based on such things as disk/sensor I/O requirements or memory size. However, to make the PPS a tractable analysis tool, we limit it to the axes listed above. The following three subsections examine each axis in detail.

### 2.1.1 The Granularity axis of the PPS

The first axis, the granularity axis, is a measure of the computation time taken for an individual subtask. The endpoints of the axis are the time of one clock tick and the runtime of the entire program. To assign a quantitative scale to this axis may be helpful.

For many scientific codes, floating point operations are a good unit of measure for computational "size". However a better measure is execution time (on a representative processor), since that is what most people care about. So, we label the granularity axis with the "Execution Time" of individual subtasks as the scale. Since the computation time of the subtasks in a group may not be identical, a set of points along this axis will be used to represent the range from smallest to largest grain.

To clarify some meaning, the execution of one subtask is defined as a grain of execution. In this context, a grain is not the source code of a basic block or some other representation, but rather the actual execution of that section of code on its data. Thus each iteration of the body in a parallel loop is considered a unique grain, even though the code being executed is the same. As stated before, the granularity of a program is not constant, it can vary (and usually does) from time to time as the program executes.

Notice that this definition for grain size does not include communication time. This is appropriate since the smallest theoretical grain size of a program is determined solely on the size of the segments of code that can be run concurrently without communications. Communication times are considered when determining how many PEs to use to execute the independent grains. If the network can transfer the result data from a grain in less time than a PE can execute that grain, it is possible to execute each grain on a separate PE to gain the maximum speedup. However, once the communication time for the result data from a single grain becomes longer than the time to execute that grain, it is no-longer beneficial to assign each grain to a separate PE. Most parallel computer systems allow computation and communication time to overlap to some extent. On the surface, this might seem to significantly relax the relationship of computation and communication time in determining the supported grain size of a machine. However, to take advantage of this potential overlap, a PE must find another grain to execute that is **independent** of the completion of the communications of the previous grain(s). This "overlap grain" must also not depend on results from grains on other processors that completed in the same time frame as its own predecessors on this PE.

To deal with communication times that are as large or larger than the grain size of the program, one must "burn parallelism." Burning parallelism is the act of reducing the amount of parallelism by putting multiple independent grains onto each PE. The effect of putting multiple grains of execution on a single PE allows the latency of the communications from completed grains to be overlapped with the execution of the remaining grains on that PE. Also, the result data of two or more grains might be combined into a single communication, although with possibly a longer message, effectively combining the grains. However, with any traditional network, sending a message with twice as much data, usually takes less than twice as long. Even if this is not the case, or if the result data of the multiple grains can not be combined into a single message, the effect of overlapping communication time with execution time can make parallel execution of finer grain programs possible. However, this overlapping has several serious side-effects as noted below:

- Lost parallelism: the computation time is increased by a factor k, where there are k grains per PE, that could have been executed in parallel
- Network complexity: each PE can have up to k communications in progress at once
- Memory costs: each PE now needs to hold the data/instructions for k grains all at once
- Cache flushing: the locality of reference for each grain may be harmed by execution of the other k-1 grains.

Thus for a given machine, if the minimum latency for a communication is $t$ seconds, the machine will be unable to effectively speed up the execution of programs with a grain size less than $t$ seconds. Some significant speedup may still be possible by "burning parallelism". However, if the available parallelism (the number of concurrent grains) was already fewer than the number of PEs, the speedup may not be high enough to justify the costs of using such a parallel machine in the first place. "Cost" is defined as the time and effort to write the parallel version of the program, as well as the dollar cost of designing/building/buying the machine.

For problems that have huge parallelism, significantly larger than the number of PEs, the latency of the communication network can essentially be ignored. This is due to the fact that as many grains can be put together onto a PE to hide the latency as needed, effectively combining many small grains into larger ones. Also, for programs where the grain size is already very large, the major requirement for a communication network is high bandwidth. However, when looking at the PPS, these are regions that are already covered by today's MIMD machines, and the goal of this research is to expand the realm.

The granularity axis is the simplest to understand, yet probably the hardest for a parallel processing machine to successfully "cover". The goal of a machine supporting instruction level granularity with a high parallelism width is achievable. The MP-1 machine from MasPar has an X-Net communications network that can access nearest neighbor PE's registers faster than they can read data from their own local RAM! In the MP-1 architecture, two neighboring PEs communicating over their X-Net link can transfer a 64-bit data value in 92 clocks, while for either PE to retrieve a 64-bit data value from its own local RAM takes 146 clocks [4]. And with up to 16,384 PE's, one would hope never to have to "burn parallelism". However, the MP-1 is a SIMD machine, with the multiformity problems inherent in a SIMD architecture. This will be discussed in section 2.2.1.

### 2.1.2 The Coordinality axis of the PPS

The coordinality axis is a measure of how the subtasks (grains) are coupled by communication events. For a given program, the transitions from one set of concurrent subtasks to the next set are separated by communication events. The complexity of these communication events can be quantified by counting the number of subtasks that needed to participate in the communication, either by supplying data, or by receiving data. The term coordinality comes from combining the two words coordination and cardinality. For use in plotting the coordinality of a program on the PPS, the coordinality of a communication event is recorded at the source subtasks. The minimum coordinality value is zero, a purely serial program where no communication is required. The maximum coordinality is achieved when each subtask communicates uniquely with every other subtask. Several example communication events with various coordinalities are barrier synchronization, broadcasts, multi-broadcasts, scatters, gathers, permutations, and reductions.

Any dynamic loop execution, where the number of iterations is data dependent, requires each grain with the "loop termination" data to tell the machine if it is done looping, or if another iteration is needed. This arises in optimization problems, where each PE has some measure of the quality of its current result. The machine as a whole might have a "good enough" result in one of the PE's, but the other PE's need to find out somehow. This example requires all $k$ PEs in the loop to supply data to either some central controller, or on a MIMD, to all the other $k$ PEs. In either case, at least $k$ PEs have to coordinate to get past the communication event.

A permutation communication event doesn't require a high coordinality, but can benefit from it. A permutation is where each subtask transfers data to a unique subtask, and each subtask thus receives data from a unique subtask. Looking at each communication separately, it appears that the coordinality is two. For a mesh-like communications network where each subtask is assigned to a PE, if all the PE's are sending data to the PE five locations to the right, the time to complete the communication can be greatly reduced if the PEs are coordinated. For example, if all of the PEs start sending at the same time, there is no network contention as the data passes to the first PE to the right. If the process continues, the data shifts in lock step without any contention. This would be similar to the situation when a stoplight turns green. If every car accelerated at the same time and at the same rate, the entire line of cars could get back up to speed in the same amount of time it takes for the first car to do so. However, cars at a stoplight are not well coordinated. They have to start accelerating individually, and the last car has to wait a considerable amount of time. Thus, for a parallel program, some communication patterns do not require very high coordinality when examining the individual data transfers, but when viewed as a whole, the communication pattern has a high coordinality that must be considered if the individual data transfers are to be performed efficiently.

Another highly coordinated event is a reduction, where a distributed vector is reduced to a scalar value. A global summation is an example of a reduction, where conceptually the vector of length k is collected in a central location, and then its sum is computed, with a possible broadcast of the result. To perform a global sum, a variety of algorithms can be used. One approach is to perform the reduction as it was described; collect all the data in a central location, perform the summation, and then broadcast the result. The gather and broadcast communication operations each have a coordinality of $k + 1$. Another approach, called recursive doubling, is to break the problem down into $\log_2(k)$ steps, where at each step a simple permutation of partial results is performed, summing the partial results at each PE. This approach avoids the broadcast operation at the end, as well as avoids collecting the data in a central location, but has created a sequence of very fine grain subtasks separated by permutations. To increase the grain size, programmers try to do several global sums at once, turning the problem into reducing a matrix to a vector, which might or might not be what the original task required. Either approach to performing the reduction as a sequence of simpler operations still required communication events with high coordinality.

Yet another example of a highly coordinated communication is the resolution of multi-way branches, where several subtasks contain the evaluated conditionals for the branch. Due to the nature of VLIW compiler optimizations, this is a very common event in VLIW codes . Thus, a VLIW machine must support a high coordinality to be able to resolve the multi-way branch efficiently.

Traditional MIMD machines do not have many facilities for coordinating the communications of groups of PEs. Yet, SIMD machines usually have excellent support for large coordinality communication events. This is true because to coordinate many PEs requires the PEs to be synchronized at some level, and PEs in SIMD machines are always synchronized whereas PEs in MIMD machines are not.

A relationship between coordinality and grain size should be noted. For a given machine, as the number of PEs involved in a coordinated communication increases, the time for said communication may increase as well. For example, the traditional recursive doubling reduction operations require $\log_2(n)$ steps, where $n$ is the number of PEs involved. This would indicate that in such a machine, the grain size that it can support is dependent on the coordinality. However, this is not a feature of parallel programs, but rather of the hardware. Thus the region of the PPS with small grain size and large coordinality is valid, even though building a machine that can exploit programs that fall in this region of the PPS may be difficult.

### 2.1.3 The Multiformity axis of the PPS

The third axis of the PPS is a measure of how many different forms of subtasks exist among the concurrent subtasks of a program. If all the subtasks are **uni**form in their control flow, instructions executed, and data access patterns (a SIMD, or data-parallel style), then the multiformity of the group is the minimum possible value, one. Subtasks that are similar, but have different control flow due to conditional branches, for example, a SPMD program, would have medium multiformity. At the high end of the scale are subtasks that have entirely distinct instruction sequences – what is sometimes called a "pure" MIMD program.

The multiformity axis is intended to indicate the suitability of running the subtasks concurrently on a diverse range of computer architectures, from SIMD to MIMD. This suitability is based on the flexibility of the architecture to issue the instructions and operand addresses to each of the subtasks simultaneously. A simple measure of this concept might be how many unique instruction streams are required to execute the collection of subtasks. Some examples follow that demonstrate ways subtasks can differ in form. For each example, processing element `i` can directly access `A[i]`, `B[i]`, `C[i]`, `D[i]` and the computation is parallelized using the owner computes rule.

Consider each iteration of the following DOALL loop to be a subtask, where there are no dependencies between iterations of the DOALL. Notice that this group of subtasks all have the same form, executing identical instructions, and accessing data in a simple regular pattern. Thus the multiformity of this section of a program would be measured as one, and would thus be efficiently executed on any SIMD machine.

```
DOALL i := 0 to N - 1 {
  A[i] = B[i] * S;
}
```

The following loop contains an IF statement that makes the set of subtasks have a multiformity of two. To execute this on a SIMD machine would take roughly twice as long as the above loop since the execution of the conditional code segments can not be overlapped significantly.

```
DOALL i := 0 to N - 1 {
  if (E[i]) A[i] = B[i] * S;
  else      C[i] = D[i] * T;
}
```

In contrast to the above loops, the following loop body is not uniform in its data access pattern, but it is still uniform in its instruction sequence.

```
DOALL i := 0 to N - 1 {
  A[i] = C[B[i]];
}
```

Would this count as a multiformity of N, or some smaller number, since the instruction streams are identical? This is a difficult question, since depending on the target architectures of the program, this difference in data access pattern might be relatively insignificant.

For example, the above loop would be efficiently executed on a SIMD machine such as the MasPar MP-1 that supports a register indirect addressing mode where the address in the register can be unique for each PE. However, it would be difficult to get this code to run efficiently on the Thinking Machines CM-1 or CM-2 since it does not support this non-uniform data access pattern. Thus when quantifying the multiformity access, it is important to consider the differences in form that are significant for the target architectures. If one does not want to consider a priori the target architectures, one could split the multiformity axis into three axis: control flow multiformity, op-code multiformity, and data access multiformity. However, this would greatly complicate the intended use of the PPS as an analysis tool and would be beyond the scope of this paper.

The following example uses the PARBEGIN/PAREND construct that lets a programmer specify a list of statements that can be executed in parallel. This parallel programming construct is a common way of expressing functional parallelism, and is sometimes called a parallel section.

```
PARBEGIN
   X = Y * Z;
   if (G) dowork1(); else dowork2();
   dowork3();
   R = S + sin(T);
PAREND
```

In this example, the individual statements between the PARBEGIN and PAREND keywords are the subtasks to be executed in parallel. The parallelism width and the multiformity are four. The multiformity of this example includes differences in control flow, instructions, and data access patterns.

One more issue remains concerning the multiformity axis. With independent instruction streams common in a task with a large multiformity, synchronization is lost among the grains, unless deliberately maintained or restored. As multiformity increases, the supported coordinality tends to decrease, because coordinated activities require synchronization. This may sound like a reason to not use both a multiformity and coordinality axis, since they appear to be inversely related. This however does not invalidate the region of the PPS that has both a large coordinality and a large multiformity, since programs do exist that have these combined properties. In fact, as will be shown later, it is possible to have a machine that can efficiently execute programs that fall into this region of the PPS.

## 2.2 How well do current machines cover the PPS?

As described above, a parallel program can be classified on several axis into a Parallel Processing Space. A given program will be plotted onto many points in the space, since each phase of the program will have different parallel execution needs, and each axis will have a range of applicable values. This collection of points forms a scatter plot which represents the region of the PPS that needs to be "covered" by a machine if it is going to make optimal use of the parallelism of the program. Since it is unlikely that an affordable machine will cover the entire PPS, selecting a machine that covers a majority of the PPS that the program occupies should maximize the speedup achieved.

Because a given program may occupy diverse regions of the PPS, particularly regions covered by different styles of parallel processing machines, many researchers have explored the possibility of using hybrid machines that can switch between parallel processing styles when needed [5][6][7][8][9]. An example machine is the Partitionable SIMD/MIMD machine (PASM) built at Purdue University [10][11]. This machine was designed to switch between SIMD and MIMD style execution modes at an instruction level granularity. With this capability, certain programs could now benefit from parallel execution that could not do so on either a pure MIMD or a pure SIMD machine. Thus, the PASM machine can effectively cover a larger region of the PPS than either a SIMD or MIMD machine alone. More importantly, it was discovered that PASM was able to perform a new style of execution called Barrier-MIMD [12]. This execution mode covers a region of the PPS that was not previously covered by either SIMD or MIMD machines. So, not only was the coverage larger, but it covered a previously "inaccessible" region.

The question now raised is "How would one economically build a machine to cover the regions of the PPS covered by SIMD, VLIW, MIMD, and Barrier-MIMD machines?" With such a machine, one could exploit most of the parallelism found in a much larger set of programs. The following subsections discuss what regions of the PPS are already covered by SIMD and MIMD architectures.

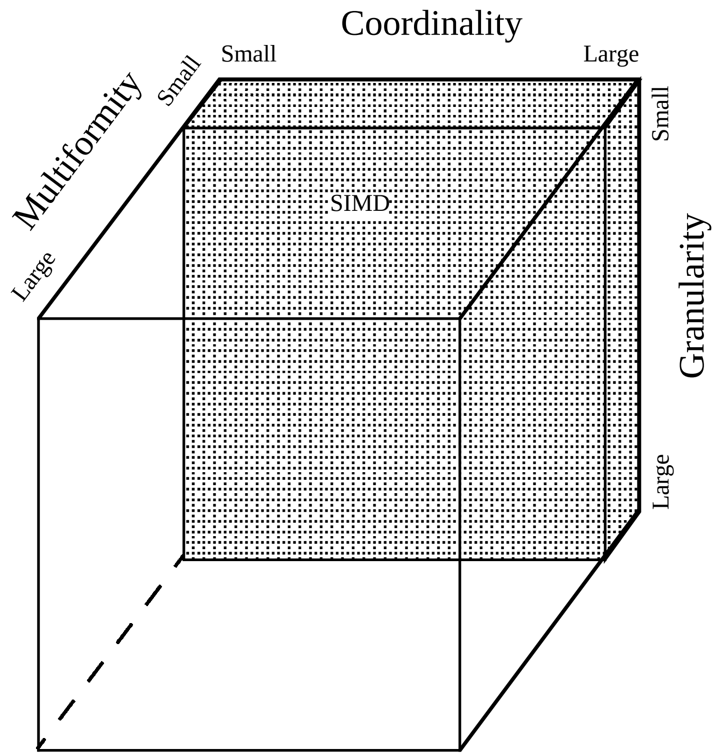


Fig. 2.2 Coverage of the PPS by SIMD machines.

**2.2.1 Coverage of the PPS by SIMD machines**

The SIMD class of machines only covers a thin slice of the PPS that covers most of the granularity and coordinality axes, as long as the multiformity is very small. This is shown in Figure 2.2. This figure, as well as subsequent PPS figures, is only an estimate, and is not based on extensive benchmarking.

SIMD machines can handle programs with very small grain sizes, with some communication patterns so fast that subtask grains can be individual instructions. As mentioned previously, the X-Net on an MP-1 can access a neighboring PE's register faster than accessing the local PE's RAM. However, when using the global router of an MP-1 for a large coordinality communication, such as a permutation, the granularity supported will not be as small. This is primarily caused by the PE to router-link ratio of 16, forcing high coordinality events to take as many as 16 router cycles to execute. As discussed in [13], some router based communication events, such as a bit reversal permutation or a shuffle permutation, can take considerably longer than the theoretical 16 cycle bound. Thus, if the communication event is complex enough, the supported granularity of the MP-1 can become significantly larger than the ideal single instruction granularity. However, most other architectures would fair even worse in granularity measures when dealing with pathologically complex permutation communications.

Compared to other architectures, SIMD machines can support programs that have very small granularity.

SIMD machines are good at coordinating the communications of all PE's at once. With both a broadcast network, and a *globalOR* reduction network, an MP-1 can have all active PE's participating in many aggregate communications without penalty. In a way, it is hard to think of non-coordinated communication activities on a SIMD machine. One of the SIMD machine's inherently good traits is its ability to support programs with large coordinality requirements.

By definition, SIMD machines can not directly support multiple instruction streams. Some SIMD machines can partition the machine into sub-machines, but this is usually limited to a small number of fixed partitions. On a SIMD machine, sequences of conditional statements are executed serially. Thus, in all respects, a SIMD machine can not efficiently execute programs that have multiformity beyond a few instruction streams. However, not all SIMD machines are really the same in this measure. On the CM-1, not only are the instructions identical for every PE, but also the local addresses of any operands are identical. Compared with a MasPar MP-1, which allows each PE to reference operands indirectly through a register, and thus to potentially different local addresses, an MP-1 has an edge over a CM-1 in the multiformity criteria.

### 2.2.2 Coverage of the PPS by MIMD machines

As shown in Figure 2.3, the estimated coverage of the PPS by MIMD machines is a triangle shaped wedge on the lower left edge. This corresponds to programs with medium to high granularity, small to medium coordinality and any amount of multiformity. Again, the shaded region is not exactly determined by extensive benchmarks, but is considered a good approximation.

Looking at the minimum communication latencies on several MIMD machines reveals a wide range in real times. When the CPU speeds are compared with the communication latencies, an even wider spread of supported grain sizes is observed. The Cray T3D has communication latencies of about one microsecond, yet its CPUs are so fast that in one microsecond, they can do 150 floating point operations (FLOP). However, that one microsecond number is for an unloaded network, and those are peak FLOP numbers. It is difficult to come up with specific real granularity numbers. As for the other MIMD machines, the granularity size is even larger. In particular, the Paragon has communication latencies of ~200 microseconds, and its fast CPUs can do 15,000 FLOPs in the same amount of time.

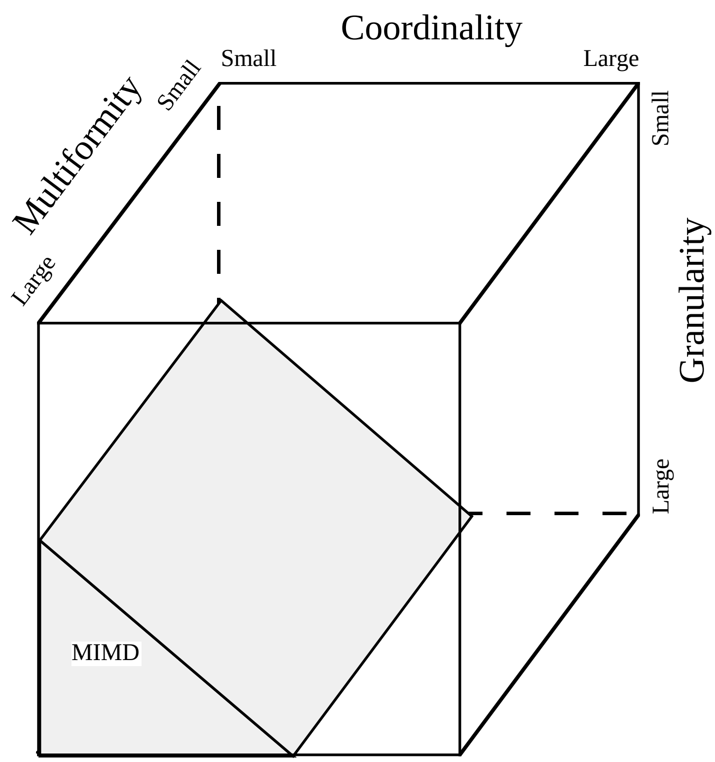


Fig. 2.3 Coverage of the PPS by MIMD machines.

These numbers are for minimum communication latencies, and for most, if not all, of these machines the coordinality is two for these "smallest" grain sizes. As stated previously, when the coordinality increases, the supported grain size may increase significantly, and for these machines, this is generally the case. This link between supported granularity and coordinality is represented by the diagonal slope of the shaded region in Figure 2.3. This slope is affected by any special network features that a MIMD machine might have that deal with higher coordinality communication events.

Some of these machines have added a barrier network that allows for "rapid" barrier synchronization. This added hardware greatly increases the coordinality that is supported. However, the general communications networks on these machines do not directly support many aggregate operations. This leads to significantly larger communication times as more PEs are involved in an aggregate operation. Thus the affective grain size supported becomes larger as the coordinality increases, leaving a region of the PPS uncovered.

With each PE able to execute entirely different instruction sequences and control flow paths, a MIMD machine supports programs with any amount of multiformity.

### 2.3 Expanding coverage of the PPS

Now that the PPS has been described, as well as how various machines cover portions of the PPS, it is logical to explore what new regions of the PPS can be covered. The SIMD machines look promising for expansion since they already cover essentially the entire granularity versus coordinality plane. Yet, it seems impossible to expand a SIMD machine's capability into the multiformity dimension. This is because a SIMD machine can not do multiple instruction streams. Thus, the question becomes, what would it take to make a common MIMD machine have the fine grain size and large coordinality of a SIMD machine, while keeping the good multiformity capabilities of a MIMD machine? In other words, what needs to be added or taken away from the common MIMD architecture to get a machine that covers all three axis of the PPS? This is the core question that this thesis explores.

#### 2.3.1 Fast barrier network hardware

The first key addition to a MIMD machine is hardware support for barriers. Many current MIMD machines have some hardware to assist in performing barriers. The Cray T3D has one of the fastest ones available (~700 ns minimum barrier time). Why, one may ask, is hardware support needed for a barrier? Can not $\log_2(n)$ point-to-point communication patterns accomplish "the same thing"? Is not this approach "scalable" because it is only $\log_2(n)$? This view is only valid if barriers are rather infrequent.

First, if a barrier takes $\alpha$ seconds to accomplish, the smallest grain size supported on this machine for SIMD style execution is more than $\alpha$ seconds. Thus, when the communications of the grains of execution are barrier events, the effective grain size supported by such a machine can be several times larger than for when the grains are using simple point-to-point communications.

Second, having a fast enough barrier network can enhance the performance of other systems in the machine. This has been demonstrated on the CM-5 in [14]. Essentially, having hardware support for barriers allows a MIMD machine to support much higher coordinality for many operations, while maintaining a similar granularity measure.

Third, in contrast to most other networks, the implementation of a barrier network can be very inexpensive. The TTL_PAPERS 951201 design [15] and primarily its successor (TTL_PAPERS 960801) have demonstrated a barrier network that scales to thousands of processors with an almost trivial pin, wire, and component cost per PE.

### 2.3.2 Low overhead interface to interconnection networks

It is common that the communication latency of MIMD computers is dominated by the time to traverse the layers of software and hardware separating the processors from the interconnection network(s). This is a major concern for those machines that have configured the interconnection network(s) to look like peripherals to each PE. With this viewpoint, is seems reasonable to force a program to use an operating system call to use the network. However, would it be acceptable to force a program to do an OS call whenever it wanted to use the floating point coprocessor in the PE? Surprisingly, the answer used to be yes. The overhead of an OS trap/call is now significantly longer than performing one floating point operation; thus the answer now is a resounding no. The same is true for the interconnection network. Parallel programs are just that – they exist across a collection of PEs. This is a key point in reducing the minimum grain size that a machine can support. On a hardware level, the interconnection network should be as close to the CPU as possible. If bus bridges and extra interface chips are placed between the CPU and the network, the latency of simple messages can become prohibitively high for fine grain parallelism. This means using CPUs that allow quick, non-cacheable, reads and writes to an external bus, possibly on a separate bus from the local RAM.

An example of one machine that has a very fast interconnection network, yet still has high PE to PE communication latencies, is the IBM SP1. In [16] the authors mention that the total latency through the multi-stage network hardware is "below one microsecond". Yet, in both [16] and [17] the minimum message passing latency achieved was around 30 microseconds. Why is the achieved latency at least 30 times longer than the latency of the network hardware? Some of the slowdown can be attributed to the use of the high latency MicroChannel as the PE's connection to the network interface card. Another source of overhead is in software level bit error detection/correction, since the underlying transport mechanism does not guarantee error free transmission. Also, some latency is contributed by software overhead in accessing the network, creating routing tags, and message headers. It is the view of this author that the majority of this inflated latency could have been eliminated when designing the SP1 system.

### 2.3.3 Multiple networks, not "One size fits all"

To build the idealized low latency and high bandwidth network interconnect between CPUs in a MIMD machine would require some quite impractical and expensive re-engineering. To be cost effective, a MIMD machine is built with COTS (Commercial Off The Shelf) microprocessors. However, commodity microprocessors do not supply a separate port that is ideal for this interconnect. A parallel processing system designer must live with what the chip manufactures supply. To do otherwise, would require re-designing the CPU, adding pins, modifying the instruction set, and increased time-to-market. To be even more cost effective, MIMD machines are being built with commodity support chips and even commodity motherboards. These CPUs, support chips, and motherboards are not generally designed for low latency transfers of data from the CPU to "slow" peripherals. Until these basic market conditions change, there will still be a significant hardware latency between CPU and the network on-ramp of a MIMD supercomputer.

The supercomputer designer must compensate for this unavoidable latency. It appears that the primary design choice has been to maximize the interconnect bandwidth to spread the cost of the latency over larger blocks of data. Other design choices have been in a variety of schemes for "latency hiding". Without argument, high cross-section bandwidth for the interconnect is good to have (and easy to advertise), but what do parallel programs really need to run faster? Programs that require high coordinality communication events are more severely penalized by high latency networks. A direct approach for improving performance of such programs would be to add a secondary network that specializes in high coordinality communication events. This secondary network would take the many separate data transfers of a high coordinality communication event and combine them into a single aggregate communication – thus reducing the effective latency of the communication.

The question now is, what should the aggregate communication network be able to do? Section 2.3.1 already made the argument that a barrier network should be added to a MIMD. In addition to supporting barriers, the aggregate communication network should include the ability to do some of the simpler reduction operations. The reduction operations that are based on logic primitives are quite easy to implement directly with trees of NAND gates. However, since the base machine is a MIMD machine, there is no central controller to receive the result of the reduction. A symmetrical solution would be to broadcast the result back to all the participating PEs, after the result is collected in a "central controller" that resides inside the network. This means that the aggregate communication network includes both broadcast and reduction primitives. This leverages the fact that to accomplish these tasks with a traditional high-bandwidth network would require $\log_2(n)$ transfer steps, thus making the new network's reduction in latency even greater. It is assumed that the barrier network added above would be used to coordinate the use of the broadcast and reduction primitives. Otherwise, the PEs would need some other way of guaranteeing that the result of a reduction is used only after all participating PEs have supplied their data to the reduction. So far, this aggregate communication network is akin to the broadcast and *globalOR* networks found in SIMD machines, such as the MasPar MP-1 [18]. This is not surprising; to make a MIMD machine cover the regions of the PPS covered by SIMD machines, we add parts of a SIMD architecture.

Reductions and broadcast are not the only communication primitives that the aggregate communication network should support. With extensive application profiling, it would seem reasonable to include hardware support for additional communication primitives that occur frequently. The choice becomes how much hardware to throw at the problem, versus how much speedup is gained. At this time, application profiling has not been done by the author, so it is unclear what additional primitives would be required, as opposed to primitives that would just be nice to have. Some promising aggregate communication primitives to consider will be presented in Chapter 3 of this paper.

### 2.3.4 Partitionability, SBM or DBM?

Finally, after re-configuring a standard MIMD machine with the above modifications and additions, one must check to see if anything important was lost on the way. This reconfiguration has lead to a loss of multiformity support. The barrier network, as well as the broadcast and reduction networks, as described, are not necessarily partitionable. This means that left as they are, all the new SIMD architectural features are limited to participation of the entire machine. Thus, at some level, each PE must be doing the same thing, even if some form of functional parallelism is warranted. With some additional complexity; the barrier, broadcast, and reduction networks can be configured to ignore some set of processors, thus allowing them to do unrelated tasks, with some facility for switching which set of processors is currently using these networks.

As found in [12], the machine would now be considered to be a Static Barrier MIMD or SBM for short. An SBM is a machine where the barrier hardware can process a single barrier group at a time, with a statically determined order of evaluation for differing sets of barrier groups. The barrier groups can be arbitrary sets of PEs, thus allowing the groups to be determined at run-time based on the data being processed/generated. This increases the multiformity support of the SBM, but the barrier groups may have to wait if the shared aggregate communication network is busy. The slowdown caused by the sequential use of a static barrier mechanism is explored in detail in [19]. The end result is that with the described SBM, the gains in supported coordinality and grain size have come with a cost of a significantly reduced multiformity support whenever using these new facilities.

This still leaves uncovered the region of the PPS that requires high coordinality, small granularity and high multiformity. To cover this region, the aggregate communication network described above may be expanded to allow dynamic re-partitioning. This calls for an example.

Consider a situation in which the machine is executing a SPMD program. At some point, a conditional expression, an "if" statement, is evaluated. Some PEs now start executing the "then" clause, while the rest execute the "else" clause. If code in the "then" clause needs to do a reduction, at the same time that the "else" clause needs to also do a reduction, on possibly different data sets, what is the system going to do? One group of PEs will have to wait if the reduction network can not simultaneously do the two different, but disjoint, operations. This can become a serious problem with nested conditionals, or with large switch/case statements. The problem arises for the same reason that SIMD machines have trouble efficiently executing SPMD codes with many conditional statements. However, the limitation is not caused by the fact that the machine has only one controller to send out the instruction stream. Rather, the limitation is caused by the existence of only one barrier and reduction network to process the independent aggregate communications.

A solution is to make all the parts of the barrier, broadcast, and reduction networks partitionable into smaller versions. Thus, when there are twenty different groups of PEs, there are now effectively twenty barrier, broadcast, and reduction networks to handle the PE groups concurrently. A subset of this functionality was described as a Dynamic Barrier MIMD (DBM) in [12] and [19]. Various machines have implemented a compromise. The Cray T3D supports up to 8 simultaneous barrier groups. (Note: the Cray T3D does not have a separate aggregate communications network as described above, but it does have the barrier network.) The PASM prototype is able to re-partition all of its network facilities so each sub-machine has the same facilities as the entire machine. Note however, that the partition boundaries of PASM are at fixed locations based on PE numbers, thus run-time evaluation of data can not be used to set up the partitions.

**2.4 Putting it all together**

To summarize, in order to convert a traditional MIMD machine into a machine that covers a majority of the Parallel Processing Space it must have the following additional features:

1) A low latency barrier network
2) A low latency aggregate communications network
3) OS support for direct user-process access to both 1 & 2,
   (also to the high-bandwidth network)
4) The ability to dynamically partition 1 & 2

With this list of features, such a machine can effectively cover all the regions of the PPS covered by SIMD, VLIW, MIMD, and hybrid SIMD/MIMD machines.  This theoretical machine could even cover the region of the PPS with high multiformity, high coordinality, and small grain size – which no current machine can cover.

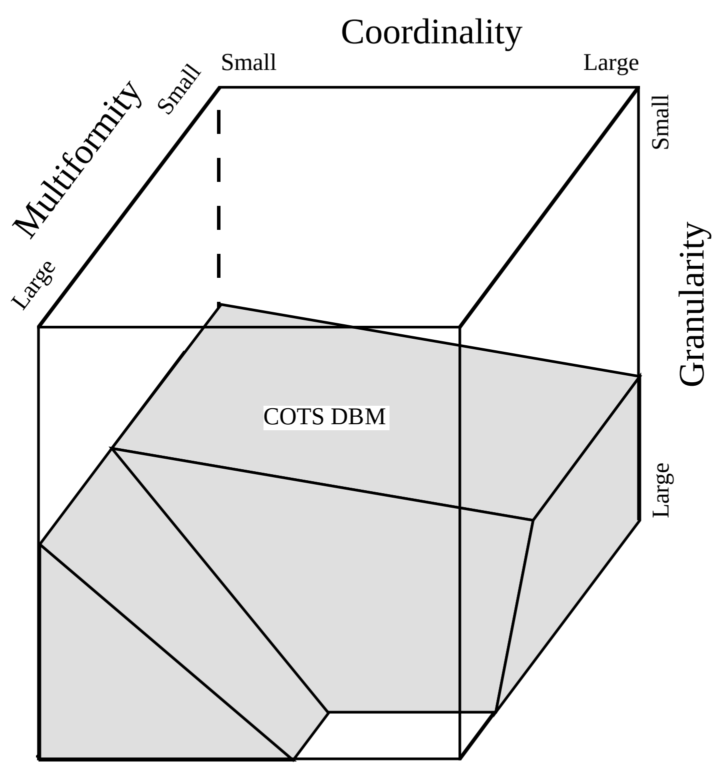


Fig. 2.4 Coverage of the PPS by COTS DBM machines.

However, as mentioned in section 2.3.3, MIMD machines built with COTS parts will have a significant communications latency for the foreseeable future.  Thus, when projecting the coverage of our new COTS DBM machine onto the PPS, the upper half of the space, corresponding to small granularity, is left uncovered, as shown in Figure 2.4. Also note that the lower front right corner is also not covered, since it is unclear that a COTS DBM can be scaled as large as a regular MIMD machine while still maintaining the full dynamic partitionability.

The PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) research group of which this author is a part, has constructed several prototypes that incorporate various aspects of the generic SBM and DBM machines described above. In particular the PAPERS1 prototype implements a full DBM with an enhanced low-latency aggregate communications network. The PAPERS1 prototype will be examined in detail in Chapter 4.

However, before describing specific implementation details, it is worthwhile to explore some of the communication primitives that would be useful in this improved MIMD architecture. This exploration is warranted since, of the four items in the above list, item number two, "A low latency aggregate communications network," has the most leeway for interpretation. Thus, Chapter 3 will expand upon the concept of Synchronous Aggregate Communications. To conclude this chapter, Figure 2.5 shows the projected coverage of the Parallel Processing Space by a variety of architectures.

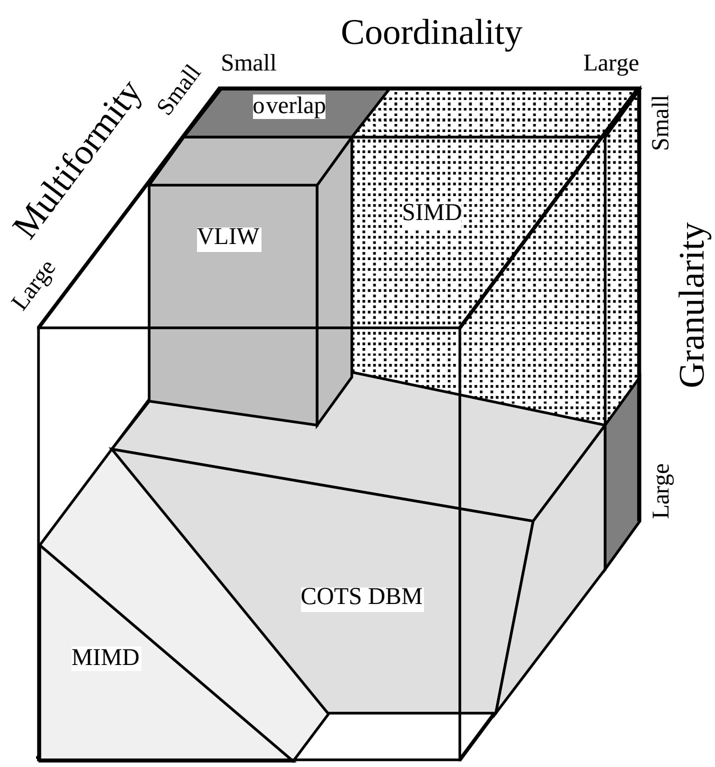


Fig. 2.5 Comparison of various architectures on the PPS.

# 3. SYNCHRONOUS AGGREGATE COMMUNICATIONS

In the previous chapter, the inclusion of both a barrier network and an aggregate communication network was advocated for improving a MIMD architecture. This chapter will describe some of the various communication operations that these two networks could perform. The operations discussed all fall into the category of synchronous aggregate communications. This category of communications is both common and useful, but, unfortunately, is rarely supported directly by today's architectures and systems.

The PAPERS project has developed a set of communication operations that take advantage of the ability to use SIMD style networks on a MIMD machine. The following few sub-sections describe these communication operations, in the order they were developed. The particular operations that will be discussed are:

- *wait*
- *waitbar*
- *putget*
- *globalNAND*
- *match*
- *vote*

The first four have been directly implemented and evaluated in various PAPERS prototypes. The last two operations, *match* and *vote*, are expansions and generalizations of the *waitbar* operation that may be directly supported in future prototypes.

For the examples in this chapter, a more specific computational model is required for clarity. The C language will be used for code examples. Unless otherwise specified, the code is identical on each PE. It is in this matter that a SPMD model is presumed. To differentiate between PEs, the value of `IPROC` is the number of "this" PE, and `NPROC` is the total number of PEs. To avoid having N unique binary executables, it is not assumed that `IPROC` is known at compile time, but that it is constant during execution. For convenience, `IXPROC` and `IYPROC` are defined to be the X and Y coordinates of "this" PE on a rectangular grid. Likewise, `NXPROC` and `NYPROC` are the number of PEs in the X and Y dimensions respectively.

## 3.1 The *wait* Operation

All synchronous aggregate communications are performed using barrier synchronization. The *wait* operation is defined as the simplest synchronous aggregate communication in which no data is exchanged. Thus, the *wait* operation is more commonly known as barrier synchronization. To perform a barrier synchronization on a traditional communication network requires many small packets to be transferred among the PEs. Even though the amount of data exchanged is quite small (perhaps literally zero data bytes), the coordinality and time complexity are large. Thus barriers are usually considered very expensive, and the programmer should avoid using them. By adding hardware support for barriers, the programmer/compiler is no longer forced to avoid using barriers simply because they are very expensive. The benefits of removing this restriction can be found in the long publication history of Dietz, Siegel, and others. The following list presents some of the benefits from having access to low-latency barrier hardware:

- Bounded timing errors for static code scheduling [19]
- Better utilization of shared resources [14]
- Repeatability – for debugging MIMD programs
- Emulation of SIMD and VLIW execution modes
- Facilitates synchronous aggregate communications

This list is by no means complete, but is sufficient to lead into the next question of what kind of barrier should be supported to maximize these benefits?

Barrier synchronization can be performed across all the PEs in a system, or across some subset of the PEs. When performing a barrier across a subset of PEs, the selection of participating PEs can be described via a barrier mask. A barrier mask is a vector of N bits for a machine with N PEs. Each bit position corresponds to an individual PE, and indicates if that PE is participating in the barrier. A participating PE is marked with a 1, and a PE that is not participating in the barrier is marked with a 0. A MIMD machine that supports this form of arbitrary grouping of PEs can be classified into two groups Static Barrier MIMD (SBM) and Dynamic Barrier MIMD (DBM) as discussed in section 2.3.4 and in [12].

As discussed in [19], it was proposed that to build a DBM would require an associative memory to hold a queue of currently active barrier masks, and it was unclear how the barrier masks would get into that queue. Another approach was discovered in October 1993 that solved the problem of building a DBM. The PAPERS0 prototype [20] was built a few months later and became the first DBM ever. The solution in PAPERS0 was to eliminate the central queue, and add the support for quickly forming new barrier masks on the fly by a new communication operation. This communication operation was originally called *waitvec* which was short for "Wait for bit vector". It was later renamed *waitbar,* and is discussed in the next section.

### 3.2 The *waitbar* Operation

The second communication primitive to be considered is called *waitbar*, which stands for "wait for a new barrier mask". As discussed in the previous section, this communication operation was designed to allow PEs to quickly construct new barrier masks. The *waitbar* operation generates a vector of bits, one bit supplied per PE. This vector is broadcast to all the PEs at the completion of a barrier as shown in Figure 3.1.

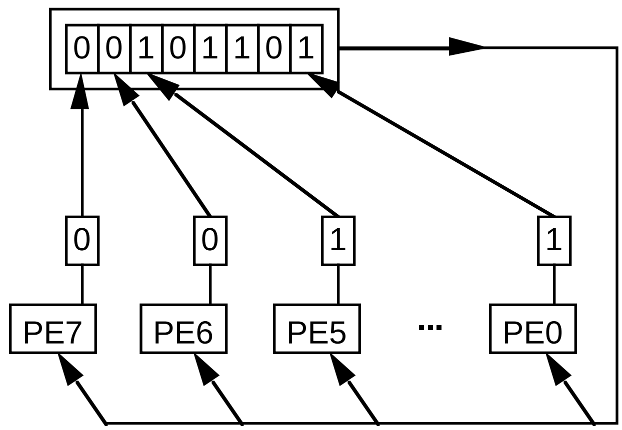


Fig. 3.1 Operation of the *waitbar* communication primitive.

When a set of PEs wish to split into two subsets, they can perform a *waitbar*, splitting into the two groups that consist of all the PEs that voted 1, and all the PEs that voted 0. For the group that voted 1, the return value from the *waitbar* is bitwise ANDed with the previous barrier mask and the result can be used as the new barrier mask. For the 0 group, the return value is first bitwise inverted before being ANDed with the previous barrier mask. In this way, a MIMD machine can be partitioned into arbitrary barrier groups with a short sequence of *waitbar* operations and simple barrier mask manipulations.

A traditional MIMD machine would find it hard to perform a *waitbar* efficiently since it requires a single data bit from each PE to be sent to every PE. If each bit is sent in its own message, this becomes the worst case all-to-all communication pattern. A tree structured "gather then broadcast" can do better, but is still rather inefficient for moving only N bits of data. However, a *waitbar* can be accomplished with very simple hardware, in conjunction with a barrier synchronization (to determine when the bit-vector is valid).

The caveat is that for machines with large numbers of PEs, the resultant bit-vector can be difficult to use if it is larger than the native word size of the CPUs.

The *waitbar* operation can be used for many things requiring global state information in addition to the barrier group partitioning described above. One example is for a MIMD machine to emulate VLIW execution. In this environment, multi-way branches based on conditionals evaluated on several PEs can occur quite often. These multi-way branches result from aggressive compiler code motions required to make VLIW worthwhile. A *waitbar* can easily collect and disseminate this "global state" so that each PE can take the correct path in a multi-way branch. More examples of what a *waitbar* can be used for can be found in [21].

### 3.3 The *putget* Operation

The next communication primitive to be considered is called *putget.* The *putget* operation encompasses most communication patterns where each PE supplies one datum to the network and receives one datum from the pool of data values in the network.

Each PE puts a datum into the PAPERS unit, and then gets a datum from the PAPERS unit. This was the basic operation mode from the original PAPERS prototype [20]. However, there was no choice of what data was received. In each instance one bit from every processor was packed into a bit vector, thus performing a *waitbar*. This idea was expanded to allow a PE to tell the PAPERS unit to return a particular datum, formed in a number of different ways. The first obvious choice was to get more than one bit from a single PE. This is what became the *putget* operation, which was developed by the author and Jeff Sponaugle during the design of the PAPERS1 prototype.

The significance and uniqueness of the *putget* operation was not realized immediately. Since the destination PE was selecting the source PE in any data exchange, it was obviously not the same as message passing, where the source PE specifies the destination PE. This was acceptable from a compiler's viewpoint, since this only shifted which PE had to tell the PAPERS unit, but both PEs would already "know" who the other PE was in the data exchange. From a programmer's point of view, the *putget* operation might take some time getting used to.

Since the destination PE specifies which PE it reads data from, there is no theoretical reason that more than one PE can not read from the same source PE. This makes multi-broadcasts a simple operation, since the sending PE does not have to specify a list of recipients, as in message passing. Rather, the recipients all specify the same source. In essence, the inherently parallel task of a broadcast is more fully realized by shifting the task of specifying the recipients from a serial to a parallel domain.

For the opposite situation of multiple senders to one receiver, the operation is inherently serial at some level, no matter which communication scheme is used. This is due to the fact that a PE is assumed to be a serial device, and can only process one incoming datum or message at a time. In a message passing scheme, either some of the senders block, or the network has "enough" buffering to hold all the messages until the destination PE can read all of them. Since the *putget* operation does not allow a program to get into this situation, the network does not have to deal with this problematic condition. The programmer or compiler must deal with it. In reality, even if a network was able to deal with this condition, the programmer or compiler should avoid it since, as was stated earlier, it is inherently serial. In some real systems it has been shown that even if the network can buffer up the messages, better performance can be achieved if the sending PEs wait until the receiving PEs are ready to read the data before sending [14].

The following is a list of some of the useful operations that a *putget* can implement and which are discussed below in sections 3.3.1, 3.3.2, and 3.3.3:

- Any permutation communication pattern
- Multiple simultaneous broadcasts from any PE to disjoint sets of PEs
- Random point to point communications

The PAPERS communications library includes a version of *putget* for each atomic data type defined in C: `float`, `double`, `unsigned int`, `char`, etc. The source for the PAPERS1 communications library used with the PAPERS1 prototype can be found in Appendix A, and a brief discussion of how the library works is in section 4.4.

### 3.3.1 Permutations with *putget*

A permutation communication is one where each PE sends and receives a datum from a unique PE. For the following and subsequent examples, the function declaration for a 32-bit integer *putget* is

```
int p_putget32(int datum, int source);
```

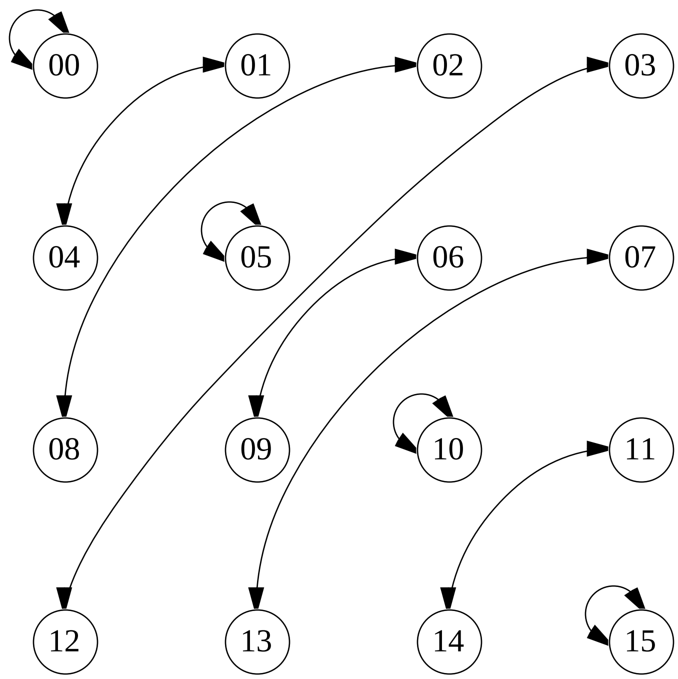


Fig. 3.2 Matrix Transpose as an example permutation.

An example of a permutation communication is a matrix transposition, where the elements of the matrix are mapped to a rectangular grid of PEs. Figure 3.2 demonstrates the communication pattern for a 4x4 Matrix transpose operation. Each PE's call to a *putget* communication library routine to accomplish this pattern for an integer matrix is (where `M` is the element of the matrix on this PE):

```
M = p_putget32(M, (IXPROC * NXPROC) + IYPROC);
```

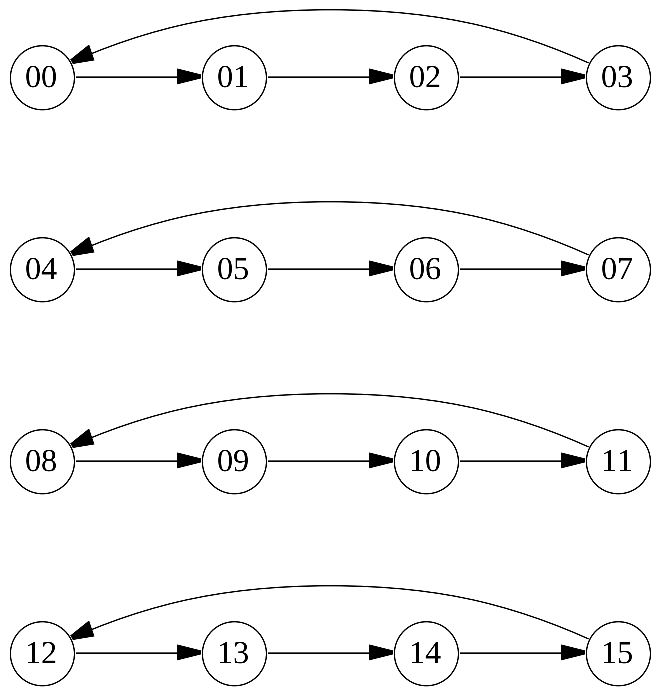


Fig. 3.3 Shift right as an example permutation.

Another example is a shift right (left, up, down, etc.) where the PEs are logically arranged in a set of rings. Shifts are common in image processing algorithms, where each PE has a section of the image, and requires information about the surrounding sections to complete its work. Figure 3.3 shows the communication pattern generated by:

```
source = IYPROC * NXPROC + (IXPROC + NXPROC - 1) % NXPROC;
result = p_putget32(myData, source);
```

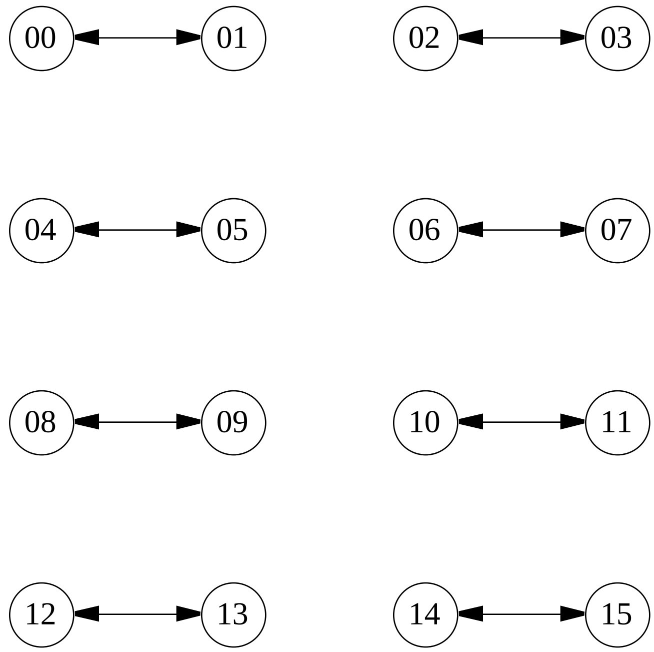


Fig. 3.4 An example of an exchange between even and odd PEs.

An exchange permutation is an example where the machine is split into halves, with the two halves exchanging data. A simple example is splitting a machine into the even numbered PEs as one half, and the odd numbered PEs as the other half. Each PE would then exchange data with the PE that differs in the least significant bit of its PE number. This pattern is shown in Figure 3.4 and is generated by the following code:

```
result = p_putget32(myData, IPROC ^ 1);
```

Usually, each PE knows from where it will receive data, as well as where its data is going. This prior knowledge comes from many possible sources: compile time calculations, run time calculations, or run time voting/communications. If a compiler needs to redistribute the elements of an array into a new layout, this is commonly a permutation. Since the compiler knows how the data is to be redistributed, it is a simple matter for each PE to determine where its is to receive data from. If the redistribution pattern is not known at compile time, the PEs must agree on the pattern at run time. This pattern information can be obtained at runtime using other aggregate communication operations.

### 3.3.2 Multi-Broadcasts with *putget*

A multi-broadcast communication is an operation where one or more PEs sends a datum to unique groups of PEs.

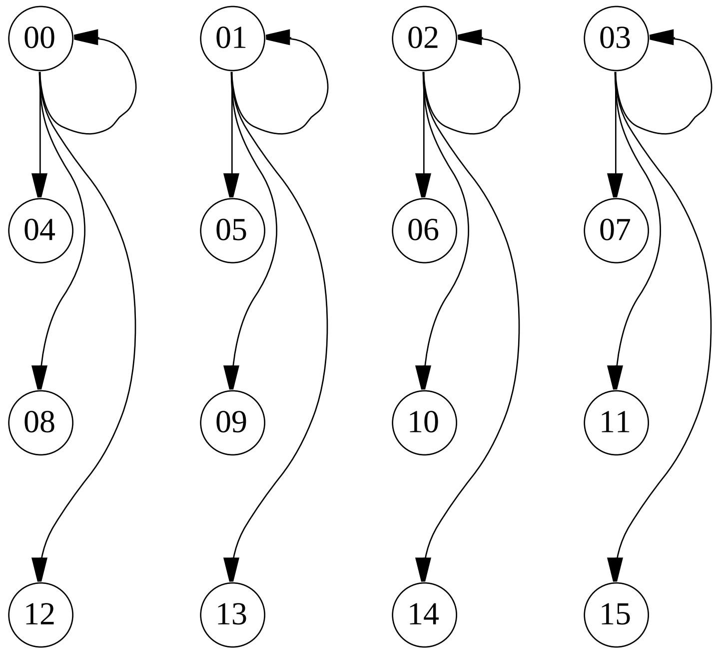


Fig. 3.5 An example of a column multi-broadcast.

An example of multi-broadcast communications are vector expansions, where a one dimensional vector is expanded into a two dimensional vector or matrix. Figure 3.5 demonstrates the communication pattern for a row vector expanded into a matrix by the following code:

```
result = p_putget32(myData, IXPROC);
```

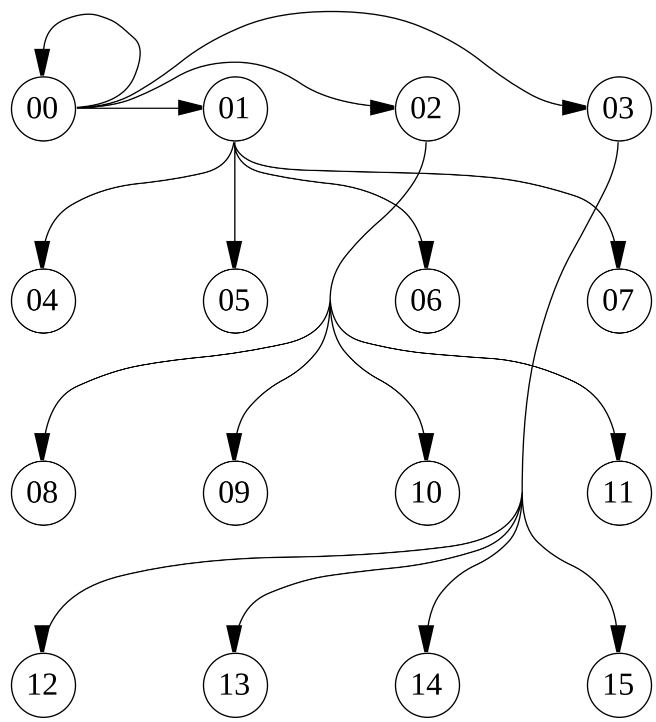


Fig. 3.6 An example of a more complex multi-broadcast.

Similarly, the following code expands a column vector into a transposed matrix, as shown in Figure 3.6:

```
result = p_putget32(myData, IYPROC);
```

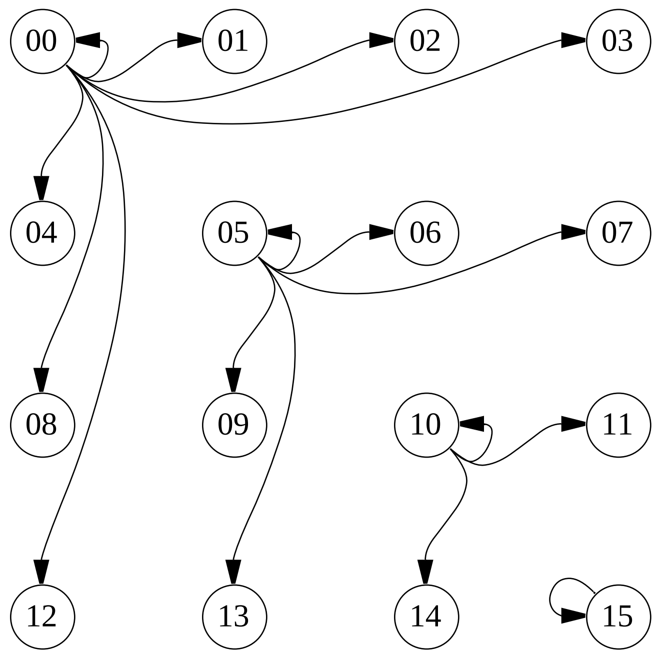


Fig. 3.7 An example row-column multi-broadcast.

The next code fragment takes data along the diagonal of a matrix and broadcasts the values down the remainder of the column and row. This operation is shown in Figure 3.7.

```
if (IXPROC < IYPROC) source = IXPROC * NXPROC + IXPROC;
else source = IYPROC * NXPROC + IYPROC;
result = p_putget32(myData, source);
```

### 3.3.3 "Random" Point to Point communications with *putget*

A point to point communication is an operation where one PE sends a datum to another PE. In message passing environments, the sender must inform the interconnect fabric of both what data to send, and who to send it to. With a *putget* operation, the sender only needs to supply the data to send, while the receiver must tell the network who to get data from. This division of responsibilities makes some communication patterns more efficient than with message passing; however, *putget* requires more cooperation among the PEs.

Since *putget* is an aggregate operation, involving several PEs, the mapping of send and receive pairs may not be a trivial task. To solve this problem, other aggregate communication primitives such as *globalNAND*, *match*, and *vote* can be used to set up the communication pattern in advance. These other operations will be described in sections

3.4, 3.5 and 3.6. For truly random communication patterns, especially asynchronous communications, *putget* is not ideal, and some other communication method might be more appropriate.

### 3.4 The *globalNAND* Operation

The next communication operation to explore is the *globalNAND*. The *globalNAND* operation is a barrier where each PE supplies a datum, and each PE receives the logical NAND of all the datums. Figure 3.8 gives an illustration of the logic and flow.

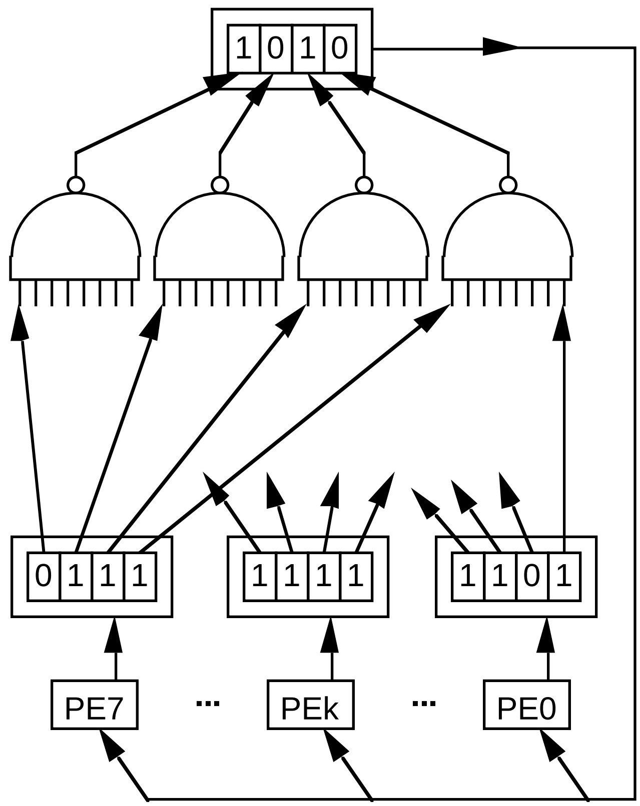


Fig. 3.8 Operation of the *globalNAND* communication primitive.

On first consideration, this may not seem all that useful. However, with some resourcefulness, the *globalNAND* is a very capable communication primitive. Like the *globalOR* network in the MasPar MP-1 and MP-2 [18], the *globalNAND* can be used to get global state information about all the PEs and to perform reduction operations. From basic Boolean algebra one can see that a set of PEs using a NAND as an OR requires only an inversion of the datum before it enters the NAND tree. By definition, a NAND is only one inversion away from an AND. The *globalNAND* can also be used as a broadcast if every PE except the broadcasting PE sets their datum to all 1s and the broadcaster (or receivers) inverts the data. If the datum is δ bits wide, δ PEs can each broadcast one bit to all N PEs. Using the *globalNAND* in this way, a *waitbar* operation can be performed in N/δ operations.

Table 3.1 lists some of the operations that can be performed with the *globalNAND*. To obtain a result ρ bits in size, the number of *globalNAND* operations is shown if the *globalNAND* data path is δ bits wide and there are N PE's. Note: for *waitbar* ρ and N are equal.

Table 3.1
The cost of various operations using the *globalNAND* communication primitive.

| | Equation | $\delta = 4, \rho = 32$ | $\delta = 16, \rho = 32$ | $\delta = 16, \rho = 64$ |
|---|---|---|---|---|
| `reduceOr` | $\rho/\delta$ | 8 | 2 | 4 |
| `reduceAnd` | $\rho/\delta$ | 8 | 2 | 4 |
| `reduceMin` | $\rho/\log_2(\delta)$ | 16 | 8 | 16 |
| `reduceMax` | $\rho/\log_2(\delta)$ | 16 | 8 | 16 |
| `waitbar` | N/δ or r/δ | N/4 | N/16 | N/16 |

The paper [21], discusses a wide range of operations that can be done efficiently with the *globalNAND* communication primitive. It is worthwhile noting that for both `reduceMin` and `reduceMax`, that the number of *globalNAND* communication operations is independent of the number of processors. This was an unexpected result, and can be exploited on very large machines to quickly find a global minimum/maximum. Thus a simple group of NAND trees (with an associated barrier unit) should be able to significantly improve the performance of genetic algorithms, which require the evaluation of a global fitness function very frequently.

### 3.5 The *match* Operation

Another aggregate communication primitive that was developed for the PAPERS research group is called *match*. The author developed this concept as part of a VLSI design project for the EE 559 course, which is discussed in section 5.2.

A *match* is performed by each PE putting a datum into the PAPERS unit, and receiving a result vector. This result vector has 1s for those PEs that had the same datum value as this PE, and 0s otherwise. The idea is to create disjoint sets of PEs, where each set is identified by a unique datum. A variation on *match* is *matchCount* that returns the count of how many PEs matched this PE's value instead of a bit vector.

A *match* operation can be used for two primary things: resolving contention for shared resources, and dividing a problem into disjoint parts.

For example, in a parallel database application, each PE must update some random entry in the database. Prior to an update operation, a PE must be assured exclusive access to a particular database entry. By performing a *match* operation on the entry's ID number, a PE can find out which other PEs also want to update that particular database entry. Once this is known, the PEs with a conflict can statically resolve it in a way that maintains correctness. For example, if multiple updates to a single entry must appear to have occurred serially, that can be arranged by sequencing through the list of PEs in some pre-determined order. Or, if correctness can be preserved by choosing one winner of the race condition, a winner can be picked locally from the list and all the losing PEs discard their updates while the one winning PE proceeds with the update.

An example of dividing a problem into disjoint parts is at the evaluation of a switch/case statement of a SPMD program. The following example shows how the *match* function might be used to partition a machine into disjoint groups of PEs where all the PEs within a group will work together to perform a particular action.

```
myGroup = p_match32(myValue);
p_setmask(myGroup);    /* Partition into disjoint groups */
switch(myValue) {
   case 1:
      /* perform action 1 with PEs in myGroup */
      break;
   case 2:
      /* perform action 2 with PEs in myGroup */
      break;
...
   case k:
      /* perform action k with PEs in myGroup */
      break;
}
p_setmask(ALL);   /* recombine groups */
```

With this scheme, the partitioning of a machine based on runtime data is fairly simple.

### 3.6 The *vote* Operation

A similar operation to *match*, is called *vote*. A *vote* is also performed by each PE putting a datum into the PAPERS unit, and receiving a result vector. However, instead of the values being arbitrary, they represent "votes" for particular PEs. Thus the result vector for a PE has 1s in the positions corresponding to PEs who voted for this PE. (i.e. for those PEs whose datum matched this PE's ID, a 1 is put into the result vector for this PE.) A variation on *vote* is *voteCount* that returns the count of how many PEs voted for this PE instead of a bit vector.

A *vote* operation can be useful for preparing contention free communication paths, by pre-determining the set of destination nodes with only one writer. These nodes would be the PEs that received only one vote.

# 4. IMPLEMENTATION

The basic premise of the PAPERS project (Purdue's Adapter for Parallel Execution and Rapid Synchronization) is to make a device to connect several conventional workstations or PCs together with the appropriate facilities for using the cluster of machines as a single parallel processing machine. This chapter describes the PAPERS1 prototype implementation. The following is a brief history of the PAPERS project preceding the PAPERS1 prototype completion.

The original prototype was built in February 1994 as a test-bed for a new barrier mechanism. An in-depth coverage of that prototype, as well as the developments leading up to it are included in Tariq Muhammad's MS Thesis [20]. The reader is directed to this work for further details. Since the first unit, a considerable number of prototypes have been built, with still more planned.

From the first prototype, many important points were recognized. One of the major ideas suggested was that barriers and aggregate communications can be combined for a significant benefit. This means that at the time of a barrier event, the MIMD machine looks like a SIMD architecture, and thus can use SIMD style communication facilities. This makes many aggregate communication operations possible that a MIMD machine could not otherwise execute easily.

The PAPERS1 prototype was designed to expand upon the original PAPERS prototype, improving raw performance as well as adding new classes of aggregate communication operations. The PAPERS1 prototype was designed and constructed by the author and Jeff Sponaugle over a three month period, from June to August 1994. The design phase, and early construction where split fairly evenly between Jeff and the author. Subsequent work on the prototype was performed primarily by the author. The actual construction was completed in early July, but various modifications were later applied to improve performance, increase reliability, and to add a few features. The many revisions of the PAL logic equations may be of historical interest, but for this paper, primarily the final setup is described. The PAPERS1 software library (found in Appendix A) was written by the author after the prototype's hardware stabilized. The functionality of this library was based on the PAPERS communication library developed by Prof. Hank Dietz, T. M. Chung and the author [22]. After completion of the majority of this thesis, the PAPERS communication library has been re-organized, improved, and expanded into AFAPI (Aggregate Function API) [23].

Section 4.1 describes the high-level features of the PAPERS1 prototype. Section 4.2 then describes the architecture of PAPERS1 that supplies those features. The state machines inside PAPERS1 are then described in section 4.3. The chapter concludes in section 4.4 with a brief description of the software support for PAPERS1.


Fig. 4.1 A photo of PAPERS1.

**4.1 PAPERS1 Features**

The PAPERS1 prototype, shown above in Figure 4.1, has the following features:

1) Full dynamic partitionability (a DBM implementation)
2) 2 cycle aggregate functions
3) 4-bit *globalNAND* communication primitive
4) 4-bit *putget* communication primitive
5) Interrupt/Priority Barriers
6) BPU (Barrier Processing Unit) ID number

A description of each of these features follows, along with commentary/discussion of each feature's significance.

The ability for PAPERS1 to partition into arbitrary sets of PEs is maintained from the original PAPERS prototype. This partitioning allows runtime dynamic data to be used to determine the sets of PEs that will work together. The PAPERS1 structure guarantees interference free operation between mutually exclusive sets of PEs.

All data and barrier operations are performed as "two cycle operations". This means that for each PAPERS1 operation, a PE needs to perform a minimum of two I/O operations, one output, and one input. The original PAPERS prototype was designed to perform barriers with a minimum of four I/O operations, but subsequent software improvements allowed barriers to be performed in a minimum of two I/O operations. The original PAPERS prototype required between four and five I/O operations to transfer a data value. The theoretical minimum would be a single cycle operation triggered by a memory mapped read by the PE's CPU, with the address of the read containing the data & control information for the PAPERS operation.

The 4-bit *globalNAND* communication primitive is a synchronous reduction operation, where each participating PE supplies a 4-bit data value, and each PE receives the bitwise 4-bit NAND of all the collected data (see section 3.4). This operation can be easily used for `reduceOr`, `reduceAnd`, broadcast, and voting operations. This operation was introduced in the first TTL_PAPERS prototype, and was retrofitted into the PAPERS1 prototype.

The 4-bit *putget* communication primitive is a synchronous multi-broadcast operation. Each participating PE puts a 4-bit data value into the PAPERS unit, and then gets a 4-bit data value from the PAPERS unit. Which PE the returned data comes from is specified by the receiving PE. Multi-casts can be used to implement simpler communications such as permutations. Section 3.3 discussed *putget* in more detail.

The PAPERS1 prototype implements an interrupt/priority barrier mechanism that allows for dealing with unexpected runtime events. Essentially, a priority barrier has a higher priority than the regular barriers, and allows a PE to interrupt an already initiated barrier. The mechanism is a true barrier, and not just an interrupt. This means that after each PE acknowledges the interrupt, the priority barriers can be used in a sequence to exchange information between the PEs about the unexpected event.

Each PE can query the PAPERS1 unit as to what the ID number is of the BPU it is attached to. This ID number is normally the same as the PE number used by the software. This allows the software to detect if the PE has been connected to the expected BPU.

**4.2 PAPERS1 Architecture**

The PAPERS1 prototype consists of a central box connected to four PCs through their parallel printer ports. Each PC acts as a Processing Element (PE) in the parallel machine. Inside the PAPERS1 box there are four interconnected Barrier Processing Units (BPUs) as shown in Figure 4.2. A photo of the inside of PAPERS1 is in Figure 4.3.

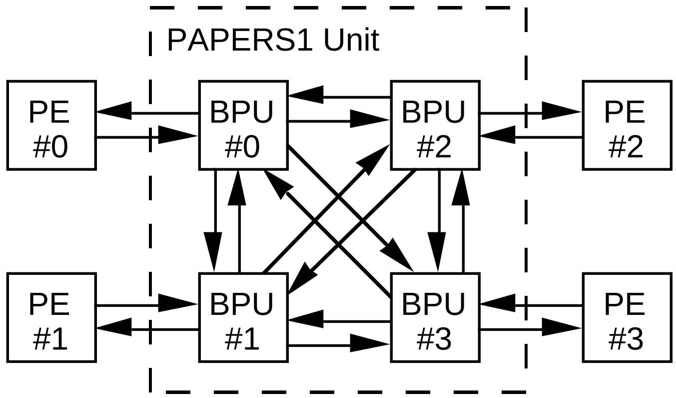


Fig. 4.2 The top level block diagram of PAPERS1.

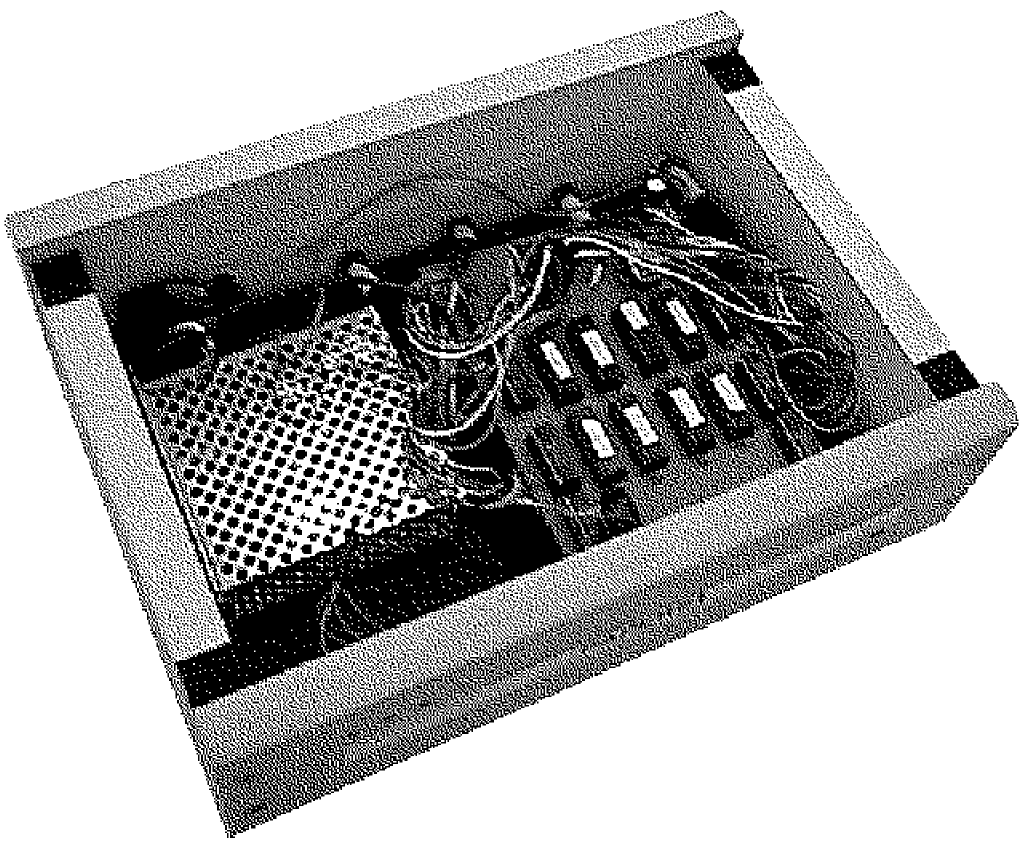

Fig. 4.3 Photo of PAPERS1 inside.

### 4.2.1 The PE's interface to PAPERS1

The signals in the interface between a PE and PAPERS1 are grouped into three sets called P_OUT, P_IN, and P_MODE. The sets correspond to the three I/O port addresses associated with a Centronics Printer Port [24]. The signal names for all three sets are shown in table 4.1. The signals are described as the software sees them. Due to the circuit design of the Centronics printer port, the actual electrical logic levels are flipped on the signals marked with a * in the table.

Table 4.1
The PAPERS1 PE interface signals.

| Port | Bit | Name | Description |
|---|---|---|---|
| P_OUT | 7 | P_S | Strobe |
| P_OUT | 6 | unused | unused |
| P_OUT | 5 | P_SRC1 | SouRCe bit 1 |
| P_OUT | 4 | P_SRC0 | SouRCe bit 0 |
| P_OUT | 3 | P_D3 | output Data bit 3 |
| P_OUT | 2 | P_D2 | output Data bit 2 |
| P_OUT | 1 | P_D1 | output Data bit 1 |
| P_OUT | 0 | P_D0 | output Data bit 0 |
| P_IN | 7 | P_I3* | Input data bit 3 |
| P_IN | 6 | P_I2 | Input data bit 2 |
| P_IN | 5 | P_I1 | Input data bit 1 |
| P_IN | 4 | P_I0 | Input data bit 0 |
| P_IN | 3 | P_RDY | ReaDY |
| P_MODE | 3 | P_B3* | Barrier mask bit 3 |
| P_MODE | 2 | P_B2 | Barrier mask bit 2 |
| P_MODE | 1 | P_B1* | Barrier mask bit 1 |
| P_MODE | 0 | P_B0* | Barrier mask bit 0 |

The P_OUT port is used to send the P_S signal to the PAPERS1 unit to initiate/join a barrier. Both rising & falling edges of the P_S signal tell the PAPERS1 unit that this PE is at a new barrier. Along with the strobe, a nibble of data, P_D3 through P_D0, is supplied for access by other PEs. Also, a two bit source field, P_SRC1 and P_SRC0, specifies which data value the PAPERS1 unit should return to this PE at the completion of the barrier.

The P_IN port is read by the PE to determine when a barrier is complete and to collect the nibble of data returned by PAPERS1. When the P_RDY signal is high, the previous barrier has completed and the data nibble, P_I3 through P_I0, is ready. The data returned is either from the PE selected by the source field in P_OUT, or is the NAND of all the other PE's data. Thus, if the PE selects itself, the data returned is the NAND of all the other PE's data.

The P_MODE port is used by the PE to tell PAPERS1 which PEs are in its barrier group. The PEs whose corresponding bit in the barrier bit mask are zero do not affect the completion of the barrier. Their data is ignored as well. Since a PE is always in its own barrier group, its bit position in the barrier mask has a different meaning to PAPERS1. This bit position is used to signal an interrupt request when it is zero. Note, the special case of the barrier mask being all zeros is used in conjunction with a P_S value of zero to do an asynchronous reset of the state machine in PAPERS1 associated with this particular PE. This allows the PE to ask for a reset without knowing the BPU ID number it is attached to. During the reset the BPU ID number is sent back to the PE.

### 4.2.2 A Barrier Processing Unit (BPU)

As mentioned above, PAPERS1 is built from four interconnected Barrier Processing Units. Each BPU is made of a Control Module, a Data Module and an I/O Module. To avoid confusing generalities, this discussion will focus on the design of the BPU attached to PE0, although each BPU is essentially identical. A block diagram of BPU #0 is shown in Figure 4.4.

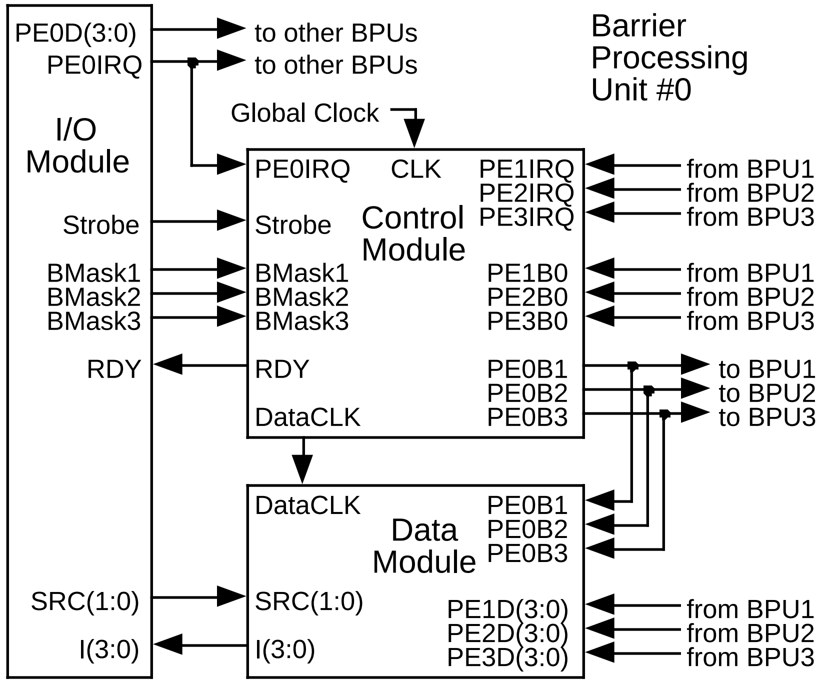


Fig. 4.4 Details of the Barrier Processing Unit for PE0.

The control module is a CE22V10 PAL which contains a state machine, barrier completion logic, and interrupt logic. The data module is also a CE22V10 PAL, and it contains the logic for the *putget*, *globalNAND*, as well as the hardwired BPU ID number. The I/O module consists of two 74ACT541 octal line drivers. These are used to isolate the electrical noise of the cables from the logic on the board, as well as supplying higher drive currents than the PALs for sending crisp signals back to the PEs. A system clock runs the four state machines synchronously, so that when barriers are satisfied, all the PEs are notified simultaneously. The signals shown in Figure 4.4 are documented in Table 4.2. See Appendix B for the complete list of PAL logic equations.

Table 4.2
The signals of Barrier Processing Unit #0.

| Name | Source | Description |
|---|---|---|
| CLK | 25 MHz crystal | Global clock that drives the state machines |
| Strobe | I/O Module | Indicates arrival at a barrier by PE0 |
| RDY | Control Module | Indicates completion of a barrier |
| BMask1 | I/O Module | PE0's Barrier Mask bit 1 |
| BMask2 | I/O Module | PE0's Barrier Mask bit 2 |
| BMask3 | I/O Module | PE0's Barrier Mask bit 3 |
| PE0IRQ | I/O Module | Indicates PE0 wants an exception |
| PE1IRQ | BPU #1 | Indicates PE1 wants an exception |
| PE2IRQ | BPU #2 | Indicates PE2 wants an exception |
| PE3IRQ | BPU #3 | Indicates PE3 wants an exception |
| PE0D(3:0) | I/O Module | The 4-bit data field from PE0 |
| PE1D(3:0) | BPU #1 | The 4-bit data field from PE1 |
| PE2D(3:0) | BPU #2 | The 4-bit data field from PE2 |
| PE3D(3:0) | BPU #3 | The 4-bit data field from PE3 |
| SRC(1:0) | I/O Module | Selects which data field to return |
| I(3:0) | Data Module | The 4-bit result returned to PE0 |
| DataCLK | Control Module | Causes the Data Module to act on its inputs |
| PE0B1 | Control Module | PE0 is at a barrier that should include PE1 |
| PE0B2 | Control Module | PE0 is at a barrier that should include PE2 |
| PE0B3 | Control Module | PE0 is at a barrier that should include PE3 |
| PE1B0 | BPU #1 | PE1 is at a barrier that should include PE0 |
| PE2B0 | BPU #2 | PE2 is at a barrier that should include PE0 |
| PE3B0 | BPU #3 | PE3 is at a barrier that should include PE0 |

It should be noted that there isn't a BMask0 signal in either Figure 4.4 nor in Table 4.2. As mentioned at the end of section 4.2.1, what would normally be bit 0 of the barrier mask is used instead as the active low IRQ signal for PE0.

The BPUs for the other three PEs are similarly configured, with the appropriate adjustments to signal names, as documented in Appendix B. To get a better feel for the wiring of PAPERS1, Figure 4.5 is a medium level block diagram that shows the logical connections between the various modules of the four BPUs. Also, a close-up photo of the PAPERS1 wire-wrap board is shown in Figure 4.6.

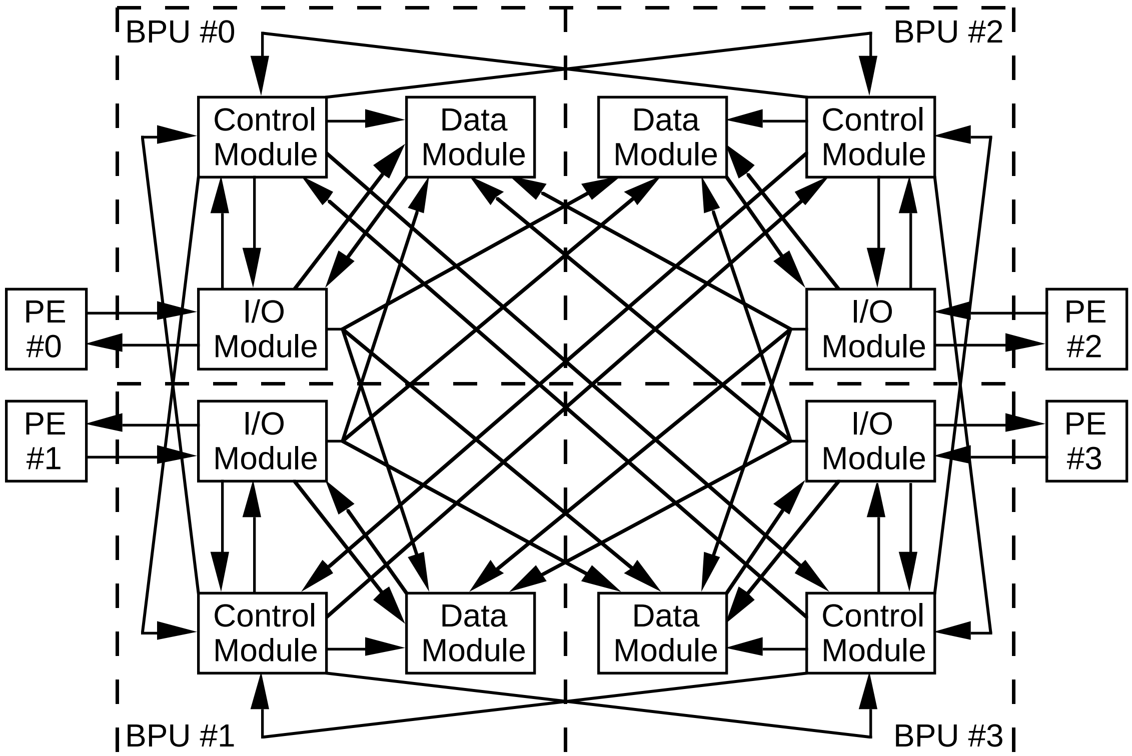


Fig. 4.5 A medium level block diagram of PAPERS1.

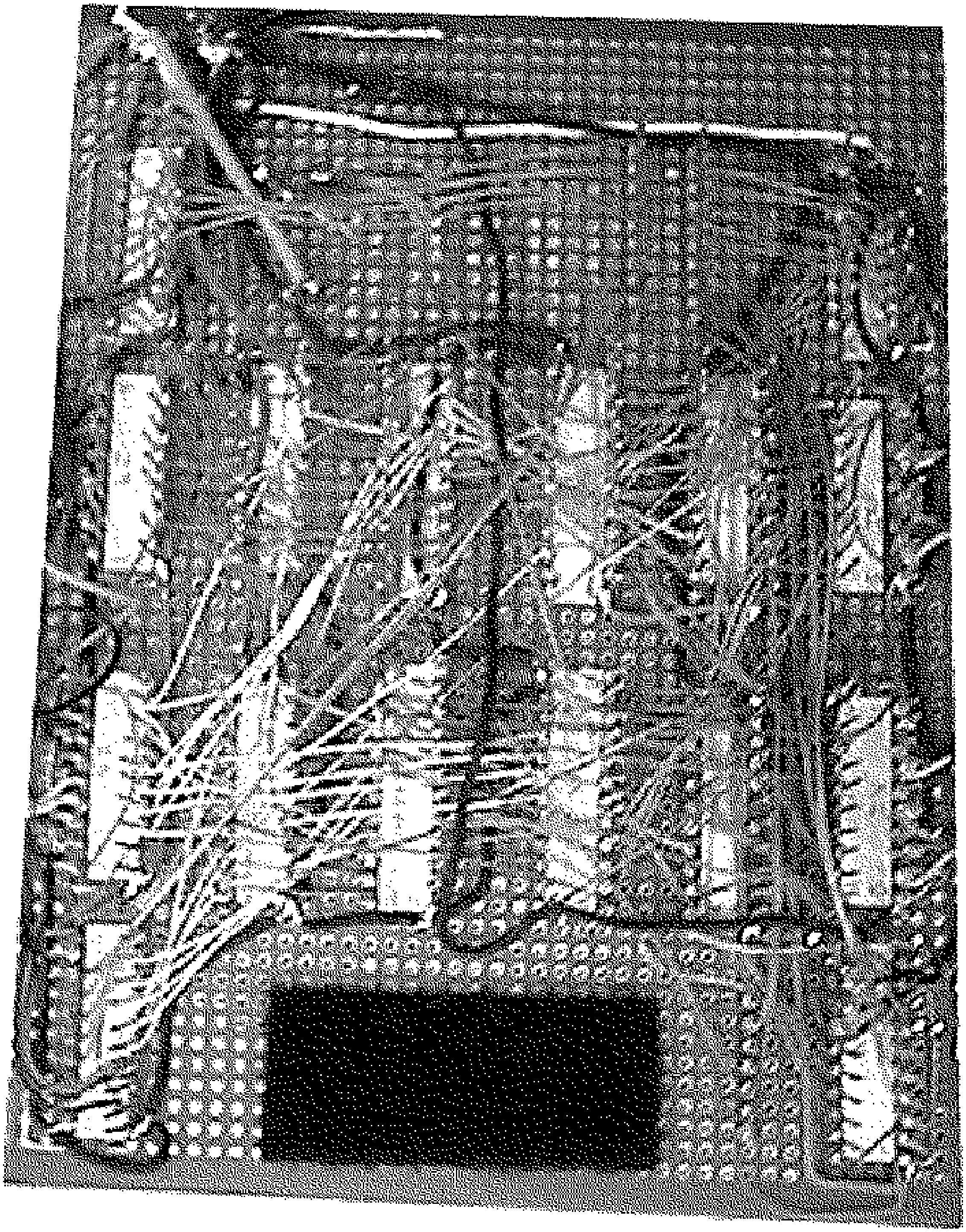

Fig. 4.6 PAPERS1 wire-wrap interconnect photo.

**4.3 PAPERS1 State Machines**

The original PAPERS prototype used an asynchronous, event driven design to process barriers [20]. This proved to be difficult to debug, had noise immunity problems, and, most importantly, required at a 4 to 5 cycle data communication as opposed to the 2 cycle structure desired. Thus, the PAPERS1 design was based on synchronous state machines to solve most of the problems encountered with the asynchronous design. The state machine design evolved considerably from the initial to the final implementation. It is instructive to examine the original state machine in PAPERS1, since it is simpler.

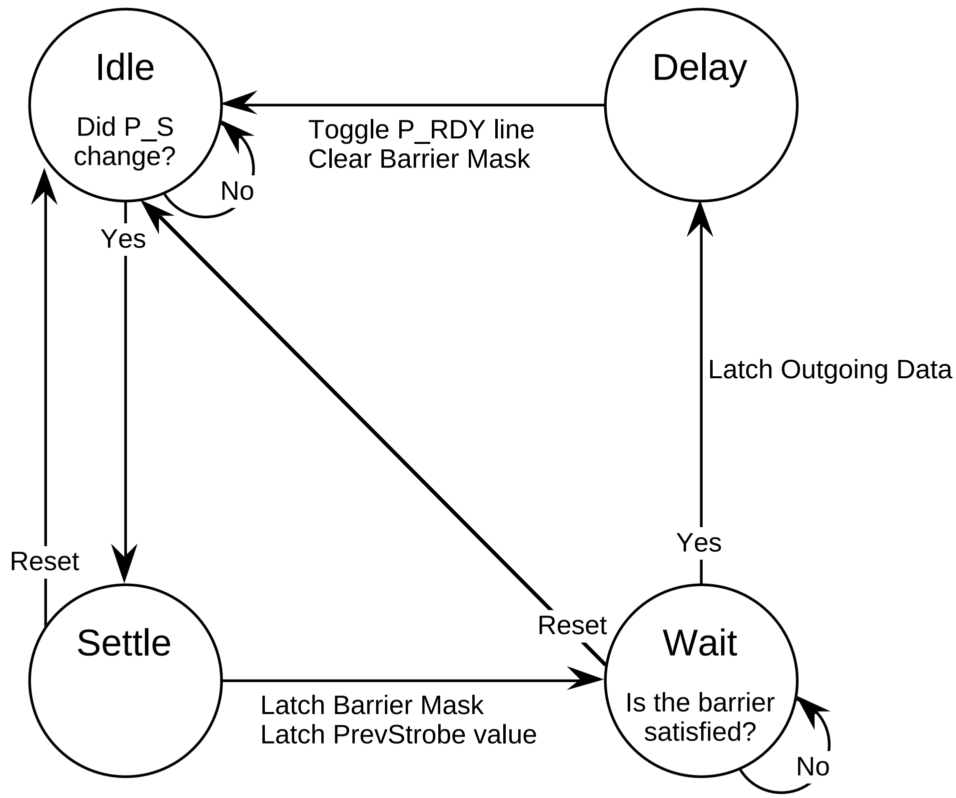


Fig. 4.7 The original PAPERS1 state machine.

The first incarnation of the state machine is shown in Figure 4.7. The *Idle* state is where PAPERS1 is idle, waiting for a new event from the PE; specifically the toggling of the P_S line. The *Settle* state is a time delay state used to give the incoming data lines time to settle, after the strobe was detected. In the transition from *Settle* to *Wait*, the barrier mask is latched, sending notification to the other state machines that this PE has arrived at the barrier. Also, the state of the P_S line is latched, so that the state machine knows what value to expect for the next barrier event. The *Wait* state is the barrier state, waiting for all the participating PEs to join the barrier. Once a barrier is satisfied, the state machines for all the participating PEs simultaneously transition from *Wait* to *Delay*. During this transition, the return data for the barrier is latched for each PE. The *Delay* state, like *Settle*, is used as a time delay to give the outgoing data lines time to stabilize. In the transition from *Delay* to *Idle*, the `P_RDY` line is toggled to tell the PE that the barrier is complete, and thus the data is ready. Also, the barrier mask is cleared to be ready for the next barrier event.

This state machine had a few flaws. The main flaw involved the specific timing requirements of the Centronics Parallel port. The final version of the state machine is shown in Figure 4.8. This state machine maintains the same basic structure as the original, but adds a separate Reset state, as well as expanding *Delay* into a sequence of states for better timing control. Another significant change was the switch of the `P_RDY` line from an edge sensitive to a level sensitive value. This new state machine was designed to run at 25 MHz, thus with a 40 ns clock period.

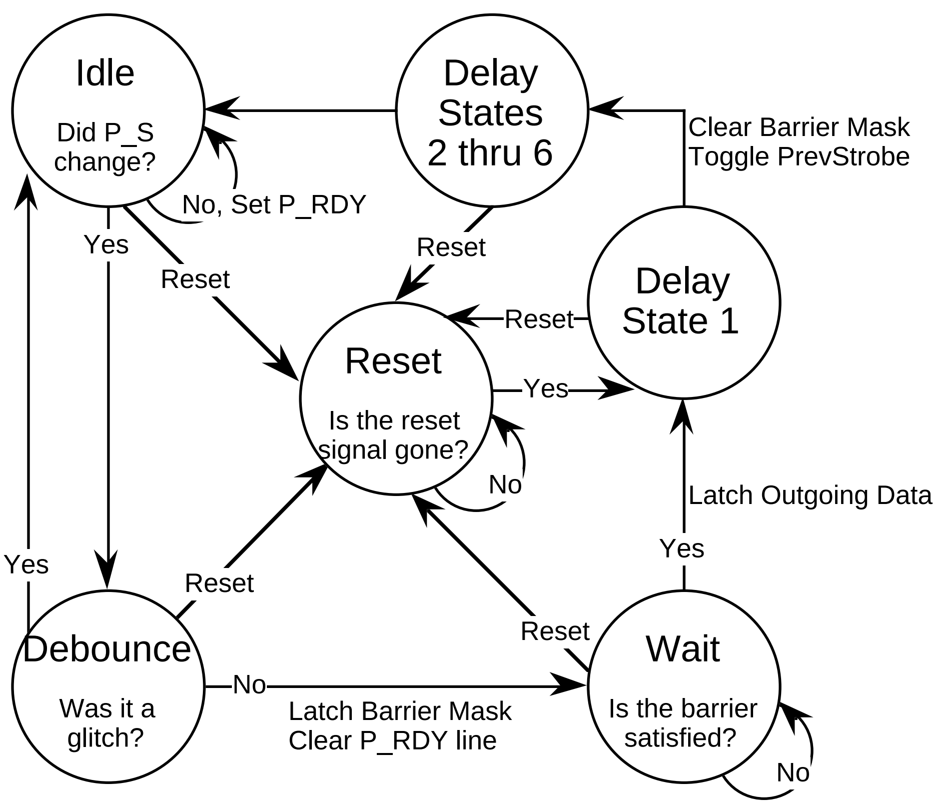


Fig. 4.8 The final PAPERS1 state machine.

This state machine again has an *Idle* state where it waits for the PE to initiate an operation. The *Idle* state is followed by a *Debounce* state that checks the P_S line a second time to verify that the PE did initiate an operation. If at the end of the *Debounce* state, the P_S line is back to the value stored in PRVSTRB (a 1-bit element inside the CE22V10 used to store the previous strobe state), the state machine returns to the *Idle* state. This gives the state machine some additional noise immunity, and therefore is able to recover from glitches on the P_S line which last less than 40 ns. The original state machine was unable to do this. Also, the *Debounce* state gives the incoming data lines 40 ns to settle after the strobe signal arrives.

During the transition from the *Debounce* state to the *Wait* state, the barrier mask is latched, telling the other state machines that this PE has arrived at the barrier. Also, the P_RDY line is cleared to tell the PE that this operation is not yet complete. In the *Wait* state, the system waits for all the participating PEs to arrive at the barrier. When the barrier is satisfied, all the participating state machines simultaneously transition to the first *Delay* state. During this transition, the result data is latched. After the first *Delay* state, the barrier mask is cleared and the PRVSTRB value is toggled to prepare for the next barrier. A total of six delay states are traversed before returning to the *Idle* state. After entering the *Idle* state, the P_RDY line is asserted to tell the PE that the operation is complete and the data is ready.

The total time delay from when the data is latched and when the P_RDY signal is sent is 7 states, or 280 ns. This 280 ns is required due to the design of the Centronics style Parallel ports. The Centronics specification actually states that on certain signal lines, transitions lasting less than 500 ns should be ignored [24]. As it turned out, the five input pins to the PC that all the current PAPERS prototypes use have some undesirable electrical properties. Specifically, rising and falling edges as seen by the PC have significantly different timing delays, with falling edges seen as much as 500 ns earlier than rising edges. Also, the specific timing delays vary from machine to machine, as well as for separate add-in Parallel Port cards. The specific timings generated by PAPERS1 work for the PCs tested.

### 4.4 PAPERS1 Software Library

A parallel processing architecture would not be complete without some support for creating programs. However, the PAPERS1 hardware was a prototype, thus software support was developed only to a level required for testing and evaluation. This section will give an overview of how this basic software works. The software support for PAPERS1 is a C library, found in Appendix A, that is linked with the user's program.

The software library for PAPERS1 was based on the libraries for the original PAPERS prototype [3], and the TTL_PAPERS prototype [22]. Some basic characteristics of the software library are:

- Direct User Space I/O operations for low latency communications
- Simplified communication model (basic data types, no buffers, etc.)
- Extensive use of GCC preprocessor macros for software engineering needs
- Minimal layering, usually direct use of basic hardware operations
- Support for a basic set of aggregate functions

These fundamental features of the software implementation yield a maintainable library with an easy to use interface that retains the low latency potential of the hardware.

The simplified communication model, mentioned in the above list, is that of transferring individual data objects as soon as they are produced. This is in contrast to message passing in which data objects are collected into a message, and then this larger block is transferred. To make use of these operations simple, each operation comes in several forms, one for each basic data type: 8, 16, 32, and 64 bit signed and unsigned integers, as well as 32 and 64 bit floating point numbers. As mentioned above, the software library supplies a basic set of aggregate communication functions, such as reductions, parallel prefix scans, and gathers. These functions are built upon the 4-bit wide *globalNAND* and *putget* communication functions that the PAPERS1 hardware directly supports.

The collection of reduction operations are split into two groups, those that are constructed from the *globalNAND* primitive, and those that are built from *putget*. Bitwise `reduceOr` and `reduceAnd` are directly implemented with sequences of 4-bit *globalNAND* operations, inverting the bits either when sending, or when receiving. The length of this sequence is directly determined by the size of the data type, 4 bits per transfer, and is independent of how many PEs are involved. As described in [21], `reduceMin` and `reduceMax` can also be performed in a fixed number of *globalNAND* operations independent of the number of PEs involved, in this case 2 bits of result per 4-bit *globalNAND*. However, since the PAPERS1 prototype can only connect at most four PEs, a recursive doubling tree reduction built from *putget* was used for `reduceMin` and `reduceMax`. This method is described below in conjunction with the rest of the reductions.

For reductions based on the minimum, maximum, add, and multiply operations, a recursive doubling tree reduction is used. This technique requires $\log_2(N)$ steps, where N is the number of PEs involved. At each step a permutation of partial results is done via *putget*. At the end of the sequence, each PE has a copy of the final answer. The tree reduction is somewhat complicated by the fact that, if the machine is partitioned, some processors might not be members of the current barrier group, in which case they do not participate in the reduction. Thus, the participating PEs must effectively be renumbered to determine their positions within the reduction tree. Fortunately, this renumbering can be performed locally on each PE based on its current barrier mask. Unfortunately, the recursive doubling part of the algorithm assumes N is a power of 2. Thus, for PAPERS1 with partitions that contain three PEs, this approach does not work directly. The PAPERS1 library has a modified recursive doubling algorithm for `reduceAdd` that deals with a partition where N is not a power of 2. If PAPERS1 were to be used in a non-prototype situation, this limitation in the software would be fixed for the rest of the reduction operations.

To perform a gather operation for N PEs, a sequence of N-1 *putget* operations are performed. Visualize the PEs linked in a ring, and each *putget* performs a shift clockwise of length *k*, where *k* is the iteration number from 1 to N-1. This performs more than a simple gather, since every PE simultaneously gets a copy of the result array. The parallel prefix scan operations shown in Appendix A for PAPERS1 have not been optimized for use in PAPERS1. Again, if PAPERS1 was to be used in a non-prototype situation, these routines would be optimized. Currently, the scan functions perform a gather, and then each PE computes its result based on the gathered array.

A vastly improved software library has been developed called AFAPI (Aggregate Function API) [23] which more formally defines a usable set of synchronous aggregate communication functions. This new library has been ported to several architectures [25] including shared memory SMP boxes. However, AFAPI has not been ported to PAPERS1 due to the limited availability additional PAPERS1 hardware units (only one has been built). Since a future port of AFAPI for PAPERS1 may occur, the development and debugging of the current PAPERS1 software library has ceased.

One final point to be made about the PAPERS1 software library is that for aggregate communication functions, the *putget* operation is a very natural building block. This comes about from its flexibility in performing inexpensive permutations and multi-casts. In other words, *putget* is ideal for implementing synchronous aggregate communication operations that require data to be moved in a choreographed way.

# 5. RESULTS

This thesis has presented the PPS as an analysis tool that revealed some limitations to the current approaches towards parallel processing. Also using the PPS analysis tool, some enhancements to a traditional MIMD machine have been suggested to produce a much more capable parallel processing machine. Some of these enhancements are support for aggregate communication operations presented in Chapter 3. The PAPERS1 prototype was presented in Chapter 4, and is the primary contribution of this thesis. This results chapter will discuss the performance of the PAPERS1 prototype system in terms of the PPS in section 5.1. Section 5.2 explores possible extensions to PAPERS1 that would be possible using more advanced technology such as VLSI. Section 5.3 discusses some of the scalability issues for future implementations.

## 5.1 Performance of PAPERS1

The PAPERS1 unit supplies a fully dynamic partitionable barrier, broadcast, and reduction network to a cluster of four PCs. This cluster is capable of executing as Dynamic Barrier MIMD (DBM), SIMD, and VLIW machines, with a fine-grain execution model for all modes. The cluster is thus able to cover all the regions of the PPS of those respective machine architectures, as well as new areas involving high coordinality, high multiformity and small grain size applications. These are rather strong claims, and deserve some detailed explanation.

### 5.1.1 Coverage of the Granularity Axis by PAPERS1

On the granularity axis of the PPS, the PAPERS1 design allows all basic barrier and communication operations to occur with very low latency. Each operation takes a minimum of two I/O port accesses per PE – one write, and one read. This means that for a PC with a raw port access time of one microsecond, all basic operations can theoretically occur in two microseconds, plus all the real-world delays of signal propagation, function call overhead, synchronization skew, etc. The end result is that with a cluster of 486DX33 PCs, the minimum latency for all basic operations is ~3 microseconds, with real-world achievable times of ~4 microseconds (including the function call overhead of accessing the unit through a communications software library). Table 5.1 presents some benchmarks of the PAPERS1 unit with four IBM ValuPoint 486DX33 systems running Linux 1.1.75 as the PEs. The times are in microseconds and include function-call overheads of using the PAPERS communication library routines. Each column is for a different size of integer or floating-point datum.

Table 5.1.
PAPERS1 communication times in microseconds.

| | 8-bit | 16-bit | 32-bit | 64-bit | Float | Double |
|---|---|---|---|---|---|---|
| `putget` | 8.5 | 15 | 27 | 62 | 29 | 63 |
| `gather` | 33 | 53 | 89 | 194 | 95 | 199 |
| `reduceAdd` | 20 | 33 | 57 | 128 | 63 | 131 |
| `reduceMul` | 21 | 34 | 58 | 132 | 63 | 130 |
| `reduceMin` | 20 | 33 | 57 | 127 | 63 | 131 |
| `reduceMax` | 20 | 33 | 57 | 127 | 63 | 131 |
| `reduceAnd` | 9 | 16 | 29 | 81 | - | - |
| `reduceOr` | 9 | 16 | 30 | 81 | - | - |

For more extensive benchmark numbers, one can look at [22] for additional communication operations on PAPERS1 and other architectures. For a quick comparison, Table 5.2 has some of the benchmark numbers from [22] for a variety of architectures – benchmarked with 4 PEs enabled. Note: the column labeled *wait* is the basic time to perform a barrier synchronization, which in the PAPERS and AFAPI [23] software libraries is invoked by the routine `p_wait`.

Table 5.2.
Communication time comparisons in microseconds.

| | `wait` | `waitbar` | `putget32` | `reduceOr32` |
|---|---|---|---|---|
| PAPERS1 | 3.1 | 3.2 | 27 | 30 |
| TTL_PAPERS | 2.5 | 6.3 | 216 | 59 |
| MasPar MP-1 (SIMD) | 0.1 | 9.4 | 44 | 17 |
| Paragon XP/S (MIMD) | 530 | - | 700 | 710 |
| PVM3 10 Mb/s Ethernet | 49,000 | - | 100,000 | 100,000 |

The PAPERS1, TTL_PAPERS, and PVM3 systems were the same set of four IBM ValuPoint 486DX33 systems running Linux 1.1.75 as the PEs. The MP-1 was the 16,384 PE SIMD commercial machine in use at Purdue University. The barrier synchronization time on the MP-1 is its basic clock period of 100 ns, since it is a true SIMD machine. The Paragon XP/S was the 140 PE MIMD commercial machine also in use at Purdue. It should be noted that PAPERS1 and TTL_PAPERS performance numbers are for a University built prototype connecting conventional (unmodified) PCs. With access to buses or ports closer to the CPU of each PE, considerable improvements in latency and bandwidth are foreseeable.

### 5.1.2 Coverage of the Coordinality Axis by PAPERS1

The coordinality axis of the PPS is covered by PAPERS1 through its collection of aggregate communication primitives. Basic barriers are supported to synchronize the PEs, allowing the static scheduling of shared resource use (such as an Ethernet or FDDI connecting the PEs). Reductions based on bit-wise logic operations (ORs, ANDs, etc.) are done 4 bits at a time, at roughly one bit of result per microsecond. Also, a multitude of operations such as permutations, multiple broadcasts, and bi-directional point-to-point communications are supported by the *putget* communications primitive. The *putget* communications primitive is a novel idea first introduced with the PAPERS1 prototype. Section 3.3 discussed the significance and implications of this novel communication style. With the low-latency aggregate communications supported by PAPERS1, a cluster of PCs attached to PAPERS1 becomes a tightly coupled system capable of high coordinality.

### 5.1.3 Coverage of the Multiformity Axis by PAPERS1

Finally, the multiformity axis must be examined. A PAPERS1 system consists of separate PCs that are the PEs of a MIMD system. The barrier and communications network of PAPERS1 is fully partitionable, where arbitrary sets of PEs can be grouped together, and disjoint groups do not interfere in any way. As PEs split off and rejoin barrier groups, the communication facilities also partition. This partitioning is dynamically controlled by the PEs. All the above leaves intact the high multiformity support inherent in a MIMD system.

Thus, a PAPERS1 system fulfills the promise (if on a small scale) of the general purpose parallel processing system introduced in this thesis.

**5.2 PAPERS1 Extensions**

The design of PAPERS1 was limited by the technology available for building the prototype. By removing these design constraints, a DBM implementation could fulfill more of its potential. The following is a list of some of the primary limitations imposed by the technology available/selected for the PAPERS1 prototype:

1) Cabling (quantity, quality, and direction of electrical signals from/to the PE)
2) Low logic complexity (gate and pin counts of 22V10 PALs)
3) Limited data storage (only 10 bits of storage per PAL)
4) Wiring density (wire-wrap vs. PCB vs. VLSI)

This section discusses some of the issues that arose from a design project that removed or reduced these design constraints.

A superset of the PAPERS1 architecture was used as the basis for a "paper" design project in the "MOS VLSI Design" course (EE 559) in the Spring of 1995. This semester long project was completed by the author, and his three partners: Alex Andrews, Rebecca Boyd, and David Ward. This project explored several design issues for scaling a DBM, such as PAPERS1, to larger numbers of processors. The project also demonstrated the potential speed of VLSI implementations of the aggregate communication operations described in Chapter 3. In the remainder of this discussion, the chip that was designed will be called the DBM16.

The DBM16 chip contained sixteen interconnected Barrier Processing Units (BPU). Each BPU had a functionality that was a superset of the PAPERS1 BPUs described section 4.2.2. The features of the DBM16 design are delineated in the following list, with the first 6 items roughly corresponding to those of PAPERS1 presented in section 4.1:

1) Full dynamic partitionability (a DBM implementation)
2) 2 cycle aggregate functions
3) 8-bit *globalOR* & *globalXOR* communication primitives
4) 8-bit *putget* communication primitive
5) Interrupt/Priority Barriers
6) BPU (Barrier Processing Unit) ID number
7) 8-bit *match* & *matchCount* communication primitives (see section 3.5)
8) 8-bit *vote* & *voteCount* communication primitives (see section 3.6)
9) An extensive set of barrier mask operations (described below)

The DBM16 design was not implemented, and thus its performance estimates are approximations derived from circuit simulations. The chip technology selected for the course, 1.2 micron CMOS with 2 metal layers, significantly limited the placement of the BPUs and their interconnect wires. With more metal interconnect layers available in recent CMOS fabrication processes, the final chip area of 484 $mm^2$ could be drastically reduced. The final chip design contained roughly 236,000 transistors housed in a 310 pin package with 240 pins used for I/O. The remainder of this section will discuss what was learned from this design project.

There were three scaling issues that were dealt with in the DBM16 design: a wider data path, connections for up to sixteen PEs, and special facilities for manipulating barrier bit masks. The PAPERS1 design has a four bit data path that was derived primarily from the available signal lines in the PC's Centronics I/O interface. Since the DBM16 design was not limited to using that interface, a wider data path of 8 bits was selected to improve performance. Since this was a single chip design, pin limitations prevented a much wider data path. When scaling a DBM to many PEs, pin limitations will probably still prevent the data path from being much wider, even with multi-chip implementations. Due to scaling to sixteen PEs, the internal wiring complexity of the DBM16 was high, but still manageable. However, it was the view of the DBM16 design team that to scale it to more than 16 PEs would probably require a different approach. This wiring complexity issue is essentially the same issue faced by a crossbar network, that the number of wires grows as $N^2$ with N PEs.

The primary scaling issue that was dealt with in the DBM16 was that of barrier bit mask manipulations. For a DBM (or even an SBM) the identity of the PEs in a particular barrier group is maintained by a barrier bit mask, where the mask is N bits wide when there are N PEs. The PAPERS1 prototype had dedicated communication lines that transmitted a PE's barrier bit mask to it's BPU. This allowed the PE to do all the bit mask manipulation, storage, and retrieval operations. However, for larger DBMs, these communication lines become excessive, and do not contribute to the available bandwidth for communication of application data. Also, as a DBM grows to more than 32 or 64 PEs, the ability of an individual PE to efficiently manipulate a barrier bit mask becomes compromised by the data width of the PE's CPU. To solve these issues, the BPU must be able to deal with the manipulation, storage, and retrieval of barrier bit masks.

In the DBM16 design, each BPU is able to construct a new bit mask from the result of any of several operations, including *waitbar*, *match*, and *vote.* This allows the DBM to quickly subdivide the machine into disjoint groups of PEs to work on a problem's independent tasks. The VLSI circuitry to do these barrier mask manipulations is very dense, and fairly simple. Prior to replacing the current barrier mask, it is usually prudent to save the current mask someplace, thus each BPU had its own 256 bit RAM arranged as a stack. In normal operation, where partition events correspond to `switch` or `if-then-else` statements of a SPMD program, new barrier groups will form subsets of the previously active group, and to return to the previous partitioning, only a stack pop operation is required. Other logical arrangements (a register file for example) of this storage might prove useful, but a stack was simple and sufficient for the task. An example of a similar concept (and usage) is the enable bit stack for each of the PEs in a SIMD machine such as the MasPar MP-1.

The ability for the BPU to handle most aspects of the barrier bit masks in a DBM design reduces one of the problems associated with scaling a DBM. However, if one has been following this closely, it appears that for truly large DBM systems, this barrier bit mask storage can become excessively large. As described above, each BPU would have $N^2$ bits of RAM, and guess what, there are N BPUs...that multiplies out to an order $N^3$ scaling problem. For a 1024 PE system, this style DBM would need 128 megabytes of storage just for barrier bit masks. Luckily there is another way suggested by the similarity to an enable bit stack. An approach for using activity counters instead of a bit stack in SIMD machines is presented in [26]. With the appropriate rules on nesting barrier bit masks (as naturally enforced by `switch` or `if-then-else` statements of a SPMD program) this scheme can also be used for barrier bit mask stacks. Thus the storage problem can be reduced to $N^2 \log_2 N$, thus a 1024 PE DBM would only need 1.25 megabytes of storage for barrier bit masks. This approach was considered for the DBM16 design, but was not chosen because of the added design complexity and only a factor of four storage savings for a 16 PE design.

The DBM16 design also demonstrated the performance possibilities of a VLSI implementation of the aggregate communication operations described in Chapter 3. Simulations of the design showed that the DBM16 chip could run at 200 MHz, with a very conservative 1.2 micron CMOS process. Each operation took from 5 to 9 clock cycles, including 2 clocks for the I/O interface. Thus any of the operations from Chapter 3 could be performed in 45 ns or less, which is faster than a reference to conventional DRAM! In addition to all the aggregate operations found in Chapter 3, the DBM16 design included a *globalXOR*, as well as the operations needed to manipulate the barrier bit mask stack. To simplify the DBM16 design, a *globalOR* was substituted for the *globalNAND* operation, which has already been shown to be interchangeable.

Many more design and implementation issues remain for building a large DBM system. The DBM16 design was a good start at assessing where future work should be applied. The following section will discuss some of these future directions.

**5.3 Scaling Issues**

When trying to scale PAPERS1 to more than four PEs, several problems arise. The following list is not comprehensive, but is useful to get a feel for where future work should be focused:

1) The $N^2$ wire complexity for the BPU interconnect
2) Barrier Mask manipulations
3) The non-hierarchical nature of a central hub
4) Distributing a Global Clock to the BPUs
5) Maintaining the high speed of Aggregate Communication Operations
6) The communications interface between the PEs and the PAPERS unit
7) The effective Data Path width

Different design decisions would have to be made to build larger systems, as was discussed in section 5.2 in regards to the DBM16 design. The DBM16 design directly dealt with items 2, 5, 6, and 7. Since it was only for sixteen PEs it did not hit the limits imposed by items 1, 3, and 4.

Of those remaining issues, the primary concern is the $N^2$ wiring complexity of the PAPERS1 and DBM16 designs. A secondary concern is the non-hierarchical nature of a central hub, forcing N cables to arrive at a central location, making pin count and density limits a factor. The global clock problem has been explored (and solved) by others. For example, the IBM SP2 switch has a distributed global clock. These problems can be avoided by various engineering compromises.

An example compromise is discussed in [15], where the design implements an SBM with the *globalNAND* communication primitive. The PAPERS 951201 scales in a hierarchical manner as a four-ary tree. Since it does not implement the DBM model, it does not need $N^2$ wiring, nor does it need a distributed global clock in the traditional sense. Thus it scales quite well, but at the loss of the DBM and the *putget* operation when compared to PAPERS1.

It should be noted that the time for basic operations in the PAPERS1 unit and the DBM16 is O(1) while their wiring complexity is $O(N^2)$. There should be engineering compromises where the complexity of the hardware is reduced at the cost of increased time for basic operations. Various theoretical designs have been explored to scale the functionality of PAPERS1 to larger numbers of PEs, but are beyond the scope of this paper and are left for future work.

# 6. CONCLUSIONS AND FUTURE WORK

This thesis has presented the Parallel Processing Space as a framework that shows the relevance of the PAPERS research towards building better parallel processing machines through the addition of synchronous aggregate communications. The PAPERS prototypes are not just inexpensive "add-on" communication networks – they add fundamentally new functionality to a MIMD machine, allowing said machine to speed up the execution of a wider range of parallel programs.

The PAPERS1 prototype has demonstrated that a MIMD machine can adopt the good communication features of a SIMD machine while maintaining PE autonomy. By combining barrier synchronization with communication, a MIMD machine can perform many communication operations much more efficiently. For any MIMD machine to work efficiently on a variety of problems, it is important to remove unnecessary communication overheads, be they from extra software/hardware layers, or from missing functionality that must be emulated, such as barrier synchronization hardware.

The new communication construct called *putget* was developed by the author in conjunction with Jeff Sponaugle. This new model of communication is different from either message passing or shared memory communication schemes. A *putget* performs very well on common communication patterns such as multi-casts and permutations. It is also well suited to compiler generated communications, and as a building block for synchronous aggregate communications in general. By disallowing the expression of write conflicts, the *putget* model allows the system designer to get most of the benefits from a crossbar network, yet avoid the complexity of deep buffers/queues, collision detection hardware, and arbitration hardware. If a program's communication events can be arranged in synchronous groups, the *putget* should be a win over other communication methods.

There is an obvious need for future research. The viability of a DBM, or a hybrid SBM/DBM, for large systems is an open question that should be answered. It has already been demonstrated that an SBM is viable for very large systems – the Cray T3D has a large subset of an SBM capability. The combined use of synchronous aggregate communications networks with more conventional networks needs further study. The compiler technology for mapping real programs to the proposed DBM architecture also needs to be explored and developed.

# REFERENCES

[1] M.J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948-960, 1972

[2] George S. Almasi and Alan Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., 1994

[3] H.G. Dietz, T. Muhammad, J.B. Sponaugle, and T. Mattox, "PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization," Purdue University School of Electrical Engineering, Tech. Rep. TR-EE 94-11, March 1994.

[4] MasPar Computer Corporation, *MasPar Assembly Language (MPAS) Reference Manual, Revision 4/3/90*, Sunnyvale, California, April 1990.

[5] T.B. Berg and H.J. Siegel, "Instruction Execution Trade-Offs for SIMD vs. MIMD vs. mixed-mode parallelism", in *5th International Parallel Processing Symposium*, April 1991, pp. 301-308.

[6] S.A. Fineberg, T.L. Casavant, and H.J. Siegel, "Experimental Analysis of a Mixed-Mode Parallel Architecture Using Bitonic Sequence Sorting," *Journal of Parallel and Distributed Computing*, vol. 11, no. 3, pp. 239-251, March 1991.

[7] G. Saghi, H.J. Siegel, and J.L. Gray, "Predicting Performance and Selecting Modes of Parallelism: A Case Study Using Cyclic Reduction on Three Parallel Machines," *Journal of Parallel and Distributed Computing*, vol. 19, no. 3, pp. 219-233, November 1993.

[8] J.J.E. So, R. Janardhan, T.J. Downar, and H.J. Siegel, "Mapping the Preconditioned Conjugate Gradient Algorithm for Neutron Diffusion Applications Onto Parallel Machines", in *1996 International Conference on Parallel Processing*, vol. II, Aug. 1996, pp. 1-10.

[9] J.B. Armstrong, M.A. Nichols, H.J. Siegel, and K.H. Casey, "Image Correlation: A Case Study to Examine SIMD/MIMD Trade-Offs for Scalable Parallel Algorithms", in *1994 International Conference on Parallel Processing*, vol. I, Aug. 1994, pp. 241-245.

[10] H.J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Computers*, vol. C-30, no. 12, pp. 934-947, December 1981.

[11] H.J. Siegel, T. Schwederski, W.G. Nation, J.B. Armstrong, L. Wang, J.T. Kuehn, R. Gupta, M.D. Allemang, D.G. Meyer, and D.W. Watson, "The Design and Prototyping of the PASM Reconfigurable Parallel Processing System," *Parallel Computing: Paradigms and Applications*, edited by A.Y. Zomaya, International Thomson Computer Press, London, UK, pp. 78-114, 1996.

[12] H.G. Dietz, T. Schwederski, M.T. O'Keefe, and A. Zaafrani, "Static Synchronization Beyond VLIW," in *Supercomputing 1989*, IEEE Computer Society Press, November 1989, pp. 416-425.

[13] H.G. Dietz and T.M. Chung, "Genetic Scheduling of Router Operations for the MasPar MP-1/MP-2," January 1995. http://dynamo.ecn.purdue.edu/~hankd/CARP/GENROUTE/paper.html

[14] Eric A. Brewer and Bradley C. Kuszmaul, "How to Get Good Performance from the CM-5 Data Network," in *8th International Parallel Processing Symposium*, Cancun, Mexico, April 1994, pp. 858-867.

[15] H.G. Dietz, R. Hoare, and T. Mattox, "A Fine-Grain Parallel Architecture Based On Barrier Synchronization," in *1996 International Conference on Parallel Processing*, vol. I, Aug. 1996, pp. 247-250.

[16] Hubertus Franke, et. al. "MPI-F: An Efficient Implementation of MPI on IBM-SP1", in *1994 International Conference on Parallel Processing*, vol. III, Aug. 1994, pp. 197-201.

[17] Ronald Mraz, "Reducing the Variance of Point to Point Transfers in the IBM 9076 Parallel Computer," in *Supercomputing '94*, IEEE Computer Society Press, November 1994, pp. 620-629.

[18] T. Blank, "The MasPar MP-1 Architecture," in *35th IEEE Computer Society International Conference* (COMPCON), February 1990, pp. 20-24.

[19] M. O'Keefe, *Barrier MIMD Architecture: Design And Compilation*, Ph.D. Thesis, School of Electrical Engineering, Purdue University, August 1990.

[20] T. Muhammad, *Hardware Barrier Synchronization For A Cluster Of Personal Computers*, Master's Thesis, School of Electrical Engineering, Purdue University, May 1995.

[21] R. Hoare, H. Dietz, T. Mattox, and S. Kim, "Bitwise Aggregate Networks," in *Eighth IEEE Symposium on Parallel and Distributed Processing*, October 1996, pp. 306-313.

[22] H.G. Dietz, T.M. Chung, and T.I. Mattox. "A Parallel Processing Support Library Based On Synchronized Aggregate Communication," *Languages and Compilers for Parallel Computing*, edited by C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Springer-Verlag, New York, New York, 1996, pp. 254-268.

[23] H.G. Dietz, "AFAPI: Aggregate Function Application Program Interface," May 1997, http://garage.ecn.purdue.edu/~papers/AFAPI/Index.html

[24] Jan Axelson, *Parallel Port Complete*, Lakeview Research, Distributed by International Thomson Publishing, ISBN# 0-9650819-1-5, 1996. http://www.lvr.com/parport.htm

[25] H.G. Dietz, T.I. Mattox, and G. Krishnamurthy. "The Aggregate Function API: It's Not Just For PAPERS Anymore," to appear in *1997 Workshop on Languages and Compilers for Parallel Computing*, University of Minnesota, Minneapolis, MN, August 1997.

[26] R. Keryell and N. Paris, "Activity Counter: New Optimization for the Dynamic Scheduling of SIMD Control Flow," in *1993 International Conference on Parallel Processing*, vol. II, Aug. 1993, pp. 184-187.

# Appendix A: PAPERS1 Library Source

The following source code files are for the PAPERS1 prototype hardware only. Since this was a prototype system, extensive debugging and testing has not been performed on the software. There are known errors in the reduction routines for partitions with 3 PEs. Development of this software library has ceased since the introduction of AFAPI. If and when a more reproducible DBM PAPERS hardware design is introduced, the AFAPI software library will be ported to support such a system. Please see the AFAPI home page at http://garage.ecn.purdue.edu/~papers/AFAPI/ for future developments.

## A.1 Source listing of "papers.h"

```
/*       papers.h  Support for basic functions using PAPERS. */
#ifndef _PAPERS_H
#define _PAPERS_H
#include "types.h"                          /* PAPERS data types */
#define  NPROC    4                         /* Number of PEs */
#define  NXPROC   2                         /* PEs in x dimension */
#define  NYPROC   2                         /* PEs in y dimension */
#define  LNPROC   2                         /* Bits needed to encode NPROC */
#define  LNXPROC  1                         /* Bits needed to encode NXPROC */
#define  LNYPROC  1                         /* Bits needed to encode NYPROC */
#define  ALL_MASK ((1 << NPROC)-1)  /* All PEs barrier mask */

extern int iproc;                  /* PAPERS PE number */
extern int ixproc;                          /* X coordinate */
extern int iyproc;                          /* Y coordinate */
extern barrier our_mask;           /* Our barrier mask */
#define IPROC     ((const int) iproc)                /* Protected iproc */
#define  IXPROC   ((const int) ixproc)               /* Protected ixproc */
#define  IYPROC   ((const int) iyproc)               /* Protected iyproc */
#define  OUR_MASK ((const barrier) our_mask) /* Protected our_mask */

/*       PAPERS PE Mapping data struct  */
typedef struct {
         barrier mask;
         int partition_size;
         int vir2phy[NPROC];
         int phy2vir[NPROC];
} p_mapping;

extern void p_source(register int src);
extern void p_enqueue(register barrier mask);
extern void p_wait(void);
extern barrier p_waitvec(register int flag);
extern int p_any(int flag);
extern int p_all(int flag);
extern void p_init(void);
extern void p_error(register char *s);
extern void p_exit(register int ecode);
extern unsigned char (*p_signalHandler)(unsigned char);
extern unsigned char p_die(unsigned char x);
extern void p_signal(unsigned char (*fnptr)(unsigned char));
#endif
```

## A.2 Source listing of "papers.c"

```
/*      papers.c

        Support for basic functions using PAPERS.  (version for PAPERS1)
*/


#include <stdio.h>
#include <stdlib.h>
#include "version.h"


/*      Include the internal support header, which includes the inline
        library, but set the DO_NOT_INLINE flag to force the library to
        be expanded as ordinary function definitions rather than inline
        expansions.  This is the only place where the flag is set, and
        simply ensures that linkable versions of the functions will be
        created in papers.o.
*/
#define DO_NOT_INLINE
#include "intern.h"


#ifdef  P_USETONE
int p_toneDivider = 0;              /* Freq. divide counter for tone */
#endif

#ifdef ZOOM
#include <unistd.h>
#endif

#define CHECKIN

barrier last_mask = 0;              /* Last barrier mask */
p_mapping last_map;                 /* Last PE mapping */
int _last_src;                      /* Last source (pre-shifted) */
int last_src;                       /* Last source */
portdata last_s;            /* Last strobe bit value */
int iproc = -1;                     /* PAPERS PE number */
barrier our_mask;           /* Our barrier mask */
int isrc = -1;                      /* Our PAPERS PE src field value */
int ixproc;                         /* X coordinate */
int iyproc;                         /* Y coordinate */
unsigned char p_irq;                /* Bit used for interrupt request */


int p_read_ct = 0;
int p_read_dble = 0;
int i0Hct = 0, i0Lct = 0;
int i1Hct = 0, i1Lct = 0;
int i2Hct = 0, i2Lct = 0;
int i3Hct = 0, i3Lct = 0;
int rdyHct = 0, rdyLct = 0;
int sHct = 0, sLct = 0;

static int
p_hw_startup(void)
{
        register int tries = 0;
        register portdata x;
```

```
        /* Enqueue global barrier */
        p_enqueue(ALL_MASK);

#ifndef CHECKIN
        p_wait();
#else
        /* Do a special barrier, expecting all data bits to be 0. */
        P_MODE(ALL_MASK ^ OUR_MASK); /* send an IRQ */
        P_MODE(ALL_MASK ^ OUR_MASK);
        P_MODE(ALL_MASK ^ OUR_MASK);
        P_MODE(ALL_MASK ^ OUR_MASK);
        _last_src = ISRC;
        do {
#ifdef DOUBELO
                P_OUT(ISRC | last_s | ALL_MASK);
#endif
                P_OUT(ISRC | (last_s ^= P_S) | ALL_MASK);
#ifdef RDY_TOGGLES
                if (!last_s) {
                        while ((x = P_IN()) & P_RDY);
                } else {
#endif
                        while (!((x = P_IN()) & P_RDY));
#ifdef RDY_TOGGLES
                }
#endif
#ifdef DOUBELI
                x = P_IN();
#endif
                tries++;
        } while (IToMask(x) != 0);
        P_MODE(ALL_MASK);
        P_MODE(ALL_MASK);
        P_MODE(ALL_MASK);
        P_MODE(ALL_MASK);
#endif
        return(tries);
}

int
p_checkenv(void)
{
/* Check that hardware wiring matches IPROC... */
        register int i = p_hw_iproc();

        if (i == -1) {
                fprintf(stderr,
"PAPERS: The hardware is not responding properly, check the cable & power.\n");
                return(-1);
        } else if (i != IPROC) {
                fprintf(stderr,
"PAPERS:  software environment IPROC=%d, but hardware is wired as PE%d\n",
                        IPROC,
                        i);
                return(-1);
        }
        return(0);
}

void
p_initenv(void)
{
        /* Initialize ONLY the environment-sensitive info... */
```

```
        register char *p;

        /* Check environment variables */
        p = getenv("PAPERS");
        if ((p == NULL) || strcmp(P_PAPERS, p)) {
                fprintf(stderr,
                        "PAPERS PE:  need %s hardware\n",
                        P_PAPERS);
                exit(1);
        }
        p = getenv("PAPERS_HW_version");
        if ((p == NULL) || (P_HW_VERSION != atoi(p))) {
                fprintf(stderr,
                        "PAPERS PE:  need PAPERS hardware version %d\n",
                        P_HW_VERSION);
                exit(1);
        }
        p = getenv("PAPERS_HW_revision");
        if ((p == NULL) || (P_HW_REVISION != atoi(p))) {
                fprintf(stderr,
                        "PAPERS PE:  need PAPERS hardware revision %d\n",
                        P_HW_REVISION);
                exit(1);
        }

        p = getenv("PAPERS_SW_version");
        if ((p == NULL) || (P_SW_VERSION != atoi(p))) {
                fprintf(stderr,
                        "PAPERS PE:  need PAPERS software version %d\n",
                        P_SW_VERSION);
                exit(1);
        }
        p = getenv("PAPERS_SW_revision");
        if ((p == NULL) || (P_SW_REVISION != atoi(p))) {
                fprintf(stderr,
                        "PAPERS PE:  need PAPERS software revision %d\n",
                        P_SW_REVISION);
                exit(1);
        }
        p = getenv("PAPERS_NPROC");
        if ((p != NULL) && (NPROC != atoi(p))) {
                fprintf(stderr,
                        "PAPERS PE:  need %d PEs, environment has %d\n",
                        NPROC, atoi(p));
                exit(1);
        }
        p = getenv("PAPERS_PORTBASE");
        if ((p != NULL) && (P_PORTBASE != atoi(p))) {
                fprintf(stderr,
                        "PAPERS PE:  compiled for port at %d, environment has
%d\n",
                        P_PORTBASE, atoi(p));
                exit(1);
        }

        p = getenv("PAPERS_IPROC");
        iproc = ((p == NULL) ? -1 : atoi(p));
        switch (iproc) {
        case 0:
        case 1:
        case 2:
        case 3:
                break;
```

```
        default:
                fprintf(stderr,
                        "PAPERS PE:  illegal PAPERS PE number (%d)\n",
                        iproc);
                exit(1);
        }
        our_mask = (1 << IPROC);            /* Our barrier mask */
        _p_construct_mapping(&last_map, ALL_MASK);   /* initialize the PE mapping
info */
        isrc = (IPROC << 4);                         /* Our PE src field value */
        ixproc = (IPROC & (NXPROC - 1));    /* X coordinate */
        iyproc = (IPROC >> LNXPROC);                 /* Y coordinate */

        /* Get permission to access the port */
        P_PERM();
        P_TONEPERM();
}

void
p_init(void)
{
        register portdata x;

        /* PAPERS initialization routine...
           this ****ASSUMES**** that a proper p_exit() call
           was used to terminate the previous parallel code.
        */

        /* First, initialize environment info... */
        p_initenv();

        /* Second, verify environment with the PAPERS1 Reset command... */
        if (p_checkenv() == -1) {
                fprintf(stderr, "p_init() failed, exiting.\n");
                exit(1);
        }

        /* Enqueue global barrier */
        p_enqueue(ALL_MASK);

        p_hw_startup();

#ifdef ZOOM
        nice(-20);
#endif
}

void
p_error(register char *s)
{
        /* Print error message and die... */
        fprintf(stderr,
                "\nPAPERS PE%d:  %s\n",
                IPROC,
                s);

        /* Cause everybody to die... */
        p_exit(1);
        exit(1);
}

void
p_exit(register int ecode)
```

```
{
        register portdata x, y;
        long abort = 0;

        /* Exit happened...  send everyone an interrupt and set our
           data bit to indicate that this is an "exit special barrier"
           Note: this does not actually exit the user program.
        */

        /* Enqueue global barrier */
        /* p_enqueue(ALL_MASK); */
        P_MODE(ALL_MASK ^ OUR_MASK); /* send an IRQ */
        P_MODE(ALL_MASK ^ OUR_MASK);

        /* Do a special barrier. */
#ifdef DOUBLEO
        P_OUT(ISRC | last_s | (ALL_MASK ^ OUR_MASK));
#endif
        P_OUT(ISRC | (last_s ^= P_S) | (ALL_MASK ^ OUR_MASK));

        /* Do we really want to wait for the others to join us at the exit? */
        y = 0xff;
        while (((x = P_IN() & P_IN_MASK) & P_RDY) == 0) {
                if (y != x) {
                        abort = 0;
                        y = x;
                } else if (++abort > 500000) {
/*
fprintf(stderr, "Aborted wait in p_exit after %ld P_IN()s == 0x%x!\n", abort, x);
*/
                        x = p_hw_reset();
                        break;
                }
        }

        P_MODE(ALL_MASK);
        P_MODE(ALL_MASK);

        /* Send the exit status? */
        /* MORE HERE */
}

/*      The following code is for handling a signal from the PAPERS unit.
        We don't actually know who caused the interrupt, but by default we
        just die...  so it doesn't really matter who caused the problem.
        Alternatively, the signal could be used to transfer execution to a
        different program segment and then to send other info about the
        cause of the interrupt and the action desired.  In any case, the
        signal does not even have to interfere with normal PAPERS operations
        in progress, since the return value from the signal handler is used
        as the return value for the P_IN() that caught the interrupt....
*/

unsigned char
p_die(unsigned char x)
{
        fprintf(stderr,
                "PAPERS interrupt detected (0x%x) -- parallel job killed\007\n",
                x);

        /* Indicate that parallel program terminated abnormally */
        P_MODE(last_mask & ~OUR_MASK); /* send an IRQ to clear the barrier */
        P_MODE(last_mask & ~OUR_MASK);
```

```
        p_exit(1);
        exit(1);

        /* To make GCC happy... */
        return(x);
}

unsigned char (*p_signalHandler)(unsigned char) = p_die;

void
p_signal(unsigned char (*fnptr)(unsigned char))
{
        p_signalHandler = fnptr;
}

portdata
p_hw_reset()
{
        /* Do a hardware reset... The following
           code is a bit paranoid, but should be safer & harmless.
           The extra code is marked as such....
        */
        P_MODE(0);        /* Begin reset sequence, send IRQ only to self */
        P_MODE(0);        /* delay */
        P_MODE(0);        /* delay */
        P_MODE(0);        /* delay */
        P_OUT(0);                  /* Set STROBE = 0, force reset to occur */
        P_OUT(0);                  /* delay */
        P_OUT(0);                  /* delay */
        P_OUT(0);                  /* delay */
        P_MODE(ALL_MASK); /* remove IRQ and exit the reset state */
        P_MODE(ALL_MASK); /* delay */
        P_MODE(ALL_MASK); /* delay */
        P_MODE(ALL_MASK); /* delay */
/*      fprintf(stderr, "P_IN(): 0x%0x\n", P_IN()); */
        last_s = 0;

#ifdef RDY_TOGGLES
        return(P_IN() & P_IN_MASK); FIX ME
#else
        return((P_IN() & P_IN_MASK) & ~P_RDY);
#endif
}

int
p_hw_iproc(void)
{
        register portdata x;

        /* Do a hardware reset... & request PE number. */
        x = p_hw_reset();

        switch (x) {
        case P_I0:        return(0);
        case P_I1:        return(1);
        case P_I2:        return(2);
        case P_I3:        return(3);
        default: /* Failure... return bad PE number */
                          fprintf(stderr, "p_hw_iproc got P_IN of %x!\n", x);
                          return(-1);
        }
}
```

## A.3 Source listing of "inline.h"

```
/*	inline.h

	This file contains all the basic functions that would normally
	be inlined.  Notice that inlining can be explicitly overridden
	by defining the symbol DO_NOT_INLINE; this feature is used to
	make a non-inlined version of the library in papers.c, so that
	ordinary versions of the routines can be accessed by linking.
*/

/*	Descriptions of the inb and outb instructions....
	These could have come from /usr/src/linux/include/asm/io.h,
	but we want things to stand alone, so, here they are....
*/


#ifndef DO_NOT_INLINE
extern inline
#endif
unsigned int
inb(unsigned short port)
{
	unsigned char _v;
#ifndef NO_ASM
__asm__ __volatile__ ("inb %w1,%b0"
		:"=a" (_v)
		:"d" (port), "0" (0));
#else
	_v = 0;
#endif
	return _v;
}


#ifndef DO_NOT_INLINE
extern inline
#endif
void
outb(unsigned char value,
unsigned short port)
{
#ifndef NO_ASM
__asm__ __volatile__ ("outb %b0,%w1"
		:/* no outputs */
		:"a" (value), "d" (port));
#endif
}

#ifndef DO_NOT_INLINE
extern inline
#endif
void
_p_source(register int src)
{
	_last_src = (last_src = src) << 4;
}

#ifndef DO_NOT_INLINE
extern inline
#endif
void
```

```
p_source(register int src)
{
        /* Only legal if source is a valid PE number and that PE is
           actually part of our barrier group.  Note that this implies
           that when both the barrier mask and source change, the
           source must change last.  In fact, changing the barrier
           mask IMPLIES setting the source to IPROC.
        */
        if (ISNOTPE(src) || !((1 << src) & last_mask)) {
                p_error("illegal PE for PAPERS source");
        }

        _p_source(src);
}

#ifndef DO_NOT_INLINE
extern inline
#endif
void
p_enqueue(register barrier mask)
{
        /* If appropriate, enqueue the new barrier pattern given by
           mask.  Note that mask represents PEk by bit k -- not quite
           the way that the port hardware works.
        */

        if (mask & ~ALL_MASK) {
                /* Bits are on for nonexistant PEs */
                p_error("enqueue of barrier containing nonexistant PE");
        }

        /* Is mask different from what we had? */
        if ((mask != last_mask) /* && (mask & ~OUR_MASK) */) {
                /* Enqueue mask with PAPERS */
                P_MODE(mask); /* Should we force IRQ not to be indicated??? */
                P_MODE(mask);
        }

        /* Update internal info */
        last_mask = mask;

        /* Likewise, set source to IPROC just in case.... */
        p_source(IPROC);
}


#ifndef DO_NOT_INLINE
extern inline
#endif
void
_p_wait()
{
        register portdata out;
#ifdef DOUBLEI
        register portdata y;
#endif

        /* Do a sync with the current mask, selecting the gathered
           bits as the input so that we can check for an interrupt,
           and oh yes, also toggle the strobe....
        */
        P_OUT(out = (ISRC | (last_s ^= P_S)));
```

```
#ifdef RDY_TOGGLES
        /* The P_RDY signal should match the strobe...  this is coded
           as two separate loops to cut polling overhead in waiting for
           a barrier....
        */
        if (!last_s) {
                while (P_IN() & P_RDY); FIX ME
        } else {
#endif
        if (!(P_IN() & P_RDY)) {
                if (!(P_IN() & P_RDY)) {
#ifdef P_CHKINT
                        register int ct = P_CHKINT;
                        register portdata x;
#endif
                        do {
#ifdef P_CHKINT
                                if (ct-- == 0) {
                                        P_MODE(OUR_MASK);
                                        P_MODE(OUR_MASK);
                                        x = P_IN();
                                        P_MODE(last_mask);
                                        P_MODE(last_mask);
                                        if (!(x & P_RDY)) {
                                                if (((*p_signalHandler)(x))
                                                    & P_RDY) break;
                                        }
                                        ct = P_CHKINT;
                                }
#endif
                        } while (!(P_IN() & P_RDY));
                }
        }
#ifdef RDY_TOGGLES
        }
#endif

        /* Toggle speaker bit */
        p_toneToggle();

}


#ifndef DO_NOT_INLINE
extern inline
#endif
void
p_wait()
{
        /* Simple barrier synchronization on current barrier.
           Nothing to do unless we are not alone...
        */
        if (last_mask & ~OUR_MASK) {
                _p_wait();
        }
}


#ifndef DO_NOT_INLINE
extern inline
#endif
portdata
_p_waitvec(register portdata b)
```

```
{
        register portdata x, out;
#ifdef DOUBLEI
        register portdata y;
#endif

        /* Do a sync with the current mask, selecting the gathered
           bits as the input sending the four bits given in b,
           and also toggle the strobe....
        */
#ifdef DOUBLEO
        P_OUT(b | ISRC | (last_s));
#endif
        P_OUT(out = (b | ISRC | (last_s ^= P_S)));

#ifdef RDY_TOGGLES
        /* The P_RDY signal should match the strobe...  this is coded
           as two separate loops to cut polling overhead in waiting for
           a barrier....
        */
        if (!last_s) {
                while ((x = P_IN()) & P_RDY);  FIX ME
        } else {
#endif
        if (!((x = P_IN()) & P_RDY)) {
                if (!((x = P_IN()) & P_RDY)) {
#ifdef P_CHKINT
                        register int ct = P_CHKINT;
#endif
                        do {
#ifdef P_CHKINT
                                if (ct-- == 0) {
                                        P_MODE(OUR_MASK);
                                        P_MODE(OUR_MASK);
                                        x = P_IN();
                                        P_MODE(last_mask);
                                        P_MODE(last_mask);
                                        if (!(x & P_RDY)) {
                                                x = (*p_signalHandler)(x);
                                                if (x & P_RDY) break;
                                        }
                                        ct = P_CHKINT;
                                }
#endif
                        } while (!((x = P_IN()) & P_RDY));
                }
        }
#ifdef RDY_TOGGLES
        }
#endif
#ifdef DOUBLEI
        p_read_ct++;
        if ((x & P_IN_MASK) != ((y = P_IN()) & P_IN_MASK)) {
                p_read_dble++;
if (last_s) sHct++; else sLct++;
if ((x & P_RDY) != (y & P_RDY)) { if (y & P_RDY) rdyHct++; else rdyLct++; }
if ((x & P_I0) != (y & P_I0)) { if (y & P_I0) i0Hct++; else i0Lct++; }
if ((x & P_I1) != (y & P_I1)) { if (y & P_I1) i1Hct++; else i1Lct++; }
if ((x & P_I2) != (y & P_I2)) { if (y & P_I2) i2Hct++; else i2Lct++; }
if ((x & P_I3) != (y & P_I3)) { if (y & P_I3) i3Hct++; else i3Lct++; }
                x = y;
        }
#endif
```

```
        /* Toggle speaker bit */
        p_toneToggle();

        /* return data bit vector */
        return(x);
}

#ifndef DO_NOT_INLINE
extern inline
#endif
barrier
p_waitvec(register int flag)
{
        /* Do a barrier wait sending flag and return the collected flag
           vector.  This is not super efficient, but is an easier
           interface than using the raw P_OUT() and P_IN() calls
           directly.
        */
        register barrier mask = flag ? OUR_MASK : 0;

        /* If we're not the only PE in our mask, we have work to
           do...  but if we are alone, we are done.
        */
        if (last_mask & ~OUR_MASK) {
                /* Must gather a bit from each PE...  translate
                   {I3,I2,I1,I0} into a standard mask
                */
                mask |= IToMask(_p_waitvec(ALL_MASK ^ mask));
        }

        /* Return constructed bit mask... */
        return(mask);
}


#ifndef DO_NOT_INLINE
extern inline
#endif
int
p_any(register int flag)
{
        /* Do a barrier wait sending flag and return 1 if any flag
           was nonzero, 0 otherwise.
        */
        if (last_mask & ~OUR_MASK) {
                /* A variation on p_waitvec()... */
                return(IToMask(_p_waitvec(flag ? 0xe : 0xf)) || flag);
        }                          /* Note: PAPERS1 Change here ^^^^^^^ */

        /* Only us... */
        return(flag != 0);
}


#ifndef DO_NOT_INLINE
extern inline
#endif
int
p_all(register int flag)
{
        /* Do a barrier wait sending flag and return 1 if all flags
           were nonzero, 0 otherwise.
```

```
	*/
	if (last_mask & ~OUR_MASK) {
		/* A variation on p_waitvec()... */
		return(!IToMask(_p_waitvec(flag ? 0xf : 0xe)) && flag);
	}			/* Note: PAPERS1 Change here  ^^^^^^^ */

	/* Only us... */
	return(flag != 0);
}


#ifndef DO_NOT_INLINE
extern inline
#endif
void
p_check_int()
{
	register barrier oldMask = last_mask;

	P_MODE(OUR_MASK); /* P_MODE(0) would not work */
	P_MODE(OUR_MASK); /* P_MODE(0) would not work */
	_p_wait();
	P_MODE(last_mask = oldMask);
	P_MODE(oldMask);
}


#ifndef DO_NOT_INLINE
extern inline
#endif
portdata
p_putgetNybble(register portdata nout)
{
	register portdata x, out;
#ifdef DOUBLEI
	register portdata y;
#endif

	if (last_mask & ~OUR_MASK) {
		/* Do a sync with the current mask, selecting the
		   appropriate source processor (which isn't really
		   kosher if we are the source, but that's taken care
		   of below), sending the four bits given in nout, and
		   oh yes, also toggle the strobe....
		*/
#ifdef DOUBLEO
		P_OUT((nout & 0xf) | _last_src | last_s);
#endif
		P_OUT(out = ((nout & 0xf) | _last_src | (last_s ^= P_S)));

#ifdef RDY_TOGGLES
		/* The P_RDY signal should match the strobe...  this
		   is coded as two separate loops to cut polling
		   overhead in waiting for a barrier....  Note that
		   interrupts are not detected within this code....
		*/
		if (!last_s) {
			while ((x = P_IN()) & P_RDY); FIX ME
		} else {
#endif
	if (!((x = P_IN()) & P_RDY)) {
		if (!((x = P_IN()) & P_RDY)) {
#ifdef P_CHKINT
```

```
                                register int ct = P_CHKINT;
#endif
                                do {
#ifdef P_CHKINT
                                        if (ct-- == 0) {
                                                P_MODE(OUR_MASK);
                                                P_MODE(OUR_MASK);
                                                x = P_IN();
                                                P_MODE(last_mask);
                                                P_MODE(last_mask);
                                                if (!(x & P_RDY)) {
                                                        x = (*p_signalHandler)(x);
                                                        if (x & P_RDY) break;
                                                }
                                                ct = P_CHKINT;
                                        }
#endif
                                } while (!((x = P_IN()) & P_RDY));
                        }
                }
#ifdef RDY_TOGGLES
                        }
#endif
#ifdef DOUBLEI
                        p_read_ct++;
                        if ((x & P_IN_MASK) != ((y = P_IN()) & P_IN_MASK)) {
                                p_read_dble++;
if (last_s) sHct++; else sLct++;
if ((x & P_RDY) != (y & P_RDY)) { if (y & P_RDY) rdyHct++; else rdyLct++; }
if ((x & P_I0) != (y & P_I0)) { if (y & P_I0) i0Hct++; else i0Lct++; }
if ((x & P_I1) != (y & P_I1)) { if (y & P_I1) i1Hct++; else i1Lct++; }
if ((x & P_I2) != (y & P_I2)) { if (y & P_I2) i2Hct++; else i2Lct++; }
if ((x & P_I3) != (y & P_I3)) { if (y & P_I3) i3Hct++; else i3Lct++; }
                                x = y;
                        }
#endif

                        /* Toggle speaker bit */
                        p_toneToggle();
                } else x = 0;

                /* If we are our source, return nout directly... */
                if (_last_src == ISRC) {
                        return(nout & 0xf);
                } else {
                        /* Given return data nybble */
                        return(IToNybble(x));
                }
}

#ifndef DO_NOT_INLINE
extern inline
#endif
portdata
_p_putgetByte(register portdata bout, register portdata controlVal)
{
                register portdata x, b_in, out, out2;

#ifdef DOUBLEO
                P_OUT((bout & 0xf) | (controlVal ^ P_S));
#endif
                P_OUT(out = ((bout & 0xf) | controlVal));
                out2 = ((bout >> 4) & 0xf) | (controlVal ^ P_S);
```

```
        if (!((x = P_IN()) & P_RDY)) {
                if (!((x = P_IN()) & P_RDY)) {
#ifdef P_CHKINT
                        register int ct = P_CHKINT;
#endif
                        do {
#ifdef P_CHKINT
                                if (ct-- == 0) {
                                        P_MODE(OUR_MASK);
                                        P_MODE(OUR_MASK);
                                        x = P_IN();
                                        P_MODE(last_mask);
                                        P_MODE(last_mask);
                                        if (!(x & P_RDY)) {
                                                x = (*p_signalHandler)(x);
                                                if (x & P_RDY) break;
                                        }
                                        ct = P_CHKINT;
                                }
#endif
                        } while (!((x = P_IN()) & P_RDY));
                }
        }
        p_toneToggle(); /* Toggle speaker bit */
#ifdef DOUBLEO
        P_OUT(out2 ^ P_S);
#endif
        P_OUT(out2);
        b_in = (x >> 4);
        if (!((x = P_IN()) & P_RDY)) {
                if (!((x = P_IN()) & P_RDY)) {
#ifdef P_CHKINT
                        register int ct = P_CHKINT;
#endif
                        do {
#ifdef P_CHKINT
                                if (ct-- == 0) {
                                        P_MODE(OUR_MASK);
                                        P_MODE(OUR_MASK);
                                        x = P_IN();
                                        P_MODE(last_mask);
                                        P_MODE(last_mask);
                                        if (!(x & P_RDY)) {
                                                x = (*p_signalHandler)(x);
                                                if (x & P_RDY) break;
                                        }
                                        ct = P_CHKINT;
                                }
#endif
                        } while (!((x = P_IN()) & P_RDY));
                }
        }
        b_in |= (x & 0xf0);

        p_toneToggle(); /* Toggle speaker bit */
        return(b_in);
}

#ifndef DO_NOT_INLINE
extern inline
#endif
void
_p_construct_mapping(p_mapping *map, register barrier mask)
```

```
{
        register int physical, virtual = 0;
        map->mask = mask;
        for (physical = 0; physical < NPROC; physical++) {
                if (mask & 1) {
                        map->vir2phy[virtual] = physical;
                        map->phy2vir[physical] = virtual++;
                } else {
                        map->phy2vir[physical] = NPROC;
                }
                mask >>= 1;
        }
        map->partition_size = virtual;
        for (; virtual < NPROC; virtual++) {
                map->vir2phy[virtual] = NPROC;
        }
}

#ifndef DO_NOT_INLINE
extern inline
#endif
void
p_construct_mapping(p_mapping *map)
{
        _p_construct_mapping(map, last_mask);
}

#ifndef DO_NOT_INLINE
extern inline
#endif
void
p_update_mapping(p_mapping *map)
{
        if (map->mask != last_mask) _p_construct_mapping(map, last_mask);
}
```

## A.4 Source listing of "intern.h"

```
/*      intern.h

        Internal header file for PAPERS support code.  (PAPERS1 version)
*/

#ifndef _INTERN_H
#define _INTERN_H


#include "papers.h"


#define  P_CHKINT 10000

#define  P_USETONE        /* If defined, generate a tone by syncs */
#undef   P_USETONE


#ifdef   P_USETONE
#define  P_TONEPERM() \
        if (ioperm(((unsigned short) 0x61), 1, 1)) { \
                fprintf(stderr, \
                        "PAPERS PE%d: can't access speaker port\n", \
                        IPROC); \
                exit(1); \
        }
#define P_TONEDIVIDER     100     /* Divider frequency for tone */
extern int p_toneDivider;  /* Freq. divide counter for tone */
#define  p_toneToggle() \
        if (--p_toneDivider <= 0) { \
                outb((inb((unsigned short) 0x61) ^ 2), \
                     ((unsigned short) 0x61)); \
                p_toneDivider = P_TONEDIVIDER; \
        }
#else
#define  P_TONEPERM()             /* Nothing... */
#define  p_toneToggle()           /* Nothing... */
#endif


/*      Port Base address for the PAPERS interface....
 *      Usually the value is 0x378, but IBM ValuePoints are 0x3bc
 * #define       P_PORTBASE 0x378
 * #define       P_PORTBASE 0x3bc
*/
#define  P_PORTBASE 0x3bc


/*      Macro to get permission to access parallel port...
        This can use either:

                ioperm(P_PORTBASE, 3, 1)
                iopl(3)

        The ioperm() call is safer, but yields slower port operations
        due to I/O protection checks and is Linux-specific.
*/
#define  P_PERM() \
        if (iopl(3)) { \
                fprintf(stderr, \
```

```
                        "PAPERS PE%d: can't access parallel port\n", \
                        IPROC); \
                exit(1); \
        }


/*      Stuff concerning the regular output port...
*/
#define  P_OUT(x) \
        outb(((unsigned char)(x)), ((unsigned short) P_PORTBASE))
#define  P_S      0x80          /* Strobe */
/* #define        P_ALT    0x40 */        /* Alternate Input */
#define  P_SRC1   0x20          /* Source Bit 1 */
#define  P_SRC0   0x10          /* Source Bit 0 */
#define  P_D3     0x08          /* Data Bit 3 Value */
#define  P_D2     0x04          /* Data Bit 2 Value */
#define  P_D1     0x02          /* Data Bit 1 Value */
#define  P_D0     0x01          /* Data Bit 0 Value */

/*      Stuff concerning the input port...
        note that P_I3 is inverted in the PAPERS1 Data PAL to compensate
        for the printer port on the PC inverting it....
*/
#define  P_IN() \
        (inb((unsigned short) (P_PORTBASE + 1)))
#define  P_I3     0x80          /* PE3 P_D0 bit if SRC == IPROC */
#define  P_I2     0x40          /* PE2 P_D0 bit if SRC == IPROC */
#define  P_I1     0x20          /* PE1 P_D0 bit if SRC == IPROC */
#define  P_I0     0x10          /* PE0 P_D0 bit if SRC == IPROC */
                                /* if SRC != IPROC, P_Ix == PE(SRC) P_Dx */
#define  P_RDY    0x08          /* Ready */
#define  P_IN_MASK 0xf8         /* Mask for all the input bits */


/*      Stuff concerning the modal output port...
        The PAPERS1 Barrier PAL compensates for the following signals
        being inverted by the printer port on the PC: P_B0 P_B1 P_B3
        Note: P_B2 is not inverted by the port.
*/
#define  P_MODE(x) \
        outb(((unsigned char)((x))), \
             ((unsigned short)(P_PORTBASE + 2)))
#define  P_IE     0x10          /* Interrupt Enable */
#define  P_B3     0x08          /* Barrier contains PE3 */
#define  P_B2     0x04          /* Barrier contains PE2 */
#define  P_B1     0x02          /* Barrier contains PE1 */
#define  P_B0     0x01          /* Barrier contains PE0 */

typedef  unsigned char portdata;        /* Data type for port data */

extern barrier last_mask;       /* Last barrier mask */
extern p_mapping last_map;      /* Last PE mapping */
extern int last_src;                    /* Last source */
extern int _last_src;                   /* Last source (pre-shifted) */
extern portdata last_s;                 /* Last strobe bit value */
extern int isrc;                /* Our PAPERS PE source field value */
#define ISRC      ((const int) isrc)    /* Protected version of isrc */

/*      The following two macros and tables map {I3,I2,I1,I0} into a
        barrier mask and a barrier mask into {I3,I2,I1,I0}.
*/
#define  IToMask(d) ((d) >> 4)
#define  MaskToI(d) ((d) << 4)
```

```
/*      Macro to convert I bits into a data nybble
*/
#define  IToNybble(d) IToMask(d)

#define  ISPE(x) (!((x)&~(NPROC-1)))         /* is x a valid PE number? */
#define  ISNOTPE(x) ((x)&~(NPROC-1))         /* is x NOT a valid PE number? */

#define p_nand(x) (IToMask(_p_waitvec((x) & 0xf)) | (((x) & 0xf) ^ 0xf))
#define p_vote(abstain, vote) \
         (0xf ^ ((abstain) ? p_nand(0xf) : p_nand(0xf ^ (1 << (vote)))))


/* The following are debugging #defines, they will be removed when the software is
stable */

#define  P_SLOWCPU
#undef   P_SLOWCPU

#define  ZOOM
#undef   ZOOM

#define  DOUBLEO
#undef   DOUBLEO
#define  DOUBLEI
#undef   DOUBLEI

#define RDY_TOGGLES
#undef   RDY_TOGGLES

extern int i0Hct, i0Lct, sLct, sHct, rdyLct, rdyHct;
extern int i1Hct, i1Lct;
extern int i2Hct, i2Lct;
extern int i3Hct, i3Lct;
extern int p_read_dble;    /* a count of how many double reads were needed */
extern int p_read_ct;      /* a count of how many total reads were done */

/* End of the debugging #defines, they will be removed when the software is stable */


/*      Ok, now include the inline versions of all the basic support
        functions for PAPERS....  Notice that these routines can be
        somewhat bulky, which is why they are used as inline routines
        only within the internal support library.
*/
#include "inline.h"



extern void p_initenv(void);
extern int p_checkenv(void);
extern portdata p_hw_reset(void);
extern int p_hw_iproc(void);


#endif
```

## A.5 Source listing of "version.h" and "types.h"

```
/*       version.h

         Version specific information for PAPERS....
*/

#ifndef _VERSION_H
#define _VERSION_H


#define  P_PAPERS "PAPERS1"         /* PAPERS (HW) unit title */
#define  P_HW_VERSION 0             /* PAPERS HW version number */
#define  P_HW_REVISION 2            /* PAPERS HW revision number */
#define  P_SW_VERSION 1             /* PAPERS SW version number */
#define  P_SW_REVISION 1            /* PAPERS SW revision number */


#endif

#ifndef _TYPES_H
#define _TYPES_H


/*       PAPERS-specific data types...
*/
typedef  unsigned char              barrier;


/*       PAPERS names for basic data types...
*/
typedef  char                       int8;
typedef  unsigned char              uint8;
typedef  short                      int16;
typedef  unsigned short             uint16;
typedef  int                        int32;
typedef  unsigned int               uint32;
typedef  long long                  int64;
typedef  unsigned long long         uint64;
typedef  float                      f32;
typedef  double                     f64;


/*       Unions for PAPERS to access bits...
*/
#define  typeunion(type)   _##type
typedef  union {  int8   datum; uint8   bits; } _int8;
typedef  union { uint8   datum; uint8   bits; } _uint8;
typedef  union {  int16  datum; uint16  bits; } _int16;
typedef  union { uint16  datum; uint16  bits; } _uint16;
typedef  union {  int32  datum; uint32  bits; } _int32;
typedef  union { uint32  datum; uint32  bits; } _uint32;
typedef  union {  int64  datum; uint64  bits; } _int64;
typedef  union { uint64  datum; uint64  bits; } _uint64;
typedef  union {    f32  datum; uint32  bits; } _f32;
typedef  union {    f64  datum; uint64  bits; } _f64;


#endif
```

## A.6 Source listing of "putget.h"

```
#ifndef _PUTGET_H
#define _PUTGET_H


#include "types.h"


/*       Basic object putget operations... */
#define  XPUTGET(name, type, bitcnt) \
extern type \
name(register type datum, \
register int source);

XPUTGET(p_putget8,    int8,  8)
XPUTGET(p_putget8u,  uint8,  8)
XPUTGET(p_putget16,   int16, 16)
XPUTGET(p_putget16u, uint16, 16)
XPUTGET(p_putget32,   int32, 32)
XPUTGET(p_putget32u, uint32, 32)
XPUTGET(p_putget64,   int64, 64)
XPUTGET(p_putget64u, uint64, 64)
XPUTGET(p_putgetf,      f32, 32)
XPUTGET(p_putgetd,      f64, 64)


/*       Bit sequence putget operations...
*/
#define  XPUTGETBITS(name, type) \
extern type \
name(register type datum, \
register int source, \
register int bits);

XPUTGETBITS(p_putgetBits8u,  uint8)
XPUTGETBITS(p_putgetBits16u, uint16)
XPUTGETBITS(p_putgetBits32u, uint32)
XPUTGETBITS(p_putgetBits64u, uint64)


/* Block transfer putget routines */
extern uint8 p_putgetByte(register uint8 bout);
extern void
p_putgetBlock(register unsigned char *to,
register unsigned char *from,
register int bytes);

#endif
```

## A.7 Source listing of "putget.c"

```
/*       putget.c

         PAPERS putget operations.

         Put a datum into PAPERS and get a datum from whichever PE we
         select.  If the source PE we read from is not a participant in
         the barrier, the result will be a zero value.
*/

#include "putget.h"
#include "intern.h"

/*       Basic object putget operations...
*/
#define  PUTGET(name, type) \
type \
name(register type datum, \
register int source) \
{ \
         register typeunion(type) result; \
         register typeunion(type) d; \
         register portdata controlVal; \
         register int bits = (sizeof(type)) << 3; \
\
         /* If we're the only PE in our mask, we are done...  */ \
         if (last_mask == OUR_MASK) return(datum); \
         result.bits = 0; \
         d.datum = datum; \
\
         /* Set the source... */ \
         p_source(source); \
\
         controlVal = _last_src | (last_s ? 0 : P_S); \
         while ((bits -= 8) > 0) { \
                 result.bits |= _p_putgetByte((d.bits >> bits) & 0xff, controlVal); \
                 result.bits <<= 8; \
         } \
         result.bits |= _p_putgetByte(d.bits & 0xff, controlVal); \
\
         if (source == IPROC) return(datum); \
         else return(result.datum); \
}

PUTGET(p_putget8,     int8)
PUTGET(p_putget8u,   uint8)
PUTGET(p_putget16,   int16)
PUTGET(p_putget16u, uint16)
PUTGET(p_putget32,   int32)
PUTGET(p_putget32u, uint32)
PUTGET(p_putget64,   int64)
PUTGET(p_putget64u, uint64)
PUTGET(p_putgetf,      f32)
PUTGET(p_putgetd,      f64)


/*       Bit sequence putget operations...
*/
#define  PUTGETBITS(name, type) \
type \
name(register type datum, \
```

```
register int source, \
register int bits) \
{ \
        register typeunion(type) result; \
        register typeunion(type) d; \
        register portdata controlVal; \
        register int bit_ceiling = ((bits + 7) >> 3) << 3; \
\
        d.datum = datum; \
        d.bits &= ((1 << bits) - 1); \
\
        /* If we're the only PE in our mask, we are done...  */ \
        if (last_mask == OUR_MASK) return(d.datum); \
        result.bits = 0; \
\
        /* Set the source... */ \
        p_source(source); \
\
        controlVal = _last_src | (last_s ? 0 : P_S); \
        while ((bit_ceiling -= 8) > 0) { \
                result.bits |= _p_putgetByte((d.bits >> bit_ceiling) \
                                    & 0xff, controlVal); \
                result.bits <<= 8; \
        } \
        result.bits |= _p_putgetByte(d.bits & 0xff, controlVal); \
\
        if (source == IPROC) return(d.datum); \
        else return(result.datum); \
}

PUTGETBITS(p_putgetBits8u,  uint8)
PUTGETBITS(p_putgetBits16u, uint16)
PUTGETBITS(p_putgetBits32u, uint32)
PUTGETBITS(p_putgetBits64u, uint64)

uint8
p_putgetByte(register uint8 bout)
{
        register portdata b_in;
        b_in = _p_putgetByte(bout, _last_src | (last_s ^ P_S));
        if (_last_src == ISRC) { return(bout & 0xff); }
        return(b_in);
}

void
p_putgetBlock(register uint8 *to,
register uint8 *from,
register int bytes)
{       /* Transfer a byte at a time... */
        register portdata controlVal = _last_src | (last_s ? 0 : P_S);
        if (_last_src == ISRC) {
                while (--bytes > 0) {
                        _p_putgetByte((*(from++) = *(to++)), controlVal);
                }
                _p_putgetByte((*from = *to), controlVal);
        } else {
                while (--bytes > 0) {
                        *(from++) = _p_putgetByte(*(to++), controlVal);
                }
                *from = _p_putgetByte(*to, controlVal);
        }
}
```

## A.8 Source listing of "gather.h"

```
#ifndef _GATHER_H
#define _GATHER_H


#include "types.h"


/*       Basic object gather operations... */
#define  XGATHER(name, type, bitcnt) \
extern void \
name(register type *dest, \
register type datum);

XGATHER(p_gather8,    int8,  8)
XGATHER(p_gather8u,  uint8,  8)
XGATHER(p_gather16,   int16, 16)
XGATHER(p_gather16u, uint16, 16)
XGATHER(p_gather32,   int32, 32)
XGATHER(p_gather32u, uint32, 32)
XGATHER(p_gather64,   int64, 64)
XGATHER(p_gather64u, uint64, 64)
XGATHER(p_gatherf,      f32, 32)
XGATHER(p_gatherd,      f64, 64)


/*       Bit sequence gather operations...
*/
#define  XGATHERBITS(name, type) \
extern void \
name(register type *dest, \
register type datum, \
register int bits);

XGATHERBITS(p_gatherBits8u,  uint8)
XGATHERBITS(p_gatherBits16u, uint16)
XGATHERBITS(p_gatherBits32u, uint32)
XGATHERBITS(p_gatherBits64u, uint64)


#endif
```

## A.9 Source listing of "gather.c"

```
/*      gather.c

        PAPERS gather operations.  These operations are the core
        mechanism for reductions, etc.

        Gather data from all PEs.  Our contribution is datum.  The
        gathered values from all PEs are placed in the consecutive
        locations starting at dest, i.e., PEk's datum gets placed in
        *(dest + k).  Values are changed only for PEs that participate
        in the barriers (PEs in mask).
*/


#include "putget.h"
#include "gather.h"
#include "intern.h"

/*      Basic object gather operations...
*/
#define GATHER(suffix, type) \
void \
p_gather##suffix(register type *dest, \
register type datum) \
{ \
        register int self, partner, dist, size; \
\
        if (last_mask == OUR_MASK) { \
                dest[IPROC] = datum; \
                return; \
        } \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist++) { \
                if ((partner = last_map.vir2phy[(self+dist)%size]) < NPROC) { \
                        dest[partner] = p_putget##suffix(datum, partner); \
                } else { \
                        p_putget##suffix(datum, IPROC); \
                } \
        } \
}

GATHER(8,    int8)
GATHER(8u,  uint8)
GATHER(16,   int16)
GATHER(16u, uint16)
GATHER(32,   int32)
GATHER(32u, uint32)
GATHER(64,   int64)
GATHER(64u, uint64)
GATHER(f,      f32)
GATHER(d,      f64)


/*      Bit sequence gather operations...
*/
#define GATHERBITS(suffix, type) \
void \
p_gatherBits##suffix(register type *dest, \
register type datum, \
```

```
register int bits) \
{ \
        register int self, partner, dist, size; \
\
        if (last_mask == OUR_MASK) { \
                dest[IPROC] = datum & ((1 << bits) - 1); \
                return; \
        } \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist++) { \
                if ((partner = last_map.vir2phy[(self+dist)%size]) < NPROC) { \
                        dest[partner] = p_putgetBits##suffix(datum, partner, bits);
\
                } else { \
                        p_putgetBits##suffix(datum, IPROC, bits); \
                } \
        } \
}

GATHERBITS(8u,  uint8)
GATHERBITS(16u, uint16)
GATHERBITS(32u, uint32)
GATHERBITS(64u, uint64)
```

## A.10 Source listing of "reduce.h"

```
#ifndef _REDUCE_H
#define _REDUCE_H


#include "types.h"


#define  XREDUCEADD(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XREDUCEADD(p_reduceAdd8,    int8,  8,  8)
XREDUCEADD(p_reduceAdd8u,  uint8,  8,  8u)
XREDUCEADD(p_reduceAdd16,   int16, 16, 16)
XREDUCEADD(p_reduceAdd16u, uint16, 16, 16u)
XREDUCEADD(p_reduceAdd32,   int32, 32, 32)
XREDUCEADD(p_reduceAdd32u, uint32, 32, 32u)
XREDUCEADD(p_reduceAdd64,   int64, 64, 64)
XREDUCEADD(p_reduceAdd64u, uint64, 64, 64u)
XREDUCEADD(p_reduceAddf,      f32, 32, f)
XREDUCEADD(p_reduceAddd,      f64, 64, d)

#define  XREDUCEADDBITS(name, type, suffix) \
extern type \
name(register type datum, \
register int bits);

XREDUCEADDBITS(p_reduceAddBits8u,  uint8,  8u)
XREDUCEADDBITS(p_reduceAddBits16u, uint16, 16u)
XREDUCEADDBITS(p_reduceAddBits32u, uint32, 32u)
XREDUCEADDBITS(p_reduceAddBits64u, uint64, 64u)

#define  XREDUCEMUL(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XREDUCEMUL(p_reduceMul8,    int8,  8,  8)
XREDUCEMUL(p_reduceMul8u,  uint8,  8,  8u)
XREDUCEMUL(p_reduceMul16,   int16, 16, 16)
XREDUCEMUL(p_reduceMul16u, uint16, 16, 16u)
XREDUCEMUL(p_reduceMul32,   int32, 32, 32)
XREDUCEMUL(p_reduceMul32u, uint32, 32, 32u)
XREDUCEMUL(p_reduceMul64,   int64, 64, 64)
XREDUCEMUL(p_reduceMul64u, uint64, 64, 64u)
XREDUCEMUL(p_reduceMulf,      f32, 32, f)
XREDUCEMUL(p_reduceMuld,      f64, 64, d)

#define  XREDUCEMULBITS(name, type, suffix) \
extern type \
name(register type datum, \
register int bits);

XREDUCEMULBITS(p_reduceMulBits8u,  uint8,  8u)
XREDUCEMULBITS(p_reduceMulBits16u, uint16, 16u)
XREDUCEMULBITS(p_reduceMulBits32u, uint32, 32u)
XREDUCEMULBITS(p_reduceMulBits64u, uint64, 64u)


#define  XREDUCEAND(name, type, bitcnt) \
extern type \
```

```
name(register type datum);

XREDUCEAND(p_reduceAnd8,    int8,  8)
XREDUCEAND(p_reduceAnd8u,  uint8,  8)
XREDUCEAND(p_reduceAnd16,   int16, 16)
XREDUCEAND(p_reduceAnd16u, uint16, 16)
XREDUCEAND(p_reduceAnd32,   int32, 32)
XREDUCEAND(p_reduceAnd32u, uint32, 32)
XREDUCEAND(p_reduceAnd64,   int64, 64)
XREDUCEAND(p_reduceAnd64u, uint64, 64)

#define  XREDUCEANDBITS(name, type) \
extern type \
name(register type datum, \
register int bits);

XREDUCEANDBITS(p_reduceAndBits8u,  uint8)
XREDUCEANDBITS(p_reduceAndBits16u, uint16)
XREDUCEANDBITS(p_reduceAndBits32u, uint32)
XREDUCEANDBITS(p_reduceAndBits64u, uint64)

#define  XREDUCEOR(name, type, bitcnt) \
extern type \
name(register type datum);

XREDUCEOR(p_reduceOr8,    int8,  8)
XREDUCEOR(p_reduceOr8u,  uint8,  8)
XREDUCEOR(p_reduceOr16,   int16, 16)
XREDUCEOR(p_reduceOr16u, uint16, 16)
XREDUCEOR(p_reduceOr32,   int32, 32)
XREDUCEOR(p_reduceOr32u, uint32, 32)
XREDUCEOR(p_reduceOr64,   int64, 64)
XREDUCEOR(p_reduceOr64u, uint64, 64)

#define  XREDUCEORBITS(name, type) \
extern type \
name(register type datum, \
register int bits);

XREDUCEORBITS(p_reduceOrBits8u,  uint8)
XREDUCEORBITS(p_reduceOrBits16u, uint16)
XREDUCEORBITS(p_reduceOrBits32u, uint32)
XREDUCEORBITS(p_reduceOrBits64u, uint64)


#define  XREDUCEMIN(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XREDUCEMIN(p_reduceMin8,    int8,  8,  8)
XREDUCEMIN(p_reduceMin8u,  uint8,  8,  8u)
XREDUCEMIN(p_reduceMin16,   int16, 16, 16)
XREDUCEMIN(p_reduceMin16u, uint16, 16, 16u)
XREDUCEMIN(p_reduceMin32,   int32, 32, 32)
XREDUCEMIN(p_reduceMin32u, uint32, 32, 32u)
XREDUCEMIN(p_reduceMin64,   int64, 64, 64)
XREDUCEMIN(p_reduceMin64u, uint64, 64, 64u)
XREDUCEMIN(p_reduceMinf,      f32, 32, f)
XREDUCEMIN(p_reduceMind,      f64, 64, d)

#define  XREDUCEMINBITS(name, type) \
extern type \
name(register type datum, \
```

```
register int bits);

XREDUCEMINBITS(p_reduceMinBits8u,  uint8)
XREDUCEMINBITS(p_reduceMinBits16u, uint16)
XREDUCEMINBITS(p_reduceMinBits32u, uint32)
XREDUCEMINBITS(p_reduceMinBits64u, uint64)

#define  XREDUCEMAX(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XREDUCEMAX(p_reduceMax8,    int8,  8,  8)
XREDUCEMAX(p_reduceMax8u,  uint8,  8,  8u)
XREDUCEMAX(p_reduceMax16,   int16, 16, 16)
XREDUCEMAX(p_reduceMax16u, uint16, 16, 16u)
XREDUCEMAX(p_reduceMax32,   int32, 32, 32)
XREDUCEMAX(p_reduceMax32u, uint32, 32, 32u)
XREDUCEMAX(p_reduceMax64,   int64, 64, 64)
XREDUCEMAX(p_reduceMax64u, uint64, 64, 64u)
XREDUCEMAX(p_reduceMaxf,      f32, 32, f)
XREDUCEMAX(p_reduceMaxd,      f64, 64, d)

#define  XREDUCEMAXBITS(name, type) \
extern type \
name(register type datum, \
register int bits);

XREDUCEMAXBITS(p_reduceMaxBits8u,  uint8)
XREDUCEMAXBITS(p_reduceMaxBits16u, uint16)
XREDUCEMAXBITS(p_reduceMaxBits32u, uint32)
XREDUCEMAXBITS(p_reduceMaxBits64u, uint64)


#endif
```

### A.11 Source listing of "reduce.c"

```
/*       reduce.c

         PAPERS reduce operations.  (Optimized for PAPERS1)

         Reduce data from all enabled PEs.  Our contribution is datum.
*/

#include "putget.h"
#include "reduce.h"
#include "intern.h"

#define  REDUCEADD(name, type, bitcnt, suffix) \
type \
name(register type datum) \
{ \
        register int self, partner, dist, size; \
\
        if (last_mask == OUR_MASK) return(datum); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self ^ dist]) < NPROC) { \
                        datum += p_putget##suffix(datum, partner); \
                } else if ((partner = last_map.vir2phy[(self ^ dist) & ~(dist - 1)]) <
NPROC) { \
                        datum += p_putget##suffix(datum, partner); \
                } else { \
                        p_putget##suffix(datum, IPROC); \
                } \
        } \
         return(datum); \
}

REDUCEADD(p_reduceAdd8,    int8,  8,  8)
REDUCEADD(p_reduceAdd8u,  uint8,  8,  8u)
REDUCEADD(p_reduceAdd16,   int16, 16, 16)
REDUCEADD(p_reduceAdd16u, uint16, 16, 16u)
REDUCEADD(p_reduceAdd32,   int32, 32, 32)
REDUCEADD(p_reduceAdd32u, uint32, 32, 32u)
REDUCEADD(p_reduceAdd64,   int64, 64, 64)
REDUCEADD(p_reduceAdd64u, uint64, 64, 64u)
REDUCEADD(p_reduceAddf,      f32, 32, f)
REDUCEADD(p_reduceAddd,      f64, 64, d)

#define  REDUCEADDBITS(name, type, suffix) \
type \
name(register type datum, \
register int bits) \
{ \
        register int self, partner, dist, size; \
\
        if (last_mask == OUR_MASK) return(datum & ((1 << bits) - 1)); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self ^ dist]) < NPROC) { \
                        datum += p_putgetBits##suffix(datum, partner, bits); \
                } else { \
```

```
                        p_putgetBits##suffix(datum, IPROC, bits); \
                } \
        } \
         return(datum); \
}

REDUCEADDBITS(p_reduceAddBits8u,  uint8,  8u)
REDUCEADDBITS(p_reduceAddBits16u, uint16, 16u)
REDUCEADDBITS(p_reduceAddBits32u, uint32, 32u)
REDUCEADDBITS(p_reduceAddBits64u, uint64, 64u)

#define  REDUCEMUL(name, type, bitcnt, suffix) \
type \
name(register type datum) \
{ \
        register int self, partner, dist, size; \
\
        if (last_mask == OUR_MASK) return(datum); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self^dist]) < NPROC) { \
                        datum *= p_putget##suffix(datum, partner); \
                } else { \
                        p_putget##suffix(datum, IPROC); \
                } \
        } \
         return(datum); \
}

REDUCEMUL(p_reduceMul8,    int8,  8,  8)
REDUCEMUL(p_reduceMul8u,  uint8,  8,  8u)
REDUCEMUL(p_reduceMul16,   int16, 16, 16)
REDUCEMUL(p_reduceMul16u, uint16, 16, 16u)
REDUCEMUL(p_reduceMul32,   int32, 32, 32)
REDUCEMUL(p_reduceMul32u, uint32, 32, 32u)
REDUCEMUL(p_reduceMul64,   int64, 64, 64)
REDUCEMUL(p_reduceMul64u, uint64, 64, 64u)
REDUCEMUL(p_reduceMulf,      f32, 32, f)
REDUCEMUL(p_reduceMuld,      f64, 64, d)

#define  REDUCEMULBITS(name, type, suffix) \
type \
name(register type datum, \
register int bits) \
{ \
        register int self, partner, dist, size; \
\
        if (last_mask == OUR_MASK) return(datum & ((1 << bits) - 1)); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self^dist]) < NPROC) { \
                        datum *= p_putgetBits##suffix(datum, partner, bits); \
                } else { \
                        p_putgetBits##suffix(datum, IPROC, bits); \
                } \
        } \
         return(datum); \
}
```

```
REDUCEMULBITS(p_reduceMulBits8u,  uint8,  8u)
REDUCEMULBITS(p_reduceMulBits16u, uint16, 16u)
REDUCEMULBITS(p_reduceMulBits32u, uint32, 32u)
REDUCEMULBITS(p_reduceMulBits64u, uint64, 64u)

#define  REDUCEAND(name, type, bitcnt) \
type \
name(register type datum) \
{ \
         if (last_mask & ~OUR_MASK) { \
                  register int bits = (bitcnt - 4); \
 \
                  /* Four bits at a time, and = not nand */ \
                  do { \
                           datum &= ~((IToMask(_p_waitvec((datum >> bits) & 0xf))) <<
bits); \
                  } while ((bits -= 4) >= 0); \
         } \
 \
         return(datum); \
}

REDUCEAND(p_reduceAnd8,    int8,  8)
REDUCEAND(p_reduceAnd8u,  uint8,  8)
REDUCEAND(p_reduceAnd16,   int16, 16)
REDUCEAND(p_reduceAnd16u, uint16, 16)
REDUCEAND(p_reduceAnd32,   int32, 32)
REDUCEAND(p_reduceAnd32u, uint32, 32)
REDUCEAND(p_reduceAnd64,   int64, 64)
REDUCEAND(p_reduceAnd64u, uint64, 64)

/* MORE HERE, p_reduceAndBits, p_reduceOR, and p_reduceOrBits
 * haven't been optimized for PAPERS1
 */

#define  REDUCEANDBITS(name, type) \
type \
name(register type datum, \
register int bits) \
{ \
         if (last_mask & ~OUR_MASK) { \
                  register barrier mask = last_mask; \
                  register type bit = 1; \
 \
                  /* One bit at a time from all PEs... */ \
                  do { \
                           if (datum & bit) { \
                                    /* Not a 1 if somebody else is 0... */ \
                                    if (p_waitvec(0) & mask) { \
                                             /* Flip the bit... */ \
                                             datum ^= bit; \
                                    } \
                           } else { \
                                    /* We've got a 0; tell others... */ \
                                    p_waitvec(1); \
                           } \
 \
                           /* Go to the next bit position... */ \
                           bit += bit; \
                  } while (--bits); \
         } \
 \
         return(datum); \
```

```
}

REDUCEANDBITS(p_reduceAndBits8u,  uint8)
REDUCEANDBITS(p_reduceAndBits16u, uint16)
REDUCEANDBITS(p_reduceAndBits32u, uint32)
REDUCEANDBITS(p_reduceAndBits64u, uint64)

#define  REDUCEOR(name, type, bitcnt) \
type \
name(register type datum) \
{ \
         if (last_mask & ~OUR_MASK) { \
                  register int bits = (bitcnt - 4); \
 \
                  /* Four bits at a time, or = nand not */ \
                  do { \
                           datum |= (IToMask(_p_waitvec((~(datum >> bits)) & 0xf)) \
                                      << bits); \
                  } while ((bits -= 4) >= 0); \
         } \
 \
         return(datum); \
}

REDUCEOR(p_reduceOr8,    int8,  8)
REDUCEOR(p_reduceOr8u,  uint8,  8)
REDUCEOR(p_reduceOr16,   int16, 16)
REDUCEOR(p_reduceOr16u, uint16, 16)
REDUCEOR(p_reduceOr32,   int32, 32)
REDUCEOR(p_reduceOr32u, uint32, 32)
REDUCEOR(p_reduceOr64,   int64, 64)
REDUCEOR(p_reduceOr64u, uint64, 64)

#define  REDUCEORBITS(name, type) \
type \
name(register type datum, \
register int bits) \
{ \
         if (last_mask & ~OUR_MASK) { \
                  register barrier mask = last_mask; \
                  register type bit = 1; \
 \
                  /* One bit at a time from all PEs... */ \
                  do { \
                           if (datum & bit) { \
                                    /* We've got a 1; do dummy barrier... */ \
                                    p_waitvec(1); \
                           } else { \
                                    /* Only a 1 if somebody else is... */ \
                                    if (p_waitvec(0) & mask) { \
                                             datum |= bit; \
                                    } \
                           } \
 \
                           /* Go to the next bit position... */ \
                           bit += bit; \
                  } while (--bits); \
         } \
 \
         return(datum); \
}

REDUCEORBITS(p_reduceOrBits8u,  uint8)
```

```
REDUCEORBITS(p_reduceOrBits16u, uint16)
REDUCEORBITS(p_reduceOrBits32u, uint32)
REDUCEORBITS(p_reduceOrBits64u, uint64)

#define  REDUCEMIN(name, type, bitcnt, suffix) \
type \
name(register type min) \
{ \
        register int self, partner, dist, size; \
         register type temp; \
\
        if (last_mask == OUR_MASK) return(min); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self^dist]) < NPROC) { \
                        temp = p_putget##suffix(min, partner); \
                           if (temp < min) min = temp; \
                } else { \
                        p_putget##suffix(min, IPROC); \
                } \
        } \
         return(min); \
}

REDUCEMIN(p_reduceMin8,    int8,  8,  8)
REDUCEMIN(p_reduceMin8u,  uint8,  8,  8u)
REDUCEMIN(p_reduceMin16,   int16, 16, 16)
REDUCEMIN(p_reduceMin16u, uint16, 16, 16u)
REDUCEMIN(p_reduceMin32,   int32, 32, 32)
REDUCEMIN(p_reduceMin32u, uint32, 32, 32u)
REDUCEMIN(p_reduceMin64,   int64, 64, 64)
REDUCEMIN(p_reduceMin64u, uint64, 64, 64u)
REDUCEMIN(p_reduceMinf,      f32, 32, f)
REDUCEMIN(p_reduceMind,      f64, 64, d)

#define  REDUCEMINBITS(name, type, suffix) \
type \
name(register type min, \
register int bits) \
{ \
        register int self, partner, dist, size; \
         register type temp; \
\
        if (last_mask == OUR_MASK) return(min & ((1 << bits) - 1)); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self^dist]) < NPROC) { \
                        temp = p_putgetBits##suffix(min, partner, bits); \
                           if (temp < min) min = temp; \
                } else { \
                        p_putgetBits##suffix(min, IPROC, bits); \
                } \
        } \
         return(min); \
}

REDUCEMINBITS(p_reduceMinBits8u,  uint8,  8u)
REDUCEMINBITS(p_reduceMinBits16u, uint16, 16u)
REDUCEMINBITS(p_reduceMinBits32u, uint32, 32u)
```

```
REDUCEMINBITS(p_reduceMinBits64u, uint64, 64u)

#define  REDUCEMAX(name, type, bitcnt, suffix) \
type \
name(register type max) \
{ \
        register int self, partner, dist, size; \
         register type temp; \
\
        if (last_mask == OUR_MASK) return(max); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self^dist]) < NPROC) { \
                        temp = p_putget##suffix(max, partner); \
                           if (temp > max) max = temp; \
                } else { \
                        p_putget##suffix(max, IPROC); \
                } \
        } \
         return(max); \
}

REDUCEMAX(p_reduceMax8,    int8,  8,  8)
REDUCEMAX(p_reduceMax8u,  uint8,  8,  8u)
REDUCEMAX(p_reduceMax16,   int16, 16, 16)
REDUCEMAX(p_reduceMax16u, uint16, 16, 16u)
REDUCEMAX(p_reduceMax32,   int32, 32, 32)
REDUCEMAX(p_reduceMax32u, uint32, 32, 32u)
REDUCEMAX(p_reduceMax64,   int64, 64, 64)
REDUCEMAX(p_reduceMax64u, uint64, 64, 64u)
REDUCEMAX(p_reduceMaxf,      f32, 32, f)
REDUCEMAX(p_reduceMaxd,      f64, 64, d)

#define  REDUCEMAXBITS(name, type, suffix) \
type \
name(register type max, \
register int bits) \
{ \
        register int self, partner, dist, size; \
         register type temp; \
\
        if (last_mask == OUR_MASK) return(max & ((1 << bits) - 1)); \
        p_update_mapping(&last_map); \
        size = last_map.partition_size; \
        self = last_map.phy2vir[IPROC]; \
        for (dist = 1; dist < size; dist <<= 1) { \
                if ((partner = last_map.vir2phy[self^dist]) < NPROC) { \
                        temp = p_putgetBits##suffix(max, partner, bits); \
                           if (temp > max) max = temp; \
                } else { \
                        p_putgetBits##suffix(max, IPROC, bits); \
                } \
        } \
         return(max); \
}

REDUCEMAXBITS(p_reduceMaxBits8u,  uint8,  8u)
REDUCEMAXBITS(p_reduceMaxBits16u, uint16, 16u)
REDUCEMAXBITS(p_reduceMaxBits32u, uint32, 32u)
REDUCEMAXBITS(p_reduceMaxBits64u, uint64, 64u)
```

### A.12 Source listing of "scan.h"

```
#ifndef _SCAN_H
#define _SCAN_H


#include "types.h"


#define  XSCANADD(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XSCANADD(p_scanAdd8,    int8,  8,  8)
XSCANADD(p_scanAdd8u,  uint8,  8,  8u)
XSCANADD(p_scanAdd16,   int16, 16, 16)
XSCANADD(p_scanAdd16u, uint16, 16, 16u)
XSCANADD(p_scanAdd32,   int32, 32, 32)
XSCANADD(p_scanAdd32u, uint32, 32, 32u)
XSCANADD(p_scanAdd64,   int64, 64, 64)
XSCANADD(p_scanAdd64u, uint64, 64, 64u)
XSCANADD(p_scanAddf,      f32, 32, f)
XSCANADD(p_scanAddd,      f64, 64, d)

#define  XSCANMUL(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XSCANMUL(p_scanMul8,    int8,  8,  8)
XSCANMUL(p_scanMul8u,  uint8,  8,  8u)
XSCANMUL(p_scanMul16,   int16, 16, 16)
XSCANMUL(p_scanMul16u, uint16, 16, 16u)
XSCANMUL(p_scanMul32,   int32, 32, 32)
XSCANMUL(p_scanMul32u, uint32, 32, 32u)
XSCANMUL(p_scanMul64,   int64, 64, 64)
XSCANMUL(p_scanMul64u, uint64, 64, 64u)
XSCANMUL(p_scanMulf,      f32, 32, f)
XSCANMUL(p_scanMuld,      f64, 64, d)


#define  XSCANAND(name, type, bitcnt) \
extern type \
name(register type datum);

XSCANAND(p_scanAnd8,    int8,  8)
XSCANAND(p_scanAnd8u,  uint8,  8)
XSCANAND(p_scanAnd16,   int16, 16)
XSCANAND(p_scanAnd16u, uint16, 16)
XSCANAND(p_scanAnd32,   int32, 32)
XSCANAND(p_scanAnd32u, uint32, 32)
XSCANAND(p_scanAnd64,   int64, 64)
XSCANAND(p_scanAnd64u, uint64, 64)

#define  XSCANANDBITS(name, type) \
extern type \
name(register type datum, \
register int bits);

XSCANANDBITS(p_scanAndBits8u,  uint8)
XSCANANDBITS(p_scanAndBits16u, uint16)
XSCANANDBITS(p_scanAndBits32u, uint32)
XSCANANDBITS(p_scanAndBits64u, uint64)
```

```
#define  XSCANOR(name, type, bitcnt) \
extern type \
name(register type datum);

XSCANOR(p_scanOr8,    int8,  8)
XSCANOR(p_scanOr8u,  uint8,  8)
XSCANOR(p_scanOr16,   int16, 16)
XSCANOR(p_scanOr16u, uint16, 16)
XSCANOR(p_scanOr32,   int32, 32)
XSCANOR(p_scanOr32u, uint32, 32)
XSCANOR(p_scanOr64,   int64, 64)
XSCANOR(p_scanOr64u, uint64, 64)

#define  XSCANORBITS(name, type) \
extern type \
name(register type datum, \
register int bits);

XSCANORBITS(p_scanOrBits8u,  uint8)
XSCANORBITS(p_scanOrBits16u, uint16)
XSCANORBITS(p_scanOrBits32u, uint32)
XSCANORBITS(p_scanOrBits64u, uint64)


#define  XSCANMIN(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XSCANMIN(p_scanMin8,    int8,  8,  8)
XSCANMIN(p_scanMin8u,  uint8,  8,  8u)
XSCANMIN(p_scanMin16,   int16, 16, 16)
XSCANMIN(p_scanMin16u, uint16, 16, 16u)
XSCANMIN(p_scanMin32,   int32, 32, 32)
XSCANMIN(p_scanMin32u, uint32, 32, 32u)
XSCANMIN(p_scanMin64,   int64, 64, 64)
XSCANMIN(p_scanMin64u, uint64, 64, 64u)
XSCANMIN(p_scanMinf,      f32, 32, f)
XSCANMIN(p_scanMind,      f64, 64, d)

#define  XSCANMAX(name, type, bitcnt, suffix) \
extern type \
name(register type datum);

XSCANMAX(p_scanMax8,    int8,  8,  8)
XSCANMAX(p_scanMax8u,  uint8,  8,  8u)
XSCANMAX(p_scanMax16,   int16, 16, 16)
XSCANMAX(p_scanMax16u, uint16, 16, 16u)
XSCANMAX(p_scanMax32,   int32, 32, 32)
XSCANMAX(p_scanMax32u, uint32, 32, 32u)
XSCANMAX(p_scanMax64,   int64, 64, 64)
XSCANMAX(p_scanMax64u, uint64, 64, 64u)
XSCANMAX(p_scanMaxf,      f32, 32, f)
XSCANMAX(p_scanMaxd,      f64, 64, d)


#define  XRANK(name, type, suffix) \
extern int \
name(register type datum);

XRANK(p_rank8,    int8,  8)
XRANK(p_rank8u,  uint8,  8u)
XRANK(p_rank16,   int16, 16)
```

```
XRANK(p_rank16u, uint16, 16u)
XRANK(p_rank32,   int32, 32)
XRANK(p_rank32u, uint32, 32u)
XRANK(p_rank64,   int64, 64)
XRANK(p_rank64u, uint64, 64u)
XRANK(p_rankf,      f32, f)
XRANK(p_rankd,      f64, d)

#define  XRANKBITS(name, type, suffix) \
extern int \
name(register type datum, \
register int bits);

XRANKBITS(p_rankBits8u,  uint8,  8u)
XRANKBITS(p_rankBits16u, uint16, 16u)
XRANKBITS(p_rankBits32u, uint32, 32u)
XRANKBITS(p_rankBits64u, uint64, 64u)



extern int
p_enumerate();


extern int
p_selectFirst();

extern int
p_selectOne();


#endif
```

## A.13 Source listing of "scan.c"

```
/*      scan.c

        PAPERS scan operations.
        Reduce data from all enabled PEs.  Our contribution is datum.
*/

#include "gather.h"
#include "scan.h"
#include "intern.h"

#define  SCANADD(name, type, bitcnt, suffix) \
type \
name(register type datum) \
{ \
        type buf[NPROC]; \
        register type t; \
        buf[0] = (buf[1] = (buf[2] = (buf[3] = 0))); \
        p_gather##suffix(&(buf[0]), datum); \
        t = buf[0]; \
        if (IPROC > 0) { \
                t += buf[1]; \
                if (IPROC > 1) { \
                        t += buf[2]; \
                        if (IPROC > 2) { \
                                t += buf[3]; \
                        } \
                } \
        } \
        return(t); \
}

SCANADD(p_scanAdd8,    int8,  8,  8)
SCANADD(p_scanAdd8u,  uint8,  8,  8u)
SCANADD(p_scanAdd16,   int16, 16, 16)
SCANADD(p_scanAdd16u, uint16, 16, 16u)
SCANADD(p_scanAdd32,   int32, 32, 32)
SCANADD(p_scanAdd32u, uint32, 32, 32u)
SCANADD(p_scanAdd64,   int64, 64, 64)
SCANADD(p_scanAdd64u, uint64, 64, 64u)
SCANADD(p_scanAddf,      f32, 32, f)
SCANADD(p_scanAddd,      f64, 64, d)

#define  SCANMUL(name, type, bitcnt, suffix) \
type \

name(register type datum) \
{ \
        type buf[NPROC]; \
        register type t; \
        buf[0] = (buf[1] = (buf[2] = (buf[3] = 1))); \
        p_gather##suffix(&(buf[0]), datum); \
        t = buf[0]; \
        if (IPROC > 0) { \
                t *= buf[1]; \
                if (IPROC > 1) { \
                        t *= buf[2]; \
                        if (IPROC > 2) { \
                                t *= buf[3]; \
                        } \
                } \
```

```
        } \
        return(t); \
}

SCANMUL(p_scanMul8,    int8,  8,  8)
SCANMUL(p_scanMul8u,  uint8,  8,  8u)
SCANMUL(p_scanMul16,   int16, 16, 16)
SCANMUL(p_scanMul16u, uint16, 16, 16u)
SCANMUL(p_scanMul32,   int32, 32, 32)
SCANMUL(p_scanMul32u, uint32, 32, 32u)
SCANMUL(p_scanMul64,   int64, 64, 64)
SCANMUL(p_scanMul64u, uint64, 64, 64u)
SCANMUL(p_scanMulf,      f32, 32, f)
SCANMUL(p_scanMuld,      f64, 64, d)

#define  SCANAND(name, type, bitcnt) \
type \
name(register type datum) \
{ \
        register int bits = bitcnt; \
        if (last_mask & ~OUR_MASK) { \
                register barrier mask = \
                        (last_mask & (OUR_MASK - 1)); \
                register type bit = 1; \
                /* One bit at a time from all PEs... */ \
                do { \
                        if (datum & bit) { \
                                /* Not a 1 if somebody else is 0... */ \
                                if (p_waitvec(0) & mask) { \
                                        /* Flip the bit... */ \
                                        datum ^= bit; \
                                } \
                        } else { \
                                /* We've got a 0; tell others... */ \
                                p_waitvec(1); \
                        } \
                        /* Go to the next bit position... */ \
                        bit += bit; \
                } while (--bits); \
        } \
        return(datum); \
}

SCANAND(p_scanAnd8,    int8,  8)
SCANAND(p_scanAnd8u,  uint8,  8)
SCANAND(p_scanAnd16,   int16, 16)
SCANAND(p_scanAnd16u, uint16, 16)
SCANAND(p_scanAnd32,   int32, 32)
SCANAND(p_scanAnd32u, uint32, 32)
SCANAND(p_scanAnd64,   int64, 64)
SCANAND(p_scanAnd64u, uint64, 64)

#define  SCANANDBITS(name, type) \
type \
name(register type datum, \
register int bits) \
{ \
        if (last_mask & ~OUR_MASK) { \
                register barrier mask = \
                        (last_mask & (OUR_MASK - 1)); \
                register type bit = 1; \
                /* One bit at a time from all PEs... */ \
                do { \
```

```
                        if (datum & bit) { \
                                /* Not a 1 if somebody else is 0... */ \
                                if (p_waitvec(0) & mask) { \
                                        /* Flip the bit... */ \
                                        datum ^= bit; \
                                } \
                        } else { \
                                /* We've got a 0; tell others... */ \
                                p_waitvec(1); \
                        } \
                        /* Go to the next bit position... */ \
                        bit += bit; \
                } while (--bits); \
        } \
        return(datum); \
}

SCANANDBITS(p_scanAndBits8u,  uint8)
SCANANDBITS(p_scanAndBits16u, uint16)
SCANANDBITS(p_scanAndBits32u, uint32)
SCANANDBITS(p_scanAndBits64u, uint64)

#define  SCANOR(name, type, bitcnt) \
type \
name(register type datum) \
{ \
        register int bits = bitcnt; \
        if (last_mask & ~OUR_MASK) { \
                register barrier mask = \
                        (last_mask & (OUR_MASK - 1)); \
                register type bit = 1; \
                /* One bit at a time from all PEs... */ \
                do { \
                        if (datum & bit) { \
                                /* We've got a 1; do dummy barrier... */ \
                                p_waitvec(1); \
                        } else { \
                                /* Only a 1 if somebody else is... */ \
                                if (p_waitvec(0) & mask) { \
                                        datum |= bit; \
                                } \
                        } \
                        /* Go to the next bit position... */ \
                        bit += bit; \
                } while (--bits); \
        } \
        return(datum); \
}

SCANOR(p_scanOr8,    int8,  8)
SCANOR(p_scanOr8u,  uint8,  8)
SCANOR(p_scanOr16,   int16, 16)
SCANOR(p_scanOr16u, uint16, 16)
SCANOR(p_scanOr32,   int32, 32)
SCANOR(p_scanOr32u, uint32, 32)
SCANOR(p_scanOr64,   int64, 64)
SCANOR(p_scanOr64u, uint64, 64)

#define  SCANORBITS(name, type) \
type \
name(register type datum, \
register int bits) \
{ \
```

```
	if (last_mask & ~OUR_MASK) { \
		register barrier mask = \
			(last_mask & (OUR_MASK - 1)); \
		register type bit = 1; \
		/* One bit at a time from all PEs... */ \
		do { \
			if (datum & bit) { \
				/* We've got a 1; do dummy barrier... */ \
				p_waitvec(1); \
			} else { \
				/* Only a 1 if somebody else is... */ \
				if (p_waitvec(0) & mask) { \
					datum |= bit; \
				} \
			} \
			/* Go to the next bit position... */ \
			bit += bit; \
		} while (--bits); \
	} \
	return(datum); \
}

SCANORBITS(p_scanOrBits8u,  uint8)
SCANORBITS(p_scanOrBits16u, uint16)
SCANORBITS(p_scanOrBits32u, uint32)
SCANORBITS(p_scanOrBits64u, uint64)


#define  SCANMIN(name, type, bitcnt, suffix) \
type \
name(register type min) \
{ \
	type buf[NPROC]; \
	buf[0] = (buf[1] = (buf[2] = (buf[3] = min))); \
	p_gather##suffix(&(buf[0]), min); \
	if (IPROC > 0) { \
		if (buf[0] < min) min = buf[0]; \
		if (IPROC > 1) { \
			if (buf[1] < min) min = buf[1]; \
			if (IPROC > 2) { \
				if (buf[2] < min) min = buf[2]; \
			} \
		} \
	} \
	return(min); \
}

SCANMIN(p_scanMin8,    int8,  8,  8)
SCANMIN(p_scanMin8u,  uint8,  8,  8u)
SCANMIN(p_scanMin16,   int16, 16, 16)
SCANMIN(p_scanMin16u, uint16, 16, 16u)
SCANMIN(p_scanMin32,   int32, 32, 32)
SCANMIN(p_scanMin32u, uint32, 32, 32u)
SCANMIN(p_scanMin64,   int64, 64, 64)
SCANMIN(p_scanMin64u, uint64, 64, 64u)
SCANMIN(p_scanMinf,      f32, 32, f)
SCANMIN(p_scanMind,      f64, 64, d)

#define  SCANMAX(name, type, bitcnt, suffix) \
type \
name(register type max) \
{ \
	type buf[NPROC]; \
```

```
        buf[0] = (buf[1] = (buf[2] = (buf[3] = max))); \
        p_gather##suffix(&(buf[0]), max); \
        if (IPROC > 0) { \
                if (buf[0] > max) max = buf[0]; \
                if (IPROC > 1) { \
                        if (buf[1] > max) max = buf[1]; \
                        if (IPROC > 2) { \
                                if (buf[2] > max) max = buf[2]; \
                        } \
                } \
        } \
        return(max); \
}

SCANMAX(p_scanMax8,    int8,  8,  8)
SCANMAX(p_scanMax8u,  uint8,  8,  8u)
SCANMAX(p_scanMax16,   int16, 16, 16)
SCANMAX(p_scanMax16u, uint16, 16, 16u)
SCANMAX(p_scanMax32,   int32, 32, 32)
SCANMAX(p_scanMax32u, uint32, 32, 32u)
SCANMAX(p_scanMax64,   int64, 64, 64)
SCANMAX(p_scanMax64u, uint64, 64, 64u)
SCANMAX(p_scanMaxf,      f32, 32, f)
SCANMAX(p_scanMaxd,      f64, 64, d)

#define  RANK(name, type, suffix) \
int \
name(register type datum) \
{ \
        type buf[NPROC]; \
        register int rank = 0; \
        register int i = 0; \
        p_gather##suffix(&(buf[0]), datum); \
        do { \
                if (last_mask & (1 << i)) { \
                        if ((buf[i] > datum) || \
                            ((i < IPROC) && (buf[i] == datum))) { \
                                ++rank; \
                        } \
                } \
        } while (++i < NPROC); \
        return(rank); \
}

RANK(p_rank8,    int8,  8)
RANK(p_rank8u,  uint8,  8u)
RANK(p_rank16,   int16, 16)
RANK(p_rank16u, uint16, 16u)
RANK(p_rank32,   int32, 32)
RANK(p_rank32u, uint32, 32u)
RANK(p_rank64,   int64, 64)
RANK(p_rank64u, uint64, 64u)
RANK(p_rankf,      f32, f)
RANK(p_rankd,      f64, d)

#define  RANKBITS(name, type, suffix) \
int \
name(register type datum, \
register int bits) \
{ \
        type buf[NPROC]; \
        register int rank = 0; \
        register int i = 0; \
```

```
        p_gatherBits##suffix(&(buf[0]), datum, bits); \
        do { \
                if (last_mask & (1 << i)) { \
                        if ((buf[i] > datum) || \
                            ((i < IPROC) && (buf[i] == datum))) { \
                                ++rank; \
                        } \
                } \
        } while (++i < NPROC); \
        return(rank); \
}

RANKBITS(p_rankBits8u,  uint8,  8u)
RANKBITS(p_rankBits16u, uint16, 16u)
RANKBITS(p_rankBits32u, uint32, 32u)
RANKBITS(p_rankBits64u, uint64, 64u)

int
p_enumerate()
{
        /* Number the PEs that are currently "active"...
           notice that this doesn't even require a sync.
        */
        register barrier maskbit = OUR_MASK;
        register int mynum = 0;
        do {
                if ((maskbit >>= 1) & last_mask) ++mynum;
        } while (maskbit);
        return(mynum);
}

int
p_selectFirst()
{
        /* Select any PE that is currently "active"
           notice that this doesn't even require a sync.
        */
        register barrier maskbit = 1;
        register int mynum = 0;
        while (!(maskbit & last_mask)) {
                maskbit += maskbit;
                ++mynum;
        }
        return(mynum);
}

int
p_selectOne()
{
        /* Select any PE that is currently "active"
           notice that this doesn't even require a sync.;
           p_selectOne() is an exact copy of p_selectFirst()
        */
        register barrier maskbit = 1;
        register int mynum = 0;
        while (!(maskbit & last_mask)) {
                maskbit += maskbit;
                ++mynum;
        }
        return(mynum);
}
```

# Appendix B: PAL Logic Equations for PAPERS1

## B.1 Control/Barrier Chip Equations for PE0

```
;PALASM Design Description
;---------------------------------- Declaration Segment ------------
TITLE    Papers1 PE0 Barrier Chip
PATTERN
REVISION
AUTHOR
COMPANY
DATE     8/1/94

CHIP  _papers1  PALCE22V10
;---------------------------------- PIN Declarations ---------------
PIN  1          CLK
PIN  2          PE0IRQ      COMB
PIN  3          PE1IRQ      COMB
PIN  4         /PE2IRQ      COMB        ;NOTE: PE2IRQ is active LOW
PIN  5          PE3IRQ      COMB
PIN  6         /BMASK1      COMB
PIN  7          BMASK2      COMB        ;NOTE: BMASK2 is active HIGH
PIN  8         /BMASK3      COMB
PIN  9          PE1B0       COMB
PIN  10         PE2B0       COMB
PIN  11         PE3B0       COMB
PIN  12         GND

PIN  24         VCC
PIN  23         PRVSTRB     REGISTERED  ;NC
PIN  22         X           REGISTERED  ;NC
PIN  21         Y           REGISTERED  ;NC
PIN  20         RDY         REGISTERED
PIN  19         Z           REGISTERED  ;NC
PIN  18         DATACLK     REGISTERED  ;Used to clock the Data PAL
PIN  17        /WAITLED     REGISTERED  ;Used to drive the wait LED
PIN  16         PE0B1       REGISTERED
PIN  15         PE0B2       REGISTERED
PIN  14         PE0B3       REGISTERED
PIN  13         STROBE      COMB

STRING CHANGED '(PRVSTRB :+: STROBE)'       ;NOTE: Operator :+: is XOR.
STRING ASYNCRST '(PE0IRQ*/STROBE*/BMASK1*/BMASK2*/BMASK3)'
STRING ASYNCINT '((PE1IRQ*PE1B0 + PE2IRQ*PE2B0 + PE3IRQ*PE3B0)*/PE0IRQ)'
STRING ASYNCALT '(/BMASK1*/BMASK2*/BMASK3)'

;***** The State Machine *****
STRING sA '(/DATACLK*/X* Y* Z)';0011 A: idle, RDY=1
;       if CHANGED==1, go to State B, else stay in State A
STRING sB '(/DATACLK*/X* Y*/Z)';0010 B: debounce
;       if CHANGED==0, go to State A, else latch b-mask, RDY=0, go to State C
STRING sR '(/DATACLK*/X*/Y*/Z)';0000 R: reset, RDY=0, PRVSTRB=1, clear b-mask
;       if ASYNCRST==0, go to State D
STRING sC '(/DATACLK*/X*/Y* Z)';0001 C: wait, maintain b-mask
;       if GO==1, go to State D and latch the data, else stay in State C
STRING sD '( DATACLK*/X*/Y* Z)';1001 D: clear b-mask, toggle PRVSTRB, go to E
STRING sE '( DATACLK*/X*/Y*/Z)';1000 E: delay 2, go to F
STRING sF '( DATACLK*/X* Y*/Z)';1010 F: delay 3, go to G
STRING sG '( DATACLK*/X* Y* Z)';1011 G: delay 4, go to H
STRING sH '( DATACLK* X* Y* Z)';1111 H: delay 5, skip I,J,K,L,M,N, go to O
```

```
STRING sI '( DATACLK* X* Y*/Z)';1110 I: delay 6, go to J
STRING sJ '( DATACLK* X*/Y*/Z)';1100 J: delay 7, go to K
STRING sK '( DATACLK* X*/Y* Z)';1101 K: delay 8, go to L
STRING sL '(/DATACLK* X*/Y* Z)';0101 L: delay 9, go to M
STRING sM '(/DATACLK* X*/Y*/Z)';0100 M: delay10, go to N
STRING sN '(/DATACLK* X* Y*/Z)';0110 N: delay11, go to O
STRING sO '(/DATACLK* X* Y* Z)';0111 O: delay 6, go to A
;---------------------------------- Boolean Equation Segment ------
EQUATIONS
WAITLED=sR + sC
RDY=((sA)*/ASYNCALT + (/ASYNCINT*ASYNCALT))
;RDY=((sA + sO + (sB*/CHANGED))*/ASYNCALT + (ASYNCINT*ASYNCALT))
DATACLK=(/ASYNCRST)*(sD + sE + sF + sG + sI + sJ + sR + /ASYNCINT*sC*(
  (/PE0IRQ*(/PE0B1+PE1B0       )*(/PE0B2+PE2B0       )*(/PE0B3+PE3B0       ))
  +(PE0IRQ*(/PE0B1+PE1B0*PE1IRQ)*(/PE0B2+PE2B0*PE2IRQ)*(/PE0B3+PE3B0*PE3IRQ))
 ))
X=(/ASYNCRST)*(sG + sH + sI + sJ + sK + sL + sM + sN)
Y=(/ASYNCRST)*(sE + sF + sG + sH + sM + sN + sO + sA + sB*/CHANGED)
Z=(/ASYNCRST)*(sR + sC + sF + sG + sJ + sK + sN + sO + sB + sA*/CHANGED)
PE0B1=sB*CHANGED*BMASK1 + sC*PE0B1
PE0B2=sB*CHANGED*BMASK2 + sC*PE0B2
PE0B3=sB*CHANGED*BMASK3 + sC*PE0B3
PRVSTRB=(sD :+: PRVSTRB) + sR
```

## B.2 Control/Barrier Chip Equations for PE1

```
;PALASM Design Description
;--------------------------------- Declaration Segment ------------
TITLE    Papers1 PE1 Barrier Chip
PATTERN
REVISION
AUTHOR
COMPANY
DATE     8/1/94
CHIP  _papers1  PALCE22V10
;--------------------------------- PIN Declarations ---------------
PIN  1          CLK
PIN  2          PE1IRQ      COMB
PIN  3          PE0IRQ      COMB
PIN  4         /PE2IRQ      COMB        ;NOTE: PE2IRQ is active LOW
PIN  5          PE3IRQ      COMB
PIN  6         /BMASK0      COMB
PIN  7          BMASK2      COMB        ;NOTE: BMASK2 is active HIGH
PIN  8         /BMASK3      COMB
PIN  9          PE0B1       COMB
PIN  10         PE2B1       COMB
PIN  11         PE3B1       COMB
PIN  12         GND

PIN  24         VCC
PIN  23         PRVSTRB     REGISTERED  ;NC
PIN  22         X           REGISTERED  ;NC
PIN  21         Y           REGISTERED  ;NC
PIN  20         RDY         REGISTERED
PIN  19         Z           REGISTERED  ;NC
PIN  18         DATACLK     REGISTERED  ;Used to clock the Data PAL
PIN  17        /WAITLED     REGISTERED  ;Used to drive the wait LED
PIN  16         PE1B0       REGISTERED
PIN  15         PE1B2       REGISTERED
PIN  14         PE1B3       REGISTERED
```

```
PIN  13         STROBE      COMB

STRING CHANGED '(PRVSTRB :+: STROBE)'       ;NOTE: Operator :+: is XOR.
STRING ASYNCALT '(/BMASK0*/BMASK2*/BMASK3)'
STRING ASYNCRST '(PE1IRQ*/STROBE*/BMASK0*/BMASK2*/BMASK3)'
STRING ASYNCINT '((PE0IRQ*PE0B1 + PE2IRQ*PE2B1 + PE3IRQ*PE3B1)*/PE1IRQ)'

;***** The State Machine *****
STRING sA '(/DATACLK*/X* Y* Z)';0011 A: idle, RDY=1
;       if CHANGED==1, go to State B, else stay in State A
STRING sB '(/DATACLK*/X* Y*/Z)';0010 B: debounce
;       if CHANGED==0, go to State A, else latch b-mask, RDY=0, go to State C
STRING sR '(/DATACLK*/X*/Y*/Z)';0000 R: reset, RDY=0, PRVSTRB=1, clear b-mask
;       if ASYNCRST==0, go to State D
STRING sC '(/DATACLK*/X*/Y* Z)';0001 C: wait, maintain b-mask
;       if GO==1, go to State D and latch the data, else stay in State C
STRING sD '( DATACLK*/X*/Y* Z)';1001 D: clear b-mask, toggle PRVSTRB, go to E
STRING sE '( DATACLK*/X*/Y*/Z)';1000 E: delay 2, go to F
STRING sF '( DATACLK*/X* Y*/Z)';1010 F: delay 3, go to G
STRING sG '( DATACLK*/X* Y* Z)';1011 G: delay 4, go to H
STRING sH '( DATACLK* X* Y* Z)';1111 H: delay 5, skip I,J,K,L,M,N, go to O
STRING sI '( DATACLK* X* Y*/Z)';1110 I: delay 6, go to J
STRING sJ '( DATACLK* X*/Y*/Z)';1100 J: delay 7, go to K
STRING sK '( DATACLK* X*/Y* Z)';1101 K: delay 8, go to L
STRING sL '(/DATACLK* X*/Y* Z)';0101 L: delay 9, go to M
STRING sM '(/DATACLK* X*/Y*/Z)';0100 M: delay10, go to N
STRING sN '(/DATACLK* X* Y*/Z)';0110 N: delay11, go to O
STRING sO '(/DATACLK* X* Y* Z)';0111 O: delay 6, go to A
;---------------------------------- Boolean Equation Segment ------
EQUATIONS
WAITLED=sR + sC
RDY=((sA)*/ASYNCALT + (/ASYNCINT*ASYNCALT))
;RDY=((sA + sO + (sB*/CHANGED))*/ASYNCALT + (ASYNCINT*ASYNCALT))
DATACLK=(/ASYNCRST)*(sD + sE + sF + sG + sI + sJ + sR + /ASYNCINT*sC*(
  (/PE1IRQ*(/PE1B0+PE0B1       )*(/PE1B2+PE2B1       )*(/PE1B3+PE3B1       ))
  +(PE1IRQ*(/PE1B0+PE0B1*PE0IRQ)*(/PE1B2+PE2B1*PE2IRQ)*(/PE1B3+PE3B1*PE3IRQ))
 ))
X=(/ASYNCRST)*(sG + sH + sI + sJ + sK + sL + sM + sN)
Y=(/ASYNCRST)*(sE + sF + sG + sH + sM + sN + sO + sA + sB*/CHANGED)
Z=(/ASYNCRST)*(sR + sC + sF + sG + sJ + sK + sN + sO + sB + sA*/CHANGED)
PE1B0=sB*CHANGED*BMASK0 + sC*PE1B0
PE1B2=sB*CHANGED*BMASK2 + sC*PE1B2
PE1B3=sB*CHANGED*BMASK3 + sC*PE1B3
PRVSTRB=(sD :+: PRVSTRB) + sR
```

## B.3 Control/Barrier Chip Equations for PE2

```
;PALASM Design Description
;--------------------------------- Declaration Segment ------------
TITLE    Papers1 PE2 Barrier Chip
PATTERN
REVISION
AUTHOR
COMPANY
DATE     8/1/94
CHIP  _papers1  PALCE22V10
;--------------------------------- PIN Declarations ---------------
PIN  1          CLK
PIN  2         /PE2IRQ      COMB        ;NOTE: PE2IRQ is active LOW
PIN  3          PE0IRQ      COMB
```

```
PIN  4          PE1IRQ      COMB
PIN  5          PE3IRQ      COMB
PIN  6         /BMASK0      COMB
PIN  7         /BMASK1      COMB
PIN  8         /BMASK3      COMB
PIN  9          PE0B2       COMB
PIN  10         PE1B2       COMB
PIN  11         PE3B2       COMB
PIN  12         GND

PIN  24         VCC
PIN  23         PRVSTRB     REGISTERED  ;NC
PIN  22         X           REGISTERED  ;NC
PIN  21         Y           REGISTERED  ;NC
PIN  20         RDY         REGISTERED
PIN  19         Z           REGISTERED  ;NC
PIN  18         DATACLK     REGISTERED  ;Used to clock the Data PAL
PIN  17        /WAITLED     REGISTERED  ;Used to drive the wait LED
PIN  16         PE2B0       REGISTERED
PIN  15         PE2B1       REGISTERED
PIN  14         PE2B3       REGISTERED
PIN  13         STROBE      COMB

STRING CHANGED '(PRVSTRB :+: STROBE)'       ;NOTE: Operator :+: is XOR.
STRING ASYNCALT '(/BMASK0*/BMASK1*/BMASK3)'
STRING ASYNCRST '(PE2IRQ*/STROBE*/BMASK0*/BMASK1*/BMASK3)'
STRING ASYNCINT '((PE0IRQ*PE0B2 + PE1IRQ*PE1B2 + PE3IRQ*PE3B2)*/PE2IRQ)'

;***** The State Machine *****
STRING sA '(/DATACLK*/X* Y* Z)';0011 A: idle, RDY=1
;       if CHANGED==1, go to State B, else stay in State A
STRING sB '(/DATACLK*/X* Y*/Z)';0010 B: debounce
;       if CHANGED==0, go to State A, else latch b-mask, RDY=0, go to State C
STRING sR '(/DATACLK*/X*/Y*/Z)';0000 R: reset, RDY=0, PRVSTRB=1, clear b-mask
;       if ASYNCRST==0, go to State D
STRING sC '(/DATACLK*/X*/Y* Z)';0001 C: wait, maintain b-mask
;       if GO==1, go to State D and latch the data, else stay in State C
STRING sD '( DATACLK*/X*/Y* Z)';1001 D: clear b-mask, toggle PRVSTRB, go to E
STRING sE '( DATACLK*/X*/Y*/Z)';1000 E: delay 2, go to F
STRING sF '( DATACLK*/X* Y*/Z)';1010 F: delay 3, go to G
STRING sG '( DATACLK*/X* Y* Z)';1011 G: delay 4, go to H
STRING sH '( DATACLK* X* Y* Z)';1111 H: delay 5, skip I,J,K,L,M,N, go to O
STRING sI '( DATACLK* X* Y*/Z)';1110 I: delay 6, go to J
STRING sJ '( DATACLK* X*/Y*/Z)';1100 J: delay 7, go to K
STRING sK '( DATACLK* X*/Y* Z)';1101 K: delay 8, go to L
STRING sL '(/DATACLK* X*/Y* Z)';0101 L: delay 9, go to M
STRING sM '(/DATACLK* X*/Y*/Z)';0100 M: delay10, go to N
STRING sN '(/DATACLK* X* Y*/Z)';0110 N: delay11, go to O
STRING sO '(/DATACLK* X* Y* Z)';0111 O: delay 6, go to A
;---------------------------------- Boolean Equation Segment ------
EQUATIONS
WAITLED=sR + sC
RDY=((sA)*/ASYNCALT + (/ASYNCINT*ASYNCALT))
;RDY=((sA + sO + (sB*/CHANGED))*/ASYNCALT + (ASYNCINT*ASYNCALT))
DATACLK=(/ASYNCRST)*(sD + sE + sF + sG + sI + sJ + sR + /ASYNCINT*sC*(
  (/PE2IRQ*(/PE2B0+PE0B2       )*(/PE2B1+PE1B2       )*(/PE2B3+PE3B2       ))
  +(PE2IRQ*(/PE2B0+PE0B2*PE0IRQ)*(/PE2B1+PE1B2*PE1IRQ)*(/PE2B3+PE3B2*PE3IRQ))
 ))
X=(/ASYNCRST)*(sG + sH + sI + sJ + sK + sL + sM + sN)
Y=(/ASYNCRST)*(sE + sF + sG + sH + sM + sN + sO + sA + sB*/CHANGED)
Z=(/ASYNCRST)*(sR + sC + sF + sG + sJ + sK + sN + sO + sB + sA*/CHANGED)
PE2B0=sB*CHANGED*BMASK0 + sC*PE2B0
PE2B1=sB*CHANGED*BMASK1 + sC*PE2B1
```

```
PE2B3=sB*CHANGED*BMASK3 + sC*PE2B3
PRVSTRB=(sD :+: PRVSTRB) + sR
```

## B.4 Control/Barrier Chip Equations for PE3

```
;PALASM Design Description
;---------------------------------- Declaration Segment ------------
TITLE    Papers1 PE3 Barrier Chip
PATTERN
REVISION
AUTHOR
COMPANY
DATE     8/1/94
CHIP  _papers1  PALCE22V10
;---------------------------------- PIN Declarations ---------------
PIN  1          CLK
PIN  2          PE3IRQ      COMB
PIN  3          PE0IRQ      COMB
PIN  4          PE1IRQ      COMB
PIN  5         /PE2IRQ      COMB        ;NOTE: PE2IRQ is active LOW
PIN  6         /BMASK0      COMB
PIN  7         /BMASK1      COMB
PIN  8          BMASK2      COMB        ;NOTE: BMASK2 is active HIGH
PIN  9          PE0B3       COMB
PIN  10         PE1B3       COMB
PIN  11         PE2B3       COMB
PIN  12         GND

PIN  24         VCC
PIN  23         PRVSTRB     REGISTERED  ;NC
PIN  22         X           REGISTERED  ;NC
PIN  21         Y           REGISTERED  ;NC
PIN  20         RDY         REGISTERED
PIN  19         Z           REGISTERED  ;NC
PIN  18         DATACLK     REGISTERED  ;Used to clock the Data PAL
PIN  17        /WAITLED     REGISTERED  ;Used to drive the wait LED
PIN  16         PE3B0       REGISTERED
PIN  15         PE3B1       REGISTERED
PIN  14         PE3B2       REGISTERED
PIN  13         STROBE      COMB

STRING CHANGED '(PRVSTRB :+: STROBE)'       ;NOTE: Operator :+: is XOR.
STRING ASYNCALT '(/BMASK0*/BMASK1*/BMASK2)'
STRING ASYNCRST '(PE3IRQ*/STROBE*/BMASK0*/BMASK1*/BMASK2)'
STRING ASYNCINT '((PE0IRQ*PE0B3 + PE1IRQ*PE1B3 + PE2IRQ*PE2B3)*/PE3IRQ)'

;***** The State Machine *****
STRING sA '(/DATACLK*/X* Y* Z)';0011 A: idle, RDY=1
;       if CHANGED==1, go to State B, else stay in State A
STRING sB '(/DATACLK*/X* Y*/Z)';0010 B: debounce
;       if CHANGED==0, go to State A, else latch b-mask, RDY=0, go to State C
STRING sR '(/DATACLK*/X*/Y*/Z)';0000 R: reset, RDY=0, PRVSTRB=1, clear b-mask
;       if ASYNCRST==0, go to State D
STRING sC '(/DATACLK*/X*/Y* Z)';0001 C: wait, maintain b-mask
;       if GO==1, go to State D and latch the data, else stay in State C
STRING sD '( DATACLK*/X*/Y* Z)';1001 D: clear b-mask, toggle PRVSTRB, go to E
STRING sE '( DATACLK*/X*/Y*/Z)';1000 E: delay 2, go to F
STRING sF '( DATACLK*/X* Y*/Z)';1010 F: delay 3, go to G
STRING sG '( DATACLK*/X* Y* Z)';1011 G: delay 4, go to H
STRING sH '( DATACLK* X* Y* Z)';1111 H: delay 5, skip I,J,K,L,M,N, go to O
```

```
STRING sI '( DATACLK* X* Y*/Z)';1110 I: delay 6, go to J
STRING sJ '( DATACLK* X*/Y*/Z)';1100 J: delay 7, go to K
STRING sK '( DATACLK* X*/Y* Z)';1101 K: delay 8, go to L
STRING sL '(/DATACLK* X*/Y* Z)';0101 L: delay 9, go to M
STRING sM '(/DATACLK* X*/Y*/Z)';0100 M: delay10, go to N
STRING sN '(/DATACLK* X* Y*/Z)';0110 N: delay11, go to O
STRING sO '(/DATACLK* X* Y* Z)';0111 O: delay 6, go to A
;------------------------------------- Boolean Equation Segment ------
EQUATIONS
WAITLED=sR + sC
RDY=((sA)*/ASYNCALT + (/ASYNCINT*ASYNCALT))
;RDY=((sA + sO + (sB*/CHANGED))*/ASYNCALT + (ASYNCINT*ASYNCALT))
DATACLK=(/ASYNCRST)*(sD + sE + sF + sG + sI + sJ + sR + /ASYNCINT*sC*(
  (/PE3IRQ*(/PE3B0+PE0B3       )*(/PE3B1+PE1B3       )*(/PE3B2+PE2B3       ))
  +(PE3IRQ*(/PE3B0+PE0B3*PE0IRQ)*(/PE3B1+PE1B3*PE1IRQ)*(/PE3B2+PE2B3*PE2IRQ))
 ))
X=(/ASYNCRST)*(sG + sH + sI + sJ + sK + sL + sM + sN)
Y=(/ASYNCRST)*(sE + sF + sG + sH + sM + sN + sO + sA + sB*/CHANGED)
Z=(/ASYNCRST)*(sR + sC + sF + sG + sJ + sK + sN + sO + sB + sA*/CHANGED)
PE3B0=sB*CHANGED*BMASK0 + sC*PE3B0
PE3B1=sB*CHANGED*BMASK1 + sC*PE3B1
PE3B2=sB*CHANGED*BMASK2 + sC*PE3B2
PRVSTRB=(sD :+: PRVSTRB) + sR
```

## B.5 Data Chip Equations for PE0

```
;PALASM Design Description
;------------------------------------ Declaration Segment ------------
TITLE    PAPERS1 PE0 Data Chip
PATTERN
REVISION
AUTHOR
COMPANY
DATE     8/1/94
CHIP  _pport  PALCE22V10
;------------------------------------ PIN Declarations ---------------
PIN     1       CLK
PIN     2       PE1D0   COMB
PIN     3       PE1D1   COMB
PIN     4       PE1D2   COMB
PIN     5       PE1D3   COMB
PIN     6       PE2D0   COMB
PIN     7       PE2D1   COMB
PIN     8       PE2D2   COMB
PIN     9       PE2D3   COMB
PIN     10      SRC0    COMB
PIN     11      SRC1    COMB
PIN     12      GND

PIN     24      VCC
PIN     23      PE3D0   COMB
PIN     22      PE3D1   COMB
PIN     21      PE3D2   COMB
PIN     20      PE3D3   COMB
PIN     19      I0      REGISTERED
PIN     18      I1      REGISTERED
PIN     17      I2      REGISTERED
PIN     16     /I3      REGISTERED
PIN     15      PE0B1   COMB
PIN     14      PE0B2   COMB
```

```
PIN     13      PE0B3   COMB
;---------------------------------- Boolean Equation Segment ------
EQUATIONS

CASE (SRC1,SRC0)
  BEGIN
    0: BEGIN
      I0 = /PE1D0*PE0B1 + /PE2D0*PE0B2 + /PE3D0*PE0B3 + /PE0B1*/PE0B2*/PE0B3
      I1 = /PE1D1*PE0B1 + /PE2D1*PE0B2 + /PE3D1*PE0B3
      I2 = /PE1D2*PE0B1 + /PE2D2*PE0B2 + /PE3D2*PE0B3
      I3 = /PE1D3*PE0B1 + /PE2D3*PE0B2 + /PE3D3*PE0B3
      END
    1: BEGIN
      I0 = PE1D0*/(/PE0B1*/PE0B2*/PE0B3) + /PE0B1*/PE0B2*/PE0B3
      I1 = PE1D1*/(/PE0B1*/PE0B2*/PE0B3)
      I2 = PE1D2*/(/PE0B1*/PE0B2*/PE0B3)
      I3 = PE1D3*/(/PE0B1*/PE0B2*/PE0B3)
      END
    2: BEGIN
      I0 = PE2D0*/(/PE0B1*/PE0B2*/PE0B3) + /PE0B1*/PE0B2*/PE0B3
      I1 = PE2D1*/(/PE0B1*/PE0B2*/PE0B3)
      I2 = PE2D2*/(/PE0B1*/PE0B2*/PE0B3)
      I3 = PE2D3*/(/PE0B1*/PE0B2*/PE0B3)
      END
    3: BEGIN
      I0 = PE3D0*/(/PE0B1*/PE0B2*/PE0B3) + /PE0B1*/PE0B2*/PE0B3
      I1 = PE3D1*/(/PE0B1*/PE0B2*/PE0B3)
      I2 = PE3D2*/(/PE0B1*/PE0B2*/PE0B3)
      I3 = PE3D3*/(/PE0B1*/PE0B2*/PE0B3)
      END
  END
;-------------------------------------------------------------------
```

## B.6 Data Chip Equations for PE1

```
;PALASM Design Description
;--------------------------------- Declaration Segment ------------
TITLE    PAPERS1 PE1 Data Chip
PATTERN
REVISION
AUTHOR
COMPANY
DATE     8/1/94
CHIP  _pport  PALCE22V10
;--------------------------------- PIN Declarations ---------------
PIN     1       CLK
PIN     2       PE0D0   COMB
PIN     3       PE0D1   COMB
PIN     4       PE0D2   COMB
PIN     5       PE0D3   COMB
PIN     6       PE2D0   COMB
PIN     7       PE2D1   COMB
PIN     8       PE2D2   COMB
PIN     9       PE2D3   COMB
PIN     10      SRC0    COMB
PIN     11      SRC1    COMB
PIN     12      GND

PIN     24      VCC
PIN     23      PE3D0   COMB
```

```
PIN     22      PE3D1   COMB
PIN     21      PE3D2   COMB
PIN     20      PE3D3   COMB
PIN     19      I0      REGISTERED
PIN     18      I1      REGISTERED
PIN     17      I2      REGISTERED
PIN     16     /I3      REGISTERED
PIN     15      PE1B0   COMB
PIN     14      PE1B2   COMB
PIN     13      PE1B3   COMB
;---------------------------------- Boolean Equation Segment ------
EQUATIONS

CASE (SRC1,SRC0)
  BEGIN
    1: BEGIN
      I0 = /PE0D0*PE1B0 + /PE2D0*PE1B2 + /PE3D0*PE1B3
      I1 = /PE0D1*PE1B0 + /PE2D1*PE1B2 + /PE3D1*PE1B3 + /PE1B0*/PE1B2*/PE1B3
      I2 = /PE0D2*PE1B0 + /PE2D2*PE1B2 + /PE3D2*PE1B3
      I3 = /PE0D3*PE1B0 + /PE2D3*PE1B2 + /PE3D3*PE1B3
      END
    0: BEGIN
      I0 = PE0D0*/(/PE1B0*/PE1B2*/PE1B3)
      I1 = PE0D1*/(/PE1B0*/PE1B2*/PE1B3) + /PE1B0*/PE1B2*/PE1B3
      I2 = PE0D2*/(/PE1B0*/PE1B2*/PE1B3)
      I3 = PE0D3*/(/PE1B0*/PE1B2*/PE1B3)
      END
    2: BEGIN
      I0 = PE2D0*/(/PE1B0*/PE1B2*/PE1B3)
      I1 = PE2D1*/(/PE1B0*/PE1B2*/PE1B3) + /PE1B0*/PE1B2*/PE1B3
      I2 = PE2D2*/(/PE1B0*/PE1B2*/PE1B3)
      I3 = PE2D3*/(/PE1B0*/PE1B2*/PE1B3)
      END
    3: BEGIN
      I0 = PE3D0*/(/PE1B0*/PE1B2*/PE1B3)
      I1 = PE3D1*/(/PE1B0*/PE1B2*/PE1B3) + /PE1B0*/PE1B2*/PE1B3
      I2 = PE3D2*/(/PE1B0*/PE1B2*/PE1B3)
      I3 = PE3D3*/(/PE1B0*/PE1B2*/PE1B3)
      END
  END
;-------------------------------------------------------------------
```

## B.7 Data Chip Equations for PE2

```
;PALASM Design Description
;--------------------------------- Declaration Segment ------------
TITLE    PAPERS1 PE2 Data Chip
PATTERN
REVISION
AUTHOR
COMPANY
DATE     8/1/94
CHIP  _pport  PALCE22V10
;--------------------------------- PIN Declarations ---------------
PIN     1       CLK
PIN     2       PE0D0   COMB
PIN     3       PE0D1   COMB
PIN     4       PE0D2   COMB
PIN     5       PE0D3   COMB
PIN     6       PE1D0   COMB
```

```
PIN     7       PE1D1   COMB
PIN     8       PE1D2   COMB
PIN     9       PE1D3   COMB
PIN     10      SRC0    COMB
PIN     11      SRC1    COMB
PIN     12      GND

PIN     24      VCC
PIN     23      PE3D0   COMB
PIN     22      PE3D1   COMB
PIN     21      PE3D2   COMB
PIN     20      PE3D3   COMB
PIN     19      I0      REGISTERED
PIN     18      I1      REGISTERED
PIN     17      I2      REGISTERED
PIN     16     /I3      REGISTERED
PIN     15      PE2B0   COMB
PIN     14      PE2B1   COMB
PIN     13      PE2B3   COMB
;---------------------------------- Boolean Equation Segment ------
EQUATIONS

CASE (SRC1,SRC0)
  BEGIN
    2: BEGIN
      I0 = /PE0D0*PE2B0 + /PE1D0*PE2B1 + /PE3D0*PE2B3
      I1 = /PE0D1*PE2B0 + /PE1D1*PE2B1 + /PE3D1*PE2B3
      I2 = /PE0D2*PE2B0 + /PE1D2*PE2B1 + /PE3D2*PE2B3 + /PE2B0*/PE2B1*/PE2B3
      I3 = /PE0D3*PE2B0 + /PE1D3*PE2B1 + /PE3D3*PE2B3
      END
    1: BEGIN
      I0 = PE1D0*/(/PE2B0*/PE2B1*/PE2B3)
      I1 = PE1D1*/(/PE2B0*/PE2B1*/PE2B3)
      I2 = PE1D2*/(/PE2B0*/PE2B1*/PE2B3) + /PE2B0*/PE2B1*/PE2B3
      I3 = PE1D3*/(/PE2B0*/PE2B1*/PE2B3)
      END
    0: BEGIN
      I0 = PE0D0*/(/PE2B0*/PE2B1*/PE2B3)
      I1 = PE0D1*/(/PE2B0*/PE2B1*/PE2B3)
      I2 = PE0D2*/(/PE2B0*/PE2B1*/PE2B3) + /PE2B0*/PE2B1*/PE2B3
      I3 = PE0D3*/(/PE2B0*/PE2B1*/PE2B3)
      END
    3: BEGIN
      I0 = PE3D0*/(/PE2B0*/PE2B1*/PE2B3)
      I1 = PE3D1*/(/PE2B0*/PE2B1*/PE2B3)
      I2 = PE3D2*/(/PE2B0*/PE2B1*/PE2B3) + /PE2B0*/PE2B1*/PE2B3
      I3 = PE3D3*/(/PE2B0*/PE2B1*/PE2B3)
      END
  END
;------------------------------------------------------------------
```

## B.8 Data Chip Equations for PE3

```
;PALASM Design Description
;--------------------------------- Declaration Segment ------------
TITLE    PAPERS1 PE3 Data Chip
PATTERN
REVISION
AUTHOR
COMPANY
```

```
DATE     8/1/94
CHIP  _pport  PALCE22V10
;---------------------------------- PIN Declarations ---------------
PIN     1       CLK
PIN     2       PE0D0   COMB
PIN     3       PE0D1   COMB
PIN     4       PE0D2   COMB
PIN     5       PE0D3   COMB
PIN     6       PE1D0   COMB
PIN     7       PE1D1   COMB
PIN     8       PE1D2   COMB
PIN     9       PE1D3   COMB
PIN     10      SRC0    COMB
PIN     11      SRC1    COMB
PIN     12      GND

PIN     24      VCC
PIN     23      PE2D0   COMB
PIN     22      PE2D1   COMB
PIN     21      PE2D2   COMB
PIN     20      PE2D3   COMB
PIN     19      I0      REGISTERED
PIN     18      I1      REGISTERED
PIN     17      I2      REGISTERED
PIN     16     /I3      REGISTERED
PIN     15      PE3B0   COMB
PIN     14      PE3B1   COMB
PIN     13      PE3B2   COMB
;----------------------------------- Boolean Equation Segment ------
EQUATIONS

CASE (SRC1,SRC0)
  BEGIN
    3: BEGIN
      I0 = /PE0D0*PE3B0 + /PE1D0*PE3B1 + /PE2D0*PE3B2
      I1 = /PE0D1*PE3B0 + /PE1D1*PE3B1 + /PE2D1*PE3B2
      I2 = /PE0D2*PE3B0 + /PE1D2*PE3B1 + /PE2D2*PE3B2
      I3 = /PE0D3*PE3B0 + /PE1D3*PE3B1 + /PE2D3*PE3B2 + /PE3B0*/PE3B1*/PE3B2
      END
    1: BEGIN
      I0 = PE1D0*/(/PE3B0*/PE3B1*/PE3B2)
      I1 = PE1D1*/(/PE3B0*/PE3B1*/PE3B2)
      I2 = PE1D2*/(/PE3B0*/PE3B1*/PE3B2)
      I3 = PE1D3*/(/PE3B0*/PE3B1*/PE3B2) + /PE3B0*/PE3B1*/PE3B2
      END
    2: BEGIN
      I0 = PE2D0*/(/PE3B0*/PE3B1*/PE3B2)
      I1 = PE2D1*/(/PE3B0*/PE3B1*/PE3B2)
      I2 = PE2D2*/(/PE3B0*/PE3B1*/PE3B2)
      I3 = PE2D3*/(/PE3B0*/PE3B1*/PE3B2) + /PE3B0*/PE3B1*/PE3B2
      END
    0: BEGIN
      I0 = PE0D0*/(/PE3B0*/PE3B1*/PE3B2)
      I1 = PE0D1*/(/PE3B0*/PE3B1*/PE3B2)
      I2 = PE0D2*/(/PE3B0*/PE3B1*/PE3B2)
      I3 = PE0D3*/(/PE3B0*/PE3B1*/PE3B2) + /PE3B0*/PE3B1*/PE3B2
      END
  END
;------------------------------------------------------------------
```