# Chapter 3

# The design and prototyping of the PASM reconfigurable parallel processing system

Howard Jay Siegel, Thomas Schwederski, Wayne G. Nation, James B. Armstrong, Lee Wang, James T. Kuehn, Rohit Gupta, Mark D. Allemang, David G. Meyer and Daniel W. Watson

## 3.1 Introduction

Complex tasks that need parallel processing to meet their execution time constraints often consist of numerous subtasks with differing computational characteristics. It is unreasonable to expect a single machine to be an optimal match for all of these different computational needs. One may attempt to maximize performance in such situations by using a reconfigurable system that can continually adapt its architecture based on the structure of each subtask. This is an overview of the PASM reconfigurable parallel processing system architectural organization and a hardware description of the control hierarchy of the small-scale proof-of-concept prototype. The prototype was constructed to validate design concepts and to serve as a tool for studying issues related to the use of reconfigurable parallel machines.

One type of parallel system, called an **SIMD** (single instruction stream, multiple data stream) (Flynn, 1966) machine, consists of $N$ processors, $N$ memories, an interconnection network and a control unit. In the SIMD mode of parallelism, all enabled processors execute the same instruction at the same time, but each on its own data. The **control unit** broadcasts instructions in sequence to the $N$ processors that execute these instructions in lockstep. The operand data for these instructions are fetched from the memory associated with each processor. The **interconnection network** permits interprocessor communication. Examples of SIMD systems that have been constructed are ASPRO (Batcher, 1982), CLIP4 (Fountain, 1981), DAP (Hunt, 1989), Illiac IV (Bouknight *et al.*, 1972; Hord, 1982), MasPar MP-1 and MP-2 (Blank, 1990), MPP (Batcher, 1982) and STARAN (Batcher, 1976).

An **MSIMD** (multiple-SIMD) system can be structured as one or more independent SIMD machines of various sizes. The proposed MAP (Nutt, 1977) and existing Thinking Machines CM-2 (Tucker and Robertson, 1988) are examples of MSIMD systems. The Illiac IV was originally designed as an MSIMD system (Barnes *et al.*, 1968).



In the **MIMD** (multiple instruction stream, multiple data stream) (Flynn, 1966) mode of parallelism, each processor has its own instructions and data. MIMD systems are typically structured like SIMD systems without the control unit, i.e. $N$ processors, $N$ memories, an interconnection network and multiple data streams. The BBN Butterfly (Crowther *et al.*, 1985), Caltech Cosmic Cube (Seitz, 1985), Intel iPSC cube (Nugent, 1988), nCUBE 2 (Hayes and Mudge, 1989), Intel Paragon (Zorpette, 1992), Thinking Machines CM-5 (Hillis and Tucker, 1993), Cray T3D and IBM RP3 (Pfister *et al.*, 1985) are examples of MIMD systems that have been constructed.

A **mixed-mode** system can dynamically switch between the SIMD and MIMD modes of parallelism at instruction-level granularity with nominal overhead. This permits different modes of parallelism to be used to execute various portions of an algorithm. Because the mode of parallelism has an impact on performance, a mixed-mode system may outperform a single-mode machine with the same number of processors for a given algorithm. A limited mixed-mode machine is OPSILA (Duclos *et al.*, 1988), an existing system that can switch between the SIMD and **SPMD** (single program, multiple data stream) (Darema *et al.*, 1988) modes of parallelism. SPMD mode is a form of MIMD mode where all processors execute the same program. The prototype Triton/1 (Philippsen *et al.*, 1993) machine is capable of full mixed-mode parallelism.

A **partitionable mixed-mode** system can dynamically reconfigure to form independent or communicating submachines of various sizes, where each submachine employs mixed-mode parallelism (e.g. TRAC (Lipovski and Malek, 1987)). **PASM** is a PArtitionable-SIMD/MIMD system concept being developed at Purdue University as a design for a large-scale dynamically reconfigurable parallel processing system based on commodity microprocessors (Siegel *et al.*, 1981; Siegel *et al.*, 1987). Each submachine can independently perform mixed-mode parallelism. PASM uses a flexible multistage interconnection network for interprocessor communication. Thus, PASM is dynamically reconfigurable along the three dimensions of partitionability, mode of parallelism and connections among processors.

Originally, PASM was intended as a special-purpose system aimed at image understanding tasks, and it was the algorithmic and computational characteristics of these tasks that guided design decisions. Furthermore, mode switching was considered only at the subtask level. Experience with the prototype has shown that the PASM reconfiguration structure is effective for a great variety of application domains and that the capability to perform mode switching at the instruction level rather than subtask level is a powerful tool, e.g. Fineberg, Casavant and Siegel (1991).

The goal of the PASM research team is to develop a unique dynamically reconfigurable research tool for studying large-scale SIMD, MIMD and mixed-mode processing. The PASM concept is a distributed memory machine, with a computational engine consisting of processor/memory pairs referred to as **PEs** (processing elements). It attempts to incorporate the flexibility needed to investigate the numerous issues related to the design and use of

reconfigurable parallel processing systems. The number of PEs that can be included in a PASM system is a function of the technology and funds available at the time of construction. Based on current technology, the PASM design concepts could readily support at least 1024 PEs. The small-scale prototype built at Purdue University (30 processors, 16 in the computational engine) supports experimentation with all three of the dimensions of reconfigurability mentioned above, and has produced insights not attainable from earlier simulations and theoretical studies (e.g. Bronson, Casavant and Jamieson, 1990; Fineberg *et al.*, 1988; Fineberg, Casavant and Siegel, 1991). PASM can be reconfigured for fault tolerance, as well as for matching the computational structures of a problem.

The PASM architectural concepts are overviewed in section 3.2. A detailed description of the control hierarchy hardware of the prototype is given in section 3.3. Section 3.4 briefly presents some PASM prototype software issues. A discussion of some of the lessons that have been learned from the development and use of the PASM prototype is given in section 3.5. Reading lists of PASM-related publications are in Siegel *et al.* (1987) and Armstrong, Watson and Siegel (1993).

## 3.2 The PASM design concepts

### *3.2.1 Overview*

As shown in Figure 3.1, the PASM system concept consists of six basic components. The **System Control Unit** is the overall system coordinator and is the part of PASM with which the user directly interacts. The Micro Controllers (**MCs**) serve as the PE control units in SIMD mode and may coordinate the PEs in MIMD mode. The **Parallel Computation Unit** contains the $N = 2^n$ PEs, physically addressed 0 to $N-1$, and the interconnection network used by the PEs. The **Memory Storage System** provides secondary storage for the PEs. It is used to store data files for SIMD mode and both program and data files for MIMD mode. The **Memory Management System** controls the transferring

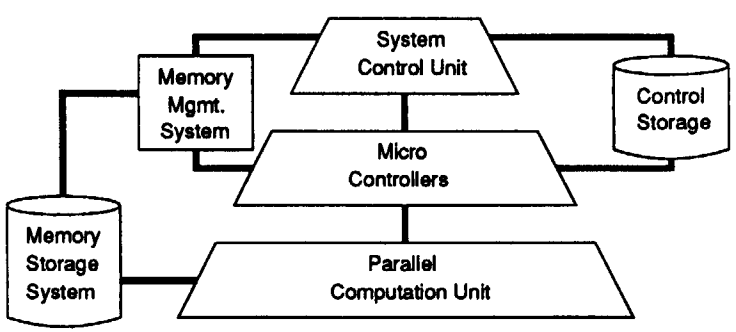


**Figure 3.1** *Block diagram of the PASM design concept.*



of files between the Memory Storage System and the PEs. **Control Storage** is the secondary storage for the MCs and the System Control Unit.

### *3.2.2 System Control Unit*

The tasks to be performed by the System Control Unit include support for program development, job scheduling, general system coordination, management of system configuration and partitioning, assignment of user jobs to submachines and connection to the host computer network. The hardware needed to combine and synchronize the MCs and PEs to form SIMD submachines of various sizes resides in the System Control Unit. It is also responsible for coordinating the loading of PE memories from the Memory Storage System with the loading of MC memories from the Control Storage.

Appropriate distribution of tasks between the System Control Unit and the other system components (e.g. the MCs and the Memory Management System) keep the System Control Unit from becoming a system bottleneck. In an $N = 1024$ system, the System Control Unit may be comprised of several processors. In the $N = 16$ prototype, the System Control Unit is a microprocessor, and program development is performed on the host computer network.

### *3.2.3 Micro Controllers*

The MCs, shown in Figure 3.2, are the multiple control units required to form an MSIMD system. There are $Q = 2^q$ MCs, physically addressed from 0 to $Q-1$. Each MC controls a fixed group of $N/Q$ PEs in the Parallel Computation

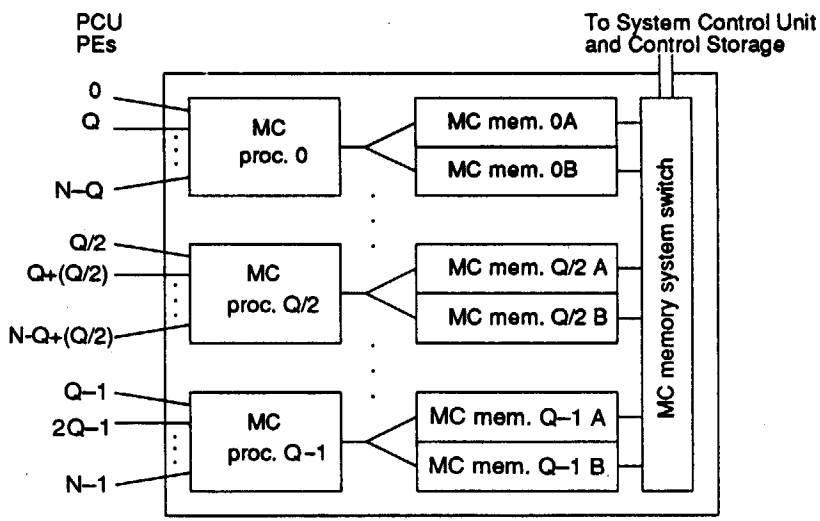


**Figure 3.2** *PASM Micro Controllers (MCs). 'PCU' is the Parallel Computation Unit.*

Unit. An MC and its associated PEs is an **MC group**. The physical addresses of all $N/Q$ PEs connected to an MC have the same low-order $q$ bits (for reasons discussed in section 3.2.4). The value of these low-order $q$ bits is the physical address of the MC. In an $N = 1024$ system, Q may be 32; for the $N = 16$ prototype, $Q = 4$.

Two memory units are provided in each MC so that computation and memory I/O can be overlapped. For example, the MC processor can execute a job in one memory unit while the next job is preloaded into the other memory unit from MC secondary storage (the Control Storage). In MIMD mode, an MC fetches from its memory the instructions and data used to coordinate the operation of its PEs. In SIMD mode, an MC fetches the instructions and common PE data from its memory units. In general, in SIMD mode control-flow instructions are executed in the MC, and data processing instructions are broadcast to the MC's group of PEs.

Submachines are formed by combining one or more MC groups. A submachine containing $R = 2^r$ MC groups ($RN/Q$ PEs), where $0 \leq r \leq q$, is formed by combining the PEs connected to $R$ MCs whose addresses agree in their $q - r$ low-order bits. The partitioning rule in PASM is that the addresses of all PEs in a submachine of size $2^p$ must agree in their $n - p$ low-order bit positions (for the reasons discussed in section 3.2.4). Thus, the addresses of all MCs in the submachine will agree in their $q - r$ low-order bits. There is a maximum of $Q$ submachines, each of size $N/Q$.

Each submachine can operate as an independent mixed-mode system. PEs can switch between modes as often as desired; however, all PEs in a submachine must be in the same mode (SIMD or MIMD) at any point in time.

When a submachine is operating in SIMD mode, the $R$ MCs must execute and broadcast the same instructions, and all PEs in the submachine must be synchronized. The MCs are given the same instructions simply by loading the memory units of the $R$ MCs with the same program (a shared memory alternative is discussed in Siegel *et al.* (1981)). The PEs are synchronized by providing a small amount of special circuitry that coordinates instruction broadcasts among these MCs (described for the prototype in section 3.3.5).

The MCs within a submachine of $RN/Q$ PEs are logically numbered (addressed) 0 to $R - 1$. For $R > 1$, the logical number of an MC is the high-order $r$ bits of its physical number. Similarly, the PEs assigned to a submachine are logically numbered (addressed) 0 to $(RN/Q) - 1$. The logical number of a PE is the high-order $r + n - q$ bits of its physical number.

Some advantages of this static PE to MC mapping as compared with a dynamic MC/PE interconnection (e.g. a crossbar switch (Nutt, 1977)) include reducing MC/PE interface hardware, eliminating the overhead of maintaining PE to MC assignments, scheduling only $Q$ MCs instead of $N$ PEs, allowing the partitioning of the PE interconnection network into independent subnetworks (section 3.2.4) and supporting the structure of the efficient parallel primary to secondary memory connections (section 3.2.6). The main disadvantage of this approach is that the size of each submachine must be a power of two, with a minimum of $N/Q$ PEs. However, for PASM's intended experimental environment, flexibility at 'reasonable' cost is the goal, not maximum PE utilization.



Masking schemes are used in SIMD mode to enable and disable PEs. The PASM system uses PE-address masks and data-conditional masks. A **PE-address mask** (Siegel, 1990) enables a set of PEs based solely on PE addresses; it does not depend on local PE data. A PE-address mask has $n$ positions, where each position contains either a 0, 1, or X ('don't care'). A PE is enabled only if the binary representation of its address matches the mask. For example, for $N = 64$, [5{X}{0}] is equivalent to [XXXXX0] and enables all even-numbered PEs. A **negative PE-address mask** enables all PEs whose addresses do not match the mask. PE-address masks are a convenient notation for enabling PEs in a large-scale parallel machine.

**Data-conditional masks** enable PEs based on some condition dependent on local PE data. Consequently, the resulting data condition may be true in some PEs and false in other PEs. An example use of data-conditional masking is the `where` statement, the SIMD counterpart to the `if-then-else` statement. When the segment

```
where <data-condition> do <where-part>
elsewhere <else-part>
```

is executed, each PE evaluates the `<data-condition>`. The PEs where the condition evaluates true execute the `<where-part>`, and the PEs where the condition evaluates false are idle. Next, the PEs where the condition evaluates false execute the `<else-part>`, and the PEs where the condition evaluated true are idle. This type of masking is used in many SIMD machines (e.g. CM-2, DAP, MP-1). The nesting of `where` statements may be done using an execution-time control stack (Nation *et al.*, 1990).

Data-conditional and PE-address masks can be used together. A hybrid masking scheme implementation has been examined (Nation *et al.*, 1990). Data-conditional masking may be supported in hardware by providing each PE with a **local enable bit stack**, which is used to indicate whether the PE is conditionally enabled or disabled. PE-address masks may be supported in hardware by providing each MC with an $N/Q$-bit **mask mode specifier** and an $N/Q$-bit **mask vector**. The mask mode specifier contains a single **mask mode bit** for each PE that indicates whether the PEs' local enable bits should be used when determining PE enable status. The mask vector contains a hardware decoded version of the PE-address mask (Siegel *et al.*, 1981). If bit $i$ of the mask vector is 1, then the $i$th PE of the MC group is enabled, otherwise the $i$th PE of the MC group is disabled. The mask mode bit and bit $i$ of the mask vector are sent to PE $i$ with each instruction. Hardware in PE $i$ can then combine bit $i$ of the mask vector, the local enable bit and the mask mode bit to determine whether PE $i$ is enabled. If the mask mode indicates unconditional masking, then the PE is enabled if its address matches the PE-address mask. If the mask mode indicates conditional masking, then the PE is enabled if its address matches the PE-address mask and its local enable bit is set to true.

There are situations in which the combined conditional status of all enabled PEs in an SIMD submachine is needed, e.g. `if-none`, `if-any` and `if-all`. For example, if the PEs of a submachine are each examining a portion of an

image to find certain objects, it is necessary to know whether any of the PEs have found such an object. Machines that support operations of this type include the CM-2 and MP-1. Because PASM is a partitionable system, operations such as these require conditional results to be communicated from the PEs to their MCs and subsequently combined among the MCs comprising an SIMD submachine. These operations may be efficiently supported by providing a small amount of special circuitry that combines MC group results according to the current system partitioning (described for the prototype in section 3.3.5).

### 3.2.4 *Parallel Computation Unit*

The Parallel Computation Unit, shown in Figure 3.3, contains $N = 2^n$ PEs and an interconnection network. Two memory units are used in each PE so that computation and memory I/O can be overlapped. For example, the PE processor can execute a job in one memory unit while the next job is preloaded into the other memory unit from PE secondary storage (the Memory Storage System). These memory units compose the primary memory of the system. In SIMD or MIMD mode, each PE can use its own PE address or local data as a basis for indirect memory addressing. The PE processors may be either commodity microprocessors or custom designed for parallel processing and/or a particular application.

The interconnection network allows PEs to communicate with each other. Two multistage networks were originally considered for PASM: the multistage cube and the augmented data manipulator (ADM) (Siegel, 1990; Siegel *et al.*, 1989). They were considered because of their overall flexibility and partitionability (each can be partitioned into independent subnetworks). As described earlier, the reconfiguration rule in PASM requires that the physical

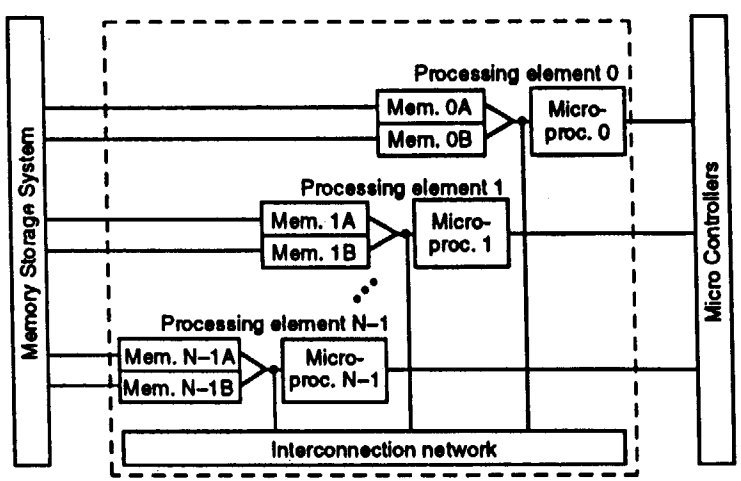


**Figure 3.3** *The Parallel Computation Unit.*



addresses of all PEs in a submachine of $2^p$ PEs agree in their $n - p$ low-order bit positions. Thus, the $p$ high-order bits of a PE's physical number form its logical number within a submachine of $2^p$ PEs. The low-order partitioning rule was chosen for PASM so that either of these two networks could be used. However, the multistage cube was selected because of its comparative cost-effectiveness.

The multistage cube network is representative of the class of networks that includes the baseline (Wu and Feng, 1980), delta (Patel, 1981), generalized cube (Siegel, 1990), indirect binary $n$-cube (Pease, 1977), multistage shuffle-exchange (Thanawastien and Nelson, 1981), omega (Lawrie, 1975) and SW-banyan ($S = F = 2$) (Lipovski and Malek, 1987) networks. This class of networks has been used in or proposed for use in many systems including the Butterfly, Cedar (Kuck *et al.*, 1986), dataflow machines (Dennis, Boughton and Leung, 1980), RP3, STARAN and NYU Ultracomputer (Gottlieb *et al.*, 1983).

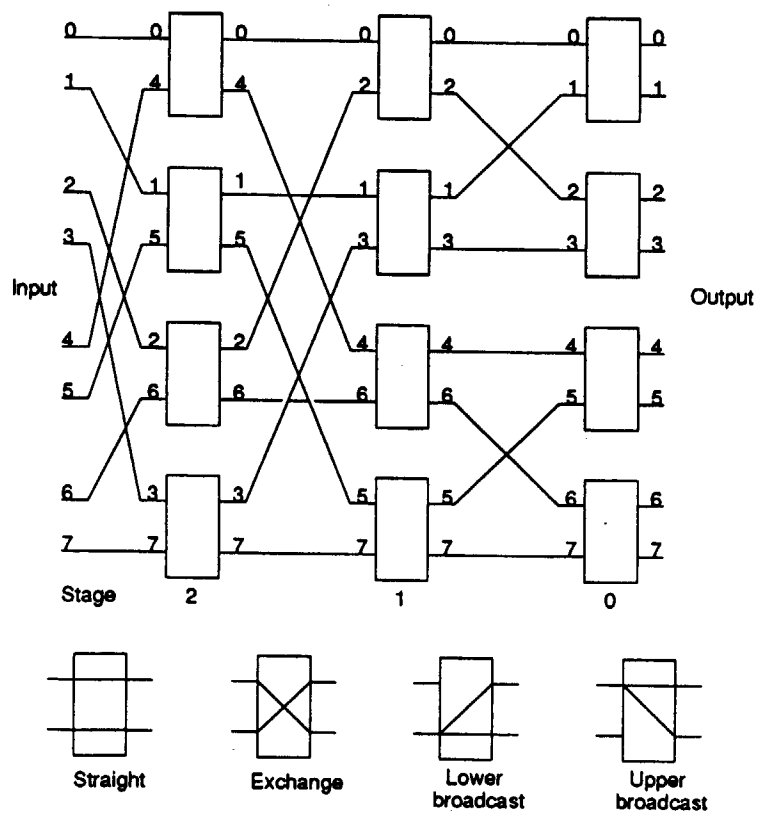


**Figure 3.4** *The multistage cube topology for* N = *8 and valid interchange box states.*

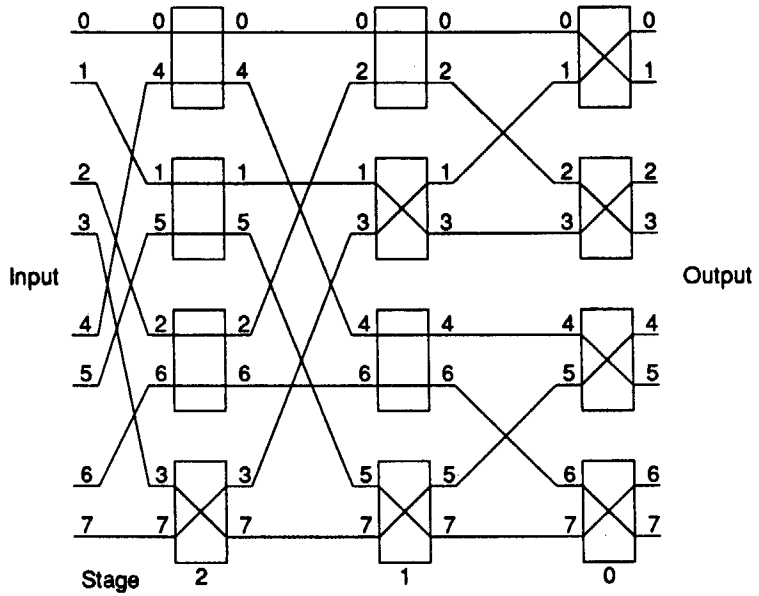


**Figure 3.5** *Example permutation for* N = *8 for input* i *to output* i+*1 mod 8.*

The **multistage cube network** has $N$ input ports, $N$ output ports, and contains $n = \log_2 N$ stages of $N/2$ 2 × 2 **interchange boxes**. Figure 3.4 shows the multistage cube network for $N = 8$. PE $i$ is connected to network input port $i$ and output port $i$. The stages are numbered consecutively from $n - 1$ at the input stage to 0 at the output stage. The upper box output label is the same as the upper input, and the lower box output label is the same as the lower input. The interconnection pattern between stages is such that at stage $j$ the two links whose labels differ only in bit $j$ are connected to the same interchange box. Each interchange box can be set to one of the four states shown in Figure 3.4. Figure 3.5 shows a permutation connection (input $i \rightarrow$ output $(i+1)$ mod 8) for an $N = 8$ multistage cube network.

One advantage of the multistage cube network is that network control may be distributed. Each network input device determines its own routing tag, which is used as a header on messages. A simple and efficient way to construct a routing tag is to set the message header tag to the $n$-bit address of the message destination (Lawrie, 1975). Each interchange box encountered by the message examines the destination routing tag to determine its own box state. If a message destined for output port $D = d_{n-1} \ldots d_1 d_0$, then at stage $i$ it is routed to the lower box output if $d_i$ is 1 and to the upper box output if $d_i$ is 0. Several benefits result from this routing method: the method is distributed, the routing tag can be used to verify that the message arrived at the correct destination, and the generation of tags is simple. The destination tags described can specify one-to-one and permutation connections, and can be extended to handle broadcast connections by adding an $n$-bit broadcast mask to the tag (Siegel, 1990).



A network is **partitionable** if it can be divided into independent subnetworks of smaller sizes that have all of the properties of the original network (Siegel, 1990). In general, a size $N$ multistage cube network can be partitioned into multiple subnetworks of different sizes, where the size of each subnetwork is a power of two. For example, Figure 3.6 shows an $N = 16$ network partitioned into two subnetworks by setting the interchange boxes in stage 0 to straight. Because stage 0 is straight, even-numbered inputs cannot reach odd-numbered outputs, and odd inputs cannot reach even outputs. The two resulting subnetworks are subnetwork A, which contains physical ports 0, 2, . . ., 14, and subnetwork B, which contains physical ports 1, 3, . . ., 15. Because each of these subnetworks is an independent multistage cube network of size $N/2 = 8$, either or both subnetworks may be further partitioned by forcing all interchange boxes in stage 1 of the subnetwork to straight.

In summary, the multistage cube network has many advantages. It has the ability to perform up to $N$ simultaneous transfers, to be partitioned into

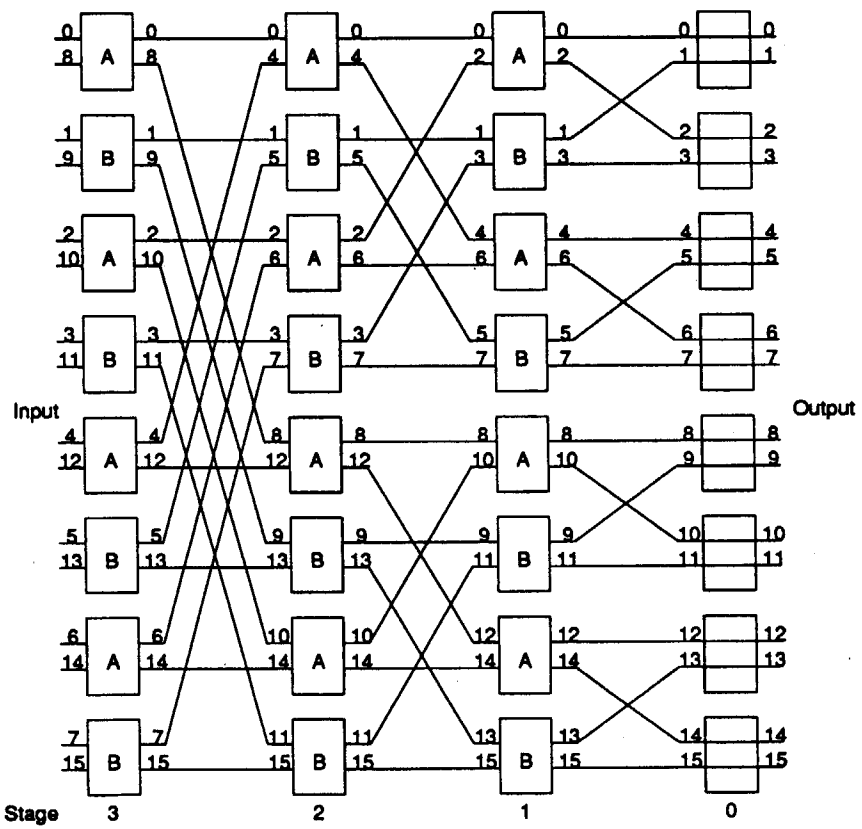


**Figure 3.6** N = *16 multistage cube network partitioned into two size eight subnetworks.*

multiple independent subnetworks of various sizes, and to perform one-to-one, permutation and broadcast transfers using routing tags. It can support efficient global as well as local (nearest neighbors) inter-PE communication, and may be used in both the SIMD and MIMD modes of parallelism. Finally, there are a variety of network implementation options (e.g. circuit switched vs. packet switched, 2 × 2 vs. 4 × 4 vs. 8 × 8 interchange boxes).

The multistage cube network, as presented here, has the disadvantage that only a single path exists for any source–destination pair. Thus, a single fault in the network makes many source–destination paths unavailable. The **Extra Stage Cube network** (Adams and Siegel, 1982; Siegel, 1990) is a single-fault-tolerant variation of the multistage cube network. There is an extra stage of interchange boxes at the input. Links whose labels differ in the 0th bit position are paired at these extra stage boxes in the same way that they are at stage 0. The Extra Stage Cube network is partitionable and may be controlled using routing tags. It is used in the prototype, and is discussed further in section 3.3.6.

### *3.2.5 Memory Storage System*

The Memory Storage System is the secondary storage for the Parallel Computation Unit, storing data files in SIMD mode and both program and data files in MIMD mode (program storage in SIMD mode is discussed in section 3.2.7). The Memory Storage System is comprised of $N/Q$ independent Memory Storage Units (**MSUs**), numbered from 0 to $(N/Q) - 1$. Each MSU contains a mass storage unit and a processor to manage the file system and to transfer files to and from its associated PE memory units.

Each of the $N/Q$ MSUs is connected to the memory modules of $Q$ PEs. MSU $i$ is connected to and stores files for the $Q$ PEs whose $n - q$ high-order address bits are equal to $i$. This high-order mapping is used so that each of

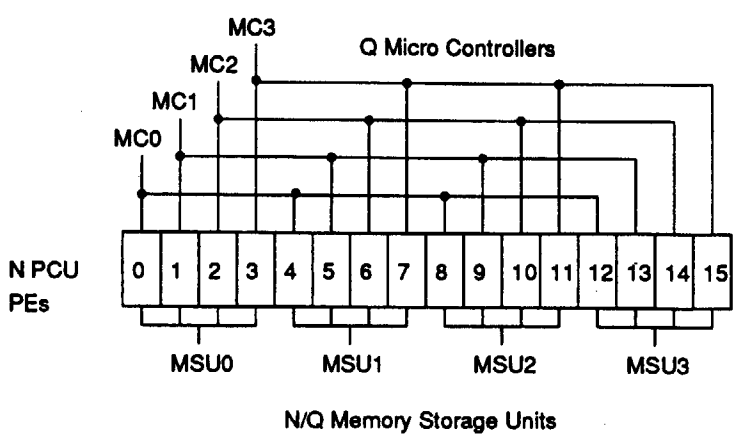


**Figure 3.7** *Organization of the Memory Storage System for* N = *16 and* Q = *4.*



the $N/Q$ PEs connected to an MC is connected to a different MSU. Recall that the low-order $q$ bits of the physical PE number correspond to the number of the MC to which it is attached (as described in section 3.2.3). An example for $N = 16$ and $Q = 4$ is shown in Figure 3.7.

One benefit of this MSU to PE connection scheme is that the memories of all $N/Q$ PEs connected to any one MC can be loaded or unloaded in one parallel block transfer. For example, in Figure 3.7, MC 2's group of PEs (PEs 2, 6, 10 and 14) are loaded by having MSU 0 load PE 2, MSU 1 load PE 6, MSU 2 load PE 10 and MSU 3 load PE 14. All four of these transfers may be done in parallel so that only one parallel block transfer is required to load all of the PEs connected to any one MC. If there are only $N/(QD)$ distinct MSUs, where $1 \leq D \leq N/Q$, then this scheme can be scaled so that $D$ parallel block transfers are required to load or unload the memories of the PEs connected to any one MC.

Similarly, a submachine formed by combining the $(RN)/Q$ PEs controlled by $R$ MCs can be loaded or unloaded in $R$ parallel block transfers if there are $N/Q$ MSUs and $RD$ parallel block transfers if there are $N/(QD)$ distinct MSUs. For example, in Figure 3.7, a submachine comprised of the PEs of MC 0 and MC 2 can be loaded in $R = 2$ parallel block loads as follows: first MC 0's PEs are loaded in one parallel block transfer, and then MC 2's PEs are loaded. Thus, the full bandwidth of the Memory Storage System can be used when transferring files between the Memory Storage System and the Parallel Computation Unit.

Data for a logical PE in a submachine is stored in the MSU whose number is the PE's $n - q$ high-order logical address bits (which equal its $n - q$ high-order physical address bits). As a result, no matter which MCs are assigned to a task, the data will be in the appropriate MSUs. For example, in Figure 3.7, for any submachine of size $N/Q = 4$, MSU 0 is connected to logical PE 0 (which could be physical PE 0, 1, 2, or 3).

### *3.2.6 Memory Management System*

The Memory Management System supervises file transfers between the $N$ PE memory modules in the Parallel Computation Unit and the $N/Q$ secondary storage devices in the Memory Storage System. As described in section 3.2.5, multiple PEs are connected to a single MSU. The Memory Management System determines which of the PEs connected to an MSU will be the source or destination for file transfers.

The Memory Storage System is composed of multiple processors that perform in a distributed manner the tasks required. The main functions that must be performed include passing file system requests from the System Control Unit, MCs, and PEs to the appropriate MSUs; controlling data transfers between the Memory Storage System and the PEs; supervising input/output operations involving peripheral devices; and enforcing consistent file naming and placement across the multiple Memory Storage System disks.

### 3.2.7 Control Storage

Control Storage is the mass storage for the MCs and the System Control Unit, holding all code and data used by these processors. It also contains the instructions to be broadcast to the PEs from the MCs in SIMD mode. It consists of a secondary storage device and a processor for managing the file system on the device. The Control Storage processor acts as a file server by responding to requests from the MCs and the System Control Unit.

### 3.2.8 Advantages of reconfigurability

As stated earlier, PASM is reconfigurable along three dimensions: partitionability, mode of parallelism and variable connectivity among PEs. Advantages of systems with such reconfigurability include:

1. *Multiple simultaneous users*: Because there can be multiple simultaneous independent submachines, there can be multiple simultaneous users of the system, each executing a different program.
2. *Program development*: Rather than trying to debug a new parallel program on, for example, 1024 PEs, it can be debugged on a smaller size submachine of 32 PEs, and then extended to 1024 PEs.
3. *Variable submachine size for increased utilization*: If a task requires only $N'$ of $N$ available PEs, the other $N - N'$ can be used for another task.
4. *Variable submachine size for decreased execution time*: There are some algorithms for which the minimum execution time is obtained when fewer than $N$ PEs are used (e.g. recursive doubling (Krishnamurti and Ma, 1988)); thus, it is desirable to create a submachine consisting of the optimal number of PEs.
5. *Subtask parallelism*: Two independent subtasks that are part of the same job can be executed in parallel, sharing results if necessary, which may result in improved overall task execution time (Nation, Maciejewski and Siegel, 1993).
6. *Multiple processing modes*: An algorithm can be executed by using a combination of SIMD and MIMD control with the same set of PEs (mixed-mode parallelism), using the mode that best matches the computations required at each step of a program.
7. *Matching inter-PE connectivity to the task*: The multistage cube allows different connection patterns among PEs to be established depending on the task (as opposed to, for example, having a fixed mesh network).
8. *System fault tolerance*: If a single PE fails, only those submachines that include the failed PE are affected.
9. *Submachine fault tolerance*: If a PE in a submachine fails, it may be possible to redistribute data and make use of mixed-mode parallelism (i.e. changing from SIMD to MIMD mode) and the variable connectivity (i.e. to establish connection patterns that do not include the faulty PE) so that the job executing on the submachine may continue on that submachine with nominal degradation.



Examples of how the PASM concept can be used to vary machine size for utilization and speed, exploit subtask parallelism, achieve improved performance through mixed-mode parallelism and establish different types of inter-PE connection patterns are given in Siegel, Armstrong and Watson (1992).

## 3.3 PASM prototype implementation

### 3.3.1 Overview

This section describes the control hierarchy hardware of the small-scale, proof-of-concept PASM prototype built at Purdue University. Figure 3.8 shows a block diagram of the PASM prototype with $N = 16$ and $Q = 4$. The processors are based on the Motorola MC68000 family microprocessor. Because only commodity components were used, the need for custom VLSI chips was eliminated and development time and construction costs were reduced. A total of 28 different types of physical board are used in the prototype, 13 of which

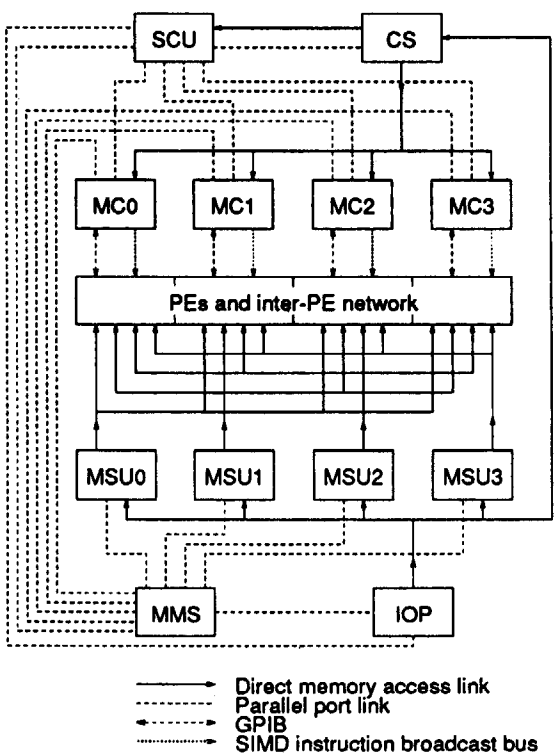


**Figure 3.8** *The PASM parallel processing system prototype.*

**Table 3.1** *The PASM design parameters for the existing prototype and a possible future large-scale machine*

| *Feature* | *General* | *Large-scale PASM* | *PASM prototype* |
|---|---|---|---|
| Number of PEs | $N$ | 1024 | 16 |
| Number of network stages (Extra Stage Cube, if 2 × 2 boxes) | $\log_2 N + 1$ | 11 | 5 |
| Number of MCs | $Q$ | 32 | 4 |
| Number of PEs per MC | $N/Q$ | 32 | 4 |
| Number of MSUs | $N/Q$ | 32 | 4 |
| Number of Memory Management System processors | Varies | ≥4 | 4 |
| Smallest size partition | $N/Q$ | 32 | 4 |
| Maximum number of partitions | $Q$ | 32 | 4 |

are commercial products. The System Control Unit (**SCU**), MCs, PEs, Memory Management System (**MMS**) processors, I/O Processor (**IOP**), Control Storage (**CS**) and MSU processors, and the PE interconnection network are all implemented by using these boards in different configurations. Table 3.1 summarizes the PASM design parameters for the existing prototype and a possible future large-scale machine.

The System Control Unit is an MC68010 microprocessor running UNIX. It has an Ethernet connection to the Purdue Engineering Computer Network, a local area network connected to the Internet. A PASM user logs into the System Control Unit and controls the prototype by issuing commands from the System Control Unit to initialize the system and to load and execute programs. The System Control Unit communicates with secondary storage and MCs through dedicated handshake-driven parallel port connections.

Mass storage for the MCs and PEs is to be provided by the Control Storage and the MSUs, respectively. The Control Storage and the four MSUs each contain a 40 Mbyte disk drive. The MSUs are to be managed and controlled by the Memory Management System, consisting of a directory look-up processor, a memory scheduling processor, a command distribution processor and an I/O Processor (IOP). The I/O Processor is to be responsible for handling interactions with external I/O devices and for distributing data to the MSUs and the Control Storage through highly efficient DMA links with their memories.

The operating system software support for the Memory Management System, MSUs and Control Storage is still under development. Currently, the PE and MC memories are loaded from a 40 Mbyte hard disk attached to the System Control Unit.

Each prototype MC is based on an MVME-110 single-board MC68000 microcomputer on a private VME bus backplane with 2 Mbytes of dynamic RAM, physically double-buffered as described in section 3.2.3. An MC is augmented with additional logic to support necessary interfaces to the System Control Unit, Memory Management System, other MCs and its associated



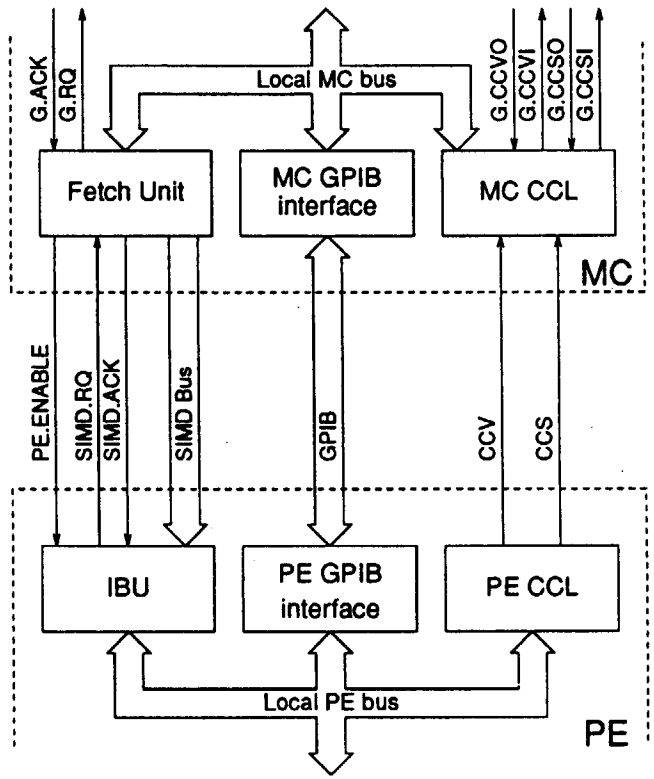


**Figure 3.9** *Communication and control connections between PEs and MCs. Of the four PEs connected to each MC, only one is shown. The SIMD Bus, the SIMD.ACK line and the GPIB are shared among all PEs. All other lines connect MC and PE on a one-to-one basis.*

PEs. The System Control Unit, Memory Management System and remote MC connections are implemented with simple parallel ports used for passing control information. The remaining special-purpose connections from the MCs to the PEs support mixed-mode operation and are overviewed here. These connections include the mechanism to perform SIMD instruction broadcasts, the logic to allow the MC to access PE conditional results and a channel for the PEs to send data to their MCs.

Like an MC, each PE consists of an MVME-110 MC68000 microcomputer connected via a VME bus backplane to 2 Mbytes of dynamic RAM and logic to interface the PE with its MC and the prototype interconnection network. The dynamic RAM is physically double-buffered, as described in section 3.2.4, with two 1 Mbyte memory modules. The memory modules themselves are dual-ported with one port connected to the PE VME bus and the other port

connected to the appropriate MSU with a high-speed DMA link. PEs access the interconnection network through a set of parallel ports mapped in the I/O space of each PE's address space.

There are three channels for communication and control between each MC and its associated PEs. Figure 3.9 shows these channels and their interfaces to both the PEs and an MC. When the MC group is in SIMD mode, the instruction broadcast bus (**SIMD Bus**) carries SIMD instructions from the MC along with the PE enable signals to the PEs. Enabled PEs receive these instructions and execute them in lock-step. A General Purpose Interface Bus (**GPIB**) link provides bidirectional communication between an MC and its PEs. The condition code channel (CCV and CCS) is used to send the result of the evaluation of a data condition from the PEs to their respective MC. The function and implementation of these channels are presented in section 3.3.2.

### *3.3.2 SIMD instruction broadcasting*

An MC in SIMD mode broadcasts an instruction stream to its PEs, which execute it in lock-step at the instruction level. Because commodity microprocessors are being used, the instruction words broadcast to the PEs are MC68000 machine instructions, not machine control words, as in the MPP, the Illiac IV and the CM-2, for example. When a PE CPU requests an SIMD instruction, the PE's Instruction Broadcast Unit (**IBU**) forwards an SIMD instruction request to the MC by asserting **SIMD.RQ**. The MC receives the SIMD instruction requests from the PEs through its Fetch Unit.

Once all PEs in an MC group have requested an instruction, the Fetch Unit asserts its group request line **G.RQ**, which is sent to the System Control Unit. When all MCs in a submachine have asserted their G.RQ lines, the System Control Unit triggers the broadcast of the next SIMD instruction by asserting the group acknowledge line **G.ACK** (this is discussed further in section 3.3.5). Upon the assertion of G.ACK, all Fetch Units of the submachine broadcast the next SIMD instruction word by placing the instruction word on the SIMD Bus, setting the PE enable bits, and asserting the SIMD instruction acknowledge line **SIMD.ACK**. The PE enable bits determine which PEs are to execute that particular instruction. The IBUs of the PEs receive the instruction and allow the enabled PEs to execute the instruction. If a PE is denied an SIMD instruction, that PE is said to be disabled for that instruction and does not execute it.

*Fetch Unit*

In SIMD mode, the MCs send instructions to their associated PEs by means of a **Fetch Unit**. In Figure 3.10, the basic components of an MC, with emphasis on the Fetch Unit, are illustrated. The MC CPU executes control instructions (e.g. loop counting and branch control), which it reads from the MC memory. Whenever the MC broadcasts an SIMD instruction to its PEs, it must first set the PE enable bits so that only the appropriate PEs execute the instruction. After setting the **Mask Register**, the MC instructs the **Fetch Unit Controller** to move the SIMD instruction from the **Fetch Unit Memory** into the SIMD instruction **FIFO**. Other machines, e.g. MPP and CM-2, also use a FIFO to



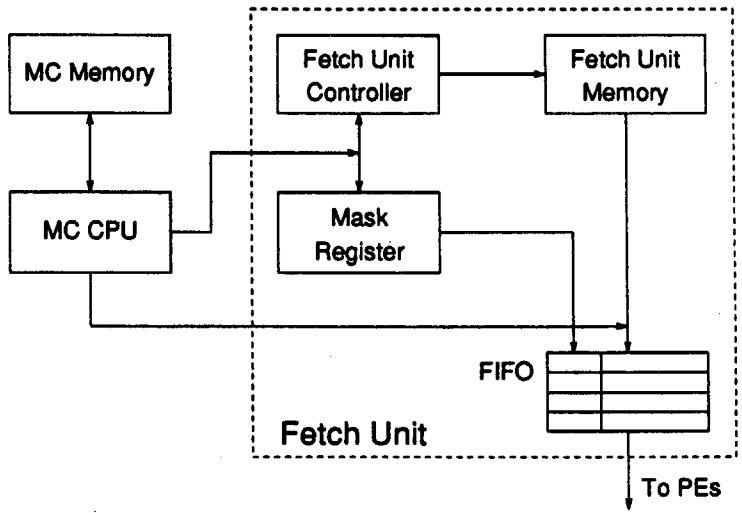


**Figure 3.10** *Simplified block diagram of an MC with emphasis on the Fetch Unit structure.*

decouple control unit execution from PE execution. This permits **control unit/PE overlap**, the concurrent execution of control flow and global common data computations on the MC with PE-local data computation on the PEs (Kim, Nichols and Siegel, 1991). However, the Fetch Unit FIFO also appears to the MC CPU as a memory-mapped I/O location, which permits the MC CPU to write directly to the FIFO. This capability is shown by the line connecting the MC CPU and FIFO in Figure 3.10. The FIFO contains 64 16-bit words.

Because the SIMD instructions are MC68000 machine instructions, each can consist of several instruction words. Frequently, several SIMD instructions are executed with the same set of enable bits. Therefore, the Mask Register in Figure 3.10 needs to be set only when the enable bits are changed. Every time an instruction word is enqueued, the enable bits currently in the Mask Register are enqueued as well. The Fetch Unit Controller can be instructed to move a block of instructions to the FIFO, thereby reducing the number of control instructions required to broadcast the SIMD instruction stream. Clearly, the enable bits for all instruction words in the block must be identical. When all PEs of a submachine have requested an instruction word, and one is available in the FIFO, an instruction word and the associated enable bits are dequeued and broadcast to the PEs. Each PE receives its own enable bit and a copy of the instruction. This way, PEs, Fetch Unit and MC can proceed concurrently.

The operation of the Fetch Unit can be described in detail by discussing the actions required to broadcast SIMD instructions to the PEs. Figure 3.11 shows a block diagram of the Fetch Unit. Before the SIMD program starts,

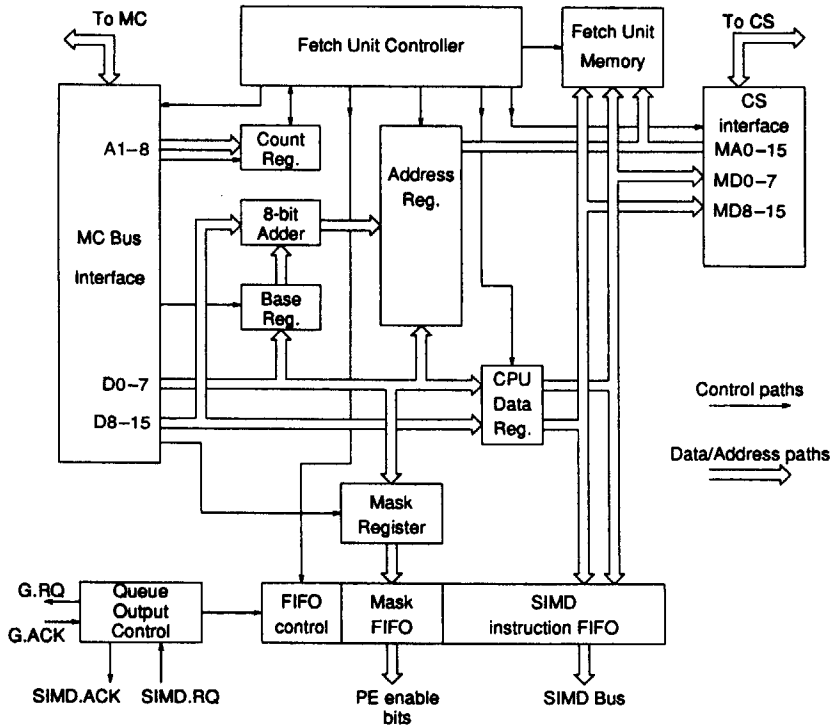


**Figure 3.11** *Block diagram of the PASM prototype Fetch Unit.*

the Control Storage loads the SIMD instructions into the Fetch Unit Memory. The Fetch Unit Memory has 32 Kbytes of storage. All SIMD programs are assembled as if they are always loaded into the Fetch Unit Memory starting at location 0, and the control instructions sent to the Fetch Unit assume this loading. Because loading at starting location 0 is not always desirable (e.g. a new program might be loaded while the old one is still running to support the double-buffering discussed in section 3.2.3), a **Base Register** is provided that permits relocation of the SIMD instructions in the Fetch Unit Memory.

Assume that the instructions of an SIMD program are loaded into the Fetch Unit Memory beginning at memory location $B$, where $B$ is a binary number with the lower eight bit positions zero. The remapping is accomplished by the Control Storage, which can directly access the Fetch Unit Memory. The Fetch Unit Controller arbitrates conflicts between accesses from the Control Storage and internal accesses. Before the MC begins execution of the SIMD control program, it sets the Base Register to the value $B$ via the **MC Bus Interface**.

Now assume that during the execution of the SIMD program a block of SIMD instructions is to be executed by the same set of PEs (designated by



the enable bits $E$), and that the address of the first instruction in the block would be $A$ if the instructions had been loaded starting at address 0. Because $B$ is the actual load starting address, the memory address of the first instruction word in the block is $A+B$. The following sequence of six events is then performed.

1. The MC CPU writes the enable bits $E$ to the Mask Register. Each MC in the prototype has four PEs associated with it, therefore, the Mask Register is four bits wide.
2. The MC CPU specifies the starting address of the instruction block to the Fetch Unit Controller and the number of instructions to be broadcast. This information can be encoded by a single write instruction. The upper 16 bits of the MC address bus are decoded by the MC Bus Interface to determine whether an instruction block is to be enqueued. Then, under hardware control, the lower eight bits of the MC data bus (D0–7) are written to the lower eight bits of the **Address Register**, the higher eight bits (D8–15) are written to the 8-bit **Adder** and the MC address bits A1–8 are written to the **Count Register**.
3. The 8-bit Adder calculates the sum of the Base Register value and the value on the higher eight data bus lines, and strobes the result into the higher eight bits of the Address Register. The Address Register now contains the value $A+B$, i.e. the physical start address of the SIMD instruction word block.
4. The Fetch Unit Controller detects a non-zero value in the Count Register. It first disables any further MC accesses to the Fetch Unit, thereby ensuring that the movement of the instruction word block is not disrupted. The controller then starts to move instructions from the Fetch Unit Memory into the SIMD instruction FIFO. The Fetch Unit Controller monitors the FIFO and waits until there is room in the FIFO. When the FIFO is not full, it places the value of the Address Register on the memory address bus (MA0–15), reads the value of the memory location and strobes the value into the SIMD instruction FIFO. Simultaneously, it enqueues the contents of the Mask Register into the Mask FIFO. It then increments the Address Register and decrements the Count Register. The controller repeats this fetch–enqueue cycle until the last item has been enqueued, indicated by a zero value in the Count Register. After the last item has been enqueued, the Fetch Unit Controller re-enables MC accesses.
5. Independent from the enqueuing process, the PEs dequeue instructions. When all PEs in the MC group have requested an instruction by asserting their SIMD.RQ lines, the **Queue Output Control** forwards a group request signal G.RQ to the System Control Unit where it is combined with the G.RQ signals of the other MCs in the current submachine. As soon as the Queue Output Control receives the combined acknowledge G.ACK, it dequeues an SIMD instruction and the associated enable bits, and asserts the SIMD acknowledge line SIMD.ACK. When the PEs detect the asserted SIMD.ACK signal, they read and (if enabled) execute the instruction.

6. There is also the capability for the MC to enqueue instructions directly word by word.

The capability of PASM to switch dynamically between SIMD and MIMD modes merits special attention. Assume that the PEs are executing an SIMD program, then switch to MIMD mode, and later return to SIMD mode. The MC's actions consist of broadcasting data-processing SIMD instructions, then sending a jump instruction that causes the PEs to start execution of the MIMD program, and finally resuming the broadcast of SIMD instructions.

*Instruction Broadcast Unit*

Normally, a PE CPU fetches instructions by placing its program counter value on the address bus and latching the data placed on the data bus by the memory device holding that instruction word. However, SIMD instructions are not physically located in the memory of the PEs.

A PE fetches SIMD instructions by reading an instruction word from the **SIMD instruction space** of the PE's memory. This is a logical address space – no physical memory is provided. Figure 3.12 illustrates the address space of a PE, showing the SIMD instruction space, physical RAM and I/O space. Logic in the PE detects read accesses to the SIMD instruction space, and any such access is interpreted as an SIMD instruction request. In the PASM prototype, the SIMD instruction space is determined by the values of the two highest order bits of the 24-bit program counter. The remaining 22 bits of the program counter value placed on the address bus during SIMD instruction fetches are ignored. Because the program counter is automatically incremented after each instruction fetch, it is possible for the program counter value to increment out of the SIMD instruction space during a very long SIMD program. This problem is eliminated by allowing the MC CPU to write occasionally

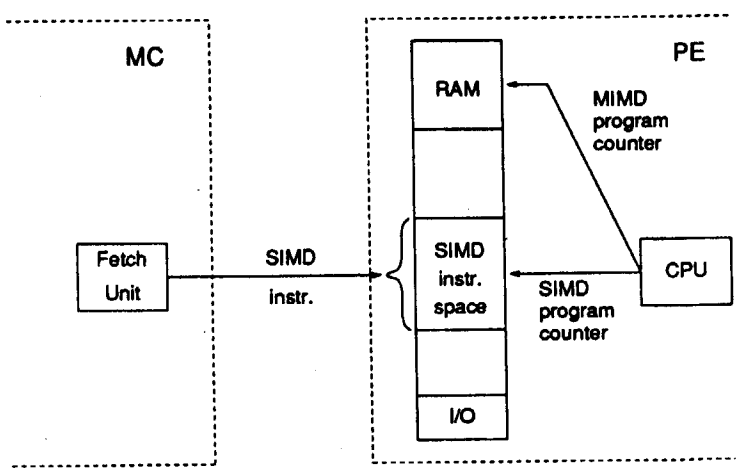


**Figure 3.12** *Logical PE address space in the PASM prototype.*



into the SIMD instruction FIFO an instruction forcing the PEs to jump to the beginning of the SIMD instruction space. Because the SIMD instruction space is large (4 Mbytes), the frequency with which this is required is very small.

Each memory access made by the PE CPU is monitored by the IBU. When its PE CPU accesses the SIMD instruction space, the IBU forwards an SIMD instruction request (asserts SIMD.RQ) to the Fetch Unit of its MC, which supplies the instruction to all PEs.

When an SIMD instruction word is broadcast by the assertion of SIMD.ACK, the value of a particular PE's enable bit **PE.ENABLE** determines whether that PE is to execute that instruction. If the PE enable bit is set for a particular PE when the asserted SIMD.ACK is received by that PE's IBU, that IBU places the instruction word on its local bus. It then forwards an acknowledge signal to the PE CPU, which latches and executes the instruction. Otherwise, the IBU does not forward an acknowledge signal to its PE CPU. In this case, the instruction word is not latched and executed by that PE. Because the PE CPU did not receive the broadcast acknowledge signal, its SIMD instruction request remains pending until its IBU receives an instruction for which its PE enable bit is set.

Because the MC places SIMD instructions in a queue, they are, in a sense, prefetched for the PE processor. This means, in general, that no RAM access penalty is paid for SIMD instruction accesses. This allows SIMD instruction accesses in the prototype to be one processor cycle (125 ns) faster than PE RAM accesses. There is, however, a penalty paid for SIMD instruction accesses when the SIMD instruction broadcast is synchronized across MC groups. It will be seen in section 3.3.5 that with this added penalty SIMD instruction access is still less than a PE RAM access time. Additionally, PE processors prefetch instruction words wherever possible and thus sometimes hide the latency of instruction fetches in either SIMD or MIMD modes.

Another advantage of this instruction broadcast scheme is the ease of switching between MIMD and SIMD modes. A PE is forced into SIMD mode by executing a jump to the SIMD instruction space. A PE can switch from SIMD mode to an MIMD program located at some address *A* by receiving a 'jump to subroutine at *A*' instruction in SIMD mode. When the MIMD subroutine is completed, executing a 'return from subroutine' instruction restores the program counter to the SIMD instruction space. Then the PE CPU requests an SIMD instruction. However, the SIMD instruction is issued only after all the PEs of a submachine have requested the instruction. Thus, all the PEs of the submachine are synchronized when an SIMD instruction is broadcast. Such flexibility in mode switching allows mixed-mode programs to be written that change modes at instruction-level granularity with nominal overhead. The SIMD instruction fetch mechanism can also be used to support a form of MIMD barrier synchronization (Dietz *et al.*, 1989).

### *3.3.3 MC–PE communications*

The GPIB connects each MC to its associated PEs. The GPIB (IEEE 488 bus) has a data path width of eight bits. It was chosen because of its ease of implementation. Only five chips are required for each MC GPIB interface,

and four for each PE GPIB interface. In SIMD mode, the GPIB is used to transmit data from a PE to an MC and if a PE needs to alert an MC to an error condition, e.g. a divide-by-zero or an error in network transmission. In MIMD mode, the MCs use the GPIB to coordinate the activities of the PEs.

The MC's interface configures the MC as a talker/listener and controller of the GPIB. Each PE is connected as a GPIB talker/listener. A **GPIB talker/listener** can send or receive data from other devices on the GPIB. The **GPIB controller** coordinates all GPIB activities (e.g. arbitrates access to the GPIB).

Transfers are initiated when a talker/listener receives bus ownership from the controller and addresses another talker/listener. The CPU originating the transfer then writes the message data to its GPIB interface, which sends the data to the GPIB. The addressed GPIB interface interrupts its CPU, which reads the incoming data from its own GPIB interface. Once the source device has finished sending the message, the GPIB is relinquished.

### *3.3.4 Conditional operations*

In SIMD mode, the MCs can perform data-conditional operations that depend on PE results (discussed in section 3.2.3). To support this, Condition Code Logic (**CCL**) is provided that enables MCs to request conditional information from the PEs. The System Control Unit CCL can be used to combine data conditions among MCs. Figure 3.9 shows the connections between PE CCL and MC CCL, and also shows connections to the System Control Unit CCL, which is discussed in detail in section 3.3.5. The function of the System Control Unit CCL is similar to the SUM-OR tree in the MPP and the GLOBAL line in the CM-2, which are used to find global results.

Consider how an MC receives conditional information from its PEs. The MC first broadcasts instructions to the PEs specifying which conditional test is to be performed (e.g. equal or negative), which causes the PEs to set appropriate hardware registers in the PE CCL. The MC then instructs the PEs to write a one-bit conditional result to the Condition Code Value (**CCV**) line. The PEs perform all SIMD instructions simultaneously. However, due to the SIMD instruction FIFO delay, the MC does not know when the PEs have actually written their conditional result to the CCV line. For this reason, a Condition Code Synchronization (**CCS**) signal is provided. After writing the condition code, each PE sets its CCS line, and the MC CCL combines these signals. The MC CPU monitors the combined signal, and reads the individual PE condition codes as soon as it detects that all PEs have set the CCS line. The MC can now use the conditions it has read from the PEs. As an example, consider a division operation. Before the division is executed, the divisor should be tested for a zero value. After the appropriate conditional test is performed, the MC can use the conditional result as PE enable bits, and thus enable only those PEs that do not have a zero divisor, thereby avoiding divide-by-zero faults.

To improve functionality, the hybrid masking scheme (described in section 3.2.3) may be incorporated into the prototype hardware in the future.



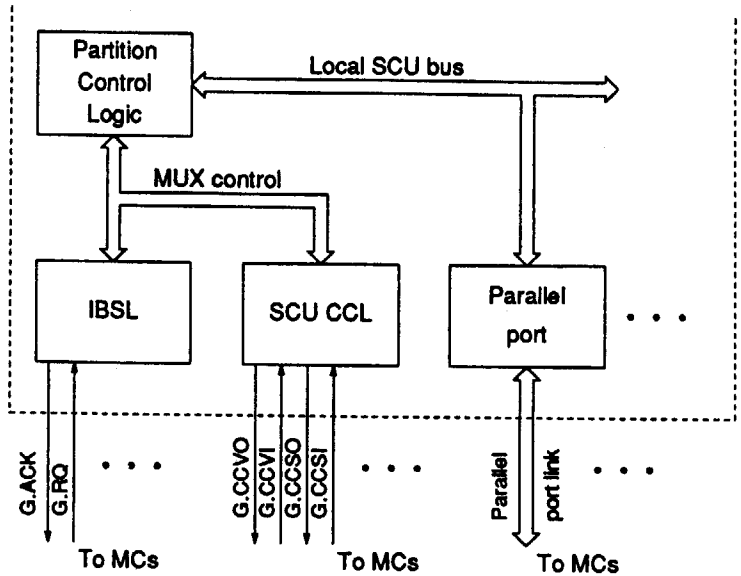


**Figure 3.13** *Communications and control connections between System Control Unit and MCs. Of the four MCs connected to the System Control Unit, only one is shown. All links connect System Control Unit and MC on a one-to-one basis.*

Currently, the MC combines the information from PE-address masks and data-conditional masks in software to load the Mask Register. The decision to have the PEs report their condition codes to the MC rather than maintaining their own local enable bit stack (as described in section 3.2.3) was made to simplify the prototype hardware.

### *3.3.5 MCs and System Control Unit interactions*

Interactions between the System Control Unit and MCs are required for data transfer and partition control. For data transfers, the System Control Unit contains four separate parallel port links, one for each MC. Figure 3.13 shows one of the four links. MCs as well as the System Control Unit can initiate data transfers. The link carries control information and interactive I/O data. Examples of control information are task control information (e.g. task scheduling and termination), error messages originating from the MCs or PEs, and MC file requests that the System Control Unit forwards to the Control Storage. During interactive programs, data must be passed between the user and the user's program executing on the PEs. Because the user session is handled by the System Control Unit, the System Control Unit must forward user input to the appropriate MCs (which in turn forward the data to the PEs), and must accept data from the MCs (which originated from the PEs).

Because the System Control Unit is responsible for allocating partitions to user programs (among other duties), it must also control the machine partitioning. Recall that the partitioning rule in PASM requires that PEs in a submachine agree in their low-order bit positions (section 3.2.3). Therefore, in the prototype, the only submachines that are possible are the MC groups working individually, MC groups 0 and 2 combined, MC groups 1 and 3 combined, and all four MC groups combined. In SIMD mode, the following hardware is required to combine multiple MC groups into a single submachine: the Instruction Broadcast Synchronization Logic (**IBSL**), the System Control Unit Condition Code Logic (**SCU CCL**), and the Partition Control Logic, which controls both the IBSL and SCU CCL through an **MUX Control** bus. Figure 3.13 shows these components and one of the parallel ports.

When multiple MC groups are combined into a submachine, and the PEs run an SIMD program, all PEs in the submachine must execute their instructions in lock-step. This is accomplished in the following manner. The PEs of an MC group request an SIMD instruction from the Fetch Unit by asserting their SIMD.RQ signal (Figure 3.9). When all PEs have requested an instruction, the Fetch Unit forwards a group request signal G.RQ to the IBSL, and the IBSL combines all G.RQ signals of a submachine. When all MC groups of a submachine have asserted G.RQ, the IBSL sends a group acknowlege G.ACK to all Fetch Units in the submachine. Only then will the Fetch Units assert their SIMD.ACK, and all PEs in the submachine receive the instruction simultaneously. Figure 3.14 shows the logic used in the IBSL to combine the G.RQ signals. Due to the restrictions placed on which MCs may be combined, an AND-tree with three 2-input gates is sufficient to generate all required combined signals. Depending on the current partitioning, the System Control Unit sets four 3-input data selectors (3:1 MUXs) via the Partition Control Logic to select the appropriate combining signal. For example, if all four MCs are combined into a single submachine, the leftmost input of the

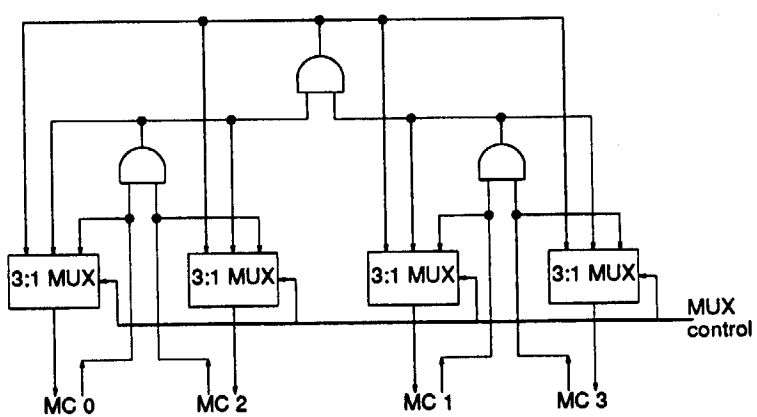


**Figure 3.14** *Combination logic for SIMD instruction broadcast synchronization and condition code combination.*



data selectors will be routed to the output. The IBSL delays the SIMD instruction broadcast by no more than 25 ns. This is small when compared with the MC68000 memory cycle of 400 ns (SIMD instruction broadcast delays were discussed in section 3.3.2).

As described in section 3.3.4, the MC CCL enables each MC to receive conditional information from its PEs. Because multiple MC groups can be combined to form a single submachine, that logic alone is not sufficient because it may be necessary to know `if-any` or `if-all` of the PEs meet a certain condition, as was discussed in section 3.2.3. For this reason, the SCU CCL is provided. Figures 3.9 and 3.13 show the connections between SCU CCL and MC CCL. To find a global condition value, the MCs first find a combined condition for all their PEs, and then write this value to the group condition code value input (**G.CCVI**). Because the MCs do this asynchronously, each MC asserts the group condition code synchronization input (**G.CCSI**), and monitors the group condition code synchronization output (**G.CCSO**), which is asserted only after all MCs in the submachine have asserted the G.CCSI. When an MC detects an asserted G.CCSO, it reads the combined condition value from the group condition code value output (**G.CCVO**).

This logic, shown in Figure 3.14, is also used to combine the group condition code signals. The AND-tree in the combination of condition code values can be used to detect `if-any` as well as `if-all` conditions. When the `if-any` condition is desired, the MCs write a logical 0 to their G.CCVI line if the condition holds for one or more of their PEs. The output of the AND-tree is a logical 0 if one or more inputs are 0. A logical 0 on the G.CCVO line therefore means that the `if-any` condition is true. Similarly, an `if-all` condition can be detected by writing a logical 1 to the G.CCVI line if the condition is met, and only if every input to the AND-tree shows a logical 1 will the result be a logical 1.

### *3.3.6 Inter-PE communication*

The PASM prototype interconnection network is an Extra Stage Cube, as mentioned in section 3.2.4. To simplify the hardware, circuit switching (as opposed to packet switching) is used. To transfer data between a network source–destination pair, a physical path must first be made connecting the pair. The source PE establishes the path by writing the appropriate routing tag to its memory-mapped network interface logic and is informed when the virtual circuit through the network is complete. Once the path is established, data transfers can proceed. The source PE writes the individual words of the message to the network interface (parallel ports), which automatically transfers the data to the network interface of the destination PE. The destination PE reads the message word by word. At the conclusion of transmission, the source PE relinquishes the path by writing a control word to its network interface.

The PASM prototype network is constructed from commodity TTL SSI/MSI components and implements a 16-bit wide data path plus two parity bits. The use of standard TTL components in the prototype network yields an inexpensive network that does not limit throughput on established paths. Each

PE CPU communicates with the network through parallel ports. Due to the implementation of the network, the operating speed of the PE CPU and the parallel ports' communication rates are the limiting factors in determining how fast the network can transfer data. Hardware in the network interface provides correct destination verification on each path established and performs parity checking on all transmitted data.

Assuming no conflicts, the time required to establish a path is approximately 2 μs. Once a path is established, the network hardware should support a transfer rate of 24 Mbits/s, but is limited to a sustained transfer rate of 3.8 Mbits/s due to the processing rate of the PE CPU. The goal of the prototype construction was to implement a tool for studying the attributes of such a reconfigurable architecture, not to maximize the raw speed of the prototype hardware. The prototype network supports all of the functionality discussed in section 3.2.4.

### *3.3.7 Advantages of the prototype*

As discussed in section 3.4, the PASM prototype is supporting experimentation with the three dimensions of reconfigurability listed in section 3.1: partitionability, mode of parallelism and connections among the PEs. Also mentioned in section 3.4 is that the prototype is being used in a parallel programming course at Purdue. The goal for the prototype effort was to build a robust, small-scale, proof-of-concept machine for the PASM design concepts. This goal was achieved, and the prototype is supporting both research and education.

## 3.4 Using the PASM prototype

### *3.4.1 Overview*

As mentioned previously, the primary goal of the PASM prototype is to support experimentation with reconfigurable parallel processing systems. Because the PASM prototype supports SIMD, MIMD and mixed-mode parallelism, accurate comparisons of different modes of parallelism can be made.

In addition to the research being conducted with the prototype, a graduate-level course on programming parallel machines using the prototype is currently being offered at the Purdue University School of Electrical Engineering. Students have the opportunity to learn about basic concepts of parallel programming. They have access to three classes of parallel processing system: SIMD machines, MIMD machines and mixed-mode machines, represented respectively by the MasPar MP-1, nCUBE-2 and the PASM prototype, all part of the School of Electrical and Computer Engineering Parallel Processing Laboratory. By programming SIMD, MIMD and mixed-mode parallel processing systems, students learn the characteristics of each mode of parallelism and the trade-offs among different modes. Moreover, they test the ease of use, interfaces and various languages of the parallel processing systems.



The focus of this chapter is the conceptual organization of PASM and a description of the prototype control hierarchy hardware. To show how the prototype is being used, sections 3.4.2–3.4.4 briefly overview some of the software and applications that have been implemented on the PASM prototype (a detailed presentation of these topics is beyond the scope of this chapter).

Section 3.4.2 discusses CAPS, a system designed to aid in software development for the PASM prototype. ELP, a parallel language with modes of parallelism explicitly specified by the user, is described in section 3.4.3. Section 3.4.4 lists examples of applications research conducted with the PASM prototype.

### *3.4.2 CAPS*

The Coding Aid for the PASM System (**CAPS**) is designed to assist in the development and evaluation of application and system software for the PASM prototype (Lumpp *et al.*, 1991). A primary goal in the design of CAPS was to provide users with the ability to execute and monitor their programs with maximum flexibility and minimum intrusion given the existing system hardware, while providing information on a wide range of program attributes. CAPS integrates hardware support and software tools to provide a remote execution and program debugging/monitoring environment for the PASM prototype. The prototype can be accessed over the Internet through CAPS.

CAPS consists of a set of dedicated I/O channels and associated hardware and software that facilitate bidirectional information flow between the individual processors of PASM and a workstation providing the user interface. The information is specified by source-level statements, added to the user's program, which transmit messages through dedicated I/O channels. Once the messages are sent from the processors, they are combined into a single stream and sent through a Local Area Network (**LAN**) to a workstation where they are used to debug and analyze the execution of the program.

CAPS allows the programmer to use some sophisticated features of a graphics workstation to process I/O coming from the parallel system. The workstation's windowing capability allows textual debugging information to be displayed from any PASM processor at an interactive virtual terminal, represented by a window. Multiple windows can be displayed to monitor different processors simultaneously. One possible future extension is to generate graphic displays summarizing collected data.

Thus, CAPS is a simple but useful programming tool for the PASM prototype. It provides remote access to PASM for multiple users, and integrates system features such as downloading code, code development, interactive I/O and run-time monitoring of programs with sophisticated workstation windowing capabilities.

### *3.4.3 ELP*

The Explicit Language for Parallelism (**ELP**) is a language designed for programming mixed-mode parallel machines (Nichols, Siegel and Dietz, 1993). The user is able to indicate explicitly the parallelism to be employed. ELP

provides constructs for both SIMD and MIMD parallelism and an ELP application program can perform computations that use these parallelism modes in an interleaved fashion for mixed-mode operation.

The syntax of ELP is based on C, extended with parallel constructs and specifiers. The SIMD and SPMD modes of parallelism are supported by a full native-code compiler, targeted to PASM, that permits these modes to be switched dynamically at instruction-level granularity. ELP is being extended to include full MIMD capability. The goal is for ELP to be uniform with respect to the SIMD and SPMD modes of parallelism, where all of the language's constructs, operations, statements, etc. have interpretations within both of these modes that are identical in both syntax and semantics.

In ELP, each variable has a variable class associated with it. A variable defined to be of class `mono` always has a single value with respect to all PEs, independent of the execution mode. A variable defined to be of class `poly` can have one or more values in different PEs, independent of the execution mode. Each `mono` variable has storage allocated for it on the MC and all PEs. If a `mono` variable is referenced while in SIMD mode, its MC storage is active. This permits the specification of the SIMD control unit/PE overlap discussed in section 3.3.2. If a `mono` variable is referenced while in SPMD mode, its PE storage is active and all PE copies of the `mono` variable will have the same value spatially (guaranteed by the compiler). For variables defined to be `poly`, each PE has its own copy with its own value. Because the PASM machine is partitionable into submachines, a `poly` variable declaration invokes a variable on each PE within the submachine upon which an ELP program is to execute. Currently, this submachine determination is made by the user at load-time.

An ELP program can use both SIMD mode and SPMD mode and may switch between the two modes one or more times at instruction-level granularity. SIMD/SPMD execution mode specification is statically scoped and uses the keywords `simd` and `spmd`. Specification can be done on a per-block basis or on a per-function basis. Execution mode specifiers can be nested and are tracked with a compile-time stack. All PEs in a submachine must be in the same mode at a given point in time and all function declarations must include an execution mode specifier.

ELP supports both data-dependent and PE-address dependent control flows. The data-dependent control-flow constructs, such as `if`s, `while`s, `do`s and `for`s are given different parallel interpretations depending on the variable class of the conditional expression. The selector statement provides PE-address-dependent control flow. It specifies which PEs will be selected for the scope of the following block of one or more statements and is independent of the execution mode in which it is called. Selector statements are based on the PE-address masks defined in section 3.2.2. As with data-dependent masks, PE-address mask scopes can be nested and a PE is selected only if it is selected by all PE-address masks in the nesting. When both data-dependent masks and PE-address dependent masks are used, a PE is active only if it is selected by both masking schemes.

ELP provides users with the ability to explicitly control different facets of mixed-mode parallel processing systems and simplifies mixed-mode programming by employing a single program model for both modes of parallelism.



Furthermore, it serves as a foundation for the design of a compiler that automatically determines and specifies the best mode for code segments, and facilitates the use of reconfiguration for fault tolerance. In general, an explicitly parallel language, such as ELP, provides a vehicle for the exploration of and experimentation with mixed-mode parallelism.

### *3.4.4 Algorithm studies*

Algorithm research activities are exploring ways to exploit the flexibility of a reconfigurable parallel processing system. The flexibility of such a system makes the efficient execution of a wide range of applications possible. However, the software challenges for partitionable mixed-mode systems are a superset of those for SIMD and MIMD single-mode systems. Consequently, PASM-related algorithm studies are an active area of research and have included theoretical analyses, simulations and experiments on the PASM prototype (Siegel, Armstrong and Watson, 1992). These studies have examined issues such as mapping tasks onto reconfigurable parallel processing systems, trade-offs among the SIMD, MIMD and mixed-mode classes of parallelism, MC/PE computational overlap in SIMD mode, impact of increasing the number of PEs used and partitioning for improved performance. Applications considered include bitonic sorting, edge-guided thresholding, FFTs, global histograming, image correlation, image smoothing, matrix multiplication and range-image segmentation (references are given in Armstrong, Watson and Siegel (1993)).

## 3.5 Experiences with PASM: lessons learned

The PASM prototype became operational in 1987 and is still running in 1995, with over 36 000 hours of execution time logged. A photograph of the prototype is in Bell (1992, p. 73). The system was built with a very limited personnel budget and approximately $150,000 for equipment. This had two major consequences: extensive use of student projects and a need to make conservative design decisions.

Student projects, typically lasting one academic semester, have the advantage of providing highly-motivated personnel at low cost. To use student projects effectively, careful supervision and planning were needed to get the right subsystems running at the right point in time. The most serious drawback to this approach was the potential for failure at each stage. Delays due to design flaws in the completion of one subsystem can have a dramatic impact on the work for successive projects. Fortunately, the effect of this type of delay was minimal during the PASM prototype construction.

A conservative design decision policy was adopted, which implied modifications to the original plans if necessary, the use of mature technology, and the exploitation of commodity components and subsystems wherever feasible. Throughout the project, the primary goal was to build a robust system on which the architectural concepts of PASM could be studied and on which applications could be programmed.

The original system hardware concept was modified in only the very few areas where cost and the difficulty of hardware implementation were not offset by the benefits for a prototype system. As one example of a system modification, a reconfigurable bus connecting the MCs (Siegel *et al.*, 1981) was not implemented because of the costly high-speed components and connections that would have been needed.

There was never an attempt to push the state-of-the-art in terms of computational power, because budget limitations certainly would have made such an attempt a failure and might have jeopardized the overall project as well. Conservative design meant the exclusive use of commodity components, particularly the microprocessors in the PEs. The concept of employing standardized microprocessors was prevalent among commercial parallel systems when PASM was being designed, although these machines did not support the SIMD mode of computation (e.g. Intel iPSC, BBN Butterfly). The validity of this approach is still borne out today because even more commercial parallel systems are based on commodity microprocessors (e.g. Cray T3D, Convex SPP-1 (Astfalk, 1992), Thinking Machines CM-5, IBM SP1 (IBM Corporation, 1993)).

There are no custom integrated circuits in the system, and clock frequencies are comparatively low. Because only 16 MC68000 CPUs were to make up the Parallel Computation Unit, performance was not an overriding issue. Thus, design decisions were always made such that timing was on the safe side, even if that caused sacrifices in performance.

Supporting SIMD mode using the MC68000 CPUs was facilitated by the fact that the MC68000 CPUs had no on-chip caches and their instruction cycle time (typically four processor cycles) was approximately equivalent to the total memory access time of a VME bus memory card (includes bus cycles and DRAM accesses). Therefore it was possible to build a system where the performance was processor-bound, i.e. providing instruction streams to the processor in either SIMD or MIMD mode could be accomplished such that the processor cycle time was still the performance bottleneck.

However, today's microprocessors have on-chip instruction caches and cycle times that are nearly an order of magnitude faster than DRAM chips. New challenges exist in supporting SIMD mode with current commodity microprocessors. The PASM prototype relied on the MC68000 explicitly requesting every instruction on the microprocessor's external bus. Using this technique with today's microprocessors would yield a very inefficient system. Discovering new ways to execute identical instruction streams on multiple processors in (apparent) lock-step represents a new challenge in this area.

Careful consideration was given to each component in the prototype to decide whether to use commodity subsystems or to construct custom solutions. As a result, CPU boards, disk controllers and VME bus backplanes were all commercial designs, while the Fetch Units, network boards and memory boards were custom-designed and fabricated specifically for the project. The use of selected commercial components proved invaluable to the progress of the project because it eliminated lengthy debug and test cycles for crucial functions. For the specialized subsystems such as the Fetch Units and the interconnection network, no alternative to special purpose design existed. Here, the use of conservative TTL technology in the specialized boards greatly eased the design and debug phase.



Due to low funds, automated design support was limited to computer-aided drawing of printed circuit board layout and circuit diagrams. No circuit simulation tools were available and no back-annotation of printed circuit boards was possible. Even with this handicap, although small modifications were required on most boards, there was no complete redesign of any board. The low-cost CAD systems available today would speed up many of the design tasks.

Great care was taken in the robustness and compact mechanical layout of the system, as well as cooling. The system is air-cooled, and in the event that a cooling failure is detected by sensors in the machine cabinets, an automatic shutdown procedure is initiated. This proved essential for the reliability and long life of the system.

During the initial boot of the system, software support was a serious handicap. The PE CPU boards were low-cost VME bus 68000 systems, with only an EPROM-based low-level monitoring system. Thus, all programming, especially for the device drivers, had to be done in assembly language, which was error-prone and slow. The availability of a C compiler at a later stage improved the situation significantly.

Once the prototype was operational, several aspects of the system that previously had not been major considerations became more prominent. One of the first items that became apparent was the need for a programming and monitoring environment that would allow PASM users to access individual support and Parallel Computation Unit processors, and that would facilitate the loading, execution and debugging of applications on the prototype. For this reason, CAPS, described in section 3.4.2, was developed.

For a future system, it would be advisable and possible (due to advances in integration) to have an operating system kernel in each PE, with a LAN connection such as an Ethernet link for code distribution and debugging access. This would ameliorate many of the challenges associated with the EPROM-based low-level monitor system and the coding and debugging environment.

Through application studies using the prototype, various characteristics of mixed-mode processing have become evident. As one example, the ability to switch between SIMD and MIMD modes at the individual instruction level is more important than was originally thought (e.g. Fineberg, Casavant and Siegel, 1991). Another significant characteristic of the system is that MC/PE overlap can theoretically improve SIMD performance by 50%. Several studies have looked at the impact of MC/PE overlap and how to optimize it (Armstrong *et al.*, 1991; Kim, Nichols and Siegel, 1991). However, some of the burden of optimizing MC/PE overlap may be removed from the compiler if the MC processor is faster than that of the PEs. Then, perhaps, all serial computation common to all PEs can be done on the MC. Experimental studies are needed to determine the efficacy of such a strategy, given the PE and MC processor type and speed. In addition, the SIMD instruction synchronization mechanism proved to be an inexpensive and effective MIMD global barrier synchronization operation. This hardware support is a strong advantage compared with doing synchronization in software.

Perhaps the most important lesson learned from the construction of the PASM prototype is the value of the conservative use of technology in a proof-of-concept design. Much of the prototype was constructed with commodity chips and boards. The use of readily available technology (e.g. two-sided printed circuit boards and standard mechanical design) for the remainder of the system helped the team maintain a careful budget compared with traditional parallel system prototypes. The conservative use of technology also allowed the prototype to be built on a rigid schedule. At the time the small-scale prototype became operational in 1987, it did not have an impressive megaflop rating. However, its balanced, characterizable design and inherent reliability have supported years of algorithm studies. Now, after several years of operation, the absolute performance of the prototype is not an issue. With the rapid progress of technology, any extra money and effort spent to incorporate faster technology would not have improved its usefulness as a prototype and conceptual research tool, nor would it have increased its lifespan.

As stated in section 3.3.7, the goal for the prototype effort was to build a robust, small-scale, proof-of-concept machine for experimenting with the three dimensions of reconfigurability that the PASM design concept involved: partitionability, mode of parallelism and connections among the PEs. This goal was accomplished, and the prototype is supporting both research and education.

## 3.6 Summary

In this chapter, the PASM parallel processing system design concepts were overviewed and the control hierarchy as implemented in the prototype was described in detail. Salient features of PASM are its ability to operate in both the SIMD and the MIMD modes of parallelism and to switch dynamically between modes (mixed-mode), to be configured into one or more independent or cooperating submachines and to vary inter-PE connectivity. Hence PASM is dynamically reconfigurable along three dimensions: mode of parallelism, partitionability and communication among processors. The flexibility of this design permits the effective implementation of a wide range of applications. To aid in realizing the performance benefits, an explicitly parallel language (ELP) compiler and a tool for debugging and performance monitoring (CAPS) have been developed. The prototype serves as a testbed for evaluating the usefulness of the PASM design concepts and enhancements. A large group of faculty and students use the prototype to study reconfigurable parallel systems and their application to real computing problems.

Current projects associated with the PASM prototype include extending ELP (Nichols, Siegel and Dietz, 1993), examining the trade-offs involved with mixed-mode parallelism (e.g. Fineberg, Casavant and Siegel (1991)), automatic mode selection (Watson *et al.*, 1994), performing application experiments on the prototype and extrapolating to larger machines (e.g. Bronson, Casavant and Jamieson (1990)), optimizing SIMD control unit/PE execution overlap (Armstrong *et al.*, 1991), exploring ways to use partitioning (Nation, Maciejewski and Siegel, 1993), mapping tasks onto reconfigurable parallel machines (Siegel, Armstrong and Watson, 1992), enhancing CAPS (Lumpp *et al.*, 1991), exploiting visualization for understanding system behavior, investigating an automatic reconfiguration system for improving performance and fault tolerance (Chu *et al.*, 1989) and researching hardware design improvements (e.g. Nation *et al.* (1990)). The PASM prototype is a constantly evolving tool for studying the programming and design of parallel processing systems.



## Acknowledgements

A large number of people have made significant contributions to the design and development of the PASM concept and prototype. These people are the coauthors of the publications in the PASM reading lists in Siegel *et al.* (1987) and Armstrong, Watson and Siegel (1993). Numerous agencies have supported aspects of PASM-related research: Air Force Office of Scientific Research, Army Research Office, Ballistic Missile Defense Agency, Defense Mapping Agency, Naval Ocean Systems Center, Naval Research Laboratory, National Science Foundation, Office of Naval Research and Rome Laboratory. IBM provided a grant for much of the prototype equipment. Donations for various parts for the prototype were provided by Amphenol Products, Augat Inc., Belden, Motorola and Power One. The Purdue University School of Electrical Engineering Head (Professor Schwartz) and past Heads (Professors Coates and Hoefflinger) have been very supportive of the PASM endeavor. Section 3.2 includes material from 'The organization of the PASM reconfigurable parallel processing system', by H.J. Siegel, W.G. Nation and M.D. Allemang, in the proceedings of the *1990 Parallel Computing Workshop*, sponsored by the Computer and Information Science Department at The Ohio State University, March 1990, pp. 1–12. Section 3.3 includes material from 'Design and implementation of the PASM prototype control hierarchy', by T. Schwederski, W.G. Nation, H.J. Siegel and D.G. Meyer, in the proceedings of the *2nd International Conference on Supercomputing*, May 1987, pp. 418–27. This work was supported by the National Science Foundation under grant number CDA-9015696 and by NASA under grant number NGT-50961. The authors thank Janet M. Siegel, Gene Saghi and Pierre Pero for their comments on this manuscript.

## References

Adams III, G.B. and Siegel, H.J. (1982) The extra stage cube: a fault-tolerant interconnection network for supersystems. *IEEE Trans. Comp.*, **C-31** (5) 443–54.

Armstrong, J.B., Nichols, M.A., Siegel, H.J. *et al.* (1991) Examining the effects of CU/PE overlap and synchronization overhead when using the Complete Sums Approach to image correlation, in *Third IEEE Symp. Parallel and Distributed Processing*, IEEE Computer Society Press, Los Alamitos CA, pp. 224–32.

Armstrong, J.B., Watson, D.W. and Siegel, H.J. (1993) Software issues for the PASM parallel processing system, in *Software for Parallel Computation* (eds J.S. Kowalik and L. Grandinetti), Springer, Berlin, pp. 134–48.

Astfalk, G. (1992) Convex's view on TFLOPS computing, in *Parallel Computing and Transputer Applications* (eds M. Valero, E. Onate, M. Jane, J.L. Larriba and B. Suarez), IOS Press/CIMNE, Barcelona, pp. 51–61.
Barnes, G.H., Brown, R., Kato, M. *et al.* (1968) The Illiac IV computer. *IEEE Trans. Comp.*, **C-17** (8) 746–57.
Batcher, K.E. (1976) The flip network in STARAN, in *1976 Int. Conf. Parallel Processing*, IEEE, New York, pp. 65–71.
Batcher, K.E. (1982) Bit serial parallel processing systems. *IEEE Trans. Comp.*, **C-31** (5) 377–84.
Bell, T.E. (1992) Beyond today's supercomputers. *IEEE Spectrum*, **29** (9), 72–5.
Blank, T. (1990) The MasPar MP-1 architecture. *IEEE Compcon*, 20–4.
Bouknight, W.J., Denenberg, S.A., McIntyre, D.E. *et al.* (1972) The Illiac IV system. *Proc. IEEE*, **60** (4) 369–88.
Bronson, E.C., Casavant, T.L. and Jamieson, L.H. (1990) Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. *IEEE Trans. Parallel Distrib. Syst.*, **1** (2) 195–205.
Chu, C.H., Delp, E.J., Jamieson, L.H. *et al.* (1989) A model for an intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture. *J. Parallel Distrib. Comp.*, **6** (3) 598–622.
Crowther, W., Goodhue, J., Thomas, R. *et al.* (1985) Performance measurements on a 128-node butterfly parallel processor, in *1985 Int. Conf. Parallel Processing*, IEEE Computer Society Press, Washington DC, pp. 531–40.
Darema, F., George, D.A., Norton, V.A. *et al.* (1988) A single-program-multiple-data computational model for EPEX/FORTRAN, *Parallel Comp.*, **7** (1) 11–24.
Dennis, J.B., Boughton, G.A. and Leung, C.K.C. (1980) Building blocks for data flow prototypes, in *7th Ann. Symp. Computer Architecture*, IEEE, New York, pp. 1–8.
Dietz, H.G., Schwederski, T., O'Keefe, M.T. *et al.* (1989) Static synchronization beyond VLIW, in *Supercomputing '89*, ACM, New York, pp. 416–25.
Duclos, P., Boeri, F., Auguin, M. *et al.* (1988) Image processing on SIMD/SPMD architecture: OPSILA, in *9th Int. Conf. Pattern Recognition*, IEEE Computer Society Press, Washington DC, pp. 430–3.
Fineberg, S.A., Casavant, T.L., Schwederski, T. *et al.* (1988) Non-deterministic instruction time experiments on the PASM system prototype, in *1988 Int. Conf. Parallel Processing*, Pennsylvania State University Press, pp. 444–51.
Fineberg, S.A., Casavant, T.L. and Siegel, H.J. (1991) Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting. *J. Parallel Distrib. Comp.*, **11** (3) 239–51.
Flynn, M.J. (1966) Very high-speed computing systems. *Proc. IEEE*, **54** (12) 1901–9.
Fountain, T.J. (1981) CLIP4: progress report, in *Languages and Architectures for Image Processing* (eds M.J.B. Duff and S. Levialdi), Academic Press, London, pp. 281–91.
Gottlieb, A., Grishman, R., Kruskal, C.P. *et al.* (1983) The NYU Ultracomputer – designing an MIMD shared-memory parallel computer. *IEEE Trans. Comp.*, **C-32** (2) 175–89.
Hayes, J.P. and Mudge, T. (1989) Hypercube supercomputers. *Proc. IEEE*, **77** (12) 1829–41.
Hillis, W.D. and Tucker, L.W. (1993) The CM-5 Connection Machine: a scalable supercomputer. *Commun. ACM*, **36** (11) 30–40.
Hord, R.M. (1982) *The Illiac IV, the First Supercomputer*, Computer Science Press, Rockville MD.
Hunt, D.J. (1989) AMT DAP – a processor array in a workstation environment. *Comp. Syst. Sci. Eng.*, **4** (2) 107–14.
IBM Corporation (1993) *IBM 9076 Scalable POWERparallel 1*, IBM Corporation GH26-7219-0, Armonk NY.
Kim, S.D., Nichols, M.A. and Siegel, H.J. (1991) Modeling overlapped operation between the control unit and processing elements in an SIMD machine. *J. Parallel Distrib. Comp.*, **12** (4), 329–42.
Krishnamurti, R. and Ma, E. (1988) The processor partitioning problem in special-purpose partitionable systems, in *1988 Int. Conf. Parallel Processing*, Pennsylvania State University Press, pp. 434–43.
Kuck, D.J., Davidson, E.S., Lawrie, D.H. *et al.* (1986) Parallel supercomputing today and the Cedar approach. *Science*, **231** 967–74.
Lawrie, D.H. (1975) Access and alignment of data in an array processor. *IEEE Trans. Comp.*, **C-24** (12) 1145–55.
Lipovski, G.J. and Malek, M. (1987) *Parallel Computing: Theory and Comparisons*, John Wiley & Sons, New York NY.
Lumpp, Jr, J.E., Fineberg, S.A., Nation, W.G. *et al.* (1991) CAPS: a coding aid for PASM. *Commun. ACM*, **34** (11) 104–17.
Nation, W.G., Fineberg, S.A., Allemang, M.D. *et al.* (1990) Efficient masking techniques for large-scale SIMD architectures, in *Frontiers '90: The 3rd Symp. Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, Los Alamitos CA, pp. 259–64.
Nation, W.G., Maciejewski, A.A. and Siegel, H.J. (1993) A methodology for exploiting concurrency among independent tasks in partitionable parallel processing systems. *J. Parallel Distrib. Comp.*, **16** (3) 271–8.
Nichols, M.A., Siegel, H.J. and Dietz, H.G. (1993) Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler. *IEEE Trans. Parallel Distrib. Syst.*, **4** (2) 222–34.
Nugent, S.F. (1988) The iPSC/2 direct-connect communications technology, in *3rd Conf. Hypercube Computers and Applications*, ACM, New York, pp. 51–60.
Nutt, G.J. (1977) Microprocessor implementation of a parallel processor, in *4th Ann. Symp. Computer Architecture*, IEEE, New York, pp. 147–52.
Patel, J.H. (1981) Performance of processor-memory interconnections for multiprocessors. *IEEE Trans. Comp.*, **C-30** (10) 771–80.
Pease III, M.C. (1977) The indirect binary n-cube microprocessor array. *IEEE Trans. Comp.*, **C-26** (5) 458–73.
Pfister, G.F., Brantley, W.C., George, D.A. *et al.* (1985) The IBM Research Parallel Processor Prototype (RP3): introduction and architecture, in *1985 Int. Conf. Parallel Processing*, IEEE Computer Society Press, Washington DC, pp. 764–71.
Philippsen, M., Warschko, T., Tichy, W. *et al.* (1993) Project Triton: towards improved programmability of parallel machines, in *26th Hawaii Int. Conf. System Sciences*, IEEE Computer Society Press, Los Alamitos CA, pp. 192–201.
Seitz, C.L. (1985) The Cosmic Cube. *Comm. ACM*, **28** (1) 22–33.
Siegel, H.J., Armstrong, J.B. and Watson, D.W. (1992) Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems. *IEEE Comp.*, **25** (2) 54–63.
Siegel, H.J. (1990) *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, 2nd edn, McGraw-Hill, New York NY.
Siegel, H.J., Nation, W.G., Kruskal, C.P. *et al.* (1989) Using the multistage cube network topology in parallel supercomputers. *Proc. IEEE*, **77** (12) 1932–53.
Siegel, H.J., Siegel, L.J., Kemmerer, F.C. *et al.* (1981) PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comp.*, **C-30** (12) 934–47.
Siegel, H.J., Schwederski, T., Kuehn, J.T. *et al.* (1987) An overview of the PASM parallel processing system, in *Computer Architecture* (eds D.D. Gajski, V.M. Milutinovic, H.J. Siegel and B.P. Furht), IEEE Computer Society Press, Washington DC, pp. 387–407.
Thanawastien, S. and Nelson, V.P. (1981) Interference analysis of shuffle/exchange networks. *IEEE Trans. Comp.*, **C-30** (8) 545–56.
Tucker, L.W. and Robertson, G.G. (1988) Architecture and applications of the Connection Machine. *IEEE Comp.*, **21** (8) 26–38.

Watson, D.W., Siegel, H.J., Antonio, J.K. *et al.* (1994) A blocked-based mode selection model for SIMD/SPMD parallel environments. *J. Parallel Distrib. Comp.*, **21** (3) 271–88.
Wu, C.-L. and Feng, T.-Y. (1980) On a class of multistage interconnection networks. *IEEE Trans. Comp.*, **C-29** (8) 694–702.
Zorpette, G. (1992) The power of parallelism. *IEEE Spectrum*, **29** (9) 28–33.

# Part Two
# Theory and models