A BUS STRUCTURE FOR MULTI-MICROPROCESSING

by

RANDOLPH B. HAAGENS

B.S., Worcester Polytechnic Institute

(1975)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

(JANUARY 1978)

Signature redacted

Certified by

Signature redacted

Accepted by

Chairman, Department Committee

MAY 15 1978
A BUS STRUCTURE FOR MULTI-MICROPROCESSING

by

RANDOLPH B. HAAGENS

Submitted to the Department of Electrical Engineering and Computer Science on 20 January 1978 in partial fulfillment of the requirements for the Degree of Master of Science

ABSTRACT

The availability of inexpensive microprocessors argues for the application of multiple-microprocessor solutions to problem areas presently dominated by minicomputers and larger systems, as well as to new problem areas that are distributed logically or geographically. To avoid the need to replicate valuable resources, the system interconnection scheme should allow access to common program memory and I/O peripherals. Multiple-processor system organization is considered briefly, and a single time-shared bus structure is favored for reasons of flexibility and cost. A new bus protocol is developed to allow concurrent operation of several processors on a single bus. The protocol adapts readily to multiple-bus interconnections via bus couplers, supporting a variety of loosely- and tightly-coupled multiple-bus system configurations.

Name and Title of Thesis Supervisor: Hoo-min D. Toong

Asst. Prof. of Elec. Eng.

and Computer Science
CONTENTS

CHAPTER 1 Introduction and Overview

1.1 Motivation for interest in multiple-microprocessor systems

1.1.1 Factors favoring multiple-processor systems in general
1.1.2 Factors favoring multiple-microprocessor systems specifically
1.1.3 Examples of trend towards multiple-microprocessor systems

1.2 Goals for interconnection scheme
1.3 Faults with present single time-shared bus
1.4 Two-part solution
1.5 Implementation
1.6 Outline of thesis
1.7 References

CHAPTER 2 Multiple-Processor Systems: Definitions and Examples

2.1 Computer system organization

2.1.1 Classification by topology

2.1.1.1 Single processor system
2.1.1.2 Multiple-processor systems

2.1.1.3 Other organizations exhibiting concurrency

- pipelined processors
- parallel (array, vector) processors
- processing ensembles

2.1.2 Classification by job type
2.1.3 Classification by other parameters

2.2 Interconnection schemes for multiple-processor systems

2.2.1 Single time-shared bus
2.2.2 Serial forms
2.2.3 Parallel forms
2.2.4 Hierarchical forms
2.2.5 Other complex forms
2.2.6 Suitability of different structures to various computational modes

2.3 Examples

2.3.1 DEC PDP-11
2.3.2 Lockheed SUE
2.3.3 BBN Pluribus
2.3.4 Intel MDS-800
2.3.5 C.mmp
CHAPTER 3  A New Bus Structure -- Pended-transaction Bus

3.1 Motivation for PTB
3.2 Two-part solution
  3.2.1 Pended transactions
  3.2.2 Multiple-bus system

CHAPTER 4  Design of a Single Bus

4.1 Bus synchronization protocols
  4.1.1 Use of acknowledge and interlock

4.2 Bus arbitration
  4.2.1 Arbitration algorithms
    4.2.1.1 Fixed-priority algorithm
    4.2.1.2 Round-robin and snap-shot algorithms
  4.2.2 Centralized and decentralized arbitration
  4.2.3 An example
  4.2.4 Arbitration synchronization -- the arbiter problem

4.3 Pended-transaction bus
  4.3.1 Signals description
  4.3.2 Pended-transaction protocol
  4.3.3 Arbitration

4.4 Issues specific to the pended-transaction bus
  4.4.1 Device-busy problem
  4.4.2 Implementing software synchronization primitives

4.5 Alternatives to the pended-transaction bus
  4.5.1 Interleaved processors
  4.5.2 "Omniscient" bus controller
CHAPTER 5 Extension to a Multiple-bus system

5.1 Multiple-bus system -- general
5.2 Bus coupler design

5.2.1 Options for the design of the bus coupler
5.2.2 Implementation of serial and parallel interconnections

CHAPTER 6 Implementation Specifics

6.1 Design

6.1.1 Use of explicit virtual bus for initial design
6.1.2 Asynchronous logic implementation
6.1.3 Backplane
6.1.4 Design of system modules

6.1.4.1 System Bus
6.1.4.2 Bus Master Interface Unit (BMIU)
6.1.4.3 Bus Slave Interface Unit (BSIU)
6.1.4.4 Distributed Bus Arbiter (DBA)

6.2 Conceptual Design

6.2.1 Processor
6.2.2 Memory
6.2.3 Input/output
6.2.4 Shared arithmetic unit
6.2.5 Bus Coupler (BC)
6.2.6 Variations of the PTB protocol

CHAPTER 7 Conclusion

7.1 Summary
7.2 Suggestions for further work

Bibliography

B.1 Bibliography

Appendix

A.1 Photographs of prototype system
CHAPTER 1 Introduction and Overview

1.1 Motivation for interest in multiple-microprocessor systems

The present availability of low-cost microprocessors and memories has sparked a renewed interest among researchers and manufacturers, in a subject which has been with us nearly since the inception of electronic computing: multiple-processor systems. It has long been recognized that multiple-processor architectures offer various advantages over conventional uni-processor systems at a given level of component sophistication. However, the rapid evolution of logic technologies in the last decade has permitted conventional architectures to continue to satisfy rising user expectations, effectively postponing serious development of multiple-processor systems. Technology improvements are likely to diminish as the operating speed of circuits approaches the speed of light. Indeed, a significant increase in component speeds beyond these fundamental limits seems unlikely. Further enhancement of computer performance must come from new architectures which increase concurrency by exploiting parallelism.

Fortunately, the new technology has also produced inexpensive building blocks for constructing multiple-processor systems, thereby removing one of the impediments to their development: the cost of components. The remaining costs are those of system interconnection, software, and input/output peripherals. This thesis addresses interconnection issues, and develops a new bus protocol for multi-microprocessing.

1.1.1 Factors favoring multiple-processor systems in general

Three advantages claimed for multiprocessor systems that serve as motivations for their development are identified in [ENSL74]:

- throughput
- flexibility
- reliability

The unifying theme here is system performance, usually quantified as cost-effectiveness. These same characteristics are also attributable, in varying degrees, to multiple-processor systems in general.

THROUGHPUT. The use of multiple processors offers improved throughput by introducing simultaneous, rather than merely concurrent, processing of
parallel tasks. The advantage of simultaneous processing is apparent when one considers that a typical job mix includes many tasks which may be processed independently, or nearly so (e.g., a multiple-user, interactive system). The assignment of a task to a processor means, ignoring resource contention problems, that the task can proceed as rapidly as if it "owned" the processor in a uni-processor system. (Actually, multiprogramming is used in general-purpose multiprocessing systems for reasons of efficiency and further increased throughput.)

The turn-around time for an individual job will not be improved unless parallel tasks within the job are identified. Even so, the processes created for these tasks may not be completely independent, and thus will require synchronization, thereby lowering throughput. Furthermore, the identification of parallelism is itself the object of research. Evidently, the best use for multiple processors is for multiple independent jobs; yet significant improvement in the processing time for certain jobs can be achieved if parallelism can be exploited.

FLEXIBILITY. If the only motivation for multiple-processor systems were increased throughput for independent tasks, then a collection of separate computers would be adequate. Significant benefits can however be had through the sharing of system resources, through the establishment of pools of resources [LEHM66]. For example, a particular task may require a larger amount of memory than that available on a single computer system. A multiple-processor system with performance comparable to the aggregate of individual processors would have a correspondingly larger available pool of memory, and the task could run. Resource sharing extends beyond memory pooling to the sharing of exotic or expensive input/output devices, large data bases, and program libraries. At a higher level, ideas, experience and results may be shared among the users of a multiple-processor system.

Another aspect of flexibility is the capability for smooth system growth through modular, incremental expansion. The success of the IBM 360/370 and the DEC PDP-11 series of computers attests to the desirability of providing a range of compatible systems. Presently this is accomplished by offering a family of computers configured around a number of processors. When the user's needs increase beyond the capabilities of his present system, he is required to upgrade to a faster processor. Furthermore, the performance spectrum covered by the various family models
is seldom continuous, complicating the decision to upgrade. Multiple-processor systems promise smooth growth in performance through the addition of various modules. The interconnection scheme and operating system will necessarily also be capable of modular growth.

RELIABILITY. The pooling of hardware resources that is typical of multiple-processor systems also gives rise to the possibility of improved reliability. The ultimate goal for a reliable system is fail-safe operation. To the extent that perfectly reliable operation cannot be achieved, two areas of emphasis may be identified: availability and integrity.

The requirement for availability means that the system should remain operational, though perhaps at a reduced performance level or increased error rate, even though certain modules may have failed. The requirement for integrity means that the results produced by the system should be correct, even if a delay or other cost is incurred. Reliability is achieved when sufficient availability and integrity are present.

Fault-tolerant systems try to achieve both high availability and high integrity. Generally, this means the use of duplexed or triplexed functional units, with voting on the results of operations. Failed modules are detected in this way and either the failure is corrected, or the module is replaced from a store of good reserve modules. If errors are detected before irreparable damage to data bases can occur, and if the system meanwhile continues to meet its realtime constraints, then fail-safe operation is attained.

Another class of systems, called fail-soft systems, exhibits the property of graceful degradation. These systems employ very little redundant hardware. When a failed unit is detected, it is eliminated from the system, and the performance of the system is correspondingly degraded. If the initial performance level is sufficiently high, then the availability criterion will continue to be met even after several failures.

Although reliability may be achieved other than through the use of multiple-processor systems, most highly reliable systems use specialized multiple-processor architectures. Indeed, every multiple-processor structure exhibits two properties which make improved reliability possible:
on-line debugging capability, and reconfigurability. On-line debugging means that the working portion of the system can be used to diagnose and debug the remainder; reconfigurability means that failed modules may be removed (electrically or physically), and the system reconfigured to make optimum use of the remaining operational hardware. The majority of the multiple-processor structures to be discussed in chapter two offer the possibility of on-line debugging, and to varying extents, reconfigurability.

It is interesting to note that historically, the interest in multiple-processor systems was motivated by the three factors cited, but in reverse order of importance. Conventional uni-processor systems provided ample throughput; and flexibility, as a goal was under-rated. The development of multiple-processor systems emphasizing throughput is a more recent phenomenon, but will become increasingly important.

Given the advantages cited for multiple-processor systems, one may wonder why they have received so little attention from computer manufacturers. Bell provides some possible explanations [BELL76]:

- conservativeness of engineering;
- the market does not perceive the need;
- faster processors are more attractive;
- many ideas for uniprocessors are as yet untried; and
- organizational, planning, and management problems.

1.1.2 Factors favoring multiple-microprocessor systems specifically

The overwhelming factor favoring the use of multiple microprocessors for research and for practical applications is their minimal cost. It is reasoned that a large number of microprocessors can be configured to provide increased performance at less cost than comparable uni-processor systems. Second, the modest performance of present-day microprocessors means that memory and other system resources are often under-utilized, inviting the addition of processors. Third, the availability of microprogrammable microprocessors means that CPU modules can be designed which are tailored to multiple-processor systems [REYL74].

A number of problems have been encountered, however, in the
The development of multiple-processor systems. Foremost is the need for an interconnection scheme whose cost does not negate the initial advantage of using microprocessor parts. Second is the poor state of present microprocessor architecture, particularly their persistent unprogrammability (e.g., the lack of addressing modes for supporting advanced software, lack of instructions for implementing synchronization primitives). Third, there remains a host of traditional problems associated with multiple-processor systems. Last, and most important, there is an observed tendency among microprocessor users and researchers to re-invent the wheel. A similar phenomenon was observed in the mid-60's among minicomputer users. Apparently, each new wave of computing technology brings with it people who are new to the profession; and their inexperience, coupled with the perceived need on behalf of manufacturers and researchers alike to proclaim revolution, often results in confusion where instead there might have been orderly development.

The drawbacks of present-generation microprocessors should not discourage their application to multiple-processor systems. The knowledge gained from such attempts will affect future generations of LSI parts. Already microprocessor modules are available which support multiple-processor interconnection [INTE77]. Further improvements in microprocessor architecture and in the application of LSI to interconnection may be expected.

### 1.1.3 Examples of trend toward multiple-microprocessor systems

Since 1972 many multiple-microprocessor systems have been proposed, by both researchers and manufacturers [ADAM76, ANDE75, ARDE75, BASK72, BAUM74, CATL76, DRYS74, FULL76, INTa75, JORD74, LAUG76, PEAS75, RAPA75, WEIS74, WIDD75]. Many different organizations have been suggested, but two have predominated in successfully implemented systems: shared-bus multiprocessing, and "distributed" systems.

One of the earliest multiple-microprocessor systems was the Intel MDS-800 development system. The full MDS system uses four microprocessors. One processor (i4040) controls the pROM programmer (UPM). Since the UPM is treated as an I/O device, this is an example of a "distributed" connection. Other processors are used for the CPU (i8080), ICE-80 in-circuit emulator (i8080), and floppy disk channel controller.
(i3000 series). These three processors may share the system bus in a multiprocessor fashion, though normally each is assigned a dedicated function.

Another Intel product, the SBC-80/20 single board computer [INTE77] is being offered for distributed control applications [ADAM76]. Up to four SBC processor modules may share a single bus with compatible memories and I/O's, and these clusters may in turn communicate via high-speed serial links.

Another application area for multiple microprocessors is in families of compatible high-performance peripherals for minicomputer systems. The use of micros to control "smart" CRT terminals began in 1974, when RS-232 was the only accepted interface standard. The adoption of the IEEE standard 488 interface bus has since prompted the development of medium-bandwidth, shared-bus peripheral systems, with the interface bus often replacing the normal I/O bus of the minicomputer system.

1.2 Goals for interconnection scheme

The development of general-purpose multiple-microprocessor structures is desirable not only because such systems will have direct application in computing, but also because a general-purpose structure is most readily adapted to specific problems, and can remain viable as system requirements evolve. Accordingly, the following goals are proposed for the interconnection scheme:

- its cost should be low, commensurate with other system hardware costs;
- it should provide for a high degree of resource sharing, first to avoid duplication (and the attendant cost), but more importantly, to facilitate smooth system evolution (flexibility); and
- it should provide for the modular interconnection of between two and approximately fifty processors, together with associated memories and I/O's (modular expansion).

These goals point directly to the choice of a single shared-bus scheme. In fact, it is for these reasons that shared-bus architectures are employed almost universally by micro- and mini-computer manufacturers.
1.3 Faults with present single time-shared bus

There are two disadvantages of the single-bus approach [DAVI72]:

(1) limited bandwidth, and
(2) possibility of catastrophic failure due to the failure of a single component in the bus.

Objection (2) can be dealt with rather handily by removing all active components from the bus to the modules themselves, making the bus as reliable as a printed-circuit board, for example. Later it will be shown that this has implications for the design of the bus controller.

Objection (1) poses a serious limitation for high-speed systems, but need not do so for microprocessor-based systems. Consider the 8080 microprocessor, for example. The 8080 makes at most one memory reference (read or write) every machine cycle (1500-2500 nanoseconds). A properly designed bus could have a cycle time of 100-200 ns, suggesting that ten processors could share the bus without significant contention [REYL74].

Unfortunately, present microprocessors were not designed with multiprocessing in mind. One consequence has been that processor bus utilization is unnecessarily high. The 8080, for example, actually uses the bus for approximately 750 ns of every machine cycle. Studies have shown that typical processors have bus utilizations of 40 to 72 percent [GER976]. This level of bus utilization is apparently too high for any degree of multiprocessing.

It is probably for this reason that the single-bus system has been rejected for multiprocessor use by most investigators [CHEN74, DAVI72, THUR72].

1.4 Two-part solution

The solution developed in this thesis consists of two parts. First, a distributed bus switch is developed which uses a new protocol to implement "pended transactions". This technique reduces the bus bandwidth requirements of processors and memories significantly, allowing up to eight microprocessor-based CPU modules and compatible memories to be configured on a single bus. The bandwidth reduction is accomplished by providing each device (processor, memory, or I/O) a "virtual" bus, while actually freeing
the bus during lengthy transactions, thereby allowing concurrent use of the bus by several processor-memory module pairs.

Second, large systems may be assembled using multiple buses interconnected via bus couplers. The pended-transaction protocol, together with these bus couplers, provides for message-switched transactions between buses while maintaining the appearance of a single system-wide virtual bus. The use of message rather than circuit switching means that the bus couplers exert very little load on each system bus; thus, for a properly partitioned system, the apparent bandwidth of the virtual bus is increased nearly proportionally to the number of buses in the system.

1.5 Implementation

The prototype pended-transaction bus (PTB) system was implemented using Intel SBC family processors, memories and I/O's. The virtual bus protocol is exactly that defined by Intel for these family modules. The modules are physically augmented with additional logic which maps the SBC bus protocol to the PTB protocol. The design of these Bus Interface Units (BIUs) was a major part of the thesis. To maximize their speed, the BIUs are implemented with asynchronous logic. It is reasonable to augment the SBC modules in this way, since future LSI processors and memories could be designed to implement the protocol directly.

1.6 Outline of thesis

The thesis progresses from introductory remarks to the circuit implementation of the prototype PTB system in a step-by-step fashion. The development of the PTB design is interrupted frequently to introduce theory, general issues and analysis.

Chapter two reviews multiple-processor system terminology, taxonomy, and interconnection schemes, and analyzes some existing systems. The reader has no doubt been somewhat confused by the terminology used in the introduction; chapter two should alleviate this distress.

Chapter three motivates the pended-transaction bus in greater detail, drawing on earlier work to support the claims made. Pended transactions and multiple-bus systems are described. The reader should develop a good intuition for the PTB system. Very rapidly it develops that more theory is
needed to continue the discussion.

Chapter four introduces the design considerations for a single bus system. Bus synchronization, arbitration and reliability are discussed. This new information is then applied to the design of a single pended-transaction bus. Several problems with the PTB are identified. Alternatives to the PTB are considered and rejected.

Chapter five considers the extension to a multiple-bus system. The discussion centers around system partitioning and topology, and options for the design of the bus coupler.

Chapter six documents the prototype system design in greater detail, and then describes the conceptual design of several additional modules. This should provide the basis for project continuity.

Chapter seven sums up and makes suggestions for further work.

Also included are a Bibliography, and in the Appendix, photographs of the prototype system.

1.7 References

A general discussion of multiple processor systems may be found in [ENSL74, LORI71, BAER75, MOSI68]. The design of a fault tolerant multiprocessor is discussed in [HOPK75]. The advantages of multiple microprocessors were first discussed by [REYL74], and later by [BORG76, WEIS77]. A number of references to proposed multiple-processor systems were given in section 1.1.3. General issues of interconnection are the subject of [CHEN74, THUR72, DAVI72]; these will be treated in detail in chapter two. The author's introduction to the PTB idea is due to [BUNZ75]; the author's initial work is described in [HAAG76]. Similar systems are detailed in [CONW77] and [HORN75]. The Intel MDS and SBC systems are documented in [INTa75, INTE77].
Chapter 2 Multiple-Processor Systems: Definitions and Examples

The motivating assumption for multiple-processor system design is that a structure can be found which yields increased performance at a given level of component sophistication; this structure will bring a number of processors to bear on a problem by exploiting parallelism, with the result that computational concurrency is maximized. A number of different multiple-processor structures are presently in use. The purpose of this chapter is to develop the terminology needed to describe these systems, and to present a few examples.

Unfortunately, the meaning of some very popular words and phrases has become clouded through misapplication, to the point where these terms often are at best meaningless, and at worst misleading. Indeed, most authors find it necessary to define (or redefine!) these terms for each paper they write. This author is no exception. However, an attempt has been made to seek out the original meaning of disputed terminology. Where this was not possible, or where the original meaning bore no relation to present use, a consensus was sought. The result is that the definitions offered will not be agreeable to some. Some authors have felt the need to define new terminology to solve this problem. Worse, some have appropriated for redefinition previously well-defined but obsolete or underutilized words. This author is not so presuming; no new terms or definitions are presented here.

A case in point is the distinction between multiprocessing and the use of multiple processors. The term multiprocessor refers to a very small class of systems where, among other criteria, each processor enjoys equal access to a shared main memory pool. Multiple-processor systems include multiprocessors as a subset. Also included are all other systems which use more than one complete processor. Specifically excluded are systems composed of partial processors commonly called parallel-processor systems.

2.1 Computer system organization

A discussion of computer organization is necessarily descriptive, due to the evolutionary nature of present designs and design processes (e.g., [BELL71]). A number of taxonomies, or classification techniques, have been proposed. No perfect taxonomy (one that describes all existing and
foreseen systems) has yet been found. Nevertheless, an examination of some existing classifications will provide considerable insight.

2.1.1 Classification by topology

This most intuitive technique classifies systems according to the way they are assembled, and what sorts of interaction are supported.

2.1.1.1 Single processor system

The simplest, and earliest, computer organization is the five-unit von Neuman machine diagrammed in (fig. 2.1) [ENSL74]. This architecture was not in great use in large computer systems by 1960, but was resurrected by mini-computer designers and later by the architects of microprocessors. The principle weakness of this organization is the routing of I/O operations through the arithmetic logic unit (ALU), which requires the use of programmed I/O. The earliest systems used a technique called device polling, whereby the processor waits for the device to be ready before performing a transfer (I/O read or write). The problem is that all processing stops while any I/O transfer is in progress. Later, device interrupts were added so that computation could be performed concurrently with I/O device motion. The ALU was still involved with the actual transfer of data.

The next steps in the evolution of single processor systems were the addition of Direct Memory Access (DMA), and then the I/O Channel. The early DMA systems gave the I/O device a direct data path to memory, but relied on the control portion of the central processor to effect the data transfer. The removal of transfer control from software to hardware meant an order-of-magnitude increase in speed, but still required the cessation of processing during I/O. The introduction of I/O channels (fig. 2.2) finally allowed concurrent I/O. The channel is actually a special-function processor which executes a channel program stored in main memory, thereby accomplishing the data transfer. The main processor's involvement is limited to setting up the channel program, initializing the channel, and processing the interrupt generated by the channel on completion of the transfer.
2.1.1.2 Multiple-processor systems

Multiple-processor systems are comprised of more than one complete processor functioning cooperatively as part of a single overall system [SEAR75]. The term processor (P) refers to the combination of the control unit (CU) and the ALU (fig. 2.3). As defined, the processor is complete—capable of interpreting and executing a single stream of machine instructions. When a number of processors are combined together with memory (M) and I/O (IO) modules, the result is termed a multiple-processor system, regardless of the topology of the interconnection. This broad definition includes multicomputers, multiprocessors, computer networks, distributed systems, and "polyprocessors", all of which are described below. It excludes systems of special-purpose or incomplete processors (e.g. pipelined processors, vector and array processors, and processing ensembles). These systems are described separately.

MULTICOMPUTER SYSTEMS are the earliest instance of multiple processors. Two or more computers cooperate but do not share main memory. Communication is via I/O channels, I/O device interfaces, or I/O medium transport. Multicomputer systems may be further classified according to the degree of coupling between the separate computers.

Uncoupled systems have no on-line connection whatever. They communicate via I/O medium transport. Examples of uncoupled systems are complete back-up systems and peripheral stand-alone systems. Complete back-up systems utilize two identical or nearly identical systems (fig. 2.4). The systems may be either operated in parallel to verify results, or one system may be held in reserve in case of failure of the active system. Peripheral stand-alone systems were introduced to speed the operation of the central system by preparing input and processing output. In a typical configuration (fig. 2.5), cards are read in, and the data is stored on magnetic tape. The tape is transported to the main computer, which reads it at high speed, runs the program, and produces an output tape, which is converted to listings, etc., in a similar fashion.

Coupled systems either share a common hardware resource such as a tape drive or disk unit (loose coupling), or are connected electrically via I/O channels or shared memory (tight coupling) (fig. 2.6). The major distinction between loose and tight coupling is the effective rate of data transfer. Coupled systems tend to be asymmetric in the sense that a main
(or central) processor is clearly discernible. The secondary processor(s) may be used for communication or, as in the case of peripheral stand alone systems, for I/O preparation and processing. Communication is accomplished in a "mailbox" fashion through the shared device (often a disk or drum) or shared memory.

The principal difference between multicomputers and multiprocessors, is that in a multicomputer system, the computers employ distinct operating systems, each of which treats the other as an I/O device. Interaction is therefore limited to the dataset level.
(fig. 2.1) Single processor system [ENSL74]

(fig. 2.2) I/O channel [ENSL74]
Memory unit

Arithmetic-logic unit

Control unit

I/O channel

(fig. 2.3) Processor [ENSL74]

SYSTEM A

system A results

input data

SYSTEM B

system B results

agreement?

output data

(fig. 2.4) Complete back-up system
(fig. 2.5) Peripheral stand-alone system

(fig. 2.6) Tightly-coupled multicomputer system [ENSL74]
MULTIPROCESSOR SYSTEMS, in contrast, are designed explicitly for the purpose of sharing as many resources as possible. The following characteristics of multiprocessor systems have been identified [ENSL74]:

- There are two or more processing units.

These may be identical (homogeneous systems), or merely similar in capability (symmetric systems). The definition may be stretched to include dedicated processors of various sorts; the result is termed an asymmetric system.

- Main memory must be shared by all processors.

By "main" memory is meant the largest block of memory in the system. Individual processors are allowed a small amount of private memory; but multiple processor/memory combinations configured around a small shared "communications page", would be classified as multicomputers.

- I/O access must be sharable to include channels, control units and devices, as appropriate.

The requirement for I/O sharing is not as strong as that for memory [HWAN74]. It suffices that each processor be able to access each device, however indirectly.

- There must be a single integrated operating system.

Three types of operating systems are commonly used. In the separate executive scheme, the system is partitioned into separate subsystems, which do not communicate. Each processor supervises and performs services for the tasks assigned to it; the system is effectively partitioned into several distinct computers. Because subsystems cannot communicate, this scheme offers little advantage over separate computers. In the master/slave system, one processor (the master) executes the operating system and performs all services for the program (or slave) processors. In the third case, the executive floats among symmetric (sometimes called anonymous) processors. A complex set of locks assures that critical code and tables may be accessed by only one processor at a time. A task may be executed by several processors during its progress through the system.

- There must be the possibility of interaction at all hardware and software levels, i.e.,
- at the level of execution of system tasks;
- at the program level--ability to share common datasets;
- at the hardware level--ability for two processors to simultaneously execute the same code, and ability to direct device interrupts to any processor.

COMPUTER NETWORKS. For most purposes, a computer network may be defined as a group of computers, each with a full complement of memory and I/O, distributed geographically, but connected for the purpose of user convenience for the sharing of system resources; e.g., program libraries, data bases, sometimes I/O devices, but never memory. The interconnection must involve data communications equipment [SEAR75]. When the various computers are in reasonable proximity, so that data communications equipment is not needed, but the system retains the other aspects of a network, notably the use of a communications protocol, the system may be termed a local network.

Computer networks may be compared with multicomputer systems. Both are a collection of computers which interact primarily at the data set level, i.e., treat each other as I/O devices. However, an examination of multicomputer systems usually reveals a main computer and one or more subordinate computers, coupled to better support the needs of a single user community. In contrast, a computer network is a union of equals, which brings together several user communities.

 DISTRIBUTED SYSTEMS. A recent discussion [SIGA76] produced no agreement as to the meaning of this term. According to [JENS76] distributed processing "entails a processing load partitioned across multiple processors which intercommunicate." Jensen defines four categories of distributed systems which together encompass the entire range of multiple- and parallel- processor systems, but then points out that popular usage of the term equates it to "multiple general-purpose processors", i.e., what we have been calling, obviously enough, multiple-processor systems. The identity of distributed systems is further clouded by those who qualify the word "distributed" with attributes such as "functionally", "physically", "geographically", and "logically".

Farber may have been the originator of the term [FARB73, FARB74], yet he, like Jensen, applies it to a wide range of previously well-described systems. Apparently any system with more than one processor is potentially
a distributed system. Thus, computer networks are geographically distributed, and local nets are physically distributed. Multiprocessor systems may not be distributed at all, unless a static task allocation is used, in which case they are logically distributed. If the form of the system follows the task allocation, then the system is functionally distributed. Thus a peripheral stand-alone multicomputer system is functionally distributed. The use of remote processors, communications front ends, dedicated-function processors in a multiprocessor system, and intelligence in data concentrators, multiplexors, and terminals are all examples of distributed processing. In addition to distributed processors, one finds distributed data bases, where part of the data base is maintained by each of a number of processors. And to make it all work, one needs distributed communication.

There is a class of systems which cannot be adequately described by the previous terminology and therefore is a candidate for being called distributed. These systems are organized in a similar fashion to computer networks, yet it is desired that the individual processors remain anonymous so that the user need not be concerned with the task or database distribution within the system. This aspect is similar to a symmetric multiprocessing system under floating executive control, in that tasks may migrate from processor to processor. An example of such a system is Farber's Distributed Computing System (DCS). It was generally agreed in [SIGA76] that this definition is more acceptable than the functional specialization notion which implies a static task allocation and might better be described as producing a "partitioned" system.

POLYPROCESSORS. This term has recently appeared in the literature on multiple-processor systems [JOSE72]. A polyprocessor system may use any of the previously described topologies, in whatever combination. There is usually a static partitioning of tasks over at least part of the system. For example, one processor might be devoted to instruction decoding, another to operand fetching, a third to memory management, and so on.

2.1.1.3 Other organizations exhibiting concurrency

Concurrency may be exploited other than through the use of multiple processors. The special techniques described below involve the replication, in time or in space, of some function or functional module
other than the complete processor module introduced in section 2.1.1.1.

PIPELINE PROCESSORS. Some complex computer operations consist of a sequence of steps; e.g., a floating point addition involves normalization, exponent comparison, and so on. Operands can be processed in a sequential fashion; however, considerable improvements in throughput may be had by processing a series of operands, in an assembly line fashion. Once the pipeline is filled with operands, new results are available each time the pipeline advances. Obviously, to properly exploit the pipeline, it must be kept filled. Pipeline systems achieve concurrency by simultaneously performing different operations on a single data stream.

PARALLEL PROCESSORS. Although this term has been applied in general to systems which exhibit concurrency, it is better limited to systems which operate on multiple related data streams in parallel, i.e., simultaneously. This is the class of array and vector processors. A representative array structure is shown in (fig. 2.7). A vector structure is similar, but one-dimensional. The system consists of a two-dimensional array of processing units under the control of a supervisor processor. The supervisor decodes the single instruction scheme and issues control signals to the processing array. At each step, the same operation is performed by every processor which is enabled by the supervisor. In this way, arrays of data are processed very rapidly.

PROCESSING ENSEMBLES. This unconventional architecture employs a number of identical special-purpose processing elements under the control of a host computer and a number of special global control units. The host issues instructions to the global control units which each control one aspect of processing element operation [EVAN73].
Control processor or network distributor

Supervisor processor

(fig. 2.7) Typical array processor organization [ENSL74]
2.1.2 Classification by job type

In the preceding, we have been concerned with the topological arrangement of the functional units which form a system. A different taxonomy [FLYN72] is based on the type of jobs a system is designed to handle. These jobs are characterized by number of independent instructions and data streams, and their interaction.

Single-instruction stream, single data stream (SISD) systems. These are the conventional single-processor systems.

Single-instruction stream, multiple data stream (SIMD) systems. Array, vector, and associative processors, as well as processing ensembles belong to this group.

Multiple-instruction stream, single data stream (MISD) systems. This class is included for completeness. An ancient example is the plug-board programmed data processor. A pipeline processor qualifies if each step in the pipeline is considered to be a separate "instruction".

Multiple-instruction stream, multiple data stream (MIMD) systems. These systems run more than one independent task, concurrently. Multiprocessors and multiprogrammed uniprocessors are included.

2.1.3 Classification by other parameters

Anderson has proposed a taxonomy based on a hierarchy of design decisions [ANDE75]:

- transfer strategy (direct, indirect).

A direct transfer strategy involves no routing decisions (choice among alternate paths) or address translation.

- transfer control method (centralized, decentralized).

A centralized strategy uses a single switching entity (e.g. star).

- transfer path structure (dedicated, shared).

A "path" may be either a bus or memory. Shared paths introduce contention and, therefore, arbitration problems.

- system architecture (e.g., star, ring, shared bus).
This taxonomy accommodates many existing structures (the crossbar switch is however noticeably missing). Other systems are seen as combinations of the systems which are described.

If a universal taxonomy is ever developed, it will probably need to consider more factors than system topology, interconnection, and number of instructions/data streams. Some of these are suggested below.

TASK DISTRIBUTION. How are tasks created, distributed to processors, and finally killed? Cyclic or permanent tasks may be assigned at design time (e.g., dedicated-function systems) or at system initialization. Other tasks are created by existing processes or in response to external events. Are tasks allowed to migrate from the processor at which they were created? At what cost?

RESOURCE ALLOCATION. Are devices sharable among processors? among processes (however indirectly)? Is shared access to databases supported? How are resources requested and assigned?

INTERPROCESS COMMUNICATION. Can processes be addressed without knowledge of their location?

2.2 Interconnection schemes for multiple-processor systems

We now consider some of the interconnection schemes which have been used for multiple-processor systems. The structures presented tend toward the general-purpose, for two reasons: (1) each instance of a special-purpose architecture is motivated by the application, and therefore unique; and (2) the emphasis of the thesis is on multiprocessing, which by its nature requires a general-purpose architecture.

Following [CHEN74], the communications bus is taken as the primitive for interconnection systems. The bus consists of a number of parallel signal lines for the communication of address, data, and transaction control information. Buses may be separated into two generic types: dedicated and undedicated [THUR72].

Dedicated buses are permanently assigned either to one function or to one pair of modules. The principal advantage of a dedicated bus is high throughput; there is little or no contention for the use of the bus, and bus protocol may be greatly simplified. An example of a system employing a
dedicated bus is shown in (fig. 2.8). In this system, the central processor enjoys private access to memory through the memory bus; the device controllers share access through the device bus.

Non-dedicated buses are shared by several modules and/or functions in a timeshared or multiplexed fashion. Their design is more complex, due to the requirement for explicit device addressing and transaction synchronization, and especially due to the need to arbitrate among potential bus users (masters). Nevertheless, the advantage of configuring multiple devices on a single bus has made them the building block for many interconnection schemes.

2.2.1 Single time-shared bus

The single time-shared bus (fig. 2.9) is the simplest multiprocessor structure. A number of modules (processors, memories and I/O's) share the bus on a time- or demand- multiplexed basis. A single bus communication involves the transfer of one word of data between modules, and is called a transaction. When a device initiates a transaction it is called the master. After being granted control of the bus, the master supplies address and control information which identify the slave module and the type of transaction (read or write, memory or I/O). The transaction is complete when the slave acknowledges the request, which may be done either explicitly through the use of an acknowledge signal, or implicitly, according to a designed-in protocol.

Permission to use the bus is generally granted in one of two major ways. Strict time multiplexing means that each potential master module is assigned a fixed-length time slot during which it may use the bus, on a rotating basis. If a module does not need to use the bus when its time arrives, the time-slot is wasted. Demand multiplexing involves the use of a bus arbiter to grant bus access to one of the requesting modules whenever the bus becomes free. The manner in which the bus is assigned is called the arbitration algorithm. Demand-multiplexing eliminates the wasted time slot problem at the cost of introducing additional hardware. Nevertheless, most systems use demand multiplexing for its greater efficiency.

The single bus offers several advantages. First, there is total freedom of communication between modules using the bus. This universality means that the hardware enforces no logical partioning of the system i.e.,
that multiprocessing is feasible. Second, no other interconnection scheme offers more minimal connectivity. Each module has only one port: its interface to the bus. The low complexity of a single-bus system makes it potentially economical. Third, the single bus offers flexibility, in the sense that it is easy to add or remove modules.

The primary deficiency of the single bus is that it offers only limited bandwidth, which is rapidly exhausted as more master modules are added to the system; thus the bus may become a communications bottleneck. Further, traditional single-bus schemes do not support the concurrent communication; this means that only one master/slave pair can communicate at a time. Third, the existence of a central bus arbitrator may pose a reliability problem in the sense that its failure will be catastrophic. Last, the bus is limited both in its length and in the number of modules which may share it. These limitations are due to the fact that bus signals must travel the length of the bus, which has impact on the design of bus drivers and receivers and the speed of the bus protocol.

The limitations of the single bus scheme can be overcome (at the expense of additional complexity) by adding buses to the system. For a properly partitioned system this has the effect of increasing the available bus bandwidth and the number of modules supported, without incurring the problems associated with long buses. Two basic interconnection forms have been identified by [CHEN74]: serial and parallel.
(fig. 2.8) System employing dedicated bus

(fig. 2.9) Single time-shared bus
2.2.2 Serial forms

Serial interconnection forms are distinguishable from parallel forms by the fact that they offer only a single route for transaction information. The simplest interconnection of two or more buses is provided by the use of a bus coupler (fig. 2.10). The effect of the bus coupler may be viewed as partitioning the virtual bus into two separate buses. If the partition is "good", then the use of the bus coupler will be minimized, and the system will exhibit concurrency on the order of two simultaneous bus transactions.

Two basic switching methods are available to the designer of the bus coupler. Circuit switching involves the electrical connection of the buses for the period of the transaction, while message switching refers to the relaying of the transaction information through the bus coupler. These are discussed further in chapter 5.

The second serial organization is the star. The simple star (fig. 2.11) connects each module to a central switching center via a single dedicated bus. All communication between modules is routed through the switching center, which is assumed to be a serially-reusable (i.e., time-shared) data distributor (e.g., fig. 2.12). The principal advantage to this scheme is that the complexity is concentrated in the switching center, providing simplified interfacing. On the negative side, the bandwidth of the switching center will be limited, and its reliability can be a problem.

The generalized star (fig. 2.13) model interconnects multiple buses, each of which supports several modules. Off-bus requests are coupled through the switching center which may be thought of as a multiport bus coupler. The obvious advantage is that, to the extent that the system is well partitioned, the bandwidth requirement for the switching center is reduced.

The third serial organization is the ring, which is most often found in the form shown in (fig. 2.14). Data flow is usually unidirectional and may be either buffered or unbuffered. Buffered rings transmit fixed-length packets of information between the nodes of the ring; nodes receive packets, determine their relevancy to the processes active at the node, and retransmit them. Unbuffered rings transmit data nearly continuously,
subject only to synchronization delays as the message passes through each node. The unbuffered ring offers the possibility of variable message length and shorter transmission delay, at the expense of slightly greater overhead.

The ring overcomes some of the star's limitations by distributing the switching function to the separate nodes, at the cost of increased total complexity. Reliability is improved, since it is possible to design the system so that a failed node can be removed by joining its input and output ports. Flexibility is also increased, since it is possible to add nodes by breaking the ring at any node. The major disadvantages of the ring are due to its serial nature; it exhibits transmission delays and finite bandwidth, both of which become severe limitations as the size of the ring (number of nodes) is increased.
(fig. 2.10) Simplest serial interconnection of two buses

(fig. 2.11) Simple star
(fig. 2.12) Example of a data distributor
("Pass the data, please...")

(fig. 2.13) Generalized star
(fig. 2.14) Ring
2.2.3 Parallel forms

Parallel interconnection schemes may be identified by the fact that they offer the possibility of multiple routes between modules.

The simplest parallel organization uses two parallel time-shared buses (fig. 2.15). With this scheme, concurrency can equal two. Also, the availability of an alternate path enhances reliability. However, a good deal of complexity is introduced at the module interface, due to two sources. First, a master module must decide which bus to use. This decision could be made statically, by assigning each master module to a single bus; but the capabilities of the system will be better exploited if the decision is dynamic. Second, each slave module must be able to arbitrate requests from both buses, requiring for example, the design of dual-port memories.

If additional buses are added to the system, eventually the multibus/multiport organization results (fig. 2.16). The cost of this interconnection is quite large, due to the number of ports and buses. If some of the buses are dedicated to specific functions (fig. 2.17), the cost is reduced, at the expense of decreased reliability and flexibility.

As more buses are added to the system, eventually a point is reached where there is one bus for every module in the system. If the bus selection and arbitration logic is now removed from the modules, one has the crosspoint switch configuration (fig. 2.18). An m x n crosspoint connects m devices to n other devices such that any pair (m(i), n(j)) may communicate concurrently with any other pair (m(k), n(l)), k = i, l = j. No pair of devices (m(i),m(j)) or (n(i),n(j)) may communicate. For example, the m(i)'s may be processors, and the n(j)'s memories and I/O's. When a pair of modules (m(i),n(j)) wishes to communicate, the corresponding switch s(i,j) is closed.

The crosspoint switch is similar to the star in that the switching function is centralized, removing complexity from the functional modules. Unlike the star, the crosspoint allows communication between pairs of modules m and n only, rather than between any pair. This poses no difficulty when the modules can be easily identified as master or slave modules. For this reason, however, a generalized crosspoint system where each module is replaced by a bus, is infeasible. The crosspoint offers
highest concurrency, at the expense of great complexity. Furthermore, expansion is increasingly painful, since the number of switches (equal to \( mn \)) grows as \( n \)-squared.

The trunk system (fig. 2.19) is somewhat more flexible than the crosspoint. The system modules are configured on a number of separate buses, which communicate via one or more trunk buses. If the bandwidth provided by the trunks is inadequate, additional trunks may be added. The number of switch points is equal to \( n \times m \) where \( n \) is the number of buses, and \( m \) is the number of trunks. Maximum usable concurrency is achieved when \( m = n/2 \), so the complexity of the system grows, in the worst case, by \( n \times n/2 \). Often, less than the maximum number of trunks will be required, and each bus may not have to be connected to every trunk. This flexibility is a distinct advantage. One disadvantage is that, as in the case of multiple parallel buses, routing decisions must be made, either statically or dynamically; another, the transaction information must always be forwarded through two switches, rather than one in the case of the crosspoint.
(fig. 2.15) Parallel bus

(fig. 2.16) Multibus/ multiport [ENSL74]
(fig. 2.17) Multibus system with dedicated buses

(fig. 2.18) Cross-point switch
(fig. 2.19) Trunk system
2.2.4 Hierarchical forms

The configurations discussed previously may be combined in a hierarchical manner by considering each system to be composed of a number of subsystems, each subsystem as a system, and so on. The most common example of a hierarchical system is the tree, which may be viewed as a hierarchy of stars (fig. 2.20). The branching factor is the number of nodes (switches or modules) subordinate to each node. Symmetric trees have uniform branching factors, with all of the modules at the leaves, or lowest level, of the hierarchy. In practice, the organization of the tree is dictated by the application, and tends to be asymmetric. The assumption behind the tree organization, that the intensity of data communication diminishes as one goes up in the hierarchy, makes it suitable for hierarchical control problems.

Hierarchies of rings, trunks and crosspoints are other possibilities (e.g., PIER72), as are hierarchies comprising combinations of the basic configurations.

2.2.5 Other complex forms

Other complex interconnection forms may be constructed which are not hierarchical. These are generally referred to as graphs. The ARPA network is an example [ROBE70].
(fig. 2.20) Tree
2.2.6 Suitability of different structures to various computational modes

Not all of the interconnection forms discussed above are suitable for multiprocessing. The single time-shared bus, multiple-bus/multiport and crossbar arrangements are best suited [ENSL74]. The star, too, has possibilities, but only if the switching can be made fast enough to satisfy the bandwidth requirements of several processors and memories. The trunk and generalized star both offer two different access rates: one for requests local to a single bus, and another for requests which must be forwarded to a different bus. This two-tier situation violates the "equal access" requirement for all processor-memory pairs in a multiprocessor. However, very tight coupling between buses is achievable, making these structures suitable for distributed processing.

The ring exhibits greater delay and generally requires the use of data packets. Thus, process communication is supported, but direct data transfers between processors and memory at different nodes is not. This looser coupling makes the ring more suitable for computer networking, though distributed computing, at the task level, is feasible.

Although tree structures have been proposed for the interconnection of processors and memory, the increased delay of several switching centers makes them less suitable for this purpose. Trees are well-suited to multicomputer communication, especially where data reduction techniques can be used to limit the bandwidth requirements at higher levels. Another application for tree structures occurs in the I/O subsystems of large computers where it is desired to share a single channel among several device controllers, which in turn are shared by a number of devices (fig. 2.21).

Relatively low-speed star interconnections may be used for multicomputer interfacing. An example of such a star network was reported in [TOON77].
(fig. 2.21) Computer I/O structure (tree) [ENSL74]
2.3 Examples

Several examples of multiple-processor systems are described in this section.

2.3.1 DEC PDP-11

The PDP-11 was introduced in 1970, and has since become very popular [BELL76]. This acceptance is attributable, in large part, to the choice of a single time-shared bus (UNIBUS) architecture. The family of UNIBUS-compatible processors, memories, and I/O's covers a wide range of performance and cost.

The UNIBUS supports a single central processor module, which in addition to normal processing duties, arbitrates requests from other units which are capable of becoming bus masters. Two types of requests are allowed. A bus request (BR), if it is of sufficient priority, will be honored between processor instruction cycles; bus requests are normally used for interrupt processing. A non-processor request (NPR) will be honored whenever the CPU is not using the bus, i.e., possibly during instruction execution. In this fashion, device controllers may access memory independent of the CPU.

The integration of the bus arbiter with the CPU makes the addition of another CPU impossible, unless the processors are modified so that only one arbiter handles all requests. However, the system will accommodate additional processors if they are made to appear as device controllers to the arbiter. The GT-4x series of display processors is interfaced in this manner.

DEC offers a bus window for interfacing pairs of UNIBUSes [DECa73]. Circuit switching and address mapping are used. A variety of configurations may be obtained; however, the processor-per-bus limitation means a rapid proliferation of buses for systems of many processors.

A significant feature of the top-of-the-line PDP-11s is the addition of a second bus for communication with high-speed memory (fig. 2.22). The bus is usually shared among device controllers, allowing concurrent I/O; a second processor can be added to this bus, with proper arbitration logic [DECb74].
2.3.2 Lockheed SUE

The Lockheed Sue processor [LOCa72, LOCb72], introduced in 1972, uses a single bus like the PDP-11. Bus arbitration in SUE systems is performed by a separate bus controller, thus facilitating multiprocessing. Snap-shot arbitration (see chapt. 4) is employed for processor requests, resulting in equal sharing of the bus among processors. Up to eight processors may be configured on a single bus. Peripherals and DMA device requests are arbitrated in a fixed priority precedence scheme similar to the PDP-11. The SUE processor is inexpensive, micro-coded, and quite slow—all factors which favor multiprocessing.

2.3.3 BBN Pluribus

The Pluribus multiprocessor system [HEAR75, MANN76, ORNS75] was designed initially for an Interface Message Processor (IMP) for the ARPAnet (fig. 2.23). Processors, memory, and I/O modules are configured on separate buses which are interconnected via bus couplers. The organization may be viewed as a trunk system where each trunk connects two buses only, or as a graph.

2.3.4 Intel MDS

The Intel MDS-800 [INTa75] is an 8080 microprocessor-based computer with capabilities approaching those of bottom-of-the-line minicomputers (e.g., DECLAB PDP-11/03). A unique feature of the MDS system is that it supports the Intel In-Circuit Emulators for developing/debugging Intel microprocessor-based products. The overall architecture of the MDS system (fig. 2.24) follows the UNIBUS tradition. The CPU module maps the rather unfortunate 8080 bus into a standard asynchronous request/acknowledge protocol (again, see chapt. 4).

The bus arbiter is separate from the CPU and arbitrates up to eight requests in parallel, using a fixed priority scheme. Thus, multiprocessing is feasible, although processor lockout can occur. The ICE and the disk channel controller are special-purpose processors which share the bus with the CPU.
2.3.5 C.mmp (Carnegie-Mellon multi-mini-processor)

C.mmp is an experimental multiprocessor system developed at Carnegie-Mellon [COHE73, WULF72]. The system is configured around a 16 x 16 crosspoint switch. Sixteen PDP-11 computers (various models) and sixteen memory modules are used. The I/O devices are attached to the individual computers and access local memory in the normal UNIBUS manner. C.mmp development was motivated by several factors: interest in multiprocessor hardware and operating systems; the need for a large general-purpose computer facility; and interest in developing techniques for parallel programming.

2.3.6 Cm*

Cm* is a modular multi-microprocessor network being developed at Carnegie-Mellon [FULL76, SIEW77]. The basic structure of the system is shown in (fig. 2.25). Cm* consists of a number of computer modules (Cm) interfaced by bus switches (S.local) to a local bus, called the "Map" bus, which is controlled by a special processor, the K.map. The Cm's are modified LSI-11 processor-memory pairs; up to fourteen Cm's may be configured on a single Map bus with a K.map to comprise a cluster. The K.map's interface each cluster to two system-wide-buses.

Three levels of communication may be identified: processor-memory communication within a Cm; communication between Cm's in a cluster, using the Map bus; and communication between clusters via the K.map and inter-cluster buses. It is assumed that locality of reference will result in relatively few requests outside each Cm. Individual processors may, however, access the local memory of any other processor in the system, through the K.map. The K.map performs address mapping and transaction forwarding functions for up to eight requests concurrently. The operation of the K.map is transparent to the processors, except for the resultant increased delay, up to seven microseconds for transactions using the Map bus, considerably more if the inter-cluster buses are involved. Packet switching techniques are used at inter-cluster level.

This topology may be viewed as a hierarchy, or a trunk system. The hierarchy view derives from the three distinct levels of communication, and is diagrammed in (fig. 2.26a). The trunk conclusion is reached by regarding the Map bus as the focal point of the individual clusters, which
communicate via two inter-cluster trunks (fig. 2.26b).

2.3.7 PRIME

PRIME is a modular multiple-processor system developed at Berkeley for interactive processing [BASK72]. The principal goals of the project were high availability and security.

The PRIME topology (fig. 2.27) is very nearly that of a true multiprocessor. The multibus/multiport memory interconnection topology is used; each of five processors is connected via its own memory bus to eight of thirteen four-port memory modules, in an overlapping fashion. Thus, forty of the fifty-two available memory ports are used. The system may be expanded to eight processors and 16 memory modules; then every one of the sixteen memory ports will be used. Although each processor can access only half the memory modules, it is possible to guarantee that every pair of processors shares at least one module. I/O devices, including disk secondary storage, are shared equally among the processors. The External Access Network (EAN) allows any processor to communicate with any disk unit or any other processor. The EAN handles all I/O except terminal I/O. Each terminal is connected directly to two processors, for reliability reasons.

In operation, one processor is designated the control processor, and the remainder, problem processors. Under the supervision of the control processor, tasks are distributed to problem processors and the EAN is configured. Each subsystem has its own operating system, and handles its own I/O. Thus, the system functions as a group of logically separate subsystems, until changing load or a failure necessitates reconfiguration by the control processor. Normally, these subsystems do not intercommunicate, except through the control processor. In this way, users are physically separated, resulting in improved security and reliability.

2.3.8 MINERVA

MINERVA (fig. 2.28) is a shared bus multi-microprocessor system being developed at Stanford [WIDD75], for the purpose of studying bus structures and protocols, reduction of bus loading, and system partitioning for LSI packaging. As such, its goals are closely allied with those of this thesis.
Two types of processor modules (a 32-bit, i3000-based processor, and an i8080-based processor), and various memory and I/O modules share a single bus (IDBUS). Bus arbitration is provided by a simple request/grant mechanism using a centralized IDBUS arbiter. The arbiter supports fixed priority arbitration for I/O devices, and FIFO arbitration for processors, thus guaranteeing service to I/O devices, while preventing processor lockout.

The MINERVA project has produced several interesting techniques regarding reduction in bus bandwidth requirements, interrupt structure, and implementing synchronization primitives. These will be presented in later chapters.

2.3.9 DCS

DCS, developed at UC Irvine, is an example of a buffered ring computer network. The network was designed with the following goals [FARB72]:

- reliability;
- easy addition of new processing services;
- modest start-up cost; and
- low incremental expansion cost.

A typical DCS system is shown in (fig. 2.29). A number of processors are connected to devices called Ring Interfaces (RIs), which communicate via a single unidirectional digital channel. The combination of the RIs and the communication channel is called the "ring"; the processor and RI together comprise a "node" on the ring.

In operation, a number of fixed-length message slots circulate among the Ring Interfaces. An RI wishing to transmit a message first waits for an empty slot; when an empty slot is received, it is replaced by the transmitted message. The message contains several fields; principal among these are the destination processor name, the message portion, and the reply field. Every RI monitors incoming messages for messages addressed to processes that reside at that node. When a match occurs, the RI copies the message into a buffer, sets a bit in the reply field, and retransmits the message. When the originating node receives the message it sent, it checks the message for data integrity and to see if it was received properly. If there is an error, or if the reply field does not show that the message was
received, it is retransmitted. Otherwise, the message is removed from the ring.

The ring nodes are generally not homogeneous; in general, each offers some special capability such as an uncommon programming language or I/O device. The ring provides for sharing of these resources by all network users. Significantly, primary memory cannot be shared; any job too large for a single processor cannot be handled by the network. Thus, DCS cannot be considered a multiprocessor; instead, it is an example of a tightly coupled multicomputer system.

2.3.10 Ethernet

Ethernet is a packet broadcast computer network [METC75]. The shared communication facility is a so-called 'branching ether', actually a number of tapped coaxial lines, electrically connected by packet repeaters (fig. 2.30). Packets of various lengths (depending on system loading) are transmitted bit-serially, monitored by all stations, and copied by those destinations which are selected by address information contained in the packet.

The ethernet may be reduced to the logically equivalent structure shown in (fig. 2.31), revealing that it is basically a shared-bus architecture. Thus, the ethernet has two problems in common with shared-bus systems: contention and limited bandwidth.

Contention is handled in the following way: each station monitors the ether until it appears to be free (no carrier signal present); when the ether is not busy, any (every) station may begin transmitting. If two or more stations begin transmitting at nearly the same time (a real possibility), a packet collision is said to occur. If a transmitting station detects a collision, it momentarily jams the ether, forcing all other stations to abort. The transmission is re-tried after a dynamically chosen wait interval. Thus, access to the ether is probabilistic.

The bandwidth wasted by arbitration can be made small by increasing the packet size, thus decreasing the fraction of bandwidth allocated to arbitration. For packet sizes of 4k words, the Ethernet is approximately ninety-eight percent efficient. As the number of stations is increased, two phenomena are observed: lockout problems arise, due to the
probabilistic access; and available bandwidth becomes exhausted. The lockout problems are diminished by decreasing the packet size to allow for more arbitration events, but then more bandwidth is wasted.

A second solution to the bandwidth problem is the introduction of packet filters, which are selective repeaters; a packet is repeated only if its destination lies further along the particular ether segment. This technique can increase concurrency by partitioning the network. Locality of reference becomes a consideration, as does the manner in which the packets are forwarded. [METC75] proposes an unbuffered scheme which is equivalent to a circuit-switched bus coupler. The main drawback here is that a packet must acquire all of the ether segments between the source and destination stations to be successfully transmitted. In a reasonably busy system, the probability of all those segments' being available may be small.
(fig. 2.24) Intel MDS

(fig. 2.25) (fig. 26a) Cm* structure hierarchy
(fig. 2.26) Equivalent Cm* structure

(b) trunk

(fig. 2.27) PRIME
(fig. 2.28) MINERVA

(fig. 2.29) DCS
(fig. 2.30) Ethernet [METC75]

(fig. 2.31) Ethernet as shared bus
2.4 References

Computer system topology is covered very well in [ENSL74], where numerous examples of multiple-processor and parallel processor systems are also discussed. A good discussion of the characteristics of multicomputer, multiprocessor, and parallel systems is contained in [LORI71]. [INFa76] is also a good source for basic ideas. [SIGA76] is a transcript of a discussion by various computer heavy-weights concerning the true meaning of distributed processing. Alternate taxonomies are the subject of [FLYN72] and [JENS76].

Interconnection issues are treated in [CHEN74, THUR72, ENSL74, PIRT67].

A number of examples of multiple-processor systems are presented in [ENSL74]. Further examples are found in [MOSI68].
CHAPTER 3 A New Bus Structure -- Pended-transaction Bus

3.1 Motivation for PTB

The advantages of the single time-shared bus for multiple-processor interconnection were outlined in chapter two:

- low cost;
- universality; and
- flexibility.

Unfortunately, the limited bandwidth that the single bus offers is rapidly consumed as more processors are added to the bus. As we shall see in the next section, conventional processors and memories are especially offensive due to their unnecessarily high bus utilization.

Multiple processors can be configured on separate buses with memory and I/O modules, and these buses linked via bus couplers or I/O channels. Serial and parallel bus interconnections were discussed in chapter two. The parallel structures allow considerable concurrency at the expense of routing problems, while the serial forms suffer from a tendency toward replication of resources for fast access. Both schemes suffer from a proliferation of buses, even for systems with only a few processors. Clearly, the problems associated with multiple buses can be diminished by configuring multiple processors on a single bus, provided the bandwidth requirements of the processors can be reduced.

3.2 Two-part solution

In traditional bus schemes, much of the bus bandwidth is wasted by devices which are considerably slower than the bus itself. The grossest disparities in speed (>100:1) between processors and I/O devices have been alleviated using device interrupts, but bandwidth continues to be wasted whenever the bus is inherently faster than the devices using it. Consider the following examples.

- example 1 (fig. 3.1).

Most processors (especially microprocessors), once granted control of the bus, will continue to hold it for an integral number of processor clock periods. This is due to the synchronous nature of processor state
transitions, and means that up to one clock period may be wasted each time
the processor uses the bus, even if an asynchronous bus protocol is used.

- example 2 (fig. 3.2)

A memory or I/O module generally will not acknowledge a request until
it has completed its read or write cycle. For read cycles, the acknowledge
cannot be generated until the data is valid (the access time of the
memory); and for write cycles the acknowledge is generally delayed until
the cycle is complete, so that a new request will not overlap the previous
cycle. In both these cases, the bus is held busy; if it could be freed
during the waiting interval, available bus bandwidth would increase.

The point of the above examples is that conventional bus architectures
waste bandwidth by tying up the bus unnecessarily. This high bus
utilization is a more serious limitation than that imposed by the finite
bandwidth of the bus, but it is caused by inefficient bus protocols and can
be removed.

Before proceeding, some definitions are needed. A bus cycle is a
single use of the bus. Bus cycle time ($T_{bc}$), the time that the bus is
busy per bus use, is determined by a number of factors, including bus
protocol, processor selection time, and processor and memory speeds.
Available (or useful) bus bandwidth is the inverse of bus cycle time.
Processor cycle time ($T_{pc}$) is the time between successive bus uses by
the processor. Bus utilization ($b$) is the fraction of available bus cycles
required by a processor, and is equal to $T_{bc}/T_{pc}$. For most
processors, the processor cycle time and bus utilization are a function of
instruction mix. Typical benchmark bus utilization values for several
processors are given in (table 3.1).

We will now present a rather simple analysis which shows how bus
utilization affects throughput in a multiprocessor system. This analysis
is based on a method presented in [RAVI73].

Ideally, one would like system throughput to increase linearly with
the addition of processors. This cannot be the case, however, because each
new processor increases bus contention. When bus contention occurs, one or
more processors must wait longer than they would if they were the sole user
of the bus. Thus processor, and therefore system, throughput is degraded.
Let each processor in an N-processor system alternate between two states, Busy (B) and Waiting (W) (fig. 3.3). The processor is in the B state for the majority of its minor cycle ($T_{pc}$). When a bus reference is made, the processor enters the W state. When the resulting bus cycle is complete, the processor returns to the B state. Note that bus arbitration time is lumped with bus cycle time in this model.

Time will be expressed as a discrete variable, in bus cycle time ($T_{bc}$) units. Every $T_{bc}$ seconds one processor makes the transition from W to B, unless no processor is in W. Also, any processor may make the transition from B to W, with probably $b = \frac{T_{bc}}{T_{pc}}$.

Suppose that at instant $t_n$ there are $R$ processors in the B state, and one or more processors in the W state. At $t_{n+1}$ one processor will have made the transition from W to B, and up to $R$ processors will have moved from B to W. The probability that $S$ processors remain in B, $A(R,S)$, is given by the probability that $R-S+1$ processors move from B to W, by the binomial distribution:

$$A(R,S) = \binom{R}{R-S+1} b^{R-S+1} (1-b)^{S-1} ; \quad 0 \leq R < N, \; 0 < S \leq R+1.$$

Note that $R$ is constrained to be less than $N$, ensuring that at least one processor is in state W. $S$ must be greater than 0, because one processor will always enter state B, regardless of how many leave. Also, $S$ must be no greater than $R+1$; this is the case where no processor leaves state B.

An exception occurs whenever $R=N$. In these cases, no processor is in W, making the probability that $S$ processors remain equal to the probability that $N-S$ processors move from B to W:

$$A(N,S) = \binom{N}{N-S} b^{N-S} (1-b)^S ; \quad 0 \leq S \leq N.$$

At any point in time, the state of the system is given by the number of processors in the B state. The $A(i,j)$ are state transition probabilities, and can be used to compute the steady-state probability that $R$ processors are in the B state as
\[ p_R = p_{R-1}A(R-1,R) + p_RA(R,R) + p_{R+1}A(R+1,R) + \ldots + p_NA(N,R) \]

\[ = \sum_{i=R}^{N} p_iA(i,R) \; ; \; 0 \leq R \leq N, \; 0 \leq i. \]  

(3-3)

For example, consider \( N=3 \). The matrix of the \( A(i,j) \) is

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & b & 1-b & 0 \\
0 & b^2 & 2b(1-b) & (1-b)^2 \\
b^3 & 3b^2(1-b) & 3b(1-b)^2 & (1-b)^3
\end{bmatrix}
\]

A check reveals that the row sums, corresponding to the total probability of leaving state \( i \), are all 1.

Also, we may write:

\[
P_3 = P_2 \ast A(2,3) + P_3 \ast A(3,3) \\
P_2 = P_1 \ast A(1,2) + P_2 \ast A(2,2) + P_3 \ast A(3,2) \\
P_1 = P_0 \ast A(0,1) + P_1 \ast A(1,1) + P_2 \ast A(2,1) + P_3 \ast A(3,1) \\
P_0 = P_0 \ast A(0,0) + P_1 \ast A(1,0) + P_2 \ast A(2,0) + P_3 \ast A(3,0) \\
P_0 + P_1 + P_2 + P_3 = 1
\]

Let \( b = 0.70 \), a typical value (see table 3.1); then

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0.700 & 0.300 & 0 \\
0 & 0.490 & 0.420 & 0.090 \\
0.343 & 0.441 & 0.189 & 0.027
\end{bmatrix}
\]

The solution for the \( P_j \)'s gives

\[
P_3 = 0.031 \\
P_2 = 0.333 \\
P_1 = 0.625 \\
P_0 = 0.011
\]
The effective system throughput is calculated as

\[
Tp_e = \sum_{j=1}^{N} \sum_{i=j-1}^{N} P_i A(i,j); \quad i>0.
\] (3-4)

\[
Tp_e = 1(0.625) + 2(0.333) + 3(0.031) = 1.38
\]

In this case the total number of processors is equal to three, thus, bus contention results in marginal return for systems composed of more than one processor.

Now let \( b = 0.1 \), then

\[
A = \begin{bmatrix}
0 & 0.1 & 0.9 & 0 \\
0 & 0.01 & 0.18 & 0.810 \\
0.001 & 0.027 & 0.243 & 0.729
\end{bmatrix}
\]

\[
P3 = 0.729 \\
P2 = 0.244 \\
P1 = 0.026 \\
P0 = 0.0007
\]

\[
Tp_e = 1(0.026) + 2(0.244) + 3(0.729) = 2.70
\]

By a similar analysis, Reyling has shown [REYL74] that with \( b = 0.1 \), up to ten processors may be configured on a single bus, with performance degradation in the neighborhood of twenty percent, on the average.

The analysis shows that reducing bus utilization can give significant increases in system throughput. Actually, the improvement may be greater than that indicated. This is because for small values of \( b \) (less than \( 1/N \)), the processors will tend to synchronize so that the period between bus contention events may last several processor cycles. The synchronization would be perfect if every processor cycle were the same length; unfortunately, most processors vary the length of the minor cycle depending on the instruction being executed. The i8080, for example, may
have three, four, or five states (clock periods) per machine cycle.

Recall that bus utilization is the ratio of bus cycle time to processor cycle time. Bus utilization can be reduced proportionally to a decrease in bus cycle time.

3.2.1 Pended transactions

The pended transaction protocol reduces bus utilization by freeing the bus during the asynchronous wait interval of the transaction (fig. 3.4). Each transaction involves two uses of the bus: one to start the transaction, the other to complete it; in the interim, other transactions may be begun or completed. Because the bus cycle time is no longer dependent on processor speed or memory cycle time, it may be reduced as far as bus technology and the protocol design allow, maximizing available bus bandwidth. If asynchronous logic is used, cycle times on the order of 150 ns may be achieved. The communication overhead to implement pended transactions is twice that for non-pended transactions, but bus utilization is nevertheless greatly reduced whenever the wait period is significant.

3.2.2 Multiple-bus systems

Ultimately, even a pended-transaction bus system will saturate when the available bus bandwidth is exhausted. If a system can be partitioned so that relatively little communication takes place between subsystems, then a multiple-bus system is a very good solution; regardless, it may be the only approach when the available bus bandwidth has been exhausted.

The pended transaction protocol lends itself very readily to message-switched transaction forwarding. In this case the unit message is either a master module request or a slave module acknowledge; the traditional two-way communication (request-acknowledge) is broken into two separate communications. These are forwarded between buses by bus couplers. The bus coupler recognizes any request to an off-bus address, pends the requesting device, and forwards the request. Later, when the acknowledge is received, it is forwarded back to the requesting device. As shown in (fig. 3.5), transaction information may pass through several bus couplers between the master and slave modules.
**Processor bus utilizations for various benchmark programs [GERS76]**

<table>
<thead>
<tr>
<th></th>
<th>8080*</th>
<th>8080**</th>
<th>6800</th>
<th>6502</th>
<th>9900</th>
<th>LSI-11</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD8</strong></td>
<td>82%</td>
<td>58%</td>
<td>92%</td>
<td>93%</td>
<td>55%</td>
<td>47%</td>
</tr>
<tr>
<td><strong>SUB8</strong></td>
<td>88%</td>
<td>58%</td>
<td>92%</td>
<td>93%</td>
<td>55%</td>
<td>47%</td>
</tr>
<tr>
<td><strong>MULT8</strong></td>
<td>74%</td>
<td>50%</td>
<td>71%</td>
<td>80%</td>
<td>36%</td>
<td>20%</td>
</tr>
<tr>
<td><strong>DIV8</strong></td>
<td>74%</td>
<td>50%</td>
<td>75%</td>
<td>86%</td>
<td>39%</td>
<td>21%</td>
</tr>
<tr>
<td><strong>ADD16</strong></td>
<td>82%</td>
<td>55%</td>
<td>92%</td>
<td>84%</td>
<td>55%</td>
<td>50%</td>
</tr>
<tr>
<td><strong>SUB16</strong></td>
<td>88%</td>
<td>58%</td>
<td>92%</td>
<td>84%</td>
<td>55%</td>
<td>50%</td>
</tr>
<tr>
<td><strong>MULT16</strong></td>
<td>74%</td>
<td>50%</td>
<td>75%</td>
<td>86%</td>
<td>39%</td>
<td>21%</td>
</tr>
<tr>
<td><strong>DIV16</strong></td>
<td>79%</td>
<td>53%</td>
<td>75%</td>
<td>83%</td>
<td>34%</td>
<td>20%</td>
</tr>
<tr>
<td><strong>ADD32</strong></td>
<td>81%</td>
<td>54%</td>
<td>92%</td>
<td>84%</td>
<td>48%</td>
<td>47%</td>
</tr>
<tr>
<td><strong>SUB32</strong></td>
<td>88%</td>
<td>58%</td>
<td>92%</td>
<td>84%</td>
<td>48%</td>
<td>47%</td>
</tr>
<tr>
<td><strong>MULT32</strong></td>
<td>87%</td>
<td>58%</td>
<td>82%</td>
<td>72%</td>
<td>37%</td>
<td>32%</td>
</tr>
<tr>
<td><strong>DIV32</strong></td>
<td>77%</td>
<td>52%</td>
<td>70%</td>
<td>77%</td>
<td>44%</td>
<td>37%</td>
</tr>
<tr>
<td><strong>SORT8</strong></td>
<td>77%</td>
<td>50%</td>
<td>57%</td>
<td>82%</td>
<td>45%</td>
<td>45%</td>
</tr>
<tr>
<td><strong>SORT16</strong></td>
<td>77%</td>
<td>52%</td>
<td>52%</td>
<td>80%</td>
<td>44%</td>
<td>44%</td>
</tr>
<tr>
<td><strong>INTERRUPT</strong></td>
<td>85%</td>
<td>57%</td>
<td>82%</td>
<td>60%</td>
<td>81%</td>
<td>46%</td>
</tr>
</tbody>
</table>

* 8080 uses bus during T1 (SYNCH) time.

** 8080 does not use bus during T1 (SYNCH) time.
(fig. 3.1) Synchronous wait period

(fig. 3.2) Typical read or write cycle

\[ b = \frac{T_{bc}}{T_{pc}} \]
(bus cycle)

Processor state model

(fig. 3.3)

\[ b = \frac{2T_{bc}}{T_{pc}} \]

(fig. 3.4) Pended transaction
(fig. 3.5) Use of bus coupler
CHAPTER 4 Design of a Single Bus

4.1 Bus synchronization protocols

Communication between modules on a bus involves a series of data transactions. The coordination of these transfers is critical, since address, data, and control information are constantly changing, as different pairs of modules use the bus.

The needed coordination is supplied by a synchronization protocol, which may be implemented with a set of synchronization primitives. Typical synchronization primitives are sender ready (SR), receiver ready (RR), data ready (DR), and data accepted (DA), for example [CHEN74]. The signalling of these primitives occurs at times which are important in the data transaction. Different protocols use various combinations of these synchronization primitives, and acknowledgements to them. [CHEN74] contains an exhaustive description of all such protocols.

Unfortunately, a discussion of bus synchronization can be clouded by the question of point of view. Every transaction involves a data source (sender) and a data sink (receiver). The transaction may be initiated by either the sender or the receiver. The initiator of the transaction is called the bus master; the addressed module is called the bus slave. If the bus master is also the sender of the data, the transaction is called a write operation; otherwise it is a read operation. We will adopt this convention of describing the transaction from the master's point of view.

One class of bus protocols, called "synchronous" protocols, uses no synchronization primitives at all. Instead, timing information is derived locally, either from a clock signal which is broadcast from a central point, or from the data itself. By contrast, asynchronous protocols rely on the transmission of synchronization information with the data. This information is not local to the receiver and therefore must be communicated to it.

Actually, the distinction between synchronous and asynchronous protocols is not so clear cut; a spectrum exists. It is helpful to consider relative synchrony in terms of the amount of synchronization information which must be provided. A purely synchronous bus system would then derive all its timing information from the elapsed time since system
initialization. This arrangement is practical for certain dedicated-function, cyclic-task systems.

A less synchronous, but more common, example of a synchronous bus protocol is illustrated in (fig. 4.1). This system uses a global clock to generate fixed equal-width time slots. Modules may be assigned to the time slots on either a dedicated or demand basis. No transaction synchronization is needed; address, data and control information are presented at the beginning of the cycle, and by agreement, the data transfer is complete before the end. This scheme has the advantage of very high speed, due to the complete lack of communication overhead. However, it requires that the response times of all units using the bus be identical, or much bandwidth may be wasted.

An asynchronous protocol which still exhibits a fair amount of synchrony is shown in (fig. 4.2). A single synchronization signal (SYNC) indicates the presence of valid address and data (if any) information. The transaction type, read or write, I/O or memory, is communicated on separate control lines. It is assumed that the addressed module will have complied with the request before the SYNC signal is removed. This protocol is really the same as the synchronous protocol of (fig. 4.1), except that the time frames are now defined by the SYNC signal.

A variation of the protocol of (fig. 4.2), called the one-way command protocol, is shown in (fig. 4.3). The timing of a read operation is unchanged, but for a write cycle, the STB (strobe) signal is delayed to ensure that the address and write data are valid. This protocol is used by many minicomputers and most microprocessors. Its principal advantages are high speed, and low cost. The drawback is that, like true synchronous schemes, it makes assumptions about the response times of the other modules on the bus.

The one-way command protocol may be improved, though rather clumsily, by allowing the slave to signal "not ready", thereby invoking an asynchronous wait period. An example is the i8080 microprocessor bus protocol shown in (fig. 4.4) (note that this is the protocol used by the (8080-8224-8228) chipset, not the SBC or MDS computer modules). The read operation is signalled by the falling edge of MEMR/ (memory read, active-low). If the slave cannot provide the read data before the end of the following clock period (typically, 750 ns), it signals "not ready" by
pulling the READY line low. When READY is released, the read cycle may continue. No provision is made for an asynchronous write operation, since the state of the READY line is sampled before MEMW/ is sent. This limitation can be overcome by using additional logic to produce a signal ADVMEMW/, which announces that a write cycle is in progress, in sufficient time for the slave to indicate "not ready". In the terminology of [CHEN74], the 8080 read cycle consists of this synchronization primitive sequence: Receiver Ready (MEMR/), Sender Ready (READY). The Data Ready and Data Accept primitives are not used. The second write cycle corresponds to Sender Ready (ADVMEMW/), Receiver Ready (READY), Data Ready (MEMW); Data Accept is not used.

The signals MEMW/, MEMR/, etc., are not true synchronization primitives, because they convey information about the type of transaction (control information). When signals carry combined timing and control information, they are often called data primitives.

A truly asynchronous protocol requires the use of at least two synchronization primitives: one to indicate the presence of transaction information on the bus, the other to signal the completion of the specified operation. In this way, the length of the transaction is controlled by the devices using the bus. An asynchronous protocol of this type is illustrated in (fig. 4.5), for a read operation. The signal RFD (ready for data) indicates the presence of a valid read address on the bus; the data is transferred when the addressed slave responds with DAV (data available). In this protocol, the primitives Sender Ready and Data Accept are not used.
(fig. 4.1) Synchronous protocol

(fig. 4.2) Asynchronous protocol using single SYNC signal
(fig. 4.3) One-way command protocol

(fig. 4.4) 8080 read and write cycles
Asynchronous protocol

(fig. 4.5) Asynchronous protocol
4.1.1 Use of acknowledge and interlock

Synchronization pulses may sometimes be too short to be reliably received at the destination module. The use of acknowledge and interlock techniques diminishes this problem.

The receipt of a synchronization signal may be acknowledged with a separate acknowledge signal, as shown in (fig. 4.6). More frequently, the acknowledge is implied by the next synchronization signal from the target device. This use of combined acknowledge is possible only if the protocol allows the master and slave to alternate their transmission of synchronization signals; even so, separate acknowledge signals are employed when a signal must be acknowledged before the delay which is generally present between synchronization signals. The use of acknowledge allows the detection of missed synchronization primitives.

The so-called request/acknowledge protocol (fig. 4.7) is an example of combined acknowledge. The signal ACK both acknowledges receipt of REQ and signals that the requested operation has been performed.

The use of interlock further improves reliability by maintaining the asserted level of a synchronization primitive until its acknowledge is received. A protocol is said to be half-interlocked if synchronization primitives, but not their acknowledgements, are interlocked. A fully-interlocked protocol uses interlock on both primitives and their acknowledgements. With a fully-interlocked protocol, every transition of a synchronization signal or an acknowledge is itself acknowledged. These relationships are indicated by "causality" arrows for the half-interlocked and fully-interlocked versions of the request/acknowledge protocol (fig. 4.8a, fig. 4.8b).
RFD = ready for data
DAV = data available

(fig. 4.6) Separate acknowledge

(fig. 4.7) Asynchronous request/acknowledge protocol
(fig. 4.8) Request/acknowledge protocol
(a) half-interlocked
(b) fully-interlocked
4.2 Bus arbitration

At a given moment, a number of potential bus users may require the bus. Since the bus can carry only one message at a time, it must be allocated to only one of these users. The process of bus allocation is called arbitration, and is implemented with a bus arbiter.

4.2.1 Arbitration algorithms

Bus arbitration may be viewed as a matter of assigning the bus to the highest-priority requesting unit. The arbitration algorithm is the process by which priorities are associated with requests.

A number of potential inputs to the arbitration algorithm may be identified. For example:

- time of day;
- elapsed time since last bus use by this device;
- a static priority permanently associated with the device;
- a bid for the bus submitted by the device;
- the slave unit for the message to be sent;
- etc.

These inputs may be communicated to the bus arbiter via special bus lines, or may be statistics maintained by the bus arbiter (e.g., elapsed time since last bus use).

The factors which should be considered in the selection of an algorithm and its implementation, are [SEAR75]

- simplicity;
- speed;
- fairness;
- expandability; and
- reliability.

SIMPLICITY is a goal mainly for cost and reliability reasons. A primary concern is the number of bus lines that must be allocated to the arbitration function;

SPEED, because the arbitration decision must be made between bus uses, or at best, concurrently with bus use;
FAIRNESS, so that no device fails to meet its real-time constraints, or is locked-out from using the bus for a long time.

EXPANDABILITY. It may be necessary to add devices to the bus; it should be possible to do so without requiring additional bus lines, and without significantly degrading the performance of the arbiter.

RELIABILITY may be a problem, since arbitration functions tend to be centralized.

4.2.1.1 Fixed priority algorithm

The fixed priority algorithm is conceptually simple, and easily implemented; hence it is very popular. A priority level is permanently associated with each possible request. Thus the arbitration algorithm is performed at the time the system is configured; each bus arbitration consists merely of selecting the highest-priority request.

Fixed priority arbitration works well when a clear ordering of priorities is discernible. For example, a disk channel almost certainly will have priority over a processor. Often, devices of equal priority (e.g., anonymous processors in a multiprocessor system) must share the bus. In this case, the imposition of a static priority scheme is unfair, especially if the system is heavily loaded. A device which is arbitrarily assigned a low priority will not enjoy equal access to the bus. An example of processor lock-out resulting from fixed-priority arbitration was reported in [JORD74].

4.2.1.2 Round robin and snapshot algorithms

The round robin algorithm provides equal access to all bus users. A circular ordering is imposed on all units which are potential bus users. Each time a unit releases the bus, control passes to the next requesting unit (fig. 4.9). In this way, the priority assignments rotate among bus users, with the current user always receiving lowest priority, the unit to its right, highest priority, and so on.

The snapshot algorithm is a modification of the fixed priority scheme which provides near-equal bus access. A "snapshot" is taken of existing bus requests and then these requests are served in fixed-priority order. When no requests remain, another snapshot is taken, and the process is
repeated.

One disadvantage of the round robin and snapshot algorithms is that they may fail to meet the real-time requirements of high-speed devices. For this reason, a two-level structure is sometimes used, where level A devices always take priority over those of level B, but within a level, priority rotates. Devices with real-time requirements may then be assigned to level A, and processors to level B.

4.2.2 Centralized and decentralized arbitration

The location of the arbiter hardware is a major consideration in the design of a bus arbiter. Centralized methods use one unit (the arbiter) to decide which of the requesting units to allocate the bus to. Decentralized methods distribute the arbitration hardware among all units; each unit decides for itself whether it has been allocated the bus. Centralized schemes generally involve less total hardware, but may require extra bus lines to communicate the inputs to, and outputs from, the arbitration algorithm. They may also pose a reliability problem, since failure of the arbiter will be catastrophic. The major problem of decentralized schemes is that their operation must be synchronized so that there will be agreement on which unit is selected.

4.2.3 An example

The PDP-11 arbiter (fig. 4.10) incorporates several of the commonly used arbitration schemes. Interrupting devices require the use of the bus to signal the interrupt and supply the interrupt vector, which identifies the source of the interrupt; these requests are entered on one of four levels BR[7:4]. Requests entered on the BR lines are assumed to be for interrupts, and are only honored between processor instruction cycles. Corresponding to the four BR lines are BG[7:4], which signal the selection of the particular request level for bus use. DMA requests use another pair of lines (NPR, NPG). A non-processor request (NPR) is honored whenever the CPU is not using the bus, and has priority over all other requests. The priority of the CPU is variable under software control. When the processor is interrupted, it assumes the priority level of the interrupting device, effectively preventing further interrupts on the same or lower level. When interrupt processing is complete, the CPU priority level is restored. When
it is not processing interrupts, the CPU remains at one of priority levels [3:0]; these levels are used for processor self-interrupts.

Requests which occur on the same priority level are resolved through a technique variously called daisy chaining, priority chaining, and precedence chaining. The bus grant signal is propagated through each unit on the level until a requesting unit is found; this unit does not propagate bus grant, and is selected for bus use. Within a request level, priority is assigned by position—the closer the unit to the arbiter, the higher the priority.

The combination of priority levels with precedence chaining produces a two-dimensional priority array, where the position of the unit within the array determines its priority.

4.2.4 Arbitration synchronization—the arbiter problem

The arbiter problem is fundamental to the design of computer systems [PATI74]. It arises whenever it must be determined which of two unrelated (and therefore asynchronous) signals occurs first. The classic solution to the problem, the arbiter latch, is shown in (fig. 4.11a). Here, if REQA is received before REQB, then GRTA-L is asserted, locking out REQB. When REQA is removed, the arbiter latch changes state, and GRTB-L is asserted. The problem occurs in the extremely unlikely event that REQA and REQB arrive nearly simultaneously, that is, within a "conflict window".

In this case, it is possible that the arbiter latch will achieve a metastable condition where both outputs remain in the dead-band region between a logic zero and a logic one. Worse, the outputs may oscillate in phase, crossing the zero and one thresholds several times before finally diverging to separate values (fig. 4.12). The metastable condition, once achieved, may persist for a long time relative to the conflict window.

When conflict arises, we obviously do not care whether A or B is finally chosen. On the other hand, we must impose some constraint. If the arbiter latch is part of a clocked-logic system (e.g., is contained in a synchronizing flip-flop), then we must insist that its outputs be stable before the occurrence of the next clock pulse. If it is part of an asynchronous system, a more strict constraint is imposed: no spurious output must appear on either output. If such an output were to appear, the
logic would react to it immediately, generating an error.

Fortunately, Patil has shown that spurious outputs from the metastable state can be suppressed by using threshold devices on the latch output (fig. 4.11b). If the excursions are limited so that they do not cross the threshold except when the metastable condition ends, then the effect of conflict is only to delay the arbiter decision. For synchronous systems, the delay may still be too great, producing a conflict between the arbiter outputs and the system clock; however, asynchronous systems can tolerate an occasional long delay with no problem.

As system speeds increase, the interval between arbitration events becomes smaller, increasing the probability of conflict. This is especially the case when a number of asynchronous requests must be handled by a bus arbiter; hence, reliable system design must deal with the arbiter problem.
Round robin arbitration

PDP-11 arbiter
(fig. 4.11) Arbiter latch (a) Basic arbiter latch (b) Basic arbiter latch augmented to suppress metastable condition

(fig. 4.12) Metastable condition at Q and Q̅ outputs of 7410 R-S flip-flop [CHAN73]
4.3 Pended-transaction bus

This section presents a detailed description of the pended-transaction bus, synchronization protocol, and arbitration scheme.

4.3.1 Signals Description

A number of signals are defined for the bus, though not all are used in the present implementation (table 4.1).

SYSTEM signals carry information that is vital to all modules, i.e., initialization (INIT), power failure detection (PF) and memory protect (MP). The INIT signal is generated by the front-panel module in response to power-on. PF and MP are not presently used, but could be generated by the power supply in the event of a power failure. MP signals that power is going down and the memory modules should not respond to further (possibly random) processor requests. It is assumed that battery backup power will be supplied to the memory modules during the power loss. When power returns, the PF signal alerts the system to the fact that a power loss occurred.

TRANSACTION CONTROL signals implement the pended-transaction protocol. Included are synchronization signals (DP, RES, NPA, PA, DB) and transaction-type signals (R/W, IO/M, N/S, SAD). The synchronization signals are described in the next section. The transaction-type signals specify read or write, I/O or memory, normal or special, and short address. The N/S signal is reserved for transaction types which may involve special processing by the slave, e.g., for a read operation S could be used to specify that the memory module should lock following the read; for a write operation, S could specify special handling, such as distinguishing byte from sixteen bit operations. The N/S signal is not used in the present implementation. The SAD signal indicates the presence of a short (16 bit) address rather than the full twenty-four bit address. It is used to eliminate the need for decoding logic on the upper eight address bits for those devices which respond to sixteen bit addresses only. NPA, PA, and DB are acknowledge signals from the slave which indicate that the request has been accepted and processed (NPA), the request is accepted but the transaction should be pended during processing (PA), or the request is rejected (DB).
BUS ARBITRATION signals are used by the bus arbiter. EB and SA signal that the bus is busy, and the next bus user has been chosen, respectively. SSYN1 and SSYN2 are used in a two-line interlocked protocol to synchronize operation of the distributed bus arbiter (DBA) units. BRQ indicates that at least one unit is requesting service. BR[7:0] are the individual bus request lines, and BPRI, BPRO are used to implement precedence chaining between adjacent devices.

INTERRUPTS. Provision for eight interrupt levels is made. The present implementation does not make use of them. (See section 6.2.3 for a discussing of interrupts.)

ADDRESS and DATA. Each module using the bus is assigned a unique device ID. The DIDs are communicated between modules on lines DID[3:0]. Twenty-four address lines and sixteen data lines are provided, though only sixteen and eight, respectively, are used.
SYSTEM (3)

<table>
<thead>
<tr>
<th>INIT</th>
<th>initialize</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF</td>
<td>power fail</td>
</tr>
<tr>
<td>MP</td>
<td>memory protect</td>
</tr>
</tbody>
</table>

TRANSACTION CONTROL (9)

<table>
<thead>
<tr>
<th>DP</th>
<th>data primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>transaction type -- read or write</td>
</tr>
<tr>
<td>IO/M</td>
<td>transaction type -- I/O or Memory</td>
</tr>
<tr>
<td>N/S</td>
<td>transaction type -- normal or special</td>
</tr>
<tr>
<td>SAD</td>
<td>transaction type -- short address</td>
</tr>
<tr>
<td>RES</td>
<td>resume</td>
</tr>
<tr>
<td>NPA</td>
<td>non-pend acknowledge</td>
</tr>
<tr>
<td>PA</td>
<td>pend acknowledge</td>
</tr>
<tr>
<td>DB</td>
<td>device busy</td>
</tr>
</tbody>
</table>

BUS ARBITRATION (15)

| BB  | bus busy |
| SA  | selection acknowledge |
| SSYN1 | selection sync 1 |
| SSYN2 | selection sync 2 |
| BR  | bus request |
| BR[7:0] | individual bus requests |
| BPRI | bus priority in |
| BPRO | bus priority out |

INTERRUPTS (8)

| INT[7:0] | interrupt levels |

ADDRESS & DATA (44)

| DID[3:0] | device ID |
| ADR[23:0] | address |
| DAT[15:0] | data |

| total | 79 |

TABLE 4.1 PTB Signals
4.3.2 Pended-transaction protocol

The pended-transaction protocol will now be described with reference to timing and state transition diagrams. Consider first the simplified state diagram for a master unit (a processor, for example) which was first presented in Chapter 3 (fig. 4.13a). Here the processor alternates between two states. It is BUSY when it is executing an instruction, but it must WAIT for each memory reference. A better state model divides the WAIT state into two sub-states (fig. 4.13b). In BR (bus request), the processor has requested, but not yet gained, access to the bus. When permission to use the bus is given (BG, bus grant), the SR (service request) state is entered. In SR, the processor presents its request, and waits for a response. When the read or write cycle is complete, the slave module responds with ACKnowledge, the master internalizes the read data (if any), releases the bus, and returns to the BUSY state.

The corresponding state diagram for the slave, shown in (fig. 4.13c), also has three states: READY, BUSY and ACKnowledge. The slave is normally READY. When a REQuest is received, the BUSY state is entered. When the cycle is complete, the ACK state is entered, causing the ACKnowledge output. When REQ is removed, the slave returns to the READY state. Note that the slave is never busy when the bus is not also busy; the bus access mechanism prevents access to busy slaves.

The timing diagram corresponding to the state diagrams just described is shown in (fig. 4.14a). Note from the "causality" arrows that the falling edge of ACK is not interlocked, giving rise to the possibility of overlap. This situation can be rectified by adding a state to the bus master state diagram (fig. 4.13d) which remains active until ACK is removed. A simple relabelling of signals results in the non-pended-transaction protocol shown in (fig. 4.14b). In the following description of pended transactions, we will not be concerned with the additional states needed to implement interlock. The interlocking is indicated on the timing diagrams, but for simplicity, the state diagrams contain only major states.

For pended transactions it is desired to split the request and acknowledge phases of the transaction into two separate transactions. The simplified state diagrams and resulting protocol are shown in (fig. 4.15, fig. 4.16). The master module enters BR, receives BG, and enters SR1 in
the usual way. (Actually, the process of acquiring the bus is a bit more complex; see the next section.) Now, however, the slave may respond to the request in one of three different ways. First, the slave may be BUSY working on another request. In this case, it signals DB (device busy), and the master enters the WA1 state. After a timeout, BR is re-entered, and the request is tried again. Second, the slave may be able to comply with the request immediately. This may be the case, for example, for a status register read. In this case NPA (non-pend acknowledge) is signalled, and the transaction completes in the usual way. Third, the slave may respond with PA (pend acknowledge), causing the transaction to be pended.

If a pended-transaction is called for, the slave first latches the master's device ID, address, and data, and then signals PA, causing the master to remove its request. The master releases the bus, and the transaction as well as the master unit are said to be pended; the slave is pending. When the slave cycle is complete, the slave requests the bus, and eventually becomes bus master. Thus we have a master-slave reversal. The slave-now-master drives the device ID of the transaction initiator onto the bus along with read data, if any, and the synchronization signal RES. The erstwhile master recognizes its DID, and enters SR2, storing the data present on the data bus in its own registers. In SR2, DP is again asserted, this time as a response to RES. When RES is removed, the transaction is complete.

The reader will recognize the PTB protocol as a fully-interlocked asynchronous protocol. In a fully-interlocked protocol, every signal transition depends on a previous transition of another signal; the first signal is said to "cause" the second. The result is minimum delay, since each state transition depends only on another state transition, not on some delay parameter which must be adjusted to a worst-case specification. Although the PTB protocol has significantly greater overhead than simpler protocols, it is adaptive in the sense that it will run reliably as fast as the bus allows. Another protocol would need to be detuned to achieve the same reliability, and the result could be slower operation.

It was desired to avoid using clocked logic in the PTB design for several reasons: (1) the required high-speed clock would be difficult to distribute without clock skew; (2) synchronous operation would mean a delay of one clock period for every "causality" arrow in the timing diagram; (3)
the clock period would have to be adjusted for the worst-case value of the delay. Again, even though a simpler protocol is possible, it would likely be slower.
(fig. 4.13) (a) master BUSY-WAIT state model
(b) corresponding slave state model
(c) improved master state model
(d) corresponding slave state model
(e) master state model with interlock
state (D)
REQ-ACK (a) Request/acknowledge protocol corresponding to state diagrams of (fig. 4.13c and 4.13d)
(b) non-pended-transaction protocol

(fig. 4.14)
(fig. 4.15) Simplified state diagrams for PTB protocol
(a) Master state diagram
(b) Slave state diagram
(fig. 4.16) Pended-transaction protocol
4.3.3 Arbitration

The design of a bus arbiter for the PTB is relatively independent of the PTB protocol. The protocol itself requires only speed from the arbiter design; in addition, it was desired to satisfy the "fairness" criterion for both processors and channels. For reasons of reliability (and research interest!), a distributed arbiter was chosen, but the bus itself was specified to support both distributed and centralized arbitration schemes.

To provide the flexibility required to make the distributed and centralized arbiters interchangeable, the bus arbiter module was conceptually "black-boxed" as shown in (fig. 4.17). The bus acquisition procedure begins with a request to the arbiter (BCR). When the arbiter has determined that the requesting module is the next user of the bus, it signals bus grant (BCG). The requesting module asserts SA (selection acknowledge), and waits for the bus to become free. When BB is de-asserted, the selected module drops its request and SA, asserts BB, and begins the transaction. As soon as SA is de-asserted, a new arbitration cycle may begin; thus, bus arbitration is overlapped with bus use, ensuring that the bus will not be idle for long between users.

For the distributed arbiter (DBA), the internal signal BCR is communicated to the bus arbiter logic on the requesting module. The arbiter logic forwards the request onto BRQ and one of BR[7:0] at the appropriate time. The DBA uses SA, SSYN1, and SSYN2 to synchronize the near-simultaneous inspection of BR[7:0] by all DBA units, and to guarantee their stability during the inspection period. Following the inspection of the BR_i, each arbiter decides for itself whether the associated unit has been allocated the bus; if it has, the signal BCG is produced, and the bus acquisition protocol proceeds as described in the last paragraph.

For the centralized arbiter (CBA) the use of the bus signals is slightly different. First, the signal BCR is connected directly to one of the BR_i, and communicated via the bus to the CBA. The CBA uses SA (but not SSYN1 or SSYN2) to synchronize BR_i inspection, and returns BG to the requesting device. The BG signals are connected in a radial fashion from the CBA to the BPRI signals of each module. Internal to the module, BPRI becomes BCG, and bus arbitration proceeds as before.

The BPRI, BPRO signals are used by both arbiter schemes to increase
the effective number of requests to sixteen. Paired modules both request on the same level; the higher priority module, if it does not require the bus, propagates the BCG signal to the other module via BPRO – BPRI, in a precedence-chain fashion. If desired, more than two modules may share the bus request level by extending the precedence chain. The increased delay will slow the arbitration process. Also, lockout may become a problem as more devices are added, since the precedence chain enforces a static priority ordering between modules. For a small number of modules, bus usage patterns can be relied upon to prevent lockout.

A detailed design for the DBA is covered in chapter six. The DBA combines the fixed priority and snapshot algorithms, allowing each device the (strappable) option of which to participate in. By assigning all the fixed priority devices higher priority levels than any of the snapshot devices, a true two-tier arbiter may be obtained.
(fig. 4.17) Bus arbiter black box
4.4 Issues specific to the pended transaction bus

There are two potential problems which arise in the pended-transaction bus design. Both are due to the fact that the bus is released during the transaction.

4.4.1 Device-busy problem

Because bus slaves pend transactions when they will be busy (pending) for an extended period, subsequent requests to the same slave from another master are extremely likely to be refused. This refusal is accomplished through the DB (device busy) response to the request, and was discussed in section 4.3.2. In the present protocol, a bus master, upon being refused service, enters a WAIT state for a period of time. The delay is set at the approximate cycle time of typical bus slaves, in the hope that the device will be free when the request is re-tried. Thus we have established a two-level access queue. First a requesting module must get access to the bus; second to the target device. If the device refuses the request, the bus cycle is aborted, and the entire two-level request cycle must be repeated.

An analogy can be drawn to the long-distance telephone network customer. First, he must gain access to the trunk system (busy circuit); second, to the destination station itself (busy signal). If the trunk is acquired, but another needed trunk is not available, or the destination telephone is busy, the entire operation must be retried; hopefully after an appropriate waiting period. Unfortunately, each time an attempt to access a device fails, bus bandwidth is wasted. Worse, the probability of a request's being refused increases with system load on the target device. The result is the "telephone network phenomenon" (fig. 4.18), where service rate (system throughput) reaches a maximum, and then diminishes, with increasing request rate (system load).

A more disturbing problem may be observed by considering the distribution of bus and device access times. Given a "fair" bus arbitration algorithm, the distribution of missed bus cycles will be uniform between 0 and N-1, where N is the number of processors actively seeking the bus. Now consider the distribution of access times to the device, once bus access is obtained. Assume M masters, on the average, are waiting to access the slave device. Each time the device becomes free, it
will accept the first request presented to it. Because the request pattern and wait delay of the processors is unpredictable, the slave may be considered to select the next master at random from the M waiting masters. Thus the distribution of missed slave cycles has mean 1/M, but, alas, it is exponential (fig. 4.19). The probability of waiting an inordinate number of slave cycle times is quite large. Apart from lockout issues, the load on bus bandwidth is discouraging.

Happily, a solution to the device-busy problem is available. The slave need only pend each of the requests in turn, and then service them in order. Pending the masters will prevent them from wasting bus bandwidth with futile access attempts. This means that each slave which anticipates the need to serve several masters concurrently should include a scratch-pad memory for queuing (storing) their requests. The calculation of average queue length is a queuing problem involving exponential arrivals and uniform service time [MUSI76].

4.4.2 Implementing software synchronization primitives

Pended transactions are intentionally divisible. It is impossible to implement test and set primitives by simply tying up the bus to prevent access by other processors. Two solutions are offered. First, the N/S signal could be used to implement a DIL (data in, lock) primitive a la PDP-11. The memory would latch the processor device ID, and then not accept any other request until unlocked by a write request from that processor. Second, it is possible to design a special test and set memory page. Each time a location is accessed for read, the memory controller writes it back as a one. Write operations are unaffected, allowing the location to be cleared.
(fig. 4.18) Telephone network phenomenon

\[ P_n = \frac{1}{N_p} \] for slave cycles

\( p_n = p(1-p)^{n-1} \) for \( p = \frac{1}{N_p} \)

(fig. 4.13) Distribution of missed bus cycles
4.5 Alternatives to the pended transaction bus

4.5.1 Interleaved processors

This technique involves the use of high speed memory modules and a bus switch at each processor [JORD74]. Pended transactions are not used. Instead the bus switches intercede to "speed up" processor requests to where they are compatible with the bus and memory. Bus utilization is reduced, allowing a number of processors to share access to the bus. The principal objection to this scheme is its dependence on high-speed memories. By switching both processors and memories, the PTB allows multiple microprocessors and compatible (i.e., slow) memories to share the bus.

4.5.2 "Omniscient" bus controller

This scheme provides the bus controller (arbiter) with information about the status of the memory units, as well as the processor requests [BODI74, LAVI77]. The controller is thus able to schedule the memories. No processor is allowed to use the bus until the memory unit it desires is free, and therefore no request is ever refused. The main objections to the omniscient bus controller are the complexity of the controller design and the difficulty of expansion. Nevertheless, it is extremely attractive for large, high-performance systems.
CHAPTER 5 Extension to a multiple-bus system

5.1 Multiple-bus system—general

When the available bandwidth of a single bus becomes exhausted, the system may be partitioned into a number of interconnected subsystems. While the PTB protocol provides for concurrent use of a single bus by several processor-memory pairs, the introduction of multiple buses allows simultaneous communication between units of independent subsystems. The success of this approach depends on the ability to find a "good" partition—one where relatively little communication takes place between subsystems.

If the bandwidth of a single bus is $B_b$, then the effective bandwidth for the entire $N$-bus system is

$$B_e = N \cdot B_b$$

The degenerate case of equality arises when there is never a need to communicate between buses. Equation (5-1) can be reformulated as

$$B_e = \frac{N \cdot B_b}{\text{ave. no. of buses used per transaction}}$$

For the simple serial connection of two buses (fig. 2.10), effective bandwidth equals twice the bandwidth of a single bus as long as the bus coupler is not used. At the other extreme, if every request involves both buses, then nothing is gained from the partition. The role of the bus coupler is to isolate the subsystems except when intercommunication is required.

We may also observe that true multiprocessing is no longer possible for a multiple bus system. This is because access to local (on-bus) units is faster than access to remote units, which may involve several bus couplers. Furthermore, the variance of service times increases with the number of buses which are involved with the transaction. Consider again the simple serial connection of (fig. 2.10). Assuming a "fair" arbiter, the distribution of access times for access to bus $A$ is uniform and discrete between 1 and $M$ bus cycles, where $M$ is the number of devices actively seeking the bus. Similarly, for bus $B$, the distribution is uniform between 1 and $N$ cycles ($N$ devices) (fig. 5.1). The distribution of access times for access to $A$ and then to $B$ is given by the convolution
of the two density functions (fig. 5.2). As expected, the worst-case waiting time, in missed bus cycles, is $M + N$.

The cost in terms of wasted bandwidth and increased access time is strong motivation for finding a good system partition. This means that certain frequently-used resources will need to be replicated at each bus or at a central point. The result is a heterogeneous system. Each bus in the system comprises a multiprocessor, but the topology as a whole bears greater resemblance to a tightly-coupled multicomputer system.
(fig. 5.1) Distribution of access times to a single bus

(fig. 5.2) Distribution of access times to two buses, $M \neq N$
5.2 Bus coupler design

We next consider the options for the design of the bus coupler, and how bus couplers can be used to implement various topologies.

5.2.1 Options for the design of the bus coupler

The simplest bus coupler is the single-channel, unidirectional coupler shown in (fig. 5.3). The simple bus coupler is capable of recognizing a request on bus A for devices on bus B, and forwarding the request. The bidirectional and multichannel bus couplers to be discussed later are correctly viewed as repeated instances of the simple bus coupler.

Option (1) Switching method.

Two basic switching methods are available to the designer of the bus coupler. Circuit switching involves the electrical connection of the buses for the period of the transaction. Suppose that module AA (bus A) desires to communicate with module BB (bus B). Module AA first gains control of bus A, and presents its request. The bus coupler recognizes that an off-bus address is being referenced, and requests control of bus B. When control is granted, the bus coupler electrically joins the two buses. Now AA's request is communicated directly to BB. When BB responds, AA removes its request and both buses are released.

Message switching involves relaying the transaction information through the bus coupler. To repeat our example, AA gains control of bus A and presents his request; the bus coupler recognizes the off-bus request as before, only this time, the bus coupler itself acknowledges AA's request, and bus A is freed. Next the bus coupler requests control of bus B, and goes on to complete the transaction. Successful completion of the transaction may or may not be communicated back to AA.

The PTB protocol was designed to support message switching for transaction forwarding. When the bus coupler recognizes an off-bus request on bus A, it pends the transaction immediately, and requests access to bus B. When access is granted, AA's original request is presented on bus B, where it is recognized by BB. Device BB may now pend the bus coupler. When BB has processed the request, the transaction resumes. When the bus coupler is finished with BB, it resumes its original transaction with AA,
and the entire transaction is finished. Thus, the bus coupler appears as a slave on bus A, and a master on bus B.

Option (2) Address recognition

The bus coupler must be programmed to recognize off-bus requests. This may be done either statically, using jumpers, or under control of a local processor. Each bus could be assigned a range of addresses in 24-bit address space. References to addresses outside of the assigned range are recognized as off-bus requests. This scheme requires every device in the system to generate and/or decode 24-bit addresses. An alternative is to use short (16-bit) addresses for on-bus requests only, and long addresses for between-bus transaction forwarding. In this scheme, every device has two addresses: a local 16-bit address, and a global 24-bit address. The mapping between short and long addresses can be done by the bus coupler, under local processor control.

Option (3) Multichannel capability

The capacity of the single channel coupler can be increased by allowing it to process several transactions concurrently, by sharing the control logic between several ports (fig. 5.4). In the worst case, the control logic will need to process one request from bus A and one response from bus B simultaneously. This requires two independent control circuits, and data storage and status flip-flops for each port.

Option (4) Block transfer capability

A common task in a multiple-bus system is moving blocks of data between memories on different buses. The bus coupler is in a good position to accomplish the transfer with greater efficiency than a processor can. If a processor on bus A wishes to move a block of data from a memory module on bus B to a local memory module, it must repeatedly perform a read through the bus coupler, and then a write into local memory. (It is assumed that the block transfer program is itself stored in the processor private memory, and so instruction fetches do not load the bus.) The processor read operation will be pended on bus A and forwarded to bus B by the bus coupler. Thus, for transfers under processor control two reads (processor, bus coupler) and one write are required.
If the bus coupler is used for the block transfer, then only one read and one write are required, plus some overhead to set up the transfer and process the interrupt when it is complete.

Option (5) Serial bus

Parallel transmission of data over distances greater than approximately one hundred feet is impractical for two reasons: first, the cost of cabling becomes excessive; and second, timing skew between data and synchronization signals (and between the data lines themselves) becomes large. For these reasons, serial transmission techniques are generally used. A bus coupler can be designed to perform serial block transfers of data between remote buses. Block transfers are preferred because the transmission of address information for each word transferred would add to the communication overhead tremendously. Data transfer rates in excess of 5 Mbaud (less than 1.6 microseconds per byte) are practical with today's technology.
(fig. 5.3) Single-channel unidirectional bus coupler

(fig. 5.4) Multi-channel bus coupler
5.2.2 Implementation of serial and parallel interconnections

The bus coupler can be used to implement any of the interconnection forms discussed in chapter two. In this section we will consider some of the possibilities and comment on their effectiveness.

(1) linear coupling

The simplest interconnection of two buses was shown in (fig. 2.10) and has been used in examples several times since. In this case a bidirectional bus coupler couples requests from A to B, and vice versa. This topology can be extended by adding bus couplers and buses in a linear fashion (fig. 5.5). The obvious limitation is that communication from one end of the system to the other involves every bus coupler, causing considerable delay.

(2) ring

The linear system with more than two buses may be improved by connecting its two "ends" with one more bus coupler (fig. 5.6). If the bus couplers are bidirectional, then in the worst case, a transaction need travel only half way around the ring.

(3) generalized star

For systems with several buses, the generalized star (fig. 5.7) offers reduced transaction delay at the expense of one additional bus. This bus is used for communication between bus couplers only. Every interbus transaction now passes through exactly two bus couplers.

(4) trunk system

If additional buses are added to handle the communication traffic between bus couplers, then trunk system of (fig. 5.8) results. Since transaction routing is no longer unique, an arbitration must be performed to determine which trunk line will carry the transaction. This amounts to choosing the first available bus coupler. The selected bus coupler pends the transaction and forwards it to its own trunk bus, where the transaction address is recognized by a second bus coupler, which then forwards the request to the destination bus.
(5) generalized trunk/crosspoint system

The trunk system can be further expanded by adding buses with shared modules (e.g., memory) on them (fig. 5.9). These shared modules will be called central resources. Unidirectional bus couplers are used to couple requests onto the central resources buses, but not off of them. The trunk lines are retained for forwarding transactions between buses when the central resources are not involved. In this fashion, central resources may be accessed by all buses through a single bus coupler, rather than through two, as normally required by the trunk scheme.
B.C. = bus coupler

(fig. 5.5) Linear coupling

B.C. = bus coupler

(fig. 5.6) Ring
(fig. 5.7) Generalized star

(fig. 5.8) Trunk system
(fig. 5.9) Generalized trunk/crosspoint system
CHAPTER 6 Implementation Specifics

6.1 Design

This section covers the detailed design for the prototype system. The implementation will be seen to diverge slightly from that presented in chapter four, as a result of the use of standard processors and memories.

6.1.1 Use of explicit virtual bus for initial design

The concept of providing system modules a virtual bus was introduced in chapter 1. The use of a virtual bus means that at some level within each module it is possible to identify a protocol which is different (simpler!) than the PTB protocol. For example, in an i8080-based CPU, the virtual bus is the 8080 processor bus. Another CPU in the system might be M6800-based, and would therefore use a different virtual bus protocol.

For the prototype PTB system, it was decided to use an existing family of compatible processor, memory and I/O modules. The Intel SBC 80/20 processor, SBC 016 RAM memory, and SBC 108 I/O modules were used. These modules normally communicate via a bus protocol designed by Intel, which we will refer to as the SBC protocol.

The SBC protocol is shown in (fig. 6.1) for a read operation, and in (fig. 6.2) for a write operation. It is a fully interlocked request/acknowledge protocol, except that the trailing edge of the acknowledge signal XACK/ is not interlocked. Overlap of the XACK/ signal is prevented by making XACK/ in SBC slave module designs a combinational function of the request signals.

The four request signals are MRDC/ (memory read command), MWTC/ (memory write command), IORC/ (I/O read command), and IOWC/ (I/O write command). The use of the slash (/) is an Intel convention which means that the signal is asserted-low.

The SBC bus also includes signals for bus arbitration, but we will not be concerned with these because the PTB uses its own bus arbitration scheme.

The SBC modules were augmented with special interface circuitry, called Bus Interface Units (BIUs), which maps the SBC protocol to the PTB protocol.
Thus the SBC protocol is the virtual protocol in our system, but it is explicit in the sense that it is identical for every system module. The prototype system consists of two processors and one memory (fig. 6.4).

6.1.2 Asynchronous logic implementation

An asynchronous logic implementation for the BIUs was chosen for a variety of reasons. The primary consideration was speed of operation of the bus switch, so that bus utilization would be minimized. If asynchronous logic is used in conjunction with an asynchronous, fully-interlocked bus protocol, then the protocol proceeds "as fast as possible"; that is, each transition of a synchronization signal causes an immediate (subject only to bus and circuit propagation delays) state change in the destination module—it is not necessary to wait for a clock pulse. The second observation is that although the present system is homogeneous, and the processors could all be synchronized, it is preferable to treat each as an asynchronous unit. Thus, growth of the system to include a variety of heterogeneous modules is not precluded. A third reason for not using synchronous logic is the general problem of clock distribution. It was desired to avoid problems of clock skew and clock buffering.

The decision to use asynchronous logic was not easily made, due to two drawbacks. Since processor operation is not synchronized, processor bus requests will be unrelated: the arbiter problem. It was necessary to incorporate a reliable arbiter latch into the design of the Distributed Bus Arbiter (DBA); a centralized bus arbiter design must also deal with the need to synchronize bus requests.

The second drawback was the apparent lack of a good circuit synthesis procedure. The methods presented in elementary textbooks (e.g., [HILL74]) are not well suited to the problem since their reliance on Karnaugh maps means an inability to handle a large number of inputs and internal states. [UNGE69] provided considerable insight into the theory of asynchronous circuits, but no simple synthesis procedure emerged.

An investigation was made into self-synchronizing asynchronous circuits. This synthesis method is an adaptation of synchronous circuit design techniques. The circuit is designed exactly as for a synchronous implementation, except that instead of synchronizing the inputs to a steady
clock, the clock is synchronized to the inputs; that is, whenever a clock is needed to cause a change in the internal state, a clock pulse is generated. The design of the clock-generating function becomes the principle focus of this procedure [HUER76]. Unfortunately, the investigation showed that self-synchronizing logic would be too slow, primarily due to delays in the clock function.

The method of [ZISS72], called "sequential equations", was finally chosen. This method uses R-S flip-flops as state variable elements. The states are first coded so that no races between secondaries exist, and then "turn-on" and "turn-off" conditions are developed for each state variable. The states were coded with the aid of a "race-free" diagram (fig. 6.5), which allows state transitions between adjacent states only. In order to comply with the race-free diagram it is often necessary to add dummy states. A state transition then becomes a cycle through one or more dummy states. The state assignments were chosen so that the state transitions which implement the PTB protocol involve no cycles, for maximum speed.

6.1.3 Backplane

The backplane is a 1/16 inch thick printed circuit board with positions for thirteen 120-pin edge connectors. Bus wiring uses 20 mil conductors over a ground plane (microstrip) for a characteristic impedance of approx. 110 ohms [BL0072]. The bus lines are terminated on each end with a 220/330 ohm parallel resistor combination which provides -2 volts at a Thevenin equivalent impedance of 132 ohms. Level shifters are used to convert the TTL signals of the bus modules to ECL levels. ECL was chosen for the backplane because of its high speed and low cross-talk properties.

6.1.4 Design of system modules

Two types of Bus Interface Units (BIUs) were designed. The Bus Master Interface Unit (BMIU) is used for master devices, such as processors. The Bus Slave Interface Unit (BSIU) is used for slave devices, such as memories. The BMIU and BSIU both include a Distributed Bus Arbiter (DBA) circuit, although it is logically distinct and is not needed if a Centralized Bus Arbiter (CBA) is used.

The design of the modules will now be described. First, a block diagram will be given, and the overall operation of the module described.
Next, the control logic will be discussed in terms of its state diagram. It will be helpful to refer to the PTB schematics. These are part of a companion document, entitled PTB Supplement [HAAG78].
(fig. 6.1) SBC protocol (read)

(fig. 6.2) SBC protocol (write)
did = device ID
 tc = transaction control

(fig. 6.3) Augmented SBC modules

(fig. 6.4) Prototype PTB system
(fig. 6.5) Race-free state transition diagram [ZISS72]
6.1.4.1 Bus Master Interface Unit (BMIU)

BMIU purpose

The purpose of the BMIU is to interface SBC master type modules to the PTB. The unit is designed for the SBC 80/20 CPU, but can be easily adapted for use with other master modules such as DMA controllers.

BMIU block diagram description

A simplified block diagram of the BMIU, showing inputs and outputs only, appears in (fig. 6.6). The complete BMIU block diagram is shown in (fig. 6.7). At the left margin are SBC bus signals; the right side contains the PTB signals. When the SBC modules wishes to use the bus, it presents address, data (if any), and control information (one of MRDC/, MWTC/, IORC/ and IOWC/). The Transaction Control logic then produces the signal XREQ, which indicates to the Control logic the presence of an external request. The Control logic requests control of the PTB (BCR, bus control request). When control is granted by the bus arbiter (BCG), the Control logic asserts SA (selection acknowledge). When the bus becomes free (BB de-asserted), the Control logic drops SA and asserts BB (bus busy) and SR1 (service request). SR1 enables address, data (write operation only), device_ID, and control information onto the PTB. The data latches are also enabled. The control information is provided by the Transition Control logic, and is simply a recoding of the information presented by the SBC module. DP (data primitive) indicates the presence of control information, while R/W, IO/M and SAD indicate the transaction type.

The addressed slave will respond with one of DB, NPA, and PA. If DB (device busy) is signalled, the control logic removes its request, and releases the bus. After a time-out, the bus will be requested again and the transaction retried. If NPA (non-pend acknowledge) is received, the data bus contents are latched (DLTCH), the bus is released, and the Control logic signals XACK/. When the original request is removed, XREQ is de-asserted, the Control logic de-asserts XACK/, and the transaction is complete. If PA (pend acknowledge) is received in response to the request then the transaction is pended, and the bus is released. Later, when the BMIU recognizes its device_ID (DDID) and a RES (resume command), the transaction continues. The Control logic responds to RES by asserting SR2 (service request) and latching the data bus contents. SR2 causes DP to be
asserted; the slave then removes RES, and the Control logic signals XACK/, as in the non-pend case.

BMIU state transition description (Control logic)

Refer to BMIU state diagram (fig. 6.8)

I state (Idle). Following power-on, and whenever all of the external request signals MRDC/, MWTC/, IORC/ and IOWC/ are unasserted, the BMIU remains in the Idle state. When one of the external requests becomes asserted, the BMIU asserts BCR (Bus Control Request), which is input to the BAU (DBA or CBA). When the BAU determines that the BMIU is the next bus user, it produces the grant signal BCG, causing the BMIU to enter the Selection Acknowledge state.

SA state (Selection Acknowledge). The SA state is entered from Idle when the BAU produces the signal BCG. In the SA state, the BMIU asserts both BCR and SA, maintaining its position as the next bus user. When the bus is free (BB de-asserted), the SR1 state is entered.

SR1 state (Service Request). The SR1 state is entered from SA when BB is de-asserted. The SA signal is de-asserted, indicating to the BAU that the next bus user may be selected. In SR1, the BB signal is asserted, and transaction information (transaction type, address, and data, if any), is presented to the bus. An LED ("Service Request Display") indicates that SR1 is active. The module will remain in the SR1 state until either the external request is removed or an acknowledge is received. If the external request is removed as the result of a bus time-out, the BMIU passes through WAI1 to I. Normally, one of the signals DB (Device Busy), NPA (Non-pend Acknowledge) and PA (Pend Acknowledge) will be asserted, causing WAI1, NP, or P1 to be entered, respectively.

WAI1, NP, P1 states (Wait, Non-pend, Pend). These states are entered from SR1 in response to DB, NPA, and PA, respectively. They are waiting states necessary to complete the interlocked protocol. The transaction information is de-asserted. BB, however, remains asserted. In response to the de-assertion of DP, the addressed slave removes its acknowledge (DB, NPA, PA all de-asserted), causing a transition to one of WAI2, ACK, and P1, respectively.

WAI1 state (Wait). The WAI2 state is entered from WAI1 when the DB
signal is de-asserted by the addressed slave. WAI2 starts a timer ("Wait Display") which causes the external request to be masked for the timer period. When the external request is seen to be masked, state D is entered, followed by a transition to I. The external request remains masked for the period of the timer, after which the request cycle begins again.

P2 state (Pend). The P2 state is entered from P1 when PA is de-asserted by the slave. The interface remains in P2 until it recognizes its DID, and the RES data primitive is asserted, causing a transition to SR2.

SR2 state (Service Request). The SR2 state is entered from P2 when the RESume condition is detected. In SR2, DP is re-asserted. When RES is de-asserted, the ACK state is entered.

ACK state (Acknowledge). The ACK state is entered from NP when NPA is de-asserted, and from SR2, when RES is de-asserted. In ACK, the signal XACK/ is asserted by the interface, signalling transaction completion. When the external request is removed, state D is entered.

D state (Dummy). The D state is entered from WAI2 when the external request has been masked, and from ACK when the external request is removed. An immediate transition to the Idle state takes place.

The BMIU state assignment, excitation equations, and output functions are shown in (fig. 6.9).
BMIU simplified block diagram

(fig. 6.6) BMIU simplified block diagram
(fig. 6.7) BMIU block diagram
(fig. 6.8) BMIU state transition diagram
state variable excitations

A on BCG
off $SR_1 \cdot DB + \overline{GXREQ}$

B on $SR_1 \cdot PA$
off $SR_2 \cdot RES + \overline{GXREQ}$

C on $SR_1 \cdot NPA + P_1 \cdot \overline{PA} + \overline{WAI_1(\overline{DB} \cdot GXREQ)}$
off $D \cdot NPA + BD \cdot \overline{PA} + AD(\overline{DB} \cdot GXREQ)$

D on $SA \cdot \overline{BB}$
OFF $WAI \cdot GXREQ + NP \cdot NPA + P_2 \cdot RES \cdot DDID$

output functions

$SR_1 \quad SR_1$
$SR_2 \quad SR_2$
$BCR \quad (I + SA) \cdot XREQ$
$DLTCH \quad SR_1 + P_2$
$SA \quad SA$
$BB \quad SR_1 + WAI_1 + NP + P_1$
$XACK \quad ACK$
$T_w \cdot \text{(reset)} \quad WAI_2$

$ABCD$
$BD$
$\overline{CD} \cdot XREQ$
$SR_1 + BCD$
$ACD$
$\overline{CD} + AB$
$ABCD$

(fig. 6.9) BMIU state assignment, excitation equations and output functions
6.1.4.2 BSIU

BSIU purpose

The purpose of the BSIU is to interface SBC slave type modules to the PTB. The unit is designed for the SBC 016 16K memory module, but is easily adapted for use with other slave modules including the SBC 104/108/116 series of combination memory and I/O boards.

BSIU block diagram description

A simplified block diagram of the BSIU, showing inputs and outputs only, appears in (fig. 6.10). The complete BSIU block diagram is shown in (fig. 6.11). At the left side are PTB signals; the right side contains SBC bus signals.

When the BSIU is idle, address and data information are transmitted through transparent latches to the SBC slave device. If the slave device recognizes an address as its own, it activates one of RAMADR/, PROMADR/, or IOADR/ (as appropriate), causing the Transaction Control Logic to output ADR, indicating that the slave has been addressed. Additionally, if the pend selection straps PR (pend read) and PW (pend write) indicate that the transaction (read or write) should be pended, the signal P (pend) is produced. The DP signal is delayed to allow for address decoding and output to the Control logic as DDP. DDP is also used within the Transaction Control block to enable the generation of the command signals MRDC/, MWTC/, IORC/ and IOWC/.

If P is not asserted when DDP is received by the Control logic, the BSIU remains idle, and the transaction is not pended. When the slave completes the requested operation it signals XACK/, which is converted to NPA and output to the bus.

If P is asserted when DDP is received, then the Control logic is excited to pend the transaction. Signal TLTCH is removed, latching the address, device_ID, data, and control information, and effectively freezing the bus from the slave’s point of view. At the same time, the signal PA is returned to the requesting module. When the slave signals completion of transaction processing (XACK/), the BSIU gains control of the bus as described previously for the BMIU. Once in control, the control logic outputs RES, DIDEN (device_ID enable) and, if the operation was a read,
DOEN (data out enable). DIDEN and DOEN enable the latched DID and the slave read data (if any), respectively, onto the bus. The addressed (by device_ID) master module replies with DP, and the transaction completes.

BSIU state description (control logic)

Refer to the BSIU state diagram (fig. 6.12).

I state (Idle). Following power-on, and whenever a transaction has been completed, the BSIU remains in the Idle state. In Idle the signal TLTCH is asserted, enabling the 8212 transaction latches, which pass address, data and transaction control information through to the slave module. The slave is thus connected to the bus as long as TLTCH remains asserted. If the slave recognizes an address on the address bus, it will assert one of RAMADR/, PROMADR/ and IOADR/. These are combined with the transaction control signals SAD, R/W, and IO/M, to form ADR (slave addressed). The transaction control signals are also decoded to form the SBC data primitives MRDC/, MWTC/, IORC/ and IOWC. The R/W signal is inverted to produce W/R, which is then combined with the strap inputs PR and PW (Pend Read, and Pend Write) to produce P (Pend). If P and ADR and DDP (Delayed Data Primitive) are all asserted, the BSIU control logic is excited to enter the PA state. If P is not asserted, then the transaction is not pended, and when XACK/ is produced by the slave, it is forwarded to the bus as NPA.

PA state (Pend Acknowledge). The PA state is entered from the I state when a transaction is to be pended. As soon as PA is entered, TLTCH is de-asserted causing all transaction information to be latched in the 8212s, effectively freezing the state of the bus from the slave's point of view. In PA the BSIU asserts the bus signal PA and waits for DP to be de-asserted. When this happens the BUSY state is entered.

BUSY state. The BUSY state is entered from PA when DP is de-asserted. If the slave is addressed while the BSIU is in the BUSY state, the signal DB (Device Busy) is returned. When the slave responds with XACK/, the signal BCR is asserted (Bus Control Request). When the Bus Arbiter responds with BCG (Bus Control Grant), the SA state is entered.

SA state (Selection Acknowledge). In this state, as in the case of the BMIU, the BSIU has been selected as the next bus user. The BSIU asserts the bus signal SA until BB (Bus Busy) is de-asserted, which causes
a transition to RES1.

RES1 state (Resume). The RES1 state is entered from SA when BB is de-asserted. BB is re-asserted by the slave, and the read data (if any) and latched DID (Device ID) are driven onto the bus, along with RES (resume). The addressed (by DID) BMIU responds by asserting DP, causing a transition in the BSIU to RES2.

RES2 state (Resume). The RES2 state is entered from RES1 when the addressed BMIU asserts DP. RES2 is a wait state to complete the interlocked protocol. RES is de-asserted. When the BMIU responds by deasserting DP, the Idle state is re-entered, and the transaction is complete.

The BSIU state assignment, excitation equations, and output functions are shown in (fig. 6.13).
(fig. 6.10) BSIU simplified block diagram
(fig. 6.11) BSIU block diagram
(fig. 6.12) BSIU state transition diagram
state variable excitations

A on  BCG
off  RES₂•DP

B on  PA•DP
off  SA•BB

C on  I•ADR•P•DDP
off  RES₁•DP

D on  I•P•DDP
off  DDP

output functions

RES  RES₁
BB  RES₂ + RES₂
SA  SA
PA  PA
NPA  XACK•P•ADR
TLTCH  IDLE
BCR  (SA + XACK)•BUSY
DB  BUSY•DDP•ADR

(fig. 6.13) BSIU state assignment, excitation equations and output functions
6.1.4.3 Distributed Bus Arbiter (DBA)

DBA Purpose

The purpose of the DBA is to provide bus arbitration for access to the PTB by the BMIU and BSIU modules.

DBA block diagram description

The DBA block diagram is shown in (fig. 6.14). The BxIU module interfaces to the DBA using a simple request/grant mechanism. Bus requests are presented at BCR. When the DBA determines that the BxIU is the next bus user, it asserts BCG. The remainder of the DBA external signals are used in the bus arbitration protocol by which the various DBAs cooperate to assign the next bus user.

Refer to the bus arbitration synchronization timing diagram (fig. 6.15). A bus arbitration sequence begins when the SA signal is de-asserted, indicating that the next bus user may be assigned. If a BCR request is present, the arbiter latch is set, indicating that the DBA should participate in the arbitration cycle. When every DBA has acknowledged the de-assertion of SA, SSY1 is asserted. Now any DBA whose arbiter latch is set will produce SBCR(A), at the output of the arbiter latch block. SBCR(A) excites the control logic to assert SA. SA's being asserted forces all the DBA arbiter latch blocks to produce either SBCR(N), indicating that the BCR was not asserted when SA occurred, or SBCR(A), indicating that it was. Only those DBA's with asserted SBCR(A) will continue in the bus arbitration cycle. Each of these remaining units outputs its request on BRQ and one of the BR[7:0]. If a greater priority bus request exists (GBR), a DBA will remove itself from the arbitration cycle. When every DBA has acknowledged the assertion of SA and inspected the BR lines, SSY2 is asserted. Now the single remaining DBA may signal BCG to the requesting BxIU.

DBA state description (control logic)

Refer to the DBA state diagram (fig. 6.16).

DBA state transition description

I state (Idle). Whenever BCR (Bus Control Request) is inactive, the DBA remains in the Idle state. If SA is asserted, the arbiter latch will
produce SBCR(N), indicating that no external request was present. If BCR is asserted while SA is not, and if for "snap-shot" devices BRQ is not asserted, then the arbiter latch produces a signal indicating that a BCR was recognized and the DBA should participate in the next bus arbitration cycle. If SSYN1 is asserted, indicating that no DBA is in the H or H' states (all have responded to the de-assertion of SA), then the state logic is excited (SBCR(A)) to enter the SA state. Note that SA being asserted also enables the transition to SA. This allows the first DBA to respond to SSYN1 to force the others, which may be slower in responding, into the SA state. In this way, every DBA which has sensed a BCR during SA's being de-asserted enters the SA state nearly simultaneously.

SA state (Selection Acknowledge). The SA state is entered from I when a BCR exists and SA is de-asserted, as described above. The signal SA is asserted, forcing all other DBAs to enter either SA or H' depending on the state of their BCR signals. Also the BR-FF is set, causing a Bus Request to be presented on one of BR[7:0]. If a "greater" (priority-wise) bus request is detected, an immediate transition to state D1, and thence to H, occurs. In this way, every requesting module except the one of highest priority enters the H state. Finally, when SSYN2 is asserted, indicating that no DBA remains in the I state (and therefore that all BR_i have been entered, and every requesting module except the one of highest priority has entered the H state), the selected module makes the transition to D2.

H' state (Hold). The H' state is distinguished from the I state by SBCR(N) being asserted. It is a waiting state that indicates that the DBA is sitting out the present arbitration cycle.

D1 and D2 states (Dummy). These states are entered from SA. An immediate transition to H or BG occurs, respectively. The signal SA remains asserted during either D state.

H state (Hold). The H state is entered from SA via D1 when a greater priority bus request is detected. In H the DBA holds for resynchronization with the SA signal. H is exited when SBCR(A) is de-asserted following the de-assertion of SA.

BG state (Bus Grant). The BG state is entered from SA via D2 when it is determined that the requesting module is the next bus user. The BG
state outputs the signal BCG (Bus Control Grant) to the requesting module, which responds by asserting SA itself (thus assuring the continuity of SA), and removing BCR. When BCR is removed, the DBA enters the I state where it remains until BCR is again asserted. Meanwhile, the (selected) BxIU monitors the BB line to determine when the bus is free. When BB is de-asserted, the BxIU asserts BB and drops SA, allowing a new arbitration cycle to begin.

The DBA state assignment, excitation equations, and output functions are shown in (fig. 6.17).
(fig. 6.14) DBA block diagram
(fig. 6.15) Arbitration synchronization
(fig. 6.16) DBA state transition diagram
state variable equations

A on
off

B on
off

C on
off

output functions

SSYN₁ I
SSYN₂ I
BCG BG
SA SA + D₂ + BG

(fig. 6.17) DBA state assignment, excitation equations, and output functions
6.2. Conceptual Design

The discussion to this point has considered the motivation for and design of the PTB protocol, and its implementation by augmenting standard computer modules. Although the implementation demonstrates the viability of the pended-transaction concept, the resulting system is not useful in a general sense, because several features needed for multiprocessing are not present. In this section we will present some of the design considerations for additional modules, to be constructed either from scratch or by further augmenting existing modules. It is hoped that this discussion will provide the means for project continuity.

6.2.1 Processor

The processor module comprised of an Intel SBC 80/20 and BMIU is deficient in several respects, principally:

1. it is difficult to program (instruction set);
2. its logical address space is limited (addressability); and
3. processor traps are not implemented (O/S support).

The blame for these faults lies primarily with the SBC 80/20 processor design: it was not meant to support the advanced features required for general-purpose multiprocessing.

A replacement processor module would not utilize the concept of an explicit virtual bus. Instead, it would implement the PTB protocol directly. The processor module could be based on an 8080 microprocessor, but extensive outboard circuitry would be required to correct for its deficiencies. In the present system, this circuitry could be added to the BMIU board. The result would be an interface module tailored to the SBC 80/20 specifically. Similarly, other custom interfaces could be developed for the remaining SBC family modules.

An alternative to processor augmentation is the use of a microprogrammable processor chipset (e.g., the Am2900 series [AMD 76]), or the construction of the processor from SSI and MSI parts. The design of a processor module should consider the following general areas.

INSTRUCTION SET. Programming a microprocessor like the 8080 for other than the most trivial of tasks can be an exasperating experience. One
reason is the lack of flexible addressing modes. Indirect or base-displacement (sometimes called indexed) addressing is necessary for accessing operating system tables and subroutine parameter blocks whose location may not be known at assembly time. The reduced dependency on direct addressing facilitates program modularity and the use of reentrant code. Other improvements are needed in multibyte arithmetic, especially signed arithmetic, and block move and bit manipulation instructions.

One solution to the instruction set problem is to abandon assembly-language programming altogether in favor of a high level language like PL/M [INTb75]. The use of a good high level language relieves the programmer from dealing with the idiosyncrasies of the processor instruction set, but does not guarantee more efficient code.

Another alternative is the use of on-board emulation techniques [WIDD75, LAUG76]. In this case, the processor's native instruction set is replaced by a redesigned instruction set which is then emulated by the processor module. The emulation program is contained in memory which is local to the processor module. This technique has the further advantage that processor main memory and bus bandwidth requirements are reduced, but slower program execution will generally result from the increased overhead of the emulation. A variation of this idea which would be practical for the SBC 80/20-BMIU processor involves augmenting the 8080 instruction set to include advanced addressing modes, etc. Only the added instructions would need to be emulated. Each of the new instructions would have a two-byte opcode consisting of one of four RST instructions (i.e., RST7-RST4); RST3-0 would be reserved for other functions), and one other byte. Although theoretically 210 new instructions could be defined in this way, the second byte would probably be divided into several fields to speed vectoring to the emulation routine.

MEMORY ADDRESSING. A typical multiprocessing system will have a large amount of primary memory. A simple heuristic for the amount of memory needed is to multiply the size of a typical processor working set (with allowance for multiprogramming) by the number of processors in the system. The PTB 24-bit address bus can accommodate up to 64 processors, each with a working set of 256K bytes, for example. Two basic alternatives exist for utilizing this large address space. First, the processor logical address space can be expanded to 24 bits. This choice is a little unwieldy since
it means increasing the length of CPU address registers to 24 bits. Second, some sort of address translation could be performed. In this case, the logical address space of the processor need be no larger than the size of the largest anticipated working set. An appropriate size is 18 address bits, although the 16 bits offered by most microprocessors will generally suffice.

A number of address translation schemes have been used [POPP77]. Memory mapping (fig. 6.18) uses the most significant bits of the processor logical address as a page descriptor. A mapping function is applied to the descriptor to produce a physical block number. The block number is concatenated with the remainder of the logical address to form the physical address. Base-displacement translation (fig. 6.19) involves adding the contents of a base register to the logical address to produce the physical address. The base register may be shorter in length than the physical address, but its MSB is always the MSB of the resulting address. More than one base register may be used, but then the instruction set must allow for base register specification. Bank switching is a degenerate case of memory mapping where no page descriptor is used; instead, a single bank register is concatenated with the entire logical address.

The use of base registers is difficult because the addition is expensive and time consuming. Bank switching is easily implemented, but requires an entire program module to be present at all times, since there is no way to detect references to non-resident addresses. Bank switching also results in decreased memory allocation flexibility unless a relocating loader is used. For example, in the case of a 16 bit logical address, each processor would have to be allocated an entire 64K block of memory, unless programs could be relocated within that block.

Address mapping can be implemented at less cost and greater speed than base register schemes, and because the page size can be made reasonably small, it offers greater flexibility than bank switching. A page size of 8K would be a good choice for an SBC 80/20-based processor module. Eight map registers would then be required.

A third alternative for increasing addressing capability is to place the address translation mechanism in the bus coupler. Addresses local to a bus would all be short addresses (16 bits), but the bus coupler would map any portion of 16 bit local address space to 24 bit system address space.
Each slave device would now have to respond to both local (short) addresses and global (long) addresses. The major drawback here is that the processors on a single bus would have to share the local address space, equivalent to dividing the 16 bit logical address space between processors. A second consideration is that the operation of the bus coupler would become much less transparent to the individual processors.

**O/S SUPPORT.** A general-purpose processor must protect itself from certain user errors. The user program should be prevented from executing privileged instructions (e.g., I/O instructions) and from running wild through memory. These considerations require further additional circuitry to monitor the user instruction stream and address bus. Two or more modes of processor operation are generally used. The simplest case uses two modes. In user mode an attempt to execute a privileged instruction or reference an illegal memory location results in the operation's being suspended and an immediate trap (processor self-interrupt) to an exception handler in the operating system. When the trap occurs, supervisor mode is entered, and the processor can correct the problem or discard the user process.

A related issue is that of processor interrupts from other units (processors, channels, or I/O devices). Processor-to-processor interrupts are especially useful in multiprocessor systems where it is desired to notify a processor of the existence of a message destined for one of its active processes. Processor interrupts can be implemented through processor addressing. In this scheme, the interrupting unit addresses the processor as a slave, and writes interrupt information (including priority) into the processor interrupt port. Special hardware then schedules the interrupt according to its priority. In general, a number of interrupts could be stacked at the processor for servicing.
LOGICAL ADDRESS

descrriptor  offset

M( )

PHYSICAL ADDRESS

physical block number  offset

M( ) = mapping function

(fig. 6.18) Memory mapping

LOGICAL ADDRESS

offset

+  

base register

PHYSICAL ADDRESS

(fig. 6.19) Use of base register
6.2.2 Memory

The SBC016 - BSIU memory module can be improved by adding the following features:

(1) ability to respond to long (24 bit) and short (16 bit) addresses (addressing);
(2) ability to implement test and set operations (memory lock); and
(3) ability to stack service requests (queuing).

These changes can be easily made by modifying the BSIU board. The result would be an interface module tailored to the SBC 016 memory board.

ADDRESSING. The extension of 24 bit addresses would require decoding the eight high-order address bits, in addition to ADR[15:14], which are presently decoded. The decoding could be implemented with two 74S85 comparators. If bus signal SAD (short address) is asserted, then the module would require an address match on bits ADR[15:0] only; otherwise, the entire 24 bit address would be used.

MEMORY LOCK. One choice for the implementation of software synchronization primitives is the use of memory lock (see section 4.4.2). If the processor is able to exercise the DIL (data in, lock) primitive, then it is desirable to equip the memory to respond correctly. This change is easily made to the BSIU circuit. If DIL command (indicated by DP = 1, R/W = 0, N/S = 1) is received, the BSIU sets a LOCK flip-flop. If LOCK is set, when the BSIU returns to the Idle state, the DID latch contents will not be allowed to change. Instead, the DID's of further requests will be compared to the latched DID, and the request will be refused, unless the DID's match. The BSIU becomes unlocked when a request is accepted which is not a lock operation.

QUEUING. The device-busy problem was discussed in section 4.4.1. There it was shown to produce a large variance in service times and a consequent increased load on system bandwidth. The problem can be resolved by providing additional circuitry at the BSIU to stack requests as they occur and service them on a FIFO basis. The pend mechanism would be used, and a small memory for storing address, data, and device_ID information would be required. The choice of depth of the stack is a system (software) design parameter, but since any stack smaller than the total number of processors in the system could be overflowed, device-busy should be
Another issue for memory design is whether write operations should be pended at all. Since they are unidirectional, write operations could be acknowledged immediately (NPA), leaving the slave in the BUSY state. A problem arises in this case if a request is received during the transition from the BUSY to IDLE states. A momentary DB response could be given, followed by a NPA. Evidently, by not pending write operations, another need for arbitration is introduced. This problem is avoided in the present design by always allowing the operation to be completed before signalling NPA, or by pending the transaction immediately. In the latter case, the normal bus arbitration mechanism resolves the conflict between further requests, and the transition from BUSY to IDLE.

6.2.3 Input/Output

There are two areas of interest in the design of the I/O structure:

(1) I/O device addressing; and
(2) Interrupt handling.

I/O DEVICE ADDRESSING. Like memory, I/O devices may be either local to a processor or shared among various processors. It is assumed here that I/O devices are local to a bus. In a multiple bus system, I/O transactions are not forwarded between buses; rather, requests for I/O are communicated through shared memory, and the actual transfer is performed by a local processor.

If the I/O device is local to a processor, then addressing is a problem for the processor design only, since the bus is not used. Shared I/O devices may be addressed directly as memory (memory-mapped I/O), or separately, through the use of I/O instructions which control a set of I/O data primitives. The PTB design supports both schemes. By convention, the SBC family modules use an 8 bit I/O address. Since I/O transactions are direct register operations, they need never be pended. Thus the BSU is not activated during an I/O transaction, and its design is independent of device addressing.

INTERRUPT HANDLING. Several methods for handling device interrupt may be identified [GOUT66, WIDD76, LORI71].
(1) Local devices

If every device is local to one processor, then neither device I/O nor interrupts need use the bus. Instead, the processor associated with the device handles device I/O directly. This scheme has the advantage of simplicity, but it is not very flexible, in that it is difficult to add devices to the system.

(2) Direct connection

If I/O modules distinct from processors are used, the interrupt condition can still be communicated directly to a single processor, independent of the bus. The processor must now use the bus to request device status, etc. This scheme offers greater flexibility than local devices (1), but is still constrained in that one processor only can respond to the interrupt condition.

A variation of the direct connection scheme involves designating a single processor as the interrupt processor. The interrupt processor receives all interrupts directly and either services them itself or schedules the interrupt for service by some other processor.

(3) Interrupt broadcast

This scheme allows any available processor to respond to an interrupt condition. The interrupt is broadcast to all processors on one of N priority levels. Any (all) processor which is enabled to respond on that level will be interrupted. Conflicts between responding processors can be settled with hardware interlocks or software synchronization primitives.

(4) Process(or) addressing

This technique is far more complex than (1-3). Two variations exist. In both cases, the interrupting device communicates the interrupt condition directly to a process or processor, by using the bus. For processor addressing, the processor stores the address of its interrupt port in a device register at the start of the I/O transfer. For each subsequent interrupt, the device requests the bus and then writes status information into the processor interrupt port. For process addressing, the process ID, instead of the processor interrupt port address, is stored at the device. To request service, the device gains control of the bus, and presents the process ID along with an "interrupt" indication. The interrupt indication
may be a special bus line or a dedicated interrupt address. The processor which recognizes the process_ID as belonging to one of its processes, responds to the request by stacking the interrupt for service.

Interrupt methods (1-3) are presently supported by the PTB design. Method (4) would require extensive additions to the BMIU and BSIU hardware, and is probably impractical.

### 6.2.4 Shared arithmetic unit

In this section we consider how a shared arithmetic unit (AU) could be added to the bus. Typical microprocessors are not capable of integer multiplication or division, or any floating point operation. The software to implement these capabilities is both lengthy and slow. The incorporation of a high-speed arithmetic unit into the system may be attractive if it can be conveniently shared by several processors.

Sharing of the arithmetic unit hardware is accomplished by using several service ports. A process which requires the AU first acquires access to one of the ports through an operating system call. When the AU is needed, a parameter block is set up containing the operand and result addresses, and the code of the desired operation. The address of the parameter block is then written to a memory location corresponding to the assigned AU port. The AU responds by pending the write operation and processing the request. When the AU finishes processing the request, the write operation is resumed, and the requesting processor can read the results immediately.

### 6.2.5 Bus Coupler

A variety of options for the design of the bus coupler were presented in section 5.2.1. We will now consider how the simplest of these can be implemented. A single-channel unidirectional bus coupler suitable for a serial connection of two buses can be constructed from a BSIU and BMIU connected back-to-back, as shown in (fig. 6.20). When the BSIU recognizes an off-bus request, it pends the transaction. Meanwhile, the BMIU unit has already forwarded the request to the coupled bus. In the course of handling the forwarded transaction, the BMIU may itself be pended. When the transaction involving the BMIU is complete, XACK/ is propagated back to the companion BSIU, which then completes the original transaction, freeing
the bus coupler for another request.

Multichannel and bidirectional bus coupling can be obtained through the addition of BSIU-BMIU pairs, or by designing a special bus coupler circuit which shares the BSIU and BMIU control logic between a number of ports.
t.c. = transaction control

(fig. 6.20) Single-channel unidirectional bus coupler
6.2.6 Variations of the pended-transaction protocol

The protocol presented in chapter four is not the only protocol which can be used to implement pended-transactions. Two changes to the protocol are considered in this section. Each adds overhead to the communication, but results in increased flexibility.

(1) Multiply-pended transactions.

The protocol of chapter four (fig. 4.16) allows a transaction to be pended only once; following RESume, it must go to completion. Ignoring bus allocation, this corresponds to the following trite conversation between a master and slave:

MASTER: Here is my request (DP).
SLAVE: Thank you. I will notify you when processing is finished (PA).
      (the bus is released and the transaction is pended; later...)
SLAVE-NOW-MASTER: Processing is finished. Here are the results (if any)
      (RES).
ERSTWHILE MASTER: Thank you (DP).

The problem here is that if requests are queued at the slave, as discussed in sections 6.2.1 and 6.2.2, all the transaction information (address, data, and device_ID) must be stored. This problem can be avoided by using multiply-pended transactions as illustrated in (fig. 6.21), and by the following conversation:

MASTER: Here is my request (DP).
SLAVE: Thank you. I will notify you when processing is finished (PA) (the bus is released, etc.)
SLAVE-NOW-MASTER: What was your request? (RES)
MASTER (reasserting position as master): Here is my request (DP).
SLAVE: Thank you. I will notify... (PA).
      (the bus is released, etc.)
      (this last exchange may be repeated indefinitely, until...)
SLAVE-NOW-MASTER: What was your request? (RES)
MASTER: Here is my request (DP)
SLAVE: Thank you. Processing is finished. Here are the results (if any) (NPA).

Now the slave module need only queue the device_ID of the requesting
module. When it is able to begin processing the request, it asks again for
the request particulars, stores them and, again, pends the transaction.
The transaction finally goes to completion when processing of the request
is finished.

A comparison of (fig. 4.16) and (fig. 6.21) shows that the multiply
pered transaction protocol requires an additional transmission delay
during the RES phase. Thus, communication overhead is increased by
one-eighth, even if the transaction is pended only once. If the
transaction is pended twice, then overhead is increased by approximately
one-half, and so on. The trade-off is between space (storage registers for
transaction information) and time (communication overhead).

(2) Use of transaction_IDs

The use of device_IDs for transaction linking limits the number of
devices which may share a bus. Since electrical considerations constrain
the bus length to a maximum of approx. two feet, allowing 16 device_IDs
seems reasonable. Device_IDs will be rapidly exhausted, however, if a
multichannel bus coupler is used, since one ID is needed per channel. A
solution to this problem is to associate IDs with transactions instead of
devices. A

The number of transaction_IDs needed is determined by the maximum
number of pended transactions likely to exist on a single bus at any
moment, and is probably fewer than sixteen.

Some method of assigning transaction_IDs to pended transactions is now
required. A solution is to use a central bus controller to maintain a
stock of TIDs, distribute them as needed, and collect them when they are
released. A protocol which implements both TIDs and multiply-pended
transactions is illustrated in (fig. 6.22). The signal PA is replaced by
two signals: PR (pend request) and PG (pend grant); similarly, RES is
replaced with RR (resume request) and RG (resume grant). Every pend or
resume operation now involves the bus controller, in addition to the master
and slave. The conversation now goes like this:

MASTER: Here is my request (DP)
SLAVE (to bus controller): I cannot meet this request immediately. Will
you please assign a TID and pend the transaction? (PR)
BUS CONTROLLER: Here is a unique TID; this transaction is pended (PG).
(master stores TID and removes DP)
(slave stores TID and when DP is de-asserted, removes PR)
(bus controller removes PG)
(master releases bus; the transaction is pended)

SLAVE–NOW–MASTER (to bus controller): Please resume the transaction corresponding to this TID (RR).
BUS CONTROLLER: The TID is returned to the pool. This transaction is resumed (RG).
MASTER (re-asserting position as master): Here is my request (DP)
(slave removes RR)
(bus controller removes RG).
SLAVE (to bus controller) (as before): I cannot meet this request, etc.
(the transaction is pended again).

--OR--

SLAVE: Thank you. Processing is finished. Here are the results (if any) (NPA)

The overhead involved in this protocol is considerably greater than the two protocols discussed previously. Also, the design of the bus controller is non-trivial. This approach should only be considered for systems where a very large number of devices and bus coupler channels must be configured on a single bus.
(fig. 6.21) Multiply-pended-transactions

(fig. 6.22) Use of transaction IDs
CHAPTER 7 Conclusion

7.1 Summary

One of the principal impediments to the development of multimicroprocessor systems has been the high cost of interconnection compared to the cost of individual processor and memory components. The single shared bus is a low-cost structure which provides the flexibility needed for multiprocessing. Unfortunately, the available bus bandwidth is rapidly exhausted as devices are added to the bus.

For microprocessors and compatible memories it was shown that traditional bus communication techniques make inefficient use of the bus. High bus utilization, rather than finite bus bandwidth, was found to be the limiting factor. A new bus protocol, called the pended-transaction protocol, was designed. The PTB protocol increases available bus bandwidth by making bus cycle time depend on the bus components only, and not on the cycle times of the modules using the bus. With the PTB protocol, several processors may be configured on a single bus, with negligible contention.

The PTB protocol supports the use of bus couplers for configuring multiple-bus systems. When the bandwidth of a single bus is exhausted, additional buses can be added. The appearance of a single system-wide bus is maintained by forwarding transaction information between buses through the bus couplers. A variety of configurations are possible.

The design of a prototype PTB system, consisting of two processors and two memories, was covered in detail. Standard OEM processors and memories were augmented with special interface logic to implement the PTB protocol. The next section contains suggestions for further development work on the prototype system.

7.2 Suggestions for further work

This section outlines the steps which should be taken in continued PTB development. The design issues presented here were all covered in chapter six.

(1) Design review

A thorough review of the PTB protocol and arbitration technique should
be undertaken. The use of multiply-pended transactions, transaction_ID's, and global device ID's should be investigated. Centralized bus arbitration should be considered as a way of reducing costs. A centralized arbiter design should reconsider the arbitration algorithm, because a more complex algorithm implementation is feasible.

(2) Specific BIUs

The BIUs should be tailored to the individual modules. Three types of BIUs are envisioned:

- pBIU--interfaces SBC 80/20 processor module

In addition to the BMIU functions, the pBIU would contain address mapping and O/S support circuitry, as well as some means of interrupting the processor via the bus.

- mBIU--interfaces SBC 016 16K memory module

A service stack should be added to the BSIU design to alleviate the device busy problem. This has implications for the protocol design, since the use of multiply-pended transactions would reduce the number of registers required for transaction storage.

- ioBIU--interfaces SBC 104/108/116 combination I/O and memory module

In addition to the service stack, this interface should include provision for interrupt handling, via any of the techniques discussed in chapter six.

(3) New system modules

The first new module which should be designed is the bus coupler. A four channel unidirectional coupler design is suggested. These could be paired to form a bidirectional coupler when needed. Next, the design of a processor module specifically tailored to multiprocessing should be undertaken. Last, a shared arithmetic unit should be considered.

(4) Multiple-bus system

Once a bus coupler is designed, a multiple-bus system can be assembled. The configuration of (fig. 7.1) is suggested for the initial system. This configuration is a modified generalized star. Processor buses use the single shared bus to communicate, via bidirectional bus
couplers. The shared bus also contains central system resources, e.g., shared memory, I/O, and a test and set memory page for implementing software synchronization. This system can later be expanded to the generalized trunk/crosspoint system described in chapter 5.

(5) Software

Initially, Intel's PL/M should be used for programming the 8080-based processors. To reduce bus utilization, the 8080 instruction set should be augmented with more powerful instructions, which would be emulated in local memory. A new compiler will then be required. When the programming tools are available, the development of an operating system for the system of (fig. 7.1) will be feasible. A floating-executive scheme is suggested.
B.C. = bus coupler
T&S = Test and Set
memory

(fig. 7.1) Suggested multiple-bus system
A.1 Photographs of prototype system.

This appendix shows photographs of the prototype PTB system, and waveforms observed during system operation.

The BMIU and BSIU were constructed using custom wire-wrap panels which resemble extender cards with circuitry on them (e.g., fig. A.1). The SBC family module plugs into these cards to form a complete processor or memory module (fig. A.2). The modules are housed in a card cage (fig. A.3), which is mounted in a 19" relay rack, along with power supplies, etc. (fig. A.4).

The waveforms shown in figs. (A.5-A.8) were obtained by causing both processors to execute the same JMP $ instruction in a single memory. (Fig. A.4) shows the initial, or DP phase of the pended transaction, while (fig. A.6) shows the RESume phase. The delay from the leading edge of DP or RES to its acknowledge (PA and DP, respectively) is controlled by a delay network contained in the BSIU and BMIU. The difference in the two delays here is because the value of the delay network is not adjusted to its minimum value. (Fig. A.7) shows a typical Bus Busy (BB) pattern for two processors: the bus utilization is seen to be small. (Fig. A.8) shows a bus exchange of approx 25 ns; this is believed to represent the minimum bus exchange time for the system, and it occurs when there is contention for use of the bus.
INTENTIONAL DUPLICATE EXPOSURE

(fig. A.1) BMU card

(fig. A.2) Complete processor and memory modules
(fig. A.3) Card case

(fig. A.4) PTB rack
(fig. A.5) DP phase
50 ns/div

(fig. A.6) RES phase
50 ns/div

(fig. A.7) Typical BB pattern
500 ns/div

(fig. A.8) Bus Exchange
50 ns/div
BIBLIOGRAPHY


[BELL76] Bell, Gordon, "Computer structures: What have we learned from the PDP-11?" IEEE 3rd Annual Symposium on Computer


Daniel, J.M., and Irwin, J.D., Memory Protection in Multiprocessor Computer Systems. Technical Report AU-T-17,


[FARB72] Farber, D.J.,

[FARB73] Farber, D.J., and F.P. Henrich, "The distributed computer system," Digest of Papers, COMPCON 73, pp. 31-34.


Grimes, J.D., "Distributed processing concepts using microprocessors," Digest of Papers, COMPCON 77, pp. 140-144.


[HAAG78] Haagens, Randolph B. PTB Supplement (MRG internal document), MIT.


[INTE77] INTEL, SBC-80/20 Hardware Reference Manual. Santa Clara, CA:


[LOCa72] Lockheed Electronics, SUE Computer Handbook. Los Angeles, CA:
Lockheed Electronics, 1972.


Maples, M.D., "Microprocessors in computing?" COMPCON 75 SPR, pp. 69-71.


Widdoes, L.C., "Architectural considerations for general purpose multiprocessors,"


Withington, F.G., "Beyond 1984: A technology forecast,"
Datamation, Jan. 1975, pp. 54-73.
