HIGH PERFORMANCE PROGRAMMABLE DSP ARCHITECTURES

by

Mordechay Toma Ilovich

Memorandum No. UCB/ERL M88/31

20 May 1988

HIGH PERFORMANCE PROGRAMMABLE DSP ARCHITECTURES

by

Mordechay Toma Ilovich

Memorandum No. UCB/ERL M88/31

20 May 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

HIGH PERFORMANCE PROGRAMMABLE DSP ARCHITECTURES

by

Mordechay Toma Ilovich

Memorandum No. UCB/ERL M88/31

20 May 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

High Performance Programmable DSP architectures

Mordechay Toma Ilovich

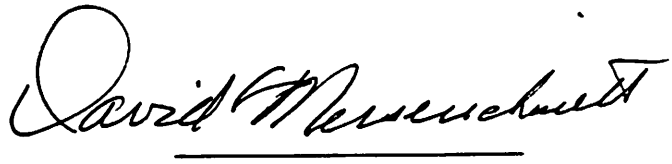
ABSTRACT

The large increase in the complexity of computations and the processing speed requirements of digital signal processing applications require us to implement high-throughput processing elements (PEs) which can also be used to implement high performance multiprocessing systems.

In this thesis we propose and explore many of the design aspects of a new PE architecture that is suitable for inclusion in multiprocessing systems. It incorporates on the same chip a processing unit (PU) and an autonomous interprocessor communication unit (AIO). Concurrently with the PU that executes the task's computations, the AIO handles and controls the data transfer between the PEs. The AIO operates as 1) an interface between the PU and the network and 2) an intermediate network switch to transfer data between PEs. To avoid network congestion and to achieve a high throughput, the use of virtual-cut-through (VCT) switching and an acknowledgement handshaking protocol is proposed. Four I/O links enables the PE to be embedded in any network with a topology consistent with this number of links, and provides the capability to expand to large multiprocessing configurations. The proposed PE might be build as catalog parts for building multiprocessor signal processing systems. Alternatively, the basic AIO unit might be designed as a macrocell to be incorporated into ASIC (Application Specific IC) implementations. Each AIO can be coupled with a different PU design to yield heterogeneous multiprocessor systems. Further, the communication configuration of the AIO can be parameterized in the macrocell and configured to suit each potential application.

Although DSPs that possess special signal processing features are fabricated in small feature size technologies, their throughput is limited by clock skew problems, limiting their usefulness for some real time applications. To overcome this clock skewing problem, an asynchronous processor architecture is proposed. In an asynchronous processor, no clock is required since the functional blocks are built of asynchronous circuits that communicate through asynchronous interconnection handshake blocks. In the asynchronous processor, the execution time of each instruction is data and instruction dependent, and therefore the "average" throughput will also increase.

When clock skewing is insignificant but the throughput of a synchronous processor is limited because of a large variation in instruction execution time due to data dependency, we propose a GSLA (Globally Synchronous Locally Asynchronous) architecture. This architecture incorporates a clock with a variable duty-cycle. The functional blocks signal the control unit upon the completion of their task and the control unit varies the clock's duty cycle to start a new task. The design principles developed in this thesis should be useful for the development of many general-purpose and special-purpose multiprocessor architectures in the future.



David G. Messerschmitt

Chairman of committee

ACKNOWLEDGEMENT

I would like to express my appreciation to my advisor Prof. D.G. Messerschmitt for his guidance, help and support throughout my study. Special thanks and appreciation to Prof. J. Walrand who advised and helped me all the time and was a member of my committee. I wish also to extend my appreciation to Prof. C. Stone, a member of my committee, Prof. R. Katz, a member of my committee, and to Prof. E.A. Lee for some interesting discussions.

Many thanks to my fellow graduate students, T. Meng, K.K. Parhi, V.K. Madisetti and W.H. Ho for the interesting discussions. Special thanks to my friend T.M. Chen for reading and commenting my draft and for our interesting discussions.

This dissertation is dedicated with all my love to my wife Hana and our children Yaron, Yuval and Yoav. Their love, help, encouragement, support, persistence and understanding were indispensable. Their role in my studies are as great as my own.

I also like to thank my parents Mr. and Mrs. Ilovich, my parents in law Mr. and Mrs. Melinarzevitz, my brother Mr. E. Ilovich and his family and our close friend Dr. B. Landkof and his family for their overseas help and encouragement.

This research was supported by grants from NSF and SRC.

Table of Contents

CHAPTER 1

Introduction

1. Introduction

Digital signal processing algorithms are used in a large variety of applications, including image processing, speech processing, sonar and radar systems, biomedical and geophysical (seismic) systems, artificial intelligence, and weather prediction. These applications involve a a large amount of data and computations and require fast computation and high throughput (computation rate).

Most signal processing algorithms are repetitive and allow a high degree of parallelism. These algorithms include complex computations such as transform techniques, convolution/correlation filtering and matrix operations. Transform type techniques include DFT (discrete fourier transform), FFT (fast fourier transform), discrete cosine transform, Karhunen-Loeve transform, Walsh-Hadamard transform, and so on. Filtering types include FIR, IIR, 1-D and 2-D convolution and correlation, 1-D and 2-D interpolation and resampling, linear phase filters: low-pass high-pass and band-pass, Wiener and Kalman filtering, adaptive filtering, window filtering (rectangular, Gaussian, Hamming), differential filtering (gradient, Laplacian), etc.. Matrix operations include matrix multiplication, matrix triangularization (QR decomposition), matrix inversion, singular value decomposition (SVD), eigenvalue computation, solution of Toeplitz linear systems, etc..

The large increase in the complexity of computations, processing speed requirements and the volume of data handled in these applications makes it important to

implement architectures that will increase the computation rate of real-time digital signal processing. The availability of low-cost, high-density, high-speed very large scale integration (VLSI) devices and the emerging of computer-aided design (CAD) facilities enable us to design fast processing elements and high performance multiprocessing systems. This dissertation's main objective is to propose new architectures for processing elements that increase the throughput of a single DSP (digital signal processor) and allow highly concurrent processing systems. The dissertation contains two themes. The first theme investigates and describes an architecture of a processing element which increases the throughput of a multiprocessing systems. The second theme investigates and describes various asynchronous processor architectures which overcome clock skewing problems.

The large degree of parallelism inherent in digital signal processing algorithms and the large amount of data and computations involved in them, suggests the partitioning of computations onto a large number of processing elements. Such multiprocessing architectures typically waste computation time on interprocessor communication, which limits the speedup and the throughput obtained by N processors operating concurrently and transferring data among themselves.

To overcome the communication latencies and the wasted computational time, a processing element (PE) which incorporates on the same chip a processing unit (PU) and an autonomous interprocessor communication unit (AIO) is proposed. Concurrently with the PU that executes the task's computations, the AIO handles and controls the data transfer between the PEs. The AIO operates as 1) an interface between the PU and the network and 2) an intermediate network switch to transfer data between PEs. Operating as an intermediate network switch enables the PE to be embedded in any multiprocessing configuration.

Chapter two describes different existing multiprocessing systems, the proposed

PE, and its operation. Chapter three describes the techniques and modes of PE-PE communication, the interconnection protocols and the different I/O configurations. Chapter four has a detailed description of the hardware implementation, including a multiprocessor performance analysis and a performance comparison between the proposed PE and existing DSPs.

Current DSPs possess special features which make them very effective for digital signal processing. Among them are:

- multiplier which can also multiply and accumulate in one cycle.
- ALU with pre- and post-shifting capabilities useful for scaling operands and results.
- address computation unit which allow to pipelining of address calculations with data path operations.
- A Harvard architecture, which means separate memories for data, program and coefficients, and facilitates parallel prefetching of data and instructions.
- multiple buses to increase the bandwidth of data/instruction transfers.

Although the existing DSPs, fabricated in a small feature size technologies (0.8-2 micron), possess these special features, their throughput is limited by clock skewing problems and they are not adequate to be embedded effectively in a multiprocessing system implementation.

To overcome the clock skewing problem an asynchronous processor architecture is proposed. In designing and implementing an asynchronous processor architecture, no clock is required since the functional blocks are built of asynchronous circuits that communicate through interconnection handshaking blocks. The communication is done by handshaking at the completion of each task. Such an synchronous design eliminates limitations on the throughput imposed by use of a clock, the throughput should therefore increase as the logic speed increases. In the asynchronous processor, the execution

time of the circuit implementation is data and instruction dependent, and therefore the "average" throughput of an asynchronous processor will also increase.

Chapter five introduces the design of an asynchronous processor. A timing analysis is performed to obtain the conditions on handshaking and clock skew delays such that the asynchronous architecture yields a higher throughput. Chapter six proposes different asynchronous architectures and their implementation.

If and when the clock skewing problems due to IC design and fabrication are solved, the major advantage of the asynchronous architecture diminishes. However, the large variation in the execution time due to data dependency still exists. To overcome this problem in the synchronous processor implementation, a new architecture named GSLA (Globally Synchronous Locally Asynchronous) is proposed in chapter seven. This architecture incorporates a clock with a duty-cycle that can be varied by the control unit. Functional blocks, which due to data dependency have a large execution time variation, signal the control unit upon completion of their task. The completion signals allow the control unit to vary the clock's duty cycle, thus allowing the initiation of a new task.

The last chapter (chapter eight) has conclusions and suggestions for further work and research.

CHAPTER 2

Multiprocessing DSP

2.1. Introduction

Algorithms and programs for real-time signal processing (e.g., tracking radar, sonar systems, image processing and multi-sensor navigation systems), artificial intelligence, weather prediction, biomedical and geophysical applications [1] inherently have a large degree of parallelism. They usually involve a large amount of data and computations (e.g., matrix manipulations - multiplications, inverse, correlations and convolutions) and require fast computation and high throughput (high computation rate). One way to implement this class of algorithms is with a multiprocessor architecture [2, 3, 4, 5, 6, 7, 8]. But multiprocessor architectures have computation latencies which limit the maximum speedup and throughput obtained by N processors operating concurrently with data transfer between them. The speedup and the throughput are limited by:

1) Idle time due to imperfect processor load balancing.

2) Communication latencies:

   - Waiting time caused by long routes and contention for links - data has to pass along too many links from the source processor to the destination processor.
   - Time required to handle and control the data transfer.

3) Processor's computation time wasted on interprocessor communication.

Effective exploitation of the algorithmic parallelism, as well as short paths for data

transfer are essential to achieve a large computational speed-up. Effective use of multiprocessor system depends upon intelligent schedulers and compilers which either partition the algorithm according to the number of PEs and their interconnection topology, or partition the algorithm and determine the number of the PEs and their interconnection topology [9, 10]. Partitioning the algorithm into many tasks and assigning them to different processors should attempt to:

- Minimize the number of the interprocessor data transfer communications.
- Keep the communications localized (short routes).
- Reduce the delays of data transfer between the PEs.
- Improve the processors load balance to reduce the idle times of the processors.

In general, the scheduling problem is NP complete; the scheduling algorithms therefore use heuristics which do not necessarily yield an optimal partitioning with respect to localization and minimal interprocessor communication. Even more, depending on the algorithm, sometimes optimal partitioning may not be good enough, therefore, the interprocessor communication hardware and protocols are vital for achieving a high computing throughput.

An independent interprocessor communication unit designed to handle and control the data transfer between processors, in parallel and concurrently with the computations, relieves the processor from wasting time on interprocessor communication and reduces the time required for data transfers.

Depending on the algorithm and the number of processors, there are two ways to implement the data transfer interconnections between the processors of a multiprocessor system. One implementation is the shared memory used in the CM*[11, 12] and in the BBN [13, 14], and the other implementation is the packet switching used in the connection machine[15] and in the NCUBE[16]. The shared memory interconnections are

composed of either a large common memory or a set of local memories of the processors. Each sample of data is accessed by translating a virtual address to a physical address. Data is transferred to and from the shared memory in one of the following ways:

- Closely coupled through interface controllers.
- Multiple buses or networks (hierarchical clusters).
- Circuit switching for direct data transfer.
- Intermediate circuits for store and forward.
- DMA channels.

In the packet network interconnection, a single byte or data packet is transferred among the processors in one of the following ways:

- Bidirectional buses.
- Shared interface routers.
- DMA channels.
- Hand-shaking.

Interconnection networks which reduce the communication delays and are well suited for general purpose and parallel processing applications [12, 11, 17, 18, 19, 20, 15, 21, 22, 23, 24, 16, 25, 26, 27] have been investigated by several researchers. Studies have shown that for multiprocessor systems containing more than 50 processors, the shared memory interconnection has long delays due to bus contention while the network interconnection which is simpler to implement, reduces communication delays compared to the shared memory and has a good tradeoff between $\frac{topology}{cost}$ and performance of the system. Bus and ring interconnection topologies are cheap to implement but have a limited bandwidth. Mesh and cross-bar interconnection topologies have a high bandwidth but are expensive to implement. The packet network

interconnection combines the advantages of both. It is a closely coupled store-and-forward network that route messages in the form of packets with information about the source, destination and the size of the message. Information (packets) passes through intermediate switches to the destination [28, 29, 30, 31].

The following chapters describe a way to design a Digital Signal Processor (PE), suitable for multiprocessor systems, which incorporates a Processing Unit (PU) and an autonomous interprocessor communication unit (AIO) on the same chip. The processing unit performs the task's computations while the autonomous interprocessor communication unit (AIO) handles the switching circuit and the data transfer between two PEs. Both units operate concurrently and thus eliminate the waste of processing time for handling and controlling data transfer. Data transfer between the PEs is executed through I/O links that have a high rate data transfer capability (≈ 10-20 Mbits/sec) at the cost of some hardware and software overhead. Data transfer between the PU and the AIO is through dual port memory which can be accessed concurrently by the PU and the AIO, thus preventing any interference between the two units.

2.2. Multiprocessor interconnection - background

Many multiprocessor systems like the Cm*[12, 11], The Connection Machine[15], BBN[14], NCUBE[32, 16], Intel iPSC[32] and the Transputer[33, 34, 35, 36] have different interprocessor communication methods for data transfer and protocols.
Here are a few examples of how interprocessor communication is handled in multiprocessor systems:

2.2.1. Butterfly Parallel Processor

The Butterfly Parallel Processor is an MIMD machine composed of processors with memory and a high performance switch interconnecting the processors. The memory of all the Processor Nodes forms the shared memory which is tightly coupled

and can be accessed by each Processor Node. The Processor Node consist of a microprocessor for computations, a Processor Node Controller (PNC) for transmitting and receiving messages, a memory, an I/O bus and an interface to the interconnection switch.

The PNC initiates all messages transmitted over the switch and processes all messages received from the switch. Using memory management, it translates the virtual memory address used by the computing microprocessor into a physical memory address. Thus, the memory of all Processor Nodes appears as a single large global memory to the user. The PNC also provides efficient communication and synchronization between tasks by executing queuing operations in a way similar to the switch operation of a packet switching network.

2.2.2. Cm* multi-microprocessor

The Cm* is a multiprocessor system with a shared memory. The shared memory is not separated from the PEs. Each PE and its local memory are closely coupled; a network of buses give every PE an access to each non-local memory. The system uses hierarchical packet switching structure. An address and/or a data from a PE is always latched into a switching node of the hierarchical buses structure, and the buses are allocated only for the time interval it takes to transfer the address or the data. The architecture of the system combines several PEs into a "cluster" which provides a shared address mapping and routing processor for handling the intercluster communication. The routing of a PE's reference to target memory is transparent to the user and is performed by special levels of addressing mapping mechanisms and buses. The sender will always receive back an acknowledgement or "Return" message containing the data. Addressing within a cluster is translated by one level of mapping mechanism while addressing between clusters is translated by two levels of mapping mechanism. An intercluster message consist of one to eight words. The sender message consists of the

following information: source ID, destination ID, control instruction, address of the data and the data (1-8 words). The returned message consists of destination ID, control instruction and the data. Reading/writing data from/to the non-local target memory is done through DMA by the routing processor. For concurrent operation, the routing processors have queues to store messages and to interface between their hierarchical levels.

2.2.3. The Connection Machine

The Connection Machine architecture provides a very large number of PEs (processor/memory units) connected by a programmable switching packet communication network, that can connect all PEs in any arbitrary pattern. The key component is a VLSI chip which contains 16 PEs and a router unit of the packet switch communication network. The PEs on the chip are connected in an array of 4×4. The router is responsible for routing messages between chips and delivering them to the destination specified by the address. The router communicates with the routers of other chips through an hypercube topology of bidirectional lines. The router can transmit new messages into the network, forward messages between chips and receive and deliver messages to the appropriate PE. The PEs on the chip communicate directly with their four neighbors without the interference of the router. The communication with the router is done by handshaking on the FCFS (first come first serve) basis. A PE initiates a message by sending a packet to its router consisting of an address followed by a "1" followed by the data followed by the parity check. The router accepts messages only if its buffers are not full. This information is then transmitted back to the PE via the router acknowledge flag. The messages will be transferred from the router by FIFO policy, i.e., the message which is at the node the longest time will have highest priority for transmission. When the message reaches its final chip, the local router will deliver it to the appropriate PE by writting it into its memory and notifying the PE that there is a new valid data.

2.2.4. Transputer

The transputer is a 32 bit microcomputer with on-chip RAM memory and four standard communication links. Communication between different transputers is point-to-point synchronized and unbuffered. Data transfer is executed only if both transputers are ready. Each link comprises two unidirectional signal lines that carry data and control information. A message is transmitted as a sequence of single bytes. Each link controller has a data transfer overhead that consists of accepting a pointer to the memory, number of bytes to be transferred and the link identity. Data fetch at the sender and data store at the receiver is done by DMA. The data transfer on the link is independent of the processor. The sending transputer initiates transfer by transmitting a byte of data on the output line. The sender then waits for acknowledgement, which is sent through the input line and which signifies that the receiver is able to receive another byte. No other data will be sent before the arrival of an acknowledgement. Each data packet is 11 bits long including 2 bits of header, 8 bit of data, and one END bit. Acknowledgement packets consist of two bits.

2.2.5. NCUBE

The NCUBE computer is a multiprocessor that incorporates 2^N PEs interconnected as an N-dimensional binary hypercube. Each PE has its own local memory, and N direct communication links with its neighbors. Communication with other PEs is performed via asynchronous DMA operations over N pairs of bidirectional lines. Two registers are associated with each link. One is the address register for the message buffer location in the memory and the other is a count register indicating the number of bytes left to send or receive. There is also a "ready" flag and an "interrupt enable" flag for each link. A data transfer is initiated by the processor after checking its flags and setting the appropriate registers. A message consists of a file of data with four associated fields of control information: source, destination, length of data file (up to 64

Kbytes) and type of data. Some hypercube systems transfer the whole message at once while others (e.g. Intel's iPSC) partition it into smaller packets that are transmitted sequentially. A new message or packet will be transferred only after the receipt of an "acknowledgement" of previous packet from the receiver. After initiation the processor continues with other operations while the link controller completes the transfer operation via DMA. When the link controller is ready for a new operation (after finishing execution of the previous operation) it will set the appropriate flags and will notify the processor by an interrupt.

2.2.6. Summary

The aboved survey shows that there are lots of alternatives to transfer data between the PEs. All the alternatives are dedicated for increasing the performance of the particular multiprocessor system. Shared memory used for data transfer between the PEs is not a single large memory but consists of the PE's local memories. Access to the local memories require a network and switches as in Cm* and BBN. The Connection-Machine uses a complicated programmable switching network for data transfer, while the NCUBE and the transputer use their I/O link as part of the interconnection network. Even though these multiprocessor systems use packet switching, DMA channels and complicated special purpose switches, data transfer in these systems require the processor to lose some computation time and/or memory access time.

2.3. Proposed Processing Element (PE)

2.3.1. Design approach

In a multiprocessor system which incorporates a large number of processors (on the order of fifty or more), transferring data through a network interconnection is simpler and more economical to implement than through a shared memory interconnection.

As was mentioned before, the speedup obtained by N processors, depicted in figure 2.1 which operate concurrently in a multiprocessing system and transfer data between them through any network topology is limited by:

1) Idle time due to imperfect processor load balancing.

2) Waiting time caused by communication latencies in the links and in forwarding data.

3) Processor time dedicated to process data messages and to forward them.

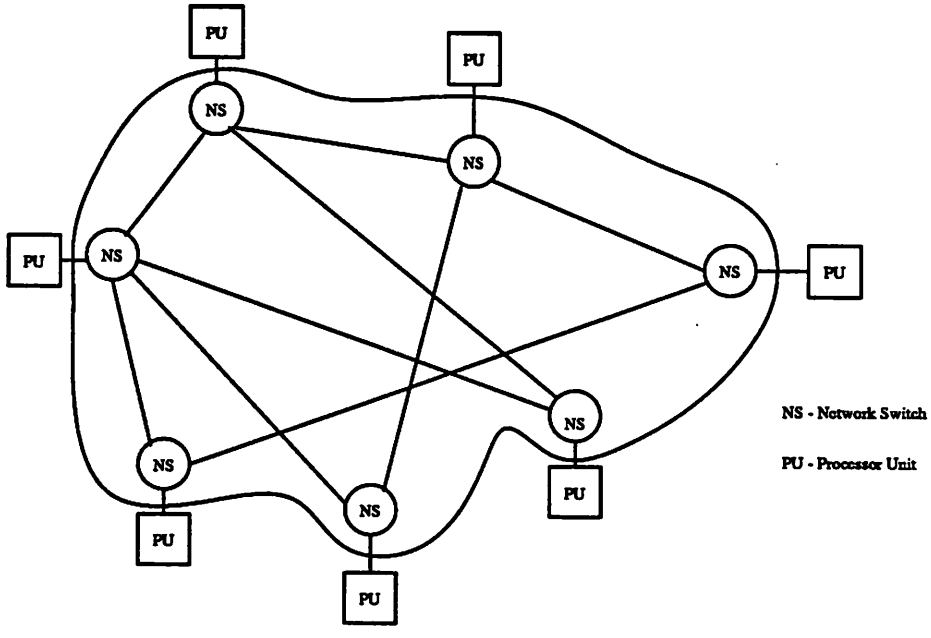


Figure 2.1 - Multiprocessor system

Appropriate processor load balancing with minimal data transfers has to be solved by the scheduler, which partitions the algorithm into tasks and assigns them to different processors so that the transfers are localized and their number minimized. But since the known methods of scheduling do not necessarily yield an optimal partitioning with respect to localization and minimal interprocessor communication, the interprocessor communication hardware and protocols are vital for reducing: 1) The waiting time caused by communication latencies, 2) The processor's time wasted on data transfer

and communication.

Therefore, the hardware of a multiprocessing system that achieves a computation speedup and a higher throughput must have the following features:

- Minimum processor involvement in controlling and handling interprocessor communication, and thus increasing the processor time dedicated to computations.
- Fast and independent interprocessor data transfer which does not interfere with the task computations and is transparent to the user (separation of processing and communication).

Figure 2.2 depicts a multiprocessing system designed with processing elements that have the above features.

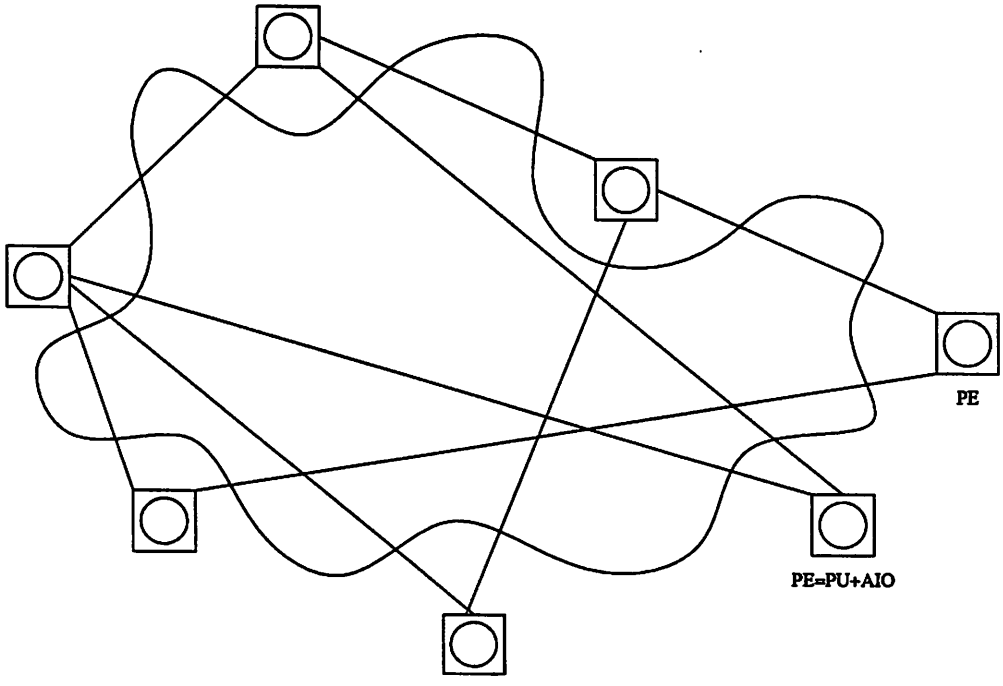


Figure 2.2 - Multiprocessor system with proposed PE

The Processing Element (PE) used in the multiprocessor system of figure 2.2 is depicted in figure 2.3. It incorporates two separate units: the digital signal processing

unit denoted PU (Processing Unit) and the interprocessor communication unit denoted AIO (Autonomous I/O).

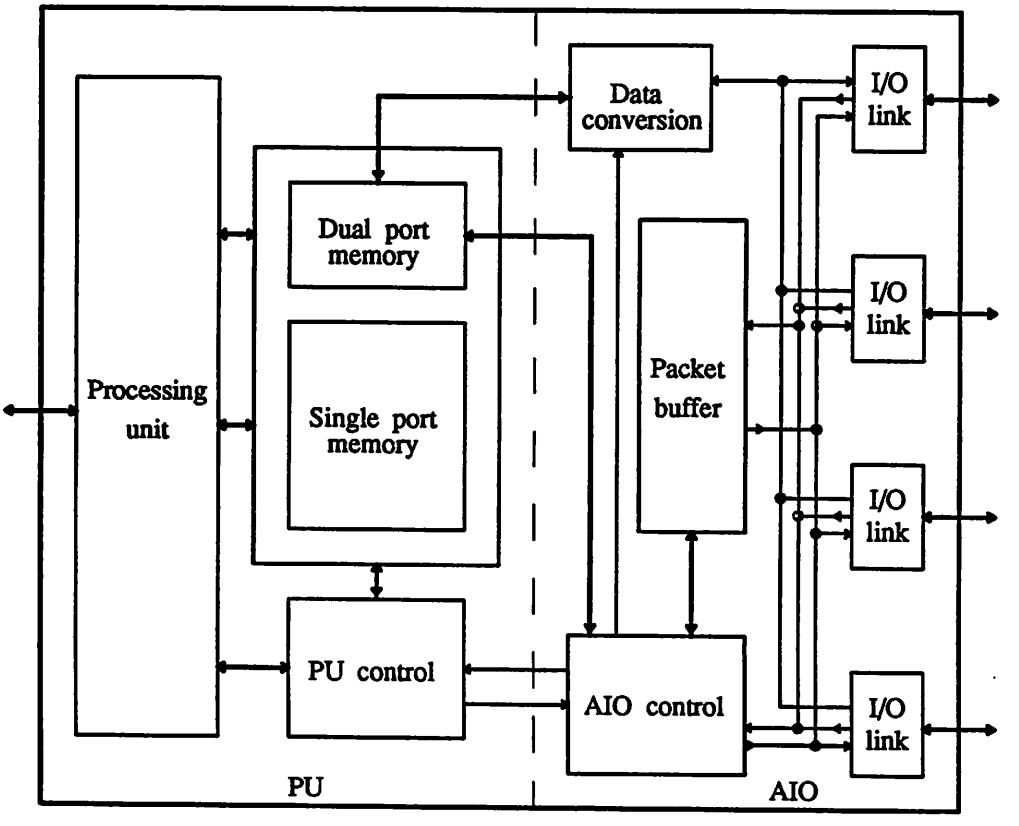


Figure 2.3 - Proposed PE

The processing unit (PU) and the communication unit (AIO) operate independently and concurrently in a way that the data transfer inside the PE between the PU and the AIO and outside the PE between the PEs is transparent to the user. The PU executes the computational part of the task, while the AIO operates either as an autonomous interface for data transfer between the PU and the network or as a network switching node for data transfer between PEs. Handling all the data transfer from the source to the destination by the AIO releases the PU to execute only the computational part of the task. The PE incorporates four pairs of unidirectional I/O links controlled by the AIO which can be used as input and output ports, thus enabling the PE to be embedded in different interconnect topologies of multiprocessor system. When two PEs

require a higher communication BW it is possible to connect them with up to four interconnection links. In such case the AIO will automatically transfer the packet through any free link between the two. When two of the links are input links and two are output links the PE through the AIO control can execute the switch functions in any interconnection network topology such as delta, banyan, cross-bar, mesh, torus, omega, multibus, 4-cube (N-cube) etc. [37, 38, 39, 40, 41]. When in addition the processing power is necessary, the PE could be part of a systolic array processor or of a N-Cube multiprocessing configuration.

The interconnection between the PEs through the I/O links can be done in several ways: half duplex, full duplex or hand-shake. Data transfer can be serial through one line or parallel through many lines, and thus depending on the application, the data bus can be parametrized during the chip fabrication.

Modularity, parametrizibility and expansibilty of the PU and the AIO, dedicated PU implementation according to the application, simple interface between the PU and the AIO and the use of similar protocols for different communication configuration, are the major properties that makes the proposed PE suitable as a macro-cell for many ASICs (Application Specific IC).

The proposed communication between the PEs is established by a bidirectional handshake protocol illustrated in figure 2.4.

When a source PE must transfer data to another PE, its AIO unit will initiate by handshaking a connection with the AIO of the adjacent PE (intermediate node's PE or destination's PE). The initiation is done by formating a control message according to the protocol and sending it through the communication link. When the AIO of the receiving PE is ready to receive the data, it will signal back and the source will send the data packet. Upon completion of the transfer, the receiver's AIO returns another control message signalling the success/failure of the data transfer. A control message which is

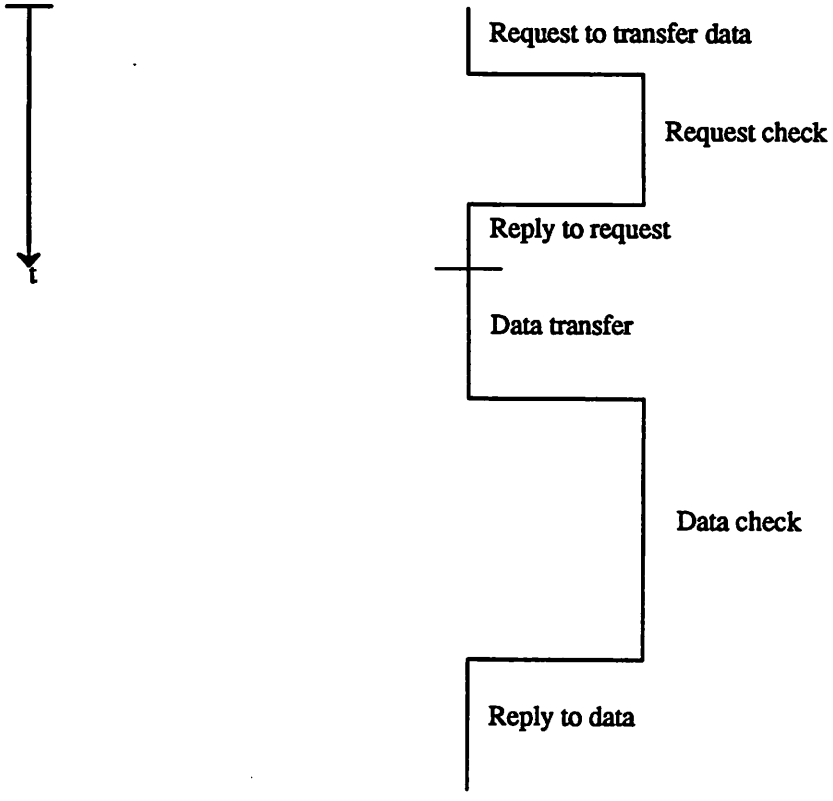


Figure 2.4 - Handshake communication

much smaller than a data packet contains the following fields: synchronization bits, control instruction, source ID, destination ID, length of data, error detection code and end of message (END). Error detection code like parity words or CRC might not be necessary when the data packets are relatively short (about 2Kbits), and the PEs are close to one another in a free EMI (Electro-Magnetic-Interference) environment.

Speeding up the data transfer can be achieved by using virtual-cut-through (VCT) switching, which is a combination of circuit switching and store-and-forward packet switching. The AIO of each intermediate switching network node checks by handshaking, according to the destination's ID, whether the next node in the path toward the destination is free to receive the data. If the next node (toward the destination) is free, the data is forwarded before it has been received and stored completely in the buffers. This switching scheme requires extra processing power from the AIO as will be described

later.

Each packet of data from the source to the destination is routed in a minimal number of hops through specific nodes predetermined and assigned by the scheduler [The scheduler may have as an input a given network topology or may output a preferred one].

Storing the data in the packet buffer is necessary for retransmission in case it arrives at the next PE with errors or is lost (according to a fault tolerance policy explained later).

When there are many packets to handle, the AIO will provide preferential service according to priority tables based upon the algorithm's partition and the tasks assigned to each PE by the scheduler.

Two types of buffers are incorporated in the processing element. One buffer is the *data buffer* for interfacing data transfer between the PU and the network when the PE is a source or destination of the data and the AIO operates as an interface between the PU and the network. The other buffer is the *packet buffer* for data transfer between PEs when the AIO operates as switching node of the network. The data buffer is implemented by a dual port memory, which may be accessed by the PU and the AIO concurrently, thus enabling both units to operate independently without any interference. This dual port memory is in the addressing space of the PU and therefore it is accessed by the PU like any other data in the memory. The other part of the memory which is accessed only by the PU is a regular single port memory. The packet buffer is accessed only by the AIO. This buffer is a temporary storage for data transfer between the PEs. An acknowledgement from the next PE enables the AIO to reuse the storage for new data transfer. The buffer can be implemented by shift registers or single port memories (as will be described later).

Data received by the AIO of the destination PE is stored in predetermined loca-

tions of the dual port memory buffers. Upon successful reception, the AIO will interrupt the PU to notify, through flags/semaphores, that valid data has arrived. When the PE operates as a switching node of the network, received data is stored temporarily in the packet buffers of the output link through which it must be forwarded to the next PE.

2.3.2. Data transfer between PEs

Communication latencies of data transfer are due to: 1) long routes, 2) time to handle message transfers, and 3) waiting time because of the FCFS (firsts-come first-serve) policy that handles and transfers messages in the arrival order and not according to their priority defined by the scheduler.

To decrease these latencies the data transfer procedure is based upon:

1) handshaking protocols.
2) priority of service.
3) routing in minimum number of hops through a virtual- cut-through switching network.

Priority of service and the routes for transferring data in a minimal number of hops are determined by the scheduler during the compilation and the partition of the program.

A handshaking procedure decreases the communication latencies because:

- Data is transferred only when it can be handled by the receiving PE, i.e., there is enough buffer space and data has the priority to be handled, thus freeing the link for transferring the necessary data.
- Saving time in forwarding data to the next PE when the receiving PE operates as a network node by using virtual-cut-through (VCT) switching technique. The receiving PE replies to the sender and at the same time checks whether the next PE is ready to receive the data. If the next PE is ready the

receiving PE operates in VCT mode.

Figure 2.5 depicts the underlying operations executed by a PE in transmitting data according to a data transfer protocol based upon handshaking procedures.

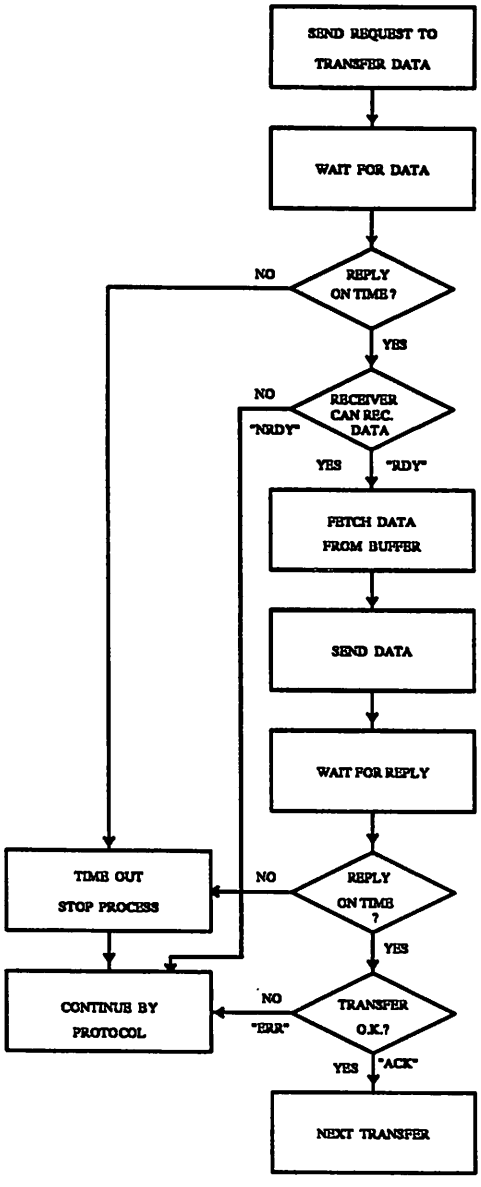


Figure 2.5 - Transmitting PE's operations

The handshaking procedure for data transfer between PEs is always initiated by the sender PE which requests a data transfer and waits for a reply from the receiving PE. If the reply does not arrive during a predetermined time interval or the receiving

PE is not ready to receive the data, the sender PE repeats the request or tries to transfer another message according to the detailed protocol described later. If the receiving PE is ready to receive data, the sender PE fetches data from the buffer, sends it and waits for an acknowledgement. When an acknowledgement is not returned during a time interval or the receiving PE replies that errors were detected in the transferred data, the sender PE retransmits the data or continues according to the protocol described later. When a response from the receiving PE verifies that the data has been received without errors, the sender PE can start a new data transfer.

Figure 2.6 depicts the underlying operations of a receiving PE during a data transfer.

Upon receiving a request to transfer data, the receiving PE (next PE) checks whether it has: 1) enough buffer space available to store the data, and 2) the priority to handle data for a specific destination and whether to receive it if multiple messages arrive from different I/O links. If the data to a specific destination has the priority to be handled and there is enough buffer space available, the AIO of the receiving PE replies back to the AIO of the sender PE that it is ready to receive the data. If the receiving PE does not have the priority to handle the data and/or its buffers are full, its AIO will notify the sender that it is not ready to receive the data.

When the receiving PE notifies the sender PE that it is ready to receive data it sets a time-out watch-dog and waits for the data arrival.

In case the receiving PE is not the final destination of the message, i.e., operates as a network switching node that forwards data to the next PE, the receiving PE initiates the handshaking procedure with the next PE at the same time as it replies to the sender PE that it is ready to receive the data.

The receiving PE operates in one of two modes. If the data can be forwarded to the next PE, the receiving PE operates in a virtual-cut-through mode, i.e., it forwards

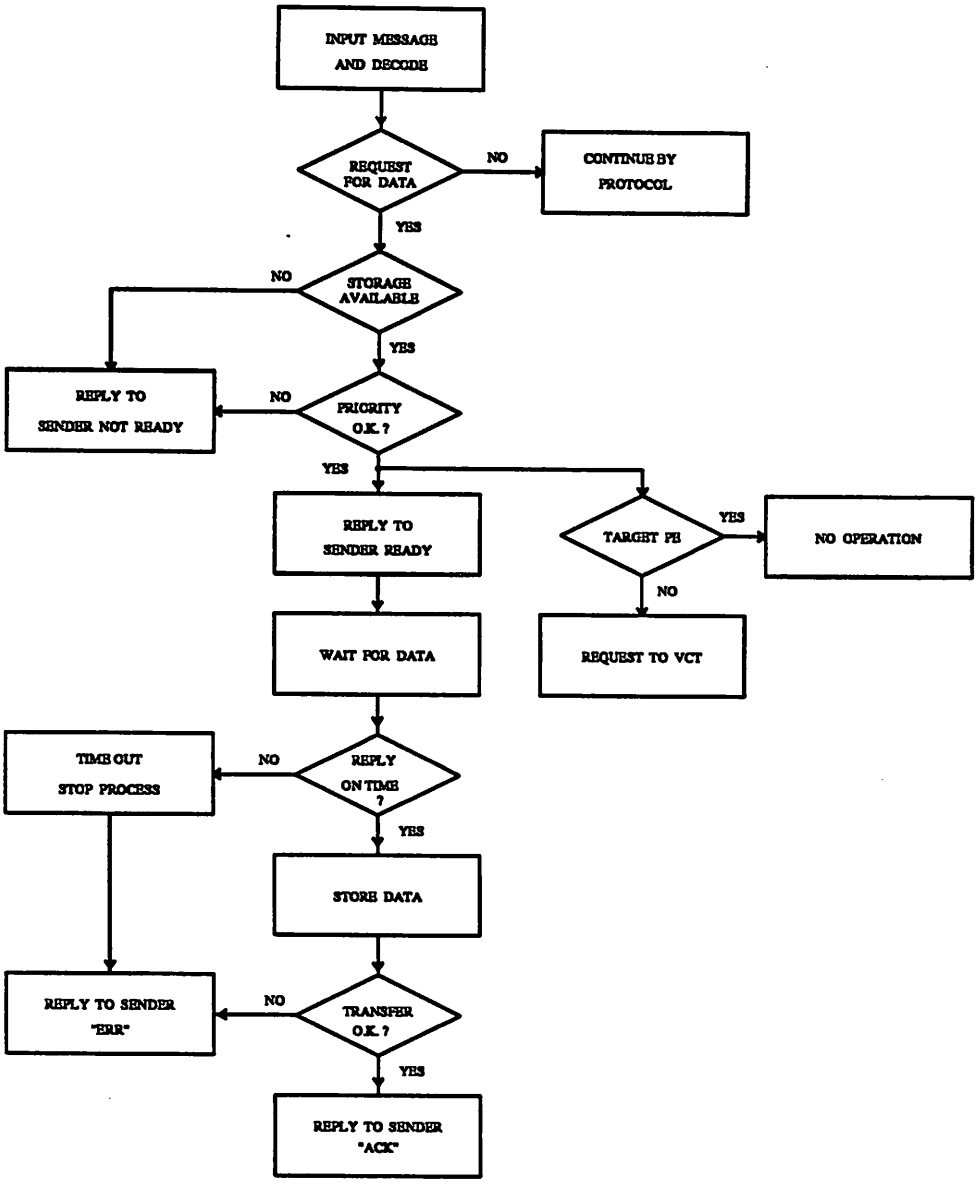


Figure 2.6 - Receiving PE's operations

data before it has been received and stored entirely. If the data cannot be forwarded to the next PE, the receiving PE operates in the usual store-and-forward mode. In both modes, because of the fault tolerance policy, the received data is always stored in the packet buffers. At the end of the data transfer the receiving PE replies to the sender PE with an acknowledgement (ACK) if no errors have been detected in the data, or with an ERR if errors have been detected. If the data has not arrived during the time interval of the watch-dog it replies to the sender PE with an error message.

As was mentioned before, data of higher priority will be handled first. Failing to establish an interprocessor connection between two adjacent PEs or to transfer data between them results in later attempts (up to two more attempts) to try the transfer. Trying to overcome failures in the system, if an interconnection link malfunctions or a data transfer fails, the sender PE will try an alternative route and if that also fails it will return the message back to its sender. Any attempt that results in three failures will result in notification to an operating system or to a human operator. A full detailed description of the protocol is described later in chapter 3.

Acknowledgements are used to avoid unnecessary repetitions of data transfers. There are two types of acknowledgements: hop-by-hop and end-to-end. Hop by hop acknowledgement is part of the handshaking procedure between PEs connected by an I/O link. This acknowledgement is to notify the sender PE, connected through an I/O link, that data has been received without errors. Upon receiving a hop-by-hop acknowledgement, the sender PE operates differently if it is a source PE or a switching network node PE. If the sender PE is an intermediate network switching node, it discards the packet after forwarding it in order to use the buffers for another data transfer. If the sender is the source PE of the data it cannot discard it until an end-to-end acknowledgement arrives. An end-to-end acknowledgement is a message from the destination PE to the source PE notifying that the data have been received without errors. Upon receiving this acknowledgement, the source PE discards the data and frees this buffer space for other data. End-to-end acknowledgement is also timed out by another watch-dog dedicated to this purpose (detailed explanation and implementation appear in chapters 3 and 4).

2.4. Summary of PE's properties

The proposed PE has the following properties:

1) Separate units for computation (PU) and for communication (AIO).

2) Independent and concurrent computation and communication.

3) Fast data transfer by employing virtual-cut-through switching and minimum number of hops.

4) Variable interconnection band-width between processing elements.

5) Macro-cell for "ASIC" implementation:

   - Modular and parametrizable.
   - PU is adjustable to the application.
   - Processor interconnection configuration is adjustable to the application.

6) Independent of network topology - PE can be embedded in any network topology.

7) Simple interface between the PU and the AIO.

8) Different buffers for data transfer between the PU and the AIO and for data transfer between the PEs.

References

1. J.L. Baer, "A survey of some theoretical aspects of multiprocessing," *Computing Surveys*, pp. 31-80, March 1973.

2. Hockney and Jesshope, in *Parallel Computers*, Adam Hilger, 1981.

3. D.J. Kuck, "A survey of parallel machine organization and programming," *Computing Surveys*, pp. 29-59, March 1977.

4. P.H. Enslow, "Multiprocessor organization- A survey," *Computing Surveys*, vol. 9, no. 1, pp. 103-129, Mrach 1977.

5. Y.Parker, in *Multi-microprocessor sytems*, Academic Press, 1983.

6. H.H. Mashburn, "The C.mmp/Hydra project: An architectural overview," in *Comuter structures: Principles and Examples*, pp. 350-370, International student

edition.

7. L. Snyder, "Introduction to the configurable highly parallel computer," *Computer*, pp. 47-56, January 1982.

8. S.Y. Kung, S.C. Lo, S.N. Jean, and J.N. Hwang, "Wavefront array processors-Concept to implementation," *Computer*, pp. 18-32, July 1987.

9. M.C. Chen, "A design methodology for synthesizing parallel algorithms and architectures," *Journal of parallel processing and distributed computing*, pp. 461-491, 1986.

10. M. Thaler, C. Loeffler, and G.S. Moschytz, "Programming, analysis and synthesis of parallel signal processing," *Inst. for signal and information processing*, p. 4, 1987.

11. R.J. Swan, S.H. Fuller, and D.P. Siewiorek, "Cm* A modular multiprocessor," *AFIPS conference proceedings*, pp. 637-644, 1979.

12. R.J. Swan, A. Bechtolsheim, K.W. Lai, and J.K. Ousterhout, "The implementation of the Cm* multi-microprocessor," *AFIPS conference proceedings*, pp. 645-655, 1979.

13. B. Lin and P.S. Tzeng, "Benchmarking multiprocessors: A performance analysis of the BBN Butterfly and the Sequent Balance 8000," *ERL-EECS Depart. UCB*, p. 30, Berkeley, California, May 1986.

14. BBN Laboratories Inc., "BUTTERFLY parallel processor overview," *BBN report no. 6148*, March, 1986.

15. W.D. Hillis, "The Connection Machine," in *Ph.D Dissertation*, MIT, Cambridge, Massachusetts, June, 1985.

16. J.P. Hayes, T.N. Mudge, Q.F. Stout, S. Colley, and J. Palmer, "Architecture of a Hypercube Supercomputer," *International coneference of parallel processing*, pp.

653-660, 1986.

17. M.A. Franklin, D.F. Wann, and W.J. Thomas, "Pin limitation and partitioning of VLSI interconnection networks," *IEEE tarns. on computers*, vol. C-31, no. 11, pp. 1109-1116, November 1982.

18. H.J.Siegel, R.J. McMillen, and P.T. Mueller, "A survey of interconnection methods for parallel processing systems," *AFIPS conference proceedings*, pp. 529-542, 1979.

19. H.J. Siegel and R.J. McMillen, "The multistage cube: A versatile interconnection network," *Computer*, pp. 65-76, December 1981.

20. K.M. Nichols, "Traffic-Specific interconnection topologies for multiprocessors," in *Ph.D. thesis, Department of EECS, University of California* , Bekeley, 1984.

21. C.D. Thompson, "Generalized connection metworks for parallel processor interconnetion," *IEEE Transactions on Computers*, vol. C-27, no. 12, pp. 1119-1125, December 1978.

22. T.Y. Feng, "A survey of interconnection networks," *Computer*, pp. 12-27, December 1981.

23. G.H. Barnes and S.F. Lundstrom, "Design and validation of a connection network for many-processor multiprocessor systems," *Computer*, pp. 31-41, December 1981.

24. L.M. Chen and J. Skalansky, "A parallel multiprocessor architecture for image processing," *Proceedings of ,the internatinal conference of parallel processing*, pp. 185-192, 1984.

25. C.H. Sequin, "Message switching for multi-microprocessors," *proceedings COMPCON spring 80*, pp. 131-137, february 1980.

26. T.N. Mudge, J.P. Hayes, G.D. Buzzard, and D.C. Winsor, "Analysis of multiple-

bus interconnection networks," *Journal of parallel and distributed computing*, vol. 3, pp. 328-343, 1986.

27. W. Stallings, in *Data and computer communications*, Macmillan publishing company, New York, 1985.

28. A.S. Tanenbaum, in *Computer networks*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

29. L. Kleinrock, "Principles and lessons in packet communications," *Proceedings of the IEEE*, vol. 66, no. 11, pp. 1320-1329, November 1978.

30. L.G. Roberts, "The evolution of packet switching," *Proceedings of the IEEE*, vol. 66, no. 11, pp. 1307-1313, November 1978.

31. D.M. Dias and J.R. Jump, "Packet switching interconnection networks for modular systems," *Computer*, pp. 43-53, December 1981.

32. G.B. Doshi and E.V. Munson, "Benchmarking the Cubes," *UCB project report*, Spring 1987.

33. C. Whitby-Strevens, Whitefriars, and L. Mead, "The trasputer," *International symp. on computer architecture*, pp. 292-300, 1985.

34. I. Barron, P. Cavill, and D. May, "Transputer does 5 or more MIPS even when not used in parallel," *Electronics*, pp. 109-115, November 17, 1983.

35. INMOS, in *IMS T414 - Product Data*, Colorado Springs, December 1986.

36. INMOS, in *IMS T800 - Product Data*, Colorado Springs, February 1987.

37. A. Gottlieb and J.T. Schwartz, "Networks and algorithms for very large scale parallel computation," *Computer*, pp. 27-36, January 1982.

38. B.W. Arden and R. Ginosar, "A multi-microcomputer interconnector," *Proceedings of the international conference of parallel processing*, pp. 353-355, 1984.

39. M.R. Samatham and D.K. Pradhan, "A multiprocessor network suitable for single chip vlsi implementation," *Proceeding of the international conference of parallel processing*, pp. 328-337, 1984.

40. K.M. Kavi, E.W. Banios, and B.D. Shriver, "Message repository facility: An architectural model for interprocess communication," *Proceedings of the international conference on parallel processing*, pp. 271-278, 1984.

41. D.A. Reed and D.C. Grunwald, "The performance of multicomputer interconnection networks," *Computer*, pp. 63-73, June 1987.

CHAPTER 3

Communication and protocols

3.1. Introduction

Our interest is in multiprocessor systems containing a large number of PEs which are connected through any arbitrary network. In such systems, the communication methods and the protocol used for data transfer between the PEs are vital to achieve a high throughput. This chapter describes different design alternatives of PE-PE communication and the baseline design choices made for implementing a multiprocessor network systems with the proposed PE. Depending on the application the PE can incorporate different I/O link configurations and/or processing units. Therefore, a particular data transfer mode or technique sometimes is chosen because it has a clear advantage and sometimes is chosen arbitrarily depending on the application, scheduling etc..

Section 3.2 begins with a tutorial of data transfer modes (simplex, half-duplex, full-duplex) and data transfer techniques (handshake, synchronous, asynchronous). A multiprocessing systems containing a large number of PEs, where each PE is limited with the number of its I/O links implies that on the average a certain number of hops is required for data transfer between the PEs. For simple and reliable data transfer in any arbitrary multiprocessor system configuration a synchronous data transfer technique is chosen. To synchronize the clocks of the PEs involved in the data transfer three clock synchronization methods are described. The choice of one of them is arbitrary and depends upon the EMI environment. Fast data transfer with a minimum delay is necessary to obtain a high throughput. Store-and-forward switching and virtual-cut-through

(VCT) switching in which an intermediate node of the network forwards data to the next one before it has been received entirely, are compared and analyzed. The comparison and the analysis show clearly that the VCT is faster and therefore this method is chosen. Broadcasting data in a multiprocessing system which transfers data by handshaking is difficult to implement. Three methods of data broadcasting are described. Choosing any one of them is arbitrary and depends upon the network configuration and the application.

The type and the formats of exchanging messages between the PEs are very important for correct, complete and simple interaction among them. Section 3.3 describes the chosen protocol and its advantages. To decrease communication latencies a VCT switching mode and an acknowledgement handshaking protocol is chosen. Flow control in the network is obtained by executing the data transfer only if the next PE in the predetermined route can receive it. To avoid network congestion and to detect errors in data transfers an hop-by-hop and end-to-end acknowledgement policy is chosen. Searching analysis is used to verify that the chosen protocol is free of deadlocks, unspecified receptions, non executable interactions and ambiguities.

The message formats described in section 3.4 are based upon the basic format of synchronous data transfer technique described in section 3.2 and upon the chosen protocol described in section 3.3. To increase the message transfer rate some ways of decreasing the message header's overhead are described. Short ID for identifying the source and the destination PEs is one example and the error detection code is another one.

Section 3.5 includes the reasons for choosing four I/O links and a description of three different I/O configurations. The I/O configurations are parametrizable and adaptable to different data transfer techniques, system applications and architecture implementations. The protocols designed in section 3.3 are adjusted to the I/O configuration

and implementation.

3.2. Data transfer techniques

Computer architecture and communication books[1, 2, 3, 4, 5, 6] describe the basic common ways of communication and data transfer techniques between computers themselves and between computers and I/O devices. The next two paragraphs introduce and explain briefly these communication modes and data transfer techniques.

3.2.1. Communication modes

The communication between two processing elements connected through I/O links can be done in one of the following modes: simplex, half duplex, or full duplex.

A simplex link transfers information in one direction only. This mode is seldom used in data communications because the receiver cannot communicate with the transmitter to indicate the occurrence of errors.

Half-duplex transmission system is one that is capable of transmitting in both directions but the data can be transmitted in only one direction at a time. In this mode a pair of wires (signal and ground) is required for proper operation.

Full-duplex transmission can send and receive data in both directions simultaneously. This can be achieved by a four-wire link, where a different pair of wires is dedicated to transmission in each direction. A common wire used by both processors reduces the number of the required wires to three. Alternatively, a two-wire circuit can support full-duplex communication if the frequency spectrum is subdivided into two nonoverlapping frequency bands, one for transmit channel and the other for receive channel.

3.2.2. Data transfer techniques

In each of the communication modes data transfer between the processing elements can be done in one of the following methods: handshaking, synchronous or

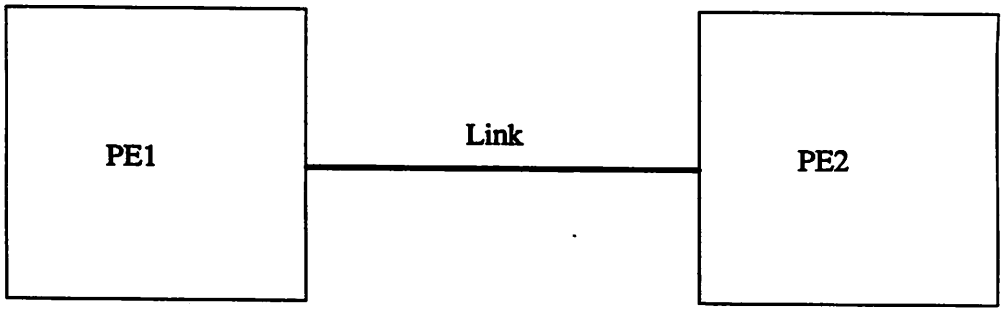


Figure 3.1 - I/O link

asynchronous.

Handshaking data transfer technique

The basic two-wire handshaking method depicted in figures 3.2 and 3.3, incorporates two control wires and one data bus (contains any number of lines). One control line is in the same direction as the data flow in the bus from source PE to destination PE. This line informs the destination PE whether there is valid data on the bus. The other control line is in the direction from destination PE to source PE and informs the source PE whether it can accept a new sample of data. The control sequence during the transfer depends on the unit that initiates the transfer.

Figure 3.2 shows the data transfer procedure when it is initiated by the source PE. The source initiates the transfer by placing a sample of data on the data bus and activating the "data valid" signal. "Data received" signal is activated by the destination after it has received the data. This signal deactivates the "data valid" signal which deactivates the "data received" signal and sets the data bus to be idle and ready for a new sample of data.

Figure 3.3 shows the data transfer procedure when it is initiated by the destination PE. The destination initiates the transfer by activating the "ready for data" signal. Detecting this signal the source places the data on the data bus and activates the "data valid" signal. After the destination has received the data it deactivates its "ready for

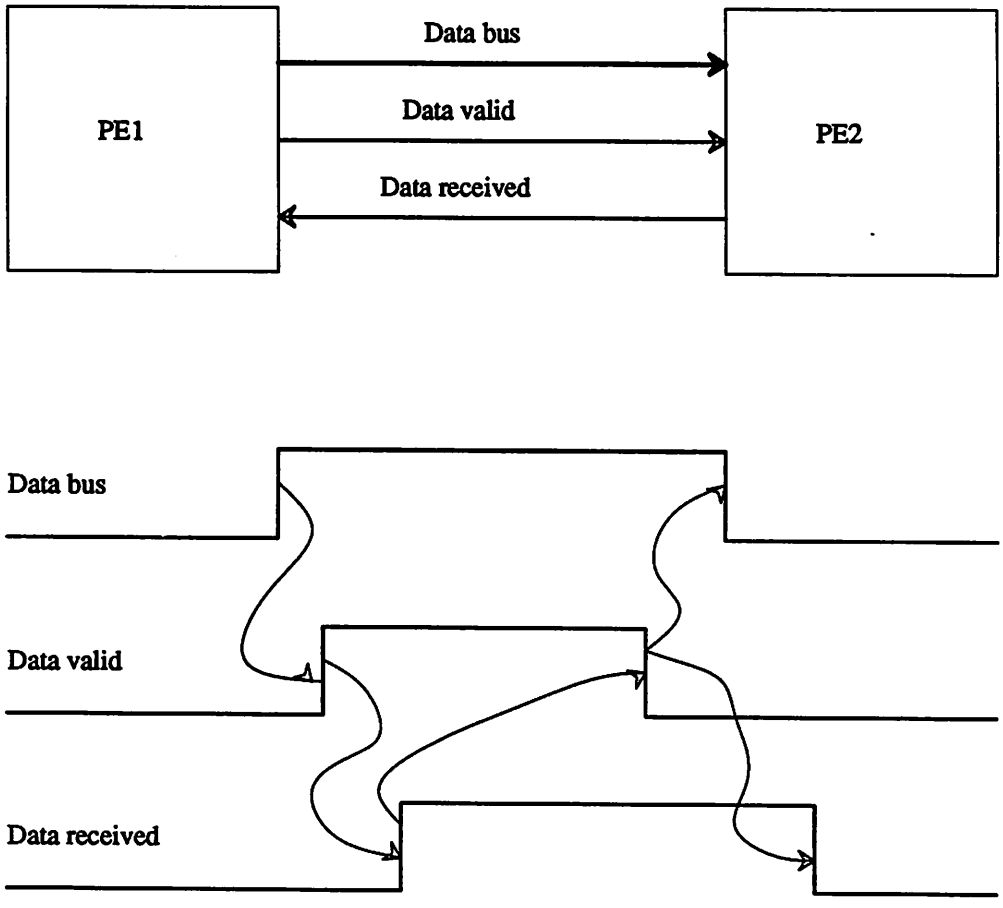


Figure 3.2 - Handshaking initiated by the source

data" signal which deactivates the "valid data" signal and sets the data bus to be idle and ready for a new sample of data.

Asynchronous data transfer technique

The serial asynchronous data transfer technique employs special bits inserted at the beginning and the end of the data. In this technique, each character (word) depicted in figure 3.4 consists of three parts: a start bit, the data bits, and the stop bits.

The receiver knows the transfer rate and the number of information bits to expect. When there is no data transmitted the link is idle in the "1" state. The receiver detects the "start" bit when the link goes from "1" to "0" and synchronizes the time intervals for receiving the data. After they have been transferred one or two "stop" bits always in the

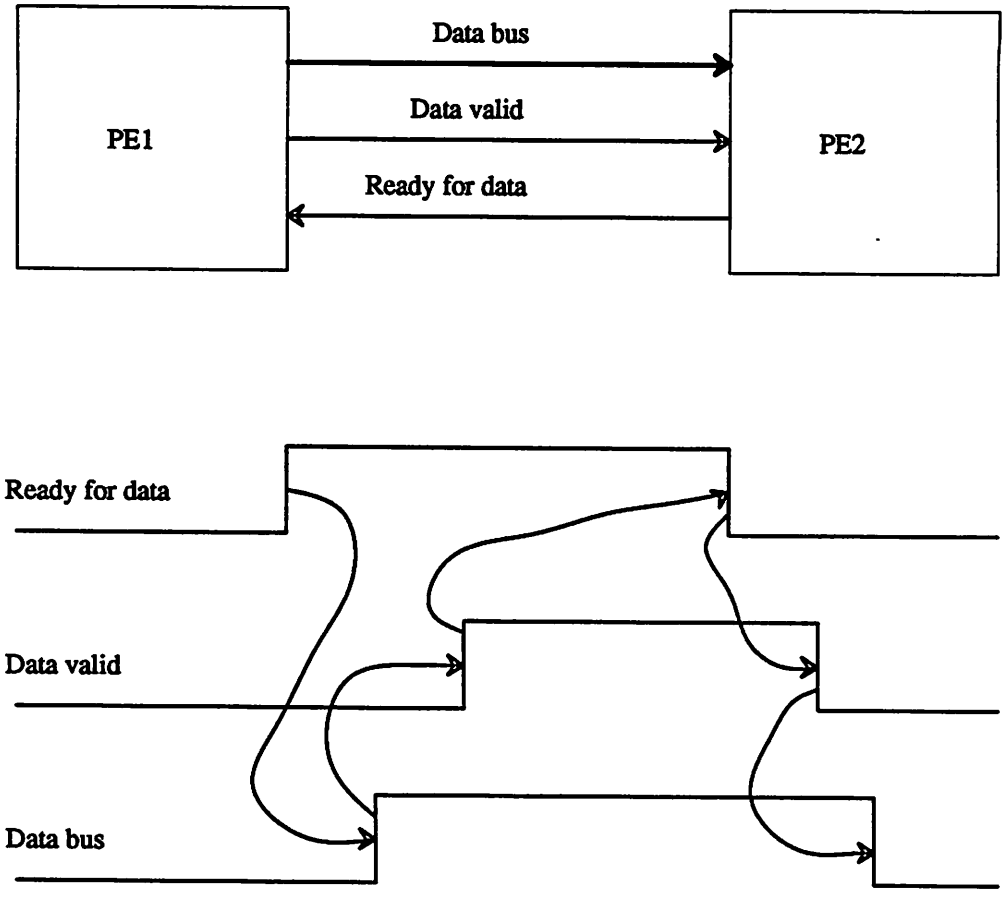


Figure 3.3 - Handshaking initiated by the destination

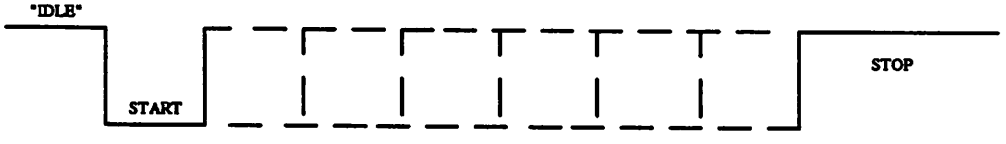


Figure 3.4 - Asynchronous data transfer

"1" state are added and the link goes to the idle state.

The asynchronous data transfer technique has the following properties:

- Larger overhead in message protocols. Each transfer of an eight bit character has an overhead of 3 bits (start and stop bits).
- Synchronization at the beginning of each character transfer.

- More complicated hardware.
- No timing problems between the transmitter and the receiver.

Synchronous data transfer

The serial synchronous data transfer technique transmits blocks of data without "start" and "stop" bits. To prevent timing drifts between the transmitter and the receiver, their clocks are synchronized through synchronization information (control bits) embedded in the message. Each block of data begins and ends with control information as depicted in figure 3.5.

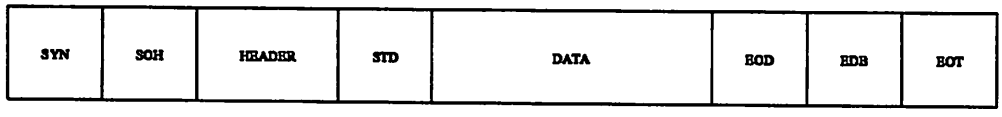


Figure 3.5 - Synchronous data transfer

The control information at the beginning of the message is the *preamble* which contains the following data:

- SYN - Establishes and maintains synchronization on the link.
- SOH - Start of header - beginning of message.
- Header - Information about the source, destination, block size, message's ID etc..
- STD - Start of data text.

The control information at the end of the message is the *postamble* which contains the following data:

- EOD - End of data text.
- EDB - Error detection bits.
- EOT - End of message.

The receiver detects the SYN bits and synchronizes its clock to read the message properly. The preamble and the postamble control bits, which have a unique bit pattern, provide the receiver with information about the data and enable the receiver to check its correctness. To distinguish between data and control bits, bit stuffing is necessary in the data information whenever it contains the same bit patterns as the control bits.

The synchronous data transfer technique has the following properties:

- Less overhead in transfer of large data blocks.
- Simpler to implement and requires less complicated hardware.
- Requires synchronization information.

3.2.3. Clock synchronization

In the asynchronous data transfer technique the clocks of the processors are very close. In this mode the phase between the clocks is synchronized by the start bit. Therefore, the following question must be asked: What is the allowable drift between the clocks of the transmitter and the receiver ? To answer this question it is required to check how many bits is it possible to transfer until the clocks have a phase delay of 180 degrees.

Assume there is a drift of Δ f between the clock frequency f_1 of PE_1 and the clock frequency f_2 of PE_2 which is measured by x in [ppm] (parts per million), i.e., $\Delta f = f_1 - f_2$, or $f_2 = f_1(1+x10^{-6})$.

The difference in the cycle time ΔT is given by:

$$\Delta T = \frac{1}{f_1} - \frac{1}{f_2} = \frac{1}{f_1} - \frac{1}{f_1(1+x\,10^{-6})} = \frac{x\,10^{-6}}{(1+x\,10^{-6})f_1}$$

And the number of bits to be transferred without a clock synchronization error is:

$$n = \frac{\frac{T}{2}}{\Delta T} = \frac{\frac{1}{2}f_1}{\frac{x\,10^{-6}}{(1+x\,10^{-6})f_1}} = \frac{1+x\,10^{-6}}{2x\,10^{-6}}$$

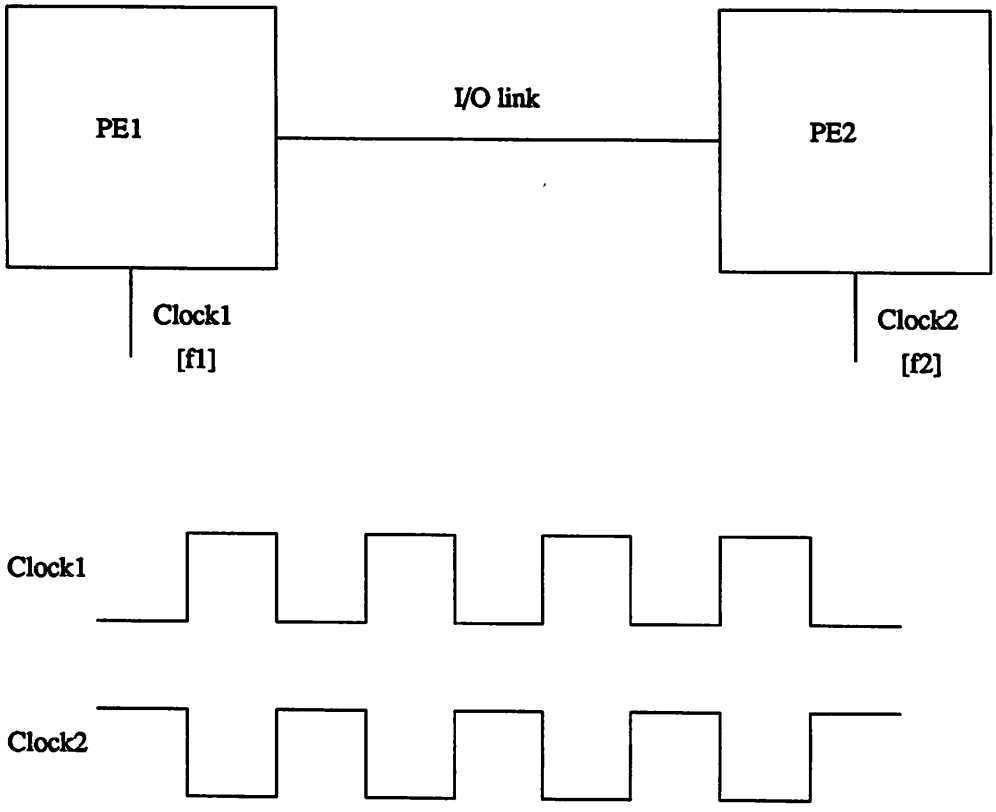


Figure 3.6 - Clock synchronization

Examples

For x=100 [ppm] = 0.0001 => $n=\frac{1.0001}{0.0002}$=5000 [bits]

For x=500 [ppm] = 0.0005 => $n=\frac{1.0005}{0.001}$=1000 [bits]

For x=1000 [ppm] = 0.001 => $n=\frac{1.001}{0.002}$=500 [bits]

As might be expected, these examples illustrate that more bits can be transferred without an error when the drift between the clocks is smaller.

In the synchronous data transfer technique, the clocks of the processor involved in the data transfer have to be synchronized. The "SYN" control bits enable the receiver to synchronize its clock to the transmitter clock, thus enabling the transfer of a large block of data. There are many ways to synchronize the two clocks: PLL (phased lock loop),

bit synchronizer and high frequency shift register.

The PLL (phased lock loop) depicted in figure 3.7 is a feedback loop comprised of three basic blocks: a phase comparator, low pass filter and voltage controlled oscillator (VCO).

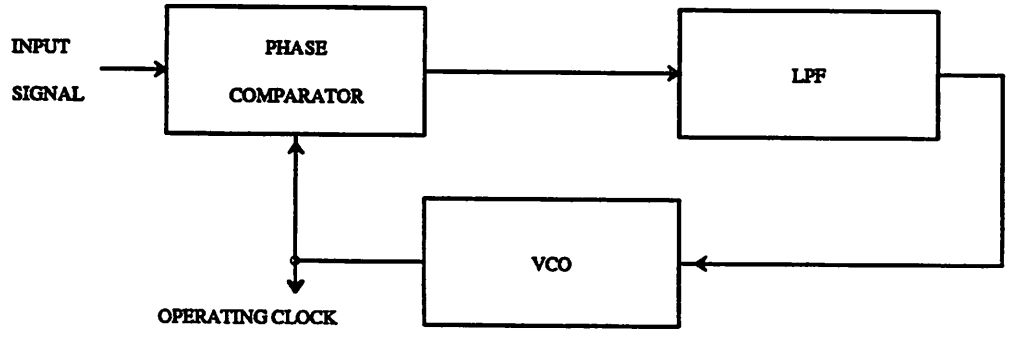


Figure 3.7 - Phased Lock Loop system

The phase comparator compares the phase and the frequency of the input signal with the VCO frequency and generates an error voltage. The error voltage is fed back to the VCO which synchronizes and locks its frequency to the input signal. Once it is locked, the VCO's frequency is the same as the input frequency except for a phase difference.

The bit synchronizer is a free running oscillator which constantly detects transitions in the input signal and synchronizes its frequency to them. Synchronization takes place after detecting a fixed number of transitions. This circuit also contains a PLL and is very efficient for low signal-to-noise (S/N) ratio signals.

A high frequency shift register is a circuit that samples and latches the input signal with a high frequency input clock (more than ten times the operating clock frequency) into a shift register and checks the number of "1"'s and the number of "0"'s during each operating clock cycle. When a transition occurs in the input signal and there is a difference in the number of "1" and "0", it is a synchronization error. This error is fed back to a VCO for correcting and synchronizing the receiver's clock.

If the input clocks to the PEs are crystal oscillators with high frequency and their operating frequencies are sufficiently close, the use of shift registers for synchronization is adequate. If the input frequency is not much higher than the operating frequency, the PLL system is required. The bit synchronizer is required only in very noisy environments or in systems with low input data frequency.

3.2.4. Virtual - cut - through switching

In chapter 2 and in section 3.1 it was mentioned that it is possible to enhance the data transfer between the source and the destination if the AIO forwards the message to the next node in the path before it has been received completely. In principle this switching system operates as a combination of circuit switching and store-and-forward packet switching. When a control message arrives to a network switching node the AIO decodes the address, looks up in tables for the next link to be used for forwarding the message (preassigned path) and checks if buffer space is available and if the forwarding link is free. If the forwarding link is free, while reponding to the sender PE that it is ready to receive a packet, the AIO tries to establish a connection with the next PE by forwarding the control message to it. If the connection is granted the AIO will begin to forward the data message to next node before it has been received completely. If the connection is not granted the message is not transferred. In both cases, due to fault tolerance and acknowledgement policy the data is always buffered like in any packet switching system.

Figures 3.8a and 3.8b depict the delay procured in transferring fixed size data packets between two network nodes of a packet switching system and a virtual-cut-through switching system. The packets arrival rates depicted in figure 3.8b is higher than the rate depicted in figure 3.8a.

In both cases, whether the packet arrival rate is high or low, the network node of a virtual-cut-through network transfers all the packets with a smaller delay equal to the

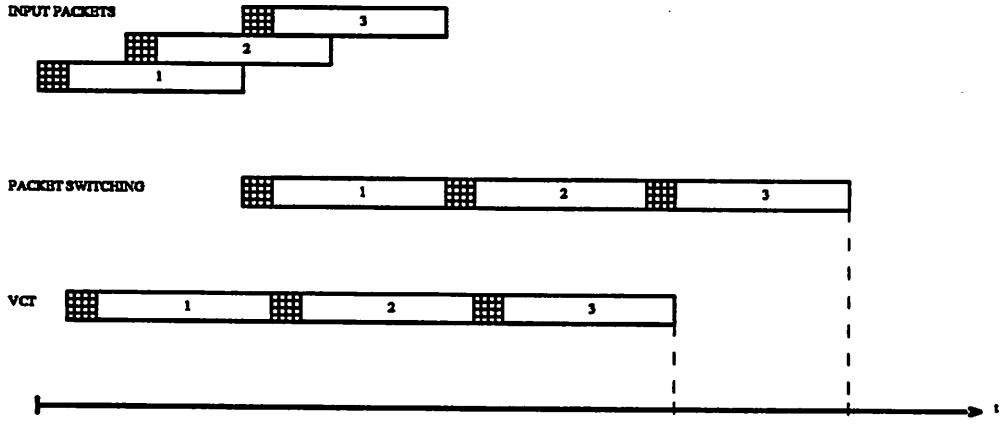


Figure 3.8a - Transfer delay - low rate fixed size packet arrival

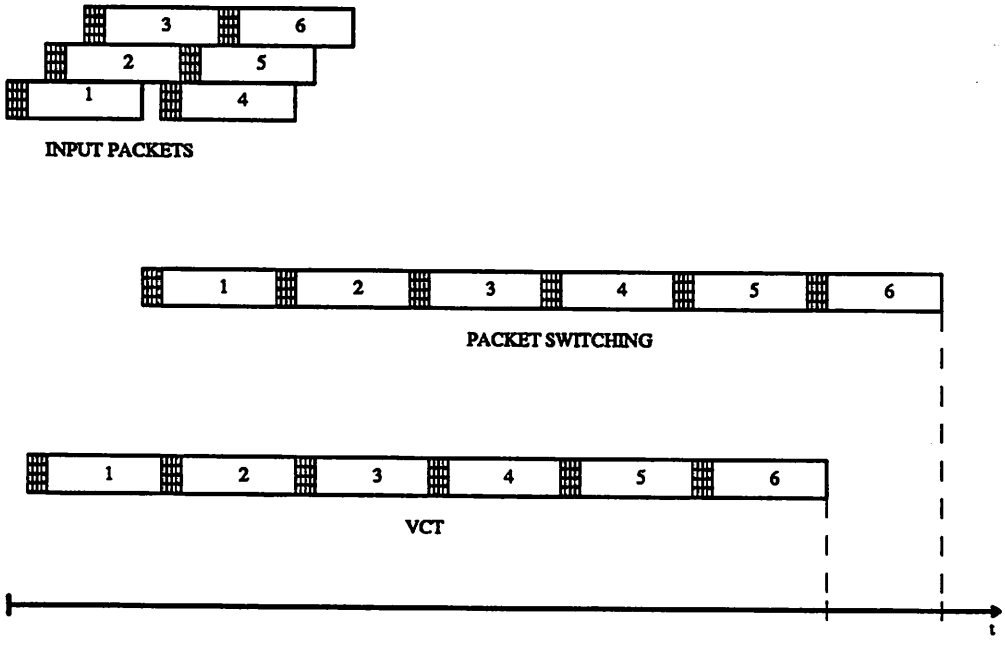


Figure 3.8b - Transfer delay - high rate fixed size packet arrival

time required to transfer the data of one packet.

Figure 3.9 depicts the delays in transferring different sized data packets with different arrival rates in both switching systems.

As before, the virtual-cut-through switching system yields a smaller transfer delay equal to the time required to transfer the data of the largest size packet.

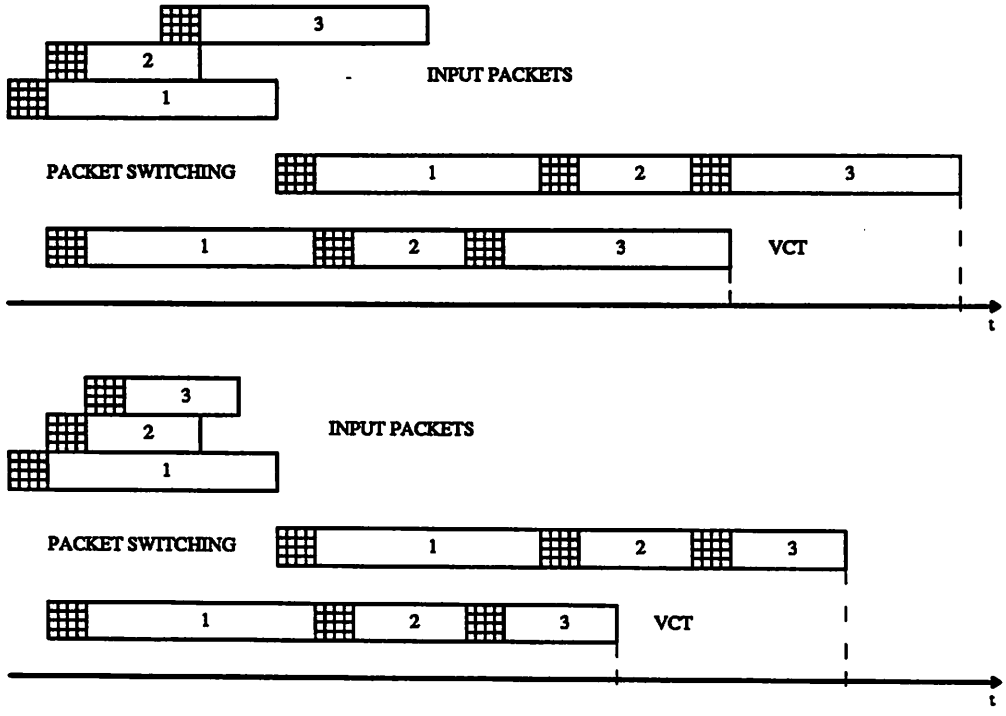


Figure 3.9 - Transfer delays - variable sized packets

If there are no transfer errors and the links are free to transfer data ("best case"), the decrease in the delay to transfer packets between two nodes of of the virtual-cut-through switching system increases linearly with the number of network nodes that the packets of an information message have to pass from the source PE to the destination PE. Figure 3.10 depicts the difference in the time that it takes for a message, consisting of three packets, to travel from the source PE, through two network nodes up to the destination PE.

It is possible to compare the "best case" throughput and the network delay of the virtual-cut-through switching system and the store-and-forward packet switching system for packets of the same length.

Denoting

* T - time required to transfer packet of data.

* t - time required to transfer the message header.

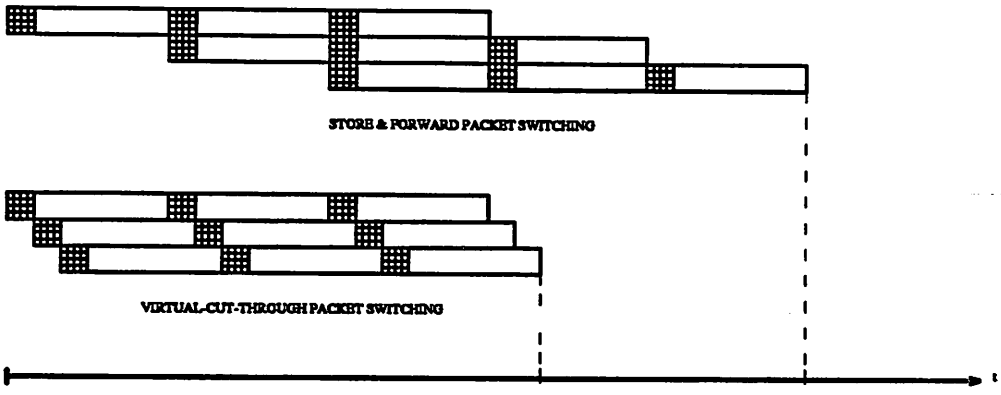


Figure 3.10 - Transfer delays between source and destination

* n - number of packets in a message.

* m - number hops between source and destination.

* T+t - time required to transfer data packet and a header.

Packet switching

$$\text{Throughput}=\frac{T}{T+t}$$

$$\text{Delay}=(n+m-1)T+(n+m-1)t$$

Virtual cut through switching

$$\text{Throughput}=\frac{T}{T+t}$$

$$\text{Delay}=nT+(n+m-1)t$$

This comparison shows that in the "best case", when there are no errors in data transfer and no queueing delays (each PE is always ready to receive and forward messages), transferring a message of n packets through a virtual-cut-through switching system has a smaller delay (higher throughput) when routing the message from the source PE to the destination PE requires more hops (m is large). The decrease in the average delay of the "best case" is (m-1)T.

Using the proposed handshaking protocol described in section 3.3 enables the receiving PE to establish a connection with the next PE while replying to the sender PE

that it is ready to receive the data. Therefore, less data is stored before it is started to be forwarded and the delay time is smaller. Figure 3.11 depicts this property.

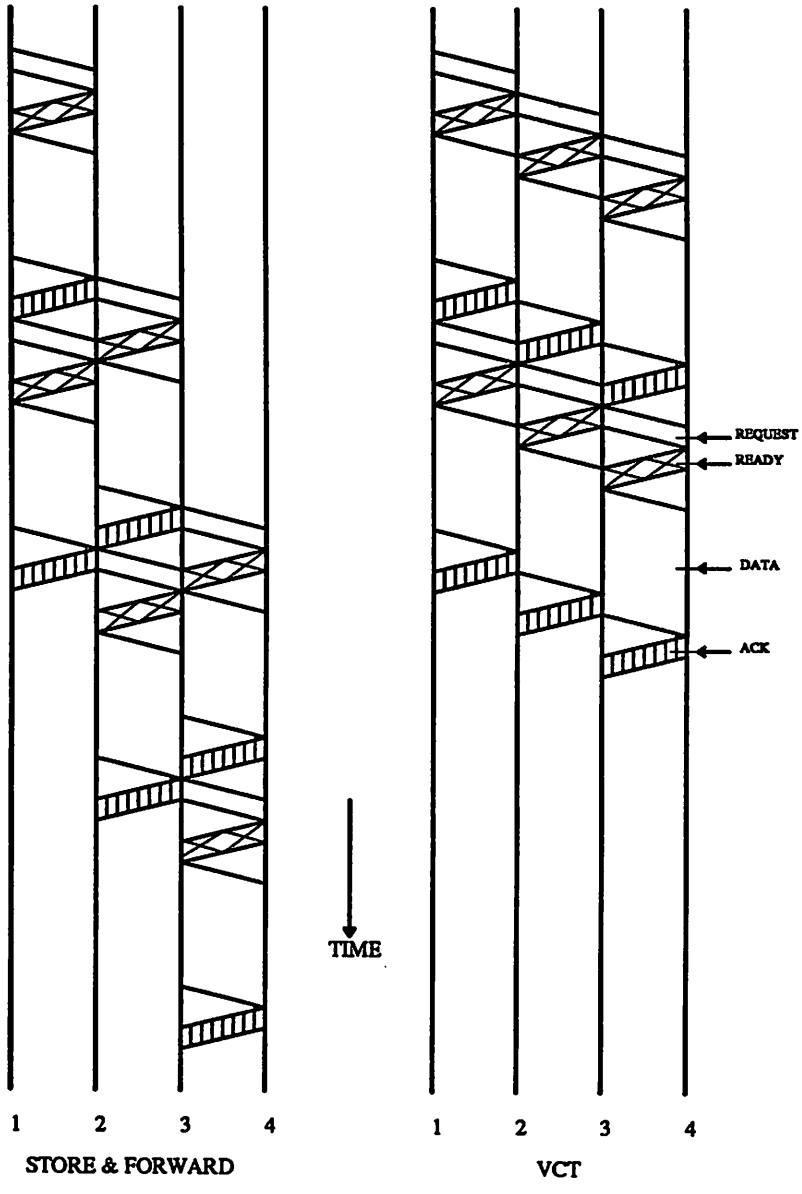


Figure 3.11 - Handshaking data transfer

Kermani and Kleinrock in[7] and Hammond and O'Reilly in[6] analyzed the virtual cut through switching system by using M/M/1 queueing theory models and the following assumptions:

- Poisson distribution of message arrival.
- Exponentially distribution of message length.
- Infinite buffer size.
- Deterministic routing.
- Balanced network utilization.
- Negligible propagation delay.
- No errors due to noise.

They showed that the average delay to send a message through a balanced network is:

$$Averagedelay = \frac{(n+m-1)}{(1-\rho)}(T+t)-(m-1)(1-\rho)T \qquad (3.1)$$

Where:

- n - number of packets of the message
- m - average number of hops
- T - time to transfer data of a packet
- t - time to transfer header of a packet
- ρ - utilization of each link (equal to all links)

The first term of the average delay is due to average packet switching delay while the second term is the improvement of the average delay due to the virtual-cut-through feature. This result agrees with the result shown before but it also takes into account the probability (1-ρ) that the outgoing link is free. In a noisy environment, the probability that the outgoing link is free has to be multiplied by (1-P_e), where P_e is the probability that an error occurred.

The result above implies the following conclusions (relative to the packet switching):

1) When the average number of hops increases, the average number of forwarding

messages without buffering also increases, which means that the average delay decreases.

2) More localized data transfer between adjacent PEs and less data transfer between non adjacent PEs yields more messages transfer without buffering.

3) VCT switching systems have smaller average network delay than store-and-forward packet switching.

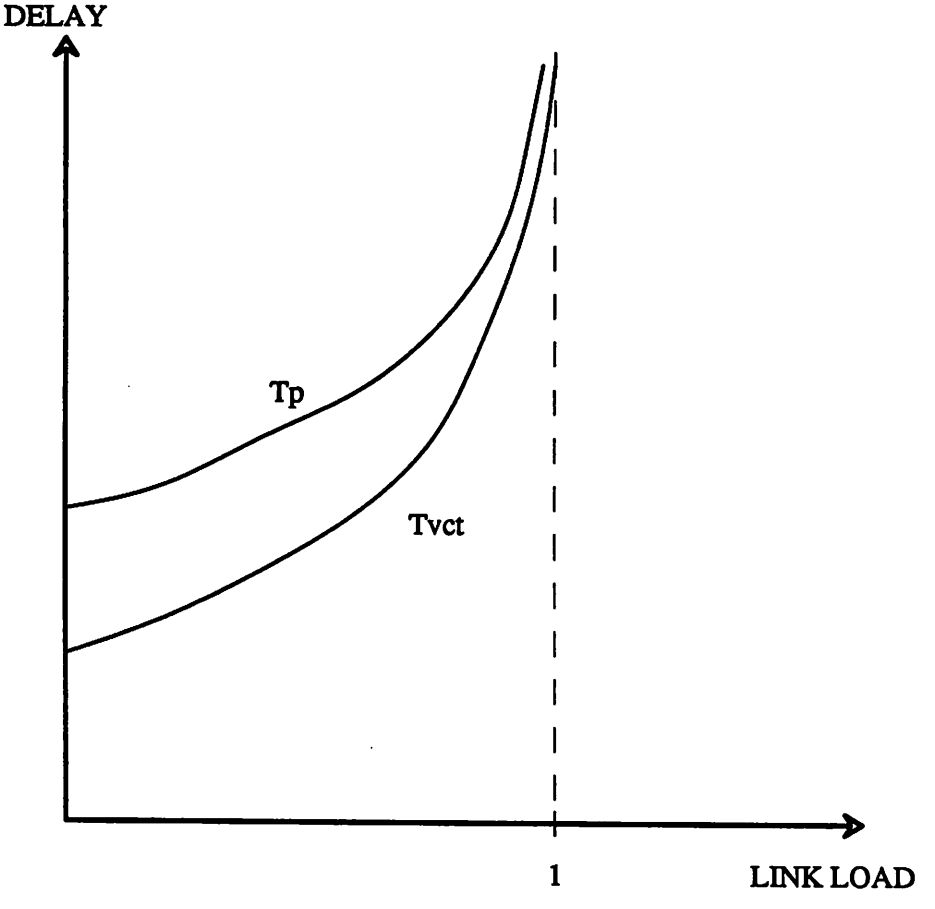


Figure 3.12 - Delay vs. link load

4) For the same network delay, the virtual cut through switching system can transfer more packets than the packet switching system and the difference is greater when the network is lightly loaded.

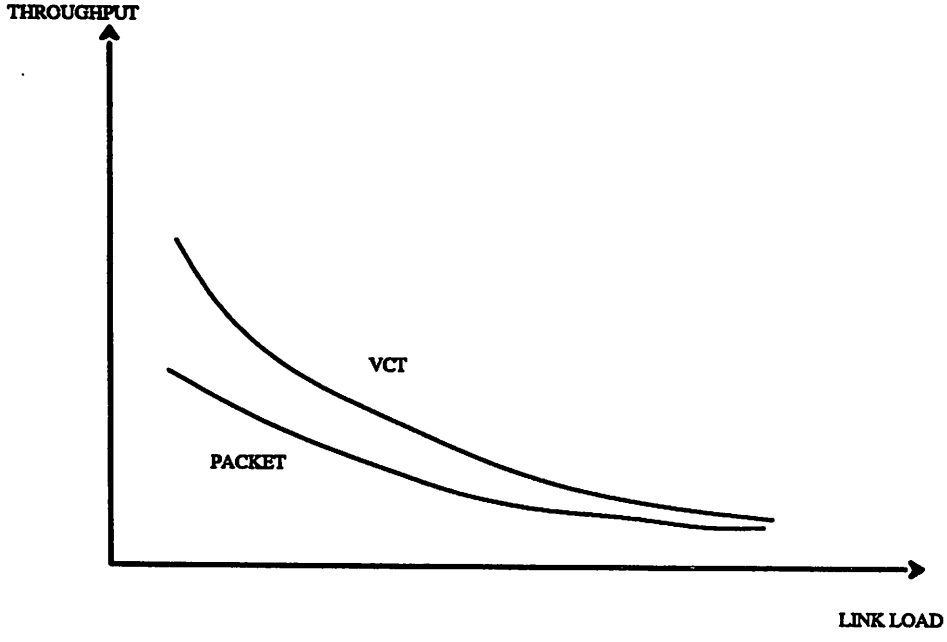


Figure 3.13 - Throughput vs. link load

5) VCT switching systems require less buffer storage than packet switching store and forward systems.

6) VCT decreases the buffer size required in each node.

7) VCT implementation is more complicated and requires more processing power.

As will be explained here the assumptions made earlier in the section for the virtual-cut-through network analysis match the requirements of a multiprocessor system derived in chapter 2.

- Finite buffer size has small effect on the calculated average delay as was shown in [7].
- Deterministic routing which is used for transferring messages in a minimum number of hops is also one of the analysis assumptions.
- Balanced network utilization and PE load, as well as minimizing data transfer imposed on the scheduler (which partitions the algorithm and assigns the tasks to different PEs) are essential for obtaining a large reduction in the average data

transfer delays.

- Low utilization of the links is equivalent to the demand of localized data transfer between the PEs imposed on the scheduler. Minimizing the number of hops reduces the advantage of the VCT (since the reduction in the average delay of the data transfer is relative to the number of hops) but localizes the data transfers.

The protocol's characteristics and properties required to implement the processing element (PE), described later in this chapter (3.4.1) and in chapter 4, also correspond to these assumptions.

3.2.5. Data broadcast

In many applications of multiprocessor systems, it is necessary to broadcast data to many processing elements (e.g. image processing, biomedical, etc.). A multiprocessor system based upon point-to-point interconnection network is not the best way to implement a broadcast feature because the data and the acknowledgement have to ripple through the processing elements. The acknowledgement is needed for the following reasons: 1) to make sure that all the PEs received the data correctly, and 2) to use the same protocol. A more efficient way to implement a broadcast capability is to connect the processing elements to a common bus through which messages and their acknowledgement are transferred. In such an implementation a message is broadcast in parallel to all the receiving PEs but their acknowledgement response is returned to the sender PE in serial. One way to implement it is depicted in figure 3.14.

The sender PE and the receiving PEs are connected through a pair of unidirectional interconnection links. Request to transfer data as well as data itself is transmitted on one link in parallel to all the receiving PEs, while their response to the sender is transmitted on the other link in serial. Every PE can be the sender PE. Response collisions on the bus are avoided by using one of the following schemes:

TDM - Time Division Multiplexer

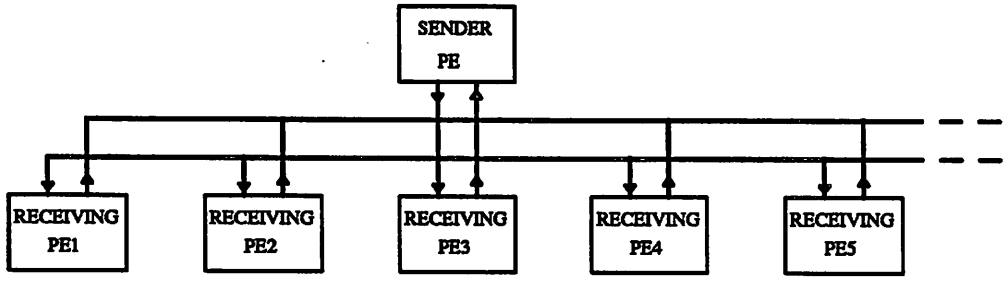


Figure 3.14 - Two bus broadcast implementation

In this scheme each receiving PE responds to the sender PE during a predetermined time slot (Δ_i) relative to the message. Receiving PE number 1 responds during time slot Δ_1, receiving PE number 2 responds during time slot Δ_2 and so on. Since the receiving PEs synchronize their operation to the sender, a response collision is avoided.

Sequential response

In this scheme, which is similar to TDM, each receiving PE responds one after the other in a predetermined sequence. A receiving PEs "listens" to the bus, checks the IDs of the responding PEs and responds after the one with the preceding ID number. To avoid unlimited waiting time, if one of the receiving PEs is malfunctioning, each PE has a "watch-dog" which times out at the maximum waiting time.

CSMA

In this scheme each receiving PE "listens" to the bus to check if it is free to transfer data. If it is free, it sets its response on the bus; if not, it waits. If two receiving PEs respond at exactly the same time, a collision occurs and both responses are corrupted. Since the receiving PEs can also "listen" to the bus they can detect the collision and retransmit the response after some predetermined or random time (like in Ethernet).

Another way to implement broadcast messages is depicted in figure 3.15.

Data is transferred in parallel to all the receiving PEs but their response is transferred in serial from one receiving PE to the successive one through all the receiving PEs up to

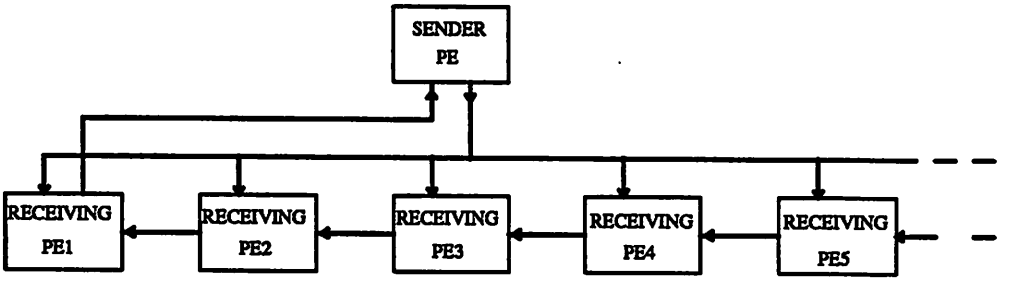


Figure 3.15 - Parallel broadcast serial acknowledgement

the sender. Each receiving PE attaches its response (RDY/NRDY or ACK/ERR) to the response of the others by setting or resetting the corresponding bits in the response message. For example, PE_6 sends its response to PE_5 that attaches it to its own response and sends it to PE_4 and so on up to sender PE. This responding scheme (propagation of acknowledgement) does not take longer than the previous one but it is simpler to implement and fits the acknowledgement handshaking protocol better. Once the network connection is set only one specific PE can be the sender PE. A "watch-dog" system is used in the sender PE and in the receiving PEs to time out the arrival time of an acknowledgement from the next PE. If an acknowledgement does not arrive during the a time interval, a receiving PE sends its own acknowledgement to the preceding one and it propagates up to the sender PE.

Another way to implement broadcast messages is "tree" type, depicted in figure 3.16, in which request to transfer data and data are transferred successively with short delays from one PE to the PE. When the last successive receiving PE is connected to the sender PE, both ends of the bus are connected, and a connection similar to a "ring" type is established. When the last successive receiving PE is not connected to the sender PE, both ends of the bus are disconnected, and a "tree" type connection is established. In both configurations, at each time only one message can use the bus.

This scheme uses the multiprocessor interconnection network described before. When a PE detects a broadcast message it forwards it immediately to the next PE that belongs

Figure 3.16 - Tree type transfer

to the same group ID. Every PE in this serial transfer receives the data after a delay which corresponds to the sum of delays of its preceding PEs. These delays are due to the time that it takes to detect the address ID and to establish a connection between adjacent PEs. The response to the sender is in serial from one PE to the other as in the "daisy chain" scheme described above. In both the "tree" and the "ring" configurations every PE can operate as a sender PE.

This scheme can be modified by establishing a circuit switching connection between the PEs during the request to transfer data. Circuit switching eliminates the time delays characteristic of store-and-forward and VCT switching. The response to the sender is in serial through the receiving PEs as before. Data to the receiving PEs is sent only after the sender PE receives an acknowledgement that all the PEs are ready (i.e. constant circuit switching connections have been established).

3.2.6. Design choice

The multiprocessing systems under investigation contain a large number of PEs connected in any arbitrary network. Transferring data packets of any size from one PE to the other may require some hops. Therefore, it is simpler and faster to transfer packets of data in a synchronous technique. Later in section 3.5 it will be shown that depending on the application and the I/O configuration data may also be transferred in an asynchronous technique but in an arbitrary multiprocessing network the synchronous technique is preferable.

The choice of clock synchronization is arbitrary and depends on the EMI (Electro Magnetiv Interference) environment.

The analysis and the comparison show that data transfer in the VCT mode is faster than in the store-and-forward mode. Since we are interested in high performance real time system implementations the VCT switching mode is chosen.

Later in section 3.3 it will be shown that the handshaking protocol is chosen. Data broadcasting in a handshaking protocol environment can be done in any of the methods described in paragraph 3.3.5. But for a given network configuration or application one method might be preferable over the others.

3.3. Interconnection protocols

3.3.1. Introduction

A multiprocessor system is a collection of processing elements (PE) connected through a network which executes multiple tasks in parallel. Such a system helps to exploit the parallelism inherent in digital signal processing algorithms by dividing a task into multiple independent sub-tasks and executing them concurrently in different processing elements. Interconnection network between the processing elements permits: 1) data transfer between them, and 2) sharing of resources by them. To coordinate data transfer and interactions among the PEs, a communication protocol is needed. A communication protocol is a set of rules established to: 1) control the operation performed by a PE when it transmits a message or when a message is received from another PE, and 2) handle and control data transfer and interactions among the processing elements. Type and format of the exchanging messages between the PEs are very important for correct, complete and simple interaction among the PEs. Some pertinent functions are required from a protocol: synchronization of PEs for data transfer, PE's addressing, different commands for control and handling data transfer, detection of communication errors and control of data flow among the PEs.

Many methods such as finite automata, petri nets, flow charts, formal grammars

and programming languages have been applied by researchers [8, 9, 10, 11, 12, 13, 6, 14, 15] to model, analyze and synthesize communication protocols. Basically, these methods of modeling and verifying protocols belong to one of two approaches: transition oriented models and language oriented models. Transition-oriented models represent protocol status and events as states and changes of states, respectively. Analyzing the state transition graphs validates the protocol's properties. Language-oriented models represent the protocols as an algorithm. The analysis is done by running the program (algorithm) and checking its output (performance) under different inputs (parameters).

Each approach has its advantages and disadvantages but the language-oriented approach is more suitable for complicated protocols and is more flexible to verify changes of network parameters, type of exchange messages and their formats.

To insure simple implementation of the PE proposed in this thesis for the multiprocessor system, a simple and well defined protocol will be described in the next paragraph. This protocol which controls and handles data transfer , operation of a PE and the interaction between them is based upon synchronous data transfer technique and VCT switching chosen in section 3.2. To decrease communication latencies, avoid network congestion and detect errors an acknowledgement handshaking protocol is chosen. The protocol is described, explained in details and verified by flow charts and finite state diagrams.

3.3.2. PE-PE communication

3.3.2.1. Data transfer principles

To decrease communication latencies in data transfer the data transfer procedures are based upon:

1) handshaking protocols.

2) priority of service.

3) routing in minimum number of hops through a virtual-cut-through switching network.

Priority of service and transferring data in a minimal number of hops, determined by the scheduler during the compilation and the partition of the program, avoids link congestion and controls the flow as will be explained in section 3.3.6.

Handshaking procedures and protocols decrease the communication latencies because:

- Data is transferred only when the receiving PE has enough buffer space for it and the data has the priority to be handled, thus freeing the link for transferring only the necessary data.
- Saving time in forwarding data to the next PE (when the receiving PE operates as a network switching node) by using virtual-cut-through switching connection whenever it's possible.

The acknowledgement of the handshaking protocol avoids unnecessary repetition of data transfer and thus reduces the links congestions. Two types of acknowledgements: hop-by-hop and end-to-end have detailed explanation in section 3.3.5. Hop-by-hop acknowledgement is to notify the sender of two adjacent PEs (connected through an I/O link) that data has been received without errors. When the sender PE is a network switching node it can discard the data and use the buffers for another data transfer. When the sender PE is the source of the data it cannot discard the data because it has to wait for an end-to-end acknowledgement. End-to-end acknowledgement is a message from the destination PE to the source PE notifying that the data have been received without errors. Upon receiving this acknowledgement, the source PE discards the data and frees its buffer space for new data.

3.3.2.2. Data transfer protocol

Transmitting Data

Figures 3.17a and 3.17b depict the protocol of transmitting data by an AIO of a processing element (PE).

Data transfer is executed through handshaking and is always initiated by the sender PE. To avoid congestion on the I/O links data is transferred only if the receiving PE is ready to accept it. Therefore, the first step of the sender PE is to establish a connection with the next PE by requesting to transfer data (RDY). If the receiving PE is ready to accept the data it responds back with RDY; if not, the response is NRDY. When the response is not ready, or there is no response within a certain time and the watch-dog system times out, the sender PE will try again two more times to request for data transfer. If the receiving PE is ready, the sender PE sends the data (DTR) to it and waits for its response. When the receiving PE responds back with an acknowledgement (ACK) the data transfer to the next PE has been completed successfully and the sender PE to its relation with the data transferred. If the sender PE operates as a switching network node which forwards the data it is free to start a new data transfer. If the sender PE is the source PE of the data it starts an end-to-end watch-dog and can start a new data transfer while waiting for an acknowledgement from the destination PE. Reception of an acknowledgement frees the buffers for new data, otherwise the watch-dog times out and if there were less than three attempts the source PE tries to transfer the data again. After three failures the PE notifies the operating system or a human operator about malfunction of the system and the system is stopped. When the receiving PE detects an error in the data and responds back with an ERR and/or it fails to respond and the watch-dog system times out the transfer, the sender PE will try two more times to transfer the data.

Three failures in establishing a connection for data transfer (RTD) or in transfer-

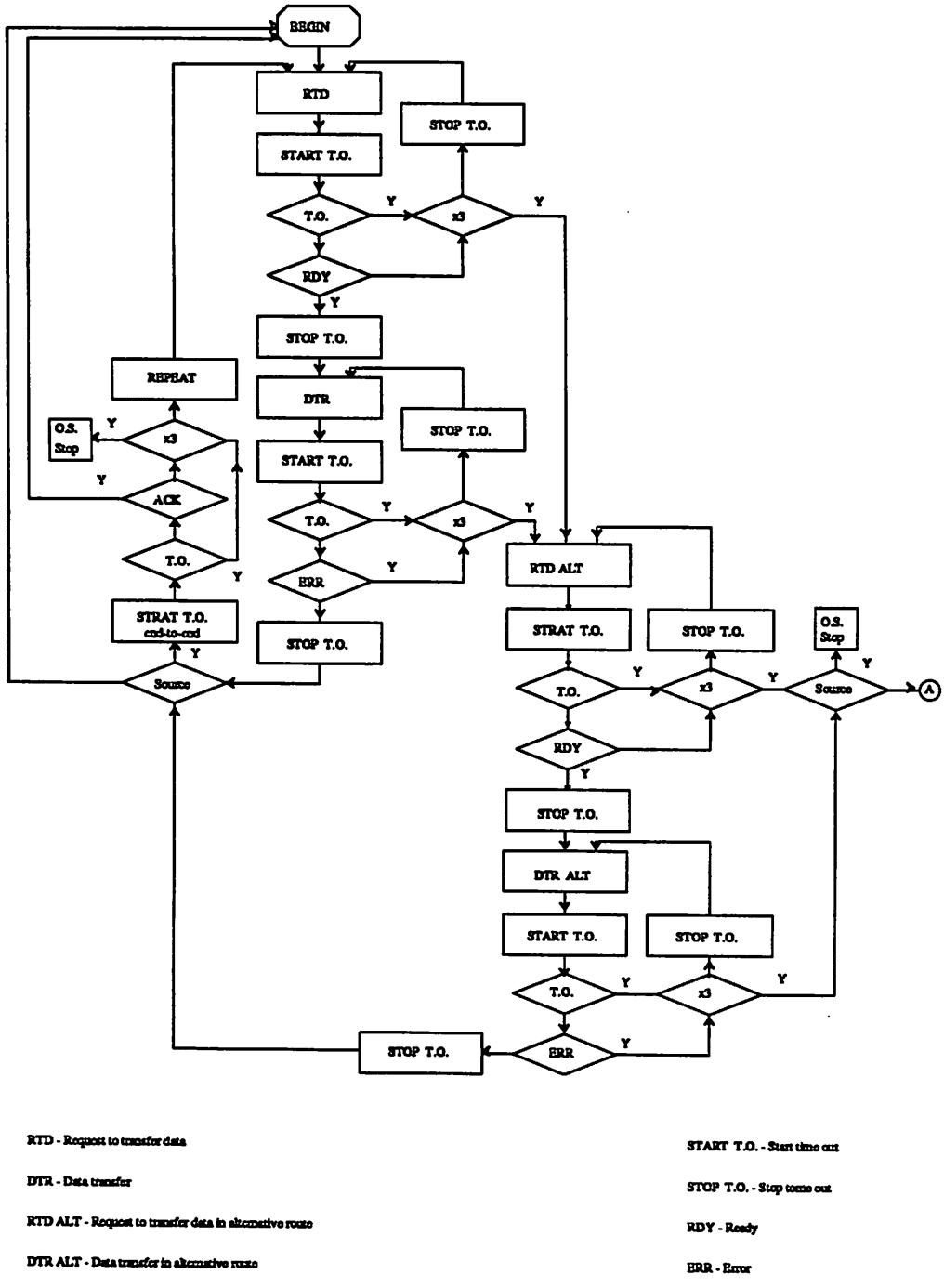
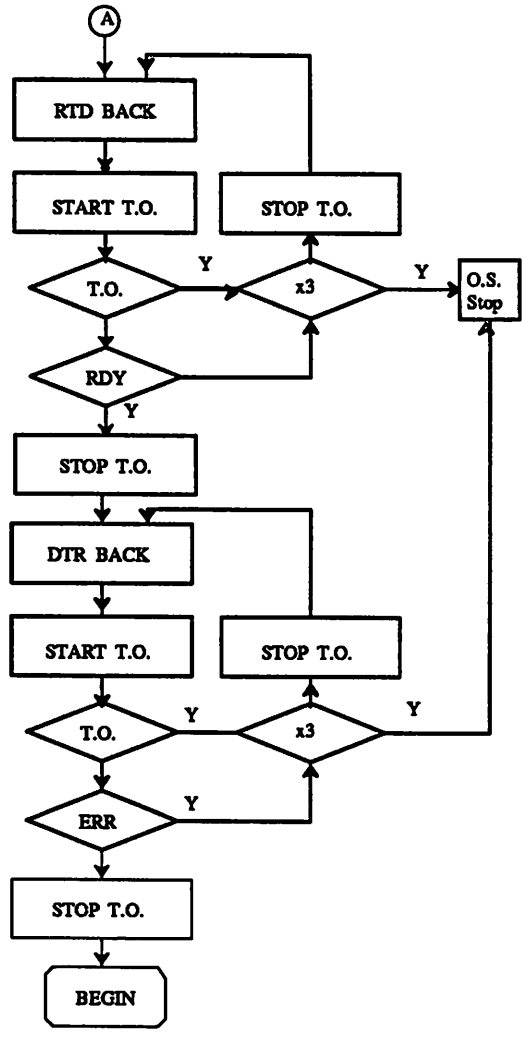


Figure 3.17a - Transmitting data protocol

ring the data itself (DTR) result in choosing an alternative route by the sender PE for transferring the data. The procedures of transferring the data through the alternative route are the same as for the main route described before. When the data is transferred



RTD BACK - Request to transfer data back

DTR BACK - Data transfer back

Figure 3.17b - Transmitting data protocol

successfully the sender PE is free to start a new data transfer. When the data transfer through the alternative link also fails three times the sender PE acts differently if it is a network switching node which forwards the data or a source PE. If it is a source PE the sender PE notifies the operating system or a human operator that the system malfunctions and the system will be stopped. If it is a switching network node it will try to return the data back to the PE that had send it. The procedures of returning back the

data are identical to forwarding it. Success in returning the data frees it to start a new data transfer, while three failures result in notifying the operating system or a human operator about the malfunction of the system and the system will be stopped.

Receiving Data

Figure 3.18 depicts the protocol and the procedures of receiving data by an AIO of a processing element (PE).

Upon receiving a request to transfer data (RTD), the receiving PE checks whether it has the priority to handle this transfer and whether enough buffer space is available. If it is ready to accept data it responds with RDY; otherwise, it responds with NRDY. When it is ready in parallel to the RDY response it starts a watch-dog which limits the time of reservation time of the buffers. If data does not arrive within a certain time the watch-dog times out and the buffers are free for new data allocation. When data arrives within the timeout limits the receiving PE checks it for transmission errors. If there are no errors it responds with hop-by-hop acknowledgement (ACK), if there are errors it responds with error message (ERR). Since a virtual-cut-through network is used, when the receiving PE is not the destination PE, while responding to the sender PE that it is ready to receive the data, the receiving PE tries to establish a connection for forwarding the data to the next PE. This virtual-cut-through data transfer protocol is identical to the transmitting data protocol described before. If a VCT connection has been established and the receiving PE detects errors in the received data, it will report them to the next PE.

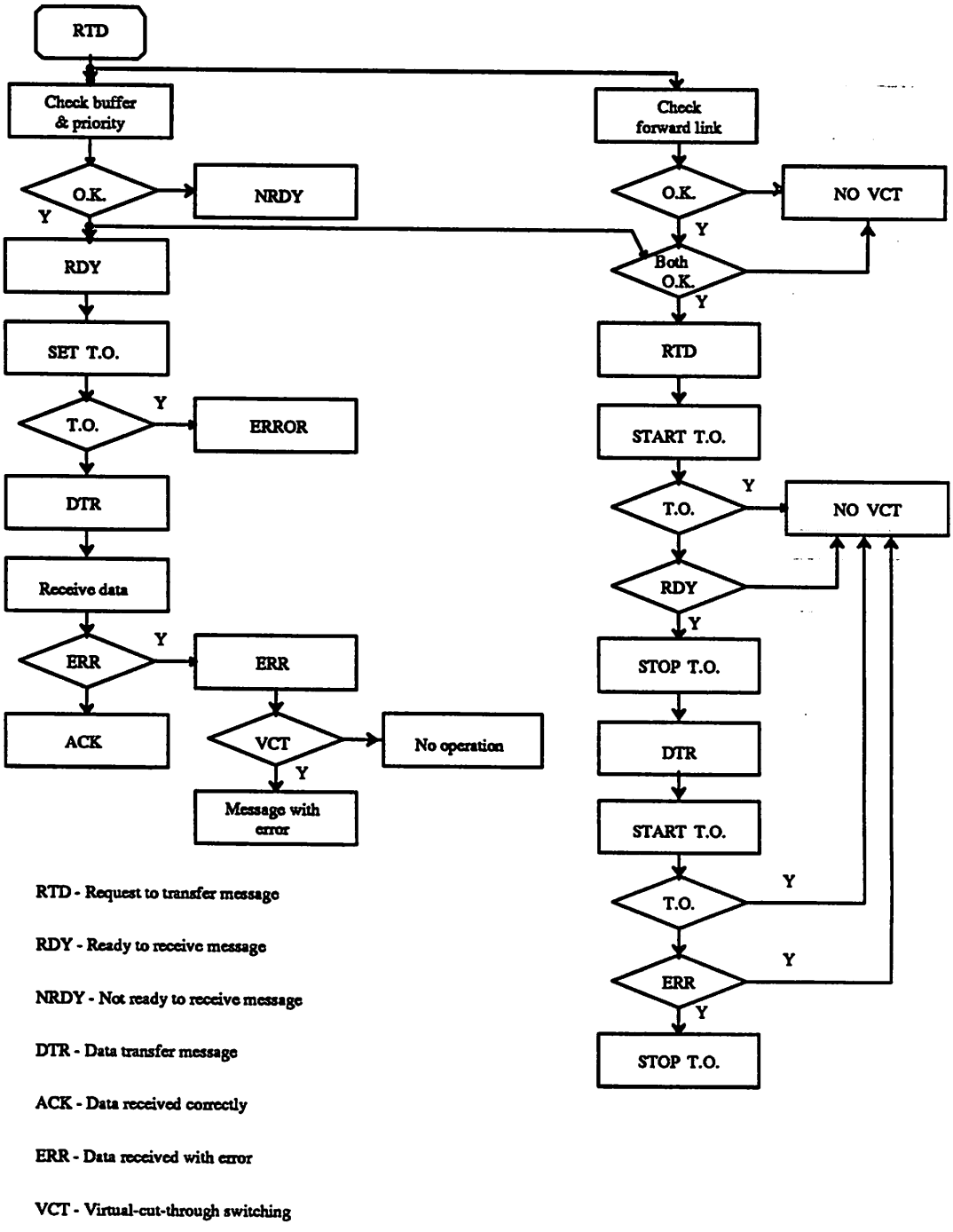


Figure 3.18 - Receiving data protocol

3.3.3. AIO-PU communication

The PE contains the AIO and the PU which reside on the same VLSI chip and

operate independently, concurrently and synchronously. Incorporation of the AIO in the PE frees the PU from dealing with the data transfer when the PE operates as source, destination or switching node. Data to be transmitted from a source PE to another PE or received by the destination PE from another PE is buffered in predefined locations of the dual port memory. The use of predefined locations in the dual port memory for specific sources and destinations simplifies the communication between the AIO and the PU.

Dual port memory has duplicate address and data latches as well as two R/W control lines which enable concurrent reading from two different locations or reading from one location while writting to another one. Such a configuration enables the AIO and the PU to access the buffers simultaneously and synchronously without any interference.

Since implementation of the dual port memory is more complicated and requires a larger area, only part of the PE's memory which is used to buffer data between the AIO and the PU should be implemented this way. The memory of the PE is built of two memories: one is a regular one port memory which is the "private" memory of the PU and contains programs and data and the other is a dual port memory which is the data transfer buffers between the PU and the AIO.

Data transmitted to another PE

During the execution of a program, the PU transfers data into the buffers like any other store instruction. Before storing a new set of data into the buffers of the same destination PE, the PU checks flags to verify whether the previous data has been transferred completely (i.e. the buffers are empty for new data). When all the data to be transferred is stored in the buffer, the PU notifies the AIO by an *OUT* instruction, that data is ready to be transferred. The *OUT* instruction contains the address of the destination PE and the size of data words (memory words) to be transferred.

When the AIO has transferred the data successfully and an end-to-end acknowledgement has been received from the destination PE, it interrupts the PU and reports by a flag/semaphore about the completion of the data transfer.

Data received from another PE

When the PE is the destination of a data transfer, the AIO of the PE checks whether enough buffer space is available to allocate for the receiving data. If there is enough buffer space the AIO receives the data, arranges it in the right format and and stores it in predetermined locations in the dual port memory buffer. While receiving the data the AIO checks it for errors.

If there were no errors, the AIO sends back to the sender (preceding node) an "ACK" message, sets flags/semaphores in a status word in the memory, and interrupts the PU to notify it that there is new valid data. [The PU will reset the flags/semaphores after using the data].

If there is an error, the AIO sends back to the sender (preceding node) an "ERR" message, frees the buffer space and does not interrupt the PU.

3.3.4. Protocol verification

An error-free protocol is essential to reliable communication. Many methods can be applied to detect errors in protocols and to verify their correctness[9, 16, 10, 17, 12]. Simple protocols that involve a small number of states can be verified by state diagram representation and reachability analysis. More complicated protocols may be verified by petri nets or other high language methods. The type of errors handled are: state deadlocks, unspecified receptions, nonexecutable interactions and state ambiguities. A state deadlock occurs when there is no way to exit a state or when a set of states are in an infinite loop without any exit (a process has no alternative but to remain indefinitely in the same state or in to loop in a set of states). An unspecified reception occurs when a correct message can take place but it is not specified in the protocol (e.g. a missing arc

in state diagram representation). Nonexecutable interaction means that a defined message can never occur and thus a state can never be reached.

The protocol of the multiprocessor system described in this dissertation is simple and can be represented by state diagram. Figure 3.19 depicts the state diagram of a sender PE.

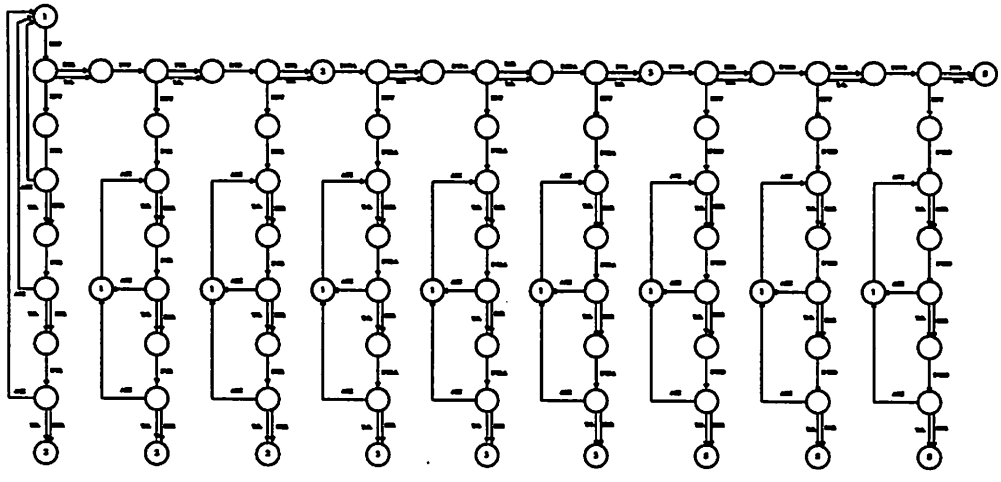


Figure 3.19 - Sender's protocol state diagram

For better understanding and simpler analysis this state diagram is divided into three parts: main route depicted in figure 3.20, alternative route depicted in figure 3.21 and backward route depicted in figure 3.22.

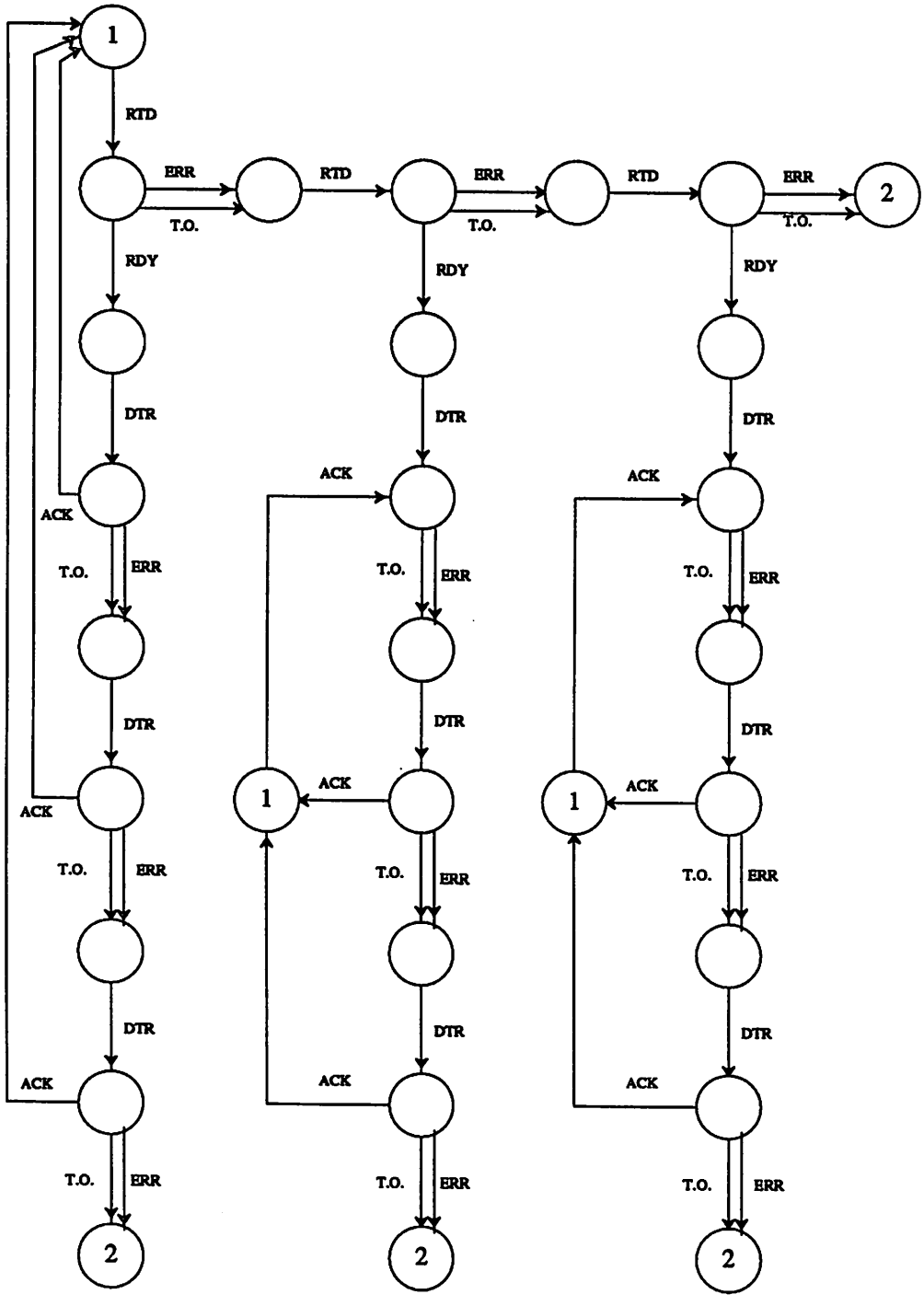


Figure 3.20 - Main route protocol's state diagram

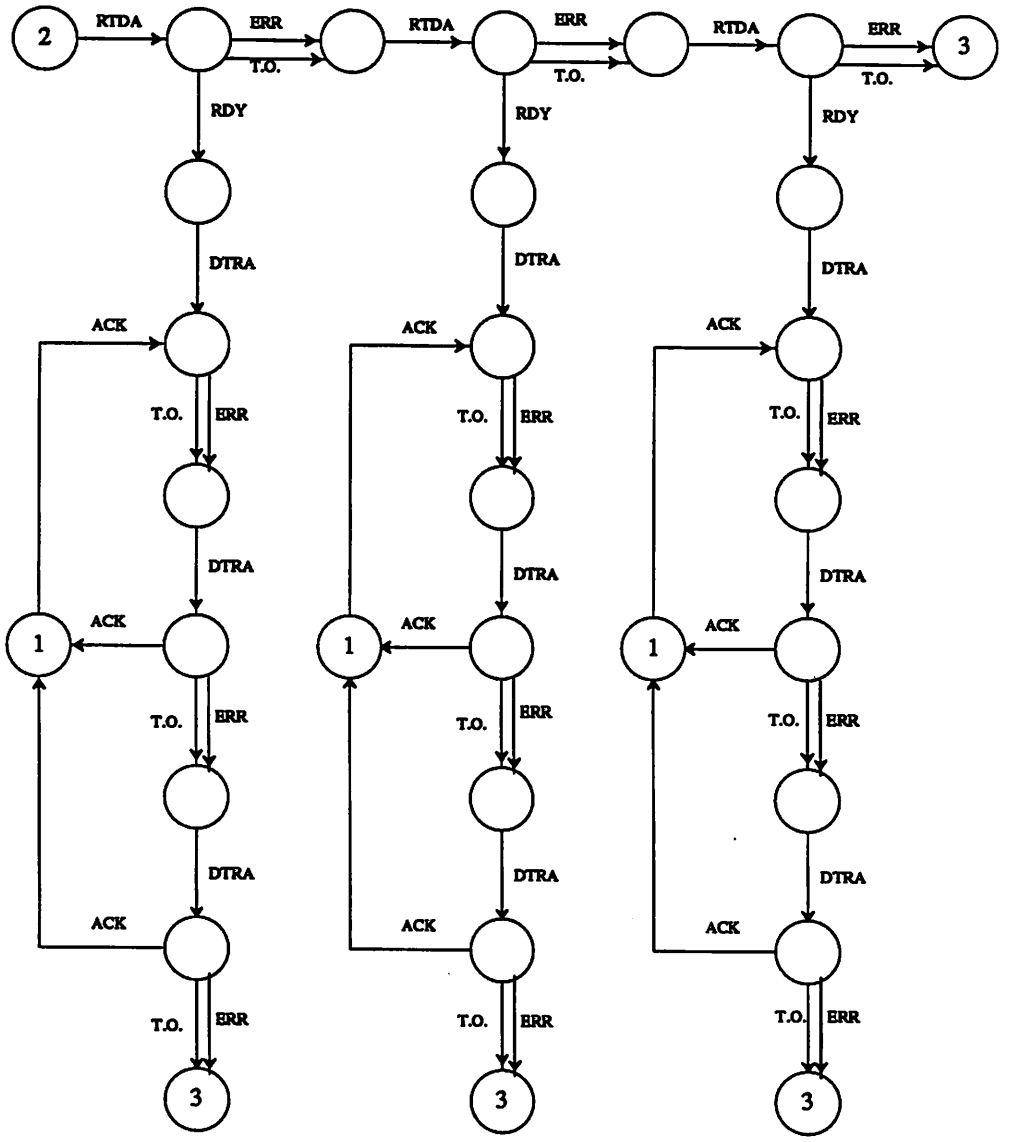


Figure 3.21 - Alternative route protocol's state diagram

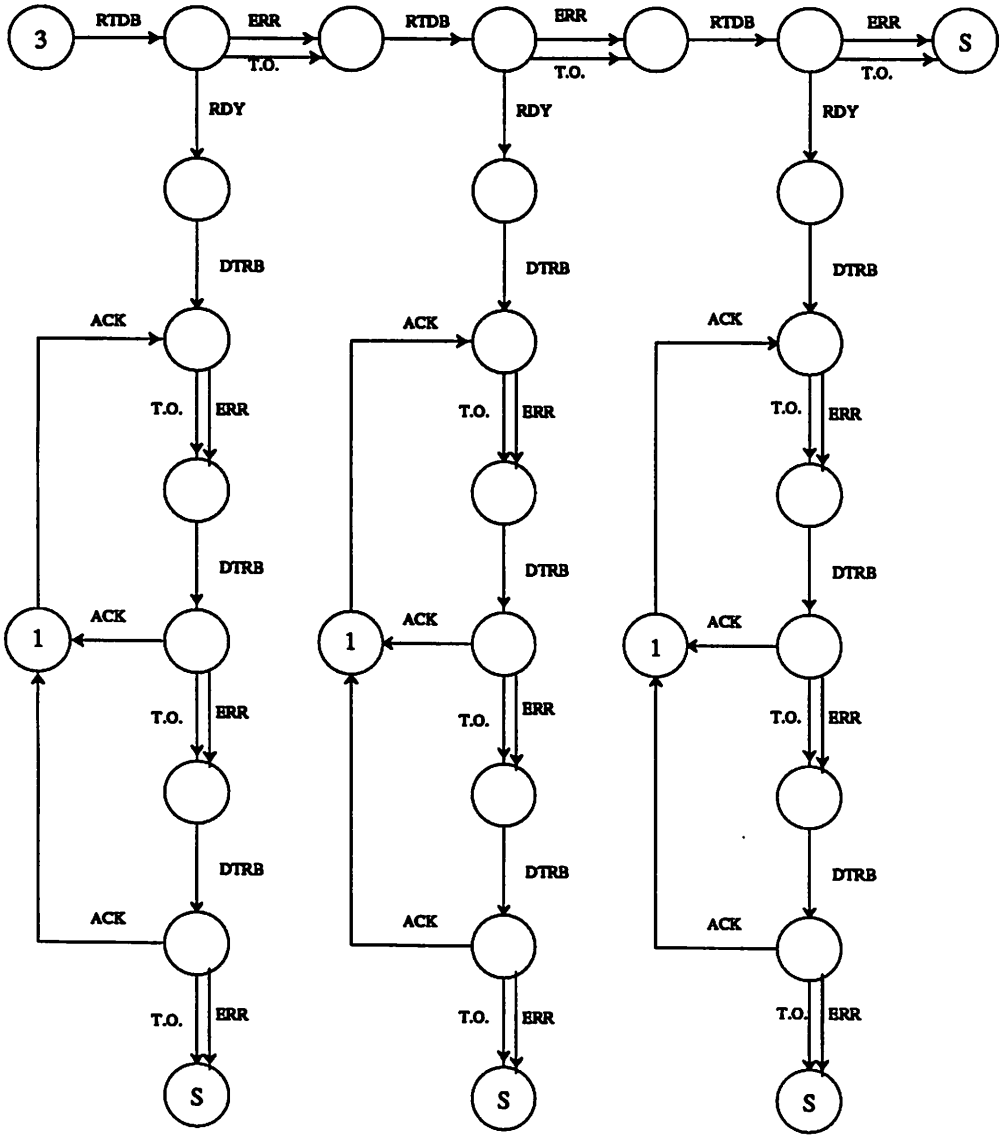


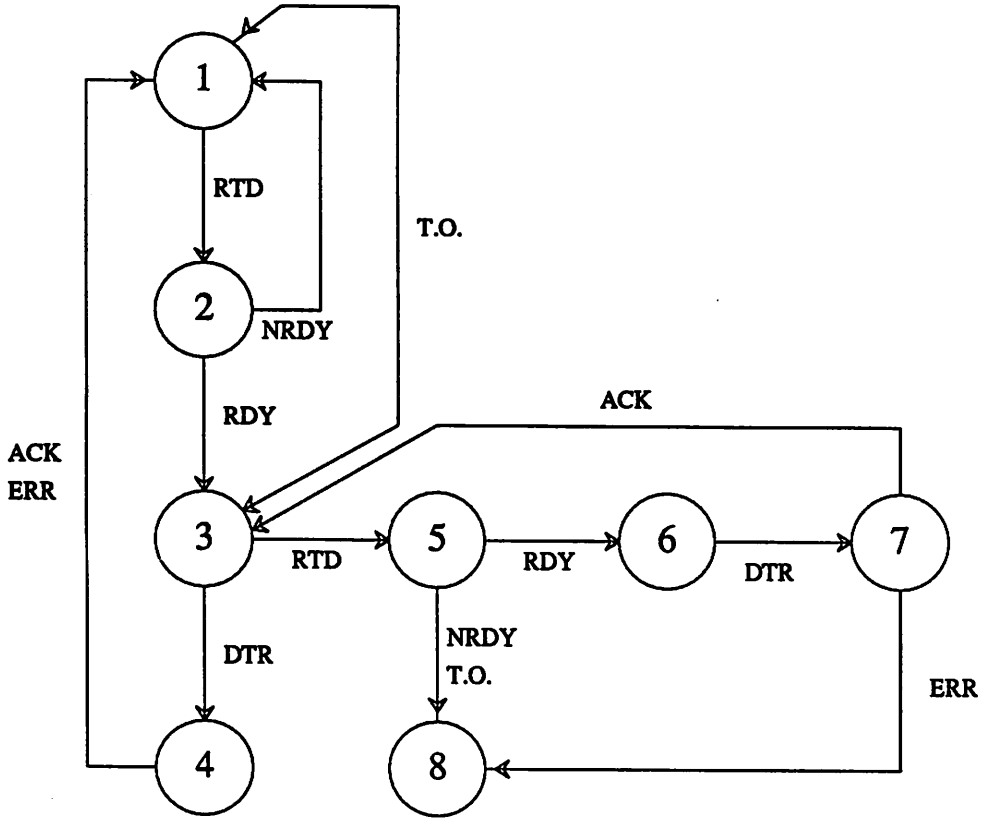
Figure 3.22 - Backward route protocol's state diagram

To check the correctness of the protocol, reachability analysis can be used. Reachability analysis is based upon exploring all the possible transitions due to arriving messages and time-outs. In the above state transition diagrams, arcs which correspond to illegal messages or undefined control commands are not described because they are not causing any state transition and the AIO ignores them and anticipates the time-out mechanism to exit from the current state.

Applying reachability analysis to the state transition diagrams of the sender PE shows that from every state it is possible to exit to another state and there are no sets of states which make an infinite loop. Legal control commands transfer from one state to another with no ambiguity. Any illegal control command does not cause any transition until a time-out occurs. The only state which does not have any exit is state "S" which indicates that the system is malfunctioning and requires the interference of an operating system or a human operator because a sender PE cannot forward a message to the next PE or backward it to its predecessor PE. Any sequence of legal control commands transfer the sender through a correct set of states starting and ending in the initial "1" state.

The state transition diagram of the receiving PE depicted in figure 3.23 consist of the receiving states (1 through 4) and the first transmitting states (5 though 8) used for virtual-cut-through switching.

Applying the reachability analysis to the state transition diagram of the receiving PE shows that there are no deadlocks and/or infinite state loop. As before, any illegal control command does not cause any state transition. As required by the protocol, since data transfer is always initiated by the sender PE, a time-out in the receiving PE always transfers it to its initial starting state.



1,2,3,4 - Receiver states

5,6,7,8 - VCT states

8 - Wait state in VCT

Figure 3.23 - Receiver's protocol state diagram

3.3.5. Fault tolerance

The fault tolerance policy depends very much on the 1) error rate in data transfer due to environment conditions (electromagnetic interference, signal-to-noise ratio), and 2) failure probability of I/O links, AIO and PU.

The error probability in data transfer can be expressed as a function of SNR and noise statistics. Different transmission media have different noise characteristics and bandwidth limitations. In multiprocessing systems the distance between the PEs is

short and therefore the probability of an error in data transfer is smaller. A rule of thumb shows that implementing the network with correctly designed fiber-optic links results in error rates around 10^{-9} to 10^{-12}. Implementing the network with coaxial links results in error rates around 10^{-5} to 10^{-7}.

The failure rate of integrated circuits is measured by MTBF (mean time between failures), and in today's technologies is a couple of thousand hours that corresponds to a couple of years. Therefore, the failure rate of integrated circuits is very small and negligible for defining a fault tolerance policy.

To cover different media and environments the fault tolerance policy incorporates: 1) an error detection character in every message, 2) hop-by-hop and end-to-end acknowledgement, and 3) repetition of transmission in case of errors.

The error detection character can be a simple parity bit, a LRC parity character or a cyclic redundancy code (CRC). Choosing any of them depends on the transfer media. The error detection can be simpler for higher SNR.

As was mentioned briefly in section 2.3 hop-by-hop acknowledgement is necessary to decide whether data has to be retransmitted again or can be discarded if the PE operates as a switching network node, and end-to-end acknowledgement is necessary to ensure that data has reached its destination and was not: 1) forwarded to another destination due to an error in defining the route, 2) lost in a node which received the data but did not forward it due to temporary malfunctioning, and 3) stuck in an endless loop.

In the event of an incorrect data transfer or a time-out the sender PE will try to retransfer the data through the same predetermined route two more times. If the transfer fails three successive times, the sender PE will try to send the data through a predetermined alternative route three times. If there are also three failures in data transfer through the alternative route, there are two ways to continue depending on the sender. If the sender PE operates as an intermediate node, it will transfer the data backward to the

previous PE. If the sender PE is the source PE, it signals a system malfunction. Any data transfer resulting in three successive failures will be reported by the sender PE to the operating system or to the human operator for further action.

3.3.6. Flow control

The handshaking protocol (send/acknowledge) adopted in this multiprocessor system provides a natural flow control mechanism. Data is transmitted over the link only if the receiving PE has the enough buffer space available for the data and it has the priority to handle the data. Retransmission of data occurs if 1) the receiving PE replied that errors were detected in the received data or 2) the time out system indicated that something is wrong with the data transfer (malfunction of the link or of the receiving PE).

Using the fault tolerance principles of predetermined main route or alternative route for data transfer in a limited number of attempts contributes to the flow control. The sender PE tries to transfer data to the next PE through a predetermined main route. If it fails three times it tries to transfer the data to the receiving PE through a predetermined alternative route. If again, it fails three times it notifies the operating system and/or human operator. When a sender PE which operates as a network node also fails to forward the data in the alternative route it tries to return it to the previous sender PE. If it fails to return it the operating system halts the operation of the system.

Send/acknowledge protocols, which control the flow and prevent congestion on the links, also help to avoid deadlocks in the system by enabling data transfer only when it can be accepted. Using different buffers dedicated to special links and send/acknowledge protocol might slow down the transfer but avoids deadlocks by regulating the available buffer space.

3.4. Protocol formats

The message formats designed in this section are based upon synchronous data

transfer technique, VCT switching and acknowledgement handshaking protocol designed in the previous sections of this chapter. Some ways to decrease the message header's overhead (short ID) are described in paragraph 3.4.2. The implementation of short ID are described chapter 4.

3.4.1. Message formats

Data transfer between adjacent PEs is executed by a handshaking procedure. The AIO of the sender PE initiates the data transfer by sending a request message with the following format:

SYN	RTD/RTSD	PACKET ID	SOURCE	DESTINATION	LENGTH/DATA	CKS	END

- SYN - Control character establishes and maintains synchronization between the AIO of the sending and receiving PEs.
- RTD - Command code for request to transfer a block of data.
- RTSD - Command code for request to transfer a sample of immediate data. The LENGTH/DATA field contains the immediate data.
- PACKET ID - Serial number of the packet to be transferred (necessary for fault tolerance).
- SOURCE - ID of the PE which is the source of the data information.
- DESTINATION - ID of the PE where the data information is to be delivered.
- LENGTH - If the command (instruction) code is RTD, defines the length of the data block (number of bits, bytes, words etc.).
- DATA - If the command code is RTSD, this field contains the immediate data.
- CKS - A character for detecting errors in the transmission.

- END - A character which indicates the end of the message.

If the buffers of the receiving PE (network switching node or destination) are empty and the receiver has the priority to handle this data, the receiver will send back a control word "RDY" indicating that it is ready to receive the data. This control word has the following format:

SYN	RDY	PACKET ID	SOURCE	DESTINATION	CKS	END

- RDY - Command code indicating that the receiver is ready to receive the data related to the RTD control command sent by the PE sender.

If the receiver's buffers are full and/or the receiver is busy and cannot handle the data, it will respond with the control word "NRDY" indicating that it is not ready to receive the data. This control word has the following format:

SYN	NRDY	PACKET ID	SOURCE	DESTINATION	CKS	END

- NRDY - Command code indicating that the receiving PE is not ready to receive the data.

Upon receiving a ready response ("RDY") from the receiving PE, the sender PE will send the data information serially in a synchronous mode with the control word "DTR" (Data Transfer). The data will be sent in a packet of the following format:

SYN	DTR	PACKET ID	SOURCE	DESTINATION	LENGTH	DATA	CKS	END

- DTR - Command code indicating data transfer.

- DATA - The actual block of data information.

Upon completion of the transfer the receiver will send back a control word which will either be "ACK" to notify the sender that the transfer has been executed successfully or "ERR" to notify the sender that the transfer failed and was not completed successfully. The format of this control word will be:

SYN	ACK/ERR	PACKET ID	SOURCE	DESTINATION	CKS	END

- ACK - Data received without errors. A response for immediate data transfer (RTSD) or block data transfer (DTR).
- ERR - Errors in the received data.

When a failure occurs in transferring some data, the sender PE will attempt to retransfer it up to two more times. If there is still a transfer failure, the sender PE will try to transfer the data through an alternative route (which has also been predetermined by the scheduler). Since the cause for failure could have been a hardware failure on the link through which the receiving PE responds, it is important to distinguish between transferring data through the prime route and transferring it through an alternative route. [Such distinction is required for discarding repeated data which has already been received correctly]. Therefore, the messages of request to transfer data and data transfer for the alternative route will have different control commands codes in the same formats as before:

SYN	RTDA/RTSDA	PACKET ID	SOURCE	DESTINATION	LENGTH/DATA	CKS	END

- RTDA - Command code of request to transfer a block of data through an alternative route.

- RTSDA - Command code of request to transfer a sample of immediate data through an alternative route.

SYN	DTRA	PACKET ID	SOURCE	DESTINATION	LENGTH	DATA	CKS	END

- DTRA - Command code indicating data transfer through an alternative route.

If the sender PE also fails to transfer the data through the alternative route, there are two cases to consider. If the sender is the source, the multiprocessor system cannot operate properly and the interference of an operating system or a human operator is required. If the sender PE is an intermediate network node, it will try to return the data back to the PE which had forwarded it to him. In both cases the sender PE has to notify an operating system or a human operator about the failure of the interconnection. The formats of returned data are the same as before except for the control commands codes which are different as shown below:

SYN	RTDR/RTSDR	PACKET ID	SOURCE	DESTINATION	LENGTH/DATA	CKS	END

- RTDR - Command code of request to transfer backward a returned block of data.
- RTSDA - Command code of request to transfer backward a sample of returned immediate data.

SYN	DTRR	PACKET ID	SOURCE	DESTINATION	LENGTH	DATA	CKS	END

- DTRR - Command code indicating backward returned data transfer.

To decrease communication latencies, a virtual-cut-through switching system is

used. In such a system a network switching node can forward data before it has been received entirely; thus the data is forwarded before errors could have been detected. Therefore, if while forwarding a message an error is detected at the end of receiving it, the forwarding PE notifies the next receiving PE by a control word with the following format:

SYN	PSE	PACKET ID	SOURCE	DESTINATION	LENGTH	CKS	END

- PSE - Command code which indicates that data described by this message was sent with errors.

Data transfer requires an acknowledgment policy. The policy that have been chosen for the multiprocessor system is a combination of hop-by-hop and end-to-end acknowledgements. Hop by hop means that each data transfer between adjacent PEs has to be acknowledged by the receiver. Once the receiver PE has acknowledged the data transfer, the sender PE if it operates as an intermediate network node can discard the data and free the buffer for another data transfer. The source PE cannot discard the data until it receives an end-to-end acknowledgement from the destination PE. This end-to-end acknowledgement is necessary to ensure that the data has been delivered to the correct destination without being lost or routed in an endless loop. Data can be lost or routed in an endless loop if an I/O link or a PE malfunctions. The format of end-to-end acknowledgement is the following:

SYN	EEACK	PACKET ID	SOURCE	DESTINATION	LENGTH	CKS	END

- EEACK - Command code which indicates end to end acknowledgement.

Many multiprocessor system application require broadcast data to many processing elements (e.g. image processing, biomedical, etc.). Several ways of implementing data broadcast were introduced in section 3.2.5. Broadcasting in a multiprocessor system based upon an interconnection network requires the data and the acknowledgement to propagate through the processing elements. In a common bus multiprocessor system, data and acknowledgement are transferred through the buses and do not have to propagate through the processing elements. But in both types of implementations it is still possible to use an acknowledgement handshaking protocol similar to the one described before except that the destination address refers to a group of PEs. Such a message would have the format:

SYN	COMMAND	PACKET ID	SOURCE	GROUP	LENGTH	CKS	END

- COMMAND - Indicates any of the following control commands:
  - RTDG - Request to transfer a block of data to multiple PEs.
  - RTSDG - Request to transfer immediate data to multiple PEs.
  - DTRG - Data transfer to multiple PEs.
- GROUP - Address of the destination PEs.

It is important to notice that in all the broadcast implementations described in section 3.2.5, the response of the receiving PEs are returned to the sender PE serially, one after the other, and cannot returned concurrently. Therefore, the receiving PEs must have the capability of responding in the correct timing and checking that its response is not colliding with other responses.

3.4.2. Short header

To implement an effective multiprocessor system for real time applications it is very important to reduce the message's overhead (i.e., percentage of the control bits used in a message). Many fields in the message header can be reduced by using special hardware or some restrictions.

If all the PEs of the multiprocessor system use a same global clock, the SYN field which synchronizes between the PEs can be eliminated.

Instead of having a header with a source and destination IDs that require $2\log_2 N$ bits, where N is the number of PEs in the system (e.g., for 500 PEs each ID requires 9 bits), it is possible to use only a short ID of numbers like : 1 , 2 , 3 .... up to the maximum paths passing through the link. Each short ID defines for its output link a specific path between the source PE and the destination PE that was predetermined by the scheduler during the partition of the algorithm and the assignment of the tasks to different PEs. Thus, the AIO at each node has a lookup table which according to the input link translates the short ID into either a source PE ID and a destination PE ID, or to a short ID for the output link that has to be used for forwarding the message. The use of translation look-up tables shortens the header and reduces the decoding time.

Another way to reduce the header is by defining the length of the data as multiples of some figure, e.g. a multiple of 256 bits (characters or words) → 512, 768, 1024 etc. By limiting the maximum length to 1024 bits (characters or words) the length field will require only 4 bits instead of 10 bits that would have been required for any block size of data up to 1024 bits (characters or words).

In a multiprocessor system the distance between the PEs is short and the EMI (Electromagnetic interference) is low, which corresponds to a large signal to noise ratio. Therefore, the probability of transmission errors is low , and instead of having complicated error detection/correction codes like CRC or LRC it is sufficient to have a simple

parity check of only one bit. Simple parity check results in reducing the number of the bits in the CKS field of the header and in increasing the speed of detection and response to the sender.

3.5. I/O configurations

3.5.1. Number of I/O links

The optimal number of I/O links per switching node was evaluated through analytical models and simulation by Fujimoto [18]. The number of interconnections pins to chips periphery is limited. Given N pins for p I/O ports, there are $\frac{N}{p}$ pins per port. Thus, I/O bandwidth per port is proportional to $\frac{N}{p}$ (i.e. more ports means less I/O bandwidth per port). Average "end-to-end" delay and total network bandwidth were used as performance measurements. Analytical models based upon queueing theory [19, 20, 21, 22, 7] showed that for a given total I/O bandwidth N, 3-5 I/O links yield the least delay and the most I/O bandwidth per link. Simulation studies which included Barnwell filter programs, block I/O filter programs, FFT programs and LU decomposition have supported the analytic results. Therefore choosing four I/O links for each PE yields a short delay and enables the PE to be embedded in many different network topologies.

The number of lines in each I/O link is parametrizable and can be any number. In the synchronous and the asynchronous techniques, the data is transferred serially and therefore the data bus is one line. In the handshake technique, words of data are transferred in parallel and therefore the data bus contains many lines according to the word length. The number of lines per I/O link depends on the I/O configuration as will be described in the next section.

3.5.2. I/O Configurations

The proposed PE incorporates four I/O links through which data is transferred between the PE and its neighbors. The AIO of the PE handles and controls the data transfer without involving the PU. Data transfer is initiated by handshaking and is executed through a "virtual-cut-through" switching system. A handshaking procedure avoids data transfer when buffer space is not available in the receiving PE. It also enables the receiving PE, if it operates as a network switching node, to establish, if it is possible, a connection for data transfer with its next PE before the data has been received completely (virtual- cut-through switching). Three different configurations of the I/O communication links are depicted in figure 3.24.

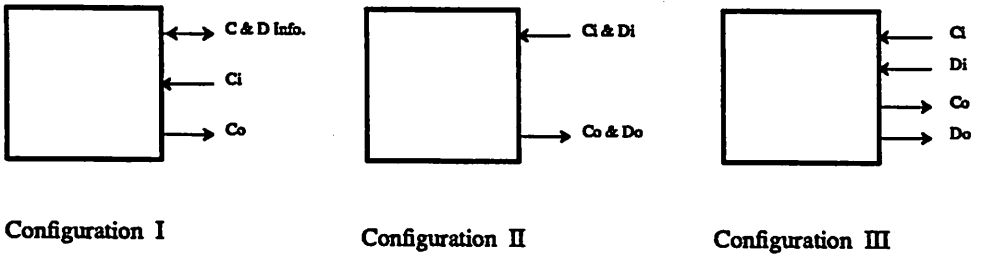


Figure 3.24 - I/O configurations

Using anyone of these configurations depends on the application and the network topology of the multiprocessor system. The I/O link configurations and their corresponding protocols were investigated and are described and explained in the following paragraphs.

3.5.3. Configuration I

Configuration I of the interconnection link depicted in figure 3.24 incorporates one bidirectional data bus and two unidirectional control lines. The data bus can be extended from one line which transfers the data in serial to any number of lines that transfer the data in parallel. Data transfer is half-duplex handshaking, each time the data bus is available to transfer data in one direction. The AIO which controls a data bus of

one line is almost identical to the AIO which controls a data lines that contains many lines. The only difference is that in the first case each bit of data is transferred in serial while in the second case sets of bits are transferred in serial.

This configuration can operate in two different modes. In the first mode control lines C_i and C_o carry only logical levels or pulses for controlling and synchronizing the information transfer on the C&D data bus which carries either data information or control words. Information on the C&D data bus can be transferred either in a synchronous mode or in an asynchronous mode.

In the synchronous data transfer mode the sender initiates the data transfer by setting clock pulses on its C_o control line (C_i control line of the receiver). When the sender PE detects that its C_i control line from the receiver PE is "high" it transfers data on the C&D data bus which is synchronized with clock pulses send on the C_o control line. Figure 3.25 depicts this mode.

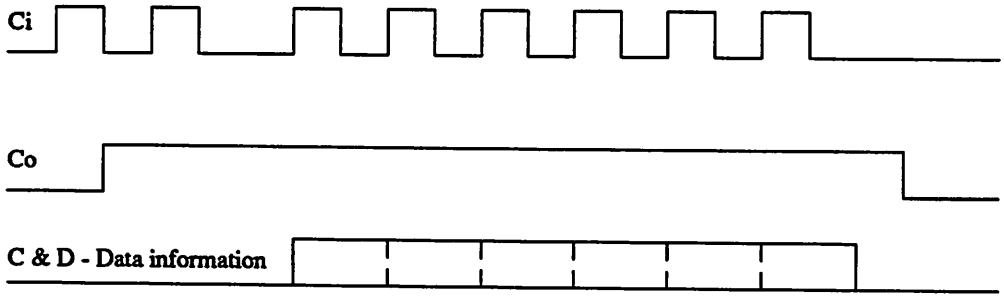


Figure 3.25 - Synchronous data transfer.

In the asynchronous data transfer mode, handshake data transfer is initiated by the sender as explained in section 3.2 and depicted below in figure 3.26.

This mode of operation (synchronous or asynchronous) is fast and efficient for transfers of short, fixed sized data blocks between adjacent PEs like the "wave-front" multiprocessor architecture. In the synchronous and asynchronous modes, the PE which initiates the data transfer determines the direction of the data transfer by becom-

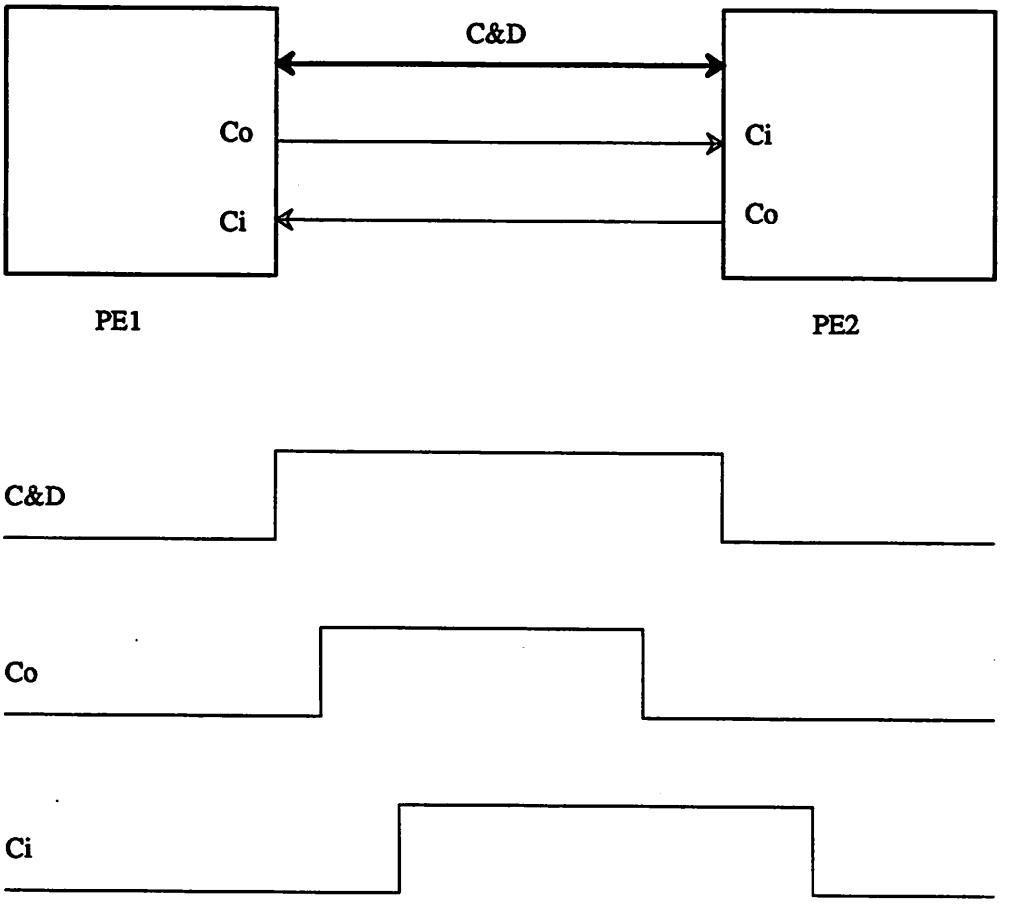


Figure 3.26 - Asynchronous data transfer

ing the sender while the other becomes the receiver PE.

In the second mode, the C&D bus carries only data information while the C_i and the C_o lines carry the control words information (not control pulses or levels). But unlike the first mode, data transfer is not limited to short blocks of information. Data information and control information is transferred in serial, either in a synchronous mode or in an asynchronous mode as was explained before in chapter 2. This mode of operation utilizes separation between control information and data information. Depending on the data transfer protocol, data information might not use a preamble but only data and a postamble. When data information doesn't have a preamble it must immediately follow the control information to avoid confusions.

The data transfer protocol of this configuration is explained by using the second mode of operation which utilizes separate data and control information lines. When the data bus is not busy the sender initiates the handshaking data transfer procedure by setting on its output control line (C_o) the control word "RTD" (Request to Transfer Data) or "RTSD" (Request to Transfer Short Data) which has the following format:

SYC	RTD/RTSD	PACKET ID	SOURCE	DESTINATION	LENGTH/WORD	CKS	END

The LENGTH/WORD field in the RTD instruction denotes the number of data words in the message, while in the RTSD instruction it is the actual data which is transferred.

If the destination buffers are empty and available, and the receiver's AIO is ready and has the priority to handle the data it will respond with "RDY". If the buffers are full or the receiver doesn't have the priority to handle the data, it will respond with "NRDY". The response will be done by setting on the control line (C_i) the control word "RDY" (Ready to receive) / "NRDY" (Not Ready to receive) which has the following format:

SYN	RDY/ NRDY	PACKET ID	SOURCE	DESTINATION	CKS	END

Upon receiving a "RDY" signal when the bus is not busy the sender will send on its output control line (C_o) a control word "DTR" (Data Transfer) which indicates that the bus is occupied for its data transfer. The control word will have the following information and format:

SYN	DTR	PACKET ID	SOURCE	DESTINATION	LENGTH	CKS	END

Following the "DTR" control word the sender will send the data serially in a synchronous mode on the data line (C&D bus). The data will be sent in a packet of the following format:

SYN	START	DATA	CKS	END

Upon proper completion of the transfer, the receiver will send back on its output control line (C_i of the sender) an "ACK" (acknowledge) to indicate that the transfer has been executed properly.

If the transfer has not been completed properly the receiver will send back on its output control line an "ERR" (Not Complete Transfer) to notify the sender that the transfer has not been executed properly.

The format of the response will be:

SYN	ACK/ERR	PACKET ID	SOURCE	DESTINATION	CKS	END

By default, if the sender doesn't receive any signal from the receiver, the transfer was not completed successfully. The sender and the receiver will have "watch-dog" time-out systems that will:

- Notify the receiver that the transfer is wrong if data doesn't follow the "RDY" signal
- Notify the sender if "ACK" or "ERR" doesn't follow the completion of the data

transfer.

Summary of properties

- Operates in either synchronous or asynchronous mode.
- Separate lines for data information and control information.
- Control is full duplex.
- Data is half duplex.
- Data can also be sent in a short protocol which only contains the preamble and the "SYN" of the preamble.
- In the short protocol data has immediately to follow the control words to match data with the destination address.
- Control lines are not utilized 100%.

3.5.4. Configuration II

Configuration II of the interconnection link depicted in figure 3.24 incorporates two unidirectional buses for transferring data and control information. The buses can be extended from one line in which one bit of information is transferred in serial to any number of lines in which bytes or words of information is transferred in serial. The control in both cases will be the same with adaptation to the number of lines in the bus. Unlike the system in configuration I, this system does not have different lines (buses) for control information and for data information, i.e., control and data information share the same bus. Data transfer is similar to full-duplex, because there are two unidirectional lines for sending and receiving data in both directions simultaneously. When no information is being transferred the buses are "idle". As in the second mode of configuration I (section 3.5.3), data transfer is not limited to short blocks of information and the data is transferred in either synchronous mode or asynchronous mode as was explained before in chapter 2.

Data transfer protocol between adjacent PEs is as follows. The AIO of the sender PE initiates the data transfer by sending a control word requesting to transfer data. The control word contains addresses of the source and destination, packet ID, number of words - data length and "RTD" / "RTSD" -control bits for request to send data.
The format of this control word is:

SYN	RTD/RTSD	PACKET ID	SOURCE	DESTINATION	LENGTH/DATA	CKS	END

LENGTH in the RTD instruction denotes the number of data words in the message while DATA is the actual data which is transferred in the RTSD instruction.

If the buffers of the receiving PE (network switching node or destination) are empty and the receiver has the priority to handle this data the receiver will send back a control word "RDY" (Ready To receive) which has the following format:

SYN	RDY	PACKET ID	SOURCE	DESTINATION	CKS	END

If the receiver's buffers are full and/or the receiver is busy and can not handle the data it will respond with the control word "NRDY" (Not Ready to receive) which has the following format:

SYN	NRDY	PACKET ID	SOURCE	DESTINATION	CKS	END

Upon receiving a ready response ("RDY") from the receiver, the sender will send the data information serially in a synchronous mode with the control word "DTR" (Data

Transfer). The data will be sent in a packet of the following format:

SYN	DTR	PACKET ID	SOURCE	DESTINATION	LENGTH	DATA	CKS	END

Upon completion of the transfer the receiver will send back a control word which will either be "ACK" to notify the sender that the transfer has been executed successfully or "ERR" to notify the sender that the transfer failed and was not completed successfully. The format of this control word will be:

SYN	ACK/ERR	PACKET ID	SOURCE	DESTINATION	CKS	END

By default, if at the end of the transfer the sender doesn't receive any signal from the receiver the sender assumes that the transfer was not completed successfully.
Both the sender and the receiver will have "watch-dog" timeout systems that will :

- Notify the receiver that the transfer is improper if data doesn't follow "RDY" signal.
- Notify the sender if "ACK" or "ERR" doesn't follow the completion of data transfer.

Summary of properties

- Operates in either synchronous or asynchronous data transfer mode.
- Efficient for transfer of large blocks of data information.
- Handshaking is done through messages (not control pulses or levels).
- Adequate for systolic array hardware implementation when the same clock is used for all PEs.

- Higher utilization of the I/O links because data and control use the same link.

3.5.5. Configuration III

Configuration III of the interconnection link depicted in figure 3.24 incorporates two unidirectional data bus and two unidirectional control lines. This configuration is a combination of the previous two configurations and has separate data and control lines with the property of full-duplex data and control transfer. Data bus can be extended from one serial line to any number of lines that transfer sets of data bits in serial. The control for one serial data line or a bus of many parallel data lines will be the same. This configuration is the most flexible one. It can handle synchronous or asynchronous data transfers by either control pulses or levels or by control messages. Thus, configuration III is suitable for short data blocks of information as well as large blocks of data information in different control modes.

The protocol used for data transfer is similar to the protocols of configurations I and II. The sender initiates a data transfer by setting on its output control line (C_o) a control word "RTD" (Request to Transfer Data) or "RTSD" (Request to Transfer Short Data) which has the following format:

SYN	RTD / RTSD	PACKET ID	SOURCE	DESTINATION	LENGTH / DATA	CKS	END

LENGTH is the number of data words to be transferred later and DATA is the actual data in the "RTSD".

If the buffer of the next PE (network switching node or destination) is available and the receiver's AIO has the priority to handle the data it will respond by setting the control word "RDY" (Ready to receive) on its output control line which has the following format:

SYN	RDY	PACKET ID	SOURCE	DESTINATION	CKS	END

If the buffer is full and/or the receiver does not have the priority to handle the data it will respond by setting the control word "NRDY" (Not Ready to receive) on the control line which has the following format:

SYN	NRDY	PACKET ID	SOURCE	DESTINATION	CKS	END

Following the receiver's "RDY" signal, the sender will send on its control line a "DTR" control word indicating that data is to be transferred on the data line. The control word will have the following format:

SYN	DTR	PACKET ID	SOURCE	DESTINATION	LENGTH	CKS	END

Following the "DTR" control word, data will be sent serially in a synchronous mode on the data line (D_o). The data will be sent in a packet of the following format:

SYN	START	DATA	CKS	END

Upon completion of the transfer the receiver will send back on the control line a control word which will either be "ACK" (acknowledge) to notify the sender that the transfer has been completed properly, or "ERR" to notify the sender that the transfer has not been completed successfully. The format of this control word will be:

SYN	ACK/ERR	PACKET ID	SOURCE	DESTINATION	CKS	END

By default, if the sender doesn't receive any signal from the receiver the transfer was incorrect. Both the sender and the receiver will have "watch-dog" time out systems that will:

- notify the receiver that the transfer is wrong if data doesn't follow the "RDY" signal.
- notify the sender if "ACK" or "ERR" doesn't follow the completion of the data transfer.

Summary of properties

- Configuration III is a combination of configurations I and II.
- Require four unidirectional lines (buses).
- Efficient transfer of short data blocks as well as large data blocks.
- Controls data transfer by either control pulses and levels or by control messages.
- Data transfer can be synchronous or asynchronous.
- Adequate for "wavefront" multiprocessor implementation and for "systolic array" implementation.

3.5.6. Summary of I/O link configurations

The underline properties of the three configurations are summarized in the following table:

Configuration I 3 links	Configuration II 2 links	Configuration III 4 links
1 bidirectional data link & 2 unidirectional control links	1 pair of unidirectional data and control links	2 unidirectional data links & 2 unidirectional control link
separate lines for data and control	same line for data and control	separate lines for data and control
synchronous or asynchronous data transfer	synchronous or asynchronous data transfer	synchronous or asynchronous data transfer
half duplex data	full duplex data	full duplex data
full duplex control	full duplex control	full duplex control
efficient for short blocks of data information	efficient for large blocks of data information	efficient for any block size of data information
control by signals or messages	control by messages only	control by signals or messages
"Wavefront" architecture	systolic array	any architecture

Table 3.1 - Summary of I/O link configurations

References

1. M. Mano, in *Computer system architecture*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1982.

2. V. Ahuja, in *Design and Analysis of Computer Communication Networks*, McGraw-Hill, New York, 1982.

3. W. Stallings, in *Data and computer communications*, Macmillan publishing company, New York, 1985.

4. A.S. Tanenbaum, in *Computer networks*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

5. M. Schwartz, in *Telecommunication networks*, Addison-Wesley publishing company, Menlo Park, California, 1987 .

6. J.L. Hammond and P.J.P. O'Reilly, in *Performance analysis of local computer networks*, Addison-Wesley publishing company, Reading, Massechusetts, 1986.

7. P. Kermani and L. Kleinrock, "Virtual Cut-Through: A new computer communication switching technique," *Computer Networks*, vol. 3, pp. 267-286, 1979.

8. S.T. Dong, in *The modeling, analysis and synthesis of communication protocols*, Ph.D. dissertation in EECS, Berkeley, Berkeley, California, 1983.

9. G.V. Bochmann and C.A. Sunshine, "Formal methods in communication protocol design," *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 624-642, April, 1980.

10. P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan, and D. Brand, "Towards analyzing and synthesizing protocols," *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 651-660, April, 1980.

11. S. Joshi and V. Iyer, "Protocols and network-control chips: a symbiotic relationship," *Electronics*, pp. 169-175, January 12, 1984.

12. P.M. Merlin, "A methodology for the design and implementation of communication protocols," *IEEE Transactions on Communications*, vol. COM-24, no. 6, pp. 614-621, june 1976.

13. T.P. Blumer and D.P. Sidhu, "Mechanical verification and automatic implementation of commun. protocols," *IEEE transactions on software engineering*, vol. SE-12, no. 8, pp. 827-843, August, 1986.

14. G.V. Bochmann, "Finite state description of communication protocols," *Com-*

puter networks, vol. 2, no. 4/5, pp. 361-372, September/October, 1978.

15. D. Brand and W.H. Joyner, "Verification of protocols using symbolic execution," *Computer networks*, vol. 2, no. 4/5, pp. 351-360, September/October, 1978.

16. A.A.S. Danthine, "Protocol representation with finite state models," *IEEE transaction on communications*, vol. COM-28, no. 4, pp. 632-642, April, 1980.

17. G.V. Bochmann, "A general transition model for protocols and communication services," *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 643-650, April, 1980.

18. R.M. Fujimoto, "VLSI communication components for multicomputer networks," in *Ph.D. thesis, Department of EECS, University of California* , Berkeley, 1983.

19. L. Kleinrock, in *Queueing systems*, John Wiley & Sons, New York, 1975.

20. R.B. Cooper, in *Introduction to queueing theory*, The Macmillan company, New York, 1972.

21. T.N. Mudge, J.P. Hayes, G.D. Buzzard, and D.C. Winsor, "Analysis of multiple-bus interconnection networks," *Journal of parallel and distributed computing*, vol. 3, pp. 328-343, 1986.

22. D. Gross and C.M. Harris, in *Fundamentals of queueing theory*, John Wiley & Sons, New York, 1974.

CHAPTER 4

Hardware implementation and performance

4.1. General description

The operation of the processing element (PE) that incorporates a processing unit PU) and an autonomous I/O unit (AIO) that operate concurrently and independently was shortly described in chapter 2.3. Implementation of a PU that executes the computational part of a task depends on the target applications. But, the implementation of the AIO which 1) handles and controls data transfer between PEs and 2) operates as an interface between the PU and the network is similar for different I/O configurations and their protocols.

The purpose of this chapter is to describe the hardware design and implementation of an AIO unit fitted to handle its tasks.

The AIO unit consist of data and packet buffers, buffer control and bookkeeping, routing & priority tables, communication control between PU and AIO and communication & data transfer control between PEs. Figure 4.1 depicts a detailed block diagram of the AIO with one I/O link.

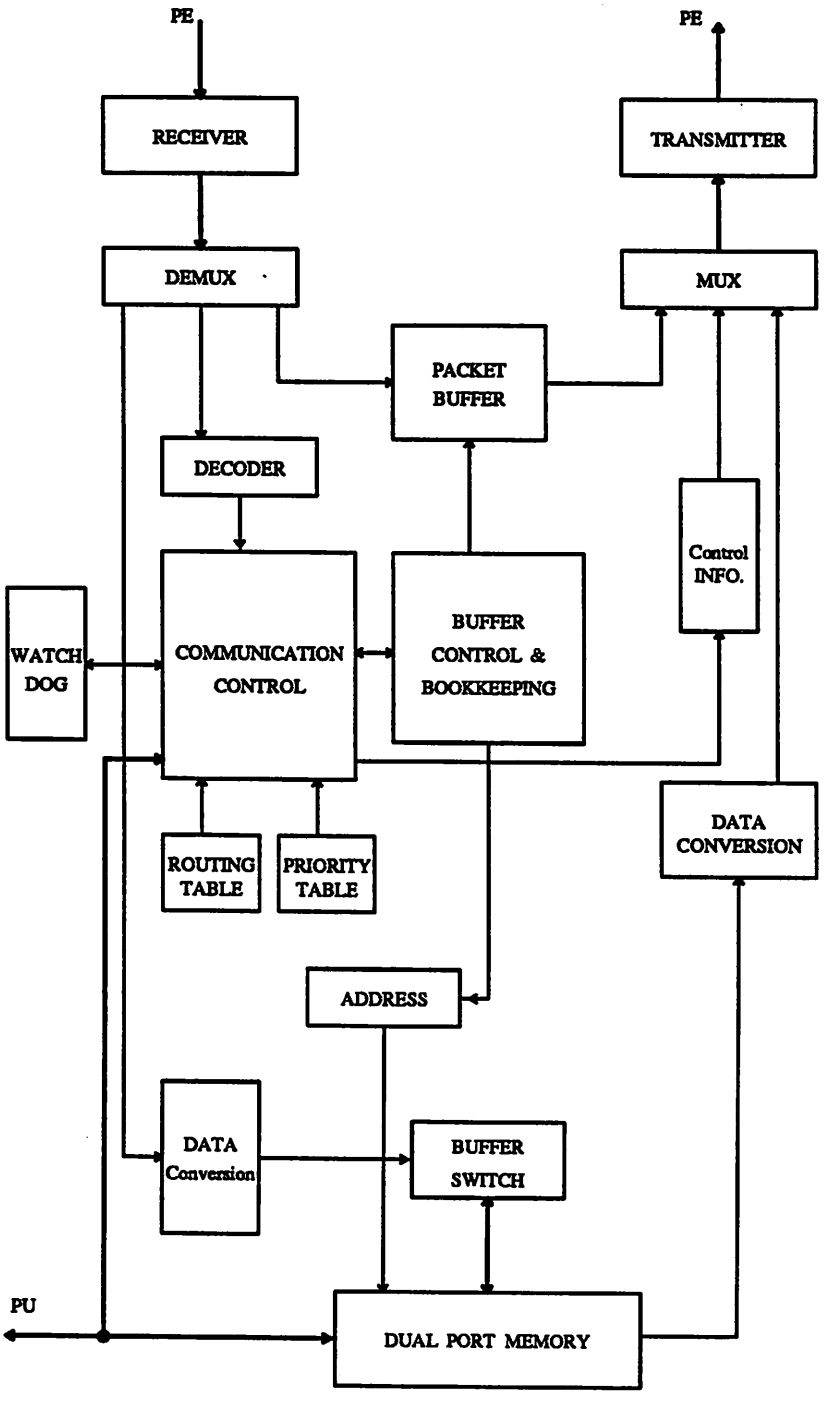


Figure 4.1 - AIO - block diagram

Receiver operation

A message received on the input I/O link from another PE is transferred through a demultiplexer to the decoder. Depending on the decoded command of the message the

communication control evaluates what should be done next. If the instruction is a request to transfer data, the communication control checks the routing and the priority look-up tables for the destination of the message and its priority to be handled. This information is transferred to the buffer control and bookkeeping unit of either the appropriate I/O link if the AIO operates as an intermediate network node, or to the data buffer controller (dual port memory) if the AIO operates as an interface between the PU and the network. The buffer control and bookkeeping unit checks the availability of buffer space according to the packet size and the number of packets already queued for service, and responds back to the sender PE through the communication control unit. Depending on the address of the destination PE, an arriving data packet is stored by the communication control unit either in the packet buffer or in the data buffer. When the PE is a network node that has to forward the packet to another PE, the packet is stored as is in the packet buffer. When the PE is the destination, the packet is converted to the appropriate word format and is stored in the data buffer (dual-port-memory). Whether a virtual-cut-through connection has been established or not a packet to be forwarded is always stored in the packet buffer (the buffer implementation is described in paragraph 4.2). Checking for errors in the data is executed by this unit before a response is returned to the sender PE.

Transmitter operation

A message (packet) to be transferred to another PE is handled by the communication control unit. This unit concatenates the control fields of the messages (command, source ID, destination ID, packet ID, length of data, check sum, end of message), and establishes the handshaking connection by the protocols described before. Data transfer is always initiated by the AIO depending on the priority and the availability of the I/O link. The communication control unit, according to the protocol, controls the number of attempts and the route (I/O link) to be used. When the PE is the source of the information, the AIO fetches the data from the dual-port-memory, converts it to the proper

format and concatenates it with the control fields of the message. When the PE is a switching node the AIO fetches the data from the packet buffer, concatenates it with the control fields and transmits it to the next PE.

AIO-PU Communication

The AIO operates as an interface for data transfers between the PU and the network (figure 2.3). Data to be transmitted from the current PE (source) to another one or received by the current PE (destination) from another one is buffered in predefined memory locations of a dual-port-memory. The use of predefined locations in the dual-port-memory as data buffers for specific source or destination, simplifies the communication between the AIO and the PU.

The PU executes a program and during its execution transfers data into the buffers as in any other store instruction. When all the data to be transferred is stored in the buffer the PU notifies the AIO by an OUT instruction and initiates the data transfer to another PE as described below:

- PU issues an OUT instruction that notifies the AIO the destination PE's address and the size of data (memory words) needed to be transferred.
- AIO executes the following operation:
  - decodes the destination PE,
  - sets a pointer with the address location of the first data word,
  - sets a counter with the number of data words needed to be transferred.
  - checks in the lookup tables which link to use and what's the priority of the transfer.
  - establishes the connection with the next PE according to the predetermined route to the destination.
  - transfers the data to the next PE.
  - interrupts the PU and notifies it by a flag/semaphore about the completion of the

data transfer.

- Before storing a new set of data into the buffers of the same destination PE, the PU checks whether the previous data has been transferred completely.

When the destination is the current PE, the AIO checks in the dual-port-memory whether there is enough space for receiving the data. If the space is available the AIO receives the data, arranges it in the right format and stores it in predefined locations of the dual-port-memory buffer. While receiving the data, the AIO checks if the data packet was transferred without errors.

- If there was no error in the received packet, the AIO sends back to sender PE an "ACK" message, sets flags/semaphores in a status word in the memory and interrupts the PU to notify it that there is new valid data. [The PU will reset the flags/semaphores after using the data].
- If there is an error the AIO sends back to sender PE an "ERR" message, the data is disregarded and the AIO does not interrupts the PU.

Two types of buffers are included: one is the data buffer and the other is the packet buffer.

- The data buffer is used to store data when the current PE is the destination and to fetch data for transmission when the current PE is the source. This buffer is the dual-port-memory accessed by the PU and the AIO which allows data to be transferred between them without any conflict.
- The packet buffers are the buffers used by the I/O links when the current PE operates as an intermediate network node. These buffers, which are described in section 4.2 in more detail, employ two type of buffers: private buffers and shared buffers. The private buffer is a fixed size buffer accessed and used only by a specific I/O link. The shared buffer is additional buffer space accessed and used by all I/O links. Depending upon the frequency that a link is used and the buffer

capacity that it needs, each I/O link has a different dedicated buffer space within the shared buffer. The packet buffer can be implemented by a set of memories organized in n×1 (n words of 1 bit), FIFO or any other implementation.

4.2. PE's buffers

4.2.1. Buffer implementation

The PE incorporates two type of buffers: data buffers and packet buffers. Data buffers are used for data transfer between the PU and the AIO when the AIO operates as an interface unit between the processing unit and the network. Packet buffers are used for temporary storage for forwarding data between PEs when the PE operates as a switching node of the network.

Data buffers are implemented by a dual-port-memory, depicted in figure 4.2, which provides two independent ports with separate address, data and control lines that permits the AIO and the PU independent, concurrent and asynchronous access to any location.

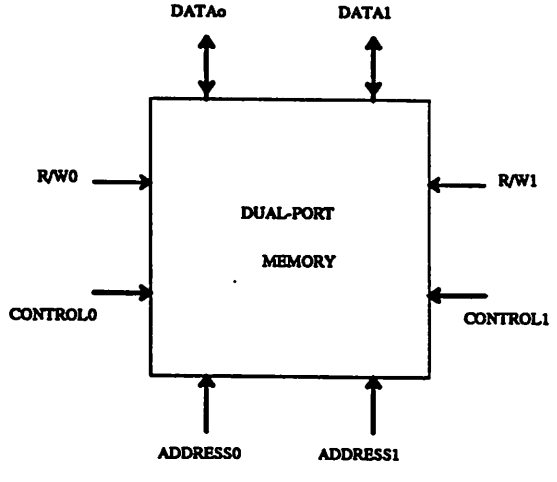


Figure 4.2 - Data buffer - Dual port memory implementation

These buffers reside in the memory space of the PU. Since implementation of a dual-port-memory is more complicated and requires more area, the memory of the PU

is divided into two types. One type is a single port memory which is the "private" memory containing instructions and data, and the other type is a dual port memory for bidirectional data transfer between the PE and other PEs of the multiprocessor system .

Packet buffers can be implemented in several ways: dual-port-memory, a set of shift registers, or a set of interleaved memories organized in n×1 (n words of one bit).

A dual-port-memory implementation incorporates two independent ports with separate address, data and control lines. It allows concurrent data input to the PE (write to the memory) and data output from the PE (read from the memory). Thus, whenever it is possible a virtual-cut-through data transfer can be employed.

Implementation by a set of shift registers incorporates two types of shift registers as depicted in figure 4.3. One type is a FIFO which allows data to be written and/or read from it at independent data rates by utilizing separate synchronous data clocks. The other type is a regular serial shift register with one clock to control the writting or reading of data. The FIFO allows a virtual-cut-through (VCT) data transfer whenever it is possible. When a VCT occurs, data from the FIFO's output which is transmitted through the output link to the next PE, is also stored into a shift register until the receiving PE acknowledges the acceptance of correct data. If a VCT data transfer cannot be executed, the data in the FIFO is transferred to the regular shift register for later transfer. To enable input of data concurrently from three adjacent PEs the input I/O links can be connected to either the FIFO or the shift registers. Since there is only one output link to the next PE, one FIFO per link is sufficient for virtual-cut-through data transfer.

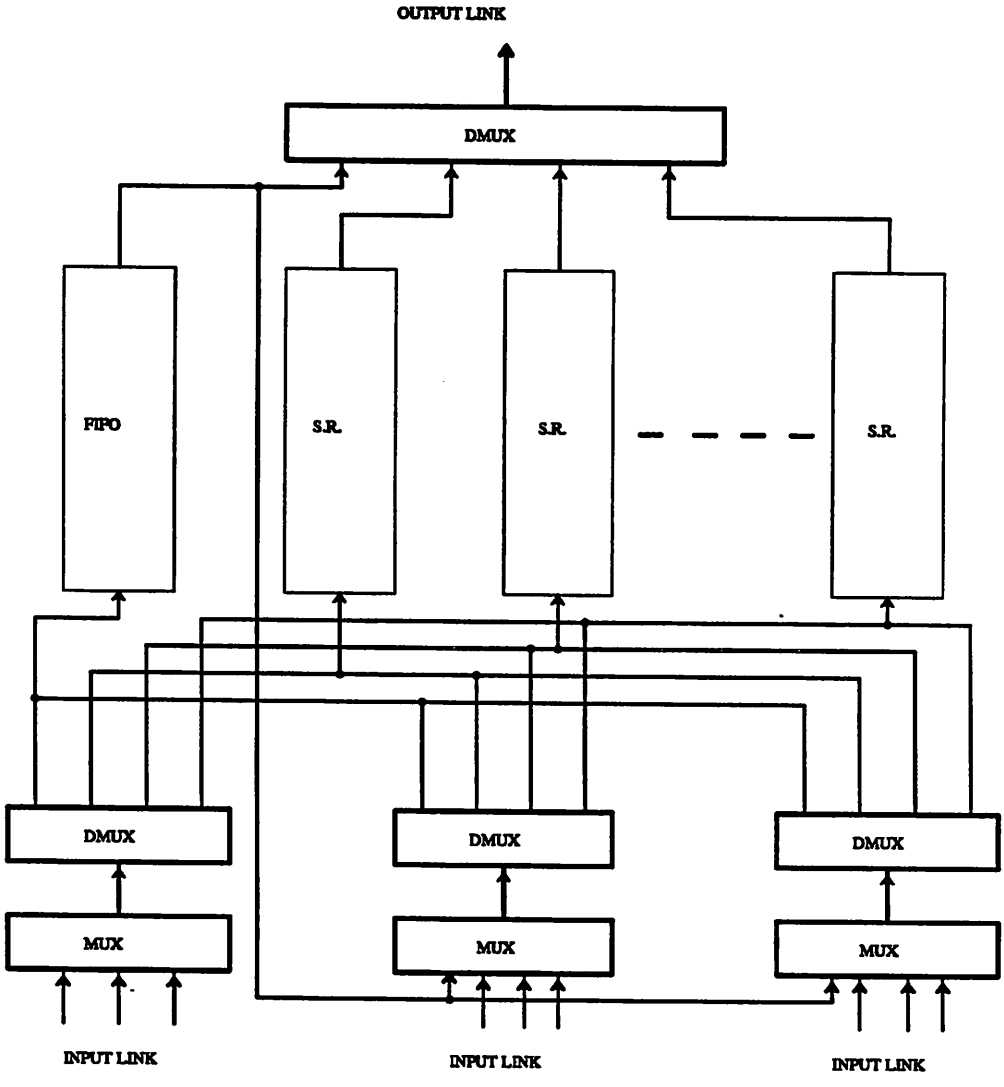


Figure 4.3 - Packet buffers - Shift register implementation

Interleaved memories is another way to implement packet buffers for virtual-cut-through switching. Since each buffer associated with an output link can receive data from the other three I/O links and can transmit data to the next PE, at least four interleaved memories are required. Figure 4.4 depicts a set of four interleaved memories.

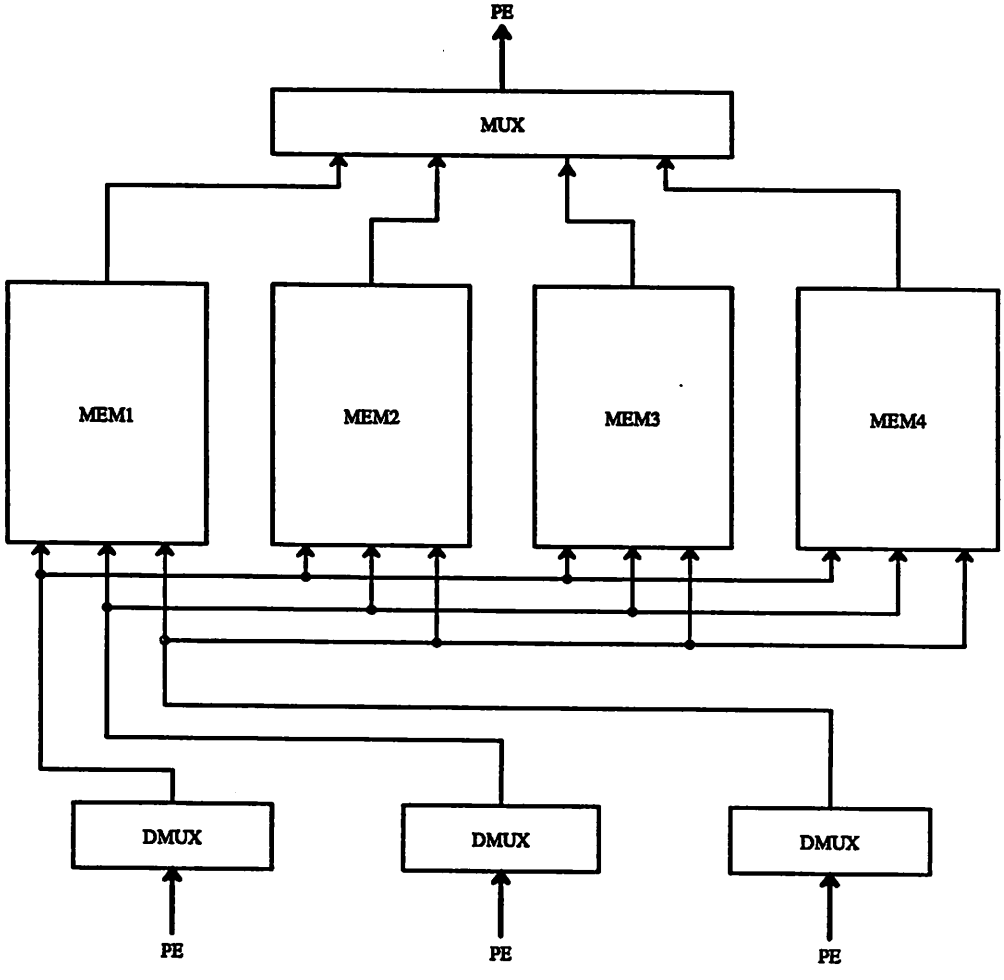


Figure 4.4 - Packet buffers - Interleaved memories implementation

Successive input data words are stored in different memories in a four way interleaved mode, i.e. word 4i is always stored in memory I (MEM1), word 4i+1 is always stored in memory II (MEM2), word 4i+2 is always stored in memory III (MEM3), and word 4i+3 is always stored in memory IV (MEM4). Providing that no two I/O links require the use of the same memory, this scheme allows each of the three I/O links to store simultaneously a word into the buffer while the output link reads a word from it. Since only one link can be granted access to a particular memory, additional registers are required to temporarily buffer the receiving data until it can be stored. Therefore, after the initial synchronization, as many as four concurrent memory accesses occur on

each cycle, and each link is able to access a different memory on each cycle. Accessing the appropriate memory is controlled simply by decoding the two least significant bits (lsb) of the address.

4.2.2. Buffer size analysis

The size of the packet buffers allocated to an output link has a large influence on flow control and the avoidance of deadlocks. Larger buffers reduce the congestion on the I/O links, as well as the probability of deadlocks, by reducing the possibility of buffer overflow. In our design the handshaking protocol precludes deadlocks but the larger is the buffer size the smaller is the probability of communication latency.

Input data to a packet buffer allocated to an output link arrive from the input links corresponding to the other three I/O links of the PE as depicted in figure 4.5.

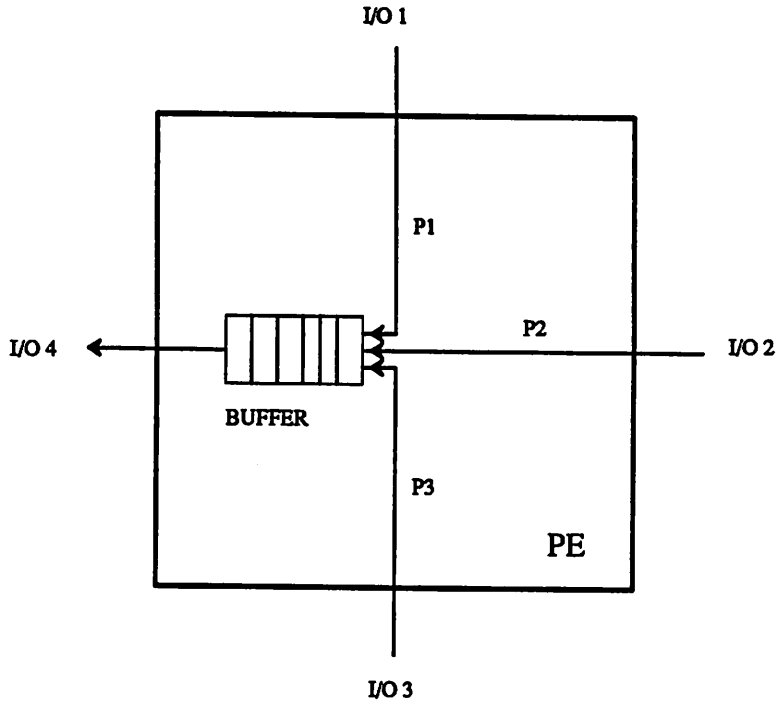


Figure 4.5 - Data input to a packet buffer

Let λ denote the total arrival rate of messages to output link I/O 4. A fraction P_1

of these arrivals are from link I/O 1, a fraction P_2 are from link I/O 2 and a fraction P_3 are from link I/O 3 ($P_1+P_2+P_3=1$). Figure 4.6 depicts a queueing model of an output link with a total combined arrival rate λ and service rate μ.

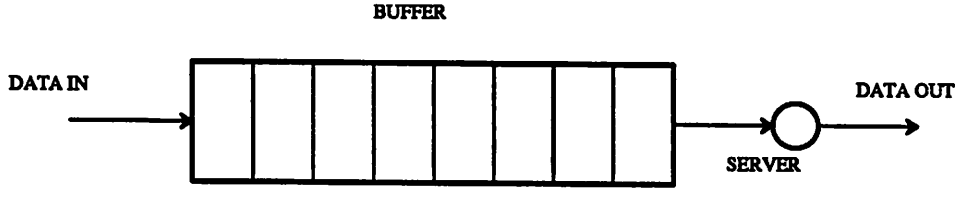


Figure 4.6 - Queueing model of an output link

Assuming that the arrival of messages is a Poisson process and the messages have variable lengths that are exponentially distributed, a M/M/1 queueing model can be used to estimate the buffer size k. The Poisson arrival time and the exponentially distributed message length assumptions allow the use of M/M/1 queues that can be easily solved. As shown in [1], relaxing each of these assumptions results in G/M/1 and M/G/1 queues respectively that are difficult to solve for large, complex multicomputer networks discussed in this dissertation. Furthermore, simulation studies of filters, FFT and LU decomposition programs [2] have shown that relaxation of these assumptions yield different performance but the conclusions drawn from the models are the same.

From queueing theory analysis [1, 3, 4, 5], the probability of having n ($n \leq k$) messages in a buffer for the case of a finite M/M/1/k queue with a link utilization factor of $\rho=\frac{\lambda}{\mu}$ is given by:

$$p_n=\frac{1-\rho}{1-\rho^{k+1}}\rho^n \qquad (4.1)$$

Therefore, the probability of blocking new incoming messages (p_B) equals to the probability that the queue is full (p_k, where k is the maximum number of messages in the buffer).

$$p_B = p_k = \frac{1-\rho}{1-\rho^{k+1}}\rho^k \tag{4.2}$$

The blocking probability p_B depicted in table 4.1 is an appropriate measurement for defining the buffer length k. As expected a larger link utilization factor (ρ) implies a larger buffer space.

ρ	k=1	k=2	k=3	k=4	k=5	k=10
0.9	0.47	0.30	0.21	0.16	0.12	0.05
0.8	0.44	0.26	0.14	0.12	0.09	0.02
0.7	0.41	0.22	0.13	0.09	0.06	0.01
0.6	0.38	0.18	0.10	0.06	0.03	2.4E-3
0.5	0.33	0.14	0.07	0.03	0.01	0.5E-3
0.4	0.29	0.10	0.04	0.02	6.0E-3	6.0E-5
0.3	0.23	0.06	0.02	5.6E-3	1.4E-3	3.4E-6
0.2	0.16	0.03	6.4E-3	1.3E-3	0.2E-3	2.0E-8
0.1	0.09	0.9E-3	9.0E-4	9.0E-5	9.0E-6	1.0E-11

Table 4.1 - Blocking probabilities for different buffer size

The next table (table 4.2) shows the upper bound of the average buffer size, obtained from analyzing a M/M/1 queue with an infinite buffer size.

ρ	k=∞
0.9	9.0
0.8	4.0
0.7	2.3
0.6	1.5
0.5	1.0
0.4	0.66
0.3	0.43
0.2	0.25
0.1	0.11

Table 4.2 - Upper bound of the average buffer size

Even though the upper bound on the average buffer size for different ρ is small, to avoid repetition of data transfer due to full buffers, the buffer size should only be determined from the blocking probability according to the link utilization and the required throughput $\gamma=\lambda(1-P_B$ (where λ is the average number of message arrivals and λP_B is the number of blocked messages).

Since a sender PE has to wait for an end-to-end acknowledgement before a packet can be discarded, the buffer size analysis of the data buffer is different than the buffer size analysis of the packet buffer. The analysis is based on M/M/1 queueing model with the following assumptions:

- Poisson distribution of the arriving messages.
- D - average time to send a message and to get back an end-to-end acknowledgement from the destination PE when the processing time per hop and the average number of hops is given by the scheduler.

- Data is transferred immediately (VCT) - no delays in intermediate network nodes.

Denoting:

1) $\lambda=\frac{messages}{second}$ - arrival rate of messages from the PU.

2) μ - service rate of a node.

3) S - average service time of a node.

$$S=\frac{1}{\mu} \tag{4.3}$$

4) W - average waiting time in a queue.

$$W=\frac{\frac{\rho}{\mu}}{1-\rho} \tag{4.4}$$

5) D - average time to route message to destination PE and an acknowledgement back to the source PE.

6) T - average time in system (including the average waiting time, the average service time and the average routing time between two network nodes).

$$T=D+W+S=D+\frac{1}{\mu-\lambda} \tag{4.5}$$

7) $P_n(T)$ - probability that exactly n messages arrived in a time interval T:

$$P_n(T)=\frac{(\lambda T)^n e^{-\lambda T}}{n!} \tag{4.6}$$

To determine the buffer length of the source PE lets assume that the average time interval between receiving a message to the buffer and discarding it after end-to-end acknowledgement is T. During this time interval (while message is in the system) there is a possibility that new messages arrive to the buffer. Lets assume that α is the probability that n or more arrivals occurred in a time interval T :

$$P_n(T)+P_{n+1}(T)+P_{n+2}(T)+....=\alpha \tag{4.7}$$

Therefore, the probability that less then n messages have arrived during the average time interval in the system (T) is given by:

$$P_0(T)+P_1(T)+.....+P_{n-1}(T)=1-\alpha \tag{4.8}$$

If α equals to the blocking probability (P_B), then for the case when only one message is in the buffer during the average time interval T, the minimum length of the buffer (n) to achieve P_B can be obtained from equation 4.9 by substituting equation 4.6 into equation 4.8 :

$$e^{-\lambda T}\sum_{k=0}^{n-1}\frac{(\lambda T)^k}{k!}=1-\alpha=1-P_B \tag{4.9}$$

This assumes the "best" case when the arriving message finds no other previous messages waiting a head of it (i.e. only one message is waiting in the buffer). If on the average there are n_0 previous messages waiting, then the minimum buffer size must be n found in (4.9) in addition to n_0 the number of previous messages.

Depending on the scheduling of a program, when the traffic is low (ρ=0.1-0.2 as in table 4.1) it is possible to assume that on the average only one message is waiting in the queue.

Since the sender PE has to wait for an end-to-end acknowledgement the data buffer size is larger than the packet buffer size.

4.3. Buffer control & bookkeeping

As stated before, to avoid delays in data transfer as well as deadlocks the buffer for each link should be large. One major problem regarding the packet buffer is how to partition and allocate it optimally for the different I/O links. There are several ways to solve this problem: 1) separate buffers for each link, 2) one common buffer "pool" accessed by every link where the buffer allocation for each link changes dynamically and 3) combination of the two, i.e. separate buffers for each link ("private" buffer)

extended if necessary with part of the common buffer "pool" (if space is available). The first option of separate buffer for each link is inefficient because most of the large "private" buffer space is unused most of the time. The second option, one common buffer, could end up to be large with the possibility that links with low traffic might be left with no buffer space because the links with high traffic have occupied all the space. The third option, depicted in figure 4.7, each link has a "private" buffer which can be extended by a restricted part of the common buffer. This option provides minimal buffer space for low traffic links and larger buffer space for high traffic links and averages fluctuations in buffer space demand. [Equivalently, each link has a minimum allocation of buffers ("private" buffer) which, depending on the buffers allocation to the other links, can be extended by part of the common buffer "pool"].

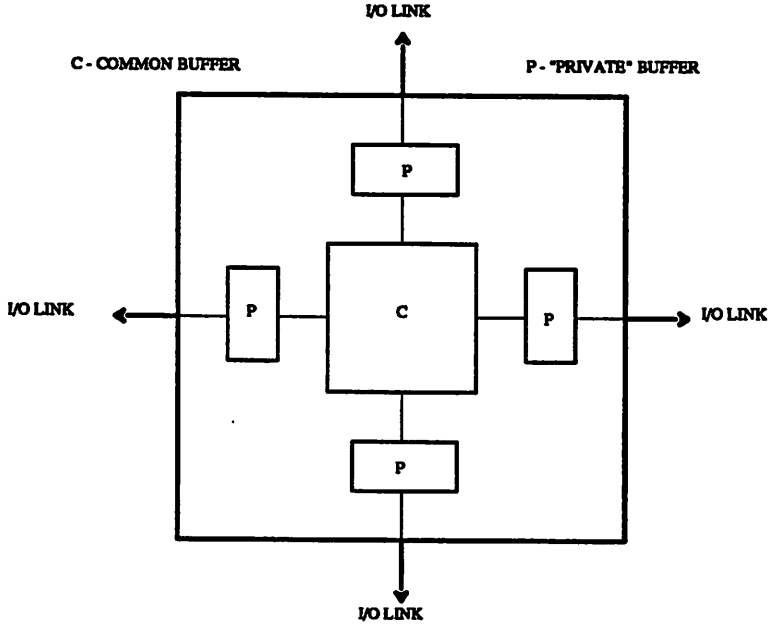


Figure 4.7 - "Private" and common buffer allocation

Irland in his paper [6] evaluated packets blocking probabilities using different shared buffer management techniques. His conclusions were that using a shared buffer of size $\frac{N}{\sqrt{P}}$ where N is the total number of basic packet size and P is the number of

links yields a throughput close to the optimal sharing.

In our case, the total buffer size of each link ("private" buffer and part of the common buffer) was evaluated in table 4.1. To decrease the size of the common buffer, the square-root rule mentioned above is one way to properly estimate the size of the common buffer.

Since the buffer size of each link is limited, a buffer control and bookkeeping unit is needed. This unit has two tasks: 1) locate free buffer to store each new arriving packet, 2) locate next packet to be forwarded through the output link.

Figure 4.8 depicts data and control paths of the packet buffer when the PE operates as a network switching node.

Input data arriving from the other three I/O links is stored in either the "private" buffer of the link, or in the common buffer "pool" depending upon the available storage. Two counter registers are involved in inputing data (receiving mode) or outputting data (transmitting mode). One is an address counter which loads the first address and increments it after each buffer access. The other is a block counter which loads the size of the block to be transferred and is decremented after each buffer access. When the block size buffer is zero the data transfer has been completed. One bit, denoted P/C (figure 4.8), of the address register selects the buffer to be accessed and controls also the data output multiplexer.

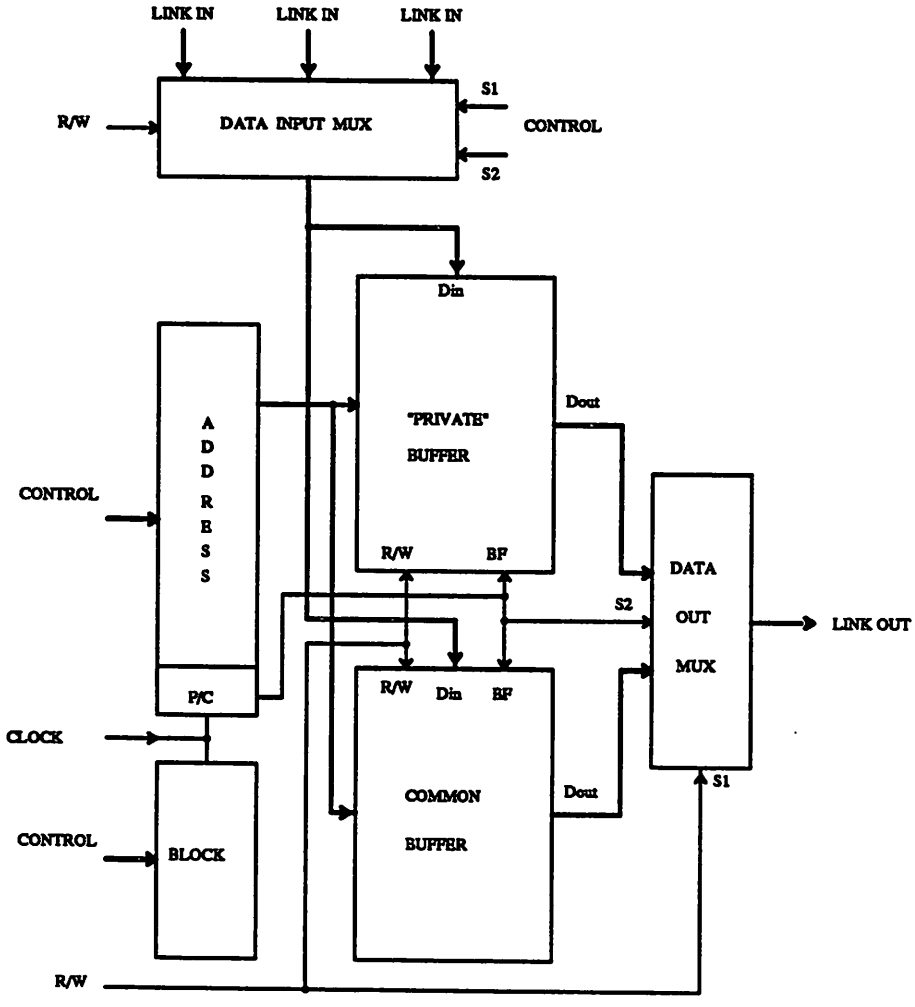


Figure 4.8 - Data & control paths in packet buffers

The control and the bookkeeping of the unit is depicted in figure 4.9.

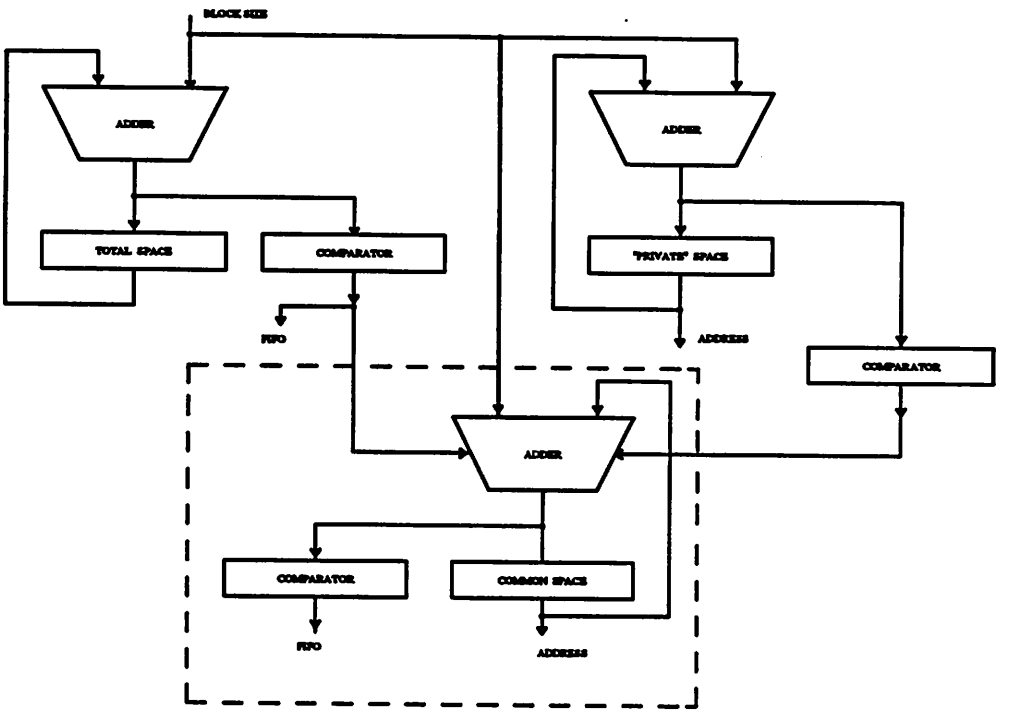


Figure 4.9 - Control & bookkeeping unit

The buffer control unit contains two subunits dedicated for the I/O link and one subunit, of the common buffer, used by all I/O links. Each subunit contains an adder, a register and a comparator. One of the dedicated subunits is used to check whether there is any available space in the link's buffer, and the other to check whether there is available space in the "private" buffer of the link. When there is a request to transfer a data packet, its block size is added concurrently in the two dedicated subunits to determine whether it can be stored in the "private" buffer or in the common buffer. If the "private" buffer has enough space the output of the corresponding adder is loaded to the address counter of the buffer. If the "private" buffer is full, and more space is still available in the common buffer, the block size of the incoming packet is added in the common buffer subunit to determine whether there is any space available in the common buffer. When there is available space in the common buffer the first address generated by this subunit is loaded to the address counter of the buffer. This checking is necessary

because the size of the common buffer is smaller than the sum of the buffer space allowed for all the I/O links. To save bits in the message's header and to simplify the buffer bookkeeping, packet lengths should be limited to multiples of a basic size like 256, 512, 768, ...etc..

Since the space of the common packet buffer is limited, a policy for fair utilization of the common space is required. A policy that guarantees a minimum buffer allocation $\underline{n}$ in the common buffer "pool" that may be extended, if space is available, up to $\overline{n}$ is described below. Given that the common buffer can accommodate N packets of the basic packet size (e.g. 256 bits) and m is the number of I/O links, the buffer space boundaries for each I/O link in the common space is given by:

$$\overline{n} \geq \text{I/O link's space in common buffer} \geq \underline{n}$$

where the lower boundary $\underline{n}$ is given by:

$$\underline{n} \geq \frac{N}{m}$$

and the upper boundary $\overline{n}$ is given by:

$$\overline{n} \leq \frac{N-n}{m-1}$$

For practical realization and implementation $\overline{n}$ and $\underline{n}$ should be integer multiples of the basic data packet length.

4.4. Communication control

The communication control of the AIO, depicted in figure 4.10, establishes and handles according to the protocols the handshaking interconnection and the data transfer from and to the PE. Its basic major components are the following:

- Control and timing unit decodes the command field of the arriving messages and controls the required operations with the correct timing. The clock of the AIO synchronizes its operation to the "syn" field of the arriving messages. A "watch dog" system times out a transmitting operation when there is no response from the

receiving PE or a receiving operation when the data is not being transferred by the sender PE.

- Three FIFOs corresponding to three priority levels store the control information and the communication status of messages involved in handshaking interconnection establishment and data transfer.
- Priority and routing tables used for translating, according to the incoming input link, the arriving message's short ID into: 1) the priority of handling the message, 2) which output link to use for forwarding the message, and 3) what is its new short ID.
- Input packets under process is a temporary storage used for distinguishing between new arriving messages and messages under process whose control information and status is already stored in the appropriate FIFO.

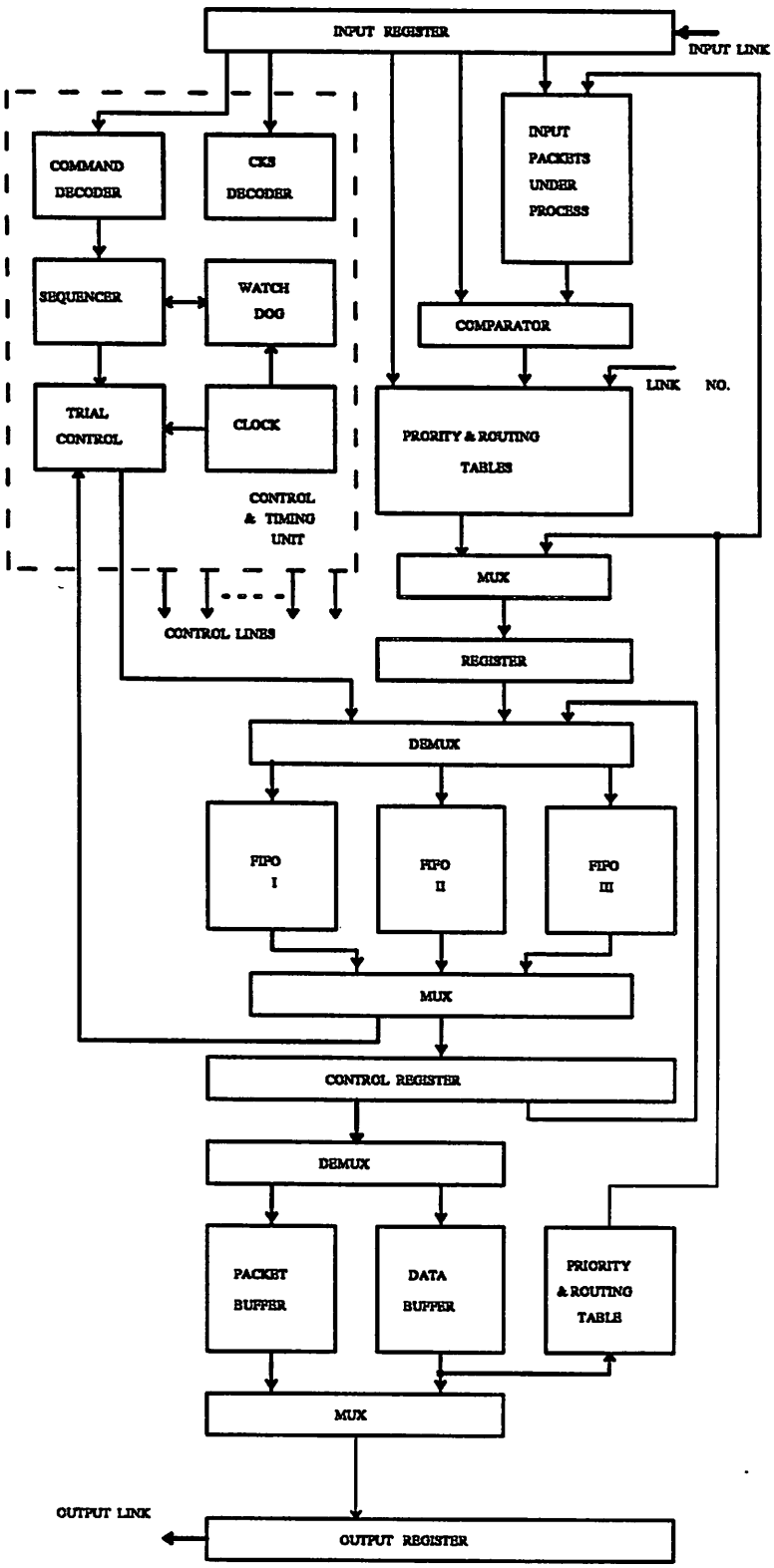


Figure 4.10 - AIO - Communication control unit

4.4.1. Input packets under process

The command field of arriving messages separates them into two types: transmission messages and response messages. RTD (request to transfer data) and DTR (data transfer) are examples of the transmission type, while RDY (ready to receive data) and ERR (error) are examples of the response type. An arriving message might be a transmission message, a response message or a false message that does not belong to either of them (e.g. response when there was no request, unidentified command code, data transfer without initial request etc.). The "Input Packets Under Process" unit (IPUP), depicted in figure 4.10, is used for checking, according to the ID, whether the arriving message is a new one, an illegal one or one which is a part of a handshaking data transfer which is in process. Packet ID and short ID fields of new requests to transfer data messages (RTD/RTSD) either initiated and sent by the current PE or received from another PE, are stored in the IPUP. The IDs of a received request message are stored only if there is enough buffer space available for the data. When a message arrives, its packet and short IDs are compared with the ones in the IPUP. If the comparison is positive, the arriving message is a continuation of a handshaking data transfer already initiated, the translation of the priority and routing lookup table is not necessary and the control unit can immediately check for the message's status in the FIFO. If the arriving message is a new one its short ID and the input link are first translated into service priority and routing path which later with other status bits are stored in the FIFO. When a data transfer has been completed (received or forwarded successfully) its packet and short IDs are removed from the IPUP.

4.4.2. Priority & routing table

To decrease the number of header bits of a message, a short ID is used to define the source and the destination IDs. For appropriate control and message processing the short ID needs to be decoded into message's service priority, source PE and destination

PE IDs, output link to be used if the message has to be forwarded or transferred and a new short ID for the next transfer. The priority and routing lookup table depicted below in figure 4.11 executes this translation.

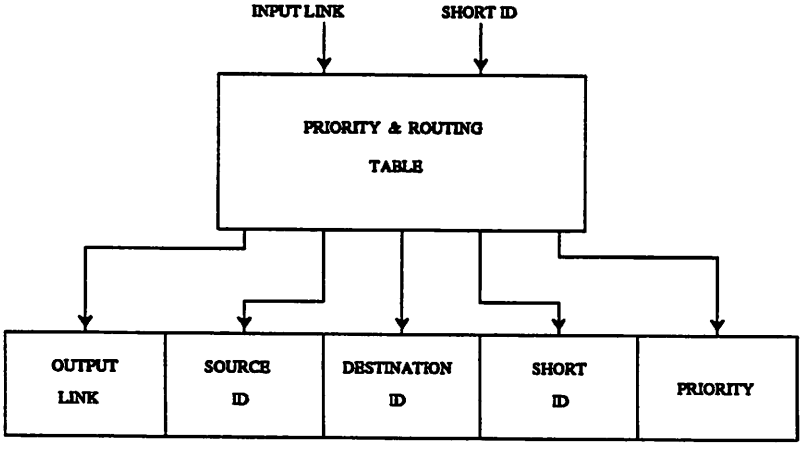


Figure 4.11 - Priority & routing table's output

Its content is determined and defined by the scheduler during the partition of the algorithm and the task allocation to the different PEs. The message's short ID and the input link number through which it arrived are the inputs to the table, and its outputs are the priority of handling the message, the source and destination IDs, the output link for transferring the message, and a new short ID.

Actually, two priority and routing tables are required, one when the PE operates as an intermediate network node that forwards messages and the other when the PE is the source of a message. In both cases, the input is transferred into the output data depicted in figure 4.11. When the PE operates as a network node a lookup table translates, according to the input link, a short ID of an arriving message into the corresponding output data. When the PE is a source PE a lookup table translates a PE's destination ID fetched from the data buffer into the corresponding output data.

4.4.3. FIFOs

Handling and processing messages according to their priorities assigned by the scheduler synchronizes the operation of the PEs, reduces the probability of deadlocks and controls the data flow in the network. Fairness in handling messages might lead to different priority policy: 1) FCFS - first come first serve, 2) Descending order of priority service - messages of higher priority are always served first, 3) Service according to some predefined priority sequence - service is given according to some priority sequence defined by the scheduler (e.g. I,I,II,I,I,III,I,II,III etc.).

Since the priorities of handling the messages are determined and defined by the scheduler during the partition of the algorithm and the task allocation to different PEs, a highest or a sequential priority service policy is an appropriate one to use.

To control and handle the message transfer between the PEs according to the protocol developed before, the status of the messages which are in the process of interconnection establishment or information transfer has to stored. The three FIFOs, depicted in figure 4.10, stores the status information of different data transfers which are being under process. Each FIFO stores the status information of the messages with the corresponding priority level, i.e. FIFO I stores the information about messages with priority I, FIFO II stores the information about messages with priority II and so on. The FIFOs can be implemented by circular shift registers or associative memories. An attribute in the FIFO is cleared upon success of data transfer or returning the data to the PE from where it had arrived. Each status word contains 16 attributes of control information. Figure 4.12 below shows the information about the data necessary for handling its transfer such as: IDs, buffers, location of the data and its length.

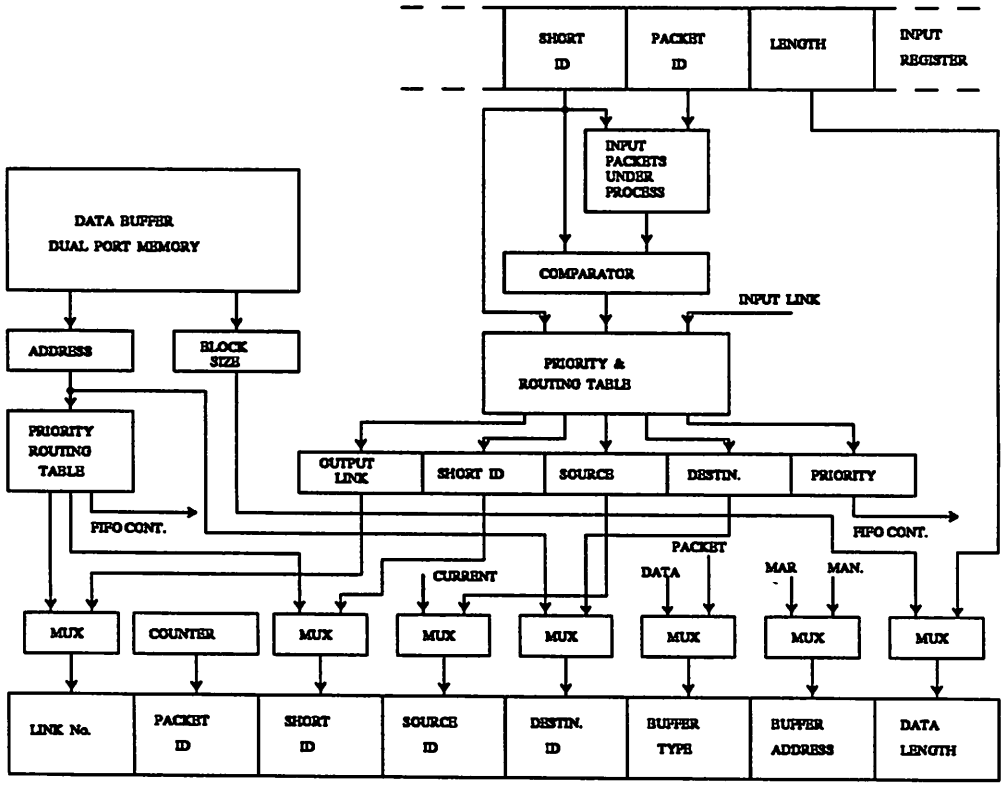


Figure 4.12 - Information of data for handling its transfer

- Link number is the output link to be used for forwarding the data. The link fetched from the priority and routing table is defined by the scheduler.
- Packet and short IDs are header fields of the messages transmitted to the next PE. The packet ID defined by a counter is necessary to distinguish messages transferred between the same source and destination PEs. The short ID fetched from the priority and routing table saves header bits in the message and defines the IDs of the source and destination PEs.
- Source and destination IDs translated by the priority and routing table from the short ID of an arriving message.
- Buffer type, buffer address and data length define how words of data have to be transferred and from where to fetch them.

Figure 4.13 contains the status information for controlling the handshake interconnection and the data transfer according to the protocol.

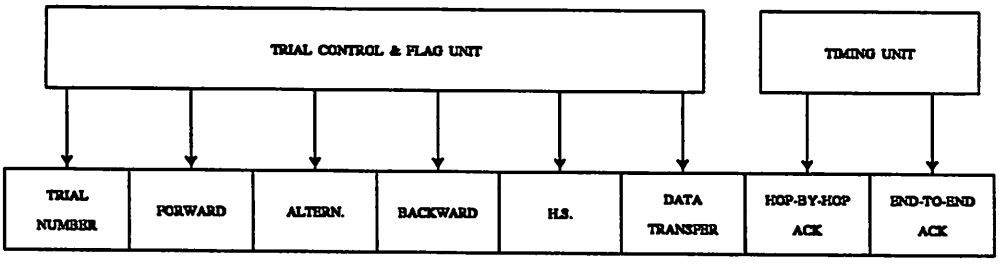


Figure 4.13 - Control information for data transfers

- Trial number indicates the number of attempts to establish an interconnection or to transfer data. This data is fetched from a modulo 3 counter.
- Forward, alternative and backward flags show the stage of the transaction. At any time only one of the flag is set. The flag status and the trial number information allow the control unit to determine the next steps to be taken, according to the protocol, if the transaction fails.
- H.S. (hand-shake) and data transfer flags show whether the transaction is in the interconnection establishment state or in the data transfer state.
- Hop-by-hop acknowledgement is the time information latched from a free running counter, that shows when the message was transferred to the next PE. Comparing this count with the updated count of the free running counter is the "watch-dog" operation that times out transactions if and when there was no response from the next PE.
- End-to-end acknowledgement is the time information latched from a free running counter, that shows when the message started its route to the destination PE. Comparing this count with the updated count of the free running counter is the "watch-dog" operation that times out transactions if and when there was no response from

the destination PE.

As mentioned previously, the status data is important and necessary to define the header of the message and to control the transaction's steps defined by the protocol. Any response or lack of response from the receiving PE is followed by moving to the next state of the protocol, updating of the appropriate flags, and retransmission of the message again. A status word is aborted from the FIFO in the following cases:

1) Data packet has been received successfully by the destination PE if the current PE is the source.

2) Data packet has been received successfully by the next PE if the current PE is a switching network node.

3) Due to failure in forwarding a data packet, it has been returned to the preceding PE which has forwarded it to the current one.

4.4.4. Control and timing unit

The control and timing unit depicted in figure 4.10 incorporates a command decoder, a CKS (check sum) decoder, a sequencer, a clock, a "watch-dog" and a trial controller.

The command decoder decodes the command field of the arriving messages. If the command is legal, its output is transferred to the sequencer for continuing the process. When an illegal command code arrives the decoder aborts the whole message.

The CKS decoder checks for errors in the arriving messages. Its output is fed to the sequencer for determining the next operations. Depending on the environment (S/N ratio) and the implementation, the CKS decoder might execute a simple one bit parity check, a LRC check sum (several column's parity check sum) or a CRC (cyclic redundancy code) check. The first two parity sums are simple to implement (T flip-flop and random logic) and can be done while the messages arrive thus, saving time. The CRC

check is more complicated to implement and its execution requires an additional time after the message has arrived.

The sequencer implemented by a PLA or random logic receives at its input the decoded command and the parity check's result. It fetches the status word from the FIFO and provides sequentially the appropriate controls lines required for the next operations. Updating the flags and transition to the next state according to the protocol is executed during these operations.

Receiving sequence

In the receiving mode of a PE the sequence of operations executed by the AIO depends upon the type of the arriving message. The operations for the different types is summarized in the following table:

sync clock		
input message		
decode command		
check errors (CKS)		
"RTD"	"DTR"	Control
check IPUP	check IPUP	check IPUP
P & R table	check W.D.	P & R table
fetch FIFO	P & R table	check FIFO
check buffer	fetch FIFO	
store IPUP	store buffer	
assign buffer	store FIFO	
store FIFO		
set W.D.		

Table 4.3 - Receiving mode - sequence of operations

Transmitting sequence

In the transmitting mode of a PE the sequence of operations executed by the AIO depends upon the type of the transmitted message. The operations for the different types is summarized in the following table:

choose FIFO	
fetch FIFO	
reset "CKS"	
Control message	Data packet
output message	output header
start W.D.	fetch data & output
update flags	output tail
store FIFO	start W.D.
	update flags
	store FIFO

Table 4.4 - Transmitting mode - sequence of operations

The trial controller depicted in figure 4.14 updates the state of the message transfer according to the response from the receiving PE. Its operation is based upon a modulo three counter. Initially the counter and the flags are reset. The first attempt to transfer a message (handshaking interconnection establishment or data transfer) results in incrementing the counter and setting the forward flag. If the message was not transferred successfully after three attempts the operation mode is transferred from one state to the next one, e.g. forward -> alternative -> backward. A successful message transfer results in resetting the counter and the flags.

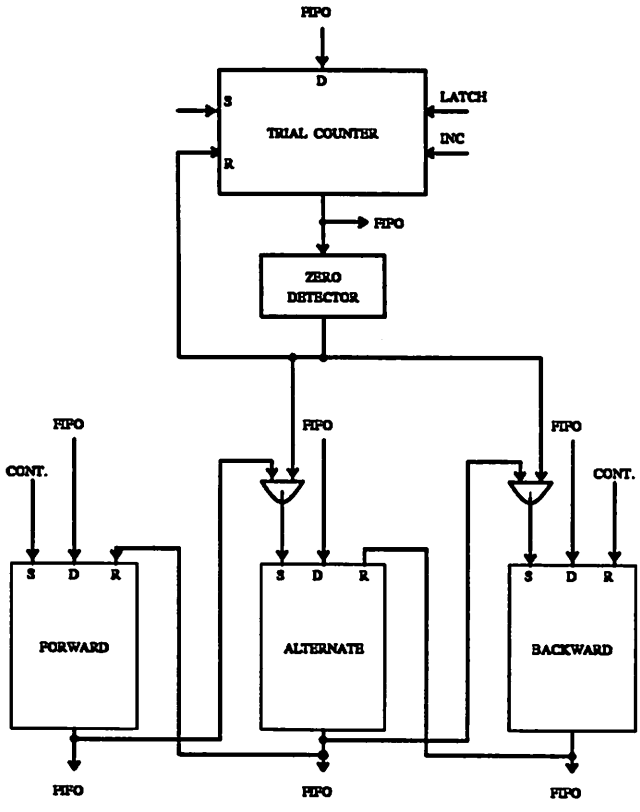


Figure 4.14 - Trial controller unit

A "Watch-dog" unit depicted in figure 4.15 is implemented with a counter triggered by the main clock (time stamp counter), an ALU and a comparator. The unit determines whether to time out the transaction or not according to the current time stamp and the time stamp of the message indicating when it was sent. Such an implementation avoids multiple dedicated counters for each message which is under transaction, and allows the same system to be used for "watch-dog" time out.

Main clock is a quartz free running clock which can synchronize its phase and frequency to the "SYN" field of arriving messages. Chapter 3.2 describes the ways to implement this feature.

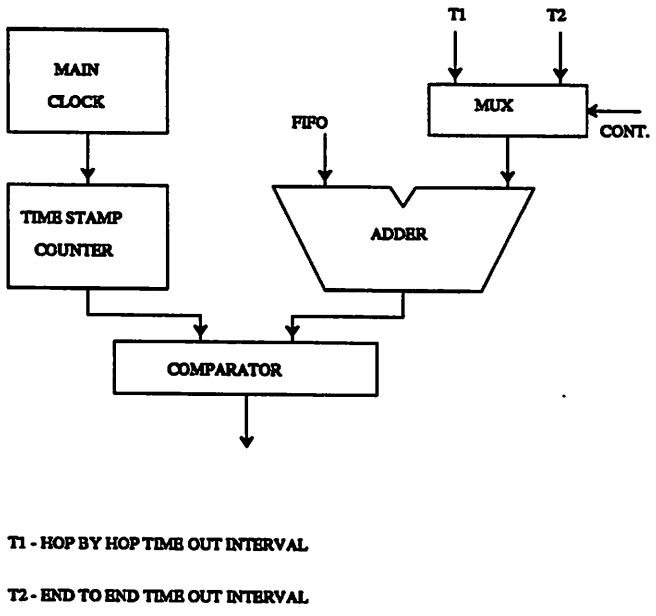


Figure 4.15 - "Watch-dog" system

To enable concurrent message transfers through the separate unidirectional output and input lines of an I/O link, the control register which latches the FIFO's output has to be duplicated. One control register should be dedicated for the status of a receiving message and the other for the status of a transmitted message. By doing so the control unit saves frequent searches in the FIFOs and the implementation is simplified.

4.5. ASIC properties

Advance in μP VLSI design and fabrication makes it feasible to implement on the same chip a processing element (PE) that contains a processing unit (PU) and an autonomous I/O unit (AIO). Independent and concurrent operations of the PU and the AIO without the involvement of the PU in the network message's transfer are very suitable for ASIC (Application Specific IC) implementations. The simple and standardized interface between the PU and the AIO and similar protocols for different communication configurations yield the following advantages:

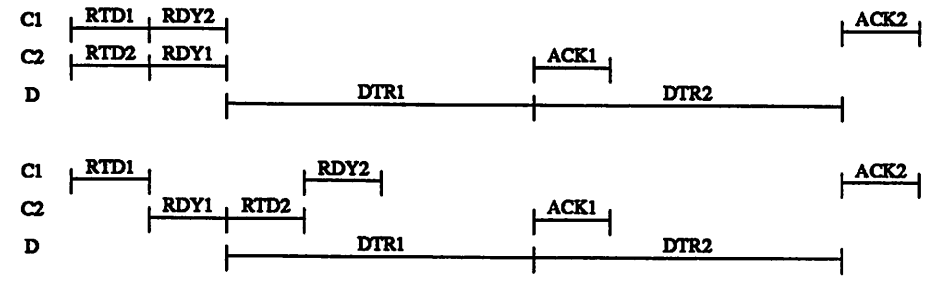
- Depending on the application, every PE can accommodate a different computing

unit and a different communication configuration that are mostly suitable and proper for the particular application.

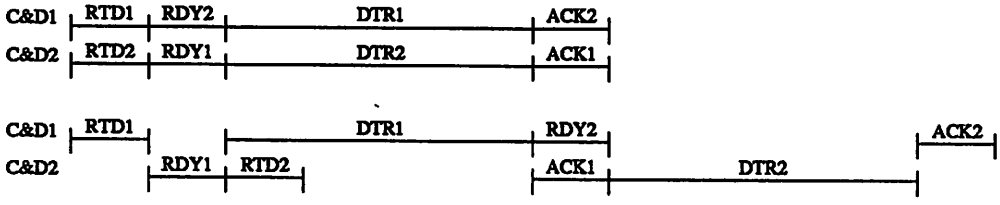
- Simple implementation of "heterogenous" systems where different PEs incorporate different processing units.
- Modularity - A multiprocessor system can have PEs which accommodate different types of processing units and/or different communication configurations.
- Parametrizable - The number of lines in an I/O link of each communication configuration can be extended from one serial line to any number of parallel lines.
- Extensible - Four I/O links enables the PE to be employed in any network and provide simple expansion to a large multiprocessor configuration.
- Higher B.W. (bandwidth) - Up to four I/O links can interconnect two adjacent PEs.

4.6. I/O link's utilization

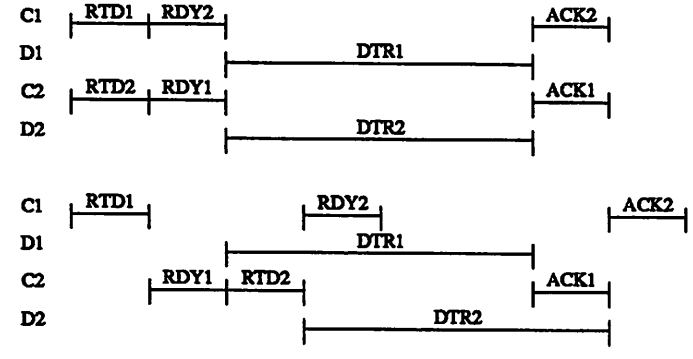
Chapter 3.9 describes three I/O link configurations. Configuration I incorporates two unidirectional control lines and one bidirectional data line. Configuration II incorporates two unidirectional control and data lines. Finally, configuration III incorporates two pairs of unidirectional lines, one for control and the other for data. One major issue to consider is how to execute message transfers with maximum I/O link utilization. Figure 4.16 addresses this issue. Assume that two PEs connected through an I/O link want to transfer a packet of data between them. For each configuration, two cases of controlling data transfer are investigated. In the first case (the upper part of each configuration in figure 4.16) both PEs simultaneously initiate a data transfer (RTD_1 and RTD_2) from one to the other, and in the second case (lower part of each configuration in figure 4.16) the second PE initiates its data transfer (RTD_2) only after responding to the initiation of the first PE (RDY_1).



Configuration I



Configuration II



Configuration III

Figure 4.16 - I/O configurations - Link utilization

It is clear from figure 4.16 that configuration III, the configuration with the largest bandwidth, will yield in both cases the higher data transfer rate, while configuration I which has only one line to transfer data will yield in both cases the lowest data transfer rate. But, if the control system of the AIO initiates first its own data transfers before responding to initiations from other PEs (first case), the data transfer rate of

configuration II will be as high as that of configuration III. For such cases the bandwidth of configuration II is better utilized. Therefore, to achieve a higher data rate transfer the scheduler and the sequencer implementation must first initiate its own data transfer before responding to initiations from other PEs.

4.7. PE's performance

4.7.1. Multiprocessor performance

Comparing the running time of a program on a multiprocessor system with its running time on a single processor is a good measurement of the performance improvement obtained by the multiprocessor system. To do an appropriate comparison, a standard task unit (a standard program unit) is defined. Assuming that the computation time of the standard task on a single processor is P time units, and the communication overhead time incurred by data transfers with other PEs during the execution time is C, the ratio $\frac{P}{C}$ is a measure of how much communication overhead is incurred per computation of a standard task. The potential of obtaining a higher performance with a multiprocessor system increases as the ratio $\frac{P}{C}$ is higher.

Performance analysis of multiprocessor systems based upon the analysis in[7] shows that the execution time of a program consisting of M units of standard task which is partitioned into two PEs is given by:

$$Execution\ time = P \cdot \mathrm{Max}\{M-k, k\} + C(M-k)k \qquad (4.10)$$

The first term is the longest execution time between the two PEs when k units of standard task is assigned to one processor and M-k to the other. The second term is a pairwise communication overhead (not overlapped with the computation time) that must take place as a function of how the tasks are partitioned to the processors.

Equation 4.10 can be extended to the case where the same program consisting of M units of standard tasks is partitioned and allocated to N PEs. Allocating k_i standard

task units to the corresponding i^{th} processor yields the following execution time for a fully connected multiprocessor system:

$$Execution\ time = P \cdot \mathrm{Max}\{k_i\} + \frac{C}{2}\sum_{i=1}^{i=N} k_i (M - k_i) \tag{4.11}$$

When the multiprocessor system is load balanced the tasks are equally divided between the N processors thus, the tasks are equal i.e. $k_i = \frac{M}{N}$ and the execution time of the program is given by:

$$Execution\ time = P\ \frac{M}{N} + \frac{C}{2}(M^2 - \frac{M^2}{N}) \tag{4.12}$$

The speed up attributable to parallel execution of a multiprocessor system is defined by the ratio of the execution time of the program on one processor over the execution time of the program in a multiprocessor system with N processors operating concurrently. Evaluating the ratio for a load balanced PE multiprocessing system yields:

$$Speedup = \frac{PM}{\frac{PM}{N} + \frac{C}{2}(M^2 - \frac{M^2}{N})} \tag{4.13}$$

In our proposed implementation, the multiprocessor system is not fully connected and therefore it is necessary to add the number of hops in the second term of equation 4.11.

$$Execution\ time = P \cdot \mathrm{Max}(k_i) + \frac{C}{2}\sum_{i=1}^{i=N} k_i \sum_{\substack{j=1 \\ j \neq i}}^{j=N} k_j I_{ij} \tag{4.14}$$

where I_{ij} is the number of hops from PE_i to PE_j.

When the multiprocessor system is load balanced, the tasks are equally divided between the N processors thus, the tasks are equal to $k_i = \frac{M}{N}$, and when the communication requires an average number of hops $I_{ij} = \bar{K}$, the execution time of the program is given by:

$$Execution\ time = P\ \frac{M}{N} + \frac{C}{2}(M^2 - \frac{M^2}{N})\bar{K} \tag{4.15}$$

The second term of the execution time (equation 4.15) is due to the additional communication overhead. There are two important factors, C and $\overline{K}$. When C and/or $\overline{K}$ are small, the communication overhead is reduced abd therefore the system's throughput is increased. C is small when the data transfer is independent and parallel to the computations, and $\overline{K}$ is small when the average number of hops is small.

The speedup defined as before is given by:

$$Speedup = \frac{PM}{\frac{PM}{N} + \frac{C}{2}(M^2 - \frac{M^2}{N})\overline{K}} \tag{4.16}$$

Equations 4.15 and 4.16 clearly show the expected result that high throughput of a multiprocessor system is achieved if:

- The algorithm contains high degree of parallelism (large N corresponds to smaller processing time $P\frac{M}{N}$).
- Balanced load PE - The program is partitioned into same length tasks for the different PEs ($k_i = \frac{M}{N}$).
- Communication overhead time compared to the execution time of the task is negligible (C is small).
- Minimum number of hops ($\overline{K}$ is small).

Since partitioning and scheduling is an NP complete problem, it is very difficult to partition a program and schedule it perfectly with load balanced PE's and minimum communication among PEs. Therefore, partitioning a processing element (PE) into two units operating concurrently and independently, one that executes the computational tasks and the other that executes the interprocessor communication, reduces the communication overhead and improves the multiprocessor throughput.

To illustrate the reduction of the communication overload on the computational task, an analysis of interprocessor data transfer will now be made between the PE

developed in this dissertation and two commercial signal processing processors: Motorola's DSP-56000 and INMOS's Transputer. The analysis compares the time, in clock cycles, it takes to transfer data packets between two processors and the communication overload imposed on the computational part.

The data transfer comparison is based upon the handshaking protocol developed in the previous chapters. Similar subroutines are used for comparing the different processing elements. A control message is assumed to be 32 bits and a data packet including the header is assumed to be 1 Kbit. Data is transferred through a serial output link at a rate of one bit per clock cycle.

4.7.2. Motorola 56000

4.7.2.1. Hand shake subroutines

The handshaking data transfer is initiated by the sender PE with the following subroutine:

Operation	clock cycles
Move M->A	2
Move A->SCI	2
Message transfer	24
Interrupt	1
Move M->A	2
Move A->SCI	2
RTI	4
Decode	8
Total	45

The receiving PE responds with "ready" or "not ready" after the message have

been detected by him. Detection starts while the message is received but there is still a non-overlapping detection time of 5 clock cycles - the last 3 instructions in the following subroutine:

Operation	clock cycles
Data transfer	24
Interrupt	1
Move SCI->A	2
Move A->M	2
RTI	4
Decode	8
Interrupt	1
Move SCI->A	2
Move A->M	2
Total	45

Since there is a non-overlap of 5 clock cycles between the sender and the receiving PEs the hand-shake interconnection establishment is executed in 45+5+45+5=100 clock cycles.

4.7.2.2. Data transfer subroutines

DSP-56000 is a 24 bit processor which implies that a packet of 1Kbit data resides in 42 memory locations. The following is the sender's PE subroutine:

Operation	clock cycles	comments
Do loop	6	
Move M->A	2	
Move A->SCI	2	

Jcc	4	jump if loop done
RTI	4	
Interrupt	1	
Jmp	4	jump to do loop
Data transfer	1000	
Total	1720	

The total execution time of the subroutine is: 17 non-overlapping clock cycles for receiving it).

Do	6
Loop	17×42=714
Data transfer	1000
Total	1720

The receiving PE's subroutine is similar but involves only the loop for inputing the data:

Operation	clock cycles
Do loop	6
Move SCI->A	2
Move A->M	2
Jcc	4
RTI	4
Interrupt	1
Jmp	4
Total	720

The total count includes 714 clock cycles of the loop and 6 clock cycles of the "DO" loop instruction.

These subroutines show that transferring 1 Kbits of data requires 1737 clock cycles (1720 for data transmitted + 17 non-overlapping clock cycles for receiving it). The total data transfer without decoding is executed in 1837 clock cycles as follows:

Handshaking	100
Transfer data	1720
Receive data	17
Acknowledgement	50
Total	1887

For an average of two clock cycles per instruction, both PEs, the sender and the receiver, waste the equivalent of about 900 instructions for decoding the message's control fields, checking the buffer space availability and transferring the data.

4.7.2.3. Control fields

Preparing headers

Operation	clock cycles	comments
CLR A	2	
ORI Syn1111	2	
Move A->M	2	
CLR A	2	
ORI 1111END	2	
Move A->M	2	
Move M->A	2	read destination
Rotate	2	

AND M	2	
Move A->M	2	
Move M->A	2	read Packet ID
Rotate	2	
AND M	2	
Move A->M	2	
Move M->A	2	read length
Rotate	2	
AND M	2	
Move A->M	2	
Move M->A	2	
ANDI 0	2	
Move M->A	2	read 24 msb of header
Do	6	24 repetitions
Rotate	2	
JNC	4	jump if not carry
ORI 1	2	(M)+1
NOP	2	
Move M->A	2	read 8 lsb of header
Do	6	8 repetitions
Rotate	2	
JNC	4	Jump if not carry
ORI 1	2	(M)+1
NOP	2	
Rotate	2	check sum
Move A->M	2	
Move M->A	2	add check sum

OR M	2
Move A->M	2
Total	386

Checking buffer space

Operation	clock cycles
Move M->A	2
ANDI	2
Move A->R	2
JMP	4
Move M->A	2
CMP	2
Header	386
Total	400

Checking CKS

Operation	clock cycles	comments
Do	6	42 times
XOR M+	4	
Do	6	24 times
Rotate	2	
JNC	4	jump if not carry
ORI 1	2	1+(M)
NOP	2	
Total	420	

4.7.2.4. Data transfer time

The above subroutines yield an approximate estimate of the time that it takes to transfer a data on the Motorola 56000 DSP.

Operation	Clock cycles
RTD	436
RDY/NRDY	450
DTR	2106
ACK	403
CKS	420
Total	3815

4.7.3. Transputer

4.7.3.1. Subroutine's execution time

The transputer is a 32 bit processor with a separate I/O which fetches data from the memory through DMA and handles the data transfer between processors. Each byte of data which is transferred with additional overhead bits must be acknowledged by the receiving PE.

Using the instruction's execution time defined in the data-sheet the handshaking is established in:

Operation	clock cycles
RTD	13×4+4×5=72
RDY/NRDY	72
Total	144

Data transfer is obtained by similar calculations. DTR - is executed in 72 clock cycles , fetch 1 Kbits from memory and output them to the next is executed in: 125×13+125×5=2250 clock cycles, and data acknowledgement in 72 clock cycles.

Using the same subroutines for preparing the control fields of a message yields the following execution times:

Header preparation - 624 clock cycles.

Buffer space availability check - 24 clock cycles.

Check sum - 750 clock cycles.

4.7.3.2. Data transfer time

The total time that it takes to execute the whole data transfer including the control fields preparation and check is summarized in the following table:

Operation	clock cycles
RTD	696
RDY/NRDY	720
DTR	2250
ACK	676
CKS	756
Total	5098

4.7.4. Proposed PE

Partitioning the processor element (PE) into a processing unit (PU) and autonomous I/O (AIO) unit that operate independently and concurrently provides the separation between the computation and the communication tasks. Such an implementation eliminates wasted computation time on interprocessor communication and data transfer. Hardware implementation for check sum, routing paths and buffer control reduces the time required to transfer messages or data between PEs.

The operation executed in transferring a packet of data are as follows:

- PU notifies the AIO by an *OUT* instruction that data is ready and available to be transferred.
- AIO accesses the dual-port-memory and fetches from two successive locations the destination's address and the data block size.
- The destination address is translated by tables to output link number, short ID and priority level.
- The output link number, short ID and priority level is transferred to the buffer management and control unit for further evaluation.

The above operations are executed in 5 clock cycles.

Buffer management and control unit issues a control message composed by the following program (using the same timing as Motorola 56000):

Operation	clock cycles
Move priority->A	2
Move A->Pointer	2
Jump relative to P	4
Move link's status to A	2
ANDI priority mask	2
Jump if priority	4
Jump if no priority	4
OR part of header	2
OR part of header	2
Transfer control message	32
Total	52

The parity check sum adds one clock delay because it is executed by a simple

hardware (counter/counters with random logic) on the fly while data is received.

Before the receiving PE responds with ready or not ready it must first check the availability of buffer space. As before the header which is translated by tables is transferred to the buffer management and control unit for further evaluation. Therefore responding back "RDY"/"NRDY" is executed in 52+2+1=55 clock cycles.

Transferring a data packet of 1 Kbits will take 1010 clock cycles where 1000 clock cycles is for the actual data transfer and 10 clock cycles is for the control fields (IDs, CKS, etc.).

The total time that it takes to execute a data transfer including the handshaking interconnection is:

Operation	clock cycles
Transmitter hand-shake	57
Receiver hand-shake	55
Data transfer	1010
Acknowledge data	55
Total	1167

4.7.5. Performance comparison

The results obtained above show that the proposed PE executes data transfer three to four times faster than the commercial DSPs analyzed in the previous paragraphs. But there is a bigger advantage because there is no wasted computation time by the PU, e.g. one wasted clock cycle in the proposed PE compared to thirty eight houndreds in the commercial DSPs. Table 4.5 depicted below summarizes these results:

Processor	Communication [clock cycles]	CKS [clock cycles]	Data transfer [clock cycles]	PU wasted time [clock cycles]
PE	1166	1	1167	1
Motorola 56000	3395	420	3815	3815
Transputer	4342	756	5098	3798

Table 4.5 - Performance comparison

Comparing the results of table 4.5 with equations 4.15 and 4.16, clearly show the advantages of implementing a multiprocessor system with the proposed PE. Since the communication overhead time (C in equations 4.15 and 4.16) of the proposed PE is much smaller than that of the commercial DSPs, the throughput and the speedup obtained by using the proposed PE is higher. But, since data transfer is executed independently and in parallel to the computations, the communication overhead time wasted by the processing unit is negligible ($C \rightarrow 0$ in equations 4.15 and 4.16) and therefore the throughput and the speedup achieved by using the proposed PE is even higher.

4.8. PE's properties - summary

The PE proposed in this dissertation has many properties that makes it appropriate for a variety of different multiprocessor systems:

1) Independent and concurrent computation and communication.

2) No involvement of computation unit in communication.

3) Macrocell for "ASIC" implementations:

   - Modular and parametrizable.

- Processing unit (PU) adjustable to the application.
- Communication configuration adjustable to the application.

4) Similar and simple protocols for different communication configurations.

5) Extensible to large multiprocessor configuration.

6) Adaptable to wide variety of applications.

7) Fast communication between PEs - Virtual-cut-through (VCT) switching with minimal number of hops.

8) Interconnection is established by handshaking.

9) Independent of network topology - four I/O links enable the PE to be embedded in any network topology.

10) Increased communication BW - up to four interconnection I/O links can be connected between PEs.

11) Two types of buffers, one is dual port memory for simple uninterfered interface and data transfer between PU and AIO and the other is for interprocessor communication.

References

1. L. Kleinrock, in *Queueing systems*, John Wiley & Sons, New York, 1975.

2. R.M. Fujimoto, "VLSI communication components for multicomputer networks," in *Ph.D. thesis, Department of EECS, University of California* , Berkeley, 1983.

3. J.F. Hayes, in *Modeling and analysis of computer communication networks*, Plenum Press, New York, 1984.

4. D. Gross and C.M. Harris, in *Fundamentals of queueing theory*, John Wiley & Sons, New York, 1974.

5. R.B. Cooper, in *Introduction to queueing theory*, The Macmillan company, New York, 1972.

6. M.I. Irland, "Buffer management in a packet switch," *IEEE Transactions on Communications*, vol. COM-26, no. 3, March, 1978.

7. H.S. Stone, in *High performance computer architecture*, Addison-Wesley publishing company, Reading, Massachusetts, 1987.

CHAPTER 5

Asynchronous Processor's Concepts & Analysis

5.1. Introduction

Down scaling of the feature sizes in integrated circuits increases the speed of the switching circuits. In 3 micron technology the gate propagation delay is 2 nsec, while in 1.25 micron technology (General Electric) and 0.8 micron technology (Bell Labs) the gate propagation delay decreases to 0.5 nsec and 0.16 nsec, respectively. Although there is an increase in the logic speeds of the switching circuits due to the down scaling, the achievable data processing throughput has not been increasing at the same rate. Researchers observed that the major limitation is due to the global synchronization and the clock skew in multi-phase clocked control[1], and to the basic problem of driving a large capacitive load on the clock line which can vary due to fabrication gradients. Architectures based on local properties like globally-asynchronous locally-synchronous systems[2, 3, 4], as well as carefully designed distribution of the global clock [5] were proposed to increase the computation speed, but the throughput rate has not increased to the extent expected from the scaling rules.

To overcome the clock skewing problem, which substantially reduces the throughput of any synchronous processor, much research is being done on methods to design reliable asynchronous circuits that communicate through handshaking at the completion of each task[6, 7, 8]. In the past, the use of asynchronous processors was less extensively used due to difficulties in designing simple circuits which overcome *hazard* and *race* conditions embedded in asynchronous logic circuits design and due to

substantial overhead in area size and propagation delay. A *hazard* is a transient state where an output of a combinational network is temporarily in error. A *race* occurs in sequential networks where more than one input signals changing at the same time causes a steady-state incorrect output.

However, in the last few years much research work has been done in:

- Designing reliable asynchronous logic circuits such as: control units[9, 10], sequential machines[11, 12, 13, 14], FIFO[7], feedback networks[15] and arbiters[16, 17, 18].
- Developing automatic design tools (CAD tools) which synthesize the asynchronous logic circuits from a high-level functional description [6, 19, 20], and eliminate the unfavorable properties of the such circuit design.

In designing and implementing an asynchronous processor architecture, no clock is required since the functional blocks are built of asynchronous circuits and the interconnection among them is done by handshaking. Such asynchronous processors eliminate the limitation on throughput imposed by the use of a clock and therefore the throughput should theoretically increase at the same rate as the logic circuit speed. In the asynchronous processor, the execution time (propagation delay) of the circuit implementation is data and instruction dependent and therefore the "average" throughput of the asynchronous processor will increase.

5.2. Asynchronous design approach

The use of automatic synthesis CAD tools enables the separate design of the functional computing blocks and the interconnection data transfer blocks. The asynchronous implementation of the computing and the interconnection blocks is based upon reliable asynchronous circuits with minimal area overhead and response time. Data transfer between computation blocks is done by handshaking through interconnection blocks. There are different types of interconnection blocks such as multiplexer (MUX),

demultiplexer (DMUX), fork (FORK), merge (MERGE), full hand-shake (FHS) etc. Figure 5.1 depicts a simple processor architecture which incorporates computation blocks such as MUL, SHIFT, ALU etc. and interconnection blocks such as FORK, MUX, DMUX and FHS.

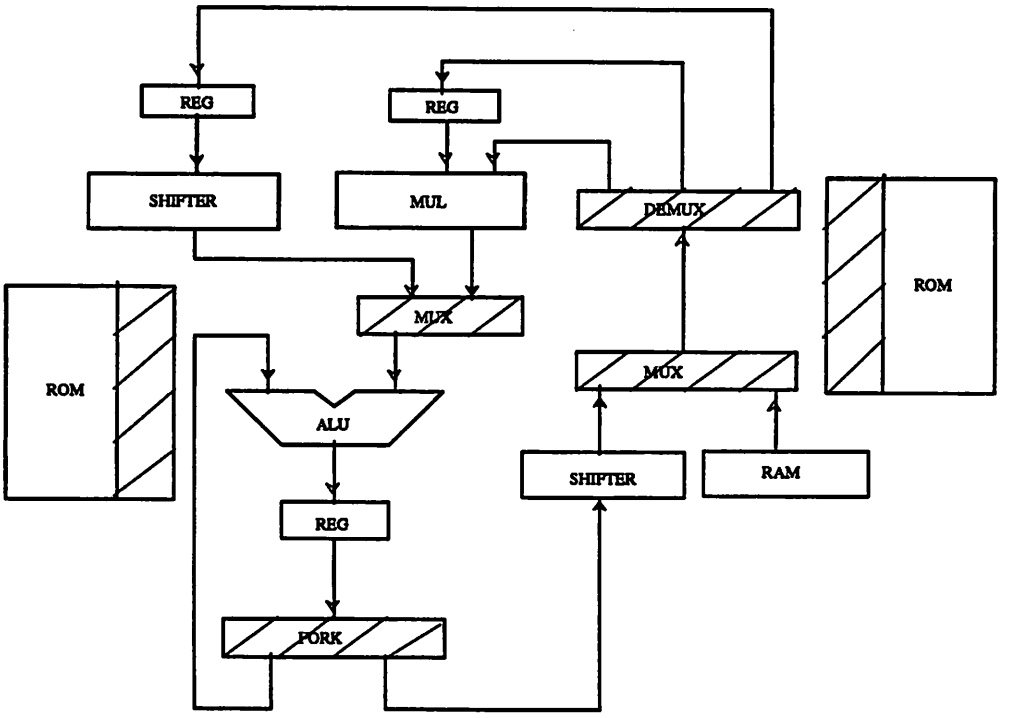


Figure 5.1 - Asynchronous processor

The design of interconnection circuits which perform the handshaking between the computation blocks [6] is based upon self-timed circuits[21, 22] which are delay-insensitive, i.e., their behavior do not depend on the speed of the elements or on the relative communication delays among them. Self-timed logic is a method for managing the complexity of the asynchronous connections between the system elements. Its correct operation is based on a request-acknowledgement protocol which guarantees that a module remains inactive until its input is available, and that the input remains available for as long as it is required. The request-acknowledgement cycle is similar to a two phase clock of a synchronous implementation. The design of the interconnection

blocks is decoupled from the design of the computation blocks and can be done by specifying their functionality and using the algorithm developed in[6].

Since there are no clocks for timing each operation, a completion signal must be generated once a computation block finishes its task. Implementation of the computation blocks by circuits of the DCVSL logic family ("Differential Cascode Voltage Switch Logic") described in [23,24,25] allows simple generation of the completion signal. The schematic diagram in figure 5.2 depicts the generation of the completion signal.

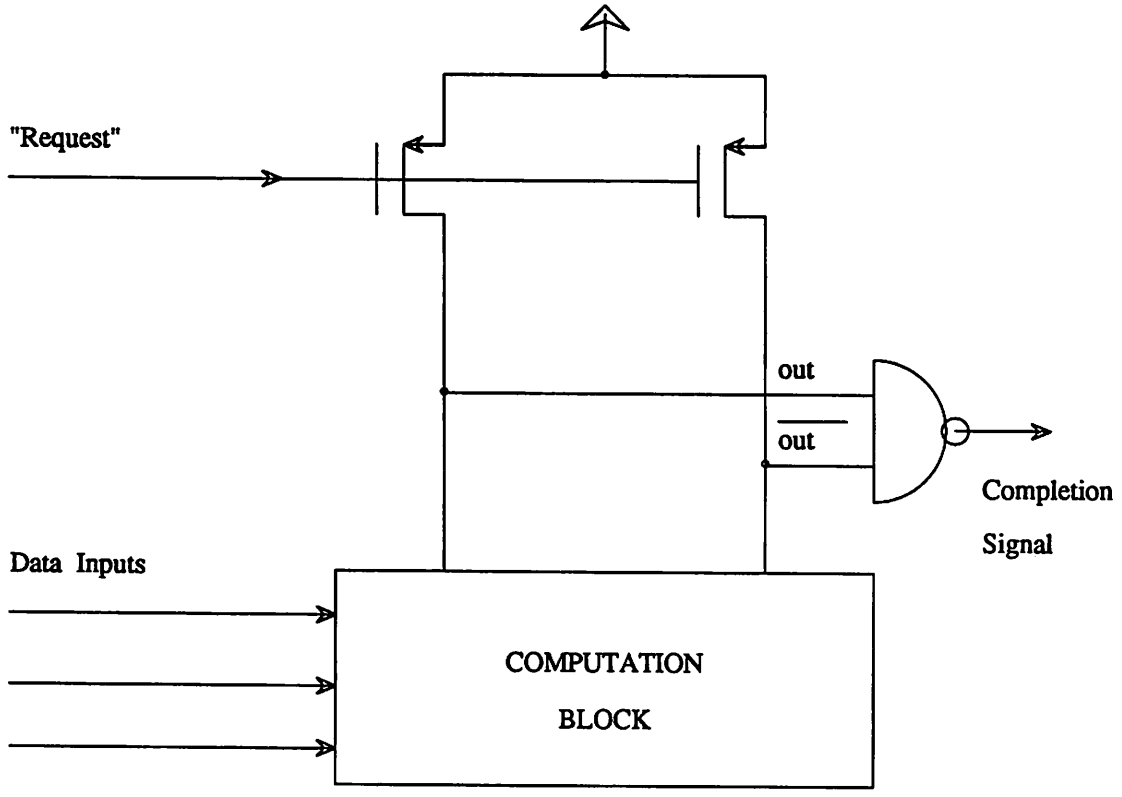


Figure 5.2 - "Completion signal" generation

When the "Request" line is low, both complementary output data lines (out and $\overline{out}$) are precharged to high thus causing the completion signal to be low. When the "Request" line goes high, the computing unit starts to evaluate the data on the input

lines. The evaluation is completed when one of the complementary output lines switches to low and the other remains high thus causing the completion signal to go high. Switching of the completion signal to high indicates to the next computational block that output data is valid, stable and ready to be transferred.

The data transfer interconnection between the computational blocks utilizes the four-phase hand-shake protocol described in [2, 26]. In this four-phase hand-shake protocol, the completion signal acts as an input request signal R_{in} from computation block A to the interconnection block depicted in figure 5.3.

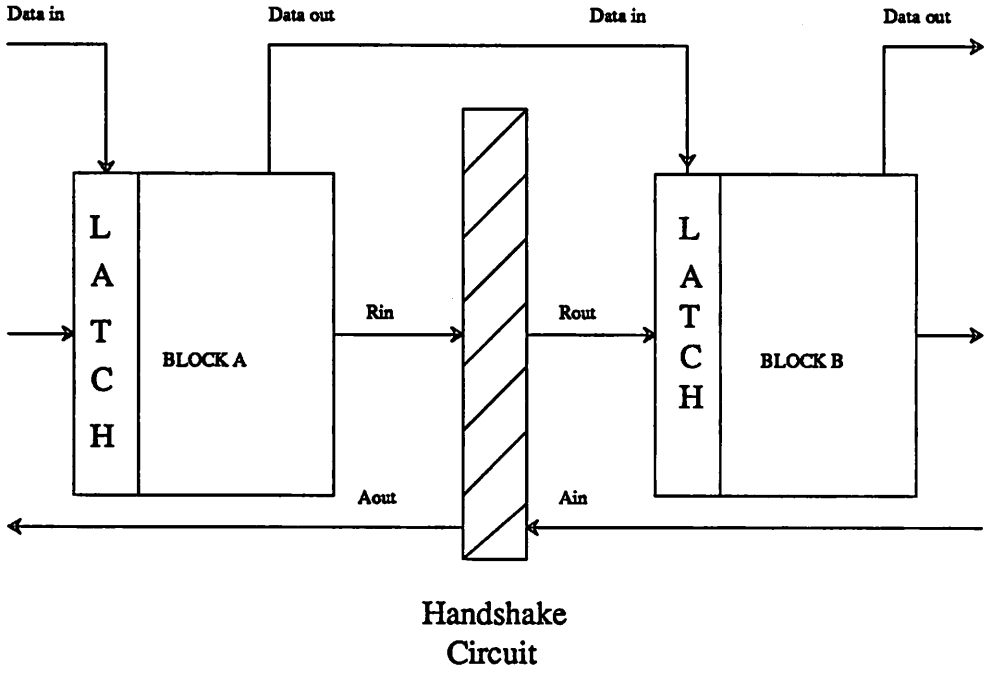


Figure 5.3 - Four phase hand-shake circuit

The interconnection block checks the feedback acknowledge signal A_{in} which indicates whether the computation block B has completed its task and is ready for the next sample of data. If signals R_{in} and A_{in} satisfy the conditions for data transfer between the computation blocks, the interconnection block sets the output request R_{out} which controls the transfer and the latching of the data into the input buffer of computation block B. Signal A_{out} of the interconnection block notifies computation block A if

block B is ready for a new sample of data and whether the data transfer was completed. The completion signal generation and the four-phase hand-shake protocol assure the proper operation of the asynchronous processor implementation.

In an asynchronous pipelined architecture each stage of the pipeline is a computing block. Data transfer between computing blocks (pipeline stages) is initiated by the preceding stage and "ripples" forward in the direction of the data flow from the first stage to the last while the beginning of task execution within the stages starts from the last stage and "ripples" backward against the direction of the flow to the first stage. Since the handshaking protocol is fast compared to the execution time of the pipeline stages the stages operate concurrently as in the synchronous architecture.

As mentioned previously, since the execution time of the computation blocks is data-dependent and instruction-dependent, the "average" throughput of the asynchronous processor will increase, but there are still more underlying questions to be asked and explored:

- In a pipeline architecture implemented asynchronously, will there also be a throughput increase in the "worst-case" performance for real-time digital signal processing applications ?
- How do we design an asynchronous processor and what are the additional delays and circuitry overheads ?
- What features and properties should be incorporated to make the asynchronous implementation more effective ?
- What are the characteristics, properties and limitations of other asynchronous processor architectures ?
- If there are no clock restrictions, is it possible to implement a synchronous processor with higher "average" throughput by exploiting data and instruction dependencies ?

The answers to these questions will be given in the following chapters.

5.3. Data path cycle time comparison

5.3.1. Introduction

As previously mentioned , due to clock skew the achievable data processing throughput has not increased at the same rate as the logic speeds of the switching circuits. Designing an asynchronous processor which does not require any clocks will increase the data processing throughput. But, it is still necessary to find out the timing conditions under which the asynchronous processor implementation will yield a higher throughput than the synchronous implementation. This can be studied through a cycle time analysis of the same processor architecture when it is implemented either by asynchronous circuits or by synchronous circuits. Serial and sequential nature of the data transfer between computation blocks in a processor with an asynchronous architecture and the assumption that data transfer through the interconnection blocks is much faster than the execution time of a task in the computation blocks suggests that we perform the timing analysis on a pipelined processor architecture in which all the stages operate concurrently.

5.3.2. Data path timing models

The data path of the pipeline architecture for comparing the throughput of the asynchronous and the synchronous implementation is depicted in figure 5.4.

Data dependency and branch conflicts are "bad" properties of a pipeline architecture that reduce the throughput but do not depend on whether the implementation is synchronous or asynchronous. Thus ignoring these conflicts does not affect the throughput analysis.

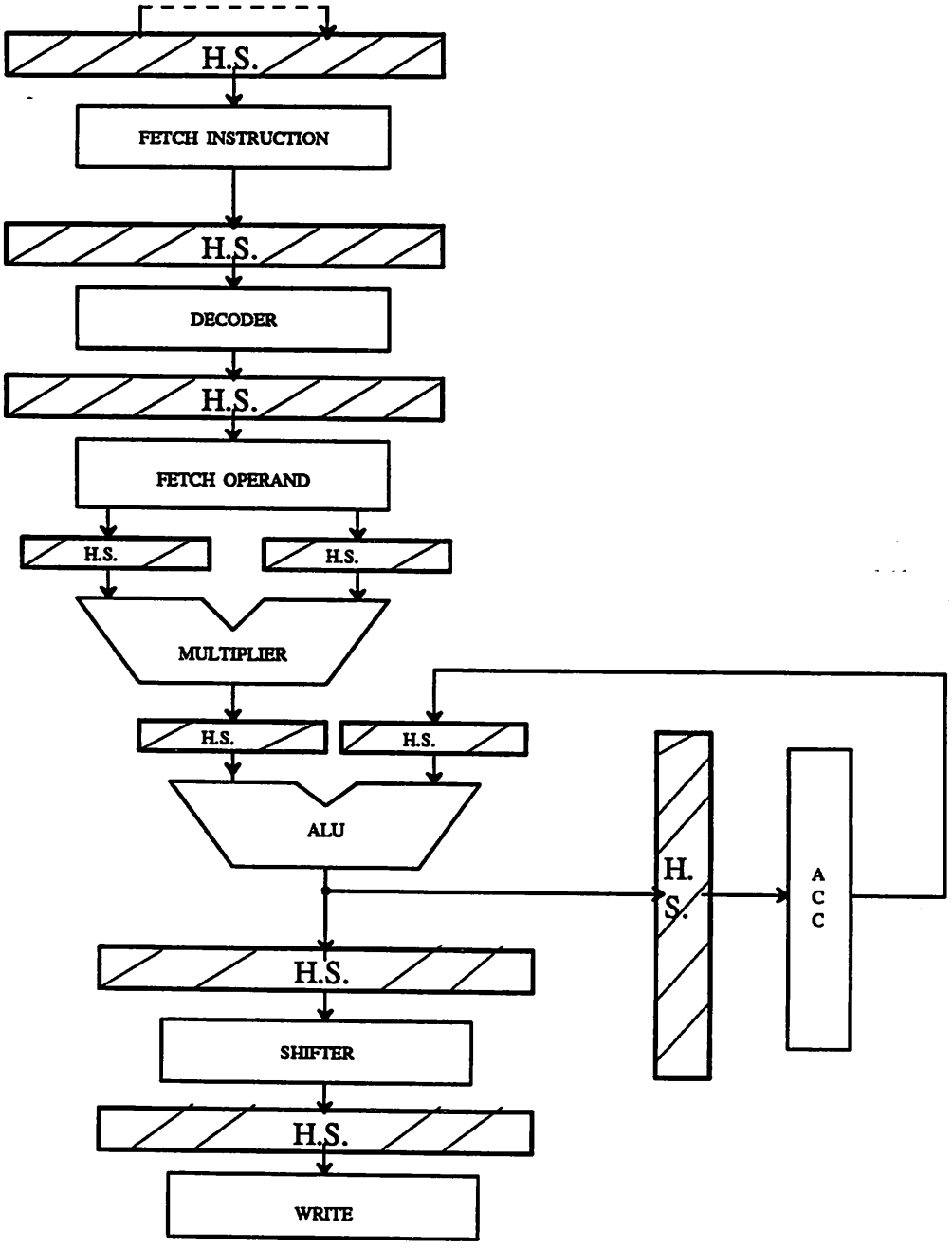


Figure 5.4 - Asynchronous architecture - Data path

The propagation delays of the data between the various pipeline stages can be modeled similarly to the models of data propagation delay between the nodes of the synchronous and asynchronous multiprocessor network systems described in[1, 5, 27]. These multiprocessor network models assume that in the synchronous case, due to

different path length of clock distribution, line capacitance and fabrication process, the clock skew delay should be added to the propagation delay of the data between the nodes of the network. In the asynchronous case, the delay of the hand-shake should be added to the propagation delay of the data between the nodes of the network.

Using similar arguments, additive timing models based upon the data propagation delays will be used for determining the cycle time in a pipeline architecture . In the synchronous case, the clock skew delay will be added to the propagation delay of the data through a pipeline stage (execution time of a task in the stage). In the asynchronous case the hand-shake delay will be added to the propagation delay of the data through a pipeline stage (execution time of a task in the stage). For concurrent operation of the pipeline stages, the synchronous and the asynchronous implementations have data buffers between the stages thus imposing an additional latch delay.

5.3.3. Synchronous & Asynchronous cycle time models

The general assumption is that the execution time of each stage of the pipeline architecture is data dependent. Therefore, the execution time of each pipeline stage also varies (not worst case all the time) and on average the asynchronous implementation should have a higher throughput for the same application.

Asynchronous circuit implementation is different from the synchronous one in that it involves hardware overhead and processing delays for completion signal generation and interconnection circuitry within the pipeline stage (computing unit). Thus, the execution time of an asynchronous pipeline stage could possibly be larger than the synchronous one. For simplifying the timing analysis, the delay of the completion signal generation is neglected, thus assuming that execution time of the different pipeline stages is identical in both asynchronous and synchronous implementations. Therefore, the results of this analysis would be the theoretical upper bounds for achieving a throughput improvement.

Depending on the data, the execution time of each pipeline stage varies between a minimum execution time denoted min.t_{sd} to a maximum execution time denoted max.t_{sd}. For the cycle time analysis of the synchronous and the asynchronous implementations we can define the following times:

- t=max{max.t_{sd}} - denotes the upper limit of the maximum execution time of all the stages.

  where max.t_{sd} is the worst case data dependent execution time of a stage.

- t_1=max{min.t_{sd}} - denotes the upper limit of the minimum execution time of all the stages.

- k=$\frac{t_1}{t}$ - denotes the ratio between the variation limits of all stage's execution time.

- t_l - denotes the time (delay) to latch the data in the buffers between the pipeline stages (input buffers of the following pipeline stage) - propagation delay of the data from the buffer's input to its output.

- t_{cs} - denotes the maximum clock skew delay between the stages of the pipeline architecture in the synchronous implementation.

- t_{hs} - denotes the maximum handshaking delay between the stages of the pipeline architecture in the asynchronous implementation.

The worst case cycle time T for the two implementation are as follows:

- In the synchronous architecture, there is a global clock and no hand-shaking is necessary. Figure 5.5 depicts the time delays for evaluating the worst-case cycle time.

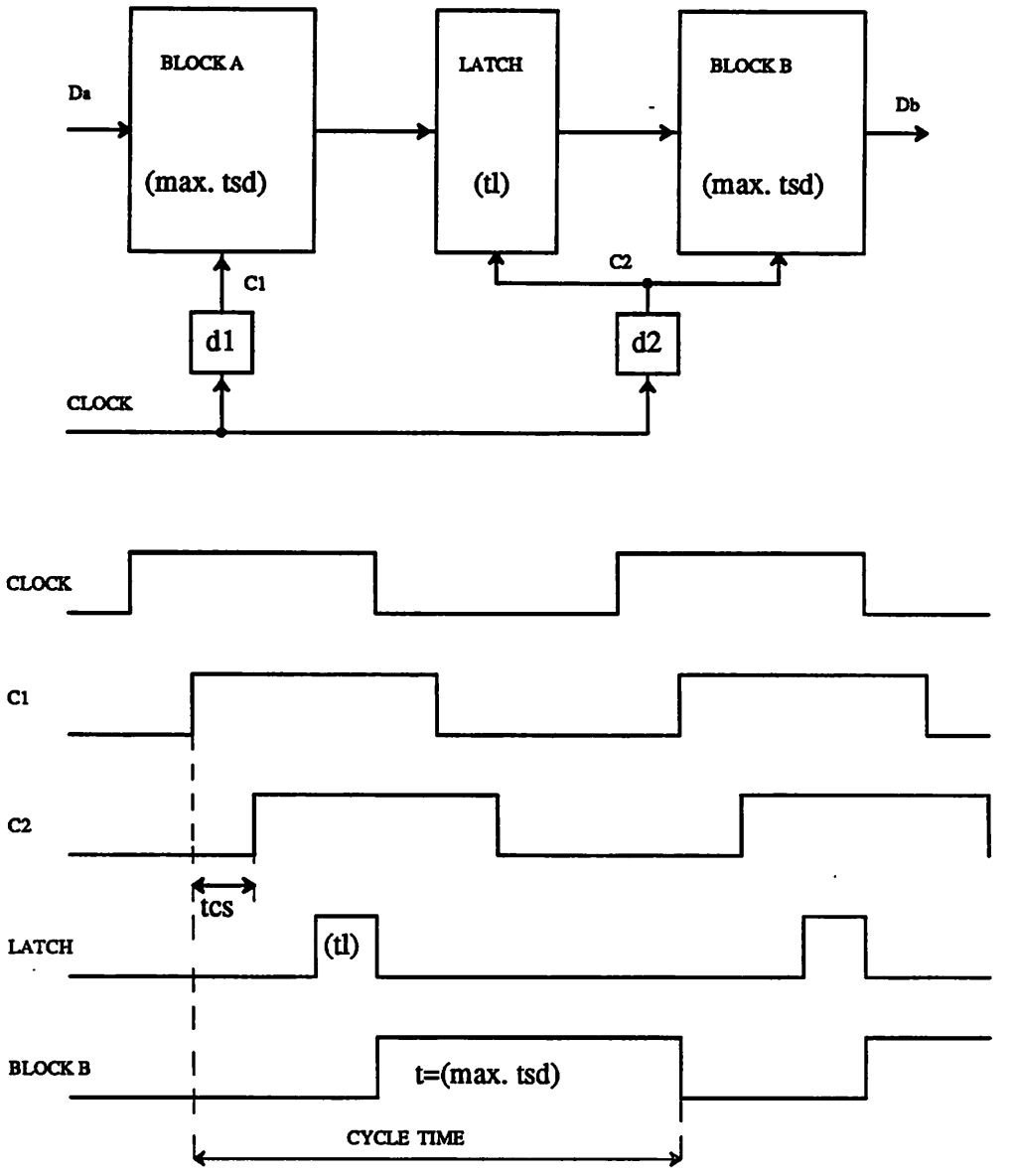


Figure 5.5 - Synchronous architecture - clock cycle

The clock skew is the time difference between the triggering of the clock at block B and at block A (d2-d1). Therefore, the worst case cycle time will be the sum of: the longest propagation delay of the data through any of the pipeline stages (e.g., block B in figure 5.5), the longest time delay due to the clock skewing (d2-d1), and the time to latch the data in the buffers of the pipeline.

$$T_{sy}=t+t_{cs}+t_l \qquad (5.1)$$

- In the asynchronous architecture there is no global clock, and therefore there is no clock skew delay, but it is necessary to add the delay due to handshaking circuits and procedure. Figure 5.6 depicts the timing delays for evaluating the worst case cycle time.

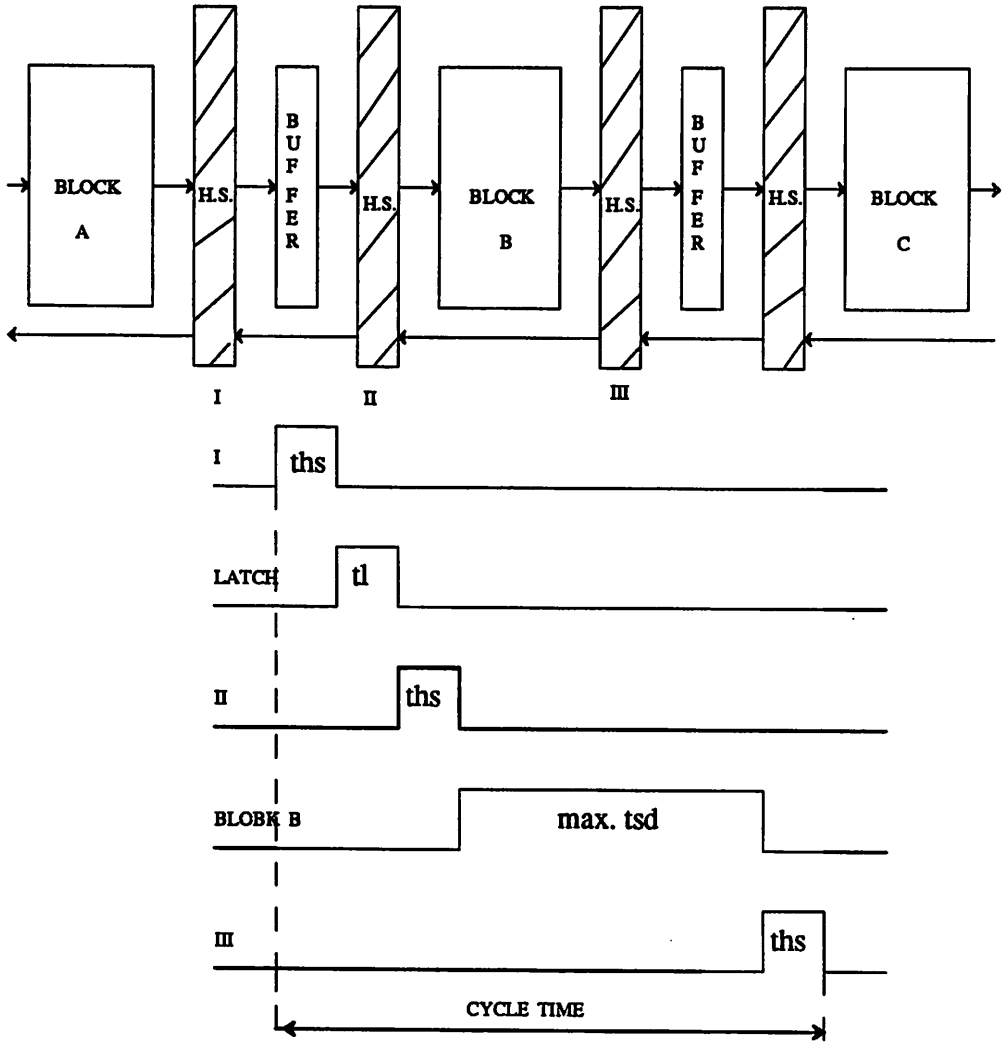


Figure 5.6 - Asynchronous processor - cycle time

It is important to note that three handshake delays are involved in the cycle time calculation. A computation block can receive a new input sample only after transferring its output data to the next computation block. This transfer involves one handshake delay. Latching data from the preceding computation block

involves one handshake delay and the propagation delay in the latch(t_l). To make sure that input data and the corresponding control signal have been latched and are valid, the latch must generate a completion signal, which by handshaking signals the computation block to start its computation. Thus, the worst case will be the sum of: the longest propagation delay of the data through any of the pipeline stages (e.g., block B in figure 5.6), the time to latch the input data from the preceding stage into the input buffer, and three times the time that it takes to perform the hand-shake.

$$T_{asy}=t+3t_{hs}+t_l \tag{5.2}$$

- In the asynchronous architecture it is also possible to evaluate an average cycle time. This average cycle time is due to the data dependent variations in the execution time of the pipeline stages. Assuming that the execution time is symmetrically distributed, then instead of using the largest execution time among all the blocks t=max{max.t_{sd}}, an average of the maximum largest execution time and the maximum shortest execution time should be used: $\frac{t+t_1}{2}$. When the handshake and the latch delays have negligible variations, but the execution time does not, the average cycle time will be the sum :

$$T_{avg.asy}=\frac{t+t_1}{2}+3t_{hs}+t_l=\frac{1+k}{2}t+3t_{hs}+t_l \tag{5.3}$$

Remark

Digital signal processing algorithms are based upon a fixed input data sample rate. When the sample rate is fixed and proportional to the inverse average cycle time ($\frac{1}{T_{avg.asy}}$), the architecture should incorporate input and output queue buffers to handle variations of the execution time (cycle time). Input buffers are necessary for storing input data samples when the execution time is larger than the average cycle time. Output buffers are necessary for storing output data samples when the execution time is

smaller than the average cycle time.

It is impossible to determine the exact length of the queues and once it has been determined the architecture will operate in the average cycle time $T_{avg.asy}$ only for a specific set of applications and input sample rates.

5.3.4. Worst case cycle time analysis

To decide when to use an asynchronous implementation it is necessary to evaluate the conditions under which this implementation yields a higher throughput compared to the synchronous one. When $T_{asy} < T_{sy}$, the asynchronous implementation has a shorter cycle time which corresponds to a higher throughput than the synchronous implementation. Thus the necessary condition that the asynchronous implementation will have a higher throughput is:

$$t_{hs} < \frac{t_{cs}}{3} \tag{5.4}$$

Using equations (5.1) & (5.2) and defining the cycle time improvement factor to be $q=\frac{P[\%]}{100}$ [q<1], where P is the percentage cycle time improvement, the ratio between the cycle times is:

$$\frac{T_{asy}}{T_{sy}} = \frac{t+3t_{hs}+t_l}{t+t_{cs}+t_l} = 1-q \tag{5.5}$$

Since the throughput is the inverse of the cycle time, the throughput improvement factor can be derived from equation 5.5 as follows:

$$\frac{(Throughput)_{asy}}{(Throughput)_{sy}} = \frac{T_{sy}}{T_{asy}} = \frac{1}{1-q} = 1+q+q^2+q^3+\ldots = \sum_{i=0}^{\infty} q^i \tag{5.6a}$$

$$(Throughput)_{asy} = (Throughput)_{sy}\,(1+q+q^2+q^3+\ldots) \tag{5.6b}$$

Equation 5.6b shows that for a given cycle time, the improvement factor q of the asynchronous cycle time yields an asynchronous throughput improvement which is greater

than q.

From equation (5.5) it is possible to derive the conditions for the different delays which will yield a throughput improvement factor higher than q for the asynchronous implementation. The handshake delay t_{hs} as a function of the improvement factor and the synchronous delays will be:

$$t_{hs}=\frac{(1-q)t_{cs}-q(t_l+t)}{3} \tag{5.7}$$

This equation is the exact expression of t_{hs}. Equation 5.4 that required $t_{hs} < \frac{t_{cs}}{3}$ is the special case derived from 5.7 when the two implementations have the same cycle time. Feasible realizations requires $t_{hs} > 0$, therefore the expression in the numerator of equation 5.7 must be positive and that yields the lower bound of the clock skew delay t_{cs} :

$$t_{cs} > \frac{q}{1-q}(t_l+t) \tag{5.8}$$

If the clock skew delay is less than (5.8), the asynchronous implementation will not yield the required throughput improvement q - i.e., there is no minimal handshake delay which will yield the required q.

Equation 5.5 also yields the bound on the improvement factor for the worst case propagation delays of the synchronous and asynchronous implementations:

$$q=\frac{t_{cs}-3t_{hs}}{t+t_{cs}+t_l} \tag{5.9}$$

The conclusion from equation 5.9 is that for larger clock skew delay relative to the other delays, the asynchronous implementation has a higher throughput (by at least a factor of q) compared to the synchronous implementation.

Results

- An asynchronous processor implementation which yields a higher throughput by the improvement factor q is feasible ($t_{hs} > 0$) only if the clock skew delay t_{cs}

satisfies equation 5.8.

- The asynchronous implementation yields a higher throughput only if $t_{hs} < \frac{t_{cs}}{3}$.
- For handshake and clock skew delays that satisfy equation (4), the deeper the pipeline architecture (i.e., more pipeline stages which corresponds to shorter execution time and t smaller) , the larger the relative importance of clock skew delay t_{cs} on the cycle time, and the more likely the asynchronous implementation will have a higher improvement factor q.
- For any given technology (gate propagation delay) and pipeline architecture when $t_{cs} >> 3t_{hs}$ and t_l is negligible, there exists an approximate upper bound on the throughput improvement factor:

$$q \approx \frac{t_{cs}}{t + t_{cs}} \tag{5.10}$$

The upper bound of the throughput improvement depicted in the following table and figure 5.7 show that as the clock skew increases relative to the stage execution time and the handshake delay, the throughput of the asynchronous implementation will increase.

q[%]	t_{cs}
10%	0.11t
20%	0.25t
30%	0.43t
40%	0.66t
50%	1.00t

Table 5.1 - Clock cycle improvement factor vs. clock skew delay.

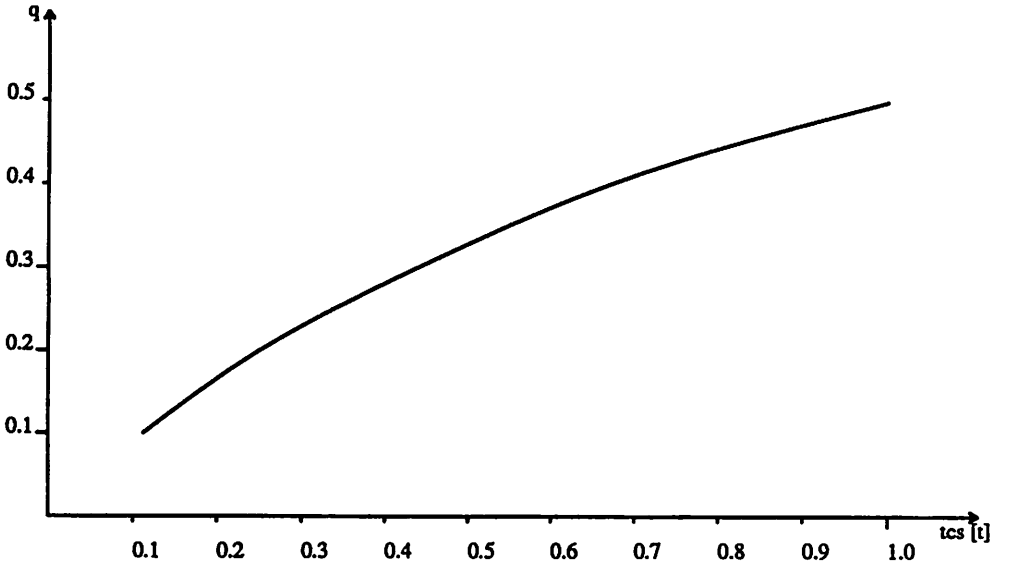


Figure 5.7 - Clock cycle improvement factor

5.3.5. Average cycle time analysis

5.3.5.1. Average cycle time analysis

The bounds on the clock-skew and hand-shake delays of the average asynchronous cycle time when the architecture has I/O queues and the propagation delay variations of the pipeline stages (k) are given, can be derived from equation 5.3:

$$T_{avg.asy}=(k+1)\frac{t}{2}+3t_{hs}+t_l$$

From the above cycle time equation and the cycle time equation of the synchronous case T_{sy}=t+t_{cs}+t_l , we can derive the ratio:

$$\frac{T_{avg.asy}}{T_{sy}}=\frac{(k+1)\frac{t}{2}+3t_{hs}+t_l}{t+t_{cs}+t_l}=1-q \qquad (5.11)$$

which yields:

$$t_{hs}=\frac{1}{3}[(\frac{1}{2}-q-\frac{k}{2})t+(1-q)t_{cs}-qt_l] \qquad (5.12)$$

As before, a feasible asynchronous implementation which yields the throughput improvement factor q, requires t_{hs}>0. It follows that the expression in the brackets

should be positive which yields the lower bound of the clock skew delay for the average case analysis:

$$t_{cs} \geq \frac{qt_l - (\frac{1}{2} - q - \frac{k}{2})t}{1-q} \tag{5.13}$$

As before, from equation (5.11) one can derive the upper bound of the improvement factor as a function of the propagation delays:

$$q \leq \frac{(\frac{1}{2} - \frac{k}{2})t + t_{cs} - 3t_{hs}}{t + t_{cs} + t_l} \tag{5.14}$$

This equation (5.14) shows that even for small values of t_{cs}, large variations in the execution time of the pipeline stages, which correspond to k<<1, yields a higher improvement factor of the asynchronous average throughput compared to the synchronous throughput.

5.3.5.2. Handshake delay variations

Equation 5.12 shows that the handshaking delay is a function of three variables: the clock skew delay, the cycle time improvement factor and the execution time variations. Rewriting equation 5.12 of t_{hs} as a function of the execution time variations k we get:

$$t_{hs} = \frac{1}{2}[(\frac{1}{2} - q)t + (1-q)t_{cs} - qt_l] - \frac{t}{6}k \tag{5.15}$$

Figure 5.8 depicts the handshaking delay as a function of the execution time variations, when the clock skew delay and the improvement factors are given.

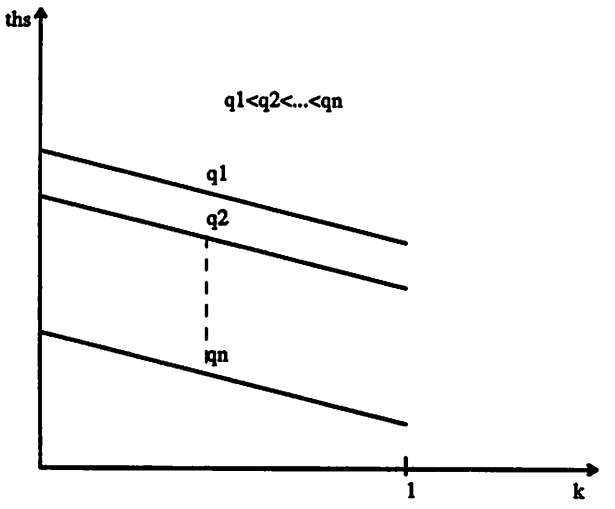


Figure 5.8 - Handshake delays vs. execution time variations

Thus, t_{hs} is linearly dependent on k. Larger variations in the execution time of the pipeline stages (corresponds to smaller k) imply that larger handshake delays can achieve the same cycle time improvement.

For an architecture with a given variations of the stages execution time, smaller handshaking delays mean a greater cycle time improvement.

If we rewrite t_{hs} as a function of t_{cs} we get:

$$t_{hs}=\frac{1}{3}[(\frac{1}{2}-q-\frac{k}{2})t-qt_l]+\frac{(1-q)}{3}t_{cs} \qquad (5.16)$$

Figure 5.9 depicts the handshaking delays as a function of of the clock skew when the execution time variations and the improvement factor are given.

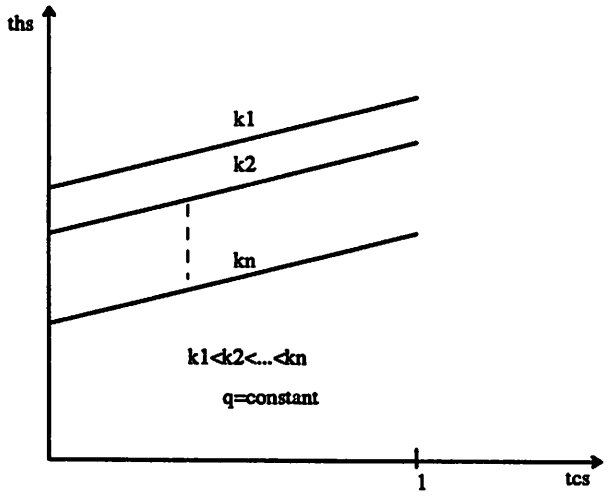


Figure 5.9 - Handshake delay vs. clock skew delay

t_{hs} is linearly dependent on t_{cs}. Larger clock skew delays allow larger hand-shaking delays in order to achieve the same cycle time improvement. Also, as before, larger variations in the stages execution time allow larger handshaking delays in order to obtain the same cycle time improvement (q).

Rewriting the equation of t_{hs} as a function of q the cycle time improvement factor we get:

$$t_{hs}=\frac{1}{3}[(\frac{1}{2}-\frac{k}{2})t+t_{cs}]-\frac{t+t_{cs+t_l}}{3}q \qquad (5.17)$$

Figure 5.10 depicts the handshaking delay as a function of the cycle time improvement factor when the clock skew and the execution time variations are given.

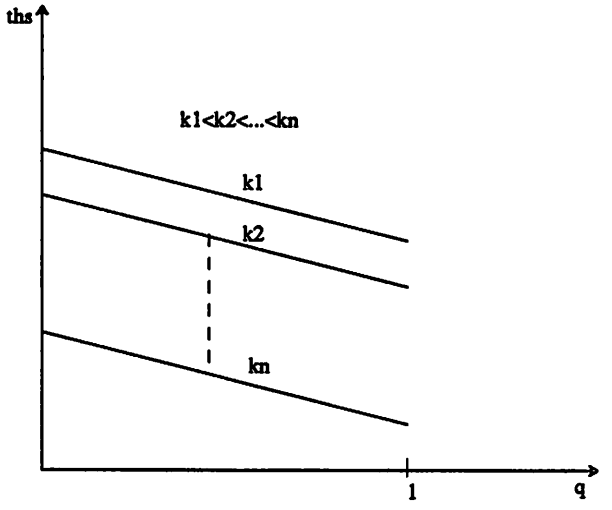


Figure 5.10 - Handshake delay vs. cycle time improvement factor

t_{hs} is linearly dependent on q. For larger improvement of the cycle time , the hand-shaking propagation delay should be smaller.

5.3.5.3. Average cycle time analysis results

Analyzing the bound of the improvement factor q in equation (5.14) yields the following conclusions:

- Larger the variations in the execution time of the pipeline stages (smaller k) imply greater throughput improvement in the average asynchronous case.
- $t >> t_l$ and $t >> t_{cs}$, which implies that $t >> t_{hs}$ correspond to an architecture with a small number of pipeline stages. In such an architecture the clock skew, handshake and latch delays are negligible, thus yielding an upper bound of the improvement factor to be $q < \frac{1-k}{2}$. This result shows that when the additive delays are negligible compared to the execution time, the maximum achievable average throughput improvement could only be 50%.
- If t_{cs} is not negligible compared to the execution time, the throughput improvement factor (q) of the average asynchronous case could be above 50%.

- k=1 corresponds to the timing of the worst-case asynchronous case. In this case, if the clock skew delay is very small ($t_{cs} \approx 0$) the asynchronous implementation will not be advantageous.
- Assuming as before that the hand-shake and latch delays are negligible compared to the clock skew delay, i.e., $t_{cs} >> 3t_{hs}$ and $t_{cs} >> t_l$, but the clock skew delay t_{cs} is not negligible compared to the stages execution time, the approximate throughput improvement upper bound is:

$$q \approx \frac{\frac{1}{2}(1-k)t + t_{cs}}{t + t_{cs}} \tag{5.18}$$

Under these conditions, the throughput improvement factor q is depicted in the following table and in figure 5.11:

t_{cs}	q[%]				
	k=0	k=$\frac{1}{4}$	k=$\frac{1}{2}$	k=$\frac{3}{4}$	k=1
	Max variations, 100% variations in t	75% variations in t	50% variations in t	25% variations in t	Worst case, no variations in t
0.00t	50%	37.5%	25%	12.5%	0%
0.11t	55%	43.7%	32.5%	21%	10%
0.25t	60%	50%	40%	30%	20%
0.43t	65%	56.3%	47.5%	38.8%	30%
0.66t	70%	62.3%	55%	47.3%	40%
1.00t	75%	68.7%	62.5%	56.2%	50%

Table 5.2 - Throughput improvement factor

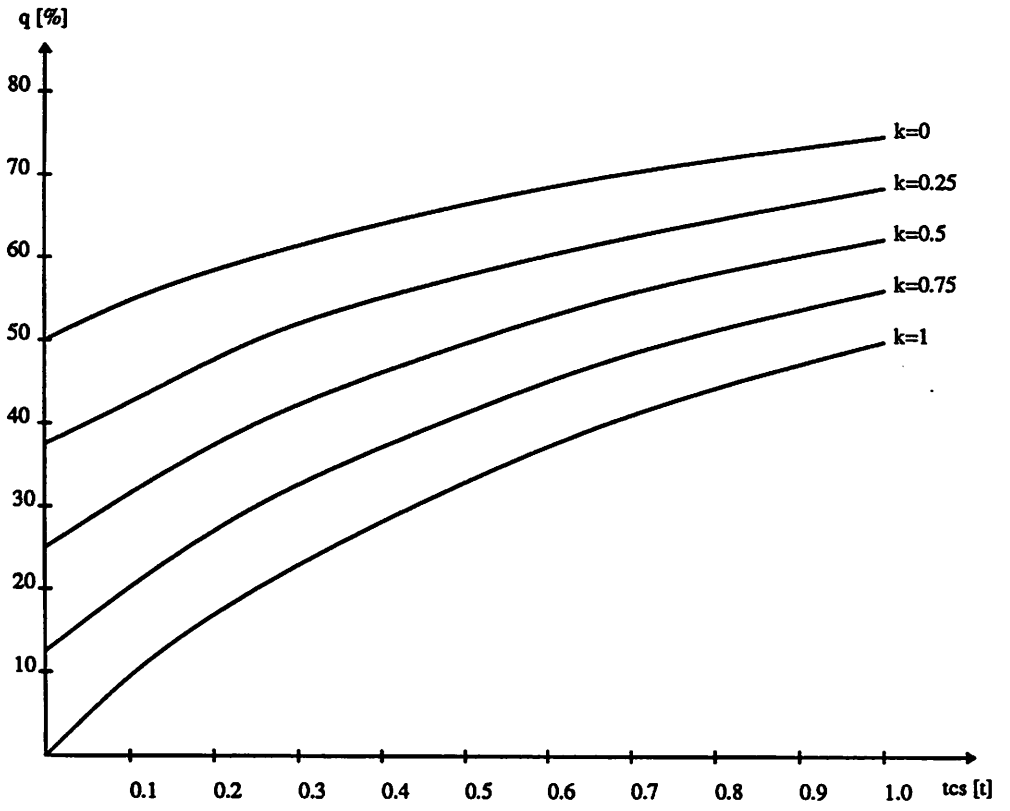


Figure 5.11 - Throughput improvement factor vs. clock skew

- Larger variations in the stages execution time (k<<1) increase the average throughput improvement factor.
- For the same variations in the stages execution time, when t_{cs} is not negligible (equation 5.17), the average throughput improvement factor will be larger.
- The larger the clock skew delay compared to the stages execution time (corresponds to "deeper" pipeline) the larger will be the throughput improvement factor of the average asynchronous case compared to the synchronous one.

5.4. Conclusions

- As before, for an architecture with a small number of pipeline stages (t_{cs} is negligible, corresponds to the first row in table 5.2) the average improvement throughput factor will be approximately $q \approx \frac{1}{2}(1-k)$.

- To achieve a higher throughput factor q in the asynchronous implementation, the clock skew delay (t_{cs}) and the hand-shake delay (t_{hs}) must fulfill the conditions of equations 5.12 and 5.13.
- When the handshaking delay (t_{hs}=0 and the latch delay (t_l) are negligible compared to the execution time (t) but the clock skew delay (t_{cs}) is not negligible, a greater clock skew delay implies a greater improvement in throughput in the asynchronous implementation.
- In a deep pipeline architecture (an architecture with a large number of pipeline stages) the clock skew delay t_{cs} has a greater effect on the throughput improvement factor (q) of the asynchronous implementation.

References

1. S.Y. Kung and R.J. Gal-Ezer, "Synchronous vs. Asynchronous computation in very large scale integration (VLSI) array processors," *SPIE vol. 341 Real Time Signal Processing*, pp. 53-65, 1982.
2. C. Mead and L. Conway, in *Chap. 7, Introduction to VLSI systems*, Addison-Wesley publishing company, 1980.
3. Daniel M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems,* pp. 1-123, Ph.D Dissertation Stanford University, October 1984.
4. S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V. BhaskarRao, "Wavefront array processor:Language, Architecture and Applicatons," *IEEE trans. on computers*, vol. C-31, no. 11, November 1982.
5. Donald F. Wann and Mark A. Franklin, "Asynchronous and Clocked Control Structure for VLSI Based Interconnection Networks," *IEEE Trans. on Computers*, vol. C-32, No. 3, pp. 284-293, March, 1983.

6. T.H.Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Automatic synthesis asynchronous circuits from high level specifications," *IEEE ICCAD 87 Digest of Technical Papers*, November 1987.

7. T.A. Chu, C.K.C. Leung, and T.S. Wanuya, "A design methodology for concurrent VLSI systems," *ICPP 85*, pp. 407-410, 1985.

8. T.A. Chu, "On the models for designing VLSI asynchronous digital systems," *North-Holland INTEGRATION*, no. 4, pp. 99-113, 1986.

9. J.R. jump , "Asynchronous control arrays," *IEEE trans. on computers*, vol. C-23, no. 10, pp. 1020-1029, October 1974.

10. L.A. Hollar, "Direct implementation of asynchronous control units," *IEEE trans. on computers*, vol. C-31, pp. 1133-1141, December 1982.

11. D.A. Huffman, C.A. Rey, and J. Vaucher, "Self synchronized asynchronous sequential machines," *IEEE trans. on computers*, pp. 1306-1311, December 1974.

12. G. Mago, "Realization methods for asynchronous sequential circuits," *IEEE trans. on computers*, vol. C-20, no. 3, pp. 290-297, March 1971.

13. D.B. Armstrong , A.D. Friedman, and P.R. Manon, "Design of asynchronous circuits assuming unbounded gate delay," *IEEE trans. on computers*, vol. C-18, December 1969.

14. R.B. Keller, "Towards a theory of universal speed independent modules," *IEEE trans. on computers*, vol. C-23, no. 1, pp. 21-33, January 1974.

15. D. Hammel , "Ideas on asynchronous feedback networks," *Proc. 5th ann. Symp. on Switching Circuit Theory and Logic Design*, pp. 4-11, November 1964.

16. W. Plummer, "Asynchronous arbiters," *IEEE trans. on computers*, vol. C-21, no. 1, January 1972.

17. J.C. Calvo, J.I. Acha, and M. Valencia, "Asynchronous modular arbiter," *IEEE*

trans. om computers, vol. C-35, no. 1, January 1986.

18. J.C. Barros and B.W. Johnson, "Equivalence of the arbiter, the synchronizer, the latch, and the inertial delay," *IEEE trans. on computers* , vol. C-32, July 1983.

19. T.A. Chu, "Synthesis of self timed control circuits from graphs: an example," *Proceedings IEEE ICCD*, pp. 565-571, October 1986.

20. D.L. Dill and E.M. Clarke, "Automatic verification of asynchronous circuits using temporal logic," *Proceedings 1985 Chapel Hill Conference on Very Large Scale Integration*, pp. 127-143, 1985.

21. C.L. Seitz, "Self timed VLSI systems," *Proc. of the Cal Tech. Conference on VLSI*, January 1979.

22. S.H. Unger, in *Asynchronous sequential switching circuits*, Wiley-Interscience, New-York, 1969.

23. L.G. Heller and W.R. Griffin, "Cascode voltage switch logic: A differential CMOS logic family," *ISSCC Digest of Technical Papers*, February 1984.

24. G. Jacobs and R.W. Brodersen, "Circuit techniques for realization of self-timed DSPs," *To be submitted to IEEE trans. on JSSC.*

25. T.E. Williams, M. Horovitz, R.L. Alverson, and T.S. Yang, "A self-timed chip for division," *Advanced Research in VLSI, Proceedings of 1987 Stanford conference*, pp. 75-96, March 1987.

26. R.E. Miller, in *Switching theory*, John Wiley & Sons, Inc. New-York, 1965.

27. H.V. Jagadish, "Techniques for the design of parallel and pipelined VLSI systems for numerical computation," in *Ph.D. Dissertation*, December 1985.

CHAPTER 6

Asynchronous Processor Architectures

6.1. Asynchronous pipeline architecture

6.1.1. Introduction

This chapter describes the fundamental concepts and the principles involved in the development and the design of asynchronous pipeline processor architectures. An asynchronous processor design does not require global synchronization and thus the clock skewing problems such as appropriate clock distribution and timing verifications are eliminated. The design approach discussed here is based upon the use of asynchronous interconnection library blocks already developed by [1] which are basically self-timed handshake circuits, and upon additional interconnection blocks such as conditioned handshake circuits, which are required for proper design and implementation of an asynchronous architecture. The circuit design of the pipeline stages (computation blocks) is based upon DCVSL logic ("Differential Cascode Voltage Switch Logic")[2, 3] and is being done in parallel by another group at Berkeley. Since the processing time in the computation blocks is instruction and data-dependent, the design methodology should take it into consideration. The processor's architectural configuration and the way that the instructions are utilized and executed will determine the data dependency and the branch constraints imposed on the programmer. The design of the computation blocks and the interconnection blocks is decoupled and can be done independently. The inevitable hardware overhead and processing delays of the various interconnection circuits, within the computation blocks and between them, will

be explained later.

6.1.2. Design Approach & Principles

A general block diagram of the basic asynchronous pipeline architecture is depicted in figure 6.1. Each pipeline stage consist of a computation block (e.g. Multiplier, ALU) plus latching and handshake circuits. Since there is no global clock, data transfers between the computation blocks are controlled and executed by handshaking interconnection blocks.

The maximum number of instructions that can be executed concurrently in the pipeline architecture is limited by the number of the stages in the pipeline. In an asynchronous architecture, there is no global clock to synchronize the operations and therefore conflicts in sharing the same resources are possible. To avoid such problems, it is essential to define and determine the appropriate timing and operations required to:

- Execute non-data path instructions such as: NOP, STORE accumulator to memory, OUTPUT data to external I/O device, SET FLAGS, BRANCH, etc.
- Discard instructions from the pipe when a BRANCH instruction has to be executed.
- Minimize or avoid the addition of control handshaking delays to the existing data path delays.

The handshaking interconnection blocks, control and execute the data transfer between the computation blocks. As depicted in figure 5.3, request to transfer data between computation blocks (pipeline stages) is initiated by the "end of operation" signal of the preceding stage through R_{in} of the interconnection block. Data is transferred to a successive stage only if the successive stage has transferred its own output data to its next stage and is ready to receive and operate on a new sample of input data. Since the pipeline stages (computation blocks) are connected in serial, each pipeline stage can transfers its data to the next one only if all the successive stages have completed their

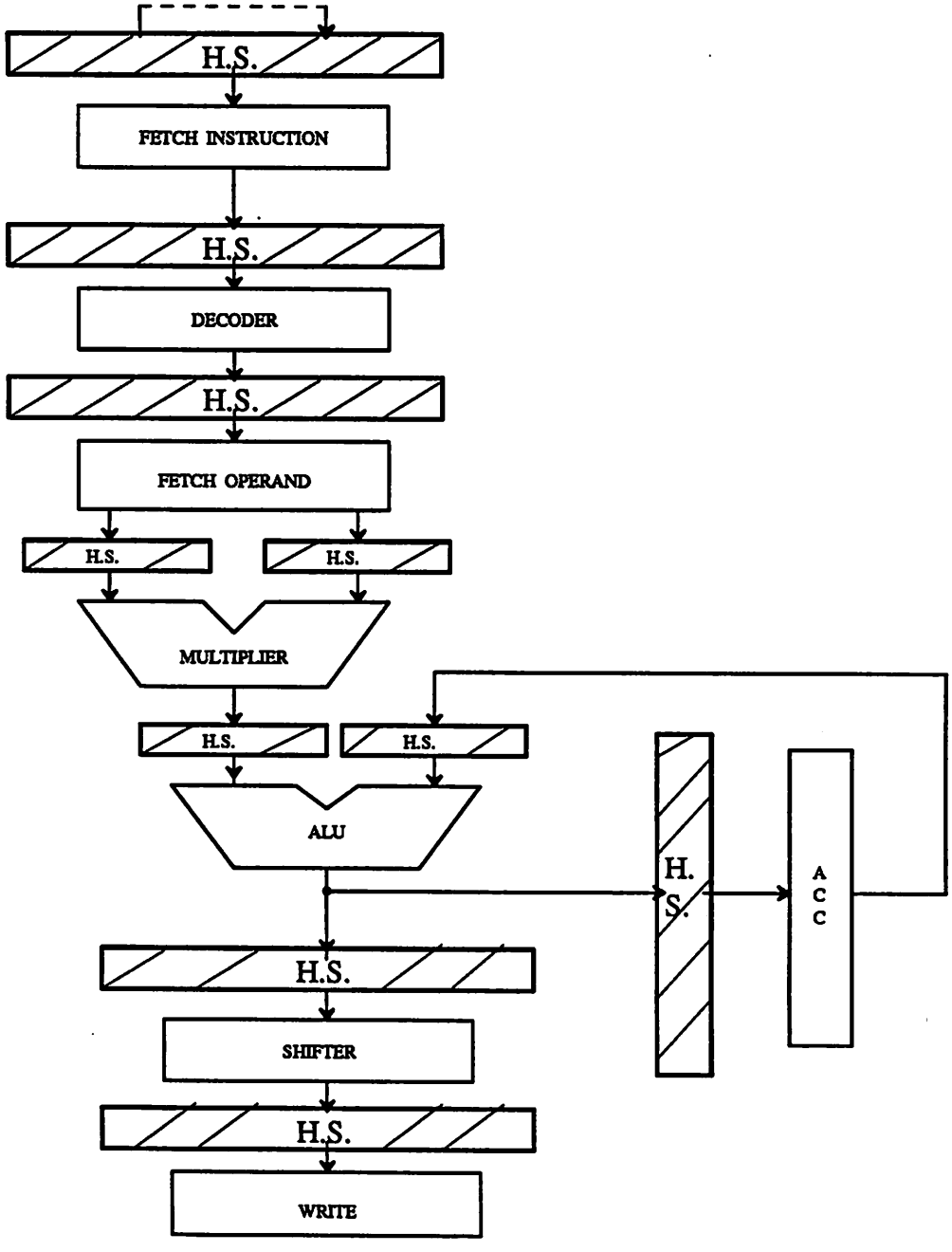


Figure 6.1 - Asynchronous pipeline architecture

data transfer. Therefore, data transfer and the beginning of task execution in the pipeline stages can be considered as if it starts in the last stage and "ripples" backward to the first one (opposite to the direction of the data flow). The following examples based on the pipeline architecture configuration depicted in figure 6.1 will illustrate this back-

ward "ripple" of start of execution in the pipeline stages.

Assume, as depicted in the figure 6.2 below, that all the pipeline stages except the last one ("write" stage) have finished their execution and are ready to transfer their output data.

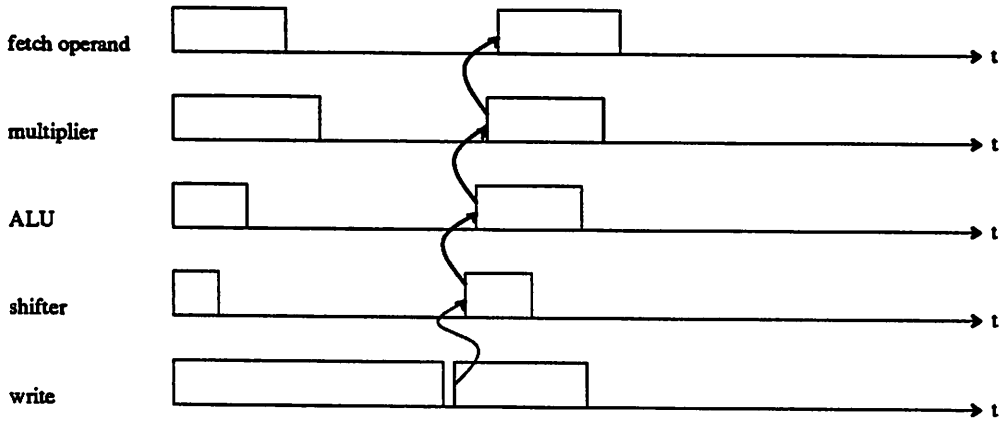


Figure 6.2 - Operation of an asynchronous processor (example 1)

Since the "write" stage is still busy with its task, the shifter stage can't transfer its output data to the "write" stage and therefore it cannot receive a new sample of input data from the ALU stage. Thus, the ALU cannot transfer its output data to the shifter, the multiplier cannot transfer its output data to the ALU and so on. Therefore, these stages cannot receive a new sample of input data to operate on and they are idle. Only when the "write" stage finishes its task, the shifter stage can transfer its output data to it and receive a new sample of input data from the ALU stage to operate on it. Once the shifter receives the output data of the ALU stage it frees the ALU to receive a new sample of input data from the multiplier stage to operate on it and so on.

Assume, as depicted in the figure 6.3 below, that all the pipeline stages except the "fetch operand" stages have finished their execution and are ready to transfer their output data to the next stage.

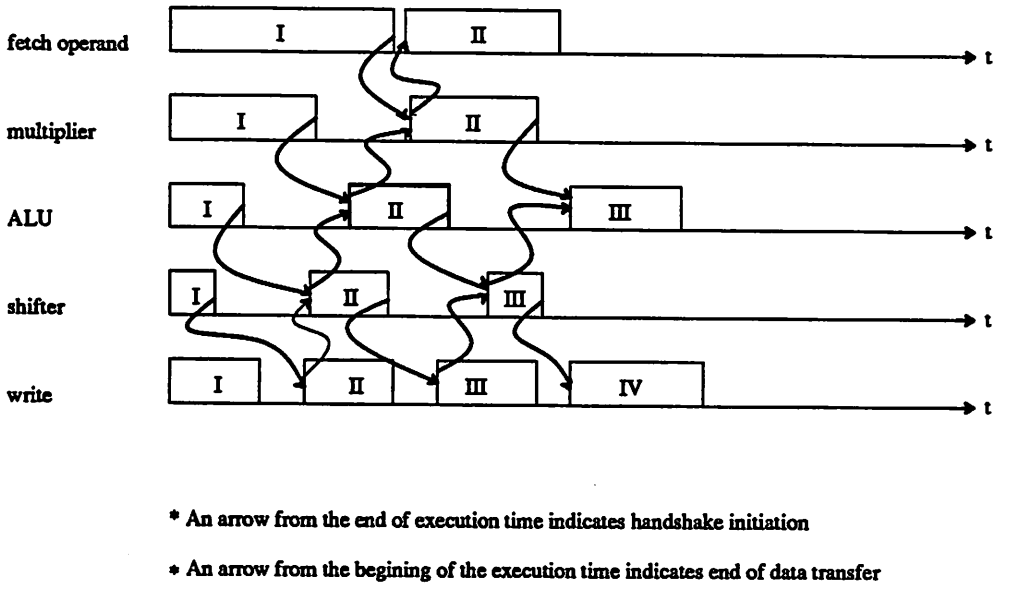


Figure 6.3 - Operation of an asynchronous processor (example 2)

In this case the ALU, shifter and "write" stages will transfer their output data to the next stage, receive a new sample of input data from the preceding stage and start to operate on it. The multiplier stage will transfer its output data to the ALU but it has to be idle until the "fetch operand" stage finishes its task and is ready to transfer its output data to the multiplier. When the "fetch operand" stage is ready to transfer its output data, it initiates the handshaking procedure with the multiplier stage. The multiplier stage latchs the new sample of input data, begins to operate on it and frees the "fetch operand" stage to execute its next task. If while the multiplier stage executes its task on the new sample of input data, all the successive stages have finished their execution the ALU, the shifter and the "write" stages will transfer their output data to their next stage but only the "write" and the shifter stages can start a new operation, the ALU has to wait for the multiplier to finish its current task.

These examples illustrate that each pipeline stage can start to execute a new task only if:

1) Its successive stages have transferred their output data and are ready to receive a new sample of input data to operate on it.

2) Its preceding stage finished its task and is ready to transfer its output data to it.

Although this handshaking protocol and the serial data transfer causes the pipeline stages to begin the execution of a new task sequentially one after the other, all the pipeline stages operate concurrently if the handshaking protocol and data transfer are fast compared to the execution time of the stages.

The handshake interconnection between the computation blocks (depicted in figure 6.1), is a four-phase full handshake (FHS). Internally within the computation blocks there are more non-pipeline handshake interconnections which guarantee that the operation of the block starts only after all the operands and the control data lines have been latched and are valid. A non-pipeline interconnection block between two computation blocks A and B, handles the data transfer between them and enables block A to accept a new sample of input data only after block B has completed its task on the output data of block A. Therefore, all internal data transfers between computation blocks incorporated in a pipeline stage are executed through non-pipeline interconnection blocks. As previously explained (see paragraph 5.3.3), it is important to implement a latch register which generates an "EOP" (End of operation) signal only after the data at its output becomes valid.

A general example of a computing block is depicted in figure 6.4.

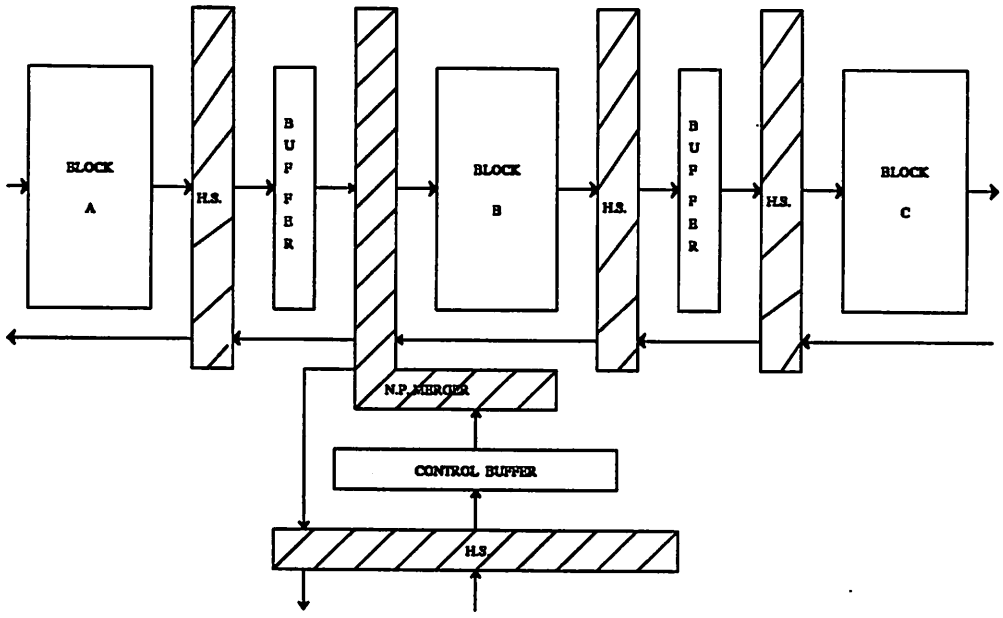


Figure 6.4 - Non-pipeline "merger"

The "merger" depicted in figure 6.4 is a non-pipeline handshake interconnection block that enables the computation in block B to begin only after the data from the preceding block (block A) and the control data from the controller are latched and valid. Computation block B will latch a new sample of input data only after block B has completed its task and has transferred its output data to the next stage. The non-pipeline "merger" block is not necessarily the only additional internal handshaking interconnection. Depending on the computation block's task and the architecture configuration, there might be other internal non-pipeline interconnection blocks which increase the area and the execution time overhead of the computation block. The pipeline architecture block diagram depicted in figure 6.1 can be partitioned into two parts: 1) Control path which includes the fetch instruction and the decoder stages of the pipeline that execute the fetch cycle of an instruction. 2) Data path which includes the fetch operands, multiplier, ALU, shifter and write data to memory stages that perform the execution cycle of an instruction.

This partition enables us to describe the pipeline architecture as two parallel paths

which are connected by control lines and status lines between the control path and the data path stages.

What follows is a more detailed description of the design of various pipeline stages of an asynchronous processor.

6.1.2.1. Fetch Instruction

The "fetch instruction" block depicted in figure 6.5 incorporates a program counter (PC), an address arithmetic unit (AAU), and a read only instruction memory (ROM). Only two of the handshake interconnections are pipeline interconnections between stages: one is the FHS (full handshake) which handles the transfer of instructions from the memory to the instruction register of the decoder, and the other is the multiplexer (MUX) which selects the memory address according to the control lines from the decoder. All other handshake interconnection blocks are internal and are therefore non-pipeline. These non-pipeline interconnections allow the "fetch operand" block (PC plus AAU) to be incremented and latched back to the PC while fetching an instruction from the memory. "MAR" is a memory address register that latches the address of the instruction to be fetched and allows the concurrency of fetching an instruction with incrementing the PC. When a *BRANCH* instruction is executed a control line will select the branching address from " BRANCH ADDRESS" register and the sequence continues as before.

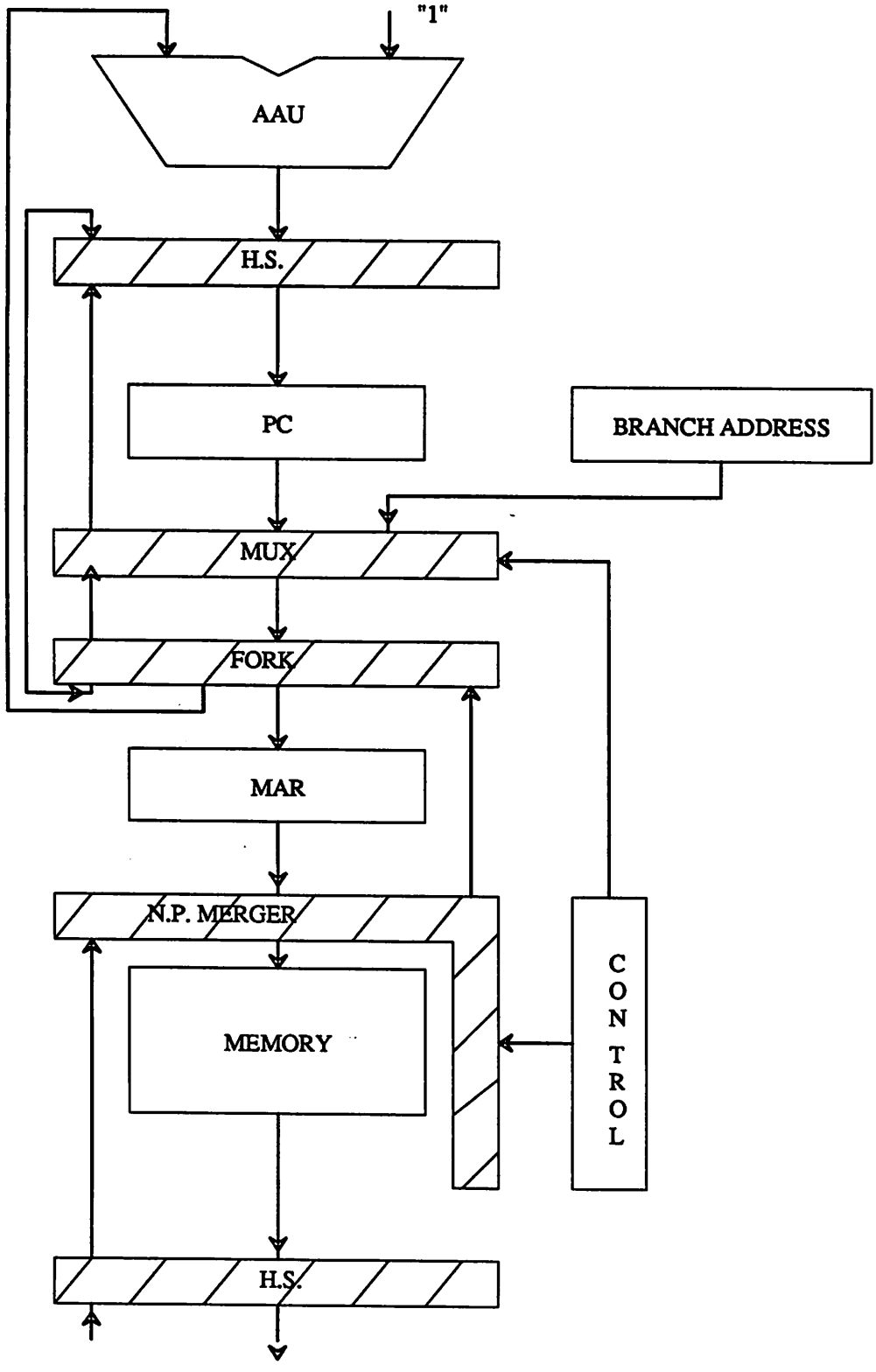


Figure 6.5 - Fetch instruction unit

6.1.2.2. Decoder (Control Unit)

Control problems

In a synchronous pipeline architecture, the control unit uses the global clock to synchronize the operations and to avoid conflicts in demands for the same resources. Unlike the synchronous architecture, the control unit of the asynchronous architecture has to synchronize the operations and avoid resource contentions through handshaking signals and status flags. Controlling data path instructions (e.g. *ADD, MLT, LOAD ACC,* etc.) which propagate through the data path stages is simple to handle with the handshake protocols and circuitry. But even for this type of instruction there is the question of how to execute instructions which for example do not require a multiplication: should the controller transfer the operand through the multiplier by multiplying it by one, or should there be another mechanism which bypasses the multiplier's operation (e.g. bypass path) and still assures the proper handshake operation ? The same question arises about instructions which do not have to execute any ALU operation: should the data be transferred through the ALU by adding it with zero, or should there be another mechanism which bypasses the ALU's operation and still assures the proper handshake operation ? Bypassing the block's operation is faster but the control is more complicated because then there is no "EOP" signal from the computation block, and an "EOP" signal must be generated from additional circuitry. Multiplying by one in the multiplier and adding to zero in the ALU is simpler because an "EOP" signal is then generated by the computation block, but unfortunately it consumes time to go through the computation blocks. Since pipeline dominated by worst case may not be a problem, controlling non-data path instructions such as: *STORE ACC, BRANCH, SET FLAGS, I/O* instructions, etc., is made more complicated by the lack a clock. The controller has to determine the correct timing to execute the instruction and how to assure proper execution and propagation of other instructions in the pipe. Another problem is how to discard from the pipe instructions already being executed when a *BRANCH* instruction has

to be executed.

PLA decoder

There are two mechanisms to control a synchronous pipeline architecture, one is time-stationary and the other is data-stationary. In both mechanisms the control source outputs control signals divided into fields where each field is dedicated for a particular stage. A time-stationary control mechanism provides the route and control and function select signals for the entire pipeline stages from one source. At each time interval, each field of the control source contains the control signals of the different instructions that are executed in the corresponding pipeline stages. In a data-stationary control mechanism the control signal "follow" the data through the pipeline providing the control signals at each stage as needed. The control source outputs all the control signals required to control the execution of one instruction during its propagation through the pipeline stages [4]. In a synchronous pipelined architecture, the use of time stationary control unit allows concurrent execution of different instructions in different stages. One way to control a pipeline architecture is to use a PLA (Programmable Logic Array - a ROM which only contains cells that are used) decodes the instructions and outputs data stationary control lines that are organized in separate fields. As explained above each field of the control lines controls the operation of a single stage of the pipeline. Delaying the different control fields through an appropriate number of shift register stages to match the pipeline stages, provides a mechanism in which the control signals of an instruction "follow" the propagation of its data in the pipeline stages. This way data stationary mechanism is converted into time stationary control [4].

Designing the control of the asynchronous pipelined architecture in a similar way, i.e., using a PLA to decode an instruction and converting its data stationary output control to time stationary control has the following advantages:

- Avoids adding any additional control delays to the existing delays of the data path.

- Enables simple timing and control of instructions that do not involve all or some of the data path stages.
- Enables instructions to be discarded simply during the execution of a conditional branch.

The asynchronous pipeline architecture depicted in figure 6.1 has a data stationary PLA control system which decodes an instruction and outputs the appropriate control word. Each control word is fully decoded, i.e., each bit is a control line, in order to avoid additional internal non-pipeline handshaking interconnection circuitry and time overhead. The control lines are organized into separate control fields where each field controls the operation of a single pipeline stage (computation block). Each control field propagates to its corresponding computation block through an appropriate number of pipeline stages of an asynchronous shift register as depicted in figure 6.6. The number of the shift register stages is designed to synchronize between the propagation of the data and the control field which controls the operation to be executed on it in the pipeline stage. In other words, the asynchronous pipelined shift register converts a data stationary control unit (PLA) into a time stationary control unit (PLA + shift registers) that allows the concurrent execution of different instructions in the stages of the data path. These asynchronous pipelined shift registers also assure that the number of handshaking stages that an "acknowledge" signal has to propagate from each pipeline stage of the data path (computation block) to the decoder stage (PLA) is the same whether it propagates through it's own control shift register stages or through any of the preceding data path stages and their control shift register stages. Therefore, each control field and the data necessary for executing an instruction have to propagate through the same number of handshaking delays until they reach the corresponding pipeline stage where the task related to the control field data can start its execution on the data. For this reason there are no additional handshaking delays imposed by the control unit.

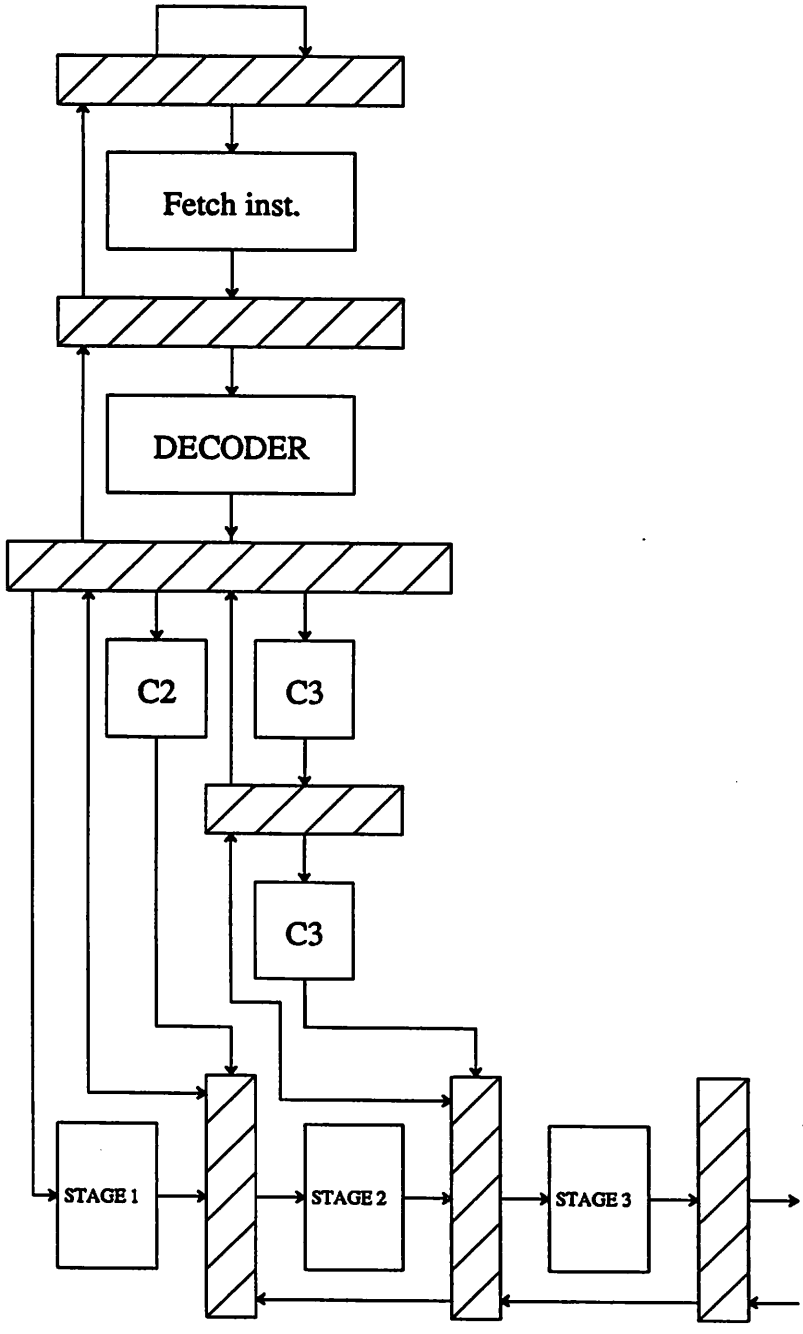


Figure 6.6 - Asynchronous data stationary control unit

As in any pipeline architecture, some instructions required that their related data propagates through some stages of the data path without changing it's value, e.g. *LOAD ACC, STORE ACC* etc. The *LOAD ACC* instruction causes an operand to be fetched from the memory and stored in the accumulator. In this instruction, the operand

has to propagate through the MULTIPLIER and the ALU without changing its value.

As was mentioned before, data can propagate in one of the two following ways: One way is to transfer the data through the multiplier by multiplying it by 1 and through the ALU by adding it with 0. Another way is to bypass the multiplier and the ALU, i.e., the data is transferred directly by handshaking from the input buffer of the multiplier or ALU block to the input buffer of the next computation block without passing through the multiplier or the ALU.

The first implementation executes the instruction in a regular asynchronous mode. Certain control fields of the decoded *LOAD* instruction control the multiplication of the operand by one in the multiplier and the addition of the operand with zero in the ALU. The appearance of the operand at the output of each computing block generates an "EOP" signal that initiates the handshaking circuit and protocol which transfers it to the next computing stage. This implementation might affect the processor's throughput because the computation delays in the multiplier and the ALU might be significant.

The second implementation requires a "NOFT" (no operation & data feed through) in the control fields of the multiplier and the ALU. "NOFT" instructs the computation block to transfer the data directly from its input buffer to its output without executing any manipulation on it. The data transfer is accomplished through a pass gate (uni-directional switch) path parallel to the combinational logic while the combinational logic of the block is idle. Since no manipulation is executed on the data an "EOP" signal is not generated by the combinational logic of the block and the data won't be transferred to next computation block. To overcome this problem, a control line of the "NOFT" control field will enable a special circuit in the computation block to generate an "EOP" signal after the incoming data has been latched properly in the input buffer. Meanwhile, this control line will cause the data to bypass the computation block through the pass gates. A parallel control line will disable the writting of the data into

the input buffer of the next computation stage while it is still operating on its previous data. The processor's throughput is not affected by this implementation because the data is fed through without any computation delays. But, there is a cost in overhead due to the additional control lines, the feed-through pass gates and the special "EOP" signal generation circuitry.

There are other instructions such as *OUTPUT* accumulator to an external I/O device, *BRANCH, SET FLAG, RESET FLAG, PUSH STACK, POP STACK, ENABLE INTERRUPT, CHECK STATUS,* etc., which are control instructions that do not require any data transfer or any operation in the pipeline stages. For such instructions the combinational logic of the computation blocks (pipeline stages of the data path) are not active during the execution phase of the instruction and therefore they do not generate any "EOP" signals. Since there is no need to execute any operation in the data path stages of the pipeline or to transfer data through them, the control fields of these stages will have "NOP"s (no operation and no data transfer). A "NOP" causes the combinational logic to be idle, disables the feed-through pass gates (unidirectional switch) and initiates a special circuit to generate the "EOP" signal. The use of "NOP"s insures the correct timing of executing such instructions by keeping the sequence of executing the instructions in the same order that they were fetched from the memory. (The sequential order of executing instructions is important for the correct flow of the program).

Another problem typical in the pipeline architecture is how to execute branch instructions with a minimum number of "bubbles". In addition, the asynchronous pipeline architecture must also discard instructions which are already in the pipe when a branch is encountered and executed.

It is important to distinguish between unconditional branch instructions and conditional branch instructions. An unconditional branch instruction preferably executes immediately after decoding (during the "fetch operand" stage), thus minimizing the

instructions following the branch that have to be discarded from the pipe. In the architecture depicted in figure 6.1, the execution of an unconditional branch involves discarding of two instructions from the pipe (one in the "fetch operand" stage and the other in the "decoder" stage). A conditional branch instruction preferably executes only after the condition was evaluated. Doing so avoids the restoration of the data and the status prior to the branch and the discarding of the instructions following it.

In both cases, it is possible to avoid the need to discard instructions from the pipe by inserting an appropriate number of "NOP" (no operation) instructions in the program after the branch or the conditional branch instructions. However using this technique wastes many instruction cycles in the conditional branch case. These wasted instruction cycles due to the insertion of "NOP" instructions after a conditional branch can be avoided by using some prediction policy and being able to discard instructions from the pipe. Two basic prediction modes are described below.

One mode predicts that the branch is not going to be executed and the processor continues to fetch and execute the instructions which follow the conditional branch. At the appropriate time, the condition is evaluated. If the prediction was correct, the processor continues it's operation without any interruption. If the prediction was wrong, the processor has to discard all the instructions which are already in the pipe, restore its status and data prior to the branch instruction, load the program counter with the branching address and continue from there.

The other mode predicts that the branch is going to be executed. In this case, the processor discards from the pipe the two instructions which have been already fetched, updates the program counter and stores it for restoration (if it will be necessary), loads the program counter with branching address and continues to fetch and execute instructions from there. At the appropriate time, the condition is evaluated. If the prediction was correct, the processor continues it's operation without any interruption. If the pred-

iction was wrong, the processor has to discard all the instructions which are already in the pipe, restore its status, data and the program counter prior to the branch instruction, and continue from there.

It is obvious that the first prediction policy is simpler to implement and requires less overhead.

In the case of an unconditional branch it is possible to avoid the "bubbles" due to discarding the two following instructions only if a delayed branch technique is used, i.e., the compiler inserts the branch instruction two instructions ahead so that its execution is synchronized with the program flow.

Discarding instructions is done by inserting, in the appropriate stages of the pipelined shift registers, "NOP"s instead of the decoded control fields of these instructions.

Summary of control concepts

In summary, it has been shown that the control unit of the asynchronous architecture is not more complicated than that of the synchronous architecture. Using the appropriate handshaking technique and circuits makes it feasible to design. The major concepts of the control unit are:

- No additional handshake delays due to the asynchronous control unit.
- Simple decoder based upon a PLA provides fully decoded "wide" data stationary control word, and saves additional handshaking circuit and delays overhead.
- The control word is divided into fields, each controlling a single computation block (pipeline stage).
- Propagation of data through a computation block (pipeline stage) by bypassing it is controlled and executed by "NOFT" (no operation & data feed through) in the control field.
- "NOP" (no operation) in the control field causes a computation block to be idle.

- Control lines of "NOP" and "NOFT" initiates the generation of "EOP" signal even though the computation block is idle.
- Pipelined shift registers convert data stationary control words into time stationary control words.
- Pipelined shift registers enable instructions to be discarded from the pipe by inserting "NOP"s in the appropriate stages.

6.1.2.3. Multiplier

The multiplier depicted in figure 6.7 consists of the combinational logic of the multiplier, pass gates for data feed through, a register for the control field and two data input registers.

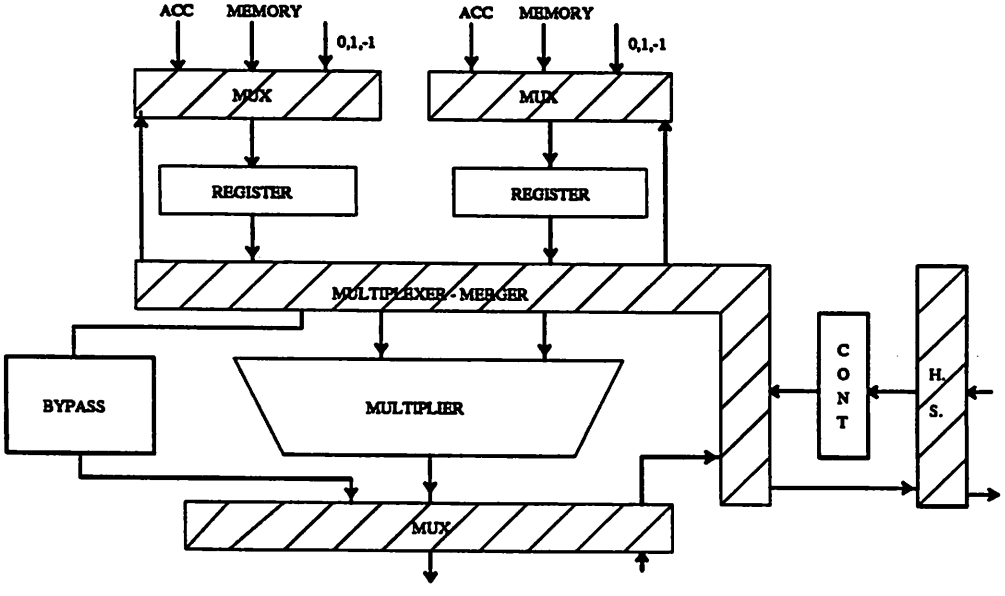


Figure 6.7 - Multiplier

Four of the handshake interconnections are pipelined: two input multiplexers and the two FHS (full handshake interconnection). One FHS transfers the control field from the shift register of the control unit and the other FHS transfers the output of the computation block to the next block. The other handshake interconnection (MULTIPLIER

MERGER) is non-pipeline. According to the instruction, the input multiplexers select data from: data memory, accumulator (feedback data), constant 1, constant 0, constant -1 and data from an I/O device. Thus, this configuration permits multiplication of two operands from the memory. The merger is required to insure the validity of the input data and the control field before the combinational logic executes its task.

As mentioned before, the multiplier is also bypassed when no multiplication is required.

6.1.2.4. ALU

The ALU depicted in figure 6.8 consist of the combinational logic of the arithmetic and logic unit, pass gates for data feed through, two data input registers, an accumulator and a register for the control field. As in the multiplier, only four handshake interconnections are pipelined: two input multiplexers and two FHS (full handshake interconnection). One FHS transfers the control field from the shift register of the control unit and the other FHS transfers the accumulator to the next block. All other handshake interconnections are non-pipeline. Data from the memory is selected to this unit indirectly through the multiplier. Therefore this configuration permits an ALU operation to be executed on only one operand from the memory. The merger is required to guarantee the validity of the input data and the control field before the combinational logic executes its task.

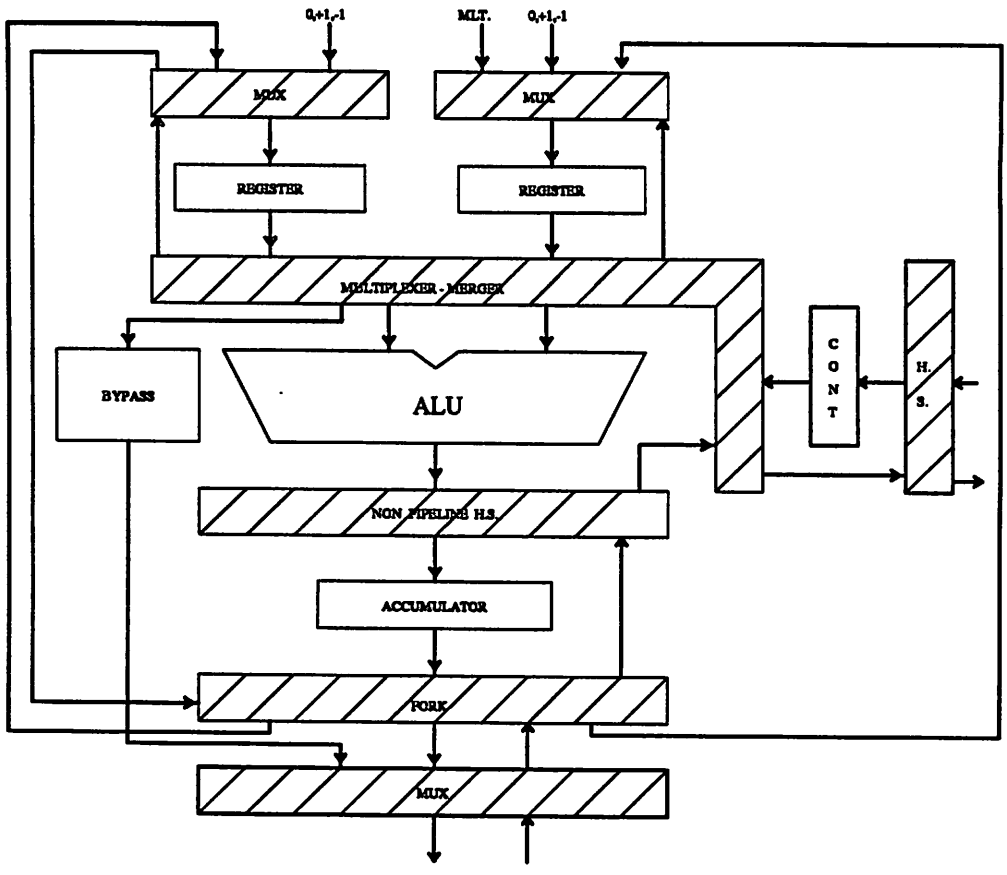


Figure 6.8 - ALU

6.1.2.5. Data memory access

Any pipeline architecture containing separate stages that access the same resource have the possibility of contention for accessing the resource. In figure 6.1 "fetch operand" and "write" stages require simultaneous access to the same data memory, thus resulting in an access conflict between the two. In the synchronous implementation this contention (resource conflict) can be solved in a number of ways. One way is to divide the basic clock cycle into subcycles (time slots). By assigning different non overlapping subcycles to different stages the conflict is solved because the memory is accessed twice in different time slots during the same basic cycle.

Another way is by eliminating the "write" stage and attaching the write operation

to the "fetch operand" stage, creating a new stage which is "fetch/ write operand". The new stage which follows the "decoder" either fetches operands from the memory or stores results in the memory. Therefore, during one cycle the resource is accessed only once, solving the problem of contention (conflict in accessing the same resource). This solution requires precaution by the programmer or the compiler since the execution of a store instruction is done before the result is ready and valid (pipelined data dependency). Placing such instructions in the right location in the program solves the problem, but it puts constraints on the programmer.

In the asynchronous pipeline implementation, there is no basic clock that can be divided into time slots and therefore it is not possible to use such solutions. The other solution which provides a new stage of "fetch/write operand" is implemented as depicted in figure 6.9. The selection between the "fetch operand" (read from the memory) and "write result" (write to the memory) is done by one bit in the control field. Again as in the synchronous implementation, such a solution puts constraints on the programmer because of the data dependency problem described above.

Nevertheless, it is still possible to implement an asynchronous pipeline architecture with two separate stages for "fetch operand" and "write result".

One simple way is to use an asynchronous dual port memory. Such a memory configuration has two separate address, data and control inputs, and also requires separate completion signals: EOP_{write} and EOP_{fetch}. Two separate "EOP" signals allow us to use the memory as an external device accessed by the "fetch operand" stage and the "write" stage without any conflicts or restrictions. Conflicts in simultaneously accessing the same location by both stages is solved as in the synchronous dual port memory, and there is no preference for either stage to access the memory first.

Another way is to impose a temporary dependency between the two stages by the control of the handshaking. Temporary dependency means that the "fetch operand"

stage will access the memory only after it has been accessed by the "write" stage. If no write operation is required, the "write" stage executes a "NOP" thus keeping the order of accessing the memory. Figure 6.9 depicts a way to implement such memory access.

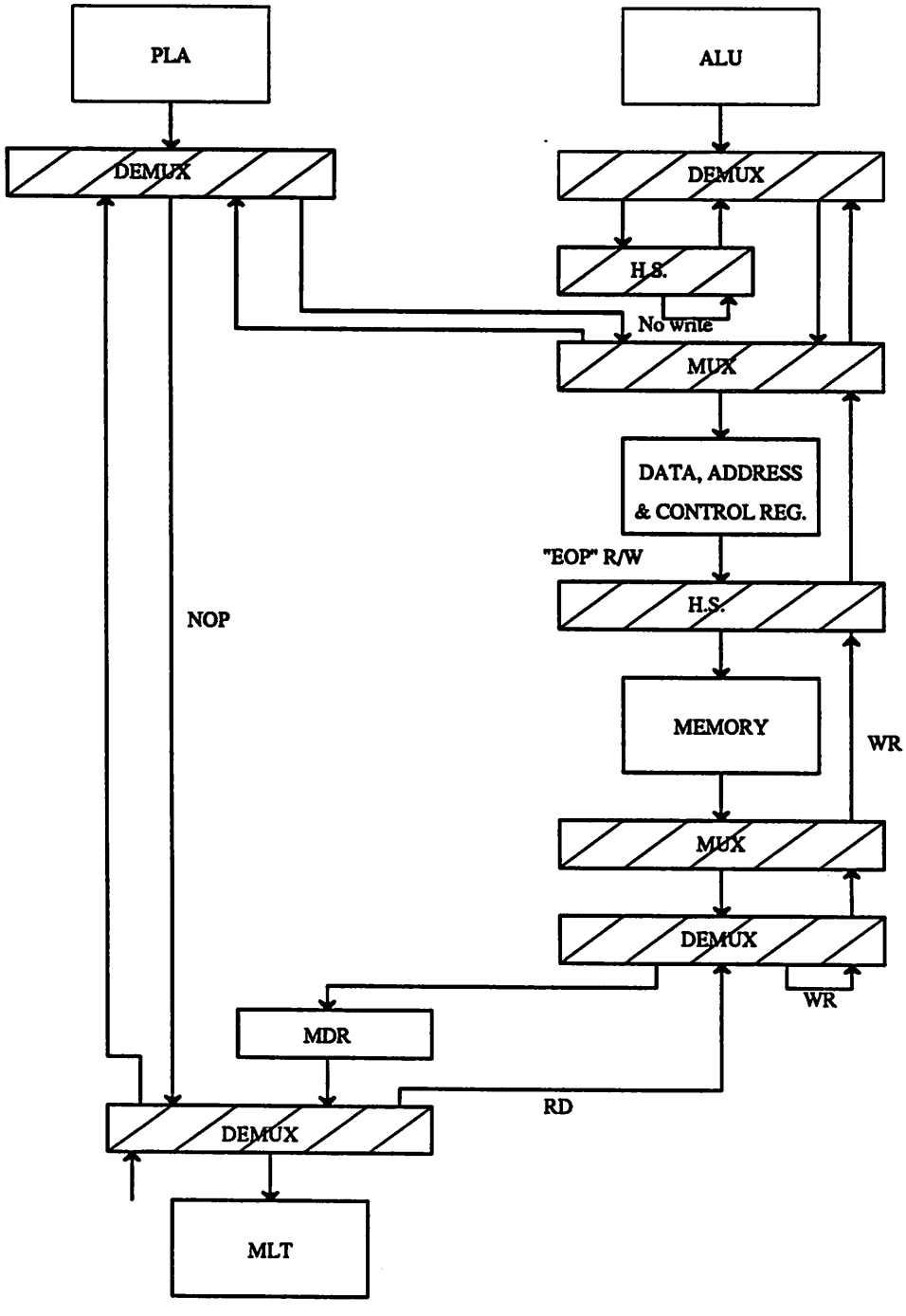


Figure 6.9 - Memory access

Memory's input multiplexer is controlled by lines from the control fields of the "write" and the "fetch operand" stages. These lines will indicate to the handshake interconnection whether accessing the memory is required by both stages, one of the stages or none of them. If both stages require an access to the memory, the handshake interconnection will first allocate the memory to the "write" stage and afterwards to the "fetch operand" stage. When only the "write" requires access to the memory, it will get the service and the "fetch operand" will execute a "NOP". When only the "fetch operand" requires access to the memory, it will get the service immediately and the "write" stage will execute a "NOP".

Since the PLA decoder is a data stationary controller and the instruction's execution order is kept by "NOP"s, the memory during one execution "cycle" can be accessed only once by "write" and "fetch operands" stages.

6.1.2.6. Feedback

Any processor requires feedback loops for adding or multiplying input data with the content of the accumulator. The control of the feedback in a pipeline synchronous processor is simple because all stages are synchronized by the same clock. In the asynchronous implementation it is also possible to implement a feedback loop but it is necessary to add one more latch in the feedback loop. The added latch, depicted in figure 6.10, operates as a temporary buffer and is accessed by full handshake interconnection. To prevent deadlocks it is very important to synthesize the feedback loop components with the right initial conditions as described in [5]. As in any pipeline architecture, more pipeline stages in the forward loop mean a longer delay for the loop [i.e., data has to propagate through more stages before it can be used as a feedback, larger latency]. Unlike the *LOAD* instruction, the execution of control instructions do not require data propagation through the pipeline stages and therefore the control lines that generate "EOP" signals are not needed. But, since the implementation is asynchro-

nous and control fields of the control instructions are "NOP"s, all the preceding instructions in the pipe and all the succeeding ones will continue to be executed properly, in the right sequence, and with the correct timing.

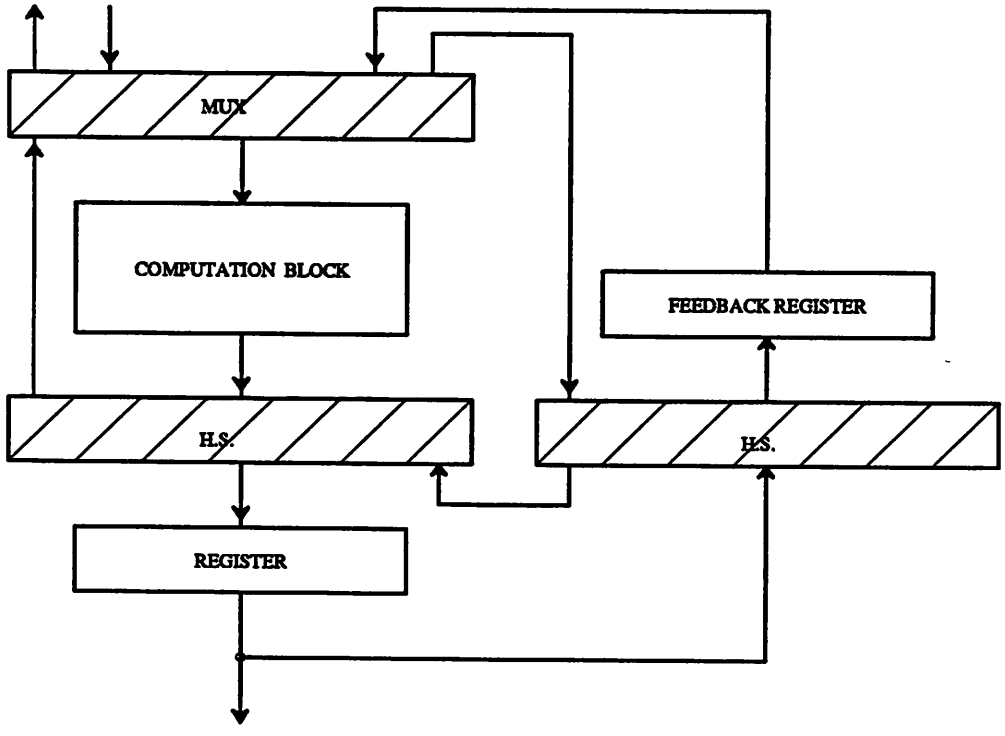


Figure 6.10 - Feedback

6.1.2.7. ASIC - Application Specific Integrated Circuits

In designing an architecture for a specific application (ASIC), it is possible to achieve a better performance. In the ASIC case, the program is compiled ahead and the PLA decoder is not required. The control is time stationary and each computation block (pipeline stage) has its own ROM controller which controls its operation. Synchronizing the various computational blocks is realized by using conditioned handshaking between them and between their ROM controllers, and thus avoiding the use of the "NOP" technique. A simple example is the control instruction *Clear flag*. Inserting this instruction only in the ROM controller of the pipeline's last stage causes the processor

to execute it with the right timing without needing to propagate the instruction through the preceding stages. In this implementation, the control ROMs of different stages will have a different number of control words. Therefore, different computation blocks will operate more times than others but the overall synchronization of the program is provided by the conditioned handshaking technique between the interconnection blocks. If the execution time of a "NOP" is short relative to the execution times of other computation blocks, the throughput of the processor is not increased by eliminating the use of "NOP".

6.2. Hybrid pipeline architecture

6.2.1. Introduction

Another configuration of an asynchronous pipeline architecture is the "hybrid" pipeline architecture depicted in figure 6.11. Unlike the regular pipeline architecture where the data path stages are connected in serial, in this architecture the data path stage are connected in parallel between two buses. The architecture can be partitioned into two parts: control path and data path. The control path executes the "fetch" cycle of an instruction and consist of 3 serial pipeline stages: "fetch operand", "decode" and "fetch operand". The data path performs the "execution" cycle of an instruction and is the fourth pipeline stage. It consist of 4 parallel computation blocks : "MAC" (multiplier and accumulate), "ALU", "shifter" and an execution block for control instructions. Parallel data path stages allows the processor to be operated either as a "RISC" (Reduced Instruction Set Computer) type processor or as a modified pipeline processor. The advantages and disadvantages of each of these two types is described in the following sections.

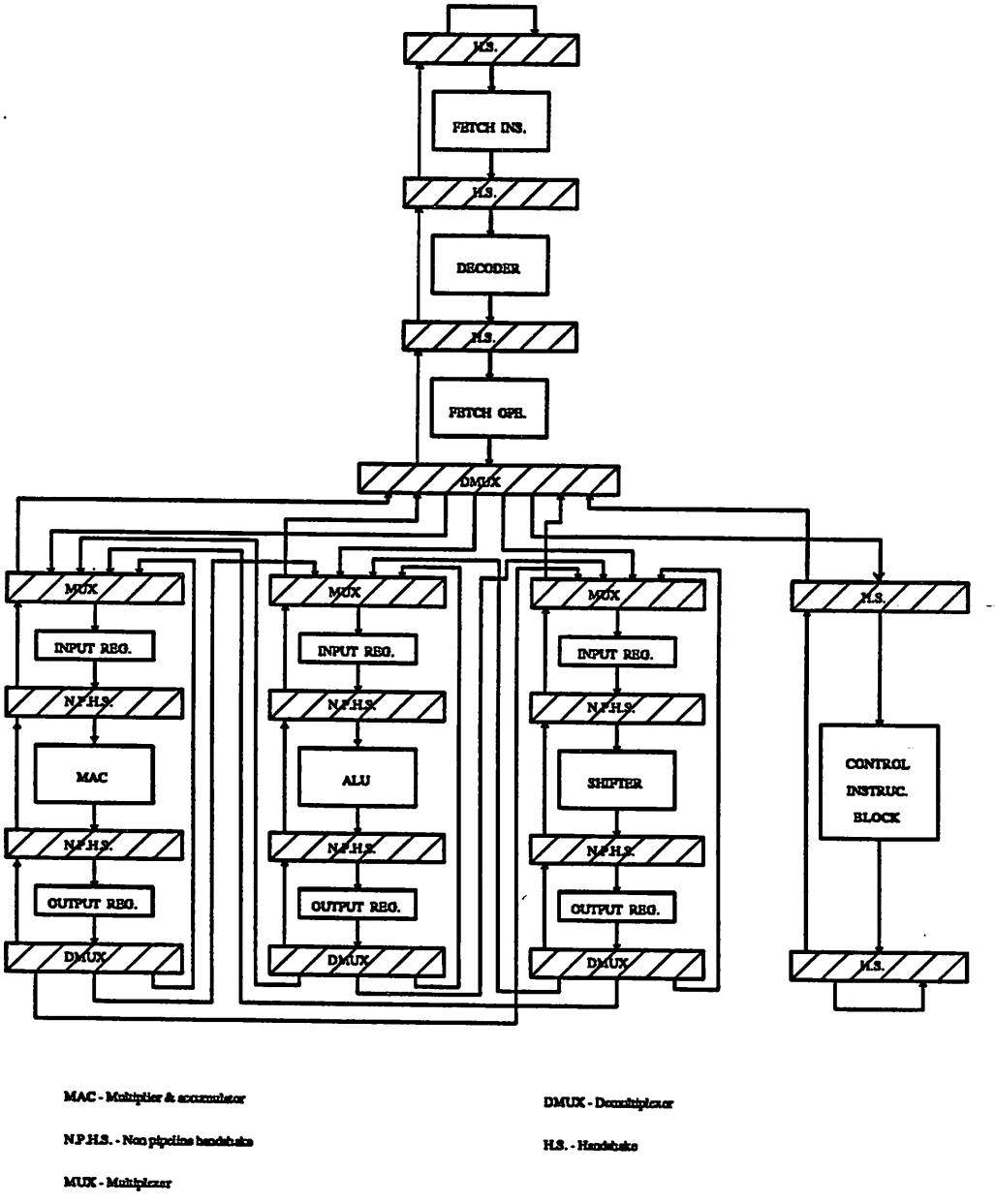


Figure 6.11 - Hybrid pipeline architecture

6.2.2. Methods of operation

As mentioned above, the "hybrid" pipeline architecture can operate either as a "RISC" type processor or as a modified pipeline processor.

6.2.2.1. Hybrid - RISC type architecture

In the "RISC" type operation only one of the parallel data path computation blocks operates in each time interval (e.g. multiplier or ALU or shifter). Therefore, this type of architecture handles and executes only instructions with one data path operation. Instructions which require multiple data path operations such as "multiply and shift" can't be executed. Such multiple data path operations are performed by executing two consecutive instructions: "multiply" and "shift". The operation and the control unit of the "RISC" type configuration is simple because there is no resource conflict between successive instructions. Non-data path instructions such as: *BRANCH, SET FLAG, STORE ACC, OUT* etc., are executed during the forth time interval which is the time interval in which a data path instruction is executed (fourth pipeline stage). To do so, the control unit inserts "NOP"s during the "fetch operand" stage of the execution of these instructions. This implementation of executing all the instructions only during the interval of the data path stage eliminates data and branch dependency problems and doesn't impose any restrictions on the programmer. Data is always ready for the next instruction, as in the case of *STORE ACC* which is executed after the result of previous computations is already ready in the accumulator. In the conditional branch case, the condition flags are ready for evaluation before the following instructions have made any new changes in the data path. The throughput of this "RISC" type "hybrid" architecture is the same as the throughput of the regular sequential pipeline architecture. The reservation tables (table 6.1 and table 6.2) of the regular and the "hybrid" architectures illustrate this fact. A reservation table is a two dimensional representation used to describe and analyze concurrent activities within the different stages of a pipeline type architecture.

Assume that the following four instructions have to executed by the two architectures. The first instruction (1) is multioperation that requires three operations (e.g. multiplication, ALU operation and shift) in the data path. The other three instructions

(2,3,4) are single operations requiring only one operation (multiplication or ALU operation or shift) in the data path. A reservation table of the regular pipeline architecture shows that the execution of these four instructions requires nine time intervals (9Δ).

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1	2	3	4	-	-	-	-	-
Decode	-	1	2	3	4	-	-	-	-
Fetch ope.	-	-	1	2	3	4	-	-	-
MLT	-	-	-	1	2	3	4	-	-
ALU	-	-	-	-	1	2	3	4	-
Shifter	-	-	-	-	-	1	2	3	4
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.1 - Reservation table - Regular pipeline architecture

In the RISC type "hybrid" architecture instructions containing multiple operations cannot be executed, therefore instruction (1) in table 6.1 which is a multioperation instruction, is partitioned into 3 single operation instructions: 1_a (multiplication), 1_b (ALU operation), and 1_c (shift), depicted in table 6.2. Because of this partition the "hybrid" architecture needs to execute seven single operation instructions. A reservation table of "hybrid" pipeline architecture shows that the execution of these seven single operation instructions requires also only nine time intervals (9Δ).

For a sequence containing many multi-operation instructions, the regular pipeline architecture achieves a higher throughput than the "hybrid" RISC type architecture. But, it is very unlikely to have a program with a sequence of multi-operation instructions which are data-independent. Hence, if the instructions are data dependent, the regular pipeline architecture requires "NOP" instructions between them which is equivalent to

the partition of the multi-operation instruction into a sequence of single operation instructions. And therefore, even for such sequences the throughput is likely to be almost the same.

Deleting instructions from the pipeline stages when a branch prediction is wrong is executed as in the regular asynchronous pipeline architecture. But since there is no branch or data dependency, there is no need to restore status flag or registers of the data path.

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1_a	1_b	1_c	2	3	4	-	-	-
Decode	-	1_a	1_b	1_c	2	3	4	-	-
Fetch ope.	-	-	1_a	1_b	1_c	2	3	4	-
MLT	-	-	-	1_a	-	-	2	-	-
ALU	-	-	-	-	1_b	-	-	3	-
Shifter	-	-	-	-	-	1_c	-	-	4
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.2 - Reservation table - "Hybrid" RISC type architecture

6.2.2.2. Hybrid - modified pipeline architecture

General Description

This architecture is a modification (parallel data path stages) of the regular pipeline architecture (serial data path stages). As later will be explained and illustrated with examples, using conditional handshake interconnections allows the parallel data path stages to operate concurrently if necessary. Therefore it is also possible to execute instructions which require multiple data path operations. Controlling this architecture is

more complicated because it is necessary to solve resource conflicts and data and branch dependency problems.

Since during the execution time different instructions might use a different number of data path stages, resource conflicts could occur. Resource conflicts occur because different instructions do not use the same number of data path stages during their execution cycle. A resource conflict occurs when two successive instructions require the same computation block during the same execution time interval. For example, when a multiple operation instruction requires the use of two or more consecutive computation blocks in the data path during its execution time intervals, is followed by a single or multi-operation instruction that during its execution time interval (which coincides with that of the multi-operation instruction) requires the same computation block as required by the first one, there will be a resource conflict (examples 1,2 and 3 described later illustrate this fact).

Resource conflict problems can be solved by using a FFFS priority policy and employing priority conditional handshake interconnection blocks. FFFS stands for "first fetched first served", which means that an instruction which has been fetched first has the priority of accessing and using a data path stage over an instruction which has been fetched later. This policy is utilized by employing priority conditional handshake interconnection. A stage of the data path can be accessed by handshake only if the previous instruction does not require it. If the previous instruction requires it, the control unit delays the access by the new instruction and the resource conflict is prevented. By doing so, the control unit imposes a delay of one stage's execution time to the instruction that was denied access and to all the instructions following it.

But this architecture has the problems of data and branch dependency which exist in the regular pipeline architecture. Data dependency means that some instructions might require data for their execution from a previous instruction which is not com-

pleted yet. Control dependency means that a conditional branch instruction cannot be executed because the flags are not ready yet for evaluation.

As in the regular pipeline architecture, data dependency and branch dependency put constraints on the programmer. Inserting "NOP" instructions whenever necessary solves the problem but reduces the throughput. Careful programming may avoid some of the dependency problems but requires the programs to be written in assembly language.

For most programs the throughput of the modified pipelined asynchronous architecture should be higher than that of the regular pipelined asynchronous architecture. The throughput is the same only if the program consists of multi-operation instructions because in the existence of resource conflicts, the "hybrid" modified pipeline architecture operates as a regular pipeline architecture. When there are no resource conflicts the instructions are executed faster because the instructions pass only through the required data path stages and they are executed concurrently.

The following simple examples illustrates the throughput comparison.

Example 1

Assume that four instructions have to be executed in both architectures. Instruction 1 and 2 are multi-operation instructions and instructions 3 and 4 are single operation instructions. Execution of instructions 1 and 2 require only the multiplier and the ALU. Instruction 3 is executed in the shifter while instruction 4 is executed in the ALU.

In the regular pipeline architecture, the execution of these instructions require eight time-intervals (8Δ) as shown in table 6.3 below.

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1	2	3	4	-	-	-	-	-
Decode	-	1	2	3	4	-	-	-	-
Fetch ope.	-	-	1	2	3	4	-	-	-
MLT	-	-	-	1	2	-	-	-	-
ALU	-	-	-	-	1	2	-	4	-
Shifter	-	-	-	-	-	-	-	3	-
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.3 - Reservation table - Regular pipeline architecture

In the "hybrid" modified pipeline architecture the execution of these instructions require only seven time-intervals (7Δ) because there is no resource conflict. shown in table 6.4 below.

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1	2	3	4	-	-	-	-	-
Decode	-	1	2	3	4	-	-	-	-
Fetch ope.	-	-	1	2	3	4	-	-	-
MLT	-	-	-	1	2	-	-	-	-
ALU	-	-	-	-	1	2	4	-	-
Shifter	-	-	-	-	-	3	-	-	-
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.4 - Reservation table - Modified pipeline architecture - No conflict

Example 2

Assume that four instructions has to be executed in both architectures. Instruction

1 and 2 are multi-operation instructions and instructions 3 and 4 are single operation. Execution of instruction 1 requires the multiplier, ALU and shifter, while execution of instruction 2 requires only the multiplier and the ALU. Instruction 3 is executed in the shifter while instruction 4 is executed in the ALU.

In the regular pipeline architecture the execution of these instructions require eight time-intervals (8Δ) as previously shown in table 6.3.

In the "hybrid" modified pipeline architecture the execution of these instructions also require eight time-intervals (8Δ) because of a resource conflict. During the sixth time interval (Δ_6), instruction 3 has a resource conflict and its execution is delayed by one time interval. These results are shown in table 6.5 below.

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1	2	3	4	-	-	-	-	-
Decode	-	1	2	3	4	-	-	-	-
Fetch ope.	-	-	1	2	3	-	4	-	-
MLT	-	-	-	1	2	-	-	-	-
ALU	-	-	-	-	1	2	-	4	-
Shifter	-	-	-	-	-	1	3	-	-
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.5 - Reservation table - Modified pipeline architecture - With conflict

Example 3

Assume that four instructions has to be executed in both architectures. Instruction 1,3 and 4 are multi-operation instructions and instructions 2 is single operation. Execution of instruction 1 requires the multiplier, ALU and shifter, while execution of instructions 3 and 4 requires only the multiplier and the ALU. Instruction 2 is only exe-

cuted in the ALU.

In the regular pipeline architecture the execution of these instructions require eight time-intervals (8Δ) as previously shown in table 6.3.

In the "hybrid" modified pipeline architecture the execution of these instruction requires nine time-intervals (8Δ) because of a resource conflict. During the fifth time interval (Δ_5), instruction 2 has a resource conflict and its execution, as well as the execution of the following instructions, is delayed by one time interval. These results are shown in table 6.6 below.

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1	2	3	4	-	-	-	-	-
Decode	-	1	2	3	-	4	-	-	-
Fetch ope.	-	-	1	2	-	3	4	-	-
MLT	-	-	-	1	-	-	-	3	4
ALU	-	-	-	-	1	2	-	3	4
Shifter	-	-	-	-	-	1	-	-	-
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.6 - Reservation table - Modified pipeline architecture - With conflict

Configuration's Constraint

This scheme of operation imposes a constraint on the execution of consecutive instructions. When two instructions A and B require the same resource C, preference is given to A, the first instruction which was fetched from the memory. Instruction B will use the same resource C only after A has finished its task and its result was transferred to the next computation stage as required. Therefore, this configuration is limited in its efficiency for single operation instructions or for special cases of multi-operation and

single-operation instructions.

Control unit of the hybrid architecture

The control unit of the asynchronous "hybrid" modified pipeline architecture is more complicated than that of the asynchronous regular pipeline architecture. The sequence of executing an instruction in the data path is still "fetch operand" -> "multiplier" -> "ALU" -> "shifter". But this architecture has the capability to bypass computation blocks which are not required for the execution of an instruction and to process data only in the required computation blocks. This bypassing feature avoids the use of "NOP"s in the control field and in some cases will speed up the execution of the instruction. As in the asynchronous regular pipeline architecture, the PLA decoder of this architecture outputs a fully decoded data stationary control word which is divided into the following fields:

Fetch inst.	Decode	Fetch ope.	Exec. 1	Exec. 2	Exec. 3	Control exec.

The first 3 control fields control the fetch cycle of the instruction while the last 4 control fields control the execution cycle of the instruction. Data path instructions are executed sequentially with the bypassing feature in the data path computation stages: multiplier, ALU, shifter. Control instructions are executed in the control execution block. To execute instructions concurrently and to synchronize their execution, the control fields propagate through an appropriate number of shift registers. Unlike the decoder of the regular asynchronous pipeline architecture, the "Exec. " control fields of the data path in this architecture are not dedicated for a specific computation block. Control field "Exec. 1" is dedicated to control the first operation on the data. According to the decoded instruction it controls the operation in the multiplier or the ALU or the shifter. Control field "Exec. 2" is dedicated to control the second operation on the data. According to the

decoded instruction, it controls the operation in the ALU or the shifter. Control field "Exec. 3" is dedicated to control the third operation on the data, and it controls the operation in the shifter. Partitioning the control fields as explained above allows the bypass of unnecessary computation blocks. The use of shift registers at the output of the decoder converts the control fields to time stationary and maintains the sequence of operations as originally defined. Each "Exec. " control field is divided into two subfields: one controls the operation of the required computation block and the other (P1-P3) controls the DMUX and the MUX interconnection blocks of the data path's computation blocks. In each "Exec. " control field only one of the control lines P1-P3 may be active. If no operation is required none of these control lines will be active.

Figure 6.12 depicts the sub fields of "Exec. 1" - "Exec. 3", and the way that they control the interconnection blocks:

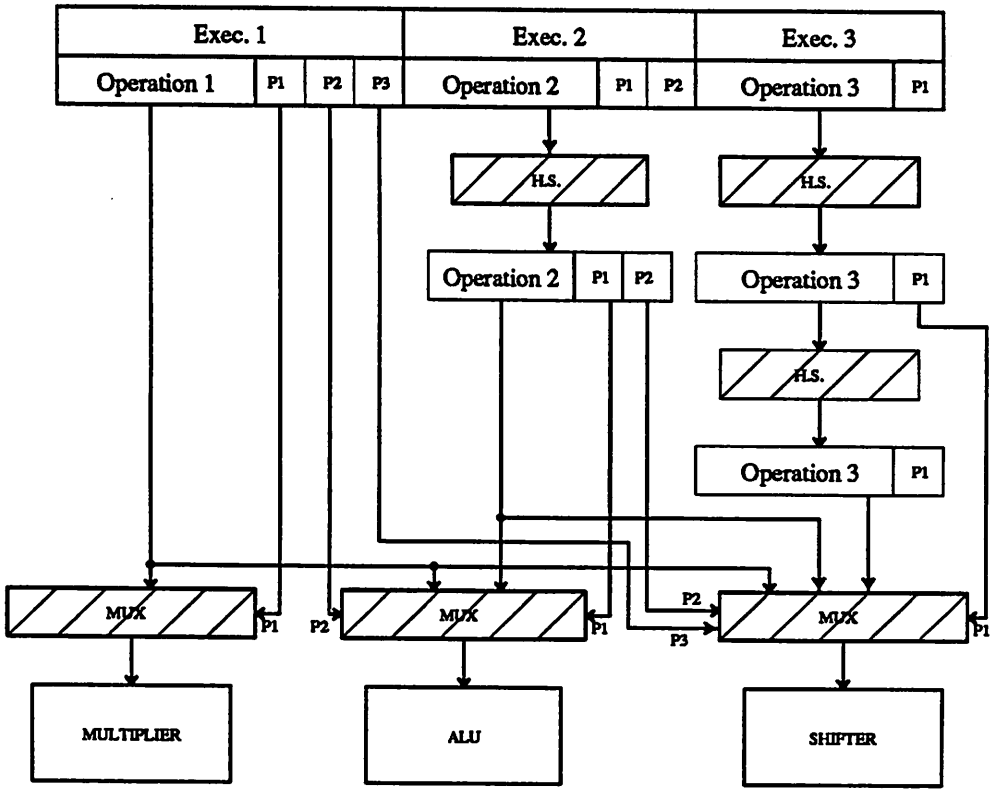


Figure 6.12 - Control fields and interconnections

Principles of operation

The multiplexers (MUX) of the computation blocks are merged-conditioned-multiplexers. The multiplexers are merged, because they start to operate only after handshake has been established with all their priorities controls. The multiplexers are conditioned, because they operate like arbiters with predetermined "daisy chain" priority policy (a detailed explanation appears later in this section). Priority control inputs P1-P3 determine the sequence of transferring the corresponding operation control field and data to the computation block, where P1 has the highest priority and P3 has the lowest (P1 > P2 > P3). When one of the priority control lines is inactive, the multiplexer skips to the next one. "A_o" signal (of the four phase handshaking depicted in figure 5.3), which indicates that the multiplexer is ready to transfer a new set of data and controls, is activated only after the current sequence of data and control has been transferred and executed in the corresponding computation block.

Conflicts between consecutive instructions requiring the same computation block are solved by the conditioned multiplexers that operate as sequential arbiters which transfer data and control to the computation block in a predetermined "daisy chain" priority policy. A new set of data and control will be transferred only at the end of the transferring and executing the current set in the computation block. Continued conflicts between instructions can cause the "hybrid" modified pipeline configuration to have the same throughput as that of a regular pipeline architecture. When conflicts are infrequent the throughput will be better.

The operation of the priority conditioned data path multiplexers and the computation blocks depicted in figure 6.12 is as follows. Sub fields P1-P3 of "Exec. 1" are connected to the control inputs of the multiplexers of the computation blocks. Since data is executed in the sequence of multiplier -> ALU -> shifter as mentioned before, P1 is connected to the first priority input of the multiplier's multiplexer. P2 is connected to

the second priority input of the ALU's multiplexer and P3 is connected to the third priority input of the shifter's multiplexer. When P1 of "Exec. 1" is active (P2 and P3 are inactive) the data from "fetch operand" block will be transferred to the multiplier. When P2 of "Exec.1" is active (P1 and P3 are inactive) data from "fetch operand" block will be transferred to the ALU and operation will be executed on it only if the preceding instruction does not require the ALU, i.e. P1 in "Exec. 2" of the preceding instruction is not active. If P1 in "Exec. 2" of the preceding instruction is active the ALU will first operate on the preceding instruction's data and afterwards on the current instruction's data. This mechanism allows the operation to by-pass the multiplier and guarantees the proper sequential operation of the ALU in case of access conflicts. When P3 of "Exec.1" is active (P1 and P2 are inactive) data from the "fetch operand" block will be transferred and executed in the shifter only if it is not required by the two preceding instructions. If the preceding instructions require the shifter, i.e., P1 of "Exec. 3" and P2 of "Exec. 2" are active, the data from the "fetch operand" stage can be transferred to the shifter and processed there only after the shifter executed the previous instructions. Again, as before, this mechanism solves access conflicts and guarantees the proper sequential execution of the instructions in the shifter. The same principle of operation applies to other instructions waiting for execution in the computation blocks of the data path.

Therefore when there is an access conflict to the computation blocks of the data path the "hybrid" modified pipeline architecture operates as depicted in the following table. 6.7

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1	2	3	4	-	-	-	-	-
Decode	-	1	2	3	4	-	-	-	-
Fetch ope.	-	-	1	2	3	-	4	-	-
MLT	-	-	-	1	2	-	-	-	-
ALU	-	-	-	-	1	2	-	4	-
Shifter	-	-	-	-	-	1	3	-	-
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.7 - Reservation table - Modified pipeline architecture - With conflict

When there is no access conflict to the computation blocks of the data path the "hybrid" modified pipeline architecture operates as depicted in the following table. 6.8.

stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Fetch inst.	1	2	3	4	-	-	-	-	-
Decode	-	1	2	3	4	-	-	-	-
Fetch ope.	-	-	1	2	3	4	-	-	-
MLT	-	-	-	1	2	-	-	-	-
ALU	-	-	-	-	1	2	4	-	-
Shifter	-	-	-	-	-	3	-	-	-
Control exc.	-	-	-	-	-	-	-	-	-

Table 6.8 - Reservation table - Modified pipeline architecture - No conflict

Conclusions

Two types of the "hybrid" architecture described in this chapter are based upon the asynchronous concepts and design rules developed before. The control unit of the "RISC" type is similar to the one employed in the standard asynchronous pipeline

architecture. The control unit of the "hybrid" modified pipeline architecture is more complicated. The property of by-passing unused computation blocks requires priority arbiter conditioned interconnection blocks. These interconnection blocks by-pass computation blocks when there is no resource conflict as well as provide proper synchronized sequential execution when there are resource conflicts. These interconnections speed up the concurrent execution of instructions by avoiding the use of "NOP"s in the control field of the data path's computation blocks. Executing non data path instructions during the time interval of the data path stage eliminates data dependency problems as well as conditional branch dependency problems. Deleting instructions from the pipeline stages is executed simply as described before. Throughput of the RISC type "hybrid" architecture is almost the same as that of the regular pipeline architecture, and the control unit of the RISC type architecture is simpler. The throughput of the "hybrid" modified pipeline architecture is better than that of the regular pipeline architecture.

6.3. Common bus asynchronous architecture

6.3.1. Introduction

As was described before in chapter 5, proper operation of the asynchronous pipeline architecture requires handshake interconnection blocks between the computation blocks. These blocks initiate and control the data transfer between the stages of the pipeline (computation blocks). The handshake initiation propagates forward, in the data flow direction, from the first stage to the last, while the start of task execution propagates successively from the last stage to the first. If the handshake interconnection is much shorter than the execution times of the different stages, all the stages operate concurrently.

Since start of task execution in the pipeline stages "ripples" successively from the last stage to the first one, some questions arise: Is it possible to avoid the

interconnection blocks circuitry overhead and to perform the data transfer between the stages only through one common interconnection block ? What would be an appropriate processor architecture ?

The answer to these questions leads to an architecture, depicted in figure 6.13 where all the computation blocks are connected in parallel to a common bus. This architecture is a pipelined architecture where data transfer between the parallel computation blocks is done through the common bus, which is controlled by only one interconnection block, thus saving interconnection circuitry overhead.

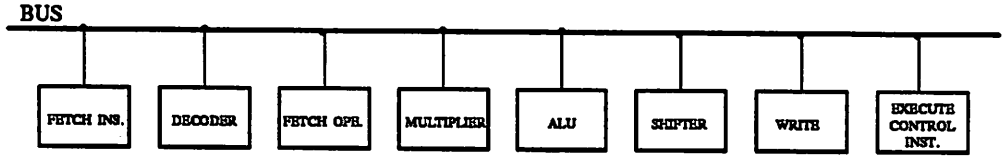


Figure 6.13 - Common bus configuration

The design of a common bus synchronous processor is simple due to the existence of a global clock. But in the asynchronous case, there is no clock and therefore the design is more complicated. This section develops a way to design a common bus asynchronous processor and gives answers to the following questions:

1) What are the properties and characteristics of an asynchronous common bus processor ?
2) How to control the common bus ?
3) How to avoid asynchronous arbiter problems ?
4) Will the common bus save circuitry overhead in the handshaking and the data transfer between the processor blocks ?
5) What are the advantages, if any, of such a configuration ?

6.3.2. Asynchronous common-bus design approach

The global clock of a synchronous architecture makes it simple to implement a synchronous common bus architecture. All blocks of the processor are connected to a bus for data transfer between them. Data transfer between the different blocks is done in a time division multiplexing mode. According to the decoded instructions, the control unit determines and assigns the appropriate time slots for data transfer so that all the units execute their tasks concurrently. Data transfer between the blocks is serial, and the throughput of such an architecture might be smaller than the throughput of a synchronous pipeline architecture.

Implementation of an asynchronous pipelined architecture with a common bus for communication between the stages, is more complicated because there is no global clock to synchronize the data transfer on the bus. Since the data transfer is through a shared common bus and there is no direct interconnection between consecutive blocks, the handshake initiation and the data transfer have to be performed differently than in the asynchronous pipeline architecture. Block (n-1) transfers data to block (n) only after block (n) has transferred its own data to block (n+1). Therefore, depending on the decoded instructions, it is more appropriate to assign the bus for a complete handshake interconnection process that includes the handshake initiation as well as the data transfer between a pair of computation blocks. The complete handshake interconnection process propagates backward, opposite to the direction of the data flow. Backward propagation of a complete handshake initiation and data transfer process saves additional handshaking circuitry in each computation block.

A design problem is how to resolve bus contention created by simultaneous access to the bus by different computation blocks. Such simultaneous accesses to a common bus are equivalent to multiple access to an asynchronous arbiter, which can result in a metastable state where the output of the bus is undefined for a random time period. A

sequential order for bus access determined by the decoded instructions and controlled by the control unit avoids this problem. Such sequential operation keeps the execution of the instructions in the order that they were fetched from the memory and allows concurrent operation of the processor blocks.

Sequential access to the bus requires a decision as to how to execute the instructions in the data path. Should the data propagate in a fixed order through all the computation blocks, including those that do not perform any operation on it (as in the pipeline architecture) ? Or should the data propagate in a fixed order only through the computation blocks which need to perform some operation on it (as in the "hybrid" architecture) ? Execution of the instructions by passing the data through all the computation stages in a fixed order is simpler to implement but it operates as a pipeline architecture with reduced throughput. Execution of the instructions in a fixed order but without passing data through unnecessary computation blocks complicates the control unit but utilizes the configuration more efficiently.

The design concepts of the common bus asynchronous architecture requires the control unit to contain a sequencer. The sequencer prevents the arbiter type of problems of multiple access to the bus. According to the decoded instructions, it determines, assigns and controls the sequence and the order of data transfer between the various blocks of the processor. It also insures the execution of the instructions in the exact order that they have been fetched from the memory.

6.3.3. Asynchronous common-bus implementation

Figure 6.14 depicts a block diagram of a processor with an asynchronous bus configuration. All of the computation and control blocks of the architecture are connected in parallel to the bus. The common bus is implemented with a multiplexer (MUX) and a demultiplexer (DEMUX) interconnection blocks. The operation of the multiplexer corresponds to the tri-state output control of the computation blocks in the

synchronous common-bus implementation. And, the operation of the demultiplexer corresponds to the input control of the computation blocks in the synchronous common-bus implementation. The control unit consists of a PLA decoder and a sequencer that control the correct operation and time sharing of the bus, the proper execution of tasks by the computation blocks, and the simultaneous execution of instructions.

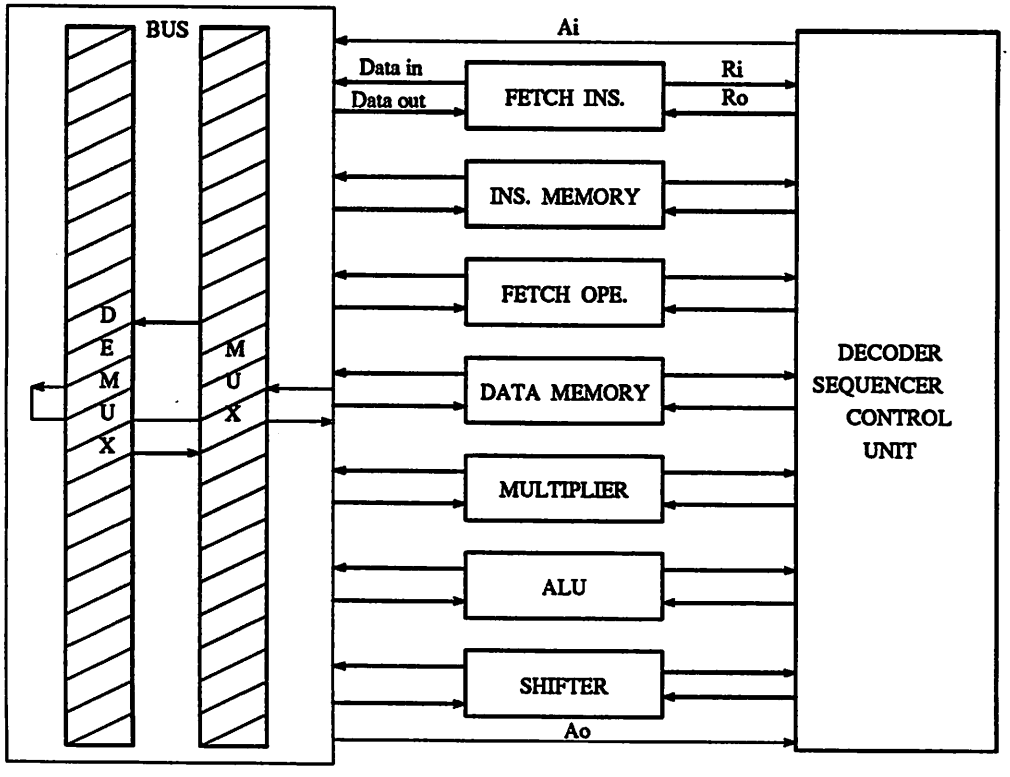


Figure 6.14 - Asynchronous common bus architecture

6.3.3.1. NOFT design approach

A design approach which utilizes "NOFT"s (No operation, data feed through) in the control field when an instruction does not require any operation in the corresponding computation block is similar to the design of the asynchronous pipeline architecture. In this design, there is no contention for resources and the data propagates through

all the computation stages even if no operation is required on it. Propagation of data through the computation stages is executed in the regular predetermined sequence, i.e., "fetch instruction" → "decode" → "fetch operand" → "multiplier" → "ALU" → "shifter". Handshaking procedures (including initiation, data transfer and start of operation in the computation blocks) propagate backward between successive computation blocks. There is no bypass of any computation block. The sequencer of this implementation is a simple modulo 6 counter which counts backward for proper assignment of the bus between two successive computation blocks. Because the sequencer is always counting six handshaking operations, this implementation yields a throughput lower than the asynchronous pipeline implementation but reduces the handshaking overhead circuitry.

6.3.3.2. Bypass design approach

This approach is based upon the principles of the control unit developed for the "hybrid"-modified pipeline architecture. The interconnection block controller of the bus incorporates a sequencer and two interconnection blocks: multiplexer and demultiplexer. Handshaking and data transfer can be performed between any two blocks of the configuration. Use of the bus is granted to the various blocks in a reverse order to the data flow: "shifter" → "ALU" → "multiplier" → "fetch operands" → "decoder" → "fetch instruction". When a resource contention is encountered the bus controller supplies sequential service to the decoded instructions on the basis of first-fetched-first-served (FFFS). As in the "hybrid" modified pipeline configuration, data conflicts might be arise due to the sequential operation when a resource conflict exists. The sequencer is a modulo 6 simple counter that counts backwards for proper assignment of the bus between any two computation blocks. "NOP" is used to decrease the counter and to enable the sequencer to grant the bus for the next block in the reverse order mentioned before.

The following example illustrates the operation of the asynchronous bus architecture. Assume that five instructions: x,y,z,t and v are to be executed. Table 6.9 shown below is a resource allocation table which depicts the sequence of the computations required to execute these instructions.

number	stage	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
1	Fetch inst.	x	y	z	t	v	-	-	-	-
2	Decode	-	x	y	z	t	v	-	-	-
3	Fetch ope.	-	-	x	y	z	t	v	-	-
4	MLT	-	-	-	x	y	-	-	v	-
5	ALU	-	-	-	-	x	y,z	-	-	v
6	Shifter	-	-	-	-	-	x	z,t	-	-

Table 6.9 - Resource allocation table - Bus architecture

Each column denoted by Δ_i indicates the operations to be executed concurrently on different instructions. In this example, the resource allocation table shows that during time slot Δ_6 an ALU resource conflict exists between instructions y and z, and during Δ_7 a shifter resource conflict exists between instructions z and t.

The first two blocks of the configuration: "fetch instruction" and "decode" make up the fetch cycle of an instruction, and therefore are used all the time and cannot be bypassed. The last four blocks of the configuration perform the execute cycle of an instruction. Depending on the instruction, some instructions do not require all computation blocks of the execute cycle and they are bypassed.

From the resource allocation table, it is possible to derive an execution table which shows the sequence of data transfers and computation blocks required for the execution of each instruction. Table 6.10 shown below is the execution table derived from the allocation table for this example.

instruction	Exec. 1	Exec. 2	Exec. 3	Exec. 4
v	2-3	3-4	4-5	
t	2-3	3-6		
z	2-3	3-5	5-6	
y	2-3	3-4	4-5	
x	2-3	3-4	4-5	5-6

Table 6.10 - Execution table - Bus architecture

Each row in the table corresponds to operations required to execute an instruction. The row which corresponds to instruction z shows that its execution requires: (1) data transfer between "decode" and "fetch operand"; (2) execution in the "fetch operand" block; (3) data transfer between "fetch operand" and "ALU"; (4) execution in the "ALU" block; (5) data transfer between "ALU" and "shifter"; (6) execution in the "shifter". Execution of this instruction does not require the multiplier and therefore it is bypassed.

The execution table is like a data-stationary reservation table. Each entry in the table shows the computation blocks involved in the data transfer (source-destination) and the computation block (destination) which has to perform a task on the data. Converting this table into a time-stationary type reservation table provides the data and the sequentiality required for controlling the bus and the proper execution of the instructions. As in the regular pipeline architecture, the conversion is done by an appropriate number of shift registers for each execution operation. The output of this conversion is shown in the timing execution table 6.11 shown below.

Exec. stage	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7	Δ_8	Δ_9
Exec. 1	$(2–3)_x$	$(2–3)_y$	$(2–3)_z$	$(2–3)_t$	$(2–3)_v$		
Exec. 2		$(3–4)_x$	$(3–4)_y$	$(3–5)_z$	$(3–6)_t$	$(3–4)_v$	
Exec. 3			$(4–5)_x$	$(4–5)_y$	$(5–6)_z$		$(4–5)_v$
Exec. 4				$(5–6)_x$			

Table 6.11 - Timing execution table - Bus architecture

Each entry in this table is in the form $(source–destination)_{instruction}$, when the blocks participating in the data transfer are showed and the computation block (destination) to operate on it. Data transfer and start of task execution propagate backward, therefore, the sequence of operations is from Exec. 4 to Exec. 1.

Two columns from this table will illustrate the sequence of operations to be executed when there is or is not a resource conflict.

During time interval Δ_5, there is no resource conflict. Data transfers are executed in serial one after the other but the operations in the computation blocks are executed concurrently as follows:

1) Transfer data from the multiplier to the ALU; start to execute the operation in the ALU required by instruction x.

2) Transfer data from the "fetch operand" stage to the "multiplier" stage; start to execute the operation in the multiplier required by instruction y.

3) Transfer data from the "decoder" stage to the "fetch-operand" stage; start to execute the operation in the "fetch-operand" stage required by instruction z.

During the time interval Δ_6 there is a resource conflict. Data transfer and operations in the computation blocks are executed as before until there is a resource conflict. When a resource conflict occurs, the continuation of the data transfer and the operations in the other computation blocks are delayed until the block of the resource conflict

finishes its task and is ready to receive a new set of data. This delay and the operations involved are as follows:

1) Transfer data from the ALU to the shifter and execute the operation in the shifter required by instruction x.

2) Transfer data from the multiplier to the ALU and execute the operation in the ALU required by instruction y.

3) Transfer data from the "fetch-operand" to the ALU and execute the operation in the ALU required by instruction z.

   Because a resource conflict exists in the ALU, this operation is delayed until the ALU finishes its operation on instruction y

4) Transfer data from the decoder to the "fetch- operand" and execute the operation in the fetch operand required by instruction t.

6.3.3.3. Results

Decoding the instructions into control fields of execution stages that are not dedicated to specific computation blocks of the data path and defining in these control fields the source and the destination of the blocks involved in the data transfer and the block which executes the operation on the data allows us to bypass unnecessary computation blocks and provides a sequential control that solves resource conflicts.

The control unit in the asynchronous common bus architecture is more complicated and requires the capability of sequential operation under certain circumstances of resource conflict. Handshaking circuits between the computation blocks are avoided and data can be transferred between any pair of computation blocks.

Although the data transfer is executed in serial when there are no resource conflicts, the computation blocks operate concurrently and the throughput might be

better than the regular asynchronous pipeline architecture. When there are resource conflicts the throughput might be worse than the regular asynchronous pipeline architecture because of the serial handshaking procedure.

References

1. T.H.Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Automatic synthesis asynchronous circuits from high level specifications," *IEEE ICCAD 87 Digest of Technical Papers*, November 1987.

2. L.G. Heller and W.R. Griffin, "Cascode voltage switch logic: A differential CMOS logic family," *ISSCC Digest of Technical Papers*, February 1984.

3. G. Jacobs and R.W. Brodersen, "Circuit techniques for realization of self-timed DSPs," *To be submitted to IEEE trans. on JSSC.*

4. P.M. Kogge, in *The architecture of pipelined computers*, McGraw-Hill, 1981.

5. T.H.Y. Meng, G.M. Jacobs, R.W. Brodersen, and D.G. Messerschmitt, *Asynchronous processor design for digital signal processing*, January, 1988.

CHAPTER 7

GSLA - Globally Synchronous Locally Asynchronous Processor

7. GSLA - Globally Synchronous Locally Asynchronous

7.1. Introduction

The timing analysis in chapter 5 (table 5.2) shows that the throughput improvement of the asynchronous pipeline architecture over the synchronous one is a function of the clock skew delay and the execution time variations of the stages. Even if the additional delays due to the internal handshaking interconnections within the stages are neglected there are still limiting conditions on the clock skew and the handshaking delays which yield the required improvement. Table 5.2 shows that larger clock skew delays and larger execution time variations yield greater throughput improvements in both the worst and the average cases of the asynchronous implementation. But since most systems operate in a synchronous mode the following questions arise:

1) If the clock skew delay is negligible ($t_{cs} \approx 0$) compared to the execution time of the pipeline stages, is it still possible to obtain an average throughput improvement in the synchronous implementation similar to the asynchronous implementation due to execution time variations ?

2) How would we implement such a pipelined processor ?

A timing analysis shows that it is possible to improve the average throughput of the synchronous implementation. This improvement can be achieved by a processor

that has a global clock with a variable duty cycle, where the clock synchronizes its output to the end of the execution of the stages that have large execution time variations.

7.2. Clock skew delay's reduction methods

Digital signal processing algorithms and most of the existing systems operate on a fixed clock rate. Therefore, in order to increase the processing speed of integrated circuits researchers have dedicated a substantial efforts to developing design and fabrication methods that reduce the clock skew delay. Most of these methods are still in the early stages of research and are not yet utilized in production.

Some methods of reducing the clock skew problems are:

- Careful design and fabrication of clock distribution and local clock buffer.
- Development of special CAD tools for clock paths and distribution design.
- Additional metal layer only for clock distribution and buffer interconnection.
- Clock distribution through waveguide paths.
- Clock distribution through fiberoptic wires with the appropriate transmitters and receivers.
- Reduction of the distance to the dielectric as much as possible by using new materials that attack only where the contact is required in the layers and not the walls themselves.
- New techniques for reducing the area dimension because there is a limit to the reduction of the dielectric distance dimension d [$C=\frac{\varepsilon * Area}{d}$, where C is the capacitance, ε is the dielectric constant, A is the area and d is the dielectric distance].
- Wire fabrication with cuprum instead of aluminum to decrease the r_0 (resistance per meter).

- Clock transfer at lower frequency with accurate frequency multipliers near each block.
- Combination of two technologies on the same chip. A TTL clock driver at each stage when the clock up to the TTL driver is routed through regular CMOS gates. The TTL requires larger area but it has a very good current drive, i.e., lower resistance than the CMOS; therefore, the RC (where C is the load) will be smaller.

It is still early to predict which of these methods will make a breakthrough, but because the clock skew problem limits the throughput when the dimensions are scaled down, it is important to find a solution to this problem. Once a solution is found, using a global clock with variable duty cycle enables to achieve in the synchronous implementation an average throughput similar to that of the asynchronous implementation. Section 7.3 describes two ways to implement a synchronous processor architecture with variable data transfer rates between the stages.

7.3. GSLA implementation

The Globally Synchronous Locally Asynchronous (GSLA) architecture is a pipelined processor architecture that operates synchronously with a fixed high frequency clock (globally synchronous) but the data transfer rate between the stages varies and depends on the variable execution time of the stages (locally asynchronous). Since the execution times vary, this architecture requires input and output queues to synchronize the processor with the data input sample rate. As in the asynchronous implementation, due to variations in the execution time, the GSLA implementation can achieve a higher throughput on average.

The implementation is based on enabling data transfer and initiating the operation of the pipeline stages only after detecting the completion signal(s) of the stage(s) with the large execution time variations ("critical" stage(s)). Detecting the completion signals is synchronized with the high frequency clock. But, data transfer between the

stages and enabling of the next operation is executed at a varying rate according to the execution time of tasks in the "critical" stage(s).

Here are two possible implementation:

1) A high frequency clock rate with fine resolution which detects the completion signal(s) of the "critical" stage(s). The maximum delay of the detection is of one clock cycle which relatively is very small compared to stages propagation delays. Each pipeline stage has a counter which reduces the clock rate to the basic one required for the proper operation of the stages. As depicted in figure 7.1, the control unit of the processor detects the completion signal of the "critical" stages and enables the data transfer between the stages and the start of a new task under the following conditions:

   * The completion signals are detected after the worst case execution time of the "non-critical" stages.

   * At the end of the worst case execution time of the "non- critical" stages if the completion signals were detected before.

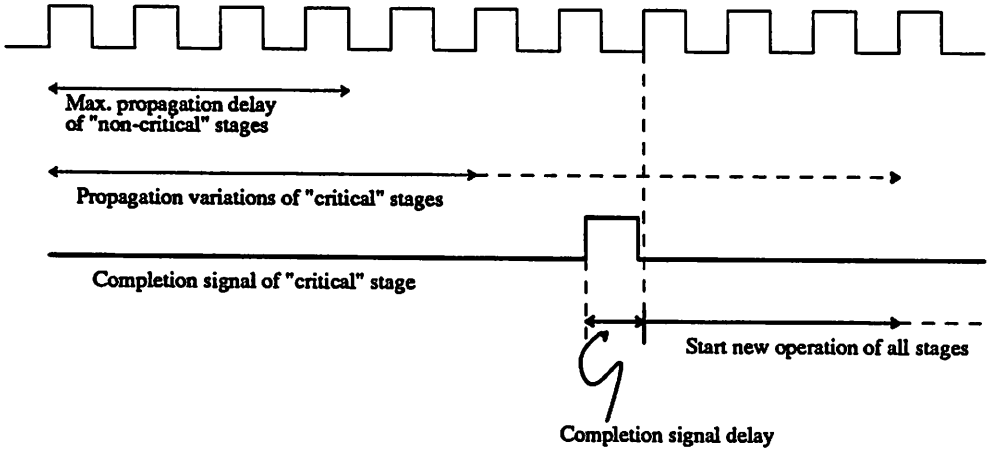


Figure 7.1 - Detection of completion signal by high frequency clock

The relatively small additional delay for detecting the completion signal has small

effect on the cycle time thus yielding approximately the same average clock rate as for the asynchronous architecture.

2) Controlling the various pipeline stages with the appropriate clock rate frequency and enabling the data transfer and a new task execution upon the detection of the completion signal(s) of the "critical" stage(s). Since the clock frequency is adequate for proper operation, the stages do not have to reduce it with a counter. Each pipeline stage has an input which enables data transfer to the next stage and initiates execution of the next task. The control unit of the processor detects the completion signal(s) of the "critical" stages and activates the enable line with the right timing. Assuming that the stages start to execute their tasks during the high level of the clock, as depicted in figure 7.2 two cases have to be considered regarding the activation of the enable control line:

   * If the detection of the completion signal occurs during the high level of the clock, the control unit activates the enable control line and expands the high level by an additional half clock cycle to provide an appropriate timing for immediate operation.

   * If the detection of the completion signal occurs during the low level of the clock, the control unit activates the enable control line on the following high level of the clock cycle thereby minimizing the idle time before executing the next operation.

As before, the activation depends on the time the completion signal is detected relative to the worst case execution time of the "non-critical" stages.

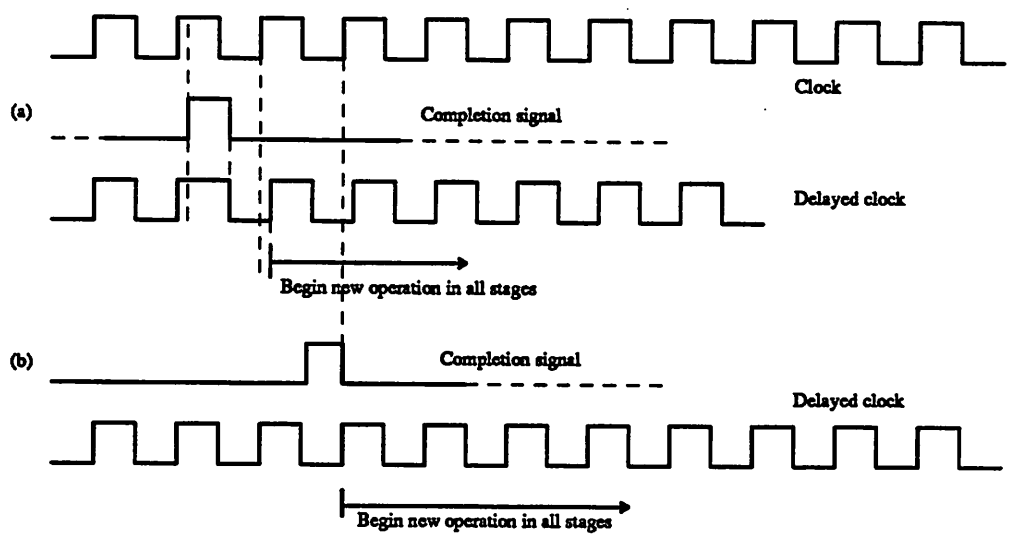


Figure 7.2 - Variable duty cycle of processor's clock

In both schemes, the completion signal circuitry of all stages except the "critical" one is also avoided. The difficulties of asynchronous circuitry design and the hand-shaking circuitry and delays overhead are also avoided. Additional circuitry to detect the completion signal(s) and to control the clock is required. Bypassing the "critical" stage(s) when it is not necessary to pass through them can increase the throughput but will require additional control for initiating the next cycle of the pipeline stages.

7.4. GSLA timing analysis

Analyzing and comparing the cycle time of the GSLA architecture is based on the same additive timing models, definitions and notations of chapter 5. Assuming that the latch delay is embedded in the execution time of the stage yields the following cycle time results:

Synchronous pipeline architecture cycle time:

$$T_{sy}=t+t_{cs} \tag{7.1}$$

Asynchronous pipeline architecture average cycle time:

$$T_{avg.asy}=\frac{t+t_1}{2}+3t_{hs}+t_l=\frac{1+k}{2}t+3t_{hs}+t_l\approx\frac{1+k}{2}t \tag{7.2}$$

GSLA pipeline architecture cycle time:

$$T_{GSLA}=\frac{t+t_1}{2}+t_{cs}=\frac{1+k}{2}t+t_{cs} \tag{7.3}$$

Writting the ratio between $T_{avg.asy}$ and T_{sy} yields the average throughput improvement factor q of the asynchronous architecture compared to the synchronous architecture,

$$\frac{T_{avg.asy}}{T_{sy}}=\frac{\frac{(1+k)t}{2}}{t+t_{cs}}=1-q \tag{7.4}$$

And the approximate upper bound to throughput improvement is:

$$q_{avg.asy}\approx\frac{\frac{1}{2}(1-k)t+t_{cs}}{t+t_{cs}}=\frac{\frac{1}{2}(1-k)t}{t+t_{cs}}+\frac{t_{cs}}{t+t_{cs}} \tag{7.5}$$

The results of the average throughput improvement factor of the asynchronous implementation are depicted in the following table:

t_{cs}	q [%]				
	$k=0$	$k=\frac{1}{4}$	$k=\frac{1}{2}$	$k=\frac{3}{4}$	$k=1$
	Max variations, 100% variations in t	75% variations in t	50% variations in t	25% variations in t	Worst case, no variations in t
0.00t	50%	37.5%	25%	12.5%	0%
0.11t	55%	43.7%	32.5%	21%	10%
0.25t	60%	50%	40%	30%	20%
0.43t	65%	56.3%	47.5%	38.8%	30%
0.66t	70%	62.3%	55%	47.3%	40%
1.00t	75%	68.7%	62.5%	56.2%	50%

Table 7.1 - Asynchronous architecture - average throughput improvement

In the same way, writting the ratio between T_{GSLA} and T_{sy} yields the throughput improvement factor q of the GSLA architecture compared to the synchronous architecture,

$$\frac{T_{GSLA}}{T_{sy}}=\frac{\frac{(1+k)t}{2}+t_{cs}}{t+t_{cs}}=1-q \tag{7.6}$$

And the approximate upper bound to throughput improvement is:

$$q_{GSLA}\approx\frac{\frac{1}{2}(1-k)t}{t+t_{cs}} \tag{7.7}$$

The results of the throughput improvement factor of the the GSLA implementation are depicted in the following table:

t_{cs}	q [%]				
	$k=0$	$k=\frac{1}{4}$	$k=\frac{1}{2}$	$k=\frac{3}{4}$	$k=1$
	Max variations, 100% variations in t	75% variations in t	50% variations in t	25% variations in t	Worst case, no variations in t
0.00t	50%	37.5%	25%	12.5%	0%
0.11t	45%	33.8%	22.5%	11.2%	0%
0.25t	40%	30%	20%	10%	0%
0.43t	35%	26.2%	17.5%	8.7%	0%
0.66t	30%	22.5%	15%	7.5%	0%
1.00t	25%	18.7%	12.5%	6.2%	0%

Table 7.2 - GSLA -throughput improvement factor

Results

The following results are based upon the assumption that the handshake delay in the asynchronous architecture is small and negligible compared to the execution time and

the clock skew delay. When the pipeline architecture is deeper, the clock skew delay has a larger effect on the analysis and is not negligible. The clock skew delay is more likely to be negligible if the architecture incorporates a small number of pipeline stage (corresponds to larger execution time of the stages). It is obvious that the GSLA architecture is effective only if the clock skew delay is negligible.

- The maximum bound on the throughput improvement factor q_{GSLA} of the GSLA architecture is 50% (for 100% execution time variations).
- As the clock skew delay (t_{cs}) increases the throughput improvement factor q_{GSLA} of the GSLA architecture decreases.
- The throughput improvement factor of the average asynchronous architecture implementation is larger than the throughput improvement factor of the GSLA architecture by a term of $\frac{t_{cs}}{t+t_{cs}}$.
- As the clock skew delay (t_{cs}) increases, the average throughput improvement factor $q_{avg.asy}$ of the asynchronous architecture increases.
- The throughput improvement factor of the GSLA architecture is the lower bound of the average throughput improvement factor achieved by the asynchronous architecture when the there is no clock skew.
- When the clock skew delay is negligible ($t_{cs} \approx 0$) GSLA architecture yields the same throughput improvement factor as the asynchronous architecture yields in the average case ($q_{GSLA} = q_{avg.asy}$).
- As the clock skew delay (t_{cs}) becomes larger the average throughput improvement factor of the asynchronous architecture becomes larger compared to the GSLA architecture.

7.5. Conclusions

- The GSLA architecture yields a throughput improvement factor as the average throughput improvement factor of the asynchronous architecture only if the clock skew delay is negligible.
- The maximum bound on the throughput improvement factor q_{GSLA} of the GSLA architecture is 50% (for 100% execution time variations).
- As the clock skew delay (t_{cs}) increases, the throughput improvement factor q_{GSLA} of the GSLA architecture decreases. Larger clock skew delay mean smaller throughput improvement.
- Larger clock skew delay (t_{cs}) means a larger average throughput improvement of the asynchronous architecture compared to the GSLA architecture.
- Implementation of the GSLA architecture is feasible.

CHAPTER 8

Conclusions

8. Conclusions

Implementation of processing elements and systems for real time signal processing applications require 1) a fast processing elements with higher throughput and 2) efficient schedulers to partition the algorithms into different tasks and allocate them appropriately to the processors. To achieve fast processing elements with higher throughput for real time signal processing applications, one has to utilize the advances of μP VLSI design and fabrication with the special features and characteristics of the signal processing algorithms. Development of improved CAD tools and fabrication processes makes it feasible to implement the PEs proposed in this dissertation.

8.1. Multiprocessing PE

Exploitation of the parallelism inherent in digital signal processing implies the building of computing systems which rely on relatively inexpensive processing elements that operate in parallel. Parallel computing systems that are not restricted by the PE's interconnection topologies and have low communication latencies achieve high performance and throughput.

An advance in μP VLSI design and fabrication makes it feasible to implement the proposed PE. This will reduce the communication latencies, allow any interconnection topology consistent with the number of ports, and increase the multiprocessing system throughput.

Communication latencies are reduced because of:

1) Independent and concurrent computation and communication achieved by separating the computation unit (PU) from the communication unit (AIO).

2) Fast data transfer through virtual-cut-through switching, minimum number of hops, and the variable interconnection band-width between processing elements.

3) Simple interface (through the AIO) between the processing unit (PU) and the network.

The proposed PE is suitable for ASIC (Application Specific IC) implementations. Similar and simple protocols for different communication configurations with the ASIC property adapt and fit the proposed PE into a variety of different multiprocessor systems and applications. Each PE can accommodate a different computing unit and a different communication configuration suitable for a particular application. "Heterogenous" systems are simple to implement because different PEs can accommodate different types of processing units and/or different communication configurations.

Four I/O links enable the PE to be employed in many network topologies and provide a simple expansion to large multiprocessor configurations.

Interconnection band-width (BW) is extensible because the number of lines in an I/O link are parametrizable and up to four I/O links can interconnect two PEs.

8.2. Asynchronous PE

It is clear that an asynchronous PE is preferable to a synchronous PE only if the clock skew delay is relatively large compared to the maximum execution time of the functional units of the processor. Good circuits synthesis programs as well as appropriate CAD tools are required for such designs. In general since most of the systems are synchronous, a synchronous processor is preferable. Development of new methods and

CAD tools for clock path and distribution design along with careful and tightly controlled fabrication of the clock distribution and the local clock buffers reduces the clock skew delay effects and favors the synchronous implementation.

As previously seen from the timing analysis, an implementation of an asynchronous processing element yields a higher throughput than the synchronous implementation only if several conditions on the clock skew delay (t_{cs}) and the handshaking delay (t_{hs}) are fulfilled (equations: 5.4, 5.7, 5.8, 5.12, 5.13).

The maximum throughput improvement of the asynchronous implementation compared to the synchronous implementation is limited. To achieve in the worst case a higher throughput compared to the synchronous implementation, a deep pipeline architecture is required, i.e., an architecture with more pipeline stages which corresponds to smaller maximum execution time (max.$\{t_{sd}\}$ is smaller).

The larger the variations in the execution time of the pipeline stages, the larger is the average throughput improvement of the asynchronous implementation. Achieving a higher average throughput improvement requires input and output queue buffers. The length of the queue buffers limits the type of applications that can be executed on such architecture.

A time-stationary control unit for the asynchronous processor can be simply implemented with a PLA and shift registers. Such a control unit does not add additional handshaking delays, and it can discard instructions from the data path stages.

When the clock skew delay is negligible the average throughput of the asynchronous implementation is smaller than 50%.

When the throughput limitation is due to large propagation delays variations of the pipeline stage(s) and not due to clock skew delay (clock skew delay is negligible), the GSLA (globally synchronous locally asynchronous) implementation achieves the same average throughput as the asynchronous implementation. When the clock skew delay is

not negligible the GSLA throughput decreases as the clock skew delay increases.

The larger the clock skew delay, the larger is the average throughput improvement of the asynchronous implementation compared to the synchronous pipeline architecture and the GSLA architecture.

Asynchronous processing elements can also be implemented in other architectures such as "Hybrid" pipeline and common bus. Their throughput depend highly on whether resource conflicts exist or not. If there are no resource conflicts, an asynchronous common bus architecture achieves a higher throughput than the asynchronous pipeline architecture. The control unit of the asynchronous "Hybrid" and the common bus architectures is more complicated.

8.3. Further Research

Further research is necessary in developing new techniques, CAD tools and simulation programs for efficient design of high speed synchronous and asynchronous processing elements and processing systems for real time applications.

Utilization of the parallelism inherent in signal processing algorithms is very important in implementing processing systems for real time applications. There are two ways to do it. One is to develop new signal processing algorithms customized for parallel processing systems. The other is to develop schedulers that partition the algorithms and effectively allocate the tasks to different PEs. The scheduler should have the capability to optimize the allocations according to the following requirements: high throughput, minimum number of PEs, load balanced PEs, minimal number of interconnection, and a given interconnection configuration.

Fabrication of the multiprocessor PE, containing a PU and an AIO, is necessary to evaluate the IC's area, speed of operation, complexity, and the difficulties involved in the design and implementation of such large VLSI chips. The implementation of the VCT switching and its efficiency is important and necessary to evaluate.

Complete design and development of new interconnection handshaking cells are mandatory for the implementation of an asynchronous PE for different signal processing applications.

Development of a simulation program to determine the exact length of I/O queue buffers for obtaining a higher average throughput is needed.

An appropriate interface between an asynchronous processor and synchronous systems is required. The design of such interface should efficiently utilize the properties of both systems.

The development of new techniques and CAD tools for designing high speed synchronous PEs is needed. Design and implementation of a GLSA processor with special emphasis on the design of clock buffers and distribution could be used as a test case.