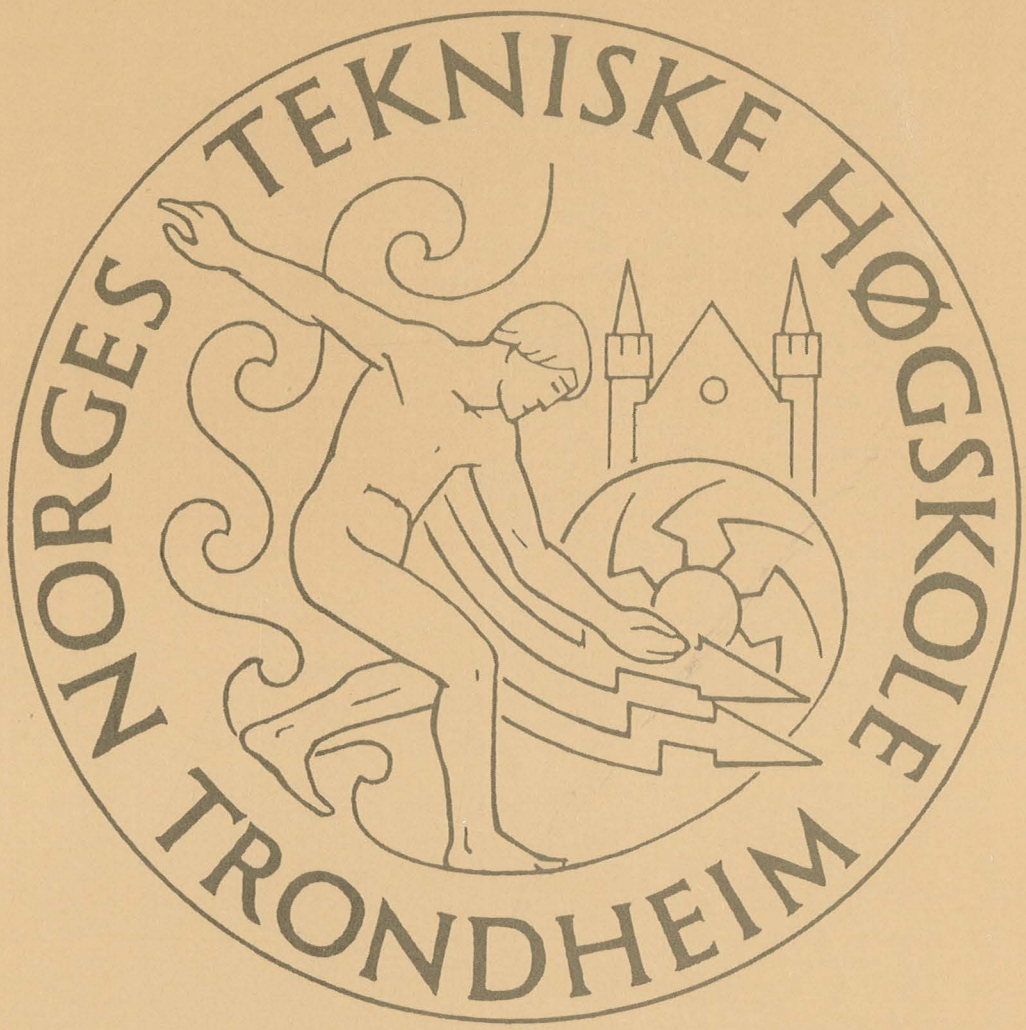
OLA LØKBERG

PROCESSING STRUCTURES FOR REAL TIME ANALYSIS OF MEDICAL ULTRASOUND IMAGES



NTH
UNIVERSITETET I TRONDHEIM
NORGES TEKNISKE HØGSKOLE

DOKTOR INGENIØRAVHANDLING 1992:4
INSTITUTT FOR TEKNISK KYBERNETIKK
TRONDHEIM

ITK-rapport 1992:4-W

Processing structures for real-time processing of medical ultra-sound images

by

Ola Løkberg

A thesis submitted for the degree of

Dr.ing.

University of Trondheim

Norwegian Institute of Technology

Division of Engineering Cybernetics

Trondheim, January 1992

NTH-TRYKK
1992

Preface

This thesis constitutes partial fulfilment of the requirements for the Dr.ing degree. The work presented here can in fact be said to have started as early as in 1984, with the "CFM-700" project. The task of that project was to develop a 2-dimensional colour flow mapping instrument for medical ultrasound diagnostics. Although Vingmed Sound A/S, Horten, as the project's industrial partner provided the financial support, the work was initiated and supervised by the academic supervisor for this thesis, Prof. Bjørn Angelsen. During the next 4 years, up to 7 engineers and scientists (myself included) at SINTEF Automatic Control were more or less engaged in the project, in close cooperation with Vingmed Sound's own engineers. Today, the CFM-700 series of instruments are still serving the market as high-quality medical ultrasound diagnostic equipment with cardiology as their prime area of application.

However, although there are still years left of their product lifetime, it is nevertheless clear that the present architecture will be unable to support the requirements of tomorrow. With the steady emerging demands for higher image resolution, multiple image display, image quality enhancement and advanced post processing features, a new architecture must be developed and this thesis will hopefully be a contribution to that work. Even with the CFM-700 as an inevitable professional background for the work carried out in this thesis, the instrument architecture as it is presented here have very little in common with the previous system. Exceptions to this are the data acquisition part of the system and of course, the application itself.

Due to several reasons, the submission of this thesis for the Dr.ing degree is the end of a rather long story. I am therefore greatly indebted to those who made it possible for me to keep on and complete my work. First of all the Norwegian Institute of Technology (NTH) and the Royal Norwegian Council for Scientific and Industrial Research (NTNF) for their scholarship and financial support, but not least my employer, SINTEF Automatic Control, whose support and encouragement has been very much appreciated. Further, thanks to my advisors, Prof. Odd Pettersen, Prof. Bjørn Angelsen and Prof. Kjell Malvig, for their support and help, and also to my colleagues at SINTEF for all fruitful discussions during the work with this thesis. Last, but by no means least, I would like to give a special thanks to Kjell Kristoffersen and Hans Christian Lønstad at Vingmed Sound for their many valuable comments.

Summary and conclusions

This thesis describes a processing structure, or system, aimed at processing medical ultrasound images in *real time*. The term "real time" in this context means that the throughput of the total system should be limited only by the rate by which it is possible to acquire ultrasound data, and not by the time required for the subsequent processing or display of those data. Although the processing structure presented is specified and designed from a medical ultrasound diagnostics point of view, its scope of application should be much broader than that. All processing jobs which can be partitioned into a set of more or less independent, pipelined sub-tasks should be able to run efficiently within the framework of the presented architecture. Comprehensive signal and image processing jobs are typical applications falling into this category.

The first part of the thesis (chapters 1 to 3) provides the necessary background as far as *ultrasound physics and diagnostics* are concerned. This includes the nature of the data involved in terms of type and amount, the type of processing capabilities required, necessary display features to provide a user interface and a discussion of the clinical environment in which the instrument is to be used. Due to the rapid development and implementations of Local Area Networks (LANs), emerging standards for medical (picture) data archiving and retrieval coupled to the increasing demand as far as hospital cost reduction and efficiency are concerned, this environment is not likely to be the same in ten years as it is today. One aspect of this may be that the data acquisition and the data presentation/ interpretation phases of an ultrasound examination will be separated to a much larger extent than it is today. This should therefore also be reflected in the system's architecture and capabilities.

The outcome of this discussion (chapter 4) is a number of vital issues which should be addressed in developing a real-time medical ultrasound processing structure. From the amount of data involved, the rate by which they arrive and the type of processing necessary, it is immediately apparent that some sort of parallel processing is required. During the last decade, numerous architectures for this kind of processing has been presented. To be able to determine whether any existing architecture (or architecture feature) could be adapted to the particular application discussed in this thesis, it would be advantageous to be able to systematically classify parallel architectures according to some set of defined characteristics. To do this, some formal classification tool is needed. Several *computer taxonomies* are therefore discussed (chapter 5), with the purpose of finding one suitable for selecting an architecture for a given application having certain characteristics. A new taxonomy is then introduced based on a classification scheme introduced by S. Yalamanchili.

According to this taxonomy and in view of the requirements of the application in question, several parallel architectures are discussed and compared (chapter 6). The result of this discussion is a *specification outline* (chapter 7), presenting a coarse sketch of a processing structure for real -time acquisition and display of medical ultrasound images. Key features are:

- MIMD (Multiple Instruction, Multiple Data) organization.
- Modules grouped into clusters, each cluster consisting of one controller and a number of Processing Elements (PEs).

- Local communication within each cluster is performed by a bidirectional ring bus.
- Module synchronization and control is implemented according to a data-driven, macro data-flow principle, making the availability of data and appropriate processing elements determining for when a computation is to take place.
- Global buffering facility managed by the controller within each cluster, thereby relieving a sending module of the responsibility of keeping and taking care of data intended for a (temporarily) busy destination module.
- Logically, all modules are interconnected as a dynamically reconfigurable, logical pipeline, supporting context dependent reconfiguration, load balancing as well as iterative processing.

To ensure a consistent use of language in describing the developed processing structure, a *nomenclature* is included (appendix C), covering all vital system and ring bus related terms and expressions.

The specification outline is then refined into a more detailed *Ring bus specification* (chapter 8), presenting the entire set of signal lines and the procedures by which information are transferred over those lines. These procedures include different data transfer modalities, cluster control operations like ring bus arbitration procedures as well as emergency message transfers. Procedures for inter-cluster communication are also described. Packet templates for transferring data and control information as well as detailed timing diagrams are presented. The disadvantage of long latency times commonly associated with packet based communication systems is avoided as far as control packet transfers are concerned, by providing a special ring bus access mechanism for those kind of packets. Data transfer timing is based on a two-phase, source-synchronous non-compelled protocol as found in Futurebus+. Other protocol features have also been inspired by features supported by already existing backplane buses. A comprehensive survey presenting the key characteristics of a number of modern digital buses (Futurebus+, SBus, Turbochannel, NuBus, Multibus II, VMEbus etc.) can be found in appendix.

The processing structure developed in this thesis is presented from a system point of view, discussing vital topics as interconnection structures and communication schemes. The detailed design and function of the different modules are in this context therefore not important, what matters are the role they play within the framework of the total system. One exception, however, to this is made: *The ring bus arbiter*. The reason for this is the arbiter's vital importance to correct system operation as well as performance. A detailed gate-level arbiter design is therefore presented (chapter 9), showing that an implementation according to the functional specification is possible in a FPGA (Field Programmable Gate Array) sized type of integrated circuit.

Finally, the architecture of a *display system* supporting real-time presentation of the ultrasound data to the user is developed (chapters 10 and 11). In accordance with the discussion carried out in chapter 3, the proposed display system is intended for data presentation during data acquisition rather than for data post processing interpretation. Display system key features are:

- Object-oriented implementation facilitating good display dynamics.
- Medium to high display screen resolution (in the range of 512 by 512 up to 1024 by 1024 pixels).
- Multiple window capability supporting up to 4 simultaneous, overlapping image windows with independent graphics and look-up tables for each window.
- Fast response to user interactions (one display frame time when moving windows in side and depth).
- Display parameters broadcast on pixel bus during screen retrace periods offering simultaneous update of all display modules automatically synchronized to screen output.

Literally speaking, the term "ultrasound" is used for sound frequencies above the audible range of 15 - 20 khz. For medical applications, most ultrasound transducers operate in the interval of 2 to 10 Mhz. The actual frequency being used is selected based on a trade-off between high resolution, requiring a high frequency, and high (deep) penetration into the body, obtained with a low frequency transducer.

A modern ultrasound system supports a number of medical diagnostic techniques, all of them based on the same basic principle:

> *The emission of ultrasound energy into the body for the purpose of observing the reflected signal.*

It is then up to the system to extract the necessary parameters from the reflected signal to generate or compute the information we are seeking. If an image of the internal organs is what we are aiming at, it is the amplitude of the reflected signal which are of interest. For the measurement of some sort of velocity or blood flow, the necessary information is hidden in the frequency and phase of the reflected signal.

1.1. Scope of work

Ultrasound diagnostics has many applications in today's clinical environment. Although based on the same basic principle, ultrasound systems are, to some extent, customized to their actual application. This is especially so as far as the data acquisition part of the system is concerned. Examples of such application specific parameters are the transducer type and frequency, and the type of analog and digital frontend filtering. These variants have, however, no impact on the system architecture as such, which will be the subject of this thesis. To avoid dealing with too many terms and phrases describing various ultrasound applications, only ultrasound imaging of the heart and its associated blood vessels will be considered. Such systems are called *cardiologic* ultrasound systems, and will be synonymous with the term "ultrasound system" throughout this text.

The reason for concentrating on cardiologic imaging, is that this is the application of medical ultrasound imaging offering the greatest challenges, seen from an image processing as well as a system architectural point of view. A system solution capable of handling real-time cardiologic imaging is applicable for other types of medical ultrasound diagnostics as well.

Ultrasound imaging of the heart and its related blood vessels is called *echocardiography*. A real-time, 2D echocardiographic image is, depending on the type of system, assembled by two main components: The *tissue image*, showing the moving cardiac (heart muscle) wall and valve structures and the *flow image*, superposing an image of the flowing blood on top of the tissue image.

1.2. The tissue image

The primary component in the echocardiographic image is the tissue image. Without the tissue image, there would be no way of relating the 2D flow image to the actual valves and vessels transporting this flow. On the other hand, a 2D tissue image has its own clinical value, it is not depending on an accompanying flow image.

The tissue image is generated by firing pulses from a piezo-electric transducer element. The emitted pulse will penetrate into the body, being reflected by the interface between organs and by the structure within each organ. By knowing the speed by which the pulse travels (approximately 1500 meter/second) and measuring the time elapsed from the pulse is fired until a reflection is detected, the depth of the reflecting object can be calculated.

1.2.1. 2D tissue imaging

2D tissue imaging is obtained by scanning the ultrasound beam over an image field. This scanning can either be *mechanical*, performed by a physical rotation of the transducer element, or *electronic*. Electronic scanning means that the direction of the ultrasound beam is steered electronically instead of physically moving the transducer element. This is the technique used by various kinds of electronic array transducers.

For each emitted ultrasound beam (pulse), the amplitude of the reflected signal is sampled, the number of samples per beam being dependent of parameters as the transducer frequency (image resolution) and the total sampling depth. The samples are then coded into grayscale values, to be displayed along the direction of each beam. After scanning the entire image field, the sampled data from all beam directions are put together, forming a 2-dimensional tissue image.

The most important parameter in tissue imaging is perhaps the transducer frequency, determining image resolution. The resolution can be said to have two components, radial and lateral. The *radial* resolution (along the ultrasound beam) is a function of the minimum pulse length, typically a couple of oscillations long. Increasing the frequency shortens the pulse length and the radial resolution is thereby increased. As far as the *lateral* resolution is concerned, it is a function of the width of the transducer beam, which is inversely proportional to the frequency when the diameter of the transducer aperture is kept constant. A higher frequency therefore means increased lateral resolution as well.

Therefore, the higher the transducer frequency is, the better the imaging resolution will be, radially as well as laterally. However, the attenuation of ultrasound energy within the body also increases with the transducer frequency. The penetration is then reduced, thereby reducing the maximum imaging depth. To go deep into the body, a low frequency transducer is required. Therefore,

> *choosing transducer frequency is a trade-off between image resolution, requiring a high frequency transducer, and sampling depth, increased by a low frequency transducer.*

The actual frequency being used will therefore depend on the application. For adult cardiology, 2.5 to 5 Mhz transducers are used while 5 to 7.5 Mhz transducers are best suited to pediatric (child) cardiology, due to the reduced depth requirements. Imaging peripheral vessels located under the skin requires high resolution imaging without the need for deep sampling, a high frequency transducer (7.5 to 10 Mhz) is therefore the obvious choice in that case. On the more extreme side, up to 40 Mhz transducers have been used for intra-arterial imaging of atherosclerosis.

1.2.2. Imaging history

According to [Angelsen 1990], the first application of medical ultrasound was to identify organ structures like the mitral valve. The acquired data were displayed in a mode called *A-mode* (Amplitude), showing a momentary, oscilloscope-like image of data (Y-axis) as a function of time (X-axis). All data were acquired along the same direction.

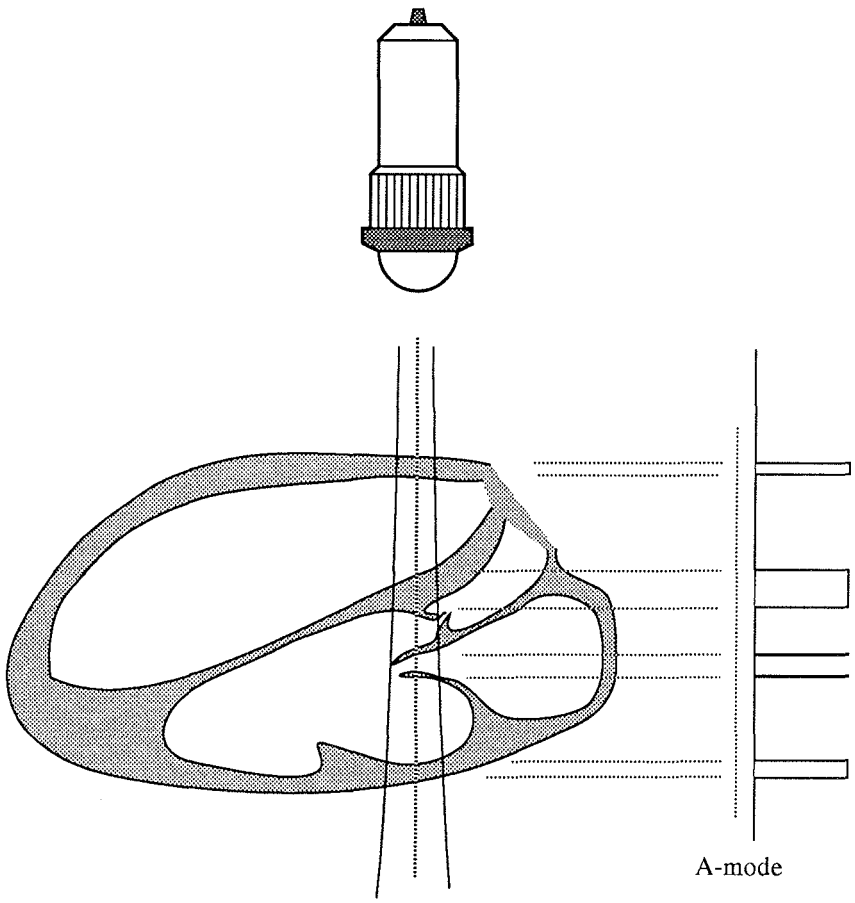


Figure 1.1. A-mode

To obtain 2-dimensional data, it was necessary to do some sort of scanning. Manual scanning, done by hand, of the transducer was then introduced. Due to the limitation as far as scanning speed was concerned, this was obviously only applicable for imaging non-moving organs. To be able to image moving structures of the heart, the *M-mode* (Motion) was then invented: The mechanical problems in conjunction with an automatically scanning transducer in this mode were avoided by replacing the spatially second dimension by time. While the transducer was kept still, the acquired data were displayed along the screen's vertical edge as gray-level coded values as a function of depth (i.e. time elapsed from the emission of the pulse until the return of the echo). Then by letting the image slide to the right while filling in new values along the screen left edge, moving structures like cardiac walls and valves could be shown in a way easily interpretable to the system user. The image height was equal to the maximum depth, while the image width was equal to the total time span between the echo shown along the left edge on the screen and the echo shown along the right edge of the screen.

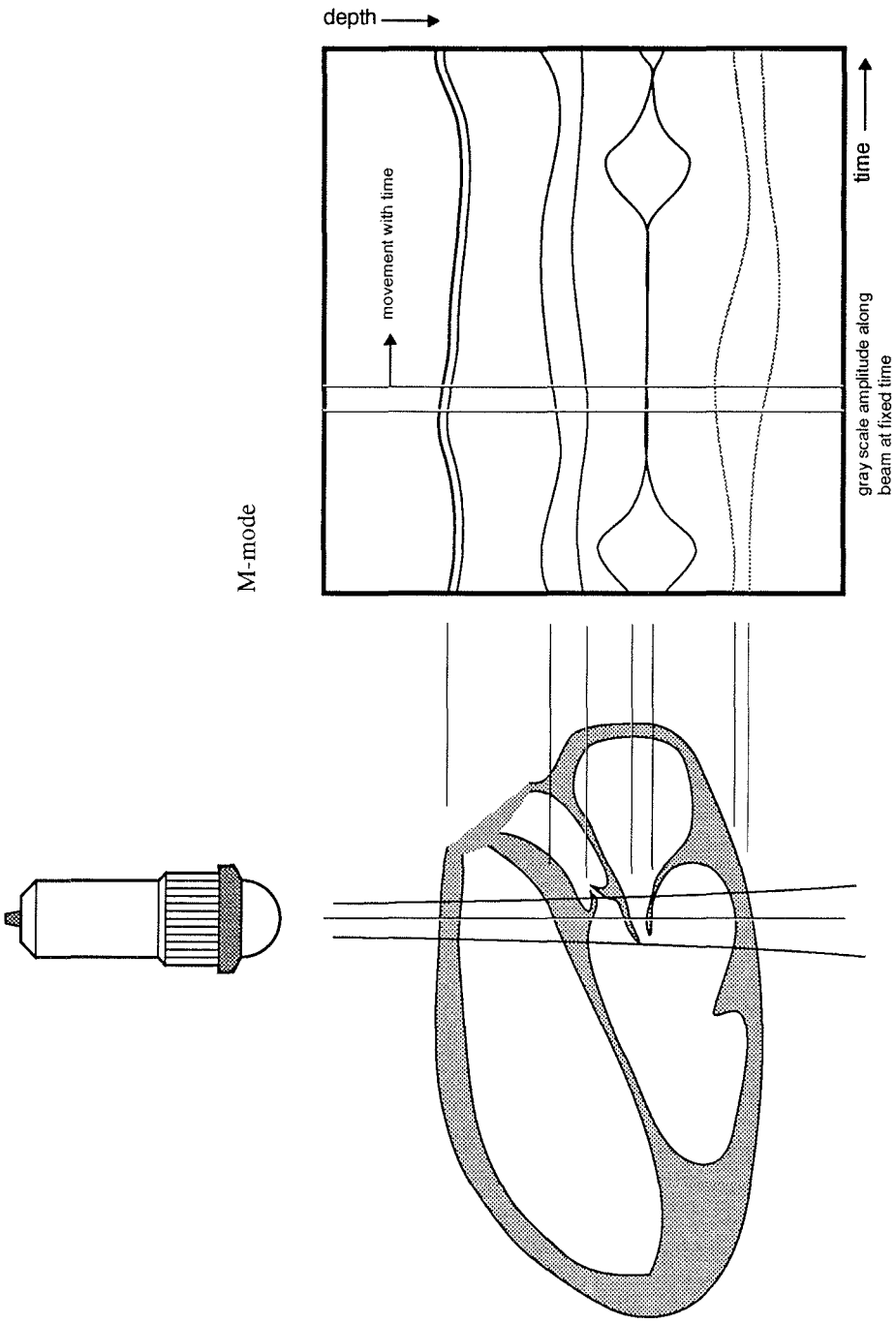


Figure 1.2. M-mode

To get a true 2 -dimensional (2D) image of a moving organ, however, showing the actual size and shapes of its components, there was no way around an automatically scanning transducer. Only in this way could an exact spatially relationship from beam to beam be ensured, a necessity for reconstructing a geometrically correct image. This mode is also known as *B-scan* (Brightness). The first systems used fast mechanical scanning transducers, later electronic steering of the beam was introduced. A 2D image is in this mode generated by letting the transducer scan the area of interest, displaying the acquired echo data as in M-mode, resulting in a sector-shaped, gray-level image.

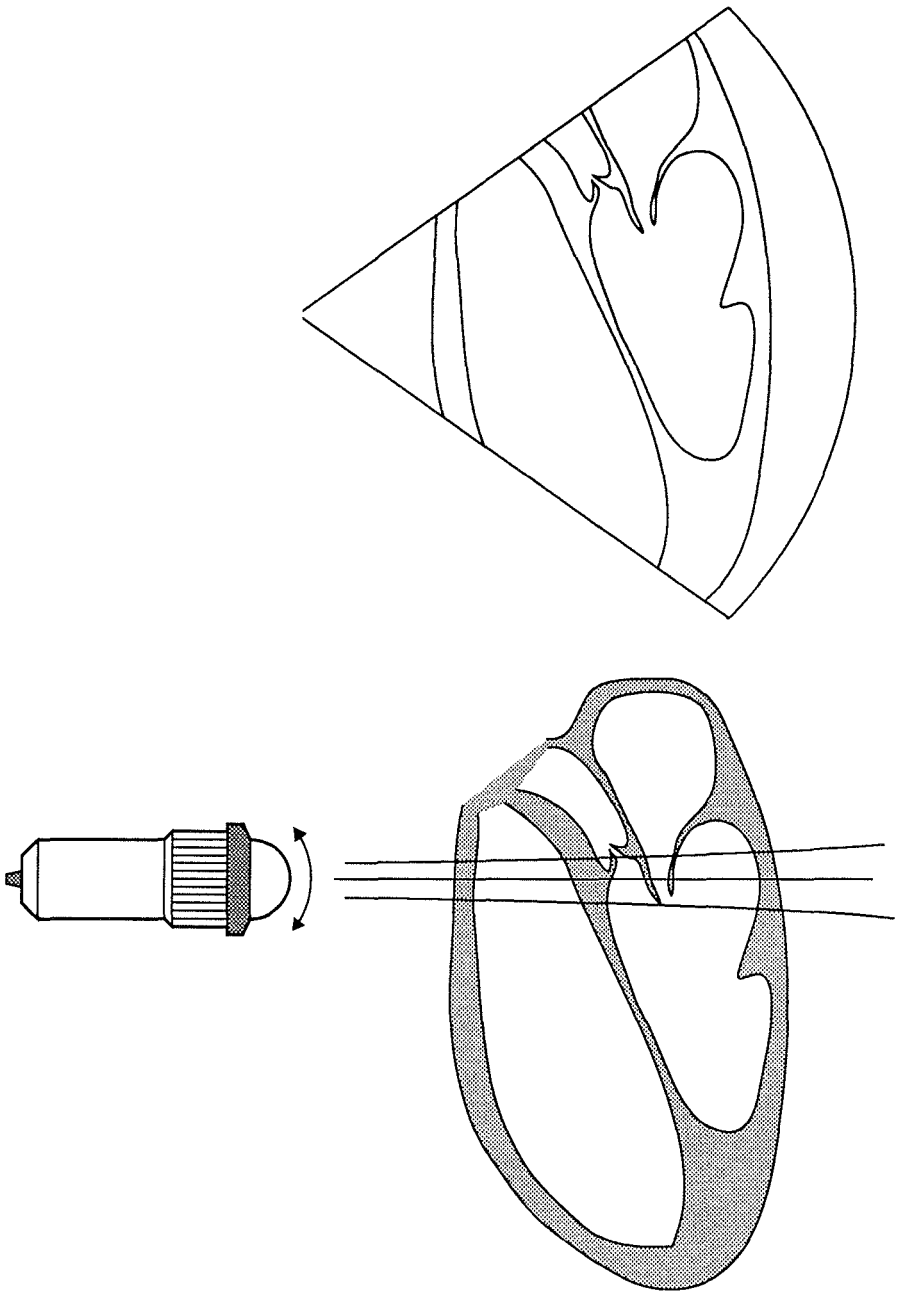


Figure 1.3. 2D imaging (mechanical transducer

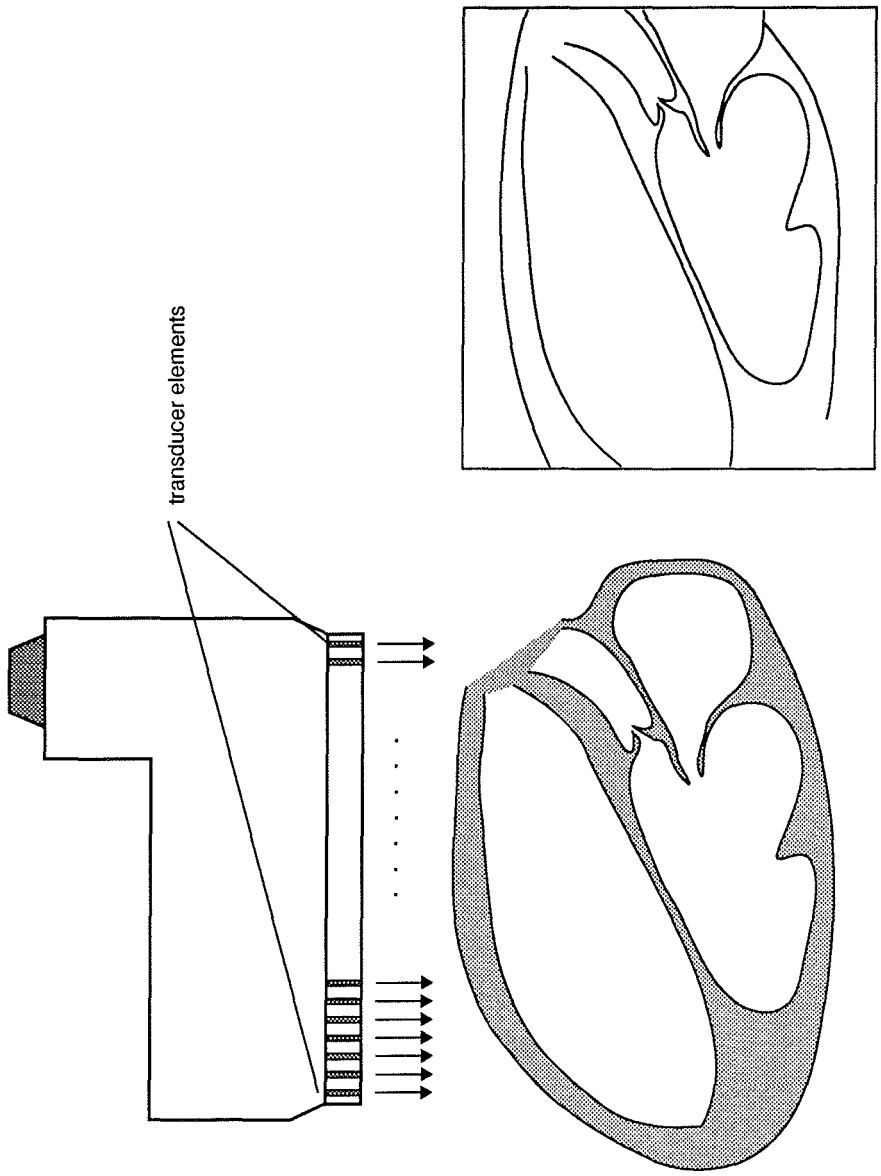


Figure 1.4. 2D imaging (linear array electronic transducer)

1.3. The flow image

An echo image does not, however, provide complete functional information as far as the heart is concerned. Although showing the heart muscle in motion, one vital piece of information is missing, namely information on how the blood is flowing in the heart as it moves. This flow (i.e. the velocities of the blood elements) can be measured in the same way we are measuring the speed of a moving car: By utilizing the Doppler effect.

1.3.1. Doppler techniques

If a sound of source and a person listening to that source are moving relative to each other, the sound frequency perceived by that person will depend on the direction and speed by which they move: If the person and the source are moving towards each other, he/she will hear a frequency higher than the actual frequency. If they are moving away from each other, the frequency will be lower. The difference in frequency between the emitted and the perceived sound is proportional to the speed component along the direction between the source and the listener. By comparing the perceived frequency to the actual (emitted) frequency, the relative movement between the sound source and the observer of that source can therefore be determined. This phenomenon was first observed by the Australian scientist Christian Johann Doppler (1803-1853) and it was therefore named the *Doppler effect.* This effect is subject to all types of waves where the source and the receiver are moving relative to each other.

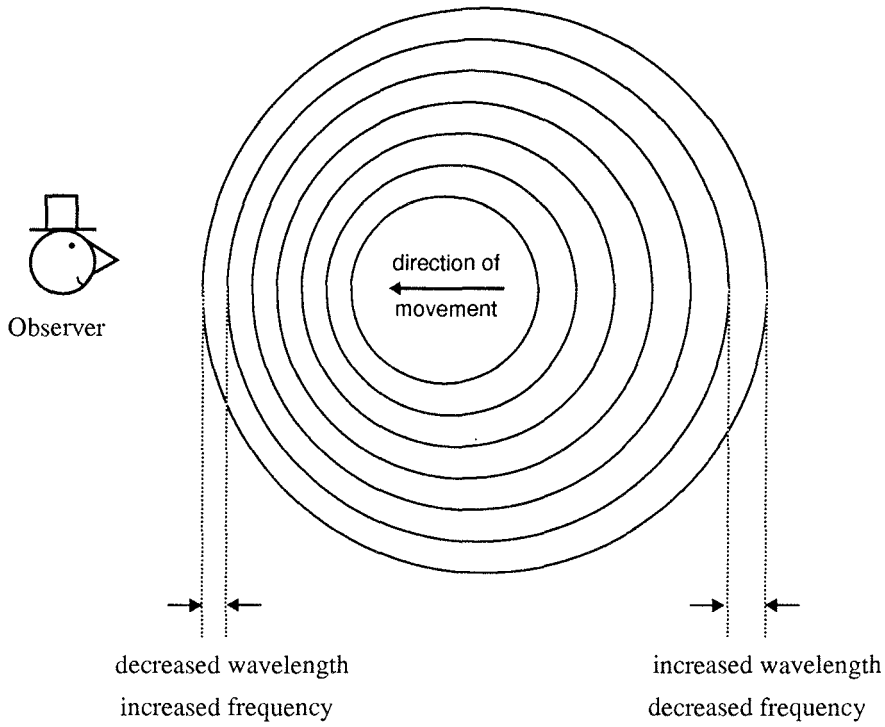


Figure 1.5. Doppler effect principle

Doppler measurements of blood velocity use lower ultrasound frequencies than is the case for imaging. The main reason for this is that the lower frequency makes it possible to measure higher velocities with pulsed Doppler without *aliasing.* Aliasing occurs when the Doppler shift in the reflected signal is larger than the Nyquist frequency, being equal to half the frequency by

which the ultrasound pulses are emitted. The velocity of the sound source can then not be reconstructed without ambiguity. Doppler sensitivity at deep depths also benefit from a lower frequency.

For medical applications, two types of Doppler techniques are being used [Hatle 1982]: *Continuous Wave* (CW) and *Pulsed Wave* (PW) Doppler.

1.3.1.a. Pulsed wave Doppler

In PW Doppler, a short burst of ultrasound (typically 8 to 10 cycles long, depending on the signal frequency) is emitted with regular intervals. The emitted signal is reflected by the moving blood elements back to the transducer. Knowing the propagation speed of ultrasound in the body, a time delay of Td from pulse emission until sampling the reflected signal then corresponds to a depth (also called range) R. For each pulse, the reflected signal is sampled once at a time Td after pulse emission. To be able to determine the velocity, however, the moving blood elements must be observed over a period of time. Data from several Doppler pulses, all emitted in the same direction, must therefore be combined to determine the elements velocity.

The frequency fs of consecutive Doppler pulses is called the *Pulse Repetition Frequency* (PRF). According to the well known *Shannon sampling theorem*, only frequencies less than half the sampling frequency fs can be reconstructed without ambiguity. If the reflected, Doppler-shifted signal contains frequencies higher than fs/2, also known as the *Nyquist frequency*, *frequency aliasing* occurs and the Doppler shift cannot be uniquely determined.

To avoid ambiguity in range, the Doppler pulse must be sampled before the next one is emitted. Otherwise, with more than one pulse out in the body at a time, it is impossible to tell whether the received signal is a first pulse reflection from a deep blood element or the second pulse reflection from a shallower one. To go deeper into the body without risking range ambiguity, the PRF must therefore be decreased. According to the sampling theorem, this in turn lowers the maximum Doppler shift that can be detected.

The feature of range resolution is PW Doppler's main attraction. This facilitates the construction of a 2-Dimensional flow velocity image.

1.3.1.b. Continuous wave Doppler

In CW Doppler, a continuous wave of ultrasound is emitted by the transducer towards the target subject to examination. The signal is reflected back to the transducer with a change in frequency corresponding to the target's velocity relative to the transducer. The main advantage with CW Doppler is that there is no theoretical limit on the velocities that can be measured. On the other hand, CW Doppler provides no range resolution. This is due to the fact that the reflected Doppler shifted signal is influenced by the motion of all blood elements along the transducer beam, regardless of their depth. Because the different blood elements have different velocities, the result of a CW Doppler measurement is a spectrum of frequencies representing the distribution of the blood element velocities.

Due to the lack of range resolution, the clinical applications of CW Doppler are where velocities exceed what is measurable with pulsed Doppler. An example of such is looking at very high velocity flow in conjunction with heart valve defects.

1.3.2. Doppler history

One of the first attempts to image real-time blood flow, described by [Omoto 1989], was called *contrast echocardiography*: An echo contrast material was injected into the body to provide information about the blood flow patterns. This method, however, had several disadvantages: It was not completely non-invasive, due to the injection of the contrast material, and it was generally suited to image flow in the right part of the heart only. To eliminate the contrast material, attempts were done to use Doppler technology in combination with 2D echocardiography to image intracardiac flow. The first approach to this was done in 1957, using continuous wave Doppler (CW) [Satomura 1957]. CW, however, does not provide range resolution, and efforts were therefore made to use pulsed wave Doppler instead. This was introduced ten years later [Baker 1967].

In the beginning, only velocities inside a small sample volume were measured at a time. To be able to have visual feedback of the location of the sample volume actually being measured, it was necessary to integrate the flow image and the echo image in some way. This was achieved by introducing the *duplex-scanner* mode, combining PW Doppler with 2D echo imaging using the same ultrasonic probe (transducer). The instrument operator was now able to position the sample volume freely within the 2D echo image, under full visual control.

In principle, the PW Doppler could easily be extended to measuring a contiguous row of sample volumes along the emitted ultrasonic beam. To achieve a satisfactory frame rate, however, each sample volume (range) must be processed in parallel. This technique is called *multigated Doppler* and was first introduced by [Fish 1975]. The data acquired was presented in several ways, most successfully as a colour coded image, superimposed on top of an M-mode echo image, today called *Colour M-mode*.

Due to the parallel processing of the multiple gates, a significant amount of extra hardware was necessary compared to the single gate PW Doppler. To reduce this hardware, serial-time-discrete signal processing similar to MTI (Moving Target Indicator), known from radar technology, was used. As increasingly faster and more complex electronic, integrated circuits were developed, the technology were in the beginning 80's ready for the next step: From one to two-dimensional Doppler. This was obtained by combining a multi-gated Doppler with a sector scanning transducer. For each Doppler beam in the image, evenly spread over the image sector, Doppler data were collected by all gates along the beam. In this way, a 2 -dimensional flow image could be constructed. To be able to maintain a satisfactory frame rate, doppler signal processing time now replaced "hardware volume" as the critical factor. The first 2D colour flow mapping systems were introduced by two independent research groups in 1982, Namekawa et al. [Namekawa 1982] and Bommer et al. [Bommer 1982].

Most available state-of-the-art ultrasound systems for cardiologic applications have more or less the same features and operational modalities. The following description is therefore *not* related to a specific system produced by a specific manufacturer, but more to be regarded as an "a la carte like" pick-and-choose from several currently available systems. Further, having the purpose of this thesis in mind, system features are omitted that are not related to and do not have an impact on architectural issues. The purpose of this chapter is solely to provide a platform for setting a specification for the processing architecture of the next generation, cardiologic ultrasound system.

2.1. Operational modalities

Operational modalities can roughly be divided into two groups: Image related modes, showing tissue structures like the cardiac walls and valves, and blood velocity (or flow) measuring modes.

2.1.1. Image modes

Of the three imaging modes discussed in the previous chapter, the A-mode is no longer relevant because it is covered by the M-mode. Image modes supported by ultrasound systems today are therefore:

- **2D imaging** or B-mode, providing a true 2-dimensional tissue image of the heart. Depending on the transducer being used and the selected trade-off between range and resolution, imaging depths are from a few up to about 30 centimetres. The image is usually sector-shaped (mechanical or annular array transducer), covering an angle somewhere in the interval from 30 to 90 degrees. If a linear array transducer is used, however, a rectangular image is the result.
- **M-mode** acquires image data along one direction only. The data are presented as a rectangular, horizontally sliding window. M-mode is mostly used to study selected details, for instance valve motion, and is therefore very valuable as a supplement to the 2D image: Having visual feedback in the 2D image, the M-mode beam can be positioned with great accuracy to show the detail of interest. This combinational operating mode is also called *duplex-tissue*.

2D imaging is by far the most important image mode in today's ultrasound systems. This because it provides a general view of the organ being imaged, forming a necessary "underlay" for the 2D flow image. Besides, it serves as a map for navigation when looking around for details in M-mode and Doppler.

2.1.2. Flow modes

These are related to some kind of blood velocity measurement. The theory behind the various modes have already been discussed in the previous chapter and will therefore not be repeated here.

- **2D flow** is based on the Pulsed Wave (PW) Doppler technique and gives a 2-dimensional image of the blood flow. Note, however, that due to the inherent nature of the Doppler effect only velocity components directed *along* the ultrasound beam are measured (i.e. the radial direction in a sector-shaped image). Because PW Doppler is used, a 2D flow image is subject to aliasing.
- **Pulsed Wave Doppler** (PW) is the normal quantitative measurement mode used at shallow depths and at greater depths for low to intermediate flow rates. PW is always used when accurate range localization of the flow signal source is desired.
- **Continuous Wave Doppler** (CW) is used for measuring high velocities. Because CW does not provide range information, the signal source range can be determined separately from velocity measurement by switching to PW Doppler.
- **High PRF PW Doppler** is a sort of compromise between PW and CW Doppler, trading off range localization accuracy against the ability to measure higher velocities. It is used for measuring flow velocities above the Nyquist limit. However, due to nature of this method, some range ambiguity must be tolerated when using HPRF PW Doppler.

2D flow is always used in combination with 2D (tissue) imaging and is the only 2-dimensional blood flow measuring mode. The remaining flow modes are all one dimensional and analog to (tissue) M-mode as data acquisition as well as display are concerned. They are used either stand-alone or in conjunction with 2D flow, to measure velocities at selected points. As for M-mode, the 2D tissue image can be used for visual guidance.

2.1.3. Diagnostic support

Synchronization of the 2D tissue and flow images with the cardiac cycle is provided for by including ECG (ElectroCardioGram) and phono traces. They make it possible to relate each 2D image to a specific point in the cardiac cycle.

2.2. Data archiving and review

To be able to do a more in-depth analysis of the data than it is possible to do in real time, some sort of review capabilities must be supported. They can be divided into two categories:

2.2.1. Video recording

All systems have video recording, which is obtained by simply hooking the system up to a video recorder (VCR). Depending on the system's video interface, the VCR can often be connected directly to the monitor's video signal. In case of an RGB interface, however, using separate signals for the three primary colours Red, Green and Blue, a special *composite sync* signal must be generated.

The VCR approach to data reviewing has its obvious advantage: It is the most standard way to store image information that exists. Every other home has a video recorder, it is portable and can be carried around. Often, especially in US clinics, it is ultrasound technicians, not physicians, that do the actual ultrasound examinations. The results are then to be interpreted later by the physicians. The "interface" between the two groups is the video cassette.

However, the disadvantages connected with video recording for data reviewing are just as obvious. One is the inevitable degraded image quality, even if a Super VHS video system is used. Often more unfortunate, however, is that with a video recording, it is the finally processed end product, the displayed video image, that is recorded, not the acquired ("raw") data itself. Therefore, no adjustments can be made whatsoever when reviewing the tape, no post processing can be done and no parameters can be changed. To overcome this problem, a number of systems support digital data reviewing.

2.2.2. Digital data reviewing

In this case, it is the acquired data that is stored. When reviewing, the data are sent through the same processing path in the system as in live mode. All parameters related to image display and post processing can therefore be manipulated just as if it was a live examination. Due to the large bandwidth requirement (in the range of 5 MBytes/second) a large semiconductor memory is required as storage medium. Replay loops, also called *cineloops* can also be set up. If the memory capacity is large enough, a full cardiac cycle can be shown at a user selectable, continuously variable speed. Synchronization to the cardiac cycle is done by using the ECG trace.

One disadvantage with this method is that it requires a full system (actually except from the acquisition part) to review the data. Another is the high bandwidth required to store the data on a non-volatile, high capacity medium. This is also necessary to use the method for data archiving, not only for reviewing.

Therefore, video recording is currently the only available method as far as data archiving is concerned. Digital data reviewing has its application during the actual examination, for example for previewing image sequences before storing them on the VCR for archiving.

2.3. Data analysis

The term "analysis" is here used to describe some form of quantitative evaluation of the acquired data. Currently, only relatively primitive functions are supported, especially as far as the 2D images (tissue and flow) are concerned. Typical examples are mean and max velocity computation in Doppler and determining the heart rate from the ECG trace. For the 2D images, available functions have a more or less geometric nature, computing parameters like area, distance and circumference. Note, however, that these functions are not fully automatic but rather operator assisted: Before the computation take place, the area or points of interest must be identified by the operator by using some sort of tracking device (e.g. mouse, track ball or joy stick). Automatic methods for edge detection or structure identification are currently not provided.

CHAPTER 3.The ultrasound system of tomorrow

The ultimate goal for most cardiologic, ultrasound system manufacturers is to be able to do true 3-dimensional, real-time imaging of a moving heart superimposing the flowing blood. This will, however, due to the vast amount of data involved and the complex computations connected with assembling those data into a 3-dimensional image, require an amount of computing power several orders of magnitude higher than today's systems. In addition to the computing power needed, this will also certainly affect the system structure involving multiple buses for transportation of the huge amount of data. Another key issue will be how this 3D image should be displayed, in real time on a 2D screen, in such a way that the user will be able to extract and interpret all that information actually available. It is probably no wild guess to say that this will require a display system structure that is totally customized for 3D imaging. Because of these and other reasons, for instance related to transducer construction, real-time 3D systems will be a topic of basic research for years to come.

3D imaging will therefore not be an explicit subject for this work, which will be restricted to 2D imaging. Many topics discussed will, however, be of equal importance to a 3D system. An example of such will be the high bandwidth digital communication channel for image data transportation.

An ultrasound system can be regarded as a collection of functional modules for data acquisition, data pre-processing, data post-processing (pre- and post being related to the data actually being displayed) and data display. The discussion about future development will therefore follow the same guidelines. Having the scope of this thesis in mind, that is, processing structures at the architectural level of the ultrasound system, such unrelated topics will *not* be discussed. An example of this will be issues concerning image quality only and not the system as such.

Before starting the discussion about the expected development of cardiac ultrasound equipment, however, it is important to have a fair idea of the environment such a system will be used in.

3.1. The diagnostic environment

Today, diagnostic equipment in a clinical environment is very much self-contained: The data are acquired, processed and displayed on the same instrument. The various instruments, even those which are of the same type and make, are seldom connected to each other. The only way of transferring data between instruments and "to the outside world" is by the video-tape. This situation is not likely to remain for very long.

It is very hard to believe that the concept of LANs (Local Area Networks) eventually will not be adopted by the hospital environment. All archived data are then accessible to every computer and medical instrument via this high bandwidth data communication link. This situation will in my view lead to a totally different direction in instrument development, at least as far as instruments at the level of complexity we are talking about here are concerned. A trend to

separate the acquisition part and the interpretation part of a cardiac ultrasound examination has been seen for years, especially in US clinics. In a few years, when the physician has a full-graphic workstation at his disposal, with a large screen permitting features like multiple cineloops and a user interface not possible in a "standard instrument", this development is likely to continue. Another benefit of a network-based system is the possibility to integrate image material from examinations with the generation of medical reports.

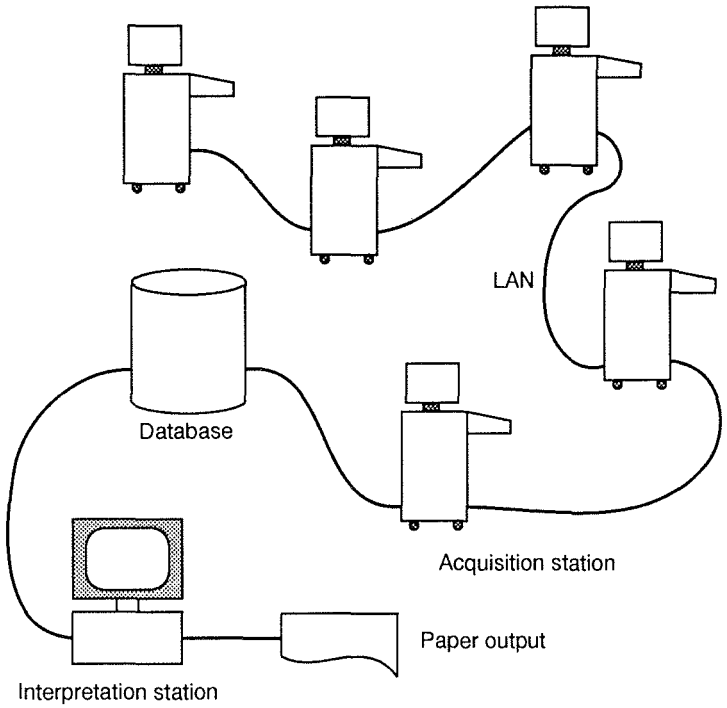


Figure 3.1. LAN-based clinical environment

Some years from now, medical instruments including top-of-the-line cardiac ultrasound equipment will therefore in my view be much more specialized towards either real-time data acquisition and examination support or data presentation and interpretation containing advanced post-processing functions. This will have a large impact on the systems' architectural design.

3.2. The ultrasound instrument

3.2.1. Data acquisition

Ideally, high beam density, deep sampling and a high frame rate are what we want to achieve. These three parameters, however, are all related to the same physical limitation, namely the speed of sound: Giving preference to one means degrading the other two. With a depth equal to 15 cm and the term "real time" defined as at least 20 frames per second, this gives time to acquire about 250 beams per image, tissue and flow together. Out of these 250 beams, a high

quality 2D tissue image will require 100 to 120 beams alone, depending on the angle covered. Image quality, for tissue as well as flow, will be improved by increasing the beam density, thereby improving the lateral resolution as well as the accuracy in determining the flow. However, as the figures indicate, there is not much room for increasing the number of beams beyond today's level without sacrificing either the frame rate or sampling depth.

The obvious solution to this problem is to introduce some sort of parallelism as far as data acquisition is concerned. This can be done in two ways, at least in theory:

By using a phased array transducer, *ultrasound beams can be emitted in several directions simultaneously*. The assumption for this approach to be successful, is that those directions are sufficiently spread apart so that interference between the beams is avoided. In practice, this is hard to do. Another factor in disfavour of simultaneous beam emission is that due to safety reasons, the total amount of ultrasound power to be emitted into the body is limited. Using several simultaneous beams means that the power of each beam must be reduced accordingly, thereby decreasing the signal-to-noise ratio of the reflected signal.

The other approach possible in a combined Echo/Doppler instrument is to *acquire tissue and flow data independently and simultaneously in a frequency multiplexed way.* According to [Kristoffersen 1985], this method was patented in the US as early as in 1979. The large bandwidth and dynamic range required make it, however, very difficult to avoid interference between the two systems.

3.2.2. Data pre-processing

As explained earlier, the term "pre-processing" is in this context related to the data actually being displayed. That is, the purpose of the pre-processing is to make the data display more pleasing to the eye and, most important, easier for the operator to interpret. In image processing terminology, this kind of processing is called *image enhancement.*

The goal of doing image enhancement is twofold: To remove (or at least reduce) noise without losing details and structures contained in the image. This can only be done by using some *a priori* knowledge about the image data. For a cardiac ultrasound image taken from an arbitrary point of view it is difficult to make a mathematical model, describing the image data in itself, which will be valid in all cases. The only practical approach when doing image enhancement on these images is therefore to use some sort of correlation, in space and/or in time.

Spatial correlation

Spatial correlation algorithms utilize the fact that the value of every given picture element (pixel) will have a strong correlation to its surrounding pixels. The "correct" value of a pixel element can then be computed as a function of the values of the (original) pixel itself and its neighbours. The actual function being used and the size and shape of the neighbourhood will depend on the more specific goal of the enhancement being done and the properties of the image. Due to the sector shaped data acquisition scheme used in cardiac ultrasound (the exception being images acquired with a phased array transducer), the lateral distance between

two samples taken at the same (radial) depth will be smaller the smaller the depth is. The degree of spatial resolution is therefore in case of a sector shaped image inverse proportional to the actual depth.

However, due to the basic idea of this approach, all techniques using spatial correlation have more or less the unwanted side-effect of degrading and blurring details in the original image, details necessary for a correct image interpretation. Examples of such details in cardiac ultrasound images are the valves and boundaries and walls of the heart muscle. Provided they are not completely lost, they can, however, be reinforced through a subsequent step of processing. In image processing terminology this is called *edge enhancement* and will in principle be a kind of differentiation operation.

Time correlation

As far as cardiac ultrasound systems are concerned, the term "real time" implies that the frame rate by which new images are acquired, is sufficiently high compared to the relative change from one image to the next. Otherwise, a very "jumpy" appearance will be the result when displaying image sequences. Another most unfortunate aspect of low frame rate (or rather slow scanning) would be the relatively large difference in time of acquisition between the data at the left and right edge of the image. Besides, correlation between tissue and flow data, which ideally should have been acquired simultaneously, will suffer. Therefore, a high (enough) frame rate is an absolute prerequisite for real-time imaging.

The consequence of this is that the pixel value at a certain location in one frame will have a strong correlation to the pixel values, at the same location, in the preceding and the succeeding frames. In the same way as for spatial correlation, pixel values can be filtered by taking the neighbouring pixels (but now in terms of time) into account. The side-effect of blurring is equally valid with this kind of filtering, a subsequent step of edge enhancement processing will therefore often be appropriate.

3.2.3. Data post-processing

To be able to do accurate ultrasound diagnostics, it is often necessary to have some form of quantitative description of the ultrasound image. Depending on which kind of information we seek, this analysis can be based on a single image, an image sequence (e.g. to do wall motion analysis) or images taken from different examinations (of the same patient) over a period of time if it is long time trends which are of interest. As already indicated, data post-processing is of a more quantitative nature than pre-processing which is qualitative. Once again to draw the analogy to image processing terminology, this kind of image analysis is often called *feature extraction.*

Because the data pre-processing is done on image data to be displayed, with the purpose of making that data more pleasing to the eye and easier to interpret, needless to say it must be done in real time. Otherwise, its only application will be as a "beautifier" for stored images, which reduces its usefulness considerably. The data post-processing, however, has no impact on the actual data being displayed and does therefore not require real-time execution. In fact, it is not necessary to do the post-processing in the ultrasound (acquisition) instrument at all. Probably it will be better to do this kind of processing off-line on a dedicated "ultrasound diagnostic

station", having better graphic capabilities and a user interface that is more suited to this kind of work. As described earlier, the diagnostic station will have access to the acquired ultrasound data through some sort of local area network in a future diagnostic environment.

The scope of work for this thesis is the *real -time processing part of the system.* The data post-processing functions are beyond that scope, and will therefore have no direct impact on architectural issues discussed. However, for the sake of completeness, a brief survey of relevant post-processing topics is given. A general feature of post-processing functions is that, seen from an algorithmic point of view, they are very sequential in nature. Edge detection, an absolute prerequisite for many post-processing functions, is a good example of that. Usually, these functions are therefore not equally well suited for parallel implementations as for instance the typical image enhancement pre-processing functions are. Post processing may therefore be implemented just as efficiently on a standard workstation as on customized hardware, which in this case, due to the sequential nature of the problem, probably would be an off-the-shelf CPU-card anyhow.

3.2.3.a. Tissue post-processing

To be able to reveal long time trends towards some sort of heart malfunction, in order to take care of the problem before that malfunction occurs, it is necessary to quantify the cardiac function according to some set of parameters. These parameters can be related to motion (wall and valves) as well as their ability to reflect the emitted ultrasound pulses.

Tissue characterization

According to [Lee 1989], tissue characterization is perhaps the most relevant application of tissue post-processing. The term *tissue characterization* is used for specific questions that can be raised about the physical object being imaged. Examples of such questions can be to decide the degree of ischemia in the heart muscle, the malignancy of a tumour etc. As far as ischemia is concerned, one tries to characterize, or grade, the ischemic myocardium in terms of its "reflectability". The reflectability of tissue is a function of its comparative compressability, which again is related to the proportional fluid (collagen) and calcium compared to its neighbours along the line of sound transmission. The reflection of sound by a target is one out of many factors which contribute to the attenuation of the sound as it travels from the transducer, into the body and is reflected back to the transducer again. If it were possible to detect changes in those parameters in some absolute and reproducible way, looking at the trend from one examination to the next, cardiac problems could have been diagnosed at a much earlier stage than is possible today.

Volume computation

The ability to locate edges in the cardiac image is a key to many post-processing applications. As far as the tissue image is concerned, this would open for the computation of atrium and ventricular volumes as a function of the cardiac cycle. These measures will be a valuable complement to the tissue characterization, having the purpose of detecting long term changes in the cardiac function.

3.2.3.b. Flow post-processing

According to [Linker 1989], analysis of 2D Doppler data can be coarsely classified as volume integration, flow profiles and feature extraction.

Volume integration

The volume of flow is perhaps the important parameter as far as 2D flow analysis is concerned. In principle, this can be calculated simply by multiplying a given area with the flow (velocity) through it. Due to a non-uniform, parabolic like velocity profile, however, with larger velocities near the centre of the vessel than at its periphery, the velocity profile must be integrated over the vessel's area to compute the volume flow. A full-velocity profile would ideally require 3D Doppler. This is because the measured Doppler shift is only affected by the blood elements' velocity component *along* the direction of the ultrasound beam. The ideal 3D Doppler can be approximated by measuring multiple flow profiles having different orientations. Because the velocities also vary during the cardiac cycle, integration must be done over time as well.

Flow profiles

Flow profiles are velocity cross-sections taken at a single point in time (snapshots). The volume flow described above can then be obtained by integrating the flow profiles over time.

Ideally, to make a true snapshot, all data covering the whole cross-section should be acquired simultaneously. In practice, however, this is not possible. Data are collected serially, the sequence being dependent on the type of transducer being used. Phased arrays are due to the electronic steering not restricted to collecting data from beams in a strictly, sequential manner, but the data are nevertheless collected serially. The beam acquisition sequence must be known to be able to compensate for the difference in acquisition time. Besides, the data from the previous (in time) flow profile must be available. This is necessary for each point to determine the direction (increase or decrease) of the compensation. Having this information, an approximated "true" flow profile can be calculated, showing the blood flow at a single point in time.

Feature extraction

Feature extraction is a very general term referring to extracting various kinds of quantified information from the 2D flow data. Examples of such parameters are width, length and area of flow jets. Another is the calculation of flow acceleration. This can be computed directly from flow profiles taken parallel to the ultrasound beam.

3.2.4. Image display

As accounted for earlier, future top-of-the-line ultrasound systems will be more specialized towards either data acquisition or data display than today's instruments are. In the data acquisition system, the subject of this work, the display will serve as operator guidance for doing the best possible examination, and recording the best possible images. To do this, needless to say, the system must support the real-time display of image data as they are acquired. This includes the 2D image data (tissue and flow) as well as data of more supportive character as M-mode, 1D Doppler and several types of traces (ECG, phono etc.).

A replay facility of some kind must be supported to determine which image sequences to record. This makes it possible to review image sequences, preferably at a variable speed, prior to recording. Simultaneous review of at least two separate image sequences should be supported to be able to compare the sequence of interest to already recorded material.

As a fundamental rule, the only limitation to system performance in a true real-time system, should be the constraints imposed on the system by the environment in which the system is designed to operate. Talking more specific about a medical ultrasound system, the system capacity in terms of frame rate and data processing capability should be the rate by which new data are available. Or in other words, the inherent physical limitations of doing data acquisition by ultrasound.

> *This is an absolute demand when setting the specification for a real-time ultrasound system.*

Data rate

As explained earlier, the frame rate, number of beams per frame and sampling depth parameters are closely related to each other through the physical value of speed of sound:

$$t_{frame} = t_{beam} * no\_of\_beams$$
$$t_{beam} > t_{sample\_depth}$$
$$t_{sample\_depth} = 2 * sample\_depth / v_{speed\_of\_sound}$$

Increasing one of the three parameters means lowering the other two. To increase the frame rate ($1/t_{frame}$) while keeping the number of beams constant (no_of_beams), the acquisition time available to each beam (t_{beam}) must be decreased. This will in turn reduce the maximum sampling depth (sample_depth). No matter how sophisticated a transducer may be or the amount of computing capacity contained in the system, the rate by which data are acquired is therefore not likely to change much beyond today's level, which already utilizes the available acquisition bandwidth very well.

As far as 2D image data are concerned, the typical rate by which data are output from the frontend block of today's systems are:

```
Tissue : 128 beams * 512 samples *  8 bit = 128 kbyte
Flow   :  64 beams * 128 samples * 16 bit =  16 kbyte
                                            ----------
                                             144 kbyte
```

With the "real time" lower limit quantified to 20 frames per second, this means a total 2D image data rate of approximately 3 Mbyte per second. Compared to the 2D data, the amount of other kinds of ultrasound data are so small that it need not be considered as far as data rate

calculations are concerned. Adjusting the bandwidth requirement according to good principles for conservative design, the required bandwidth for the acquisition and processing of ultrasound data can be set equal to 5 Msamples (Mbyte) per second.

4.1. System basic blocks

An ultrasound system can roughly be divided into four basic blocks as shown in Figure 4.1. The term *block* is here to understand rather as a conceptual than a physical term, grouping a number of *modules* functionally belonging together. Each module has a dedicated function and can, depending on its size and complexity, be implemented on one or more physical cards.

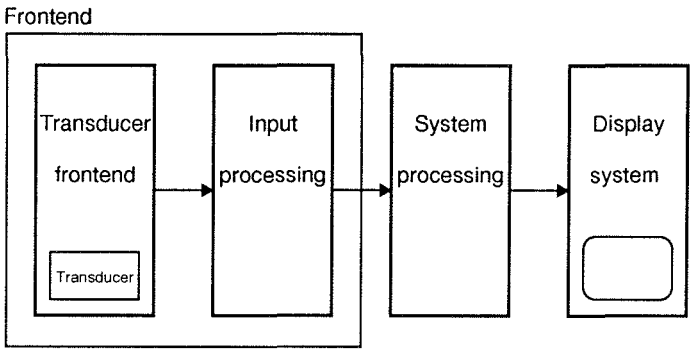


Figure 4.1. System basic blocks

The blocks shown in Figure 4.1.have the following functions:

4.1.1. Transducer frontend

The transducer frontend block contains the ultrasound transducer and all electronics necessary to control it. The actual realization of the frontend is strongly dependent on the type of transducer being used. For a mechanical transducer, the ultrasound beam is steered by physically positioning the transducer element before emitting the ultrasound pulse, this positioning being done by some sort of electrical motor. One task of the frontend is to control that motor to make the element point along the right direction, and then fire the pulse. In the case of an electronic transducer, beam forming and steering are done electronically by the various transducer elements emitting ultrasound signals slightly different in time and phase. The signals from all transducer elements will thereby interfere with one another, generating a resultant signal. This resultant signal can electronically be steered in direction and focused in depth by varying the relationships between the signals emitted from the various transducer elements.

In addition to beam control, the transducer frontend contains all electronics related to the emission and reception of the ultrasound signals. Generally speaking, the size and complexity of the frontend block will be significantly larger with an electronic than with a mechanical transducer.

4.1.2. Input processing

The input processing block can be regarded as a sort of "level 0" processing, drawing analogies to the hierarchical models known from other computer disciplines, for instance data communication. In other words, it contains signal processing functions very closely related to the physical properties of the emitted ultrasound signal as well as the objects being imaged. Examples of input processing functions are

- envelope detection of the echo (tissue) signal
- estimation of Doppler frequency for velocity measurements
- gathering of velocity measurements from different directions into a 2D flow image

The output of this block will be in the form of digital tissue and flow data, each organized as a 2-dimensional matrix with one dimension being equal to the beam number and the other equal to the depth position within the beam.

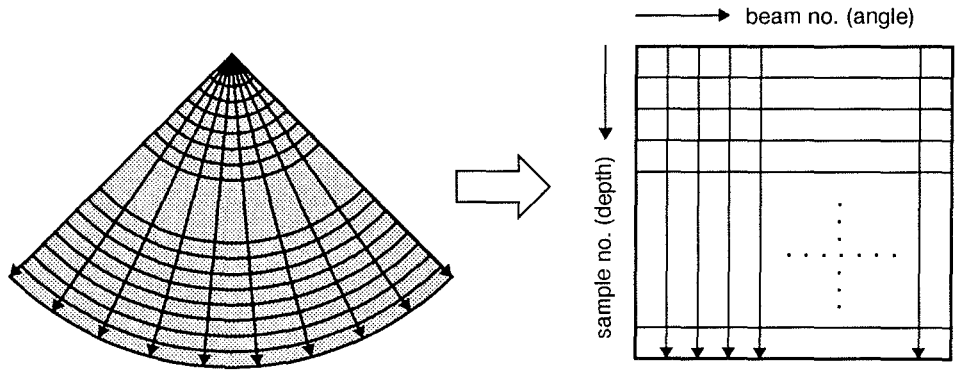


Figure 4.2. Raw data format

With all transducer types except from the linear array, the data output from the input processing block describes a sector shaped image and will therefore be on polar form (angle vs. distance). From now on these data will be referred to as the *raw data*. In addition to (scan) conversion into a format more suitable for display, a lot can be done with the raw data to make them more pleasing to the eye and thereby easier to interpret. The more quantitative post-processing functions will also use the raw data as input.

In the rest of this work the transducer frontend and the input processing blocks will be regarded as a self-contained module, simply called the *frontend*. This block will provide input to the rest of the ultrasound system, but will have no influence on the system architecture as such other than the format and the rate by which it provides the (raw) ultrasound data.

4.1.3. System processing

The system processing block is the main and the most complex block in the system, at least as seen from a system level point of view. It will contain a variety of functional modules, the actual configuration being to some extent application dependent.

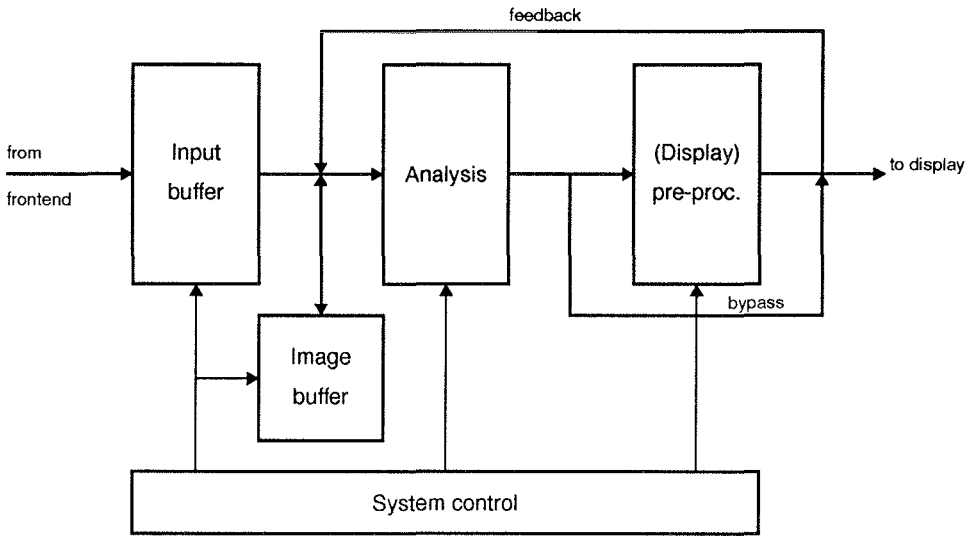


Figure 4.3. System processing block

The main functions handled by the system processing block are:

Reception and buffering of raw tissue and flow data from the frontend. For live data, this is the gate into the system processing block.

(Display) pre-processing. As already explained, ultrasound data are due to the very difficult acquisition conditions by nature noisy and difficult to interpret. The purpose of the pre-processing is ideally to remove that noise without losing details and structures in the image. For best results, it should be possible to customize the pre-processing according to the application as well as the data being processed. Given a set of filtering modules contained in the system processing block, these may be combined into different processing sequences, possibly including a loopback feature if the image after processing does not comply with some specific set of measures. As far as the modules contained in the display pre-processing part are concerned, some sort of dynamic reconfiguration capability should therefore be supported.

Quantitative analysis. The data must first be analysed to be able to adjust the pre-processing algorithms to the characteristics of the data being processed. The result of this analysis can have the form of statistical measures like mean and variance values, or a histogram. Depending on the actual purpose of the analysis, they can be global for the image as a whole or be individually computed for different regions of the image. In case of an iterative processing scheme, it will be the task of the analysis modules to provide data determining whether to let the image data

continue to the next stage in the processing sequence or to loop it back to repeat one or more of the processing steps contained in the current stage. Real-time calculations of image parameters as Doppler mean and maximum velocities will also be done by the analyse modules.

Buffering for image review. All ultrasound data together with a complete set of parameters describing how that data were acquired (i.e instrument setup at acquisition time) should be conceptually stored in a large ring-buffer. With today's storage medium technology, this ring-buffer must be implemented by semiconductor memory to obtain real-time storage as well as real-time review. To have maximum flexibility as far as display pre-processing and quantitative analysis are concerned, the data being stored are the raw data output from the frontend block. On review, the data can then be replayed following the same path through the system as it did during acquisition. In this way, review data can be manipulated as live data as far as display and pre-processing are concerned.

System control. Depending on the implementation, this module may be more conceptual than physical. The function of synchronizing the other modules and controlling the data transfer between them may just as well be distributed onto each module, instead of using a dedicated, centralized controller module. The best solution will depend on the chosen architecture and the communication scheme by which the data are transferred between the modules.

4.1.4. Display system

Needless to say, an absolute demand to the display system is that it is able to display data in real time. A complete medical ultrasound displayed image will be assembled by components from several modules. These components can be grouped into static and dynamic parts.

The **static** part contains text and graphics. The text is used for displaying information about the current image. Examples of such information are patient information (name, date of birth etc.), date, instrument operational mode and the value of relevant image parameters concerning instrument setup (gain, reject) as well as continuously updated parameters computed from the acquired data (Doppler mean and max velocities). Graphics are used to create image markers for depth, Doppler sample volume, visual feedback of the TGC (Time Gain Control) setting, cursor etc. Due to its static nature, the text and graphics will have the character of being an overlay to the dynamic part of the display.

As the name implies, the **dynamic part** consists of data changing from frame to frame. Like the static data, the dynamic data is assembled by components from several modules, the 2D tissue and flow data being the most important. Additionally, the dynamic data also includes other ultrasound data (e.g. tissue M-Mode, Doppler) as well as various traces (ECG, phono etc.).

Several factors will have impact on the display system architecture:

Display modularity

It should be possible to freely compose various display modes, without any other constraint than the display size, by (re)arranging the display components available in the system. There should therefore be no such thing as "unfortunate" combinations being impossible to implement due to processing power limitations. To be able to compare several 2D image sequences to each

other, the display system should support at least two 2D windows. This feature will basically be used for comparing images rather than doing diagnostic interpretation of the recorded images (which should be done on a dedicated, high quality graphic workstation).

Display dynamics

The components constituting the dynamic part of the display have very different characteristics as far as image updating are concerned. The 2D image components tissue and flow have **full update** by that a set of data describing the whole image component is transferred when the component is to be updated. The display dynamics is therefore caused by changing the data itself, and not by rearranging data that is already stored and displayed.

The contrast to this is the M-mode and Doppler spectrum display components, best supported by a **vector update** mechanism. They can best be described as rectangular images sliding from the left to the right side of the screen while new data are filled in as vertical vectors (bars) along the right edge as the image is moved. Display updating is therefore mainly done by redisplaying the old data in a shifted position compared to the previous image.

A similar mechanism is required for updating the traces, which are simply value-vs.-time curves. On every update, the curves are shifted one position to the right with the new data value plotted in the extreme left pixel position (**pixel update**). If the vertical (i.e. value) distance between the previous and new data are more than 1 display image pixel, the intermediate pixels must be filled in to get a contiguous curve.

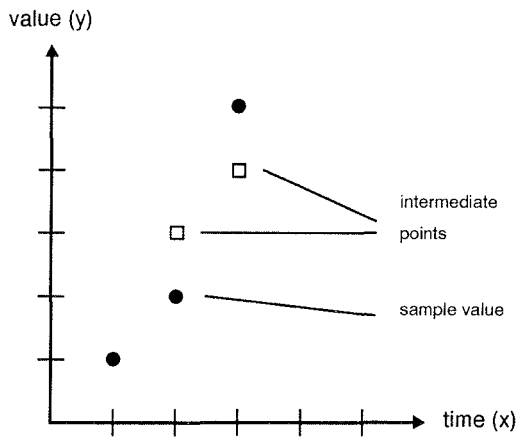


Figure 4.4. Trace display

The display system architecture should support display component positioning by some kinds of pointer mechanism to avoid the need for physically moving (i.e. copying) the displayed data into the new position. Then there would be no danger of getting into "unfortunate" display combinations requiring excessive computer power as earlier described.

Display response

While display dynamics is a term telling how the image display changes with the change of image data, *display response* is used to describe a situation where the display must be changed due to some sort of instrument - user interaction. This interaction might be by selecting a new operational mode or changing a parameter which in turn will cause the display to change. This parameter can be a data acquisition as well as an image display parameter. Even though the execution of all actions the change involves may take some time, the *display* should change momentarily to give the user the impression of a short response time. Because the frame rate in some operational modes will be as low as a few frames per second, the time to change the display should not be dependent of the current frame rate.

Resolution requirement

A 2D tissue image will typically have up to 512 samples per beam, distributed over a maximum angle of 90 degrees.

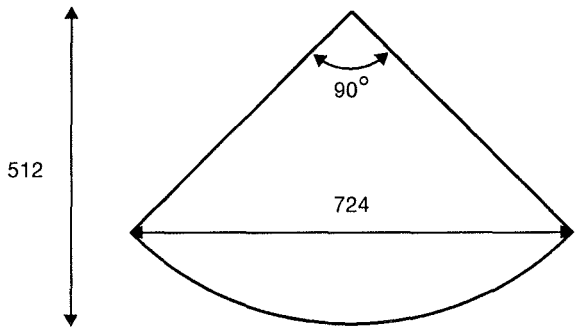


Figure 4.5. Image sector geometrical format

To be able to display most of the data actually acquired, the display resolution should be at least 512 by 512 pixels. By shrinking the image somewhat, a full size 90 degree sector will then fit into the screen together with the necessary text and graphics. By increasing the resolution up to 1024 by 1024, a full quality review of up to 4 simultaneous image sequences are supported.

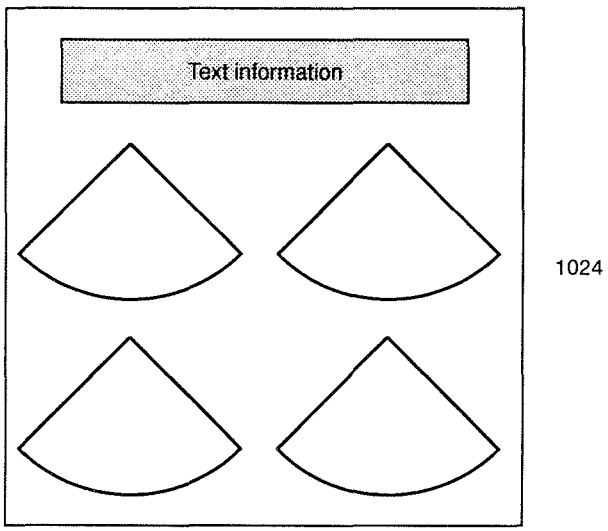


Figure 4.6. Quad sector display

The usefulness of this feature will depend on whether a diagnostic workstation is available or not. If not, high quality review of multiple image sequences should be supported by the ultrasound acquisition instrument itself. For this reason, it should be possible to customize the display system as far as the resolution of the display is concerned.

4.2. Design guidelines

It is always difficult to specify something where it is not completely known what it is. What can be done, however, is to determine the kind of characteristics and features a system should have in order to suit the application(s) it is intended for in the best possible way. These characteristics and features can then be summarized into a set of *guidelines* governing the actual system design.

As already accounted for, out of the four described basic blocks the design of the "system processing" and the "display system" blocks are defined to be within the scope of this work. The other two, called by their common name the "frontend" block, defines together with the user itself the environment in which the system and display blocks must be designed to operate.

By extracting the essence of the discussion so far, a set of design guidelines for the next generation ultrasound system can be expressed as follows:

Performance

- The system should be capable of handling a throughput of at least 5 MSamples of ultrasound data per second, measured at the frontend output. Depending on the actual implementation, the required bandwidth of the communication channel within the system processing and display blocks may be significantly higher.

System structure

- To ensure optimum results for all applications within the instrument's scope of operation, utilizing the processing resources actually available within the system in the best possible way, dynamic (re)configuration of the functional modules should be supported.
- For the same reason, it should be possible to dynamically adjust the actual processing being done to the characteristics of the ultrasound data (data dependent processing). This adjustment can have the form of rearranging the functional modules as well as changing various algorithmic parameters.
- Iterative processing should be supported.
- To be able to increase processing capacity, the system should be capable of handling several identical functional modules as a "pool" of resources selecting the first one available. Which module actually is used should be invisible to the rest of the system.

Display system

- The display should have a resolution of at least 512 by 512 pixels. To be able to adopt to the actual user's need, however, the display system architecture should support higher resolutions, possibly of up to 1024 by 1024 pixels.
- Special care should be taken to avoid the display system itself being a limitation the system's operational capacity and performance.
- The three types of display updating schemes (full, vector and pixel update) should be supported to ensure maximum display dynamics.
- Display response as it appears to the user should be independent of the current frame rate by which the ultrasound data are acquired.

CHAPTER 5.Computer taxonomies

When trying to select a computer architecture for a given application, there are a number of candidates to choose from, each one having its own vices and virtues. To help making the right choice, it is important that the different architectures are systemized according to some criteria or features. In other words, some sort of architecture classification scheme, or *taxonomy*, is needed.

Generally speaking, there are at least three reasons for classifying architectures [Skillicorn 88]:

- The number of systems with different kinds of parallelism has been explosive since the introduction of the von Neuman machine. Classifying the various architectures makes it easier to understand what has already been achieved.
- A good, and for the purpose in question, appropriate classification scheme, might reveal architectural permutations and combinations of existing architectures, creating new architectures with other (and may be useful) features.
- Implicitly, the very fact that an architecture is classified according to some sort of classification scheme means that certain things about that architecture's features and characteristics can be said. This facilitates the building of useful models of the architecture's performance, for instance for the purpose of doing performance analysis and comparison with other architectures.

Most important in this context, however, is to have an appropriate classification scheme as a tool for selecting the best system architecture for a given application. With that in mind, to be "appropriate", the classification scheme should have the following properties:

System level. The scheme should be high-level in nature to serve as a *system* classification scheme, reflecting the system behaviour as well as its characteristics. That is, the emphasis should be on system related issues as module connectivity, topology and inter-module communication. The different modules themselves can in this context be treated as "black boxes", but with clear specifications as far as their interface to the outside world (that is, the system) is concerned.

Completeness. The taxonomy should cover all relevant architectures, including possible permutations and combinations. One way to ensure this is to organize the classification criteria as levels in a hierarchy, where each level is covering the full set of architectures presented by the level above. The various architectures as defined by the permutations of the different classification criteria will then come out as leaf nodes from the bottom level. Not all leaves will have a name in terms of a specific, existing architecture. They may, however, represent (hopefully useful) combinations of architectural properties from the levels above those not yet implemented as a system architecture.

Consistency. The term "consistency" implies that the grouping in categories should be well defined and non-overlapping. The hierarchical model outlined above is well suited to achieve this. Consistency also means that all architectures should have their "natural" place in the chosen classification system. There should be no need for creating special categories for architectures not fitting into the scheme, thereby disrupting the whole classification system.

Classification properties. Classification properties should be selected according to the purpose of the classification scheme. In this context, it is to select the best suited architecture for a given type of application. The classification should be done on the basis of system behavioural features rather than lower level issues such as implementation techniques and module design.

Having this in mind, we can now consider some of the numerous attempts at doing computer architecture classification.

5.1. Computer taxonomies

5.1.1. Flynn's taxonomy

The first (and now classic) systematic approach to classify various computer architectures according to some set of criteria was done by Michael J. Flynn in 1966 [Flynn 66], [Flynn 72]. His work was inspired by the development of the large-scale scientific computers and the different computer architectures proposed to overcome the so-called von Neuman bottleneck.

Flynn's classifications scheme is based on the information floating around in a computer being decomposed into two separate streams: Instructions to be executed by some sort of Processing Unit(s) (PU) and data being operands to those instructions. Depending on whether the two streams are common or separate to all PUs, a computer architecture can be classified into one out of four categories:

SISD: Single Instruction stream, Single Data stream. This is the conventional von Neuman architecture. Only one instruction can be decoded in a unit time, thus, no more than one instruction can be executed in the same time quantum.

SIMD: Single Instruction stream, Multiple Data stream. All PUs execute the same instruction at the same time working in complete lockstep with each other, but on their own (and most often exclusive) part of the total set of data.

MISD: Multiple Instruction stream, Single Data stream. In computers of this category, every PU executes its own instructions but on the same data. It is difficult to imagine a practical, realizable architecture fitting completely into this category, but by stretching the definition a bit, pipeline architectures can be said to belong to this group.

MIMD: Multiple Instruction stream, Multiple Data stream. A MIMD machine consists of a set of more or less autonomous PUs, each one executing its own set of instructions on its own set of data.

As a general computer architecture classification scheme, Flynn's taxonomy has several shortcomings:

- The scheme does not include a classification of how the two streams, instructions and data, interact.
- It is based on large-scale scientific computer architectures, making it difficult to fit some of the more modern architectures (e.g. pipeline machines).
- The classification scheme is restricted to a shared memory, static configuration. The PUs are organized in a homogeneous, single level hierarchy, that is, all PUs are equal and interchangeable.
- The aspect of input and output channels is ignored.

5.1.2. Danielsson's taxonomy

Danielsson's way of architecture classification [Danielsson 81] is based on the different types of parallelism possible in image processing operations. According to [Danielsson 81], there are four orthogonal dimensions of parallelism:

Operator parallelism is equivalent to pipelining: The computational task is divided into a set of consecutive stages, each stage executing its own operator (program). A stage receives its input data from the previous stage and delivers its output (result) data to the next stage in the pipeline. All operators, that is, the processing within each stage, are executed in parallel.

Image parallelism means that the PUs are computing separate output pixel values for separate neighbourhoods in the same output image and in full synchronism with each other.

Neighbourhood parallelism means that each PU computes the values of several output pixels within the same neighbourhood at the same time. This requires immediate parallel access to at least one bit-plane at the same time.

In computations utilizing **pixel-bit parallelism**, all bits constituting the pixel value (typically 8 bit), are processed simultaneously in a word-parallel fashion known from most conventional computers.

Each of the four parallelism dimensions can be assigned a value giving the degree of parallelism along that dimension. By multiplying the 4 values, the architecture's total degree of parallelism K can be computed as

```
K = ko * ki * kn * kp
```

Provided that the task's degree of parallelism along each of the four dimensions are equal to or greater than the architecture's degree of parallelism along the same axis, the computed total degree of parallelism K can be regarded as a measure for the architecture's potential of parallel computation.

Danielsson's approach is therefore useful for quantifying an architecture's potential as far as its parallel processing capability is concerned. As a qualitative or descriptive instrument for classifying computer architectures on the system level, however, it is less suited. The reasons for this are:

- The classification (degree of parallelism) is dependent on which architectural level you look at.

- The parallelism criteria including terms such as neighbourhood, pixel-bit etc. makes the scheme better suited for classifying low level structures than system level descriptions.

5.1.3. Kidode's taxonomy

Masatsugu Kidode [Kidode 83] uses the image memory structure of the processing system as the classification criteria. The reason for this is that the time spent for transferring data between mass and core memory for some tasks may be higher than the computation time itself. It will then be more relevant, at least as far as potential processing power is concerned, to group systems according to the organization of their image memory than to the (inter- or intra-) structure of the processing elements. Image processing systems should therefore have either large core memory or high speed transfer mechanisms. Existing systems can be classified as belonging to one out of 7 categories according to their memory structure.

As for the previous two schemes, Kidode's approach is not very well suited as a general, up to date taxonomy for the classification of computer architectures:

- It does not contain any information about control structure or topology.

The classification criteria is itself more or less obsolete due to the availability of low cost, high speed and high capacity semiconductor memory. The majority of current image processing systems will therefore fit into one out of a few (2 or 3) groups.

5.1.4. Preston's taxonomy

Preston [Preston 83] introduces a classification scheme for cellular logic computers (CLC). CLC computers are used for computation of two- and three-dimensional logical neighbourhood operations in image processing. A CLC is distinguished from an "ordinary" computer by that the CLC only performs logical rather than numerical transforms. A CLC image processing system is therefore by nature best suited for operations on binary images. Gray-level image processing requires the image to be divided into a stack of binary images using multi-thresholding. After processing, the binary images must be put together into a gray-level image by some sort of arithmetic recombination.

The basic idea behind Preston's classification scheme is the distribution of processing power: The total array of image data can be divided into a number of sub-arrays. According to the number of processing elements, PEs, doing simultaneous processing on the image sub-arrays, image processing architectures can be divided into three categories:

Single-element sub-array machines. The image is processed in a serial manner by one single processing element. The output from each processing operation is the (centre) pixel value of one neighbourhood.

Multi-element sub-array machines. To speed up execution, these architectures use multiple processing elements, processing multiple neighbourhoods (one neighbourhood pr. processing elements) in parallel.

Full-array machines. Full array machines contain a large number of processing elements, interconnected as an array.

The absolute distinction between the multi-element and the full-array machines is from [Preston 83] not clear, except that the full-array machines contain a considerably larger number of PEs (hundreds to thousands) than the multi-element machines (in the order of tens).

As is the case for a number of other classification schemes, pipeline machines do not fit well into this one either. Pipeline machines are therefore included as their own (the fourth) category. However, this represents a break with the scheme's original idea: A pipeline machine implements parallelism in the time domain while the first three categories are based on parallelism in the space domain.

Due to its restriction to a special subset of computer architectures as well as to its non-consistent classification (i.e. the pipeline machine), Preston's scheme is not particularly suited as a system classification taxonomy. It seems to be a better choice for classifying low level image processing architectures, for example SIMD machines (see [Flynn 66]).

5.1.5. Duncan's taxonomy

Duncan's approach to computer taxonomies [Duncan 90] is based on [Flynn 66], a classification according to instruction and data streams. Several modern architectures which are not easily accommodated by Flynn's scheme, nevertheless seem to intuitively merit inclusion as parallel architectures. An example of such an architecture is the pipelined vector processor which according to a pure Flynn classification would be a sort of MISD (pipeline) and SIMD (vector) hybrid. One of Duncan's goals in defining a taxonomy of his own was to include those architectures. On the other hand, architectures that only incorporate the low-level parallelism commonly found in most computers of today, should be excluded as parallel architectures. Low-level parallelism in this context means features such as instruction pipelining, multiple CPU functional units (e.g. dedicated co-processors for mathematical computations) and separate CPU and I/O processors. Although they all make significant contributions to the machine's overall performance, they do not justify the architecture being classified as a parallel architecture as such.

The result of Duncan's effort is a rather pragmatic and informal approach to computer classification based on three main categories of architectures:

Synchronous parallel architectures coordinating concurrent operations in lockstep through global clocks, a central control unit or vector unit controllers.

MIMD architectures employ multiple processors that can execute independent instruction streams on local data. These architectures are most often implemented as asynchronous computers, with (task) synchronization through software by some sort of message passing scheme and decentralized hardware control.

MIMD-based architectural paradigms accommodate architectures based on the MIMD-principles of independent, asynchronous operation and concurrent manipulation of multiple instruction and data streams. However, they also possess their own distinct and individual characteristics making them hard to classify as pure MIMD architectures in the classical point of view.

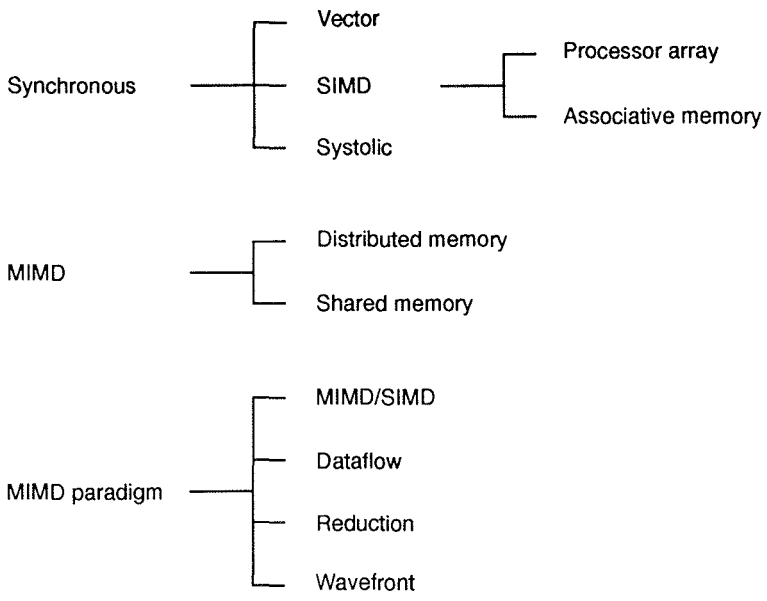


Figure 5.1. Duncan's taxonomy

In my view, the weakness of Duncan's taxonomy as a system classification scheme is its lack of formality. In most taxonomies, the grouping of architectures is done on the basis of some sort of a more or less systematic and formalized idea. Although based on the Flynn taxonomy, the result of Duncan's effort has more the nature of an ad hoc approach. The attempt to include modern, parallel architectures is in my opinion the main reason for this. By not taking system topology into account, the classic Flynn's taxonomy may simply not be the optimum basis for a system classification scheme to include today's parallel architectures?

5.1.6. Yalamanchili's taxonomy

A multiprocessing, image processing system can be thought of as assembled by two fundamental type of elements [Yalamanchili 85]:

- Processing Elements (PE) and
- Communication Elements (CE)

As far as memory organization is concerned, a processing element can either have its own local memory or share a global memory with other, but not necessarily all, PEs. The combination of local and global memory is also possible. Depending on the chosen architectural philosophy, the PEs in the system can be classified as

Homogeneous. This will be the case if the system is a general purpose, programmable system where the concept of multiprocessing is applied with the purpose of achieving load distribution and thereby (at least intentionally) execution speed-up. Because the PEs all are identified, one element can be replaced by another without changing the functionality of the system.

Heterogeneous. In a system built of heterogeneous or different processing elements, each (type of) element will have its own dedicated function and can not be replaced by another element of a different type.

To connect the processing elements together, making a system, several kinds of interconnection structures are possible. It is important here to distinguish between a *physical* and a *logical* interconnection structure: By appropriate control and reconfiguration mechanisms, one physical interconnection structure can realize a number of logical PE interconnection structures (e.g. tree, ring, pipeline etc.). Three major classes of physical interconnection structures can be identified:

Fixed Interconnection Structures. Each PE module is connected to a fixed number of neighbouring PE modules through a set of CE modules.

Bus Interconnection Structures. All PE modules are interconnected through a single CE module. This structure facilitates direct contact between any two PE modules. Any type of logical interconnection structure can therefore be realized with the Bus Interconnection Structure. Because all transfers are done on the same physical transmission medium, there will obviously be a bandwidth penalty compared to a corresponding fixed interconnection structure.

Reconfigurable Interconnection Structures. By reprogramming the appropriate CE modules, the PE modules can be configured through dedicated links to realize any one of several fixed interconnection structures. The requirement that the interconnection is done by dedicated links excludes the bus interconnection structures, the most reconfigurable of all structures, from this category.

By combining the two classification criteria for the processing elements and the interconnection topology, respectively, a global taxonomy for multiprocessing computer systems can be expressed. As a system classification scheme, such a taxonomy has a number of advantages compared to the other approaches described:

- The taxonomy is symmetrical as well as hierarchical, with the type of processing elements as the first level and the interconnection topology by which they are interconnected as the second level.
- All multiprocessing architectures can be included in this classification scheme without having to break with its basic idea.
- It reflects the overall, principal system architecture more than, in this context, implementation details of lesser importance.
- It focuses on the physical backbone of the system rather than on its current configuration (may be one out of several possible). Thereby, the basic capabilities of the systems in terms of performance as well as possible applications are easy to evaluate.

5.1.7. Skillicorn's taxonomy

Skillicorn's approach to defining a computer architecture taxonomy [Skillicorn 88] has many similarities to [Yalamanchili 85]. Both propose a two-level hierarchy, reflecting a functional view of the architecture and on how information flows between the processing elements. Skillicorn, however, combines Yalamanchili's both two levels into one. By including the processing elements as well as their interconnection, an abstract model of the architecture is formed. This makes it easier to focus on the essence of a particular architecture, without hiding important issues behind, in this context, irrelevant details.

Any abstract computer architecture model can as suggested by Skillicorn be built from four types of functional units:

- An instruction processor, acting as an interpreter for instructions.
- A data processor, performing data transforms, usually as a sequence of basic arithmetic operations.

The instruction and data processor as a unit is equivalent to what usually is called a processing element.

- A memory hierarchy, an intelligent storage device that passes data to and from the processors.

The term *intelligent* implies that the memory hierarchy may have its own controller. *Hierarchy* means that the functional unit can be regarded as a collection of one or several levels of memory storage with different "near-ness" to the processors. The term near-ness in this context relates to parameters such as ease of access and access time. The model may have one or a number of memory hierarchy units, facilitating local as well as global memory models.

A fourth type of functional unit is introduced to be able to describe how these three functional units are interconnected:

- A switch, an abstract device providing connectivity between the instruction and data processor(s) and the memory hierarchy(ies).

To include pipelined execution in the abstract machine model, each functional unit is labelled as one out of two types: "simple" or pipelined. In this way, pipelined behaviour, at least on the intra-functional unit level can be described without adding any new functional units.

To distinguish variants or different implementations of a given abstract architecture, a second level in the hierarchy is included. Parameters relevant to this level are the kind of technology being used, implementation size etc. To draw an analogy to programming, the second level can be regarded as the machine seen from an "assembly language" programmer's point of view.

As already mentioned, Skillicorn's model is very similar to Yalamanchili's. In fact, it can be regarded as an augmentation. From the point of view of my *purpose* of trying to find or describe a suitable computer classification scheme, I find Yalamanchili's is best. There are two reasons for this:

- In my view, a multiprocessing system's interconnection topology is a very important (if not *the* most important) aspect of an architecture. The

interconnection topology is more emphasized in Yalamanchili's model than in Skillicorn's.

- The greater level of detail makes Skillicorn's model better suited as a classification tool for existing architectures than a structured method for searching for computer architectures with certain properties, which is my intention of looking into architecture taxonomies.

5.2. A new taxonomy

As mentioned earlier, the whole purpose of going through these different architecture taxonomies has been to find, or if necessary specify, a classification scheme to be used as a tool for selecting the right architecture to a given application. With that in mind, the hierarchical model introduced by Yalamanchili represents in my view the best approach to this.

Yalamanchili's scheme has one shortcoming, however. It only covers the aspect of system communication at the lowest, physical (topological) level. That is, how the various modules are physically inter-connected. But system communication is influenced by far more than the physical interconnections, they only represent roads on which it is possible to drive. The *need* for driving, however, is determined by the stream of data and control information through the system. Therefore, something must be said about the memory and control structures as well. Yalamanchili's hierarchical model should therefore be augmented with two more levels to cover the gap between the PE structure and the interconnection topology:

Memory structure

Seen from an image or signal processing point of view, which is the actual application as far as this work is concerned, there are two basic configurations of system memory: Shared or distributed.

In a **shared memory** structure, "system data" are conceptually stored in one large memory structure, accessible in some way or another by all processing elements, PEs. The term system data means in this context image data as well the control data structures necessary to handle the shared memory mechanism. The "transfer" of data between two or more PEs are done by exchanging pointers, no physical movement of the array of data itself is involved. Whether the receiving PE chooses to actually transfer the data, as one or several blocks, into a local, on-board memory for easier access during processing, is a matter of implementation and does not make it a distributed memory system.

Synchronization in a shared memory system is done by some sort of mailbox mechanism: By writing a message, using a pre-defined format, into a predefined location in the (shared) system memory, the sending module notifies the receiving module(s) that new data are available. The location of the actual data will either be predefined or a part of the mailbox message format. The mailbox mechanism is often supported by a so-called software interrupt, an interrupt is then issued whenever the mailbox location is written to. Otherwise, the receiving module(s) must poll the mailbox at regular intervals to see if new data has arrived.

In a **distributed memory** system, there is no single memory containing all system data. Instead, the storage of data is distributed over the PEs. Each PE then has a local, on-board storage of the data that this particular PE needs to do its processing. A distributed memory scheme in an image processing system therefore inevitably involves the physical movement of large amounts of data, especially if the same data are to be copied to a number of different receiving modules.

Synchronization in a distributed memory system is done by message passing: Messages of predefined formats are transferred between the modules involved, either in the way that the receiver module is notified that new data is about to arrive (issued by the sending module) or as a request for new data (issued by the receiver module). Messages are transferred directly between the local memories of the PEs involved, not through any global mailbox system. The messages can either be transferred as a part of the data or as a separate entity. In the latter case, they may be transferred on the same physical interconnection medium as the data, or they may be transferred on a separate interconnection medium dedicated to control purposes.

Control structure

As far as system control is concerned, there are two alternatives: Centralized or local control.

Centralized control implies, as the name says, that all PEs are controlled by a common controller module. The meaning of the term "controlled by", however, will to a certain extent be context dependent and therefore needs some further explanation. In a computer system consisting of several processing modules, control exists on two levels:

- Control of processing activity (instruction execution) within each module (intra-module control).
- Control and coordination (i.e. synchronization) of processing activities between modules (inter-module control).

One possible implementation of a homogeneous PE system is that all PEs execute the same program in complete lockstep with each other. This will be the case for a SIMD classified system [Flynn 66]. The PEs will then probably be of relatively low complexity with a limited set of instructions. For such a system, the term control means "program control", the control of instruction execution.

Another type of a homogeneous PE system will be a system where high performance is aimed at by distributing the total processing load over a number of identical PEs (CPUs). The PEs will execute separate parts of the task's code and must therefore obviously have their own program controller. In that case, the term "control" is therefore equivalent to inter-module (e.g. synchronization) rather than intra-module control.

In a heterogeneous PE system, needless to say, lockstep execution of programs is no viable mode of operation. When dealing with heterogeneous PE systems, therefore, the term control will unambiguously refer to inter-module control and synchronization.

It may be objected that since the control structure classification criteria is context dependent represents a weakness to the whole classification scheme. In my view this is not the case. It can easily be handled by augmenting the homogeneous PE branch to explicitly take care of the two types of homogeneous PE systems, without disrupting the classification scheme as such.

To sum up, computer architectures suitable for image parallel processing will now be investigated according to a four level, hierarchical taxonomy. These levels are:

The types of Processing elements. Homogeneous or heterogeneous.

Memory structure. Shared or distributed.

Control structure. Centralized or local.

Interconnection topology. Fixed, bus or reconfigurable network.

Finally, it should again be emphasized that the sole purpose of developing this computer taxonomy was to provide a tool or an investigation scheme to be used in the process of finding and discussing a suitable (system) architecture for the application in question. Because the developed taxonomy is qualitative rather than quantitative in nature, it will however *not* be used to work out a performance comparison between the various architectural candidates. To do this, a load model for the application must also have been developed, in itself a comprehensive task due to the large amount of flexibility and iterative, data dependent processing allowed.

CHAPTER 6.Selecting an appropriate architecture

Generally, a state-of-the-art system level architecture for real-time image processing should confirm to the following requirements:

High performance

As far as the ability to do a specific job within a specific time is concerned, performance is the key issue. But, performance in this context means more than just processing power from a single device or a processing element expressed as a number of MIPS or Mhz clock rate. Real-time image processing applications are so complex and have so high performance requirements that they must be supported by a *system* of processing modules. The actual performance achieved from a given architecture is therefore heavily influenced by how well the architecture is adapted to the application. That is, how well the (system) algorithm is mapped onto the architecture. Performance is therefore also a function of the way the processing modules are interconnected, how they communicate with each other and the bandwidth of this communication. Additionally, performance is also strongly influenced by the degree of success by which the total processing load is partitioned and distributed over the processing modules.

Flexibility

Flexibility with respect to which operations the system is able to perform, is important from three reasons:

First, developing a real-time image processing system is a very expensive project, in terms of time as well as personnel. It will therefore be of great advantage if the final product has as many potential applications as possible. This will improve the chance of eventually getting the development money back, which again will improve the chance of the project being started in the first place.

On the non-commercial side, it is important to have an architecture that is as flexible as possible simply because not all applications, or the entire processing structure, needed to serve a given application can be predicted at the time of system development. There must always be made room for "surprises" and new ideas.

What we have been talking about so far, is flexibility as far as a system's *static configuration* is concerned. That is, how a system is configured to conform to a specific application or set of applications. This configuration job is done either by the system manufacturer or the personnel running the system. A *dynamic configuration* capability, however, requires that the system can be reconfigured "on the fly", with the system up and running and without the manual assistance from an operator. Physical reconfiguration by changing cards or moving cards to other backplane positions is then prohibited. Dynamic (re)configuration therefore requires that the system's processing structure is programmable rather than determined by its physical structure.

Scalability

The term "scalability" is used to describe how well the system's performance scales with the addition of more processing elements. Ideally, the relation between system performance and the number of processing modules should be linear. When this is not so in nearly all cases, it is due to several reasons:

- To obtain a linear relation, the added PE(s) must be perfectly balanced with the original PEs, no PE must be idle at any time. In practice, this is very difficult to achieve.
- Further, adding PEs means that the total computation task is even more divided, increasing the amount of necessary inter module communication. This increased communication need goes for exchanging data as well as synchronization and control information.

Consequently, in addition to reducing the effect of the added PE(s), the increased communication will also lower the performance of the PEs already in the system.

> *This is an inevitable effect of a multiprocessing system's nature and can therefore not be avoided.*

The most important issue as far as poor scaling is concerned, however, is a product of bad architectural design: An unsuitable communication system, introducing saturation effects when adding more processing elements. To avoid this, high performance system communication should not only be enforced through raw transfer power in terms of Mhz and wide datawords but equally important through efficient and flexible interconnection and transfer schemes.

Fault tolerance

In the world of computer science, "fault tolerance" is a field of its own, having its own nomenclature. We will therefore first define some of the established terms used when describing fault tolerance characteristics of computer architectures ([Johnson 1989]):

Fault

> A physical defect, imperfection or flaw that occurs within some hardware or software component.

Error

> The manifestation of a fault. Specifically, an error is a deviation from accuracy or correctness. The result of a fault is therefore an error.

Failure

> Non-performance of some action that is due or expected, or the performance of some function in a subnormal quantity or quality. Therefore, if an error results in the system performing one of its functions incorrectly, a system failure has occurred.

The cause-and-effect relationship between faults, errors and failures are shown in: *Faults* results in *errors*, and errors may lead to system *failures*.

Figure 6.1. Faults, errors and failures

A *fault tolerant* system means that the system is able to continue its operation in a more or less degraded way even if an *error* occurs with a system component, without leading to system *failure*. This implies that architectures relying on single, non-replicated components vital to system operation should be avoided. Examples of such components will be centralized system and bus controllers.

However, fault tolerance must always be paid for in terms of added complexity and cost. By nature, the electronics added to achieve fault tolerance will be more or less redundant during normal system operation. Consequently, they will therefore not contribute to the system's performance. By sacrificing the fault tolerance requirement, this electronics can often be applied in another way, to increase performance. When feasible, the best of those two worlds are of course to be able to take advantage of the added electronics during normal system operation (increased performance), allowing the performance to be reduced in case of a component (module) error. This is, however, not always possible. Due to this frequent trade-off between these two properties, fault tolerance and performance, the following question must therefore be answered:

> *Given the actual application, how tolerant does the system have to be against component errors?*

Obviously, systems for aeroplane control or supervision of oil production will have an entirely different requirement as far as fault tolerance is concerned than an imaging system for medical ultrasound diagnostics. In the latter case, a failure will have no other implication than that the examination will have to be restarted, or in the worst case, be disrupted. No danger either for man or material will be involved. This fact should therefore also be reflected in the level of fault tolerance chosen, which I for this application will try to express as follows:

> *There should be no single component or module in the system who's error would cause the operator to totally lose contact of the system. In case of a failure, system diagnostics can then still be run, the fault identified and corrected. To the greatest possible extent, retrieval of system status (including image data) prior to the failure should be possible, facilitating system restart at the point of failure after the fault has been corrected.*

With the requirements of performance, flexibility, scalability and fault tolerance in mind, what would then be a suitable architecture for a real-time, image processing system aimed at medical ultrasound diagnostics? Using the classification tool developed in the previous chapter, the following architectural sketch can be made:

6.1. The types of processing elements

As explained earlier, the processing elements can be either homogeneous or heterogeneous.

Homogeneous processing elements means that all elements are equal and of the same type. Consequently, they can be interchanged without altering system operation. Homogeneous PEs are mostly used in general purpose multiprocessor systems to achieve increased performance through load distribution and balancing. Failure of one processing element will only result in a performance reduction, the degree of that reduction being smaller the higher the number of processing elements is.

Heterogeneous processing elements are used in systems where each element has its dedicated function and is specially designed with that in mind. Whatever design technique appropriate can therefore be used to optimize each and every element with respect to its dedicated function. Because each PE has its own place in the logical processing chain, the failure of one element will break the chain and corrupt system operation.

From the specifications outlined in chapter 4, it should be fairly obvious that a system consisting of dedicated, specially designed processing elements will be the most appropriate basis for a real-time system of this kind. Due to the real-time requirement and the large amount of data involved, the various algorithms involved must be implemented by dedicated hardware as look-up tables and custom designed integrated circuits rather than as programs running on some sort of CPU. To implement the system by using general purpose processing elements will therefore be a very inappropriate way to do it. To obtain the performance required, each subtask (algorithm) in the system must then be distributed over a number of processing elements, resulting in an unnecessary large and complex system compared to the dedicated PE approach.

> *The system will therefore be based on a collection of dedicated, specifically designed processing elements.*

6.2. Memory structure

As far as how the system's memory structure is concerned, there is basically two alternatives: Shared and distributed memory.

6.2.1. Shared memory

The shared memory minimizes, at least in theory, the need for transferring data over the system interconnection network and thereby reduces the risk for performance degradation due to network contention. This is because data are "transferred" from one processing element to the next by the exchange of pointers rather than physically moving the bulk of data. However, this implies that data are read from the shared memory into the PEs during algorithm execution, using word-by-word transfer modes far less efficient than transfer modes available for bulk data

transfer. Of course, the inefficient word-by-word transfer can be avoided by transferring data from the shared memory to the PE in blocks, e.g. containing one column or row of image data. The local PE storage will then function as a sort of cache for the data in the shared memory. Due to the regularity of the data access patterns of most image and signal processing algorithms, there will in many cases be possible to have the data ready in the local PE memory when needed by the processing algorithm. In this way, processing can be done at full speed without being delayed by slow transfers from the global, shared memory.

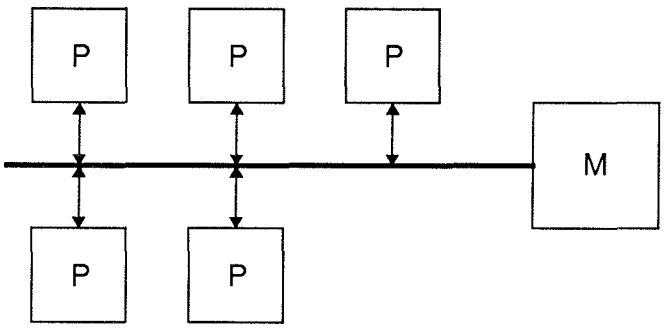


Figure 6.2. Shared memory

This scheme will function well under the following conditions:

- A regular access pattern, facilitating the need for specific data to be predicted long enough in advance to transfer the data from the shared to the local memory before they are actually needed by the processing algorithm.
- "Neighbourhood" processing. Only a small portion of the total amount of data is needed at a time. The size of the local buffer will grow with the size of the neighbourhood.
- Each set of data is needed by one PE only. If not, the problem of cache coherency will occur.

The last point is specially important: If the same data are to be used by several PEs, a PE cannot write the data back into the same location in shared memory after processing. The result would be an inconsistent data set, containing both processed and unprocessed data. There are two generally applicable solutions to this problem: static and dynamic partitioning of the shared memory.

Static partitioning

The shared memory is in this case partitioned into a set of separate memory segments. Each PE is allocated its own segment into which output data generated by that PE is written. If a PE is generating more than one type of output data (e.g. a processed image as well as a set of parameters describing some features of that image), one segment for each data type must be

allocated to these PEs. The segments should therefore be of variable size. This partitioning of the shared memory will then represent a more or less one-to-one map of the logical processing structure of the whole system, with input and output memory segments corresponding to the communication paths between the different modules. In other words, it is implementing a distributed memory structure by the partitioning of one, globally accessible shared memory module. If iterative processing is to be supported, however, the exact logical processing structure, and thereby the number as well as the "connectivity" of the memory segments required, will not be known before run-time. To take care of the time-varying memory segment requirements created by iterative processing, the static partitioning scheme must therefore be supplemented by a dynamic memory allocation mechanism.

Dynamic partitioning

In a dynamically partitioned, shared memory system, the entire memory module is regarded as one big pool of memory where segments are allocated and deleted as required. Segments are naturally allocated on request by the PE producing the data by which the segment is to be loaded. Segment deletion, however, is more difficult to handle. In the most general case, the set of data contained in a particular memory segment will have several consumer PEs. Further, in case of iterative processing, each consumer PE may need to read the set of data several times. A mechanism should therefore be included telling if all the consumer PEs have finished reading the data or not. The actual segment deletion could then either be performed by a special "house-keeping" task or the consumer PE being the last to issue the "finished" signal.

To be able to identify the type of data contained in a specific segment, as well as in which segment to find a specific set of data, a data structure containing these kinds of informations must also be maintained as a part of the dynamic partitioning scheme. Needless to say, a complex logical processing structure with iterative processing support may yield a very complicated, and unpredictable, segment structure.

Additionally, because the data transfers are initiated by the PE requesting data, the same data must be transferred as separate transfer operations to all PEs requiring these data.

6.2.2. Distributed memory

In a distributed memory system, each PE has its own local memory storing data during algorithm execution. This local memory is typically a dual-port memory with one port connected to the system network and the other port to the PEs local bus. Because the data transfer in this case can be initiated by the PE generating the data rather than by a PE requesting them, data can be transferred to all PEs needing the data as one single transfer operation, using

an appropriate broadcast or multicast transfer mechanism. In practice, the load on the system interconnection network may therefore be less in a distributed memory than in a shared memory system.

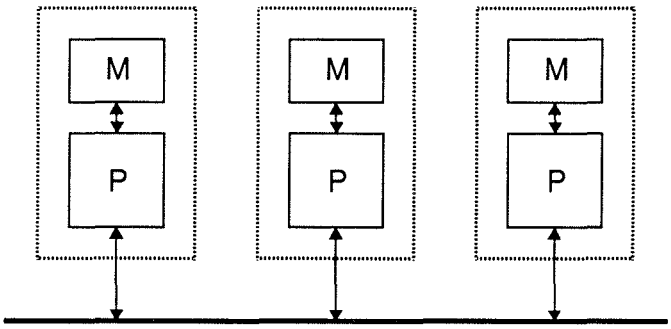


Figure 6.3. Distributed memory

The only possibility I can see for a shared memory structure in a system of this kind, is if each PE sharing the memory is connected to it through its own communication channel, facilitating the memory to be accessed as easy as if it was local to the individual PEs. This will, however, be prohibitive in terms of cost, complexity as well as lack of flexibility. In a distributed memory system, data will therefore be "nearer" to the PE during processing. Thereby the access of data will be faster as well as purely local to the PE: No complex synchronizing and protecting mechanisms are then necessary if several PEs must have access to the same set of data, simply because they will all have their own copies. Consequently,

the system will be based on a distributed memory scheme.

6.3. Control structure

In a distributed memory system consisting of a number of heterogeneous Processing Elements (PEs), each PE will contain its own control unit and execute its own program. To perform according to a specified algorithm, these PEs must be orchestrated into a complete task force, requiring some sort of master control. This master control can be either centralized to a single, dedicated module or distributed over a number of modules.

In a **centralized control** system, all task synchronization and communication are performed via the central controller module. Depending on which module, the controller or the PE, taking the initiative to the action in question, this is done in two ways:

Controller driven

In this case, the controller is either able to monitor the state of the different PEs directly by the means of dedicated backplane signal lines or it interrogates PE status variables at regular intervals. Based on this "map" of system status, the controller will initiate the appropriate actions.

PE driven

> With this approach, the PEs themselves will take the initiative to inform the controller about (a change in) their current status. Based on this information, the controller will effectuate the necessary synchronization and communication actions.

In a **distributed control** system, there is no dedicated control module, at least not on the (higher) levels of control we are discussing here. Instead, every PE will be equipped with the means and mechanisms needed to communicate directly with the involved PE(s), that be exchange of control information for synchronization purposes or transfer of data. The use of a central controller for lower levels of control (e.g. bus arbitration and global buffer management) is, however, not precluded by a distributed control scheme for task synchronization and communication.

As emphasized by [Saponas 1980] in his description of the FDPS system (Fully Distributed Processing System), it will be very difficult for a central controller to maintain a consistent and deterministic view of the status of the different PEs at all times and during all processing phases. This will be necessary for the central controller to do its job. Further, this status view collected by the controller must be communicated down to the involved PEs in conjunction with synchronization and communication operations. To guarantee all PEs at any time to have a consistent view of the entire system will therefore, due to the amount of explicit synchronization necessary, imply a significant reduction in performance compared to the situation where the PEs were allowed to "run free".

As far as *system scaling* is concerned, the number of PE *pairs* being potential candidates for synchronization and communication will grow exponentially with the number of PEs. So will also the complexity of the centralized control mechanism. The centralized control approach will therefore scale very poor when the number of PEs increases. Distinguishing between the controller and the PE driven control schemes, this will especially be the case as the control driven scheme is concerned, which is the "purest" centralized scheme of the two. The number of PEs in a centrally controlled MIMD machine will therefore be very limited.

Another aspect of central vs. distributed control is *fault tolerance*. A central controller module being responsible for all task synchronization and communication in the system will introduce a very vulnerable point in the architecture. Although fault tolerance should not be exaggerated as far as this application is concerned, it is nevertheless another factor in favour of a distributed control scheme.

When it comes to the implementation of an actual communication mechanism, distributed systems are in nature very related to the *message passing* scheme. A feasible alternative to that would be to do the communication through a set of shared variables, contained in a common, globally accessible memory. In addition to requiring a dedicated memory module with an integrated resource manager controlling the access of that module, the number of required variables will grow very fast as the number of PEs increases. The reason for this is that due to the dissimilarities of the PEs, each (or at least many) PE pair combinations may require their own set of variables. Another potential obstacle is multicast and broadcast transfers, requiring

a number of shared variables, belonging to the PEs to be involved in the transfer, to be updated/ interrogated "simultaneously". To support this, the shared variable access arbitration mechanism must be augmented by a higher level access control scheme, allowing several variables to be read or written by one module without being interrupted by other modules. To ensure this, some sort of *test-and-set* or *semaphore* mechanism is required, altogether leading to a mechanism far more complex than that of the message passing scheme.

The conclusion as far as the system's control structure is concerned must therefore be:

> *The system is to be based on a distributed control scheme where the different PEs themselves, through direct PE-to-PE message passing, are responsible for taking the necessary actions in conjunction with task synchronization and data transferring.*

6.3.1. Control concept

Up to now, it has been determined that the architecture we are seeking should have the following key features:

- Heterogeneous Processing Elements (PEs).
- Distributed memory.
- Distributed control based on a message passing communication scheme.

As far as selecting a scheduling and control strategy for a system architecture having these features, there are two alternatives: Data driven or demand driven.

6.3.1.a. Data driven architectures

In a data driven architecture, the natural flow of data is controlling the operation of the system. Data driven architectures are therefore also called dataflow machines. The key concept of operation of a dataflow machine [Srini 1980] is that

> *the execution of commands (tasks) are determined by the availability of the required data (that is, the input operands), and not by the explicit sequence as specified by some traditional computer language program.*

Dataflow computation has for long been a well established region within the field of computer architecture. It can be said to be based on the following two principles:

- **Asynchrony**. All operations are executed when and only when the required operands are available.
- **Functionality**. All operations are functions; that is, there are no side effects.

The first denotes an execution mechanism where the "program" to execute is specified as a graph connecting a number of nodes. Each node, representing a particular operation, will receive its operand data on its input arcs and produce results out on its output arcs, being in turn

connected to the input arcs of other nodes. As far as the second principle is concerned, it implies that any enabled operations (that is, operations having their input data ready) can be executed in either order or concurrently.

Basically, there are two types of dataflow architectures:

Static architectures

> All nodes of a program graph are then loaded into memory before the computation begins, and at most one instance of a node is enabled for firing at a time.

Dynamic architectures

> A dynamic architecture facilitates simultaneous firing of several instances of a node, being created at runtime. A loop can thereby be unfolded at runtime by creating multiple instances of the node representing the loop body. These instances can be executed concurrently.

Further, depending on the grain of computation, dataflow machines can be divided into two groups: instruction level and task level machines.

Instruction level dataflow machines

Initially, the dataflow concept was introduced to exploit parallelism on the instruction level in "ordinary" computer programs: By assembling all input operands required by one instruction into a packet and tagging that packet with the type of the instruction (add, multiply etc.), a scheduler could assign each packet to one out of a set of parallel processing units.

The best known examples of this kind of dataflow computers are the Manchester machine ([Gurd 1980/1 and 2] and [Watson 1982]), the MIT TTDA computer [Arvind 1987], TIP ([Hanaki 1982] and [Guerra 1985]). Today, the largest activity as far as fine grained dataflow computers are concerned is may be taking place at the Electrotechnical Laboratory (ETL) in Japan, known for its SIGMA-1 and EM-4 computers [Kahaner 1990].

The problem with the fine-grained dataflow architectures, however, is that the overhead required for assembling and scheduling the packets tends to outweigh the speed gained by the parallel execution. The research effort put into fine-grained dataflow architectures therefore seems to have decreased somewhat during the last years.

Task level dataflow machines

Instead, the activity has turned to macro dataflow architectures, partitioning the computational problem into a set of tasks rather than basic instructions. The nodes in the dataflow graph then represent asynchronous tasks and the arcs connecting the nodes represent communication paths for the messages (tokens) generated by the nodes or supplied by the external environment. The dataflow principle is then used for managing these set of asynchronous tasks into a working computer, solving a specified problem. Compared to instruction level dataflow machines, the node firing rules will be more complicated for a task level machine. This is due to the fact that the basic operations in this case may be relatively complex. There will therefore often be possible to start execution before all operands are available on the input arcs, which is a

fundamental rule in instruction level dataflow machines. In addition, the concept of *streams* should be included. A node then does not have to wait for an entire data structure to arrive, but may process (fire) as the components of the stream arrive on the input arc. In this way, the use of streams introduces another level of asynchrony in the system. Examples of task (or macro) dataflow architectures are TOPPSY [Engbersen 1983], DDM [Srini 1986], EDFG [Srini 1986], PDFP [Sawkar 1983] and DCS [Srini 1985].

6.3.1.b. Demand-driven architectures

In a demand-driven architecture, it is the *need* for data, and not the availability which triggers an operation. Each Processing Element (PE) must therefore have a list of which data it needs to perform a specific task and where (that is, on which other PEs) these data can be found. When the PE then is ready to start execution, it must go out to these other PEs and request the data. If the requested data are not "raw" data already received from the environment, but the result of some intermediate (not yet performed) computation, this request will in turn cause a new request to be issued to the PE responsible for producing that result. In this way, an operation initiated by the system's environment (e.g. the operator) will cause requests for data to be propagated backwards through the whole system straight up the system's frontend where the raw data are acquired. Then data will start flowing forwards, following the same path as the requests but in the opposite direction, until they eventually reaches the user interface PE making the original request. A requesting PE must therefore very often wait until data becomes available. Examples of demand-driven architectures are presented in [Treleaven 1982].

Compared to data-driven architectures, a demand-driven machine will inherently imply a larger amount of execution overhead to perform the same computation due to the explicit generation of demands and demand propagation. In addition, the load on the communication system will be much larger. This is due to two factors:

- Demand and demand propagation messages.
- Less utilization of broadcast and multicast transfers.

In an application of the kind we are dealing with in this thesis, the same data will often be needed by several PEs. The point in time when the need for these data actually occurs, however, will most probably be different for the different PEs. This means that the transfer of a set of data which in a data-driven machine would have been performed as one multicast transfer, in a demand-driven machine will be done as several single destination transfers, creating extra (and unnecessary) load on the communication system.

In this author's opinion, a demand-driven scheduling strategy is only suited for applications where the producer of a set of data does not know whether there is actually any use for these data or not. If it is not, a demand-driven strategy will then prevent an otherwise unnecessary computation and transfer from taking place. For a real-time system, however, this is not an actual approach because it is indeterministic in terms of computing time.

To make the picture complete, however, it must also be said that a demand-driven architecture has one advantage over a data-driven architecture: For the system regarded as a whole, it will most probably require less buffer space. This is because the modules in a demand-driven system

will only receive data when they are ready to process it, no buffering on input is thereby required. As far as output is concerned, however, the modules are responsible for keeping the data until they are fetched (consumed) by the next modules in the processing chain. Data must therefore be buffered on output. For data-driven architectures, the situation is opposite: Data must be buffered on input, not on output. Because most processing operations will imply a data reduction of some degree, or at least a one-to-one transformation from input to output, the required size of an input buffer will in most cases be larger than the size of the corresponding output buffer. It can therefore be argued that a demand-driven architecture will require less buffer space than the equivalent data-driven architecture.

In an attempt to combine the best features from the two models, the lesser amount of overhead associated with the data-driven model and the prevention of useless value calculation and transfer related to the demand-driven model, hybrid systems have been proposed [Jagannathan 1984]. In a hybrid system, the most appropriate of the two models are selected on an individual sub-computation basis: Whenever possible (and useful), sub-computations are data-driven, only sub-computations that are only potentially useful are demand-driven. But again, the indeterministic nature represented by the term "potentially useful computation" has no place in a real-time system. For a system for off-line data post-processing, however, a hybrid model could prove useful: A module doing image analysis may, depending on the actual quality of the image data, need more or less statistical data, produced by another module, to support its analysis. The co-operation between these two modules, the analysing and the statistical, may in this case be demand-driven.

As far as the real-time part of the ultrasound system is concerned, however, being the subject of this thesis, the conclusion must be:

the scheduling and control strategy is to be based on a data-driven model.

6.4. Interconnection topology

The interconnection topology is the backbone of any processing system. No matter the processing power of the individual modules, the performance of the total system will suffer if the way the modules are interconnected is not matched to the communication pattern of the application. Among all parameters determining a system's performance and "suitability" with respect to a given application, the interconnection topology is may be the most important.

When trying to describe a given interconnection topology, this is often done according to a number of key characteristics. Some of these characteristics will be universal in the sense that they are valid and applicable for any communication system, others will be application specific. That is, they will only make sense for the specific type of system in question.

Universal features by which an interconnection topology can be characterized are:

Transfer rate

The maximum rate by which data can be transferred from one module to another, usually measured as a number of MBytes/ second.

Scalability

A measure, or may be a qualitative expression, of the topology's ability to support an increased number of modules. Ideally, performance should grow linearly with the number of modules in a multi-processing system. Due to required communication overhead, synchronization as well as saturation of the communication channels, this will never be the case. On the contrary, systems operating at the limit of their capacity may experience a *reduction* in the overall performance when adding one more module to the system.

Flexibility

Just as important to a system's total throughput as the transfer rate of its individual communication channels, is its ability to adapt to the processing structure of the application in question. More specific, this adaptability can be related to

- Smooth scaling, up as well as down. Does the interconnection topology support any number of modules (at least up to a certain limit), or must the system be augmented/ reduced in steps of multiple modules?
- How well is the topology suited for hierarchical partitioning? As the size of the computational problem is growing, it is often advantageous to partition the total system into a hierarchical structure.
- Communication channel utilization. Are the communication channels statically assigned to specific modules, or can they be regarded as a pool of resources, allocatable on request? If they are statically assigned, and this assignment is not well matched to the communication pattern of the application, performance will suffer.
- Symmetry. To achieve maximum flexibility, all modules should have the same communication capabilities, no matter their position in the total system. To some extent, this may represent a conflict to the feature of hierarchical partitioning.

Fault tolerance

Fault tolerance is always an issue as far as multi-processing systems are concerned. As already discussed, however, the resources put into fault tolerance must always be viewed in the light of which consequences a fault may have, and how the same amount of resources otherwise may be used, e.g. to increase the system's performance.

Simplicity

Needless to say, a simple system will both be cheaper and more reliable than a complicated system.

All these features are selection criteria generally valid for any computer system. Because, however, the application is known and a coarse outline of the system architecture already have been sketched (heterogeneous PEs, distributed memory and data flow control), some application specific features can be added to the universal ones:

Pipeline execution

The total computational job, from the acquisition of "raw" data from the ultrasound transducer up to the display of those data can logically be divided into a set of sequential, more or less independent, sub-tasks. The processing of these tasks may therefore be overlapping in time, organized as a sort of macro pipeline [Briggs 1982].

Multidestination transfers

Especially at the beginning and at the end of the processing pipeline, multidestination transfer support (multicast, broadcast) will be important to system performance as well as minimizing the load on the communication channel(s). In this respect, an interconnection topology where the data "passes by" all modules will clearly be easier to handle than a topology where data must be routed explicitly to each of the destination modules. Explicit routing is more complicated as far as channel allocation is concerned, and an efficient implementation may also require buffering on the intermediate nodes.

In view of all these features, both the universal and the application specific, the different alternatives for an interconnection scheme will now be discussed. Every alternative will have both its advantages and disadvantages. Because these are difficult to quantify, the individual features as absolute measures as well as their relative significance by which their quantitative rating should be weighted, the final selection must be based on a rather qualitative discussion.

Basically, there are two fundamentally different approaches to the implementation of a multiprocessor interconnection scheme: The *shared bus* and the *interconnection network*. In addition, not logically belonging to any of these, is the *multiport* approach.

6.4.1. Shared bus

The shared bus is probably the most common interconnection topology, both because it is simple and because commercially available modules always support a standard bus interface (see Appendix A). In a shared bus system, all modules (P) are connected to the same set of signal lines (the bus), by which both control and data information are transferred.

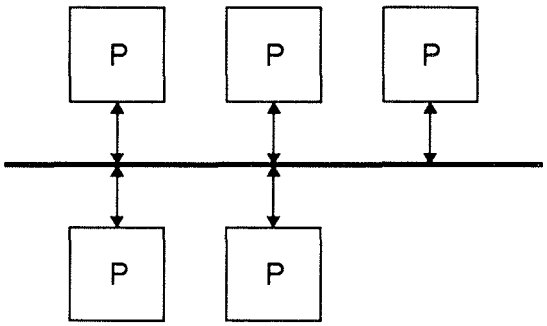


Figure 6.4. Shared bus

Because the bus is a resource global to all modules, its use must be regulated according to some set of generally agreed specifications (module priorities, response time requirements etc.). This can be done in two different ways.

The most usual approach is to regard the bus as a resource "always" being available to all modules. When a module wants to perform a transfer, it must ask for permission to use the bus by issuing a *bus request.* The request is either addressed to a dedicated bus module called the bus arbiter (*centralized arbitration*), or the modules resolve any conflicts themselves without the assist from a central module (*distributed arbitration*). If the bus is idle, a *bus grant* is momentarily issued (centrally or locally), signalling to the requesting module that it is allowed to start using the bus. If the bus is busy, the request is coordinated with any other pending requests according to their relative priorities and the specific arbitration algorithm being used. Eventually, when the bus becomes idle and no higher priority modules are waiting, the bus is granted to the requesting module.

The alternative approach is to divide a certain period of time into a *fixed number of timeslices.* Each processing module connected to the bus is then allocated its own timeslice, in which the bus is to that module's exclusive disposal. An example of such a system is the TAMIPS machine vision system [Viitanen]. The advantage with this approach is simple arbitration logic and implicit transfer source identification (through the timeslice in which the transfer is done). A significant disadvantage, however, is that it requires that the system's total need for communication is evenly distributed over all processing modules for the available bandwidth of the bus to be fully utilized. If not, modules needing the bus must wait for more or less empty timeslices to elapse before they can start transferring data. Even with all modules having equal communication needs, it is shown by [Bain 1981] that fixed timeslice arbitration incurs much higher waiting times than all other bus arbitration algorithms, especially during light load conditions. The bus is then idle most of the time and a bus request being serviced according to other arbitration algorithms (e.g. First Come First Served, Round Robin, Fixed Priority) will usually be granted immediately.

Speaking in general terms, the shared bus has many advantages: It is simple (and thereby cheap and reliable) as well as efficient. Because all modules have equal communication capabilities, it is also very flexible. The fact that all transfers can be simultaneously observed by all modules, provides an excellent support for multidestination transfers. Several image and signal processing systems are therefore based on the shared bus approach, among them the PICAP II system being described in [Danielsson 1980], [Kruse 1980] and [Antonsson 1981].

The shared bus has one significant disadvantage, however, and that is its poor scalability: As the number of connected modules increases, so will the electrical load on the signal lines and the maximum transfer rate will thereby decrease. Measured in terms of available bandwidth pr. module, this will be further reduced when the number of modules increases. To overcome this problem, several approaches have been taken. In [Vranesic 1991], a system tying a number of small bus sections together through hierarchical, bit-parallel rings is presented. Another solution is to use multiple buses, with either all or a subset of the modules connected to each bus. In the latter case, special gateway modules must be used to connect the buses to each other. An example of a multiple bus system is presented in [Hasegawa 1981], using two uni-directional buses for the two directions of transfer. As far as bus arbitration is concerned, the timeslice scheduling scheme is used.

6.4.2. Interconnection networks

In an interconnection network, the communication paths are provided by more or less direct one-to-one links rather than common access to one or more shared buses. Depending on its reconfigurability, an interconnection network can either be classified as *static* or *dynamic* [Hwang 1984].

6.4.2.a. Static networks

As suggested by its name, a static network is made up of dedicated, fixed communication links connecting the modules. Static networks can be classified according to the dimensions required for layout. An example of a one-dimensional network is the linear array, two-dimensional topologies include the ring, star, tree and mesh networks and the n-cube is a representative of the n-dimensional network. Of these, we will take a closer look at the linear array, ring bus, star and n-cube networks.

Linear array

In a linear array architecture, the global, shared bus is replaced by a number of dedicated one-to-one links, each link connecting one module (P) to its immediate left and right neighbours.

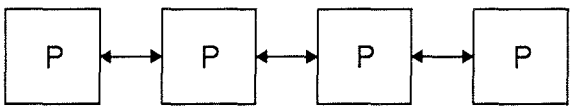


Figure 6.5. Linear array

Because the communication channel is broken at both ends, the links must necessarily be bi-directional. Compared to a system of uni-directional links, bi-directional links means lower speed and more complex control. To avoid this, the last module of the linear array may be connected back to the first. By adding this single link, we then have a 2-dimensional network, the

Ring bus

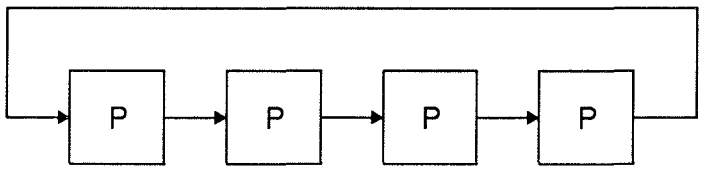


Figure 6.6. Ring bus

The communication path is now closed and we can therefore make the links unidirectional. Because the links are unidirectional, connecting one module (P) to one other module, using the word "bus" about this connection may seem somewhat misleading. Usually, a bus is used in the sense of a shared bus, describing a number of modules sharing the same set of signal lines through open-collector or three-state interfaces. The term "ring bus" is however to be used because it is a commonly accepted and recognized name of this type of interconnection.

Compared to a shared bus solution, the ring bus approach shares many of its positive characteristics:

- It is simple, implying low cost and high reliability due to the uni-directional communication link rather than a three-state bus interface.
- It is symmetric in the sense that all modules have equal communication possibilities.
- High degree of utilization even with a non-uniform pattern of communication.
- Very suitable for multidestination transfers, no explicit routing is required. The data are transferred to the most remote destination module while copied "on the fly" as they passes by into the intermediate destination modules.

A N-module ring bus system consists of N separate and independent communication links, connecting the N modules. The theoretical bandwidth of such a system will therefore be equal to the bandwidth of each link times the number of links (N). However, this will only be the case if all modules operate in a carefully balanced and tuned synchronism to each other. Or in other words, as a *SIMD* machine (Single Instruction Multiple Data Stream). In our case, with an architecture more resembling a free-running *MIMD*-machine with randomly distributed transfers between the different modules, the effective bandwidth will be considerably lower.

The exact figure will of course depend very much on the communication pattern and the applied load model, but works carried out indicates that it will be in the order of one [Gaillat 1983] to two [Spragins 1979] times the bandwidth of the individual links.

In addition, due to the simplified interface logic and the reduced electrical load of a ring bus link compared to that of a shared bus, the achievable transfer rate will be significantly higher. Compared to a "defacto standard" 21 slot backplane using equivalent technology, a speed improvement of two to three times is realistic. Multiplied with a concurrency factor between one and two, a total improvement in performance compared to a shared bus implementation in the order of 3 to 5 should therefore be achievable. Furthermore, the scaling properties of the ring bus are much better than that of the shared bus.

In this author's opinion, the ring bus has only one disadvantage compared to the shared bus: The larger *transfer latency*. In a N-module system, the average distance between any two communicating modules is N/2. That is, the transfer must as an average pass through (N/2-1) intermediate modules before reaching the destination module. The significance of this depends on the length of the latency time compared to the length of the total transfer time: For large (block) transfers, the latency time can be ignored while it for single word transfers will be prohibitive. In our application, this will represent no problem because both data and control information will be transferred as larger entities of data (blocks and messages, respectively). Besides, measures can be taken to reduce the transfer latency: By adding another ring bus transferring data in the opposite direction, the average transfer latency is reduced to N/4. A priority mechanism preventing large data transfers from blocking smaller control transfers could further be included.

Star network

In the star network, each module is connected to a *central server* (the centre of the star) via a bidirectional link. All communication must therefore go through the central server.

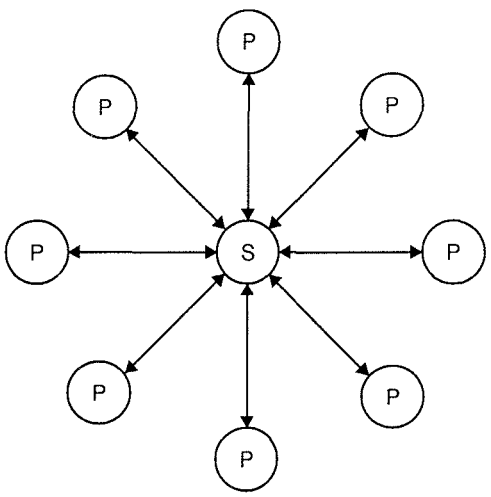


Figure 6.7. Star network

To perform a transfer between any two modules, two links are needed: The link between the source (requester) module and the server, and the link between the server and the destination module. While the status of the former is known to the source module, the status of the latter is not. Before a transfer can take place, the source module must therefore request the server for access to the destination module. In this way, the transfer procedure of a star network very much resembles that of a shared bus: Only one transfer can take place at a time (depending on the implementation of the server), and its use must be regulated by a centrally located resource. From a fault tolerance point of view, the star is probably somewhat better. Even if the star server necessarily must be more complicated than the corresponding shared bus arbiter (due to the switching elements), a single corrupt bus interface can not block the entire communication as in the case of the shared bus. Another advantage of the star is that the central server may be equipped with buffers, permitting high-speed source modules to transfer their data at full speed regardless of the (lower) speed of the actual destination module. The destination module may even be busy, temporal storage is then provided by the central server buffer. In this way, by enhancing the server from a simple switch into an intelligent buffer, temporal fluctuations in transfer activity can be efficiently handled. It is also well suited for multidestination transfers. The problem with the star network is that it does not match the layout of a conventional backplane very well. An high performance solution with minimum length communication links requires some form of cylinder shaped construction. Another difficulty is the large number of ports required on the central server.

n-cube

Another static network topology is the n-cube. A total of 2^n modules, each having n ports, are connected as a n-dimensional cube as shown in Figure 6.8 (n=3).

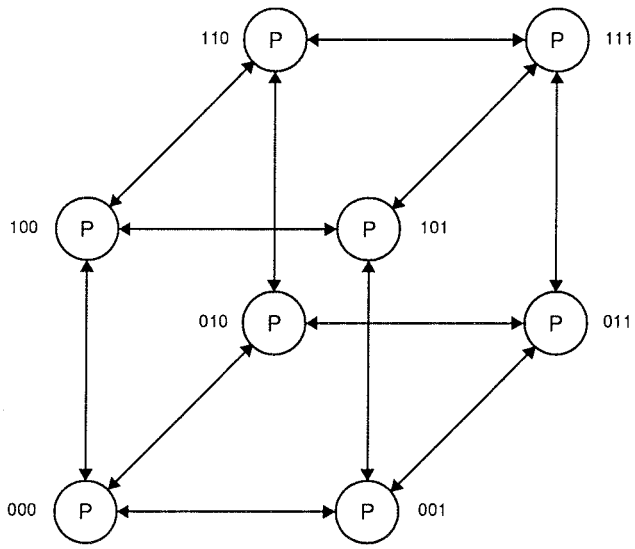


Figure 6.8. n-cube

In the n-cube, the addresses of any two connected modules will only differ in one bit, and the maximum distance between any two modules will be n.

As far as the last "hop" in the communication path between any two modules is concerned, there is n alternatives. Depending on the modules' relative positions, there is also a number of alternative paths up to this last hop module. Altogether, for a free-running multiprocessor machine with randomly distributed transfers, the n-cube topology represents a considerable routing challenge if optimum utilization (minimum transfer distance, maximum number of concurrent transfers) is to be achieved. The requirement of multidestination transfer support makes this even harder. On the positive side, the n-cube is from a fault tolerance point of view very good due to the multiple ports.

6.4.2.b. Dynamic networks

A dynamic network contains no direct module-to-module communication links. Instead, programmable switching elements is included in the communication paths. By changing the setting of the switches, the communication paths are rearranged. Depending on how this switching is implemented, we distinguish between *single-stage* and *multistage* dynamic networks.

Single-stage networks

A single-stage network is a switching network for inter-connecting a number of processing elements. To provide communication paths between N elements, N *input selectors* (IS) and N *output selectors* (OS) are required. Each input selector is essentially a 1-to-D demultiplexer ($1 \leq D \leq N$), while the output selectors are M-to-1 multiplexers ($1 \leq M \leq N$).

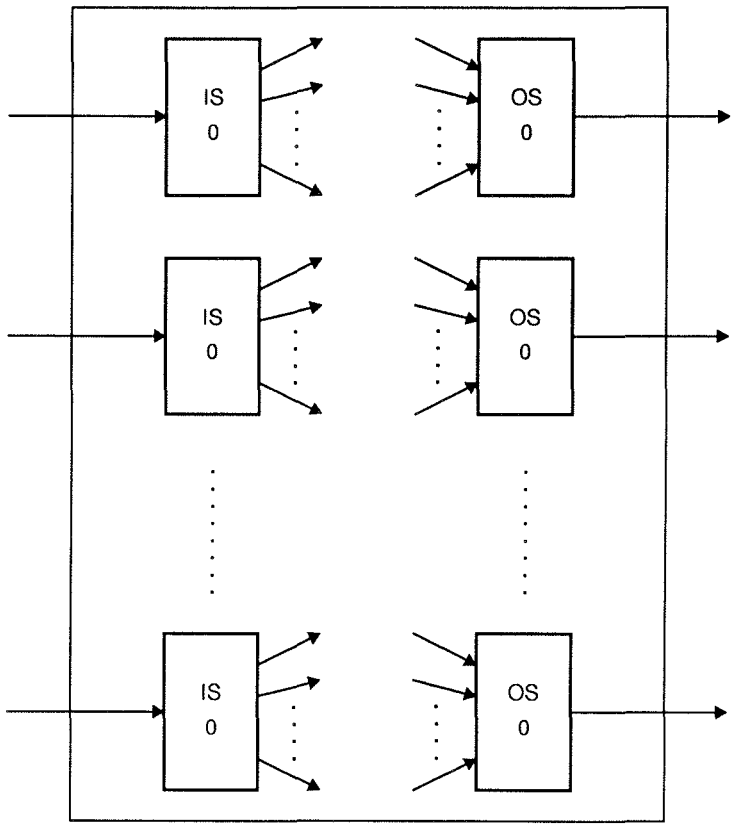


Figure 6.9. Single-stage network [Hwang 1984]

The larger the values of D and M are, the larger the connectivity of the network is. Full connectivity, that is a direct path between any pair of processing elements, requires D=M=N. This configuration is called a *crossbar* system. Because all possible communication paths then have their own dedicated communication channel, there will in a crossbar system be no waiting for an available communication channel (but the module at the other end of the communication channel, however, may of course be busy, causing the data transfer to be deferred). Examples of crossbar systems are the IDATEN ([Sasaki 1985], [Gotoh 1985]) and BASIS [Andersen 1990] machines.

Because there are no conflicts in a crossbar network, it is providing the highest performance of any interconnection topology. However, it has a very high cost which may be prohibitive already for a moderate number of modules. In addition, the high potential performance, required to justify the high cost, is only utilized in applications where the communication need is evenly distributed over the modules. Work has therefore been done in trying to achieve the same level of performance at a lower cost by using multiple buses [Lang 1982]. It is there shown

that under a certain set of assumptions, a N processor multi-bus system with a number of busses slightly larger than N/2, provides an effective bandwidth less than 5 percent smaller than that produced by the crossbar.

Multistage network

By implementing the switching function by a number of subsequent switch boxes rather than a demultiplexer/ multiplexer pair, we have a multistage network. Each switch is basically a 2-input, 2-output element with possible functions as shown in Figure 6.10.

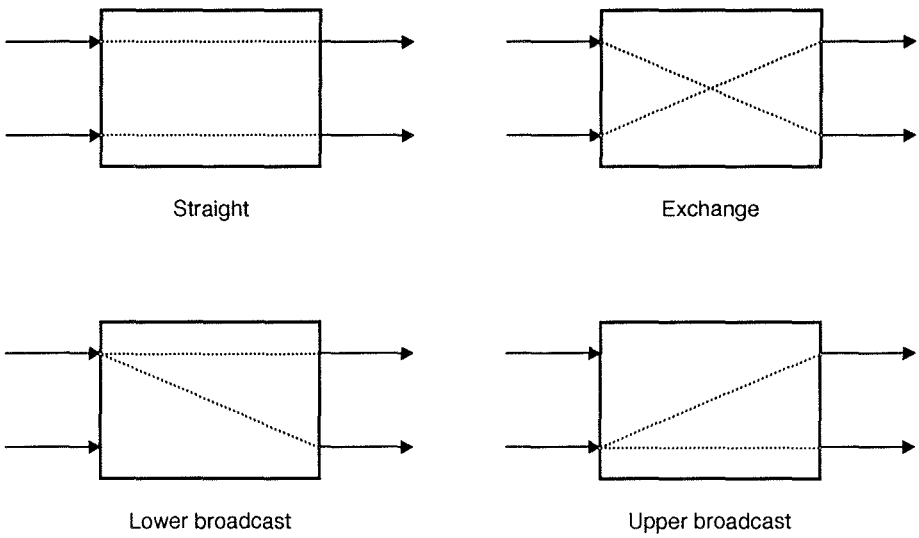


Figure 6.10. Multistage network switching element [Hwang 1984]

A four-function switch will support all four functions, while a two-function switch only contains the *straight* and *exchange* modes.

Obviously, an interconnection network, whether it is a single-stage or a multistage network, will be a very complex unit with a large number of connected signals, and with a correspondingly high cost. To justify this cost, such networks are therefore best suited for architectures requiring direct, "zero-delay" communication paths between the processing elements. In other words, SIMD machines, with the processing elements operating in complete lockstep with each other. For a system consisting of a number of free-running, heterogeneous processing elements, this cost and complexity can in the author's opinion not be justified.

6.4.3. Multiport

Like a crossbar system, a multiport machine is based on dedicated one-to-one communication channels between the different modules. The difference lies in the switching element, which for a crossbar system is realized as a separate, centrally located switch. In a multiport system, there

is no such switch. Instead, the output from all modules are connected to the input of every other module. In a N-module system, each module must therefore have a N-to-1 multiplexer on its input. An example of a multiport machine is the FLIP system ([Luetjen 1980], [Gemmar 1982]), consisting of 16 8 bit Processing Elements, with each element connected to all other by an 8 bit path.

Due to the similarity to the crossbar, all arguments applied to the crossbar are equally applicable to the multiport approach: High performance on the positive side, high cost and complexity on the negative side. Because of the removal of the central switch, a multiport implementation has better fault tolerance properties than the corresponding crossbar system. Like the crossbar, however, a multiport system requires the communication need to be evenly distributed over the modules to fully utilize the available bandwidth.

Conclusion

To make a conclusion, every interconnection scheme has its own vices and virtues: The simplest and most flexible is the shared bus, suffering, however, from its poor scalability. On the other side we have the multiport and interconnection networks, providing the highest possible performance but at a prohibitively high cost. A compromise between the two is the ring bus, with better performance, better scalability, better fault tolerance properties (but to a somewhat higher cost) than the shared bus. The ring bus' only disadvantage compared to the shared bus is its long transfer latency, making single word transfers ineffective. In our application, however, transferring data and control information as packets, this argument has little or no significance. The conclusion must therefore be:

The architecture is to be based on a ring bus interconnection topology.

When describing a complex system, necessarily by using of a lot of more or less familiar terms and expressions, it is important to be consistent in the use of words. Although it is generally considered good writing style to vary the language using synonyms and alternative expressions when possible, this is not a good idea as far as writing a system specification is concerned. A nomenclature explaining all vital words and expressions used throughout the ring bus specification part of this thesis is therefore presented in appendix C The reader is strongly encouraged to use this nomenclature for reference.

According to the discussion carried out in the previous chapter, the interconnection system selected for the real-time imaging system presented in this thesis will be a ring bus. Generally speaking, the interconnection system must provide mechanisms for managing two types of information flow: *Data* and *control*. The distinction between data and control information is based upon the art as well as the amount of the information. All transfers involving bulks of information will be treated as data information, regardless of the nature of that information. Examples of such will be image data (obviously), but also program and lookup-table downloading to the individual modules, even if the latter two according to their art just as well could have been handled as control information.

7.1. Data transfer mechanisms

As already explained, a ring bus system is based upon 1-dimensional, point to point communication links between each module and its immediate left and right neighbours. Some people might oppose to the term "bus" being used here, thinking of a bus only in the context of a *shared bus*, where all modules are connected to the same physical set of signal lines. As defined in the included nomenclature, bus can also be used in a much broader sense, which is adopted here.

Every word to be transferred between a source and a destination module must therefore in most cases be relayed through one or more intermediate modules, the number of intermediate modules being equal to the distance between the source and the destination module. The total transfer time can then be regarded as being equal to the sum of two components:

- The delay through the intermediate modules until the first transferred word reaches the destination module (transfer *latency*).
- The actual transfer time, being proportional to the number of transferred words.

The effective communication bandwidth will be equal to the number of words transferred divided with the total transfer time. To minimize the reduction in effective bandwidth due to the transfer latency overhead, data should therefore be transferred as *packets*, each consisting of a relatively large number of words. Single word transfers would be extremely inefficient in a ring bus system.

A ring bus system can be unidirectional or bidirectional. In a **unidirectional** ring bus system, all information flow is following the same direction along the ring, from source to destination, no matter the relative position of the two modules involved. In a N module system, the direction of transfer will be from module "n" to "(n+1)modulo N". The average *transfer distance* (defined as the number of intermediate modules) between two modules in a N module unidirectional ring bus system will be N/2, assuming that the need for communication is evenly distributed over the modules.

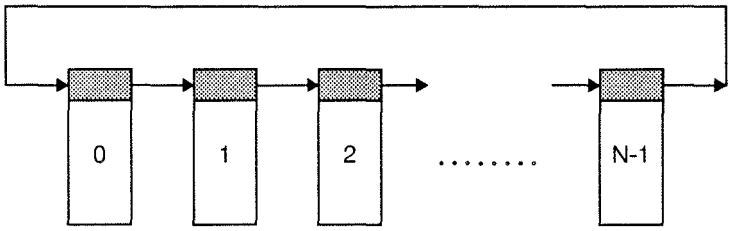


Figure 7.1. Unidirectional ring bus system

In a **bidirectional** ring bus system, however, the transfer can go both ways, to the left as well as to the right. The transfer mechanisms for the two directions can be operated fully independently, each ring bus module can therefore participate both in a left and a right direction transfer simultaneously. Between any pair of source and destination modules, there will exist two possible *transfer paths*, one left and one right. Provided that both paths are free to use, the shortest transfer path should be selected, assuming that no a priori information about the communication pattern suggesting anything else is available. The average transfer distance in a bidirectional ring bus system will compared to the unidirectional system be reduced to N/4. A bidirectional system will therefore be more balanced with respect to "forward" and "backward" transfers, and the average transfer latency will be reduced to the half compared to a unidirectional system.

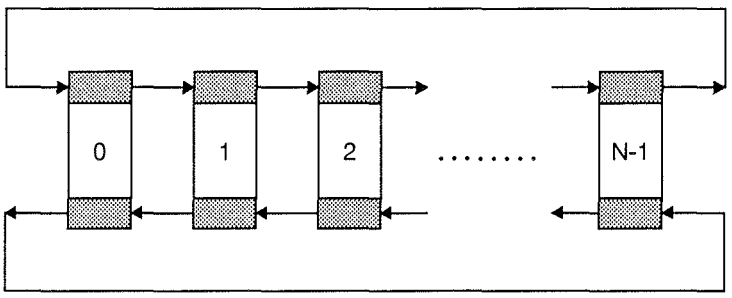


Figure 7.2. Bidirectional ring bus system

However, keeping the width of the transfer words constant, a bidirectional system will require nearly twice the number of signal lines (some control lines may be shared) as a unidirectional system. An obvious alternative to the use of those extra lines would be to simply double the transfer word width, keeping the system unidirectional. In theory, both approaches would double the communication bandwidth. Compared to bidirectional solution, doubling the transfer word width will have the following advantages and disadvantages:

Advantages

- **Simple implementation.** Bidirectional ring bus transfers requires more complex control hardware on nearly all levels (additional control signal lines, more complex arbitration mechanisms and algorithms, larger data structures to store the current state of the ring bus system etc.). Doubling the transfer word width of a unidirectional ring bus system, however, has no further implication than simply doubling the number of data lines and its associated hardware (buffers, FIFOs etc.). The control hardware is not affected at all by the increased word width.
- **Bandwidth pr. transfer doubled.** Even if the potential *system* bandwidth is equal for a bidirectional and a double word-width/ unidirectional ring bus system, the double word-width system transfers twice as many bytes pr. time unit as the bidirectional system does. The bandwidth *pr. transfer* is therefore doubled.

Disadvantages

- **Increased transfer latency.** Due to the fact that the average transfer distance in a N module uni-directional system increases from N/4 to N/2 compared to a equal sized bidirectional system, the average transfer latency will increase correspondingly. The transfer latency is the delay from the transfer starts until the first transferred word is received by the destination module. When transferring large packets of data, the transfer latency will for all practical purposes be negligible. As far as smaller packets are concerned, e.g. for transferring control information between modules, this increase in transfer latency can be of importance. This is due to the simple fact that the latency time will count for a larger portion of the total transfer time the smaller the number of transferred words is.
- **Increased access latency.** Access latency can be defined as the time from a ring bus transfer is requested until it is granted. If the ring bus segments forming the transfer path from the source to the destination module(s) are not all idle, the grant will be deferred until so is the case. With two alternative transfer paths as we will have in a bidirectional system, the chance of finding one available will be larger than with only one path. On the other hand, however, the double word-width of the unidirectional system will imply that a transfer of a given size (in bytes) will occupy the ring bus segments only half the time compared to transferring the same amount of data on the left or the right bus in a bidirectional system. What these two contradictory factors actually sums up to, will be application dependent and have no single answer, valid for all situations.

Taking all factors into account, it is therefore not possible to make a unique decision on which system to prefer: The bidirectional or the unidirectional, double word-width ring bus system. The optimum solution will depend on the actual communication need in conjunction with the physical location of the communicating modules. As earlier explained, the processing path through the system will be allowed to be data dependent. This will be the situation if the result of a processing operation must be qualified by some sort of analysing module before the data (the result) is permitted to enter the next stage in the processing path. In addition to being application and configuration dependent, the pattern of communication can therefore also vary dynamically within the same application. From these reasons, simulations will be of little value as far as choosing between a bidirectional and a unidirectional, double word-width ring bus system is concerned.

To modify a unidirectional system into a bidirectional system will imply severe changes to the various ring bus control mechanisms. Doubling the transfer word-width, however, will be a considerably easier task, only involving a duplication of the hardware involved (signal lines, buffers etc.). To make room for future enhancements and modifications, the decision taken is therefore:

> *For high speed data transfers, a bidirectional ring bus system is chosen, ensuring a balanced communication capability between the two directions of transfer. In case of a later demand for higher communication bandwidth, this can easily be accomplished by increasing the transfer word-width, without changing the underlying control structure.*

7.1.1. Transfer word-width

To comply with available integrated circuits as well as the "defacto" standard characteristics of image data, the transfer word-width should be a multiple of 8 bits. Preferably, it should also be a power of two. The size (in number of bytes) of the data to be transferred is usually a power of two. If the transfer word-width is a power of two as well, padding of the last transferred word is avoided. A bidirectional 8 bit system will require 32 signal connections to each module (8 bit * 2 directions * both input and output), a 16 bit system 64 connections and a 32 bit system 128 connections. Inclusive an additional number of control lines a 16 bit system will fit into a single 96 pin connector, it can therefore be regarded as a suitable basic unit of construction.

> *The ring bus transfer word-width is chosen to be 16 bit.*

7.1.2. Units of transfers

A packet always consists of a header and a data part. The header identifies the sending (source) module, the receiving (destination) module(s), additional information necessary to identify the data contained in the data part and, in many cases, what to do with it. The data part contains the actual data being transferred, the header can therefore be regarded as a sort of necessary "bureaucracy" to get the data transferred.

The packet size is the sum of the header and the data part size. When changing the packet size, the size of the header will remain constant. The change will therefore be in the size of the data part. Accordingly, a suitable packet size will be a trade-off between:

- **High effective bandwidth.** The effective bandwidth is equal to the number of data bytes transferred divided by the packet transfer time. The larger the data part is compared to the header, the higher the effective bandwidth will be. For a ring bus system, the transfer latency also makes large packets advantageous as far as high effective bandwidth is concerned.
- **Short ring bus access latency.** Assuming that an ongoing packet transfer can not be interrupted, large packets inevitably implies that other modules requesting access to the ring bus must wait longer to get their requests granted. The larger each packet is, the longer the ring bus access latency will be.

When determining the packet size, the logical structure of the data to be transferred should also be considered. In our case, the bulk of data will be ultrasound image data. Each ultrasound image is acquired as a number of beams, with each beam consisting of a number of samples.

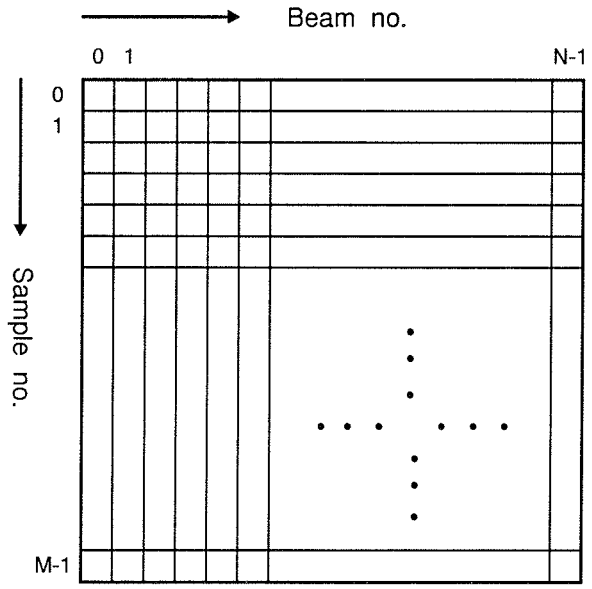


Figure 7.3. Ultrasound image data organization

If the processing of an image must be distributed over several processing elements (PEs) due to the amount of processing power necessary to comply with the real-time requirement, beams will constitute the natural dividing lines. Preferably, a packet should therefore contain an integral number of beams, or the transfer of one beam should fill an integral number of packets. In addition to the aspect of sharing the processing load, this will also be advantageous with respect to the PE hardware design: Local address calculations, data buffering and memory

segmentation. All these issues will be easier to implement if the packet size is tuned to the logical structure of the data. For several image processing algorithms (e.g. Fast Fourier Transforms FFT, edge detection algorithms), the beam will also be a suitable macro-unit of operation.

One beam in a medical ultrasound image will typically consist of 512 bytes, equal to 256 transferred words in a 16 bit ring bus system. Due to the uninterruptable nature of a packet transfer, this is too much to transfer as one packet. I therefore choose to divide one beam of image data into two packets.

> *Data are transferred in packets consisting of 256 bytes (128 transferred words) of data. In addition comes the packet header.*

7.1.3. Data transfer modes

To obtain effective system communication, it is important that the transfer modes available for moving data from one module to another is adapted to the characteristics of the underlying communication system as well as to the application's communication pattern. Due to the nature of a ring bus system, the only type of transfer in question is the block or packet transfer. Single word transfers will imply an intolerable overhead due to the access and transfer latencies.

When ultrasound image data enters the system from the frontend, the same data are in many cases to be transferred to several destination modules: One module may perform some sort of preprocessing removing noise or adjusting the contrast of the image. Another may do a statistical analysis of the data with the purpose of computing filtering parameters to be used by a module at a later stage in the processing path. A third module may store this (original) image data for later retrieval and replay. It will be a waste of communication bandwidth to transfer this data as separate transfer operations to each and every one of the destination modules. By transferring the data to the most distant destination module, counted along the direction of transfer, the intermediate destination modules can make their own copies of the data on the fly as the data passes by. Provided available buffer space, this eavesdropping mechanism will imply no transfer speed penalty compared to not making intermediate copies.

> *Transfer modes supporting multiple destination transfers will therefore be of vital importance to overall system performance.*

There are two types of multiple destination transfer modes:

Multicast. The simultaneous transfer of data from one source module to several (but not all) other modules.

Broadcast. The simultaneous transfer of data from one source module to all other modules.

Broadcast is most often used for transferring control information (reset, interrupt disable etc.) from a controller to its slave modules, transferring the same data to all other modules is a more unusual situation as far as data transfers are concerned. Multicast transfers, however, will be very useful for the transfer of both control as well as data information.

7.1.4. Data buffering

Each module will have a limited amount of local buffer space, large enough to handle temporal fluctuations in the real-time flowing stream of image data. However, due to the data dependent processing feature, involving iterative processing until some quality criteria is met, the stream of data is no longer fully predictable. There will then always be a chance of the local buffers going full. A handshake mechanism must therefore be established ensuring that data is not transferred to a destination module unless it is able to take care of it. How should the situation where a module is not ready to accept new data, especially in connection with multiple destination transfers, be handled?

There are three possibilities:

Wait until all ready. The entire transfer operation is postponed until the destination module (all destination modules) are ready to accept the data. Whether it is some controller's or the sending module's responsibility to initiate a transfer retry, is a matter of discussion. Until the transfer can be executed, the data must be kept in the sending module's local buffer.

Transfer when module is ready. In case of a multiple destination transfer, the busy module(s) are deleted from the destination list and the transfer is done to the remaining (not busy) destination modules. As far as the busy modules are concerned, a retransfer must be done when those modules become idle. Implementation issues like whether to retransfer the data to a destination module as soon as it becomes idle, or to wait until all remaining destination modules become idle so that only one retransfer is required, will be discussed later. The same applies to whose responsibility it is to initiate the retransfer(s). Regardless of the chosen strategy, the data must be stored in the sending module's local buffer until it is successfully transferred to all destination modules.

Buffer data for modules not ready. The busy module(s) are removed from the destination list and the address of a buffering module capable of storing the data being transferred is added to the list. The transfer operation is executed, then it is the responsibility of the buffering module or a controller to retransfer the data to the busy modules whenever they become idle.

There is one important difference between the first two methods and the third: As far as the first two are concerned, it is the responsibility of the *sending module* to buffer the data until the transfer operation is complete (that is, the data have been transferred to all destination modules). If this buffering fills the module's available buffer space, this will block for incoming transfers and a chain reaction of busy modules may start to propagate backwards the processing path, towards the data acquisition module(s). Due to the iterative processing feature, this will create a great danger of system deadlock.

By using the third method, however, the sending module's local buffer is released when the data are transferred, whether it is to the destination module(s) originally requested or some intermediate buffer. The sending module is then free to accept new data. The need for extra buffer space will, depending on the type and amount of processing being done, be unevenly distributed over the modules. Due to the data dependent processing, the amount of buffer space required, and where it is needed, will also vary dynamically. It will therefore be a far better solution to have one large buffer, accessible as a global resource to all modules, than to

distribute the same amount of buffer space over all modules resulting in a static and inflexible configuration. Not burdening the sending module with the responsibility of data buffering and retransfer to busy destination modules, is also more in accordance with the principles of data flow computing: When a processing element has executed its operation and a result is computed, its only responsibility is to deliver that result to some controller module. The controller then transfers that result, along with any other data needed, to the next processing element in the data flow path where it will be used as operands for that processing element's operation. Conclusion:

> *Data buffering is done in a global buffer resource, accessible by all modules. In case of any busy destination modules, the data transfer to those modules are readdressed to the buffer module, thereby releasing the sending module for the responsibility of storing and retransferring the data. A protocol with an underlying hardware mechanism supporting efficient handshake between controller and processing elements as well as destination address remapping must be developed.*

7.1.5. Signalling protocol

When trying to set down the specifications for an appropriate signalling protocol, the first issues to be discussed are: The type of *synchronization scheme* to use to maintain proper timing relationships between the module supplying and the module receiving the data and the *handshake protocol* between the two modules ensuring that no data are lost during the transfer process.

7.1.5.a. Synchronization schemes

Basically, there are two fundamentally different ways to implement the synchronization mechanism, *centralized-synchronous* and *source-synchronous*:

Centralized-synchronous

All signal timing is then referred to a central clock source. In the Ring bus system, this clock may be global to all modules or local to each Ring bus segment.

With a clock global to all modules, the perception of the clock on each module will be subject to *spatial skew*: It will not be truly simultaneous from module to module. It will therefore be an uncertainty in the timing relationship between the clock signal and the data which timing is referred to that clock. This is caused by spatial separation of the two signal sources, the clock is generated by a central clock generator while the data is supplied by any one out of the Ring bus modules. The minimum spatial skew for a centralized-synchronous backplane, occurring when the clock generator is localized in the middle of the backplane, is one end-to-end backplane delay. In addition to the skew introduced by the physical separation of modules, there will also be a skew component due to the inter-device skew between clock receivers, obviously located on physically distinct devices on the different modules.

If the clock is local to each Ring bus segment, the spatial skew will be reduced due to the reduced geographical distance between the (two) modules involved. Besides, because a Ring bus segment is a uni-directional one-to-one link, it must necessarily be the same module which supplies the data as generates the clock. As far as the electrical characteristics are concerned, they will therefore be as for the source-synchronized scheme.

Source-synchronized

In a source-synchronized timing scheme, there is no central system clock to which signal timing is related. Instead, the module supplying the data onto the bus also emits a synchronizing signal to which the timing of the data signals is related. Because the synchronization and the data signals are now emitted by the same physical module, the spatial skew is significantly reduced. It can not be totally eliminated, however, due to the potential different load conditions for the synchronizing and the data signals as well as inter-device skew between physically distinct devices. Examples of source-synchronized protocols can be found in various asynchronous buses. The VMEbus [Motorola 1985], for instance, uses source-synchronization for its address as well as its data signals, each having its own synchronization (strobe) signal.

In summary, an inherent characteristic of a centralized-synchronous clocking scheme is an uncertainty in timing relationship between the synchronizing signal (the clock) and the signals to be synchronized (the data). This uncertainty, called skew, will limit the maximum speed by which data can be transferred. The only reason to use a centralized-synchronous clocking scheme is that interface design is simplified. To accommodate for slower modules or modules temporarily busy, a wait-state protocol must be included. This will introduce performance degradation in terms of reduced transfer speed in discrete steps, inserting one or more wait-states. A centralized-synchronous clocking scheme with clocks local to each Ring bus segment will, as in fact implied by the name, be a sort of self-contradictory solution. The simplicity of the central clock is lost, and nothing is gained compared to the source-synchronous scheme. On the contrary, more control signals are needed to implement the centralized-synchronous scheme. Another issue is that there is no reason whatsoever to force two otherwise totally independent modules to synchronize to a common clock only to transfer data between them. Conclusion:

> *The transfer of data on the Ring bus will be implemented by a source-synchronous transfer scheme.*

7.1.5.b. Handshake protocols

Having selected the source-synchronous transfer scheme, there are two possible protocols for implementing handshake between the two modules involved in the transfer: The *compelled* and the *non-compelled* protocols.

Compelled protocol

The basic idea behind the compelled protocol is that no module is required to have any explicit knowledge about the timing requirements of the module it communicates with. Instead, the speed of transfer is dynamically adapted to the speed of the communicating modules. Asynchronous buses like the VMEbus [Motorola 1985] operates in this way.

The advantage of a compelled protocol is that the sending as well as the receiving module is in full control of the transfer. Timing is dynamically adapted to the two participating modules. The inevitable disadvantage, however, is that the speed of transfer is limited by the round-trip delay introduced by the handshake protocol.

Non-compelled protocol

To avoid the speed penalty introduced by the handshake round-trip delay, the receiver module must abandon its power to dynamically control the speed of transfer as it does with the compelled protocol. Instead of having a handshake signal acknowledging every transfer, the transfer speed must not exceed a maximum value, included in the protocol's specification, which all modules must be capable of handling. The non-compelled, source-synchronized protocol is in many senses like the centralized-synchronous protocol except that the source of the synchronizing signal (the "clock") changes with time. With the former, however, the transmitter of data emits *both* the synchronizing signal and the data. Another difference is that a non-compelled transfer is not constrained to follow a single, periodic data rate but can dynamically slow down. A non-compelled transfer mode is included in the Futurebus specification [IEEE 1987].

However, the lack of receiver control of the transfer may in certain cases cause the maximum transfer speed to be set to a lower limit and/or the units of transfer (packet size) to be decreased compared to a situation where the receiver dynamically is able to control the transfer speed. To achieve maximum performance without reducing flexibility, the key issue is therefore:

> *How can the receiver be made able to control (that is, slow down) the transfer without sacrificing maximum transfer speed?*

This can be done if the receiving module does not require an instant response to a slow down request, but can handle a few words (typically one or two) being transferred *after* the request has been issued. In other words, the receiving module must be able to issue an "early warning" when its internal buffer is about to go full, early enough to give the transmitting module time to react and hold the transfer until the receiver eventually signals that the transfer may continue. By using FIFOs or RAM-based ring buffers to buffer data input to a module, this can easily be accomplished. In this way, the receiver will be in full control of the transfer even if it is done at full speed as long as there is buffer space available. Conclusion:

> *Data transfer should be based on a modified, non-compelled transfer protocol. The modification consists of a "transfer hold" mechanism making a receiving module able to freeze the transfer when or if its internal buffer is about to go full.*

7.1.5.c. Signalling schemes

As described by Charles L. Seitz in his chapter "System Timing" in [Mead 1980], there are two alternative signalling schemes in self-timed systems: *Four-phase* (RZ) or *two-phase* (NRZ) signalling.

Four-phase (RZ) signalling

In a system using four-phase, or Return-to-Zero, signalling, signals must return to an inactive state between each time they are asserted. This is the signalling scheme used by most modern, industrial buses (VMEbus, MultibusII, NuBus etc.) and tends to result in a very simple and natural implementation. Actions are only related to one of the two signal transitions, usually the falling edge when the signals changes from a high to a low value. Because the names "four-phase" and "two-phase" is derived by counting the necessary phases for transferring one word using a compelled mode transfer scheme, such a transfer is used to illustrate four-phase signalling as shown in the following timing diagram.

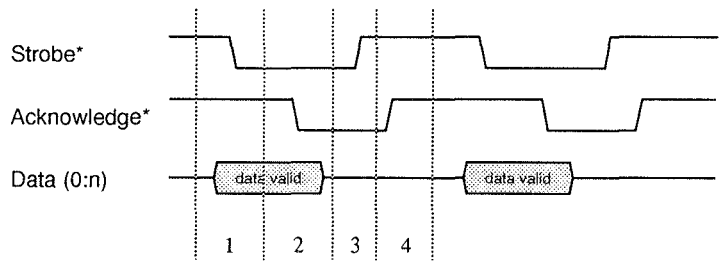


Figure 7.4. Four-phase (RZ) signalling

Two-phase (NRZ) signalling

With two-phase (Non-Return-to-Zero) signalling, both transitions of the signals, the rising and the falling edge, have the same meaning. Two-phase signalling is therefore also called *transition signalling* [Sutherland 89]. The same timing sequence as used to illustrate four-phase signalling, but now implemented by two-phase signalling, is shown in Figure 7.5.

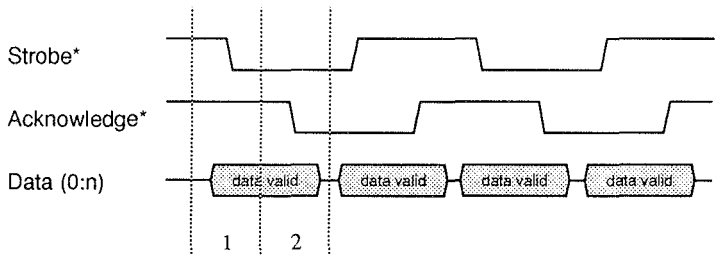


Figure 7.5. Two-phase (NRZ) signalling

Compared to four-phase signalling, two-phase signalling only uses half the number of signal transitions to signal a certain action. Because there are time and energy costs with driving a transition onto a wire, generally speaking it is advantageous to use as few transitions as possible in self-timed signalling conventions. All responses to transition signals are edge-triggered, and triggered both on rising as well as falling edges. Due to the fact that both edges are used as trigger events, transition signalling may offer twice the speed potential of the conventional, four-phase clocking scheme. To my knowledge, only Futurebus [IEEE 1987] of today's digital buses uses a two-phase signalling scheme.

Two-phase signalling is the optimum scheme as far as speed and energy efficiency are concerned. This must be paid for, however, by a more difficult and costly implementation: Because logic devices tend to be sensitive to logic levels or transitions in a particular direction, extra logic is required to make the interface implementing the two-phase signalling scheme symmetric with respect to signal transitions. Nevertheless,

> *To achieve maximum transfer speed, the source-synchronized, non-compelled mode synchronizing signal is implemented by using two-phase signalling.*

Because it is to be regarded as an exception event, it is nothing to gain by implementing the already described "transfer hold" line by two-phase signalling. A simple level-sensitive mechanism will suffice.

7.2. Control transfer mechanisms

Depending on the level of control, the control mechanisms required to handle a ring bus system as described can be divided into two groups, micro- and macrolevel control.

7.2.1. Microlevel control

This is the low level control mechanisms managing the various hardware resources in the system. They are implemented by the use of dedicated hardware, either centrally located (on the controller) or distributed over the processing elements. The microlevel control mechanisms required to run the ring bus system are:

- Ring bus access control.
- Module handshake, including destination address remapping.
- Packet retransfer.
- Exception handling.

7.2.1.a. Ring bus access control

As already explained, all modules are connected to their immediate left and right neighbours by a ring bus. Each ring bus segment connecting two modules is a unidirectional point-to-point link. To transfer data between two modules, all intermediate segments must be available for the

transfer to take place. The ring bus must therefore be regarded as a global resource, whose access and use must be controlled by some sort of arbitration mechanism. Two main approaches to sharing this kind of resource are available:

Time slot scheduling

In a N-module ring bus system, the bus is allocated to each of the modules 1/N of the total time. The allocation is done on a round robin (circulating) basis, controlled by a global clock and a global counter, whose value is visible by all modules: At any moment of time, the module having the identity (address) equal to the counter's current value, has the bus to its disposal. Alternatively, the time slot scheduling mechanism may be implemented as a sort of circulating baton (the short stick used in relay races) or *token passing* procedure. The module currently having the baton (token) in its possession has access to the bus.

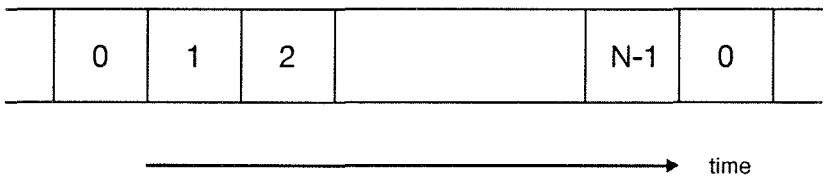


Figure 7.6. Time slot scheduling

Because the actual data transfer is done during a module's time slot, the size of the time slot must at least be equal to the time needed to transfer one packet. The data transfer from any module is then fixed to certain periods of time where the module has exclusive disposal of the bus. An arbitration mechanism of its own is therefore not necessary in this case. However, due to the length of the timeslots involved, this will imply a prohibitively large access latency.

Time slot scheduling, especially with the length of the time-slots involved in this case, therefore has the major disadvantage of being ineffective in non-symmetrical load situations, where the need for doing data transfers are unevenly distributed over the modules: Time-slots belonging to modules not requesting data transfers still have to elapse before the baton can be passed on to the next module along the scheduling line.

Request when needed

Alternatively, the ring bus system may be regarded as a common resource to be requested and used by a ring bus module whenever it has a need for it. Compared to time-slot scheduling, this will imply a far better utilization of the available bandwidth when the modules have unequal needs for doing data transfers, which will be the general case in a system as flexible and reconfigurable as this is intended to be. Conclusion:

> *Ring bus access and use is to be based on "request-and-use-when-needed" principle.*

However, this means that a dedicated ring bus arbitration mechanism is needed to control the access and use of the bus. When designing this kind of mechanism, several important issues must be addressed:

- **Efficiency**. The process of requesting and granting the use of a resource must in nature be regarded as an overhead, decreasing the overall performance of the system. It is therefore important that this process is as efficient as possible, preferably taking place in parallel with the data transfer itself.
- **Fairness**. The selected arbitration mechanism must ensure that each module competing for the bus get its fair amount of bus usage. What is "fair" will be application and system dependent: In some systems, some of the modules may have a more urgent need for bus transfers, for instance by being subject to hard real-time requirements, than other modules. They must therefore be given priority to as far as bus usage are concerned. In other systems, all modules are symmetrical with respect to their communication needs. "Fair" in that case therefore means that all modules should get equal shares of the available bus bandwidth.
- **Flexibility**. With reference to the above discussion, the "fairness" criterion may be dynamic rather than static: It may change according to the current operating mode of the system. The arbitration algorithm, including the bus allocation philosophy and the fairness criterion, should therefore preferably by programmable rather than hardwired into fixed logic and backplane signal lines.
- **Ease of implementation**. Needless to say, the lesser amount of hardware resources (signal lines, electronics) necessary to implement a given function, the better.

Having those four principal guidelines in mind, we will now discuss some of the most important issues in arbitration mechanism design:

The heart of every arbitration mechanism is a logical, functional unit called the *arbitration controller*. Physically, the arbitration controller may be either *centralized* to one module or *distributed* over several (all) modules. When a ring bus access request is issued, the arbitration controller must decide whether to grant or reject that request. To be able to do that, two kinds of information must be available to the controller:

> *What is the current state (busy or idle) of the ring bus segments required to take part in the transfer?*

Because any ring bus segment can handle only one transfer in each direction at a time, a busy segment will block for another transfer through that segment until the first transfer is released. It is therefore necessary for the controller at any time to have a complete, consistent picture of the state of all ring bus segments.

> *Between which two modules (source and last destination module) are the transfer to take place?*

A module requesting access to the ring bus must therefore identify itself as well as the last destination module down along the transfer path. As already discussed, however, we will use a bidirectional ring bus. In case of a multiple destination transfer (multicast), the "last destination module" will then depend on the direction of the transfer. The requesting module must therefore either specifically ask for a particular direction of transfer, or it must supply a complete destination list to the controller, which then will be able to figure out which module will be the last destination module for a given direction of transfer. The best solution will to a certain extent depend on whether a *distributed* or a *centralized* controller implementation is chosen.

Distributed arbitration controller

In this case, the controller function is distributed over all modules with no centralized storage of a system state vector. A protocol must therefore be included where the source module asks all modules to be involved in the transfer (destination module(s) as well as intermediate modules), if they are able to participate in the transfer (that is, whether they are idle or not). This may easily lead to requested transfers being blocked unnecessarily when several requests are issued by different (source) modules at the same time. This is shown in Figure 7.7.

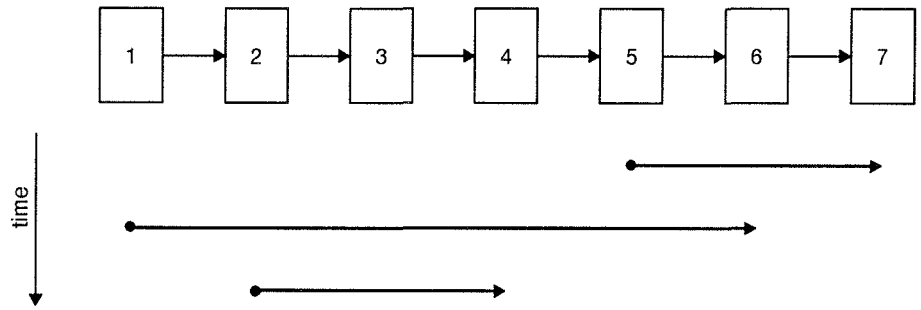


Figure 7.7. Multiple request situation

The first request (5 to 7) is immediately granted while the second (1 to 6) will be deferred until the first transfer is terminated. The third request (2 to 4), however, will also be blocked because segments 2 to 4 already has been allocated to the deferred second request. This request may have been granted and the transfer performed in parallel with the 5 to 7 transfer. One solution would be to demand requests which can not immediately be granted to be withdrawn and the already allocated ring bus segments released until a later point in time. They will then not cause other requests with all required segments actually available to be rejected. However, an inevitable side-effect of such a scheme would be that long transfers would experience a lower bus usage priority than short transfers, running the risk of being blocked entirely by two alternating short transfers. With the arbitrating function distributed over all (requesting) modules, it will be very difficult to implement a mechanism being both fair and efficient. In short, the mechanism needed to implement a distributed arbitration controller will be both complex and time-consuming, and may yield less than optimum performance as far as bus usage fairness and efficiency are concerned.

Centralized arbitration controller

The controller function is in this case located on one single module, presumably a local controller. Because all requests then are serviced by the same physical module, a state vector showing the current state of all ring bus segments is easily maintained. When servicing a request, it is then not necessary for the controller to go out and ask the segments to be involved in the transfer whether they are busy or not, this information is already in the state vector. Further, because the controller function will reside on the same module all the time, it is possible to use dedicated backplane signal lines instead of a software protocol for implementing the necessary handshake between the controller and the ring bus modules. In practice, this is not possible when the controller function is distributed over all modules. Needless to say, an on-module accessible state vector together with hardware handshake lines means a significantly faster request service than what obtained by the software based scheme described for the distributed arbitration controller. Another benefit of servicing all requests by one single controller module is that there is then no danger of system inconsistency leading to possible deadlock situations where two (or more) modules are blocking each others request.

From this discussion, the following conclusion should be fairly obvious:

> *The arbitration mechanism for controlling the access and use of the ring bus system should be based on a centralized arbitration controller.*

Because the arbitration controller at any time will have a complete, general view of the state of all ring bus segments, it will be better to let the controller decide whether to perform a requested transfer on the left or the right ring bus instead of having the source module to ask for a specific transfer direction. This implies that the controller must know the complete list of destination modules, or at least the destination modules to the immediate left and right of the source module, prior to granting a request. If both transfer paths are available for servicing the request, the controller should choose the shortest one, assuming that the need for communication is evenly distributed over all modules.

However, to transfer the necessary arbitration information (identity of requesting module, destination list) to the controller, a communication channel other than the ring bus would be desirable. Although the arbitration information *could* have been transferred on the ring bus using the *control packet* scheme as described in Section 7.2.2., this would lead to poor performance due to the transfer latency. Because of the small amount of data involved, an ordinary shared bus will be the simplest and most effective way of transferring the necessary arbitration information. Because this information will be a list of addresses, this shared bus will be called the *address bus*. In addition to a (small) number of data lines required to specify module addresses ($M = \log_2 N$ for N modules, a single line is needed to signal when the bus is busy

> *The required address information to be supplied by a source module, requesting the ring bus, to the controller, is to be transferred to the controller by a dedicated, shared bus called the Address bus.*

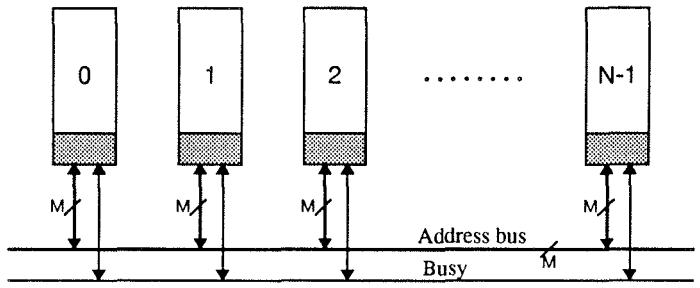


Figure 7.8. Address bus

Like the ring bus, the address bus itself will be a resource global to all modules and must therefore have an arbitration mechanism of its own. No matter whether it is a shared bus or a ring bus which is the global resource to arbitrate for, the four principal guidelines for arbitration mechanism design already discussed is still valid. We will now take a look at the various alternatives for implementing the address bus arbitration mechanism:

Time slot scheduling

Time slot scheduling can also be used for bus arbitration purposes, rather than for doing actual data transfers. Due to the reduced amount of required information involved in a bus arbitration rather than a data transfer process, the length of each time slot may in this case be considerably shortened, resulting in reduced access latency compared to data transfer time slot scheduling. Time slot scheduling can therefore be said to be an approach better suited for doing bus arbitration than transferring relatively large amounts of data. As already discussed, to keep track of the module currently possessing the time-slot, an actual implementation must be based either on a global counter or a circulating baton.

However, even though somewhat reduced due to the shorter length of each time slots, time slot scheduling has still the major (and inevitable) disadvantage of being ineffective in non-symmetrical load situations. This is because time slots belonging to modules not requesting data transfers have to elapse, wasting communication bandwidth due to the idle bus, before the next module along the line can request the bus.

Request/ grant arbitration

This mechanism is based on a number of backplane signal lines upon which a protocol is built whereby a module can signal its need for doing a bus transfer. Several implementations are possible with this scheme:

Single line request. The simplest approach is to let all modules share the same bus request line. Each module must then be connected to this line through an open-collector like interface, permitting simultaneous assertion by several (all) modules. The assertion of the shared request line is detected by a centrally located arbitration controller. Due to the use of a single line, the

controller is unable to determine the identity of the requesting module. The grant is signalled to the modules through a dedicated bus grant line. To be able to handle a situation with several simultaneous requesting modules, the bus grant line must be daisy-chained from module to module. A module not requesting the bus will pass the grant signal to the next along the chain, while a requesting module will remove the grant signal and start using the bus. In case of several simultaneous requesting modules, the modules will therefore have mutual priorities according to their placement relative to the arbitration controller: The nearer the controller, the higher the priority.

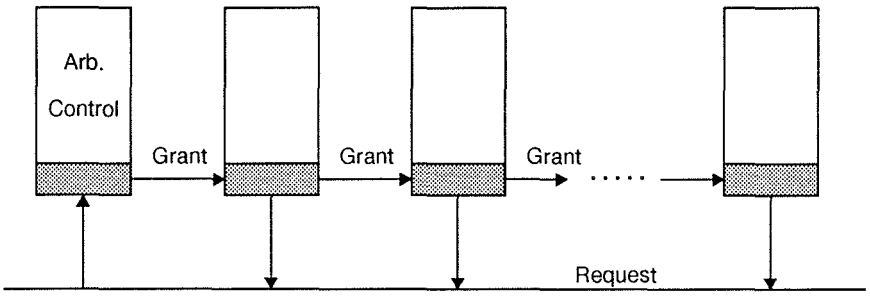


Figure 7.9. Single line bus request mechanism

An obvious deficiency with this mechanism is that it is impossible for the arbitration controller to know the identity of the requesting module. Consequently, it is also impossible to implement arbitration algorithms giving different priorities to different modules, or algorithms ensuring equal treatment of all modules. Due to the method of resolving conflicts in case of simultaneous requests, heavy load situations may imply a great danger of starvation for the modules having the lowest priorities (that is, the modules furthest away from the controller). This will be totally beyond the control of the arbitration controller. In addition, the single line request/ grant mechanism will be slow due to the long bus grant daisy-chain line.

Multiple line request. Extending the described single line request/ grant scheme to a mechanism where the requesting module is identified, every module is required to have its own bus request line. As far as bus grant signalling is concerned, one solution is to let every module have its own grant line as well. Alternatively, a single, shared bus grant line can be used in conjunction with a relatively small number of lines coding the identity of the module being granted. Knowing the identity of the requesting module and being able to grant each module individually makes it possible to implement any arbitration algorithm. The disadvantage with this approach, of course, is the large number of backplane signal lines required.

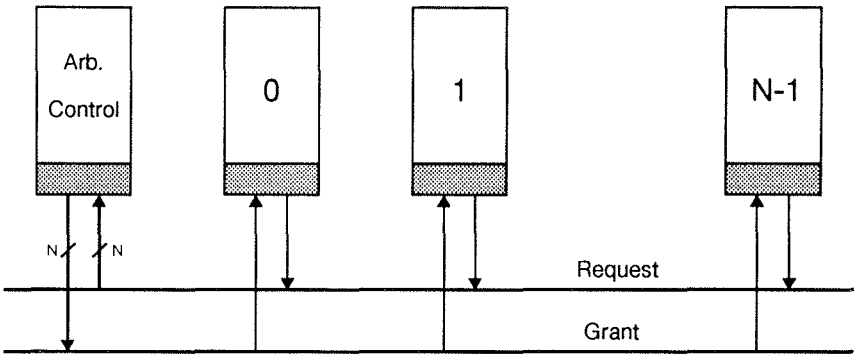


Figure 7.10. Multiple line bus request mechanism

The compromise. A compromise between the two methods would be to partition the modules into several groups, each group having its own request/ grant signal line pair. By assigning different priorities to each group, either on a fixed or a rotating basis, different arbitration algorithms can be implemented. Simultaneous requests from several modules within the same group is resolved by daisy-chaining the bus grant lines.

Distributed arbitration

In contrast to the centralized request/ grant mechanism, where bus usage is regulated by a central arbitration controller, a distributed arbitration scheme has no such controller. Instead, every module is able to observe all bus requests issued by all modules as well as the identities of the requesting modules. This is possible by assigning each module one or a set of arbitration codes, each one being unique to a specific module. Implicit in the arbitration code value is also a request priority. With several simultaneous requests, a module is therefore able to decide whether the highest priority request is its own request or if it is a request from another module. In that case, the module having the lowest priority will withdraw its request. Eventually, only the module having the highest priority request will remain requesting the bus. The distributed arbitration scheme is described in Appendix B.

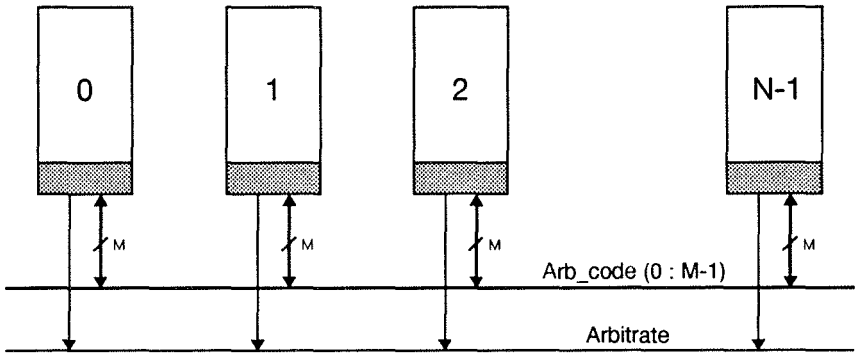


Figure 7.11. Distributed arbitration

A condition for this mechanism to work is that two (or more) modules never requests for the bus using the same arbitration code. One way to ensure this is to use the module's identity (address) as a part of its arbitration code. The rest of the arbitration code may be formed by bits signalling the purpose of the module's request. In this way, different module activities may have different priorities as far as bus usage are concerned.

One special application of the distributed arbitration scheme, is for *high priority message passing*. Due to the fact that an asserted arbitration code can be observed by all modules, messages can be broadcasted by reserving a set of codes for this purpose rather than actually requesting the bus. Emergency messages signalling master clear, power fail and other exception events are candidates for this kind of signalling.

The distributed arbitration scheme has the advantage of being fast, compact (a small number of arbitration lines) and flexible. Several arbitration algorithms may be implemented and the mechanism offers the additional benefit of facilitating a high priority emergency message transfer scheme. Conclusion:

> *For controlling the access and use of the shared address bus, a distributed arbitration mechanism will be used. This mechanism should support transfer request, transfer release as well as broadcasting emergency messages.*

7.2.1.b. Module handshake

To make the controller able to decide whether a requested destination module is ready to accept new data, a handshake mechanism must be established between the controller and the other ring bus modules (that is, the processing elements, PEs). As previously mentioned during the discussion of the arbitration mechanism, there are two principal ways of implementing such a handshake:

Software protocol

By reading a status word on board each PE, it can be decided whether the PE is ready for new data or not. This reading can be done by the source module or the controller. If it is done by the source module, the source module must in case of any busy destination modules inform the controller about their identity so that the controller knows for which modules to buffer the data. Therefore it will probably be a better solution to let the controller take care of the software handshake procedure. In any case, the handshake procedure must be carried out sequentially, at least as far as the answers from the destination modules back to the controller or the source module is concerned. It may be implemented by transferring small packets on the ring bus, or better, by using the shared address bus.

Hardwired control lines

Having already decided that ring bus arbitration should be carried out by a central controller, it is natural to let the controller take care of the module handshake, too. This especially so because the outcome of the handshake procedure in case of any busy destination modules will be a re-routing of data intended for those modules. This re-routing must also be handled by the controller. As already mentioned during the discussion about a centralized vs. a distributed ring bus arbitration mechanism, it is feasible to let each PE have its own, dedicated hardwired handshake line directly interconnecting the PE with the controller. If each Pe then is driving its handshake line according to its busy/ idle status, the controller can by reading all handshake lines in parallel get an instant picture of the status of all associated PEs. Further, if those handshake lines are accompanied by a few extra signal lines, driven by the controller and connected in parallel to all PEs, the controller is able to dynamically control the function of the handshake lines through the state of these extra lines. Finally, by doing the handshake lines bidirectional, they will rather than being dedicated lines for monitoring PE status be multipurpose status/ control lines facilitating direct interaction between the controller and its PEs. Conclusion:

> *Direct interaction between the controller and its PEs are implemented through a single, hardwired backplane multipurpose status/ control line connecting the controller to each one of its PEs. The current function of this line is determined by the state of some additional lines, driven by the controller and monitored by all PEs.*

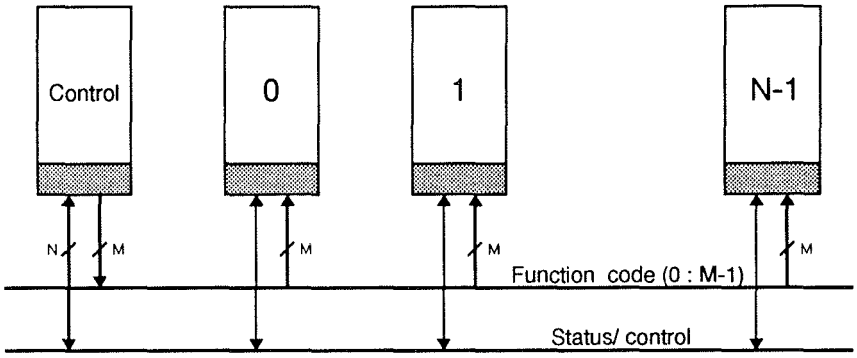


Figure 7.12. Hardware handshake mechanism

7.2.1.c. Destination addressing

Initially, a list of destination modules is provided by the source module when requesting a ring bus transfer. In case of a destination module not being able to receive new data due to lack of local buffer space or some other reason, the data intended for that module must be buffered for later retransfer when the module eventually becomes ready. As already discussed, it is the controllers responsibility to take care of that data as well as initiating a retransfer at a later point of time. The initial destination list may therefore be modified by deleting one or more destination modules and a adding a buffer module (presumably the controller) to the list. As far as destination addressing is concerned, this may be implemented in two different ways, depending on which module is doing the actual addressing:

Source module controlled

Based on the outcome of the module busy/ ready handshake operation, the controller must then send some sort of control message back to the source module, informing the source module whether some of its requested destination modules are unable to receive data. The source module must then modify the packet header, containing the addresses of the destination modules according to the buffering scheme described in section 7.1.4.: The addresses of the busy destination modules are removed from the destination list and replaced by the address of the buffer module In addition, the header must contain the address of the last destination module. This is because the last destination module, unlike the other destination modules, must remove the data from the ring bus after making a local copy. The control message from the controller containing the modified destination list can be transferred either on the ring bus, or better, on the shared address bus.

Controller controlled

Information whether a destination module is busy or not is collected by the controller from all modules simultaneously via the multipurpose status/ control line. By changing the code on the associated control bus, the controller can change the function of this line to enable the

(destination) modules which have their status/ control line asserted. Correspondingly, by applying yet another code on the control bus, the module being the last destination module can be noticed. Only two additional status/ control line cycles are thereby necessary to set up the destination modules to receive data.

In this way, destination addressing is completely taken care of by the controller after the source module has provided the initial destination list. The source module will therefore not know, and do not need to know, to which of the destination modules the data is actually being transferred and for which modules the data are buffered for later retransfer. This is because the source module as already discussed has no responsibility for the data whatsoever after the (initial) transfer, whether they reached their intended destination or were buffered for later retransfer.

Dynamic load sharing

Another aspect of this discussion is the possibility of doing address mapping with a central controller: If a certain module in the system is very heavily loaded and thereby represents a bottleneck to the overall system operation, this bottleneck can be removed by inserting additional, physically identical modules into the system, and sharing the processing load among them. As far as the source module supplying the data is concerned, it is of no interest to know which one in the set of identical modules actually receiving and processing this data. The destination address supplied by the source module to the controller can therefore be interpreted by the controller as a *logical* rather than a physical address, specifying either one of a number of identical modules. No special means or modes of addressing is necessary to accomplish this: When the controller receives a destination address addressing one in a set of several identical modules, it implicitly knows that it can direct the data to either one of them. It is up to the controller to select the actual module to use. If all modules are busy, the data are buffered and eventually retransferred to the first module being ready.

To implement a feature like this with a distributed control mechanism, it would require all modules to have a complete map of the physical rather than the logical resources in the system. In addition, allocating and releasing those resources must be done by software controlled packet transfers, implying large delays and thereby increased possibilities of deadlock situations compared to the centralized control method.

This scheme of dynamic load sharing must not be confused with the corresponding static approach: In that case, a processing task is statically divided between several modules, e.g. the processing of one image frame is done in parallel by several filtering modules, each one processing its own part of the total image area. From a system point of view, each module is then statically allocated to a specific set of data, leading to a much less flexible (and thereby powerful) system structure than if the modules are allocated dynamically at need. In conjunction with iterative processing, it might for example be better to allocate different modules to different iterations rather than to different parts of the image. Because iterative processing will be data dependent, the optimum solution can not be decided until runtime, thereby requiring a dynamic allocation scheme to be used.

As a termination of this discussion of a centralized vs. a distributed control approach, the conclusion should be fairly obvious:

> *After providing the initial destination list, the responsibility of destination addressing is taken over by the controller. By using the multi-purpose status/ control line, the destination modules are enabled to take copies of the next packet being transferred on the ring bus. The last destination module are notified via the same line that it is the last module along the transfer path. This means that the packet it is about to receive shall not be transferred to the next ring bus module. To avoid certain modules being bottlenecks to overall system performance, these modules may be off-loaded by inserting additional, physically identical modules into the system. If a source module is specifying one of those identical modules as destination for its data, it is at any time the responsibility of the controller to select the particular module to which to direct the data.*

7.2.1.d. Packet retransfer

When a packet is buffered due to a module being busy, it is the responsibility of the controller to retransfer the packet, that is, to initiate the pending transfer, when the module eventually becomes ready. A module becoming ready can be detected by monitoring the multi-purpose status/ control line with the appropriate code asserted on the associated control bus. Two different approaches to packet retransferring are possible:

Transfer-when-ready

As soon as the module becomes ready, data are transferred to the module. This works fine for a single destination module, but if the buffered data are going to be transferred to several modules, the chances are great that this method will result in the data being transferred to one module at a time.

Periodic retransfer

The alternative approach is to sample the module status/ control line at periodic intervals to see if any modules having a pending transfer waiting have become ready. If so has happened, the data are transferred to those modules. In this way, the possibility of servicing more than one module with one (each) pending transfer is much higher than when using the "transfer-when-ready" scheme. Conclusion:

> *Packet retransfer is implemented by using both mechanisms: "Transfer-when-ready" if only one module is waiting for the pending transfer, and by a periodic retransfer mechanism if the pending transfer is destined for more than one module. This must be supported by the pending transfer queue mechanism.*

7.2.1.e. Exception handling

Exception handling is a common denominator of the process of events requiring immediate action, either from the controller only or from all modules. Examples of such events are

- System reset.
- Power fail.
- Module fail.
- Buffer overflow.

In case of a power fail situation, it is of vital importance that preparing for the later recovery is initiated as soon as possible. Exception handling signalling must therefore be ensured the highest possible priority. Among the mechanisms available for message transfer in the ring bus system, the distributed arbitration bus is obviously the one best suited for this purpose. By using this bus, a message can be transferred with a maximum delay equal the time it takes to perform one arbitration. This will be in the order of 30 to 100 nanoseconds, depending on the implementation and the technology chosen. Another benefit of using the arbitration bus is that the messages can be observed by all modules simultaneously.

> *Exception handling events are signalled by transferring emergency messages on the arbitration bus.*

7.2.1.f. Microlevel control summary

The following mechanisms supporting microlevel control within a cluster has been described:

Address bus. Transferring address information (source address, destination address list) between a PE module requesting the ring bus and the controller module.

- **Address.** Transferring the actual addresses. Shared bus, size equal to $\log_2$(number of modules).
- **Busy.** Address bus status line (busy / idle).

Arbitration bus. Used by the PE modules and the controller module according to a distributed arbitration scheme to request access to the address bus or to issue an emergency message.

- **Arbitration code.** Identifying the requesting module or the emergency message. Shared bus, size dependent of the number of modules and emergency messages.
- **Arbitrate.** Asserted by the requesting module(s) signalling to the other modules that an arbitration is in progress.

Status/control bus. Transferring control information from the controller to the PEs and status information from the PEs back to the controller.

- **Function code.** Determining the current function of the status/control line. Driven by the controller, received by the PEs. Number of lines dependent of the number of implemented functions.

- **Status/control.** Transferring the actual status/ control information. Single, bidirectional line directly connecting the controller and each PE in the cluster.

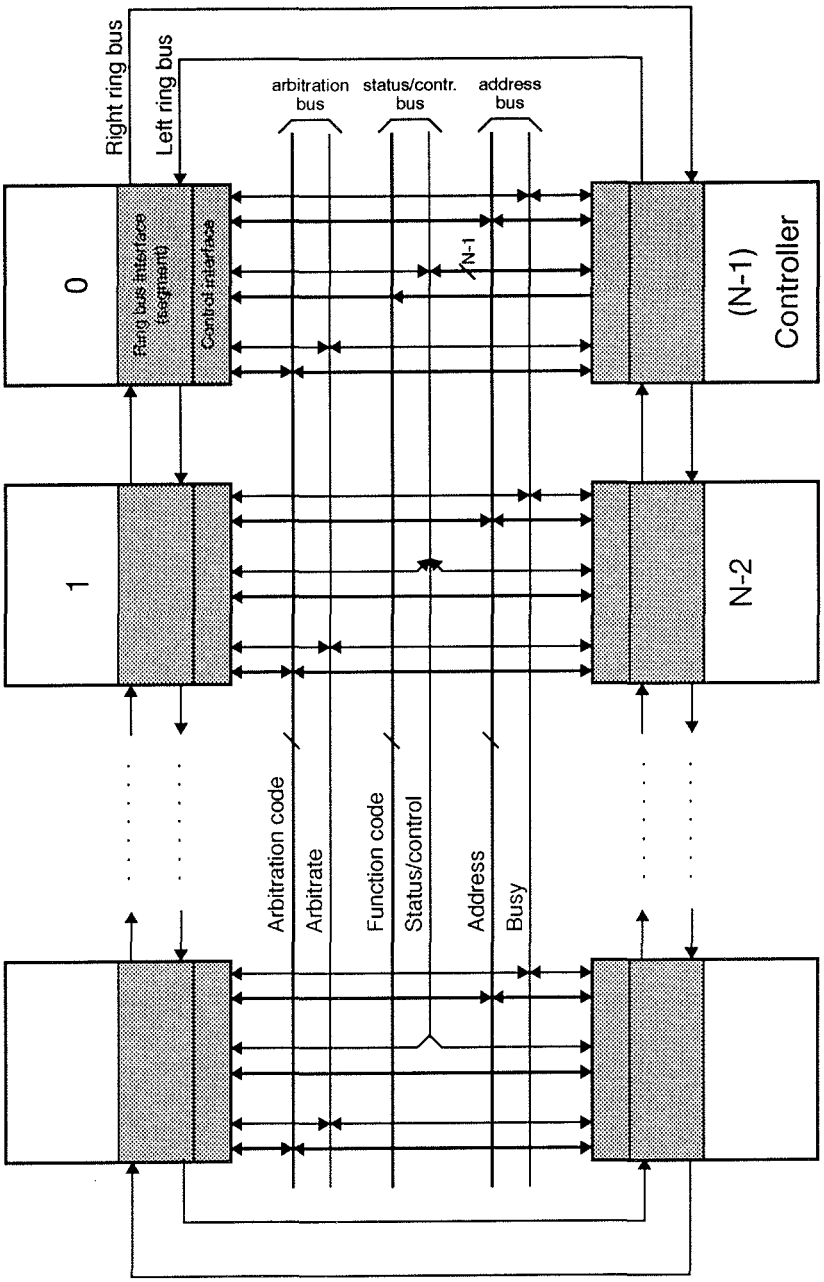


Figure 7.13. Inter-cluster microlevel control

7.2.2. Macrolevel control

While the term "microlevel control" has been used to denote control of system hardware resources, macrolevel control is applied as a common denominator for *global task level management*. Global is here used in the sense that more than one module is involved. Control local to a single module, whether it is on the micro- or macrolevel, will not have any impact on the ring bus system as such and is therefore not discussed here.

According to this definition, one example of macrolevel control will be the coordination of processing on the various modules. Others are operations as system initialization and (re)configuration. Unlike the microlevel control, macrolevel control mechanisms are not implemented by dedicated hardware constructs but rather by software algorithms running on some sort of microcontroller, locally located on each module. As discussed in section 6.3., the responsibility of macrolevel control is distributed over all modules. The reason for this is that it will be impossible for a centralized controller at any time to have a consistent and up-to-date status of all modules throughout the system.

Because macrolevel control by definition encompasses several modules, a macrolevel control event will always involve some sort of module communication. Due to the variable amount and nature of the information necessary to exchange between the modules participating in the macro level control event, this communication can not be carried out by dedicated signal lines but must be implemented by some kind of software protocol. A *control packet* signalling a macro level control event must include the following information:

- The identity of the module issuing the control event (source).
- The identity(ies) of the module(s) to be notified by the control event (destination(s)).
- An operation code specifying the purpose of the command, or what action to take in response to the command.
- Additional parameters, number and nature depending on the particular command issued (the operation code).
- If the command has a variable number of accompanying data or parameters, the size of the control packet must be included. Otherwise, the size will be a function of the particular operation command issued.

If not a new communication channel is to be introduced, there are two possibilities for transferring these control packets:

Address bus

The address bus is a shared, synchronous bus, used for transferring destination module addresses from a source module, issuing a ring bus transfer request, to its local controller. Because only addresses are intended to be transferred on this bus, only a few bits of data-width is necessary. As an example, to be able to address 16 modules, a 4 bit address bus is necessary.

Ring bus

The ring bus is the main data highway in the system. Its access and use is restricted and controlled through a dedicated arbitration mechanism, involving address transfers on the address bus. The ring bus is 16 bit wide.

Due to its significantly larger bus width, the ring bus will be a better alternative for control packet transfers than the address bus. However, ring bus latency time may a problem, especially because the control packets will be relatively small (typically 4-6 words of transfer). As previously mentioned, the ring bus latency can be regarded as consisting of two components, the access latency and the transfer latency. The transfer latency is an inherent characteristic of any ring bus system and can not be avoided. The access latency is the delay from a transfer is requested until the transfer actually begins. The minimum value of this delay is the time it takes to go through the arbitration procedure. However, if the ring bus segments required to do the transfer should be busy, the transfer must be made pending until the current transfer is terminated. If this is a just started 128 word data packet, the total latency for transferring the small control packet will be intolerable. It is no solution to allow an ongoing transfer to be pre-empted, this will only create a lot of administration overhead back and forth. To temporarily terminate the transfer and then restart it, disabling and enabling of all involved modules must be performed.

To let control packets have priority over data packets on the ring bus without a pre-emption capability, will in practice be of little value. This will only have effect when control and data packets are requesting a ring bus transfer simultaneously.

The access latency can be avoided, however, by letting control packets enter the ring bus in a way *transparent* to an already ongoing data packet transfer. For this to be possible, three conditions must be satisfied:

- Each ring bus segment interface must include a mechanism for temporal storage of incoming data. This will be necessary to hold back an ongoing data packet while transferring the control packet. The size of this temporal storage does in principle not need to be more than the length of the largest control packet.
- To be able to detect a control packet intervening a data packet as described, data belonging to the two types of packets must be *tagged* differently.
- Control packets must, unlike data packets, have their own addressing mechanism, included as a part of the packet header. The requirement of transparency towards an ongoing data packet transfer prohibits the use of the controller controlled module enable/ disable addressing mechanism.

As far as tagging is concerned, this must be implemented by extra backplane signal lines, running in parallel with the ring bus data lines. Needless to say, it is advantageous to keep the number of extra lines as small as possible, that is, use as few tag bits as possible. To distinguish between control and data, only one bit is required. Control packets will then not be allowed to disrupt other control packets. However, because control packets are so short and at least as a starting point can be regarded as being of equal importance, this will be of little use anyhow. Conclusion:

Control packets are to be transferred on the ring bus. They are permitted to enter the ring bus without prior arbitration, and without any other restriction than that they are not allowed to disrupt another ongoing control packet transfer.

7.3. System structure

There should be no fixed limit on the number of modules in the total ring bus system. However, it will not be a good solution to expand the system by just inserting new modules into the ring bus. There are two main reasons for this.

- Dedicated backplane lines are used for handshake signalling between the processing elements (PEs) and the controller. These lines are connecting the local controller, which must reside in a fixed location on the backplane, to all its PEs. If new PEs are to be included, beyond the number by which the system is originally designed, more handshake lines must then be added and the controller's interface to these lines changed.
- Ring bus transfer latency is increasing linearly with the number of inter-connected modules.

The total system should therefore be implemented as a hierarchy consisting of groups of modules. Such a group will from now on be referred to as a *cluster* and will consist of one controller module and a number of PEs. No matter how clever the inter-connection between clusters are designed, the access of modules of modules outside the local cluster is bound to be slower than accessing a module within the cluster, that is, a local module. Among other things, two ring bus arbitrations must be performed when accessing a remote module, one for the local cluster bus and one for the remote cluster bus. The number of modules contained in each cluster will therefore be a trade-off between keeping the latency low and having a small number of handshake lines at one hand and a large number of easily accessible modules at the other. Because the number of modules actually needed to be easily accessible will depend very much on the application, the cluster size must be determined by some sort of ad-hoc approach. Assuming that it from practical reasons (addressing, width of data paths in integrated circuits etc.) will be advantageous to let the number of modules be a power of two, 16 seems to be a good compromise. 16 slots (including the controller) is enough to room all necessary modules up to what can be called a medium sized system, and without at the same time having to drag around with too much superfluous cabinet space when designing a small system. Conclusion:

Ring bus modules are grouped in clusters, with one controller and 15 PEs in each cluster. A cluster will therefore be a self-contained ring bus system, supporting all features brought forward in the qualitative discussion carried out in this chapter.

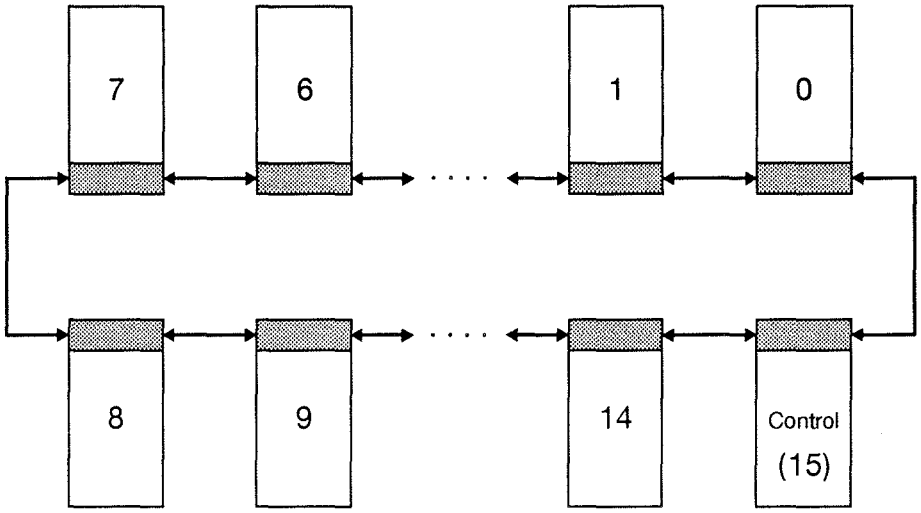


Figure 7.14. Ring bus cluster

7.3.1. Address/ arbitration bus widths

By choosing the maximum number of modules in each cluster to be 16, this determines at the same time the width of the address bus to be 4 bit. Only module addresses are to be transferred on the address bus, and 4 bit is needed to represent 16 different addresses.

As far as the distributed arbitration bus is concerned, 4 bit is now needed to represent the module's address. In addition, one bit should be included to be able to distinguish between low (request) and high (release) priority arbitration. Yet another bit is required to implement the emergency message mechanism. That gives an arbitration bus of a total of 6 bit.

7.3.2. Emergency message formats

Only a limited number of arbitration code combinations are available for message transfers on the arbitration bus. As is the case for the bus arbitration procedure, it must be ensured that no two modules at any time will access the arbitration bus with the same arbitration code. As far as the distribution of the available code combinations is concerned, there are two possibilities:

Symmetric distribution

In this case, the mechanism will be fully symmetric in the sense that all modules, the controller as well as the PE modules, can issue the same number of emergency messages on the arbitration bus with the same ease and access priority. Strictly speaking, this will not be true if two or more modules simultaneously wants to issue an emergency message, but this will be a very rare situation.

Controller biased

With a controller biased distribution, the controller is assigned a larger share of the total number of emergency message code combinations than the other (PE) modules. However, the principle that all PEs should be equal with respect to their communication capabilities should be maintained. This means that every PE module must be allocated the same number of code combinations.

With a 6 bit wide arbitration bus, a total of 32 code combinations are available for emergency message signalling. According to the symmetric distribution scheme, all modules, including the controller, will have two code combinations each. If the controller shall be able to use the arbitration bus as a mean of broadcasting high priority messages as system reset, power fail etc. to the modules in its local cluster, two code combinations will clearly not be enough. The symmetric distribution scheme will therefore not be suited for implementing the emergency message mechanism.

Looking at the controller biased approach, maintaining the principle that all PEs should be equal with respect to their communication capabilities, then leaves two alternatives: Either half (16) or all (32) of the communication codes are allocated to the controller alone. If all 32 codes are to be used by the controller, there will be no codes left to the PE modules for emergency signalling. Because far less hardware and software resources will be involved when using the arbitration bus than if the message must be formatted as a control packet and transferred on the ring bus, it is highly desirable that the PE modules have the possibility of using the arbitration bus as a simple and reliable way of emergency signalling, for instance in case of "module failure". The obvious compromise is therefore:

> *As far as emergency message signalling is concerned, 16 of the 32 available message codes are allocated to the controller. The remaining 16 codes are evenly distributed over the modules with one code to each module, using the module's address as the 4 least significant bits of the message code.*

7.3.3. Inter-cluster connection

As already discussed in a previous chapter, there are numerous ways of connecting a group of computing elements to each other. The term "computing element" is in literature used to denote an entity carrying out some sort of computation, this entity's construction and complexity may vary from a simple PE doing a set of basic arithmetic operations up to complex computer substructures. In this context, the cluster is the computing element.

Before starting on a detailed discussion about the pros and cons of different inter-connection topologies, it can be useful to try to extract some of the characteristics of the cluster as a basic building block for a larger network, and their implications for the construction of such a network.

Bandwidth requirement

The cluster is a very powerful computing element, facilitating very complex computational tasks to be performed within the cluster itself. Consequently, the cluster can from the total system's point of view be regarded as a simple input/ output black box, producing output results on the basis of input data. The communication bandwidth requirement will most probably be lower for inter-cluster communication than for intra-cluster communication. There are several reasons for this:

- Because the computing power, in terms of performing complex operations, of the cluster as a whole will be much larger than that of a single PE, a lot more multi-destination transfers, with an increased possibility of retransfer, and iterative processing will occur within a cluster than between clusters.
- By assigning modules functionally being "close" to the same cluster, the data needed to transfer between clusters will have more the nature of high level computed image parameters than raw image data, implying a significant data reduction.

Size of system

Due to the power contained within each cluster itself, a total system will rather consist of one or a few than many clusters. However, the system should not be designed in a way imposing a fixed limit on the number of connected clusters. The optimum inter-connection topology is likely to be a function of the number of clusters as well as the characteristics of the actual application.

> *The cluster interface should therefore be flexible enough to allow several inter-connection schemes to be implemented.*

Remote cluster addressing

Because a local cluster controller will have no direct control over modules in remote clusters, it will be very inconvenient and time consuming to carry out the same module handshake procedure towards a set of destination modules in a remote cluster as performed by the controller when doing a local cluster transfer. In a large system consisting of many clusters, there will also always be a possibility that this handshake procedure must be relayed through several intermediate clusters before reaching the destination cluster.

However, it is no reason why a local cluster controller should have to deal with busy modules in a remote cluster. This should be the responsibility of the remote cluster controller entirely, which also should be responsible for any buffering and retransfer of data destined for modules located in its own cluster. Therefore,

> *When transferring packets to a remote cluster, the packet is transferred from the source module in the local cluster to the remote controller, under control of the local controller. It is then the remote controller's responsibility to take care of the handshake procedure towards the destination modules, residing*

in its own cluster, including any buffering and retransfers. Handshake between the local and remote controller prior to the transfer is carried out by control packets.

System organization

Another consequence of the assumption that the majority of systems will contain one or a few clusters is that *a flat organization should be allowed, permitting two clusters to communicate directly with each other without assistance from some kind of "master cluster"*.

Having in mind that we are essentially looking for a topology with a limited number of elements, the most relevant inter-connection schemes are the linear array, the ring and to a certain extent the hierarchical tree. These are shown in Figure 7.15. to Figure 7.17.

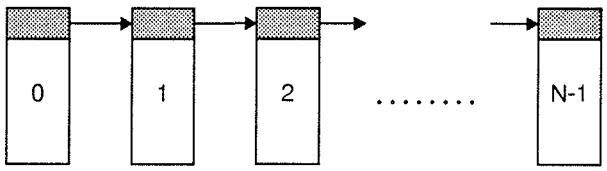


Figure 7.15. Linear array topology

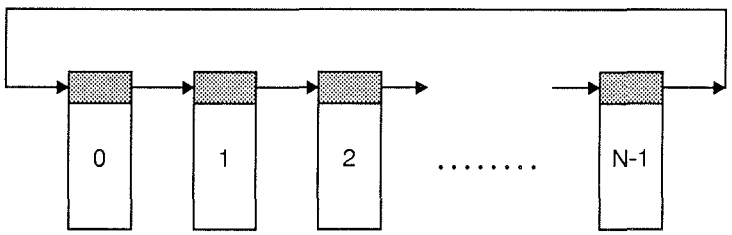


Figure 7.16. Ring bus topology

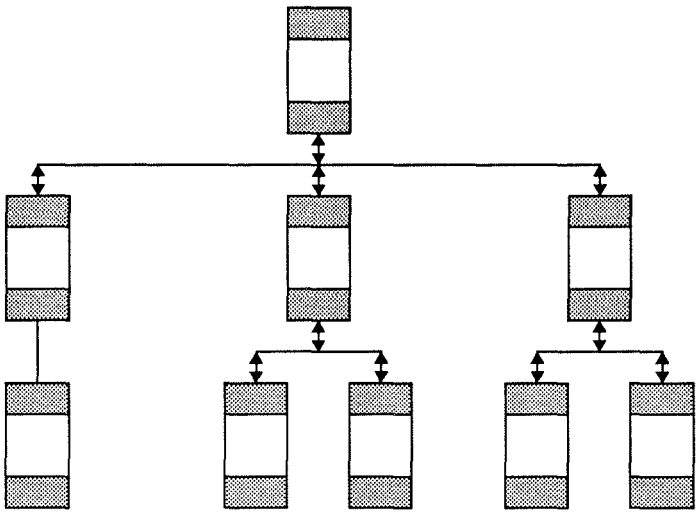


Figure 7.17. Hierarchical tree topology

To implement the linear array and the ring bus, each cluster must have two bidirectional ports. The hierarchical tree, however, will require at least three ports for connecting a node in a tree directly to the levels above and below as well as the neighbouring node at the same level. Because the hierarchical tree will be most appropriate for systems having a relatively large number of clusters, it seems not to be worth the extra cost and complexity introduced by adding a third port to achieve a more efficient implementation.

Other topologies which can not be implemented by dual-port, bidirectional cluster interfaces are mesh and cube interconnection schemes with dimension larger than two (a two-dimensional cube is actually a four-element ring). However, these topologies have, like the hierarchical tree, their strength when the number of elements to interconnect are considerably larger than in our case. Conclusion:

> *The mechanism for inter-cluster communication and connection should be flexible in terms of choosing a topology, and optimized towards small to medium sized systems in terms of number of connected clusters. This is obtained by equipping each cluster with a dual-port, bidirectional link for inter-cluster communication. Furthermore, in a two cluster system, the two clusters should be able to communicate directly with each other without the assistance of any additional master cluster controller.*

Inter-cluster arbitration

To ensure completely independent operation of the two ports, each port should have its own arbitration mechanism. The arbitration task is here very simple, because on each inter-cluster connection link, only two units competing for mastership are involved. By using a time slot

scheduling mechanism, only a few signal lines is needed to do the arbitration. It is therefore not hard to justify the extra amount of hardware needed to implement two independent arbitration mechanisms. Besides, sharing itself also requires extra resources compared to a mechanism dedicated to one port only.

In addition to independent port operations, an extra benefit of equipping each port with its own arbitration mechanism is that selection of the current cluster to arbitrate can be done on a port-by-port basis, rather than based on the cluster's address or some other form of cluster identification. With a cluster based addressing scheme in a system containing an odd number of clusters with all ports connected (e.g. ring), problems will arise when trying to find a consistent scheme for selecting which cluster to arbitrate. This is shown in Figure 7.18., where the extreme (left and right) end clusters both have even addresses (0 and 4): If clusters 0 and 4 both should request the 0-4 connection at the same time, the clusters identity (even) can not be used to determine the winner of the arbitration contest, simply because both clusters have the same identity. To change one of the cluster's identity to odd would only move the problem to the 0-1 or 3-4 arbitration, depending on which cluster's identity is changed.

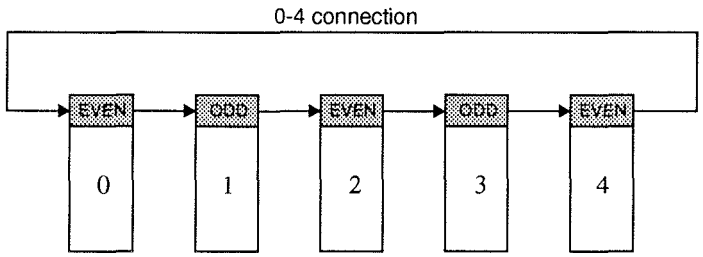


Figure 7.18. Cluster even/odd addressing

However, when the *ports* rather than the clusters are individually named and arbitrated, a port of "one" kind (even) can always be connected to a port of the "other" kind (odd) as shown in Figure 7.19.,and a consistent arbitration scheme can then always be found.

> *Each inter-cluster port should have its own, independent arbitration mechanism.*

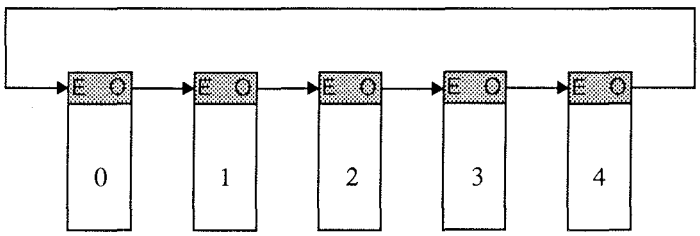


Figure 7.19. Port even/odd addressing

7.3.4. Packaging considerations

To obtain the best possible performance, it is important to keep the length of the communication paths to a minimum.

7.3.4.a. Single cluster systems

If we are positioning the modules in a cluster along a single, linear array, we will get a very long "feedback path" from the last back to the first module. Due to the much shorter path between all other modules, this will create an imbalance in the system and may be limiting to the overall system's performance.

The problem can easily be avoided, however, by dividing the maximum of 16 modules constituting one cluster (15 PEs and one controller), into two equal-sized arrays, with separate backplanes mounted back-to-back and connected at both ends. As we will see when going to multi-cluster systems, it is advantageous to position the controller at the end of one of the two arrays.

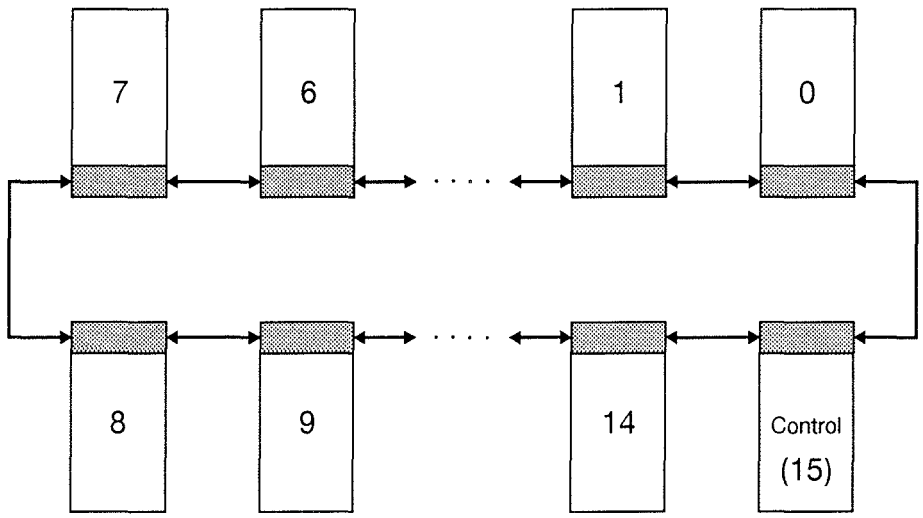


Figure 7.20. Single cluster system

7.3.4.b. Multi-cluster system

The principle of keeping the communication paths to a minimum applies equally well to the paths between clusters (that is, between the cluster controllers) as it does to the paths between modules. By positioning the cluster controller into a "corner", a four-cluster system can then be assembled by mirroring the clusters, vertically and horizontally, as shown in Figure 7.21.

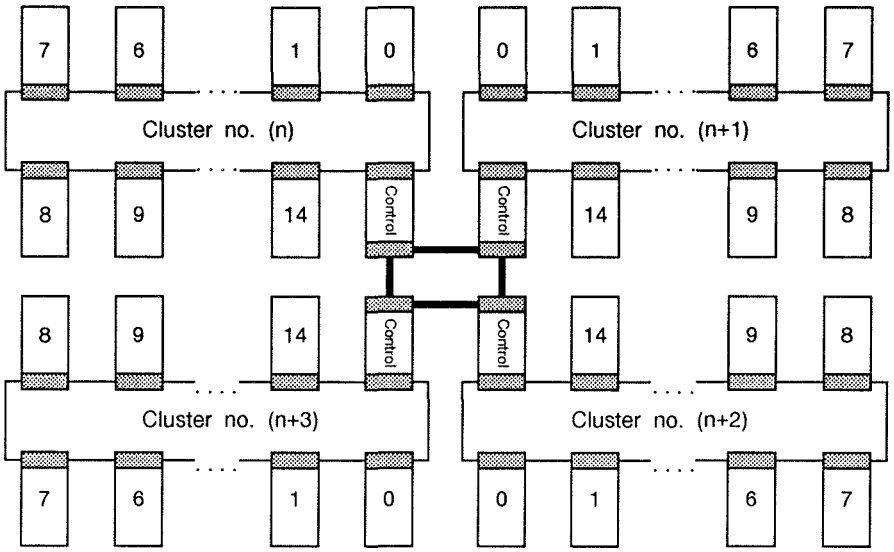


Figure 7.21. Four-cluster system

When increasing the number of clusters beyond 4, however, the clusters must be stacked vertically to maintain a short path between the cluster controllers. By stacking the clusters as four-cluster assemblies, we will get a very compact communication "backbone" in the centre of the system. Figure 7.22. shows this for a sixteen-cluster system, containing a total of 256 modules.

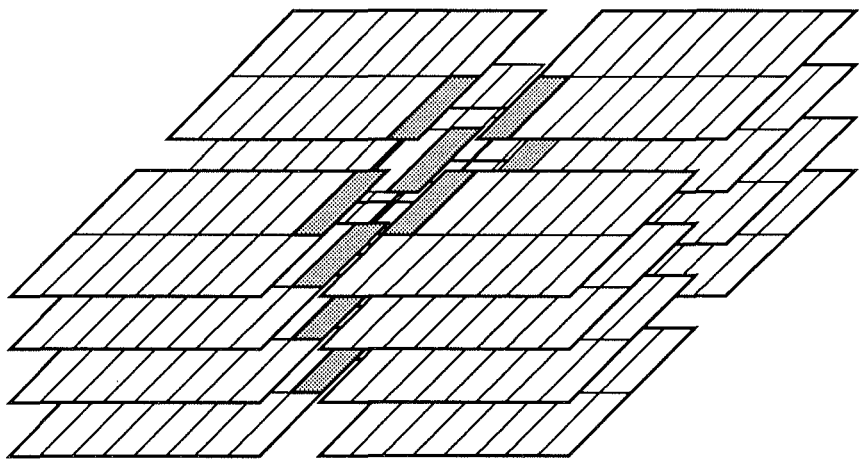


Figure 7.22. Sixteen-cluster system

The Ring bus specification defines a communication system to interconnect a number of data processing modules in a closely coupled hardware configuration, primarily intended for doing real-time image processing. The system has been conceived with the following objectives:

- To provide a flexible, high speed mechanism for transferring data between modules in the system.
- To provide means for efficient inter-module synchronization and control.
- To specify protocols that precisely define the interaction between modules interfaced to the Ring bus.
- To provide terminology and definitions that describe the system's protocol.

To be a complete bus specification, the mechanical and electrical system characteristics required to do a design should also have been included. However, this is considered to be beyond the scope of this thesis, which will focus on the logical and behavioural aspects of the system rather than the lower level implementation details.

8.1. Basic definitions

As far as the more high-level and generic ring bus terms are concerned, these can be found in chapter 8, "Ring bus system nomenclature". To set down a bus specification for this system, however, a precise nomenclature describing module interaction at a lower level is required. The following set of guidelines and definitions are tried used in consistent manner throughout the text:

Left and right convention

Due to the bidirectionality of the ring bus, many signals must include the letter "L" for left or "R" for right in their names. Each of the two buses have an input port as well as an output port, meaning that a total of 4 groups of signals are connected to each module. In the entire text, "left" and "right" are used as if the 16 modules in the cluster are connected as a linear, one-dimensional array, with module 0 at the extreme left and module 15 (the controller) at the extreme right. To close the ring, module 15 is then connected back to module 0.

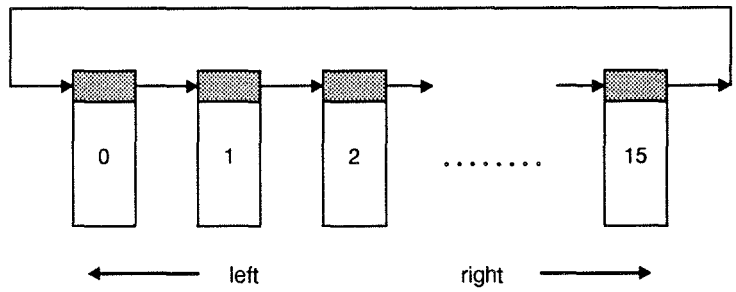


Figure 8.1. Left/right convention

Signal naming

The *connections* for the two ring buses are named according to their direction of transfer. Although it intuitively may be a good idea to name signals according to which side of the module they are physically connected, this would create problems when trying to find appropriate names for the *signal lines* inter-connecting two neighbouring modules. Therefore, to be consistent with bus naming conventions,

> *module connections are named according to which direction of transfer they serve.*

As far as the output ports are concerned, the direction of transfer and the physical location of the connection will be the same, while they for the input ports will be opposite.

A bus or bus connection, encompassing multiple signal lines, is named by a text followed by the lower and upper signal line numbers, separated by a hyphen, put in parenthesis, e.g. DLI(0-15). When referring to the entire bus, in this case all lines 0 to 15, the line number part can be omitted, using only the text part of the name (DLI).

A signal can be active low or active high, where active high is considered the default case. If a signal is active low, this is indicated by appending a "*" (asterisk) at the end of the signal name (e.g. LSI*).

Signal states

As far as signal states are concerned, it must be distinguished between signals which have a clearly defined active state, implying some action being initiated, and signals to be interpreted as representing some sort of value. To the first category belongs the *control* signals. They will be either in the *active* or *inactive* state. To the second category belongs the *bus signals*. They will be either in the *valid* or *non-valid* state. Because a bus can exhibit any value, represented by its signal lines, a set of bus signals must always be qualified by a clock or a control signal.

Signal transitions

When going from the in-active state (control signals) or in-valid state (bus signals) to the corresponding active or valid states, the signal(s) is said to be *asserted.*

When going from the active state (control signals) or valid state (bus signals) to the corresponding in-active or in-valid states, the signal(s) is said to be *released.*

Some signals have actions associated with both their edges rather than one of their levels. An example of this will be the Ring bus strobe signals where a transition, high-to-low or low-to-high, notifies that the data on the Ring bus data lines are valid. Such signals are called *toggle* signals.

8.2. Basic Ring bus structure

The Ring bus system is based upon modules grouped into clusters. Each cluster consists of a total of up to 16 modules, one controller and 15 general purpose Processing Elements (PEs). Within each cluster, data are transferred on a bidirectional Ring bus system, connecting each Ring bus module to its immediate left and right neighbours. To inter-connect several clusters, each cluster controller is equipped with a dual port. This facilitates the construction of higher level interconnection topologies as rings, trees or linear arrays.

The Ring bus functional structure can be divided into four categories, with each category consisting of a bus (a set of related backplane signal lines) and a protocol for exchanging information over those lines. The following categories can be defined:

- Ring bus transfer.
- Ring bus arbitration.
- Module handshake control.
- Inter-cluster communication.

8.3. Ring bus transfer

The Ring bus is a 16 bit, bidirectional module-to-module communication channel transferring *data* as well as *control* information formatted as *packets* between the two connected modules. In accordance to their direction of transfer, the two buses are named the *left Ring bus* and the *right Ring bus*. The left bus are transferring information from module "n" to "n-1" (n = 0-15) and the right bus from module "n" to "n+1".

The transfer of data on the Ring bus is done according to a *source synchronized, non-compelled* protocol. Source-synchronizing means that the module supplying the data onto the Ring bus data lines also supplies a signal qualifying these lines. To achieve maximum speed, both edges of the qualification signal is used for validating the data lines. By using a non-compelled protocol, transfers are not required to be acknowledged on a word-by-word basis. However, to be able to handle temporal fluctuations in the incoming data stream relative to the consumption or retransfer of data to the next module, a signal line is provided by which the receiving module can control the stream of data from the sending module. Asserting this line will freeze the data stream (and the qualification line), by releasing it the stream will continue.

To be able to distinguish between *control packets* and *data packets*, the content of the Ring bus data lines are qualified by a *tag*. Three additional signal lines are used for tagging purposes.

8.3.1. Module connections

Right Ring bus, output port

DRO(0-15)

Data bus Right Output, 0 to 15. Totem-pole. 16 data lines connecting a module's output (n) to the input of its right neighbour (module n+1).

TRO(0-2)

Tag bus Right Output, 0 to 2. Totem-pole. 3 tag lines giving information about the data currently presented to the module's right neighbour on data lines DRO(0-15).

RSO

Right Strobe Output (toggle). Totem-pole. Whenever a module presents new data on the DRO and TRO bus lines, it signals this to its right neighbour by toggling RSO.

RHI*

Right Hold Input (low). Asserted by the module's right neighbour whenever it wants to put the stream of information from the module on the DRO and TRO bus lines to a hold. When RHI* is released, the module will continue the information stream from the point where it was disrupted.

Right Ring bus, input port

DRI(0-15)

Data bus Right Input, 0 to 15. 16 data lines connecting a module's input (n) to the output of its left neighbour (module n-1).

TRI(0-2)

Tag bus Right Input, 0 to 2. 3 tag lines giving information about the data currently presented by the module's left neighbour on data lines DRI(0-15).

RSI

Right Strobe Input (toggle). Whenever new data is presented to the module on the DRO and TRO bus lines by its left neighbour, the left neighbour signals so by toggling RSI.

RHO*

Right Hold Output (low). Totem-pole. Asserted by the module whenever it wants to put the stream of information from its left neighbour to a hold. When RHO* is released, the information stream will continue from the point where it was disrupted.

Left Ring bus, output port

DLO(0-15)

Data bus Left Output, 0 to 15. Totem-pole. 16 data lines connecting a module's output (n) to the input of its left neighbour (module n-1).

TLO(0-2)

Tag bus Left Output, 0 to 2. Totem-pole. 3 tag lines giving information about the data currently presented to the module's left neighbour on data lines DLO(0-15).

LSO

Left Strobe Output (toggle). Totem-pole. Whenever the module presents new data on the DLO and TLO bus lines, it signals this to its left neighbour by toggling LSO.

LHI*

Left Hold Input (low). Asserted by the module's left neighbour whenever it wants to put the stream of information from the module on the DLO and TLO bus lines to a hold. When LHI* is released, the module will continue the information stream from the point where it was disrupted.

Left Ring bus, input port

DLI(0-15)

Data bus Left Input, 0 to 15. 16 data lines connecting a module's input (n) to the output of its right neighbour (module n+1).

TLI(0-2)

Tag bus Left Input, 0 to 2. 3 tag lines giving information about the data currently presented by the module's right neighbour on data lines DLI(0-15).

LSI

Left Strobe Input (toggle). Whenever new data is presented to the module on the DLI and TLI bus lines by its right neighbour, the right neighbour signals so by toggling LSI.

LHO*

Left Hold Output (low). Totem-pole. Asserted by the module whenever it wants to put the stream of information from its right neighbour to a hold. When LHO* is released, the information stream will continue from the point where it was disrupted.

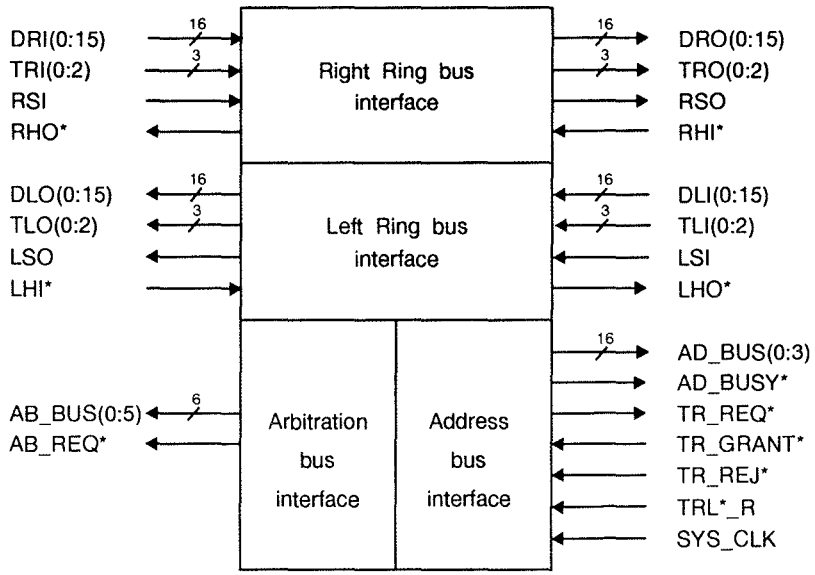


Figure 8.2. Module Ring bus interface

8.3.2. Signal lines

The modules are interconnected by connecting a module's input port to the neighbouring module's output port and the corresponding strobe and hold lines. Due to the established naming convention, the names of the connected module connections will only differ in the "I" and "O" designating an input or output connection, respectively. By removing that "I" or "O", the names of the signal lines connecting two neighbouring modules will therefore be:

Right ring bus

DR(0-15)

Data bus Right. The interconnection of one module's Data bus Right Output port DRO(0-15) and its right neighbour's Data bus Right Input port DRI(0-15).

TR(0-2)

Tag bus Right. The interconnection of one module's Tag bus Right Output port TRO(0-2) and its right neighbour's Tag bus Right Input port TRI(0-2).

RS

Right Strobe (toggle). The interconnection of one module's Right Strobe Output RSO and its right neighbour's Right Strobe Input RSI.

RH*

Right Hold (low). The interconnection of one module's Right Hold Input RHI and its right neighbour's Right Hold Output RHO.

Left ring bus

DL(0-15)

Data bus Left. The interconnection of one module's Data bus Left Output port DLO(0-15) and its left neighbour's Data bus Left Input port DLI(0-15).

TL(0-2)

Tag bus Left. The interconnection of one module's Tag bus Left Output port TLO(0-2) and its left neighbour's Tag bus Left Input port TLI(0-2).

LS

Left Strobe (toggle). The interconnection of one module's Left Strobe Output LSO and its left neighbour's Left Strobe Input LSI.

LH*

Left Hold (low). The interconnection of one module's Left Hold Input LHI and its left neighbour's Left Hold Output LHO.

8.3.3. Ring-bus data transfer timing

Data and control information are transferred between neighbouring modules on the Ring bus formatted as packets. Within each packet, the individual words are transferred using a source-synchronized, non-compelled transfer protocol. To achieve maximum transfer speed, a transition signalling scheme [Sutherland 89] is used as far as data strobe timing is concerned: Every time new data is presented on the DL (DR) and TL (TR) lines, the LS (RS) strobe line is toggled. Handshake is only required from the receiving module when it is unable to receive more data. The receiving module then asserts the data hold signal LH* (RH*), which will be held active until the module again is able to receive data. Until then, the connection will be frozen and no further data will be output by the sending module. This is shown in the following timing diagram.

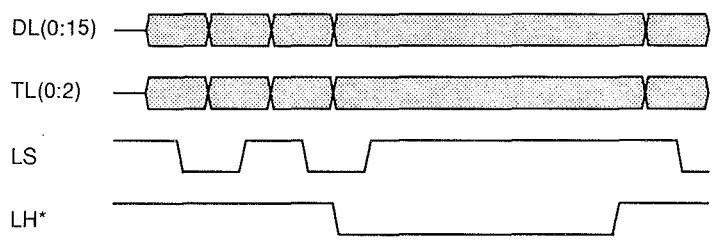


Figure 8.3. Ring bus data transfer timing

8.4. Ring bus arbitration

The Ring bus arbitration procedure is implemented by using two separate buses, each one in turn required to perform a complete Ring bus arbitration cycle.

8.4.1. Address bus

For a (source) module to get access to the Ring bus to transfer data to one or more destination modules, the module must first present a list of requested destination modules, identified by their addresses, to the controller. This list of addresses is transferred to the controller over a shared, synchronous bus called the *Address bus.*

8.4.1.a. Signal lines

Because the Address bus with its associated control lines are shared between all modules, the signal lines will have the same names as the module connections they are connected to.

Signal lines driven by the controller

SYS_CLK

SYStem CLocK. Totem-pole. All signal transitions on the Address bus and its associated control signals are related to the SYS_CLK signal.

TR_GRANT*

TRansfer GRANT (low). Totem-pole. By asserting this signal, the controller notifies the requesting module that the requested Ring bus transfer is granted.

TR_REJ*

TRansfer REJect (low). Totem-pole. By asserting this signal, the controller notifies the requesting module that the requested Ring bus transfer is rejected.

TRL*_R

TRansfer Left*/Right. Totem-pole. Through the state of this signal, the requesting module is told the direction of the granted transfer, that is, whether to use the left or the right Ring bus. Values:

0 : The module is granted a left transfer.

1 : The module is granted a right transfer.

Signal lines driven by the requesting module

AD_BUS(0-3)

ADdress BUS. Output - open collector or three-state. After winning the Address bus arbitration contest, the module is allowed to use the Address bus and its associated control signal lines to request a Ring bus transfer. The four AD_BUS lines are used to transfer the list of requested destination module addresses to the controller. Value:

0-15 : Module address.

AD_BUSY*

ADdress bus BUSY (low). Output - open collector. A module winning the Address bus arbitration contest is not allowed to start driving the Address bus until AD_BUSY* is inactive. Values:

0 : The Address bus is currently in use.

1 : The Address bus is currently not in use.

TR_REQ*

TRansfer REQuest (low). Output - open collector. By asserting this signal, a module notifies the controller that it requests a Ring bus transfer.

8.4.2. Arbitration bus

Because the Address bus, like the Ring bus, is a resource common to all modules, its use and access must be controlled by an arbitration mechanism. The Address bus arbitration mechanism is implemented by a 6 bit *Arbitration bus*, supporting a distributed arbitration algorithm. The distributed arbitration algorithm and its implementation is described in detail in Appendix B.

When a module wants to access the Address bus, it signals this, visible to all other modules in the cluster, by asserting the AB_REQ* line and outputting the appropriate arbitration code *ab_bus(0-5)* onto the Arbitration bus lines AB_BUS(0-5). The 6 bit arbitration code is formatted according to the following template:

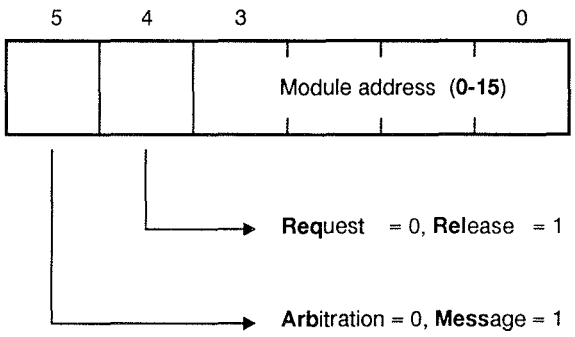


Figure 8.4. Arbitration code template

As shown in the Figure 8.4., the arbitration code can be divided into three separate fields:

Adr

Address. Bits 0 to 3. Value:

0-15 : Arbitrating module's address ("address").

Priority

Priority. Bit 4. Makes it possible for a module to do arbitrations with two levels of priority. Value:

0 : Low priority arbitration ("low").

1 : High priority arbitration ("high").

Mess/Arb

Message/Arbitration. Bit 5. Distinguishes between an emergency message and an ordinary arbitration cycle.

0 : Arbitration cycle ("arb").

1 : Emergency message cycle "mess").

OBSERVATION: Because the arbitrating module's address is included as a part of the arbitration code, it is ensured that in the case of several modules arbitrating at the same time, all arbitration codes will be unique.

OBSERVATION: With the two most significant arbitration code bits (4 and 5) being equal, the controller module (address = 15) will always have priority over all other modules in the cluster.

OBSERVATION: By setting the priority bit to "high", any module will have priority over all other modules having this bit set to "low".

OBSERVATION: Emergency message broadcasts will have priority over any ordinary arbitration cycles.

After a stabilizing period, necessary for the arbitration algorithm to settle, the arbitrating module(s) will check the value on the AB_BUS lines. If this value is equal to the module's own arbitration code *ab_bus*, this means that no other module is requesting access to the Address bus with a higher priority than the module itself, and it may therefore start using the Address bus (possibly after waiting until signal AD_BUSY* is released).

8.4.2.a. Signal lines

Because the Arbitration bus and its associated control line are shared between all modules, the signal lines will have the same names as the module connections they are connected to. All lines are driven by the module requesting the bus.

AB_REQ*

ArBitration REQuest (low). Output - open collector. By asserting AB_REQ*, a module signals a wanted access to the Address bus. The signal stays asserted until the arbitration contest is decided and a single module remains as winner.

AB_BUS(0-5)

ArBitration BUS. Output - open collector. After asserting AB_REQ*, a module outputs its arbitration code *ab_bus(0-5)* onto the AB_BUS(0-5) lines, and the arbitration contest begins. In case of several arbitration being issued simultaneously, the module having the highest arbitration code number wins the arbitration contest.

8.4.3. Arbitration procedures

There are four different arbitration procedures, the term "arbitration procedure" being here used as a common denominator for the use of the arbitration mechanism to get access to the Ring bus. In addition, there will also be a fifth arbitration procedure, namely the transfer of emergency messages. This will be discussed in a later section.

The four Ring bus access arbitration procedures are:

- Transfer request.
- Transfer grant.
- Transfer reject.
- Transfer release.

Because all four arbitration procedures involves signalling on the Address bus, the procedures will consist of two distinct phases: One *arbitration phase* to get access to the Address bus and a subsequent *Address bus transfer phase* where the appropriate information is transferred on the Address bus.

8.4.3.a. Transfer request

To signal to the controller that a module wants to transfer data on the Ring bus, the module performs a "transfer request" procedure.

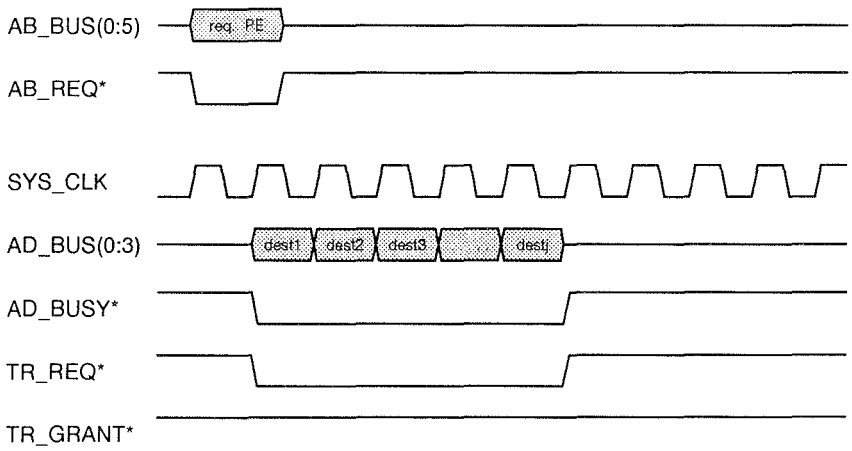


Figure 8.5. Transfer request procedure

OBSERVATION: Because arbitration and address transfer are done on separate bus lines, the two operations are allowed to overlap in time (pipelined).

Arbitration phase

In case of a transfer request, the module uses the arbitration code "arb/low/<address>" to compete for the Address bus.

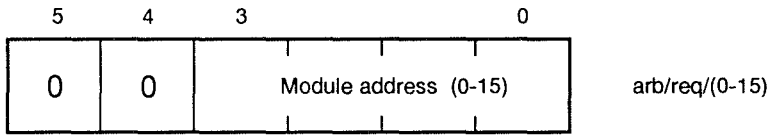


Figure 8.6. Transfer request arbitration code

Address bus transfer phase

Following the arbitration phase, the requesting module must transfer an address list to the controller of modules to which it wants to transfer data (destination modules). A destination module address is transferred on the AD_BUS(0-3) lines each SYS_CLK cycle. A valid value (address) on the AD_BUS lines is qualified by signal AD_BUSY* being active. In addition, the transfer request operation must be indicated by asserting the TR_REQ* line. The AD_BUSY* and TR_REQ* lines will stay active until all destination module addresses are transferred to the controller.

OBSERVATION: Because the activity on the arbitration lines AB_BUS can be observed by all modules (including the controller) during the entire arbitration phase, the identity of the requesting module will be known to the controller prior to the address bus transfer phase. It is therefore no need to explicitly transfer the identity of the requesting module to the controller during the address bus transfer phase.

When the address bus transfer phase is terminated by releasing the TR_REQ* line, the controller will start checking the status of the requested destination modules. If one or more of the requested destination modules currently are unable to receive new data (because they are *busy*), these modules will be removed from the destination list and replaced by the cluster's buffer module located on the controller module. On the basis of this modified destination list, the controller will check all Ring bus segments required to service the requested transfer, performed as a left as well as a right direction transfer. If there is a contiguous path of free segments from the source to the last destination module, either in the left or the right direction, the requested transfer is granted. In case of both paths, left and right, being open to the requested transfer, the shorter path is selected.

If a requested transfer can not be granted, this is from one out of two possible reasons:

- Both transfer paths (left and right) are busy.
- The cluster buffer module is needed for data buffering, but is out of buffer space.

In case of being unable to grant the requested transfer, the request is not immediately rejected but made *pending*. This means that it is placed on a list containing requested, but not yet granted transfers (*pending transfer list*) for later retry.

OBSERVATION: Note that it is the original destination list supplied by the requesting module which is saved in the pending transfer list, *not* the modified destination list. This is because the latter is to be regarded as a snapshot of the modules' state of processing at the very moment it is taken, and will therefore could have changed at a later time when the requested transfer is to be retried.

Whenever a transfer is released, the pending transfer list is checked, starting with the oldest request, to see whether one or more transfers on the list can be granted. If a pending, requested transfer can not be granted within a specified timeout period, the transfer is removed from the list and a *transfer reject* message is sent back to the requesting module.

OBSERVATION: Because the fact that a transfer can not be granted within the timeout period in itself is an indication of something being wrong, the cause of the reject should be investigated by the controller and measures be taken to remove that cause.

8.4.3.b. Transfer grant

If the requested transfer can be performed, either by a left or right direction transfer, the controller signals this back to the requesting module by a "transfer grant" procedure as shown in the following timing diagram:

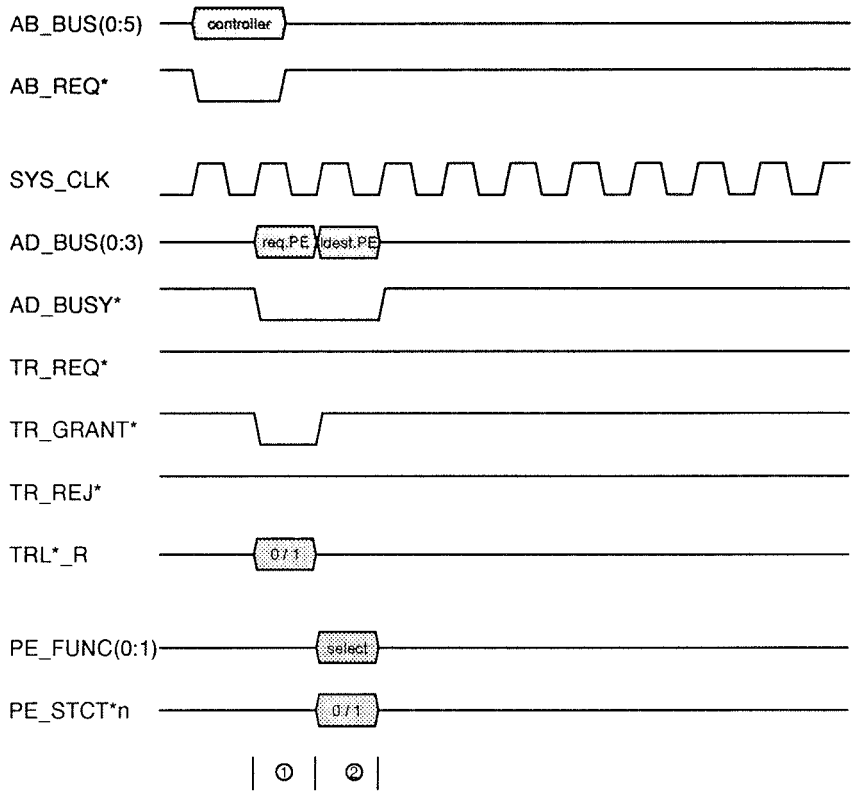


Figure 8.7. Transfer grant procedure

Arbitration phase

The controller requests access to the Address bus by using the "arb/low/15" arbitration code, presented on the arbitration lines AB_BUS(0-5):

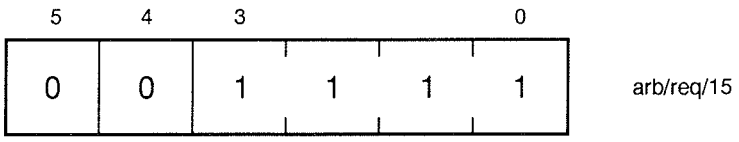


Figure 8.8. Transfer grant arbitration code

OBSERVATION: Because the controller's module address is equal to 15 and thereby higher than the addresses of all other modules (PEs) in the cluster, a transfer grant arbitration will have priority over any transfer request arbitration as far as Address bus access is concerned. Both transfer request and transfer grant arbitration codes have the priority bit set to "low".

Address bus transfer phase

As shown in the timing diagram, the transfer grant procedure is executed as two consecutive Address bus SYS_CLK cycles. In the first cycle, the particular request being granted is identified. To allow several pending transfers within the same cluster, the originating request must be identified because requests are not necessarily granted in the same order as they are issued. The request is identified by that the address of the module having requested the transfer (the source module) is presented on the Address bus, qualified as usual by asserting signal AD_BUSY*. At the same time, the transfer grant operation is signalled by asserting the TR_GRANT* line. Finally, the direction of the transfer being granted is identified through the state of the TRL*_R line: A logic zero (0) means that the transfer is to be performed as a left transfer, a logic one (1) means a right transfer.

OBSERVATION: Because the protocol is unable to distinguish one out of several transfer requests issued by the same module, a module is allowed to have only one pending transfer request at a time. If a module already has a pending request, it is therefore not allowed to issue another transfer request before the first one has been served. However, due to the possibility of transferring the same data to multiple destinations in one transfer, this is not likely to be a practical limitation to system operation and performance.

In the second SYS_CLK cycle, all modules to be destinations for the data are notified. This is done through the module handshake mechanism: By placing the appropriate (command) code on the PE_FUNC(0-1) lines and asserting the individual status/ control lines PE_STCT*n connected to the modules to be selected as destination modules, these modules are enabled for the upcoming transfer. The particular Ring bus to be used is indicated by the state of the TRL*_R line. The module being the last destination module according to the selected direction of transfer can identify itself by that its address is found on the Address bus during this second SYS_CLK cycle.

OBSERVATION: To be able to verify correct operation of the transfer request and grant mechanisms, all modules should latch the source module address from the Address bus presented during the first SYS_CLK cycle. The destination modules, selected in the second cycle, should then check this latched address against the Source Module Address (SMA) found in the header of the next packet entering the module on the Ring bus indicated by the state of TRL*_R line. If the two addresses do not match, a system failure has occurred and the controller must be notified.

8.4.3.c. Transfer reject

If the requested transfer can not be performed within the specified timeout period, the controller signals this to the module which requested the transfer by a "transfer reject" procedure, shown in the following timing diagram:

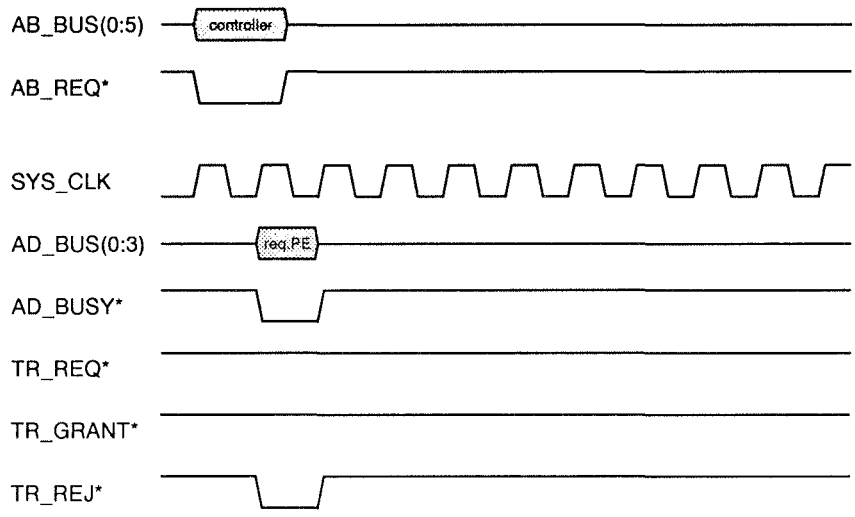


Figure 8.9. Transfer reject procedure

Arbitration phase

The controller requests access to the Address bus by using the "arb/low/15" arbitration code, presented on the arbitration lines AB_BUS(0-5):

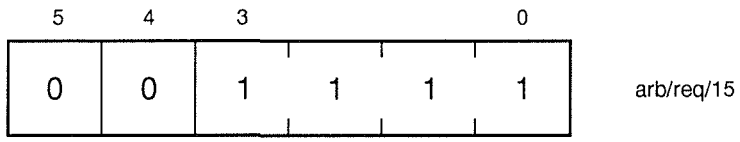


Figure 8.10. Transfer reject arbitration code

OBSERVATION: Transfer request, grant and reject procedures have equal priority as far as Address bus arbitration is concerned.

Address bus transfer phase

The Address bus transfer phase takes one SYS_CLK cycle. It is recognized by the assertion of the TR_REJ* signal. The identity of the request being rejected is signalled by that the address of the module having issued the request (the source module) is placed on the Address bus during this phase. The address is as usual qualified by the AD_BUSY* signal being active.

OBSERVATION: As already mentioned, a transfer reject is to be regarded as a system exception event. Measures should therefore be taken by the controller to identify and remove the cause of the reject, and thereby restore normal system operation.

OBSERVATION: The specific action to be taken by a module receiving a "transfer reject" from its controller in response to a "transfer request" will depend on the actual application and system configuration. It is therefore not included as a part of the Ring bus specification.

8.4.3.d. Transfer release

When the last destination module receives the last word of a data packet, the transfer is terminated. It is the responsibility of the last destination module to signal this to the controller, so the controller can release all Ring bus segments allocated to that particular transfer. The transfer release procedure consists of an arbitration phase only and is shown in the following timing diagram:

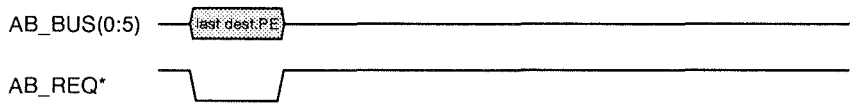


Figure 8.11. Transfer release procedure

Arbitration phase

The last destination module signals to the controller that the transfer is to be released by placing the "arb/high/<source address>" arbitration code onto the arbitration lines AB_BUS(0-5):

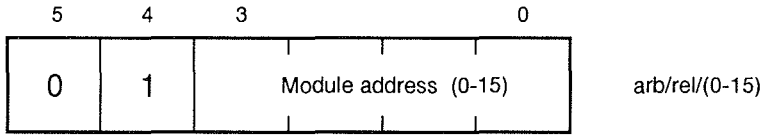


Figure 8.12. Transfer release arbitration code

OBSERVATION: Intuitively, it may seem more appropriate and consistent with other system operations to let the source module do the transfer release instead of the last destination module. However, it must be ensured that the transfer (and thereby the allocated Ring bus segments) is not released before the transfer has been completely terminated. The transfer can therefore only be released after the last destination module has received the last word in the packet. Due to the indeterminacy introduced by control packets allowed to enter the Ring bus without prior arbitration (and notification of the controller), the only module that can detect the termination of a transfer with complete certainty is the last destination module.

OBSERVATION: Transfer release is the only arbitration procedure using high priority arbitration. In case of several modules entering the arbitration procedures simultaneously, the transfer release(s) will therefore be serviced first. The reason for giving the release procedure a higher priority is that there may be situations where Ring bus segments being released by a transfer release operation is needed to grant a simultaneous transfer request.

OBSERVATION: Unlike any other arbitration procedure, a module performing a transfer release procedure is not placing its own address onto the Arbitration bus lines AB_BUS during the arbitration phase, but the address of the module being the source of the transfer. Because the transfer release procedure is the only procedure using high priority arbitration, and each module only can have one transfer active (or pending) at a time, it is therefore no danger of conflicting arbitration codes.

OBSERVATION: The destination modules are not disabled or unselected explicitly (by the controller) when the transfer is terminated. End of transfer must be detected and acted upon by the modules themselves when receiving a transfer word tagged as ELDP (End of Local Data Packet).

8.5. Module handshake control

To provide a fast and reliable way of checking module status as well as broadcasting a limited set of module control commands *to all modules in the cluster simultaneously*, each module is connected to its local controller by its own dedicated status/ control line. The current function of this line is determined by two additional signal lines, connected to all modules in parallel. Signal transitions on all lines are synchronized to the system clock, SYS_CLK.

8.5.1. Signal lines

All signal lines have names equal to the module connections they are connected to. This is the case for the PE modules as well as the controller.

PE_FUNC(0-1)

Processing Element status/ control line FUNCtion. Input - totem pole. At any time, the two bit value presented on these lines determines the current function of the status/ control lines PE_STCTn* (n=0-14). The function is selected according to the following table:

PE_FUNC	Function	Signal driven by
0	PE ready	PE
1	PE select	controller
2	spare	( )
3	spare	( )

Figure 8.13. Module handshake function select codes

PE_STCTn*

Processing Element STatus and ConTrol. Bidirectional. "n" is equal to the PE module address and will therefore be in the range of 0 to 14. The current function is determined by the value of PE_FUNC.

8.5.2. Timing diagrams

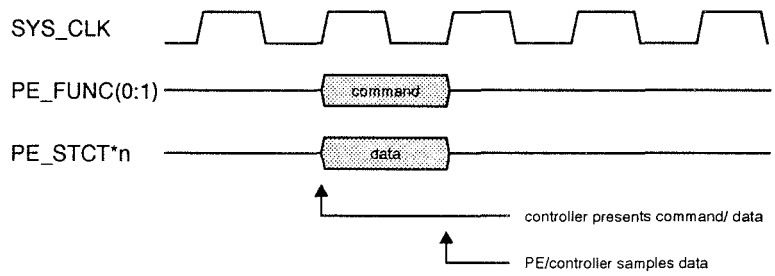


Figure 8.14. Module handshake timing

8.6. Ring bus transfer protocol

A complete Ring bus transfer cycle, from a module signals that it wants to do a transfer until the transfer is terminated and the allocated Ring bus segments released, will consist of four phases (numbers are referring to Figure 8.15.):

- **Transfer request** (1). The module wanting to do the transfer (the source module) is requesting the controller's permission to transfer a data packet to a specified set of destination modules. This is done by executing the already described transfer request procedure via the Arbitration and Address buses.
- **Transfer grant** (2). Upon receiving a transfer request, the controller will check the status of all Ring bus segments necessary to perform the transfer, from the source module to the requested destinations. If there are a contiguous line of free segments either in the left or the right direction, *and* all destination modules are ready *or* there is space available in the global buffer, the requested transfer is granted and can then be initiated by the requesting (source) module. If the transfer can not be granted within the specified timeout period, the transfer request is rejected. A reject, however, is not a part of the system's normal mode of operation, but indicates a failure of some kind. In addition to issuing a transfer reject to the requesting module, measures should therefore also be taken by the controller to find and remove the cause of the reject.
- **Data transfer** (3). The data packet is transferred on the Ring bus. To be able to detect the last word in the packet, the last word has a different tag than the other words in the packet.
- **Transfer release** (4). When the last destination module receives the last word of the packet, tagged as ELDP (End of Local Data Packet), it is the

responsibility of this module to notify the controller that the packet transfer is completed. This is done by using the arbitration mechanism. The controller can then release all Ring bus segments allocated to the transfer.

A complete Ring bus packet transfer is shown in Figure 8.15.:

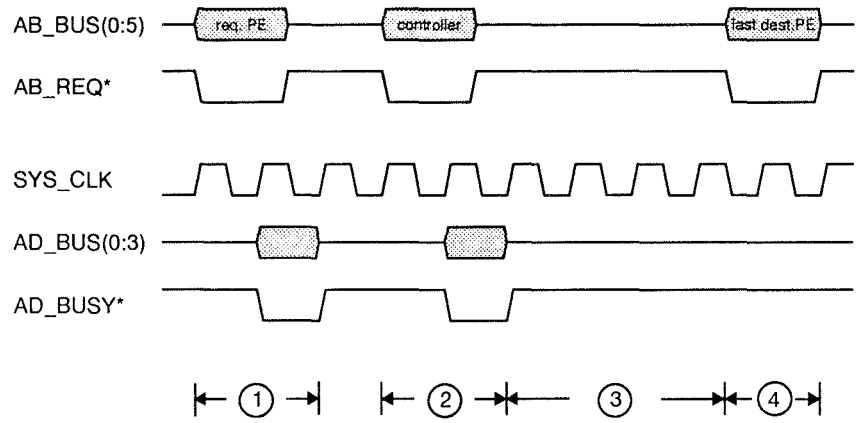


Figure 8.15. Ring bus packet transfer cycle

Information are transferred on the Ring bus as *packets*. Two types of packets are supported, *data* packets and *control* packets. Both types of packets are assembled by a header part and a data part. The header part contains information necessary for the purpose of packet addressing and identification, while the data part contains the actual data. The two types of packets are distinguished by their tagging.

8.6.1. Packet tagging

Through the associated 3 bit Tag bus, information transferred on the Ring bus can be identified as belonging to a data packet or a control packet. Tagging is done according to the following scheme:

Tag	0	1
Bit 2	local packet	remote packet
Bit 1	data packet	control packet
Bit 0	default	last word in packet

Figure 8.16. Packet tagging scheme

Coded into three bits, the following 8 tags are used:

LDP

Tag code 0. Local Data Packet. Tags all but the last word in a local data packet.

ELDP

Tag code 1. End of Local Data Packet. Tags the last word in a local data packet.

LCP

Tag code 2. Local Control Packet. Tags all but the last word in a local control packet.

ELCP

Tag code 3. End of Local Control Packet. Tags the last word in a local control packet.

RDP

Tag code 4. Remote Data Packet. Tags all but the last word in a remote data packet.

ERDP

Tag code 5. End of Remote Data Packet. Tags the last word in a remote data packet.

RCP

Tag code 6. Remote Control Packet. Tags all but the last word in a remote control packet.

ERCP

Tag code 7. End of Remote Control Packet. Tags the last word in a remote control packet.

OBSERVATION: Start-of-packet is not explicitly coded. This means that each module's Ring bus interface need to maintain a state variable telling whether a data or a control packet transfer, or both, is taking place at the moment. The state variable is set when detecting the first data or control packet word with the variable cleared, it is cleared by system reset or detecting a word tagged as ExxP (End-of-local/remote-control/data-Packet).

When detecting a word tagged as ELDP (End of Local Data Packet), it is the responsibility of the module selected as the last destination module to notify its local controller so that the Ring bus segments allocated to the transfer can be released.

8.6.2. Data packets

Data packets are used for transferring image data, lookup tables, program code to load into the various PE modules and other kinds of information. Prior to transferring a data packet, the module wanting to do the transfer must request its local controller for permission to use the Ring bus. When the request is granted, the transfer can take place.

8.6.2.a. Local data packets

Local data packets are data packets transferred to destination modules within the local cluster, that is, to destination modules located in the same cluster as the source module.

Packet tagging

Local data packets can be identified by their associated tags LDP (Local Data Packet) and ELDP (End of Local Data Packet).

Packet addressing

As a part of the Ring bus arbitration procedure, the requesting module must supply a list of destination modules to which it wants to transfer the data. By reading the modules status/ control lines PE_STCTn* (n=0-14) with the appropriate code output on the function control lines PE_FUNC, the controller can decide whether the destination PE modules are ready to accept the data or not. By applying another code to the PE_FUNC lines and asserting the status/ control lines connected to all ready destination PEs, the controller will enable those PEs to receive the packet being transferred. The last destination PE along the transfer path is notified by a second assertion of the status/ control line with the appropriate code output on the PE_FUNC lines.

If there are one or more requested destination modules not ready to receive the data, they will not be enabled for reception by the controller. Instead, it will enable the cluster buffer module, located on the controller module, which will buffer the data for later retransfer. In this way, the module requesting the transfer is released from the responsibility of keeping the data as soon as the data is transferred. This will be the case no matter whether the data actually reaches the requested destination modules or is buffered in the cluster buffer module, waiting for the destination module(s) to be ready.

Packet format

Data packets have a fixed, maximum size of 128 transferred data words (256 bytes) pr. packet. In addition comes the packet header. The packet header is formatted according to the following template:

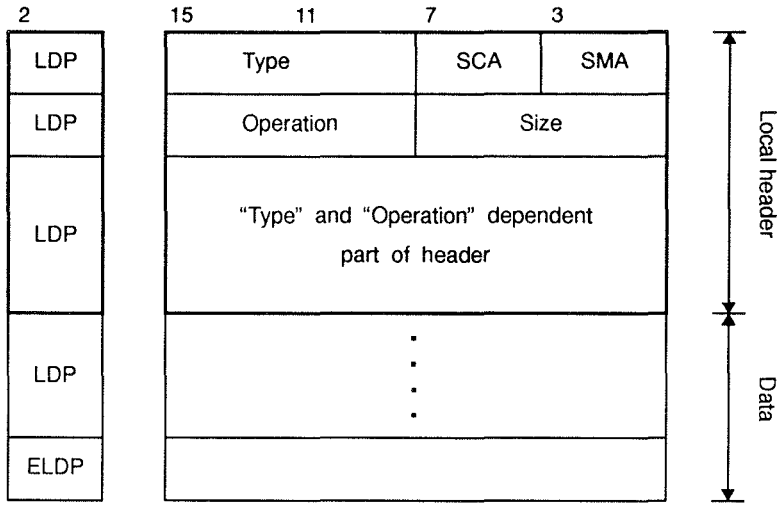


Figure 8.17. Local data packet template

The local data packet header contains the following fields of information:

Type

Type of data contained in packet.

SCA

Source Cluster Address. Address of cluster in which the source module is located.

SMA

Source Module Address. Address of module within the cluster.

Operation

Identifies the operation to perform on the data in the packet.

Size

Number of transferred words contained in the data part of the packet.

Although the actual format layout will depend on the kind of data contained in the packet, the following points should be noted:

- Destination module addressing is not included as part of the header, but is performed during the module handshake procedure prior to the actual packet transfer.

- To avoid padding packets to fill up to the maximum size (e.g. in conjunction with the last packet in a series of packet transfers), the packet header contains information of the packet's actual size.
- The actual header format is determined by the header's *type* and *operation* fields, specifying the type of data contained in the packet and what to do with it.

8.6.2.b. Remote data packets

Remote data packets are data packets transferred to destination modules in a remote cluster, that is, to destination modules located in another cluster than the source module.

Packet tagging

Remote data packets can be identified by their associated tags RDP (Remote Data Packet) and ERDP (End of Remote Data Packet).

Packet addressing

When transferring packets to a remote cluster, the handshake procedure towards destination modules in the remote cluster is carried out by the remote cluster controller. A data packet destined for a remote cluster must therefore be equipped with an additional header, called the *remote cluster header*, preceding the (local cluster) packet header described in the previous section. The remote cluster header must contain the address of the (remote) destination cluster as well as the addresses of the destination modules within the remote cluster.

When the data is transferred to its final destination modules in the remote cluster, the remote cluster header is removed. The packet words must then be retagged to LDP and ELDP, respectively, before the packet is transferred onto the remote cluster Ring bus.

OBSERVATION: No change or reformatting of the local cluster header is required. Within the remote cluster, multicast and broadcast transfers are thereby supported in the same way as for local data packet transfers.

Packet format

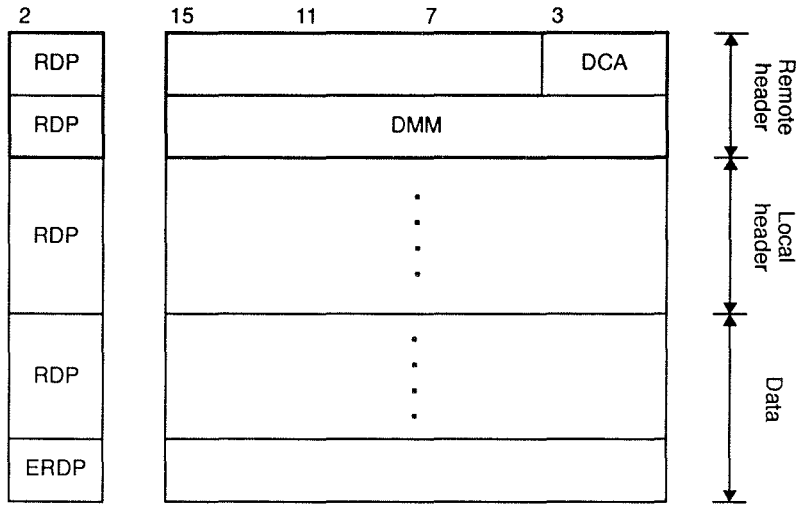


Figure 8.18. Remote data packet template

The remote data packet header contain the following fields of additional information:

DCA

Destination Cluster Address. The address of the cluster in which the destination module(s) reside.

DMM

Destination Module Mask. A 16 bit mask identifying the destination module(s) in the remote cluster, each bit corresponding to the module with address equal to the bit's position within the 16 bit mask. Values:

0 : The module is not a destination module for the current transfer.

1 : The module is a destination module for the current transfer.

8.6.3. Control packets

While data packets are used for transferring what with a common denominator can be called "bulks of data", *control packets* are transferring limited amounts of information to be used for system or module control. This can be information needed for synchronization between processing tasks on different PEs, system (re)configuration commands issued by the controller to the PEs and other control related issues.

Unlike data packets, control packets are transferred on the ring bus without any prior arbitration. However, two requirements must be met when transferring a control packet:

- The module issuing the control packet must monitor its own Ring bus input buffer, ensuring that a simultaneous incoming data packet will not cause a

buffer overflow. If that is about to happen, the control packet transfer must be temporarily halted until there is again free space in the input buffer to accumulate any incoming data.

- The tagging scheme employed does not support more than one control packet transfer simultaneously. In other words, one control packet is not allowed to disrupt another. Therefore, if the module wanting to issue a control packet already is relaying a control packet from another module through its Ring bus interface, it must wait until the transfer of that packet is terminated.

8.6.3.a. Local control packets

Local control packets are control packets being transferred to destination modules within the local cluster, that is, to destination modules located in the same cluster as the source module.

Packet tagging

Local control packets can be identified by their accompanying tags LCP (Local Control Packet) and ELCP (End of Local Control Packet).

Packet addressing

OBSERVATION: Because control packets are not subject to a controller controlled arbitration cycle before the actual transfer takes place, packet addressing can not be accomplished by using the same select/ unselect mechanism as the data packets.

Control packet addressing is implemented through the packet header: Included in the header is a 16 bit Destination Module Mask, DMM. Every bit in the DMM corresponds to the module within the cluster having the same address (0-15) as the bit's position in the DMM. Values:

0 : This module is not a destination for this control packet.

1 : This module is a destination for this control packet.

A module being destination for the control packet, must clear its own DMM (address) bit when it receives the packet. If the entire DMM thereby gets cleared, this means that the module is the last destination module for this packet and the packet is therefore removed from the Ring bus. However, if there are still one or more bits set in the DMM, the module immediately transfers the packet to its neighbouring Ring bus module along the direction of transfer.

Packet format

Control packets have a length varying with their actual operation, but is typically less than 10 transferred words pr. packet, including the header. The packet header is formatted according to the following template:

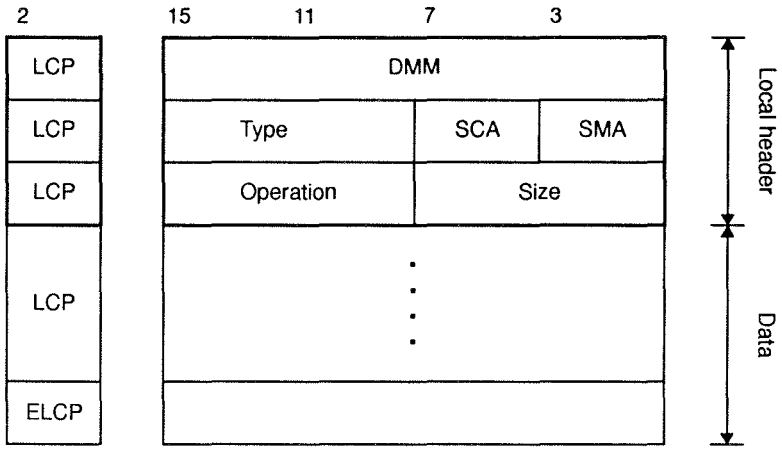


Figure 8.19. Local control packet template

The local control packet header contain the following fields of information:

DMM

Destination Module Mask. 16 bit mask used for destination module addressing.

Type

Type of data contained in packet.

SCA

Source Cluster Address. Address of cluster in which the source module is located.

SMA

Source Module Address. Address of module within the cluster.

Operation

Identifies the operation to perform.

Size

Number of transferred words contained in the data part of the packet.

8.6.3.b. Remote control packets

Remote control packets are control packets transferred to destination modules in a remote cluster, that is, to destination modules located in another cluster than the source module.

Packet tagging

Remote control packets can be identified by their accompanying tags RCP (Remote Control Packet) and ERCP (End of Remote Control Packet).

Packet addressing

A remote control packet transfer is identified through its RCP tagging. For addressing, a complete remote control packet address is assembled by a cluster part, the Destination Cluster Address (DCA), and a module part (DMM), giving the module address(es) within the remote cluster.

Like data packets, control packets destined for a remote cluster must therefore have a remote cluster header, supplying the Destination Cluster Address DCA. The remote cluster header precedes the (local cluster) packet header described in the previous section, containing among other things the DMM.

When the control packet in transfer reaches the remote cluster controller, the remote cluster header is removed. The words contained in the packet are then retagged to LCP and ELCP, respectively, before the packet is transferred onto the remote cluster Ring bus.

OBSERVATION: No change or reformatting of the local cluster header is required. Within the remote cluster, multicast and broadcast transfers are thereby supported in the same way as local control packet transfers do.

Packet format

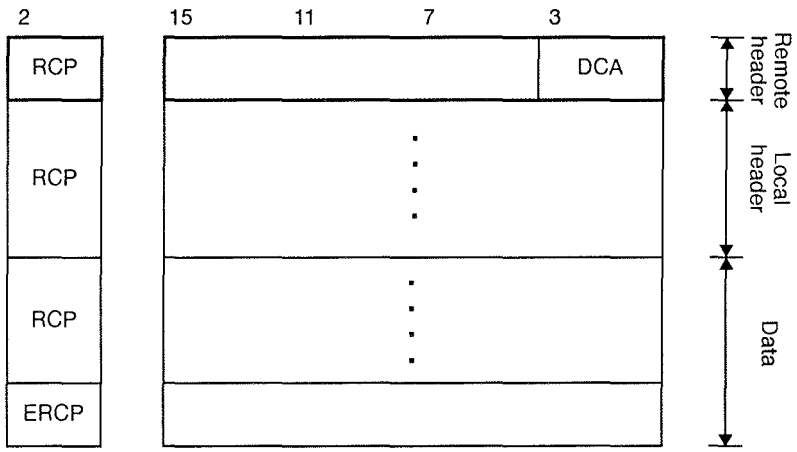


Figure 8.20. Remote control packet template

The remote control packet header contain the following field of additional information:

DCA

Destination Cluster Address. The address of the cluster in which the destination module(s)s reside.

8.6.4. Emergency messages

Emergency messages are transferred on the 6 bit Arbitration bus AB_BUS(0-5). An error message is distinguished from an ordinary arbitration cycle by having the most significant AB_BUS bit (5) set.

Any emergency message has priority over any arbitration signalling.

The total of 32 different messages are shared between the controller (16) and the ordinary Ring bus modules (one each).

8.6.4.a. Module alert message

Module alert is signalled by a module whenever a situation occurs which the module can not handle without external (controller) assistance. The actual *cause* of the alert being issued must be investigated by the use of Ring bus control packets, if possible. Otherwise, the alerting module must be resat.

A module alert message is primarily intended for the controller, but can, like all other arbitration bus activity, be observed by all modules.

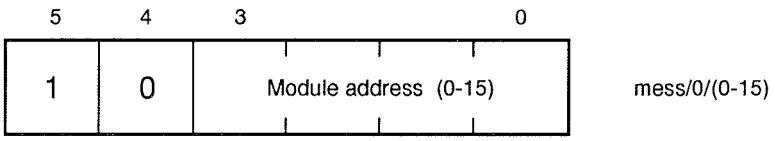


Figure 8.21. Module alert message code

8.6.4.b. Controller high priority message

Controller high priority messages are messages issued by the controller, broadcasted to all other Ring bus modules within the cluster. These messages are used to signal events as system reset, power fail and other events requiring immediate action to be taken and/ or a very reliable transfer mechanism.

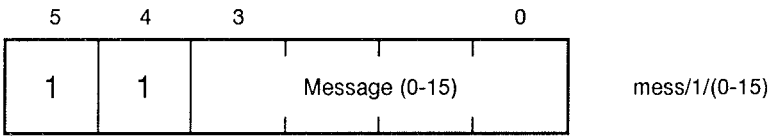


Figure 8.22. Controller high priority message code

8.7. Inter-cluster communication

The purpose of the inter-cluster (IC) interface is to be a simple, efficient and flexible way of connecting a number of clusters. To be able to connect the clusters according to the topology best suited for the actual application, each cluster interface is equipped with two identical, bidirectional ports. By having two ports, topologies as linear arrays and rings can easily be implemented.

The IC interface is shared between the two connected clusters by using a simple time-slot arbitration mechanism. This mechanism determines which of the two clusters currently controlling the arbitration mechanism, thereby having the possibility to request to use the interface for doing a data transfer. The cluster in control is called the *arbitration master*, the other cluster the *arbitration slave*.

Each of the two ports have its own arbitration mechanism. In this way, a totally independent operation of the two ports is achieved. The data transfer itself is, like the intra-cluster data transfer, based on a *source synchronized, non-compelled* protocol, supported by a pair of request/ grant lines. To match the intra-cluster Ring bus, the width of the inter-cluster data bus is 16 bit together with a 3 bit tag-bus.

8.7.1. Cluster connections

Inter-Cluster port A arbitration connections

IC_SELAB

Inter-Cluster interface SELect port A or B. Output - totem pole. IC_SELAB is a 50/50 duty-cycle clock signal used to implement the time slot scheduling mechanism. Value:

0 : Port A is selected.

1 : Port B is selected.

IC_ARBEN*

Inter-Cluster interface ARBitration ENable (low). Output - totem pole. IC_ARBEN* is a 75/25 (low/high) duty-cycle clock signal. It is low whenever arbitration is enabled.

IC_AREQ*

Inter-Cluster interface port A REQuest (low). Bidirectional - open collector. Asserted when a cluster wants access to the inter-cluster interface port A.

IC_AGRANT*

Inter-Cluster interface port A GRANT (low). Bidirectional - open collector. Asserted as a positive acknowledge to an IC interface request.

Inter-Cluster port A data connections

IC_ABUSY*

Inter-Cluster interface port A BUSY (low). Bidirectional - open collector. Active when IC port A is in use.

IC_ADATA(0-15)

Inter-Cluster interface port A DATA, 0 to 15. Bidirectional - three state. IC interface port A data lines.

IC_ATAG(0-2)

Inter-Cluster interface port A TAG, 0 to 2. Bidirectional - three state. IC interface port A tag lines.

IC_ADS

Inter-Cluster interface port A Data Strobe. Toggle, bidirectional - three state. Toggled by the cluster currently driving port A when valid data is placed on the IC_ADATA and IC_ATAG lines.

IC_ADH*

Inter-Cluster interface port A Data Hold (low). Bidirectional - three state. Asserted by the cluster currently receiving data over port A, requesting the driving cluster to temporarily hold the data stream.

Inter-Cluster port B arbitration connections

IC_SELAB

Inter-Cluster interface SELect port A or B. Input. IC_SELAB is a 50/50 duty-cycle clock signal used to implement the time slot scheduling mechanism. Value:

0 : Port A is selected.

1 : Port B is selected.

IC_ARBEN*

Inter-Cluster interface ARBitration ENable (low). Input. IC_ARBEN* is a 75/25 (low/high) duty-cycle clock signal. It is low whenever arbitration is enabled.

IC_BREQ*

Inter-Cluster interface port B REQuest (low). Bidirectional - open collector. Asserted when a cluster wants access to the inter-cluster interface port B.

IC_BGRANT*

Inter-Cluster interface port B GRANT. Bidirectional - open collector. Asserted as a positive acknowledge to an IC interface request.

Inter-Cluster port B data connections

IC_BBUSY*

Inter-Cluster interface port B BUSY (low). Bidirectional - open collector. Active when IC port B is in use.

IC_BDATA(0-15)

Inter-Cluster interface port B DATA, 0 to 15. Bidirectional - three state. IC interface port B data lines.

IC_BTAG(0-2)

Inter-Cluster interface port B TAG, 0 to 2. Bidirectional - three state. IC interface port B tag lines.

IC_BDS

Inter-Cluster interface port B Data Strobe. Toggle, bidirectional - three state. Toggled by the cluster currently driving port B when valid data is placed on the IC_BDATA and IC_BTAG lines.

IC_BDH*

Inter-Cluster interface port B Data Hold (low). Bidirectional - three state. Asserted by the cluster currently receiving data over port B requesting the driving cluster to temporarily hold the data stream.

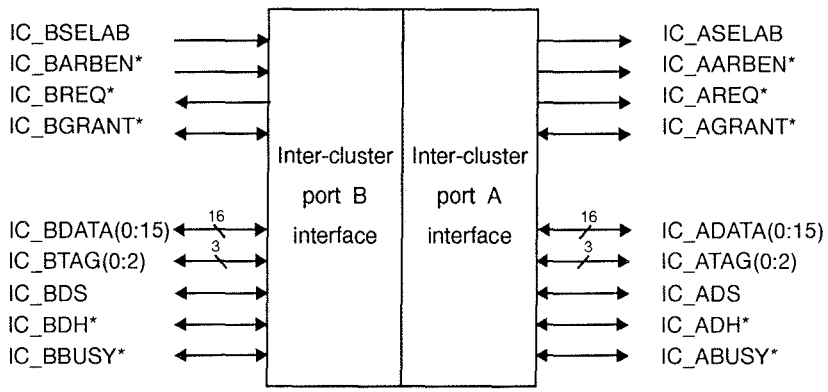


Figure 8.23. Inter-Cluster interface

8.7.2. Signal lines

The clusters are interconnected by connecting one cluster's port A to another cluster's port B connections. The names of the connected cluster connections will only differ in the "A" and "B" designating port A or B, respectively. By removing that "A" or "B", the names of the signal lines connecting the two clusters will therefore be:

Inter-Cluster arbitration lines

IC_SELAB

Inter-Cluster connection SELect port A or B. Signal line driven by the connected port A, input to port B.

IC_ARBEN*

Inter-Cluster connection ARBitration ENable (low). Signal line driven by the connected port A, input to port B.

IC_REQ*

Inter-Cluster connection REQuest (low). Output (arbitration master) or input (arbitration slave). The connection of the IC_AREQ* and IC_BREQ* lines.

IC_GRANT*

Inter-Cluster connection GRANT (low). Output (arbitration slave) or input (arbitration master). The connection of the IC_AGRANT* and IC_BGRANT* lines.

Inter-Cluster data lines

All lines are driven by the master cluster, except IC_DH*, which is driven by the slave cluster.

IC_BUSY*

Inter-Cluster connection BUSY (low). The connection of the IC_ABUSY* and IC_BBUSY* lines.

IC_DATA(0-15)

Inter-Cluster connection DATA, 0 to 15. The connection of the IC_ADATA and IC_BDATA lines.

IC_TAG(0-2)

Inter-Cluster connection TAG, 0 to 2. The connection of the IC_ATAG and IC_BTAG lines.

IC_DS

Inter-Cluster connection Data Strobe. The connection of the IC_ADS and IC_BDS lines.

IC_DH*

Inter-Cluster connection Data Hold (low). The connection of the IC_ADH* and IC_BDH* lines.

8.7.3. Inter-Cluster interface arbitration

Arbitration is done according to a time slot scheduling mechanism based on the signals IC_ARBEN* and IC_SELAB. The inter-cluster interface is a direct port-to-port connection, always connecting the A port of one cluster to the B port of the other cluster. The IC_ARBEN* and IC_SELAB signals are always driven by the A port, with input to the B port.

> *This is the one and only instance in which the two (A and B) ports sharing the IC interface is asymmetric.*

According to the current state of the IC_ARBEN* and IC_SELAB signals, each IC_SELAB cycle can be divided into two distinct phases enabling arbitration for the two connected ports (A and B) in turn. Between each enable phase, there is an intermediate arbitration disable phase. The previous definition of an arbitration master/ slave can now be expressed as follows: A port being enabled for arbitration, that is, being within its own enable phase, is called the *arbitration master*. The other port is called the *arbitration slave*.

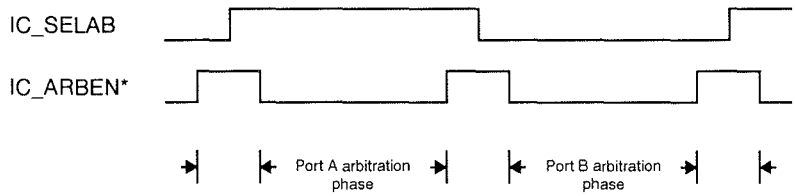


Figure 8.24. Inter-Cluster arbitration phases

The various arbitration phases can be summarized as follows:

IC_ARBEN*	IC_SELAB	
	low	high
low	Port B arb. phase	Port A arb. phase
high	arb. disabled	arb. disabled

Figure 8.25. Table, Inter-Cluster arbitration phases

A port is only allowed to make a request to use the inter-cluster connection if it is the current arbitration master. Provided that the port is free to use, that is, the connection's status line IC_BUSY* is inactive *when entering the port's arbitration enable phase*, a request can be made

by the arbitration master asserting the IC_REQ* line. If the connection is busy, the port must wait until the on-going transfer is terminated and the busy line released before making the request.

OBSERVATION: The state of the IC_BUSY* line is sampled by the arbitration master only once during each IC_SELAB cycle, at the very beginning of the enable phase. In this way, a master cluster doing a inter-cluster transfer is not required to synchronize the release of the IC_BUSY* with the IC_SELAB and IC_ARBEN* signals, but is allowed to release IC_BUSY* whenever the transfer is finished.

Before any data transfer can take place, an issued request must be acknowledged by the arbitration slave by asserting the IC_GRANT* line. Finally, before leaving its enable phase, the arbitration master must assert the IC_BUSY* line to signal that it actually starts using the requested port.

After asserting the IC_BUSY* line, the IC_REQ* line may then be released, with a subsequent release of the IC_GRANT* line by the arbitration slave If the arbitration master does not receive an acknowledge or does not have time to assert the IC_BUSY* line before its enable phase expires, it is forced to withdraw its request. *No request is allowed to be active during the arbitration disable phase.* A new request may then be issued at the earliest the next time the port becomes arbitration master, with the danger being, of course, that the connection is then already taken by the other cluster.

When a cluster is granted the connection and has asserted the IC_BUSY* line, this cluster becomes the *master cluster* of the connection until the transfer is terminated and IC_BUSY* released. The other cluster will accordingly be named the *slave cluster*. The IC_BUSY* line must stay active during the whole transfer. When the master cluster wants to release the connection, it does so by releasing the IC_BUSY* line.

As an example, a timing diagram is shown where port A is the arbitration master.

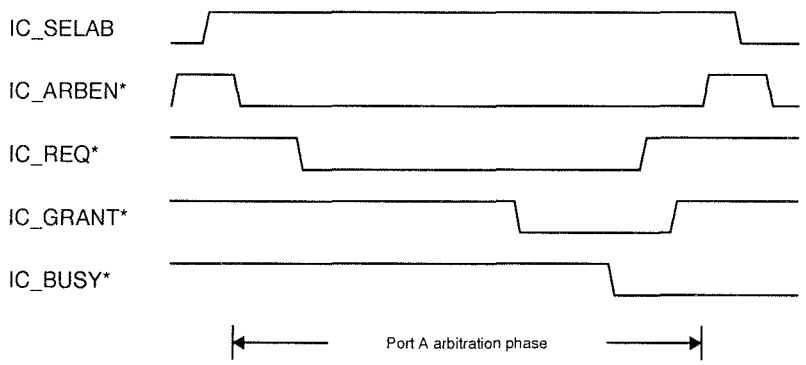


Figure 8.26. Port A arbitration

OBSERVATION: In a two cluster system, or for clusters being located at the edge of a larger configuration, only one of the inter-cluster ports are probably needed. The cluster controllers should therefore include provisions for individually disabling each one of the two ports.

8.7.4. Inter-cluster data transfer protocol

Data and control information are transferred between clusters formatted as *remote data packets* and *remote control packets* as already described. The protocol used is a source-synchronized, non-compelled, transition signalling [Sutherland 89] transfer protocol like the one used on the Ring bus: The master cluster supplying the data toggles the IS_DS strobe line when new data is presented on the IC_DATA and IC_TAG lines, a handshake is only issued by the slave cluster when the slave cluster is (temporarily) unable to receive more data. The slave cluster will then assert the data hold signal IC_DH*, which will be held active until the slave cluster again is ready to accept new data. Until then, the connection will be frozen and no further data will be output by the master cluster. This is shown in the following timing diagram.

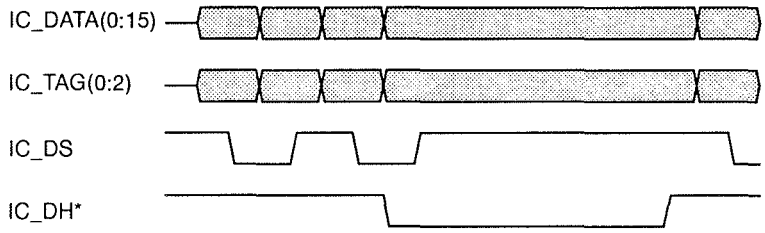


Figure 8.27. Inter-cluster data transfer timing

As described earlier, the communication within each 16 module cluster is implemented by a bidirectional ring bus. This means that a transfer from one *source* module to one or more *destination* modules can take one out of two alternative paths, transferring data in the left or right direction, respectively. Each of the two communication paths are resources globally accessible to all modules in the cluster, their use must therefore be controlled and regulated by an *arbitration* mechanism. For reasons already accounted for, the ring bus arbitration mechanism is implemented as a centralized arbiter, located on module 15, the controller module.

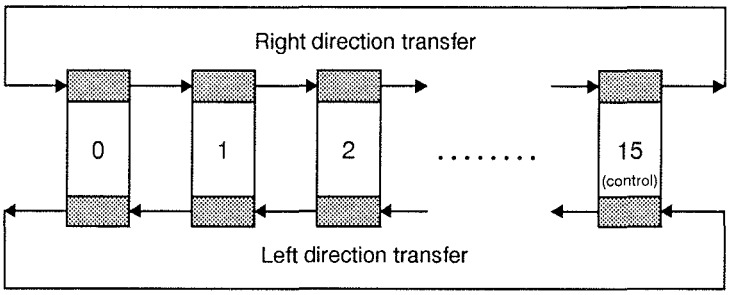


Figure 9.1. The bidirectional ring bus system

Even if Figure 9.1. may suggest that each communication path, left and right, has its own bus interface, this is true only as far as the Ring bus part is concerned. The arbitration mechanism itself with its associated backplane lines (the Arbitration and the Address bus) is common to the two paths.

Scope of design

Although the design presented in this chapter at least by a superfluous look may seem to be a fully detailed, ready-to-implement solution, this is by no means so. The description is aimed at showing that the arbitration specification possible to extract from the preceding chapters easily and with a relatively small amount of hardware resources can be implemented in silicon. The scope of the presented design is therefore

> *to show the logic and the various functional blocks necessary to build a mechanism according to the described algorithm.*

9.1. Documentation syntax

The design and function of the Ring bus arbitration mechanism is presented in two ways, electrical schematics and logical equations.

9.1.1. Electrical schematics

The electrical schematics show how the arbitration mechanism is constructed, and the interconnection structure connecting the different functional blocks. The schematics are presented on two levels of detail, the block level and the logic level.

On the *block level*, functional blocks are shown as squares or rectangles with a piece of text indicating the function of the block. To be able to tell the direction of signal flow, connections on this level are therefore equipped with arrows.

Schematics on the *logic level* are assembled by basic components as gates, decoders, encoders, multiplexers etc. Their inputs and outputs are implicitly defined by their function, connections on this level are therefore non-arrowed. The only arrows on the logic level schematics are used in conjunction with PLA- or PAL-like components where input/ output is not implicit in the component's function.

New signals will always be explained at the time of introduction, in other words at the block level. No signals will be introduced at the logic level. To help understanding the schematics, the following symbols for signal connections are used:

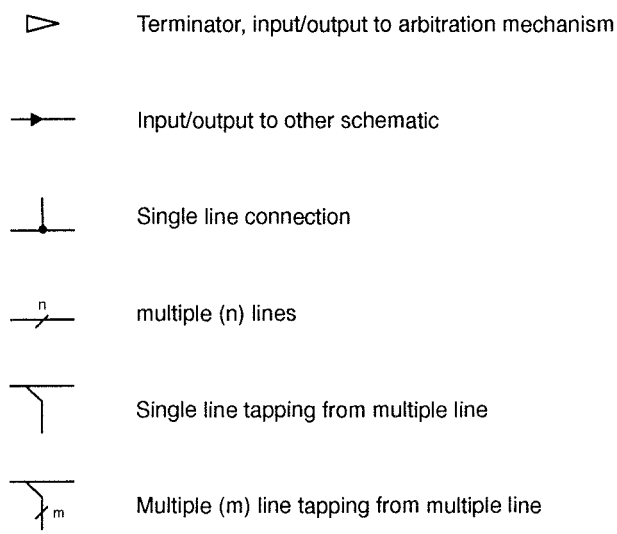


Figure 9.2. Signal connection notation

9.1.2. Logical equations

To describe the function of various control blocks, as well as a supplement to the gate-level design used in the logic level schematics, logical equations are used. The syntax is equal to the notation used in logic description languages as ABEL [DATA I/O 1989] and CUPL, and is based on the following set of operators:

```
=    assignment
!    logical (bitwise) negation (one's complement)
&    logical (bitwise) AND
#    logical (bitwise) OR
==   "is identical to"
!=   "is not identical to"
```

9.2. Functional description

By using the Arbitration and Address buses, a (source) module issues a request to transfer data on the Ring bus to a specified set of destination modules. Which Ring bus to use, the left or the right, is not to be specified by the requesting module, this is up to the controller to decide. After some necessary conversion and reformatting through the Ring bus interface, the following signal lines are presented to the Ring bus arbitration mechanism:

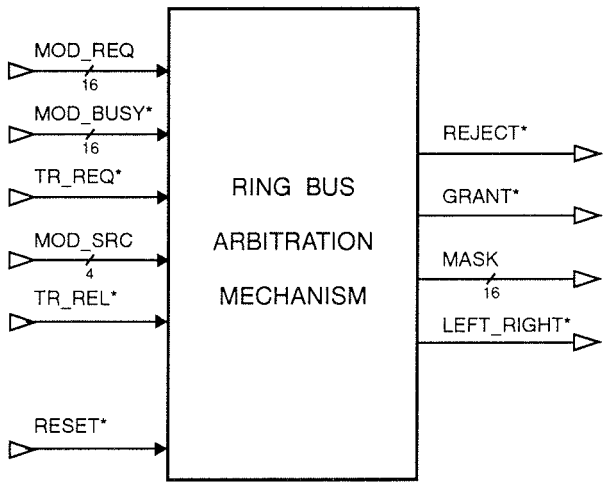


Figure 9.3. Ring bus arbitration mechanism

Input signals

MOD_REQ(0-15)

MODule REQuest. The list of destination module addresses provided by the requesting module is assembled into the 16 bit mask MOD_REQ, each bit corresponding to the module having the same address as the bit's position in the mask. Values:

0 : The module is not requested as a destination module.

1 : The module is requested as a destination module.

MOD_BUSY*(0-15)

MODule BUSY (low). A 16 bit mask telling whether the corresponding module is busy or not. Busy in this context means that the module is unable to receive data input due to insufficient local buffer space or other causes. Each bit in MOD_BUSY* corresponds to the module having the same address as the bit's position in the mask.

MOD_BUSY*(15), corresponding to the cluster controller, has a slightly different meaning, and thereby influence on system operation, than the other 15 bits. It tells whether the global buffer, located on the controller module, is able to receive more input data. If it is not, and at least one of the destination modules are unable to receive input data (and thereby needs assistance from the global buffer), the request must be rejected. Values, bits 0 to 14:

0 : The module is unable to receive input data.

1 : The module is able to receive input data.

Value, bit 15:

0 : The global buffer is unable to receive more data.

1 : The global buffer is able to receive more data.

MOD_SRC(0-3)

MODule SouRCe. A 4 bit value identifying the requesting (source) module. Value: 0 to 15.

TR_REQ*

TRansfer REQuest (low). Asserted to signal a transfer request.

TR_REL*

TRansfer RELease (low). Asserted to signal termination of a transfer with a corresponding release of all Ring bus segments allocated to that particular transfer.

RESET*

RESET arbitration mechanism (low). When asserted, all status information contained in the arbitration mechanism is cleared to the initial state: No ongoing transfers, all Ring bus segments are idle and free to use.

Output signals

GRANT*

transfer/release GRANT (low). Asserted when the requested transfer is granted, or when the transfer release has been successfully serviced.

REJECT*

transfer/release REJECT (low). Asserted when the requested transfer is rejected, or when the transfer release can not be serviced.

LEFT_RIGHT*

transfer/release LEFT or RIGHT (low). The interpretation of this signal is depending on whether it is issued in conjunction with a transfer request or a transfer release. Value, transfer request:

0 : Use right Ring bus for the requested transfer.

1 : Use left Ring bus for the requested transfer.

Value, transfer release:

0 : The transfer released was a right Ring bus transfer.

1 : The transfer released was a left Ring bus transfer.

MASK(0-15)

transfer/release MASK. In conjunction with a transfer request, this 16 bit mask shows which modules to participate in the transfer, from the source to the last destination module (both inclusive). Values:

0 : The module is not to participate in the transfer.

1 : The module is to participate in the transfer.

For a transfer release, the mask shows which modules were used by the transfer. Values:

0 : The module was not used by the transfer.

1 : The module was used by the transfer.

MASK is therefore a 16 bit value consisting of a contiguous sequence of either 1's or 0's. The latter is the case if the transfer path is going "round" the ring, called a wrap-around transfer.

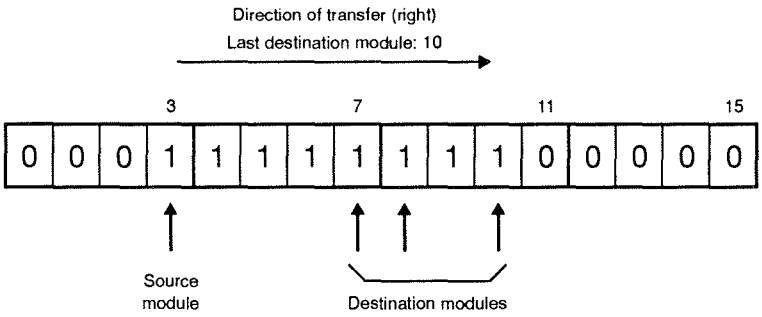


Figure 9.4. MASK, non wrap-around transfer

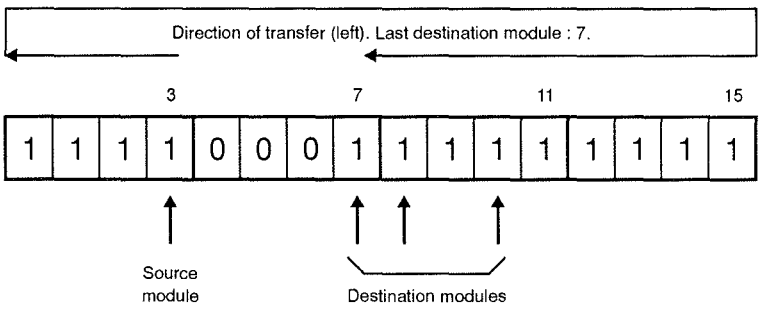


Figure 9.5. MASK, wrap-around transfer

9.3. Arbitration timing

As far as the operation of the Ring bus arbitration mechanism is concerned, transfers are either *requested* or *released.*

9.3.1. Transfer request

When requesting a transfer, the requested (destination) modules, the state of those modules (including the global buffer) and the identity of the requesting module must be available to the arbitration mechanism to be able to make a decision. The signals supplying this information must therefore be valid and stable before asserting the TR_REQ* line, starting the transfer request procedure. After some amount of time, typically in the order of some hundreds of nanoseconds depending on the actual implementation and technology used0, the requested transfer is either *granted* or *rejected.*

Transfer grant

"Transfer grant" is signalled through the assertion of the GRANT* line. To tell which modules to take part in the transfer as well as the direction of transfer, the signal lines MASK and LEFT_RIGHT* must be valid and stable before asserting the GRANT* line. The GRANT* line and the MASK and LEFT_RIGHT* lines must remain valid and stable for as long as TR_REQ* is asserted. First when TR_REQ* eventually is released, MOD_REQ, MOD_BUSY* and MOD_SRC may take invalid values and GRANT* may be released together with MASK and LEFT_RIGHT*. No signals are guaranteed to be valid after the release of TR_REQ*, the output signals MASK and LEFT_RIGHT* must therefore be latched by hardware external to the arbiter mechanism on the rising (back) edge of TR_REQ* at the latest.

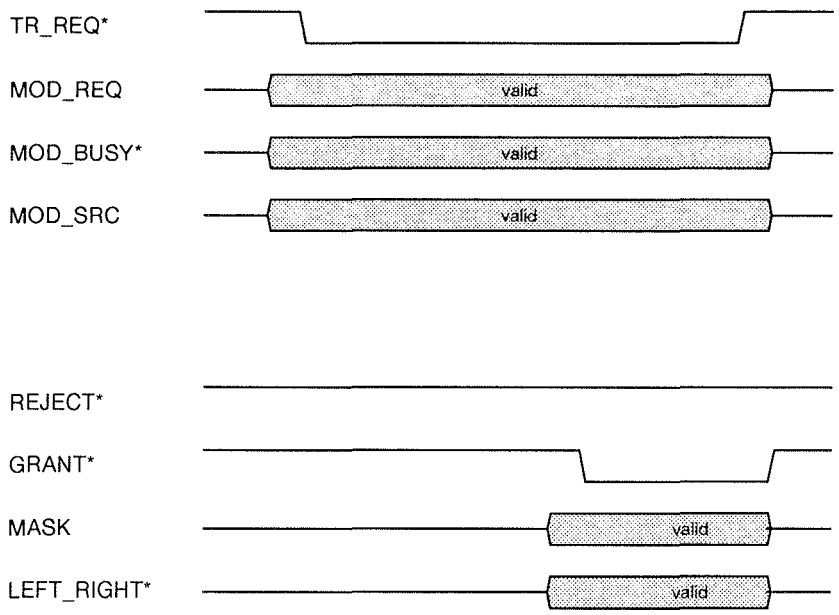


Figure 9.6. Transfer grant timing

Transfer reject

A transfer may be rejected from two reasons, path busy or module busy. *Path busy* occurs if there are no contiguous line of free Ring bus segments between the source and the last destination module in either direction. *Module busy* is the situation where at least one of the modules are busy and at the same time the global buffer is incapable of receiving more data. It is then no buffer space to store data destined for the busy module(s), and the transfer must therefore temporarily be deferred. "Transfer reject" is signalled by asserting the REJECT* signal line. The MASK and LEFT_RIGHT* signals remain invalid.

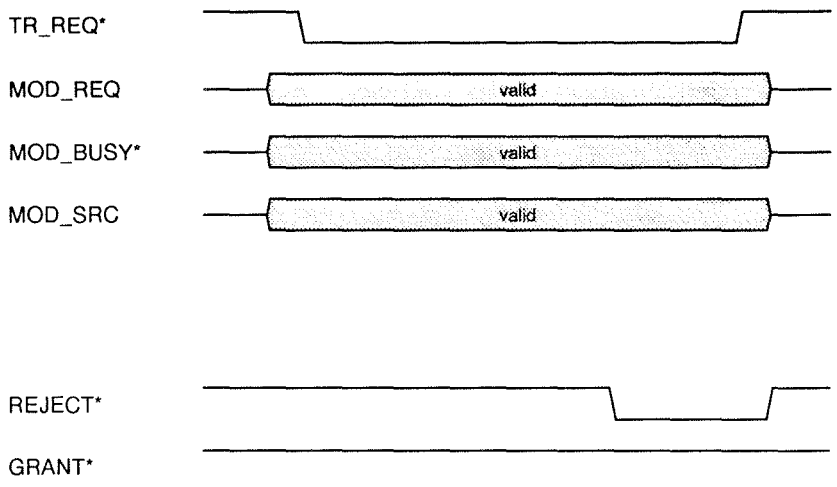


Figure 9.7. Transfer reject timing

9.3.2. Transfer release

When a transfer is terminated by the transfer's last destination module, this must be signalled to the Ring bus arbitration mechanism in order to release all Ring bus segments being allocated to that transfer. This is done by asserting the TR_REL* line. Prior to assertion, the identity of the source module for the transfer being released is signalled through the MOD_SRC lines. Acknowledge from the arbitration mechanism that the release request has been serviced is given by asserting the GRANT* line. The modules participating in the transfer being released as well as the direction of transfer are presented on the MASK and LEFT/RIGHT* signals, respectively.

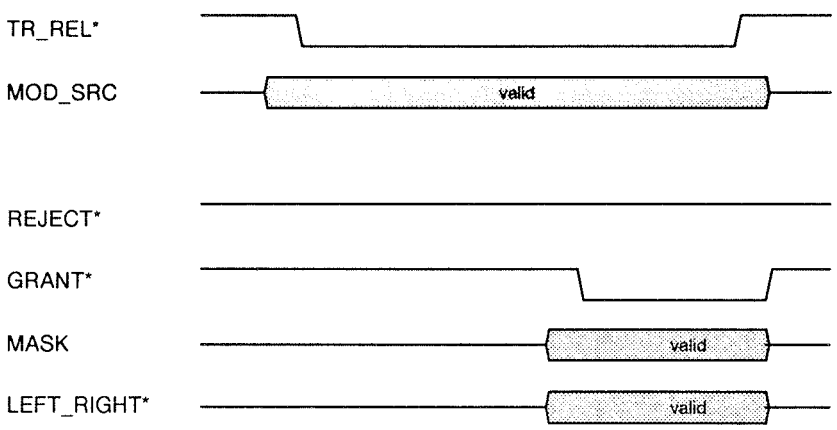


Figure 9.8. Transfer release/grant timing

If no active transfer is registered on the module identified by the MOD_SRC lines, the release request cannot be serviced and must therefore be rejected. This is done by asserting the REJECT* line. In this case, the MASK and LEFT_RIGHT* signals remain invalid.

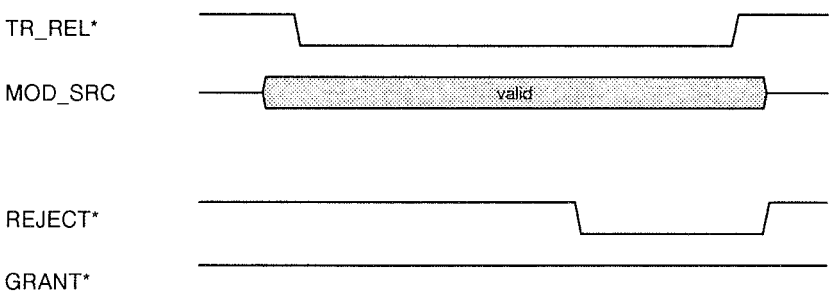


Figure 9.9. Transfer release/reject timing

The GRANT* or REJECT* line must remain asserted until TR_REL* is released.

9.4. Arbiter implementation

The task of servicing a transfer request is a two-step procedure: First, based on the location of the source module and the list of destination modules, the two alternative transfer paths are computed. Then, the "best" of the two paths is selected. This two-step procedure is also reflected in the arbiter architecture as shown in Figure 9.10.:

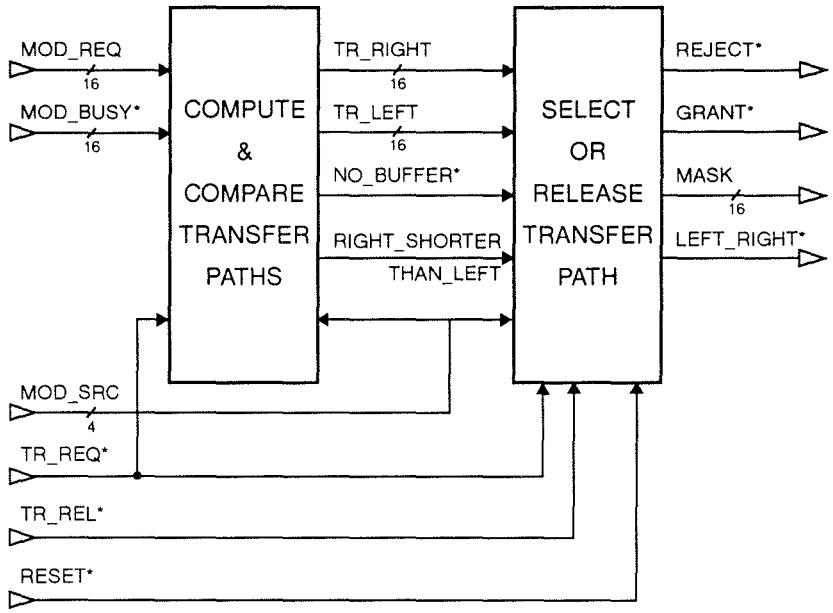


Figure 9.10. Arbitration mechanism architecture

The interface between the two sections consists of four signals, or signal groups, all output from the "Compute & Compare Transfer Paths" section. These are:

TR_RIGHT(0-15)

TRansfer RIGHT. This 16 bit mask shows which modules to participate in the requested transfer, from the source to the last destination module (both inclusive), if the transfer is performed as a right direction transfer. Values:

0 : The module is not to participate in the transfer.

1 : The module is to participate in the transfer.

TR_LEFT(0-15)

TRansfer LEFT. Corresponding to TR_RIGHT, except that the transfer is now to be performed as a left direction transfer. Values:

0 : The module is not to participate in the transfer.

1 : The module is to participate in the transfer.

NO_BUFFER*

NO BUFFER available (low). In case of one or more requested destination modules being busy and the global buffer unable to receive more data, signal NO_BUFFER* is asserted. The transfer request being serviced must then be rejected.

RIGHT_SHORTER_THAN LEFT

This signal is the result of the comparison between the two alternative transfer paths, left and right. Value:

0 : The left path is the shortest.

1 : The right path is the shortest.

9.4.1. Compute & Compare Transfer Paths

The function of the "Compute & Compare Transfer Paths" (C&C) section is to compute and then present the two alternative transfer paths to the subsequent "Select Transfer Path" section, which will select the actual path to use. The C&C section is in turn assembled from 6 blocks as shown in Figure 9.11.

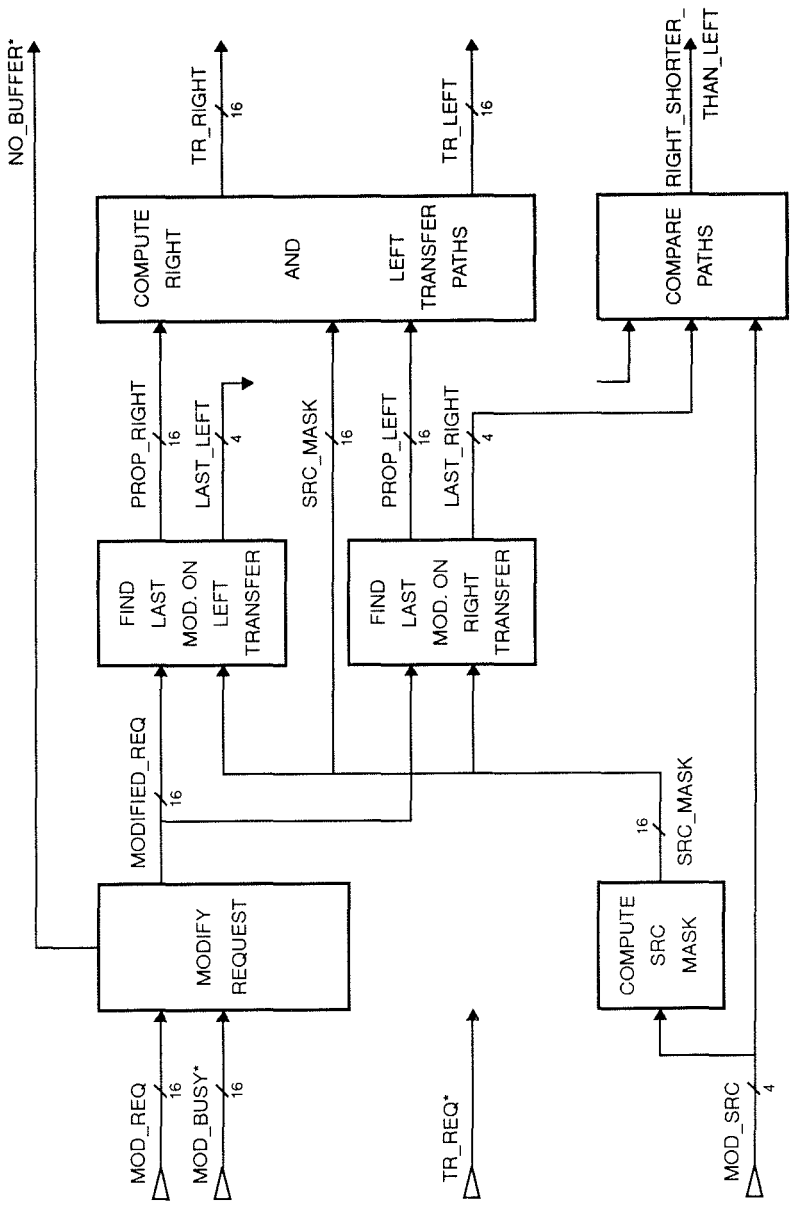


Figure 9.11. Compute & Compare Transfer Paths

Signal descriptions

MODIFIED_REQ(0-15)

A 16 bit mask containing only non-busy, requested destination modules. If one or more destination modules were busy (and thereby removed from the original destination list), this (these) modules are replaced by the global buffer in the modified destination list (bit 15). Values:

0 : The module is not to be enabled as a destination module.

1 : The module is to be enabled as a destination module.

SRC_MASK(0-15)

A 16 bit mask obtained by decoding the 4 bit MOD_SRC value. SRC_MASK has only one bit set, which position (0 to 15) within the 16 bit mask is equal to the source module address. Values:

0 : The corresponding module is not the source module.

1 : The corresponding module is the source module.

PROP_RIGHT(0-15)

A 16 bit mask with all bits set equal to 1 from the source module up to the first destination module (both inclusive), moving in the right direction. All other bits are set equal to 0.

LAST_LEFT(0-3)

A 4 bit value equal to the address of the destination module nearest to the right of the source module. In other words, the position of the first destination module bit in the PROP_RIGHT mask.

PROP_LEFT(0-15)

A 16 bit mask with all bits set equal to 1 from the source module up to the first destination module (both inclusive), moving in the left direction. All other bits are set equal to 0.

LAST_RIGHT(0-3)

A 4 bit value equal to the address of the destination module nearest to the left of the source module. That is, the position of the first destination module bit in the PROP_LEFT mask.

Functional description

The list of requested destination modules is presented to the "Modify Request" block as a 16 bit mask (MOD_REQ). According to the 16 bit status mask MOD_BUSY*, any requested destination modules which are busy is removed from the destination list and replaced by the global buffer. The result of this is a modified, requested destination list (MODIFIED_REQ). The NO_BUFFER* signal is also generated by the "Modify Request" block.

Together with the source module address, represented as a 16 bit mask (SRC_MASK value, computed by the "Compute Src Mask" block), the modified destination list is presented in parallel to two blocks computing the last destination module in case of a left and right transfer, respectively. This is done by the "Find Last Mod. on Left Transfer" and "Find Last Mod. on Right Transfer" blocks.

The outcome of these two blocks are in turn input to the "Compute Right and Left Transfer Paths" block, along with SRC_MASK, identifying the location of the source module. The masks of the two alternative transfer paths, TR_RIGHT and TR_LEFT, can now be computed.

By doing a modulo 16 subtraction between the source module (MOD_SRC) and the last destination module addresses (LAST_LEFT and LAST_RIGHT for left and right direction transfers, respectively), the shorter of the two alternative transfer paths can be determined. This is done by block "Compare Paths", producing the signal RIGHT_SHORTER_THAN_LEFT.

The various blocks contained in the "Compute & Compare Transfer Paths" section will now be described in greater detail.

9.4.1.a. Modify Request

The function of this block is to remove all busy modules from the original list of destination modules requested by the source module. "Busy" in this context means that a module currently is unable to receive input data. This may be due to lack of local buffer space or heavy processing load. The busy modules are removed from the destination list by executing the operation

```
MODIFIED_REQ(i)
    = MOD_REQ(i) & MOD_BUSY*(i);     i = 0-14
```

As far as module 15 is concerned (the controller, containing the global buffer), it is added to the destination list if any module is removed from the original list. That is,

```
MODIFIED_REQ(15)
    = MOD_REQ(15) # "any module removed"
    = MOD_REQ(15) # ((MOD_REQ(0-14) & !MOD_BUSY(0-14))
```

The MODIFIED_REQ(15) bit now says if the global buffer is a destination for the transfer request being serviced. If it is, it is because some other destination module is busy or because the module is explicitly requested as a destination module, which it can be as any other module. If the global buffer is busy (expressed by the state of signal MOD_BUSY*(15)), being a destination module, the transfer request must be rejected. This is told through signal NO_BUFFER*.

```
NO_BUFFER*
    = !(!MOD_BUSY*(15) & MODIFIED_REQ(15))
```

The implementation of block "Modify Request" is shown in Figure 9.12.

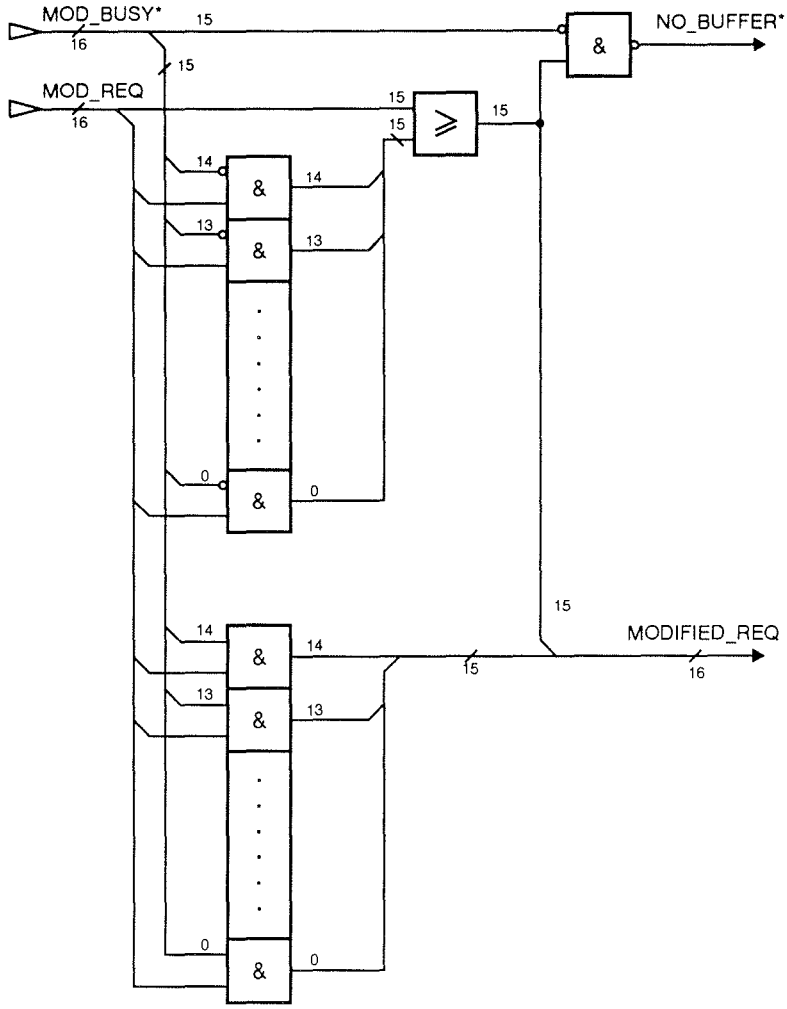


Figure 9.12. Modify Request

For better to be able to explain the function of the arbitration mechanism, we will now introduce a concrete transfer request example which will be followed step by step through the entire mechanism:

```
Source module                          :  8
Destination modules                    :  4, 6, 9 and 11
Last dest. module (left  transfer)     :  9
Last dest. module (right transfer)     :  6
Modules busy                           :  2, 3, 7 and 9
```

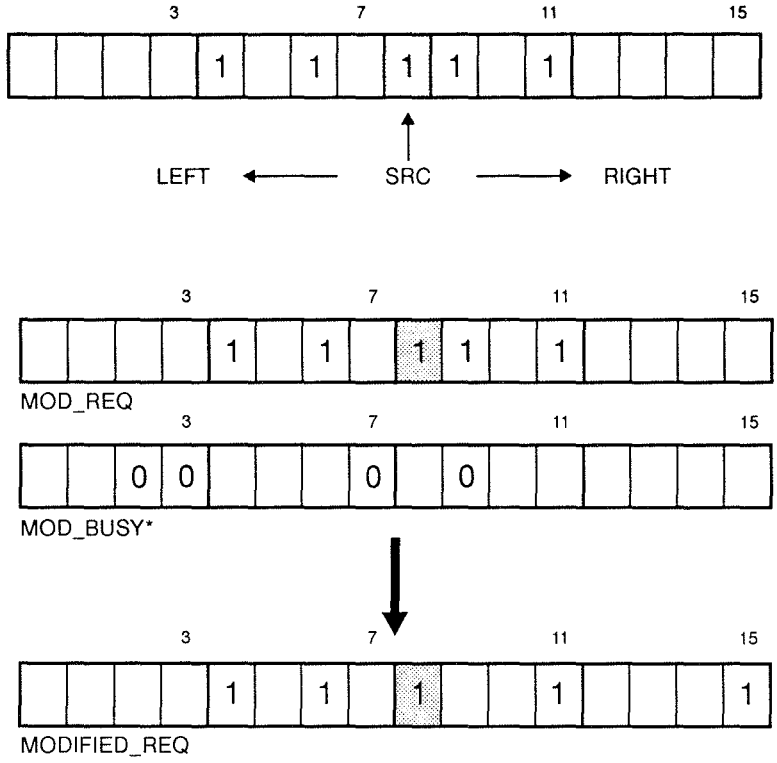


Figure 9.13. Transfer request example

MOD_REQ and MOD_BUSY* values will then be as shown in Figure 9.13. The busy module 9 will be removed from the destination list and replaced by the global buffer, module 15 (MODIFIED_REQ). To avoid overcrowding the figures, only asserted bits (1's for active high signals and 0's for active low signals) will be shown. Further, bit position 8 will be shadowed to help keeping track of the source module.

9.4.1.b. Compute Src Mask

This block is an ordinary 4 to 16 decoder, generating the 16 bit SRC_MASK from the 4 bit MOD_SRC value.

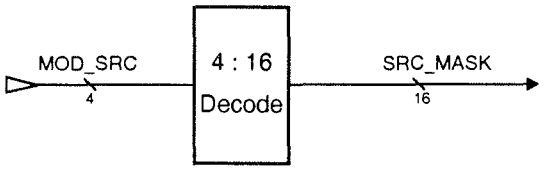


Figure 9.14. Compute Src Mask

With source module address equal to 8, SRC_MASK value will be as follows:

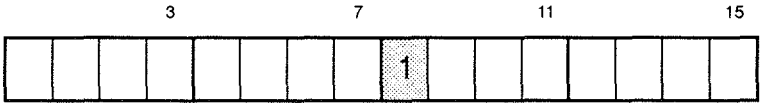


Figure 9.15. SRC_MASK value

9.4.1.c. Find Last Module on Left Transfer

The basic idea behind this and the corresponding "right" block is to determine the *last* (destination) module for a given transfer by finding the *first* module going in the opposite direction. This is done by propagating the single "1" in SRC_MASK in the direction opposite to the direction of transfer until the first "1" in MODIFIED_REQ is encountered.

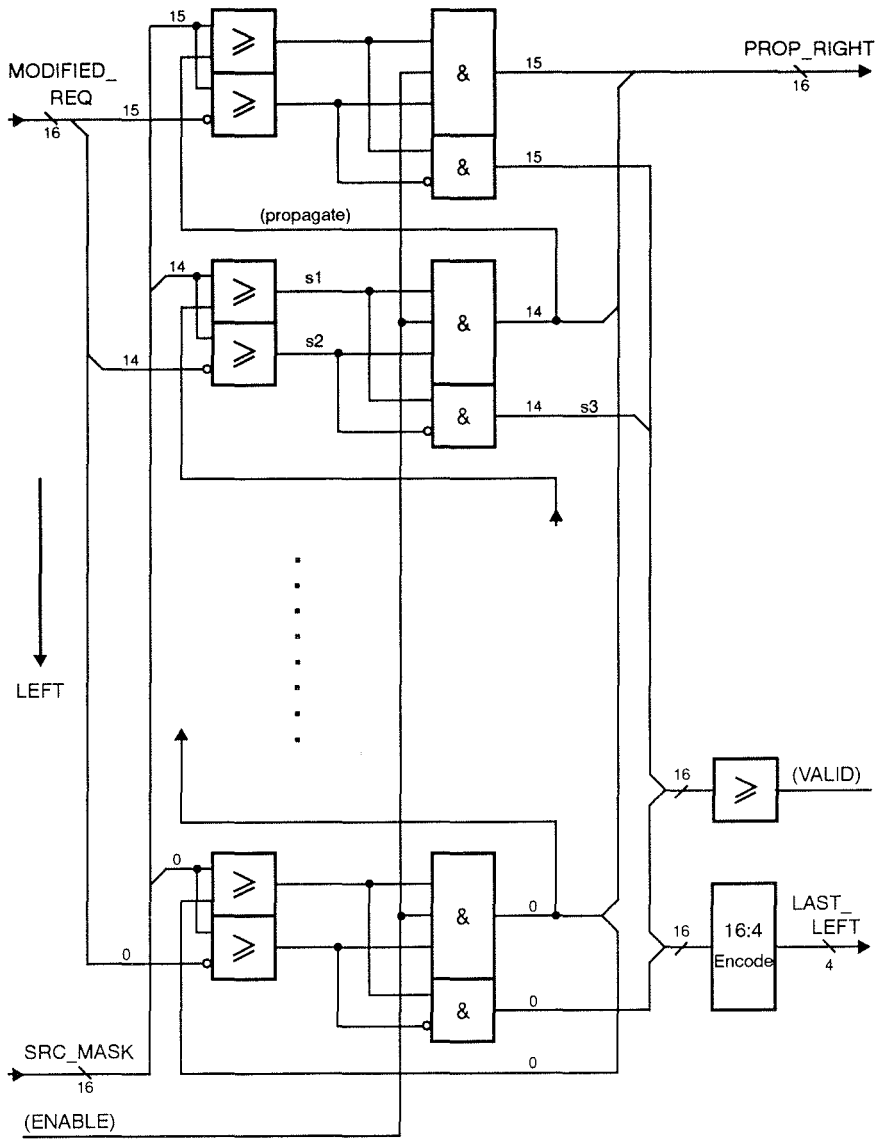


Figure 9.16. Find Last Module on Left Transfer

Basically, a propagation line daisy-chained from bit to bit needs a *start* condition and a *stop* condition. The start condition must be unique while there can be several stop conditions, the one encountered first will then terminate the propagation line. The left transfer propagation process from a source module "n" (n = 0-15) to the last destination module "m" is shown step by step in Figure 9.17.

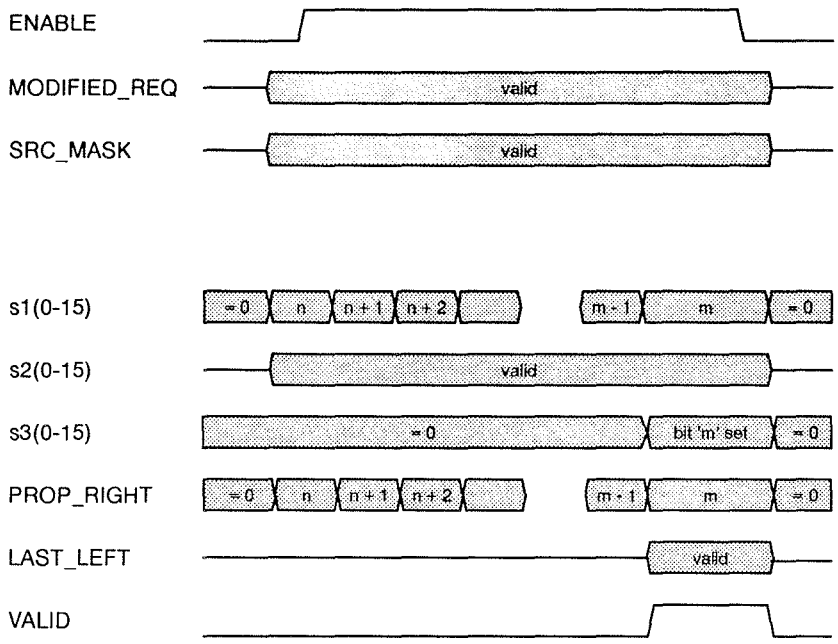


Figure 9.17. "n" to "m" propagation timing

Back to our example: The modified destination list (module 9 removed, 15 added) is presented to the left transfer propagation line, shown in Figure 9.18. The propagation line is initially disabled by keeping signal ENABLE low. Signals s1, s2, s3 and PROP_RIGHT will then have the following initial state:

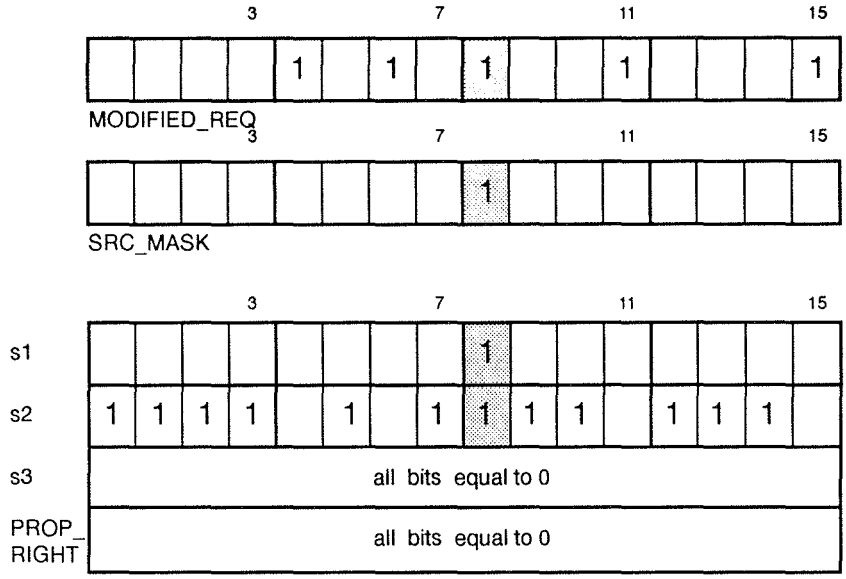


Figure 9.18. Left transfer propagation line, initial state

After setting up the MODIFIED_REQ and SRC_MASK values, the propagation process is started by asserting the ENABLE signal. s1(i) and s2(i) will always both be equal to 1 for "i" equal to the source module (8) and for this bit position only. Asserting ENABLE will then cause the corresponding PROP_RIGHT(8) to be equal to 1 (step 1).

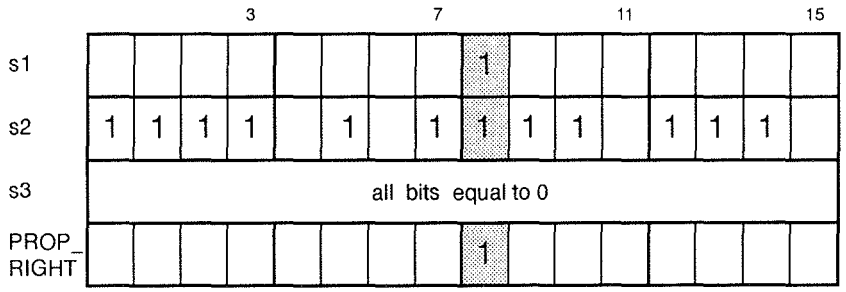


Figure 9.19. Left transfer propagation line, step 1

PROP_RIGHT(8) is connected to the input of the upper OR-gate of bit position 9, s1(9) will therefore go to 1 (step 2).

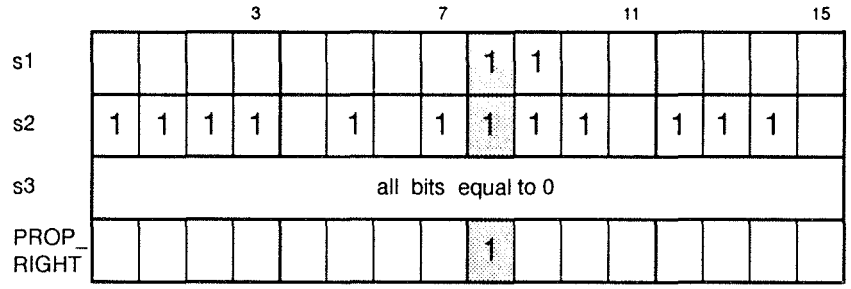


Figure 9.20. Left transfer propagation line, step 2

The propagation process will not stop until the first s2(j)=0 is encountered. s2(9) is equal to 1, signal s1(9) going to 1 therefore causes PROP_RIGHT(9) to go to 1, too (step3).

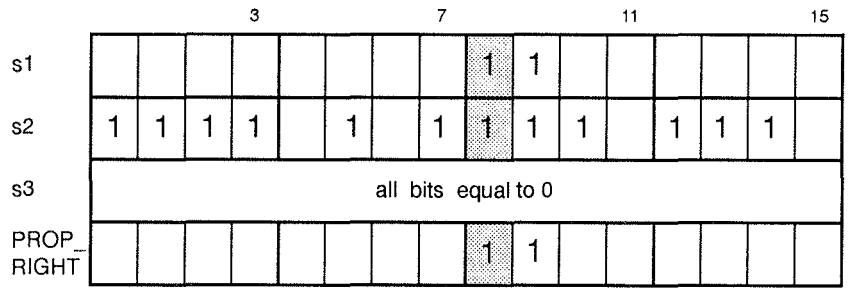


Figure 9.21. Left transfer propagation line, step 3

This process will go on through steps 4, 5 and 6 until s1(11) goes to 1 (step 6).

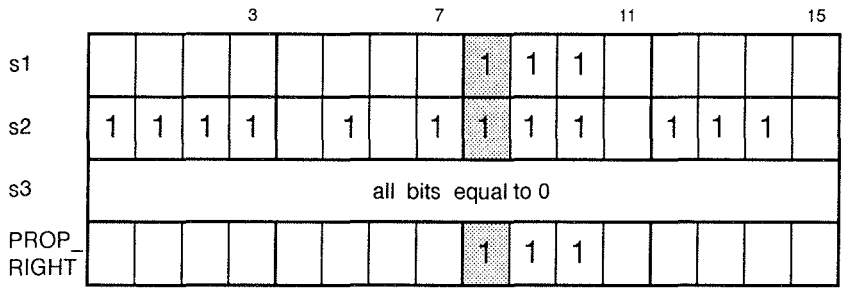


Figure 9.22. Left transfer propagation line, step 4

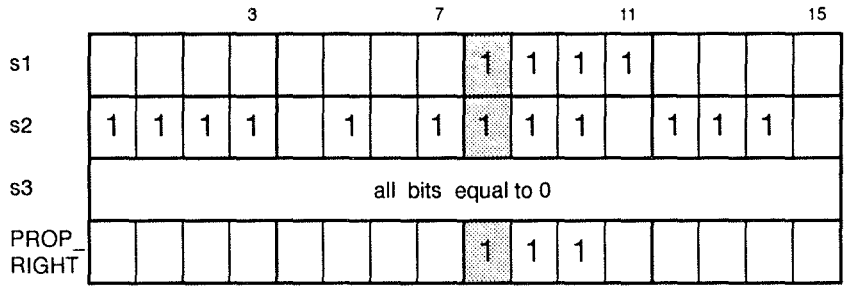


Figure 9.23. Left transfer propagation line, step 5

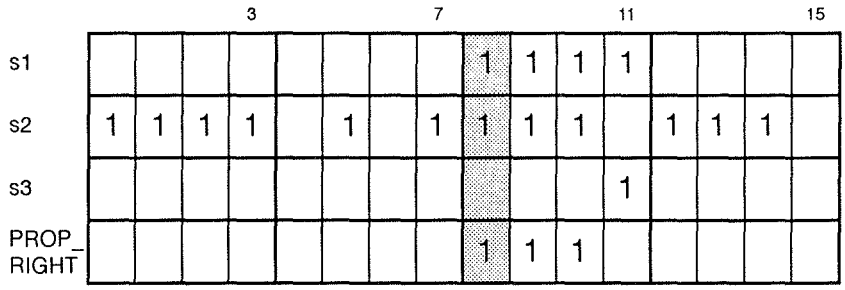


Figure 9.24. Left transfer propagation line, step 6

For the first time since the propagation process started, the corresponding s2 bit, s2(11), is equal to 0 and the 1 on s1(11) is therefore not propagated through the triple-input AND-gate to PROP_RIGHT(11), and the propagation is thereby terminated. The stop condition, a high s1 and a low s2, causes the corresponding s3, in this case s3(11), to be set, as the only s3 bit. The location of the *last* destination module in case of a *left* transfer, has now been determined by finding the *first* destination module searching in the *right* direction, starting from the source module. By encoding s3, the 4 bit value LAST_LEFT will be equal to the location of that last destination module.

Upon termination, signal PROP_RIGHT will be a "recording" of the travelled propagation line, having all bits set from the source module (inclusive) up to the last destination module (not inclusive).

Because all 16 possible values LAST_LEFT can take are valid, and termination neither can be detected by from the state of PROP_RIGHT, a dedicated "output valid" signal is needed to notify the next block in the arbiter mechanism that it may start processing its input signals (that

is, LAST_LEFT and PROP_RIGHT). Unlike the other blocks, the "Find Last Module on Left (Right) Transfer" blocks will not have a fixed delay from input to output, but the delay will be data dependent. If a fixed delay nevertheless is used to provide timing between input and output, it must be set to the worst case value which will be 30 propagation steps (15 bits times 2 steps if there is only one destination module, located to the immediate left of the source module). However, a valid output can easily be signalled as soon as it occurs by detecting s3 going from zero to non-zero. This is done through a 16 input OR-gate, producing signal VALID.

In our example, output from the "Find Last Module on Left Transfer" block is the 16 bit PROP_RIGHT as shown in Figure 9.25. and LAST_LEFT equal to 11.

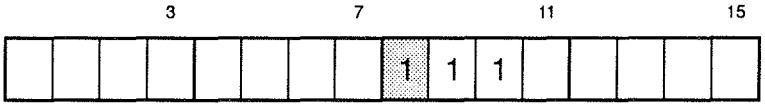


Figure 9.25. PROP_RIGHT output

9.4.1.d. Find Last Module on Right Transfer

This block's function and construction is completely analogue to the corresponding left block already described in detail.

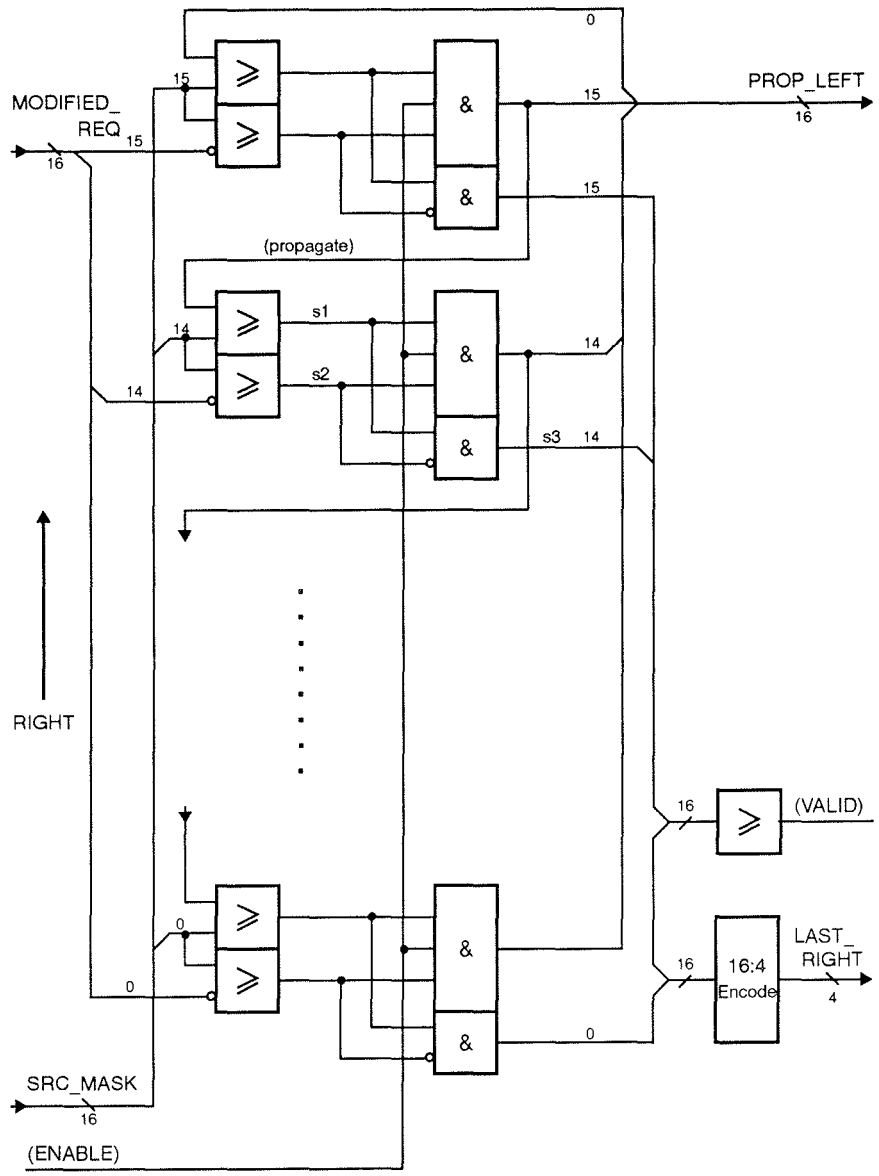


Figure 9.26. Find Last Module on Right Transfer

By applying input data according to our example, output from the block will be equal to PROP_LEFT as shown in Figure 9.27. and LAST_RIGHT equal to 6.

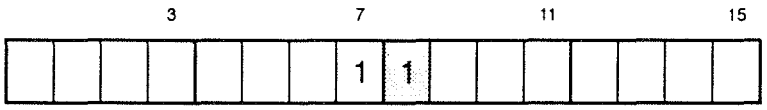


Figure 9.27. PROP_LEFT output

9.4.1.e. Compute Left and Right Transfer Paths

The purpose of this block is for each of the two transfer paths to set all bits corresponding to modules to be involved in the transfer (the source and last destination module included) equal to 1. This is obtained by executing the function:

```
TR_RIGHT(i)
    = !(PROP_RIGHT(i) & !SRC_MASK(i));      i = 0-15

TR_LEFT(i)
    = !(PROP_LEFT(i)  & !SRC_MASK(i));      i = 0-15
```

Implementation and output applying example data input are as shown in Figure 9.28. and Figure 9.29.:

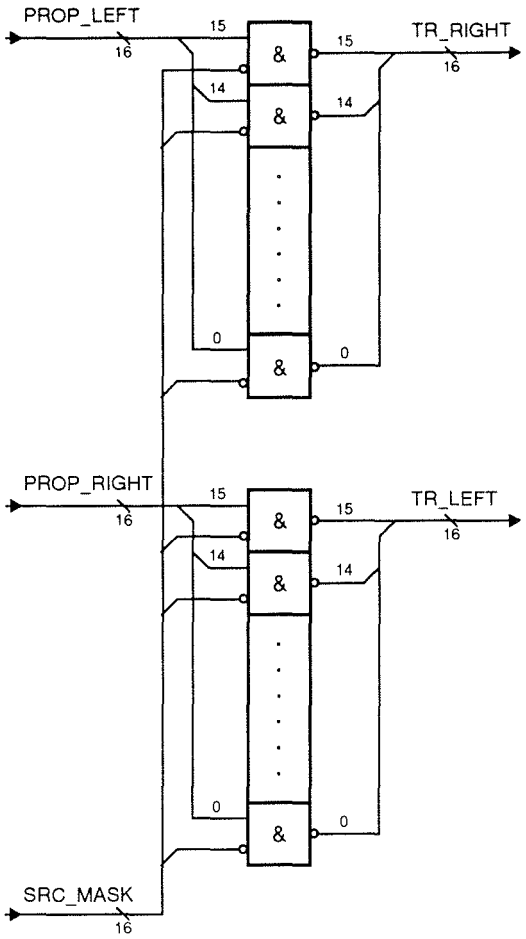


Figure 9.28. Compute Left and Right Transfer Paths

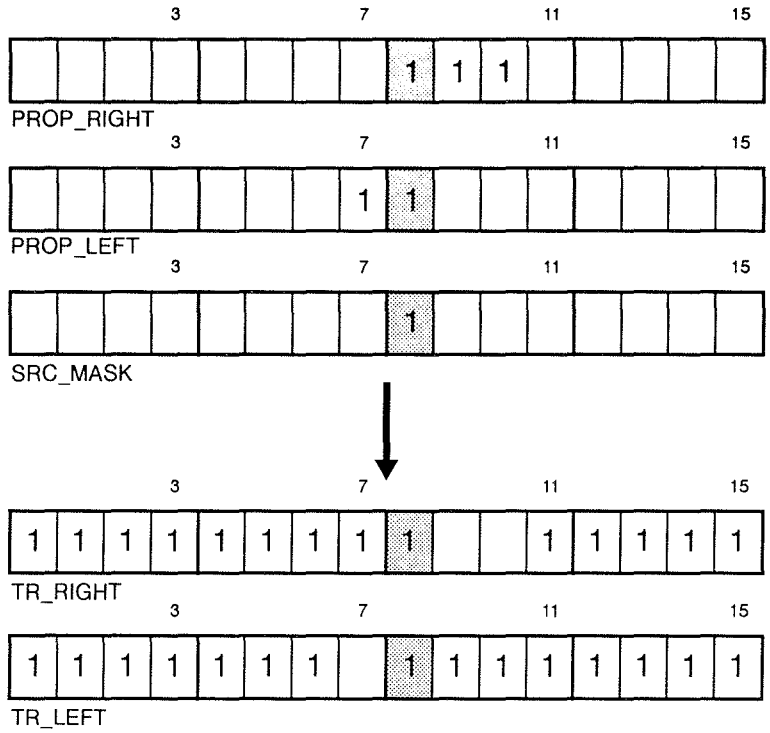


Figure 9.29. "Compute Left and Right Transfer Paths" output

9.4.1.f. Compare Paths

If the necessary Ring bus segments for both paths are free so either one can be used, the shorter of the two will be selected. Because no a priori information is available as far as future requests are concerned, the most "intelligent" selection strategy possible is simply to select the shortest path. This is done by calculating the distance between the source and the last destination modules for the left and right transfer alternatives, and then comparing the results.

```
ltd = (LAST_LEFT  - MOD_SRC)modulo 16
rtd = (LAST_RIGHT - MOD_SRC)modulo 16

if (rtd < ltd)
    RIGHT_SHORTER_THAN_LEFT = 1
else
    RIGHT_SHORTER_THAN_LEFT = 0
```

Implementation is done with two subtracters feeding a subsequent comparator as shown in Figure 9.30.

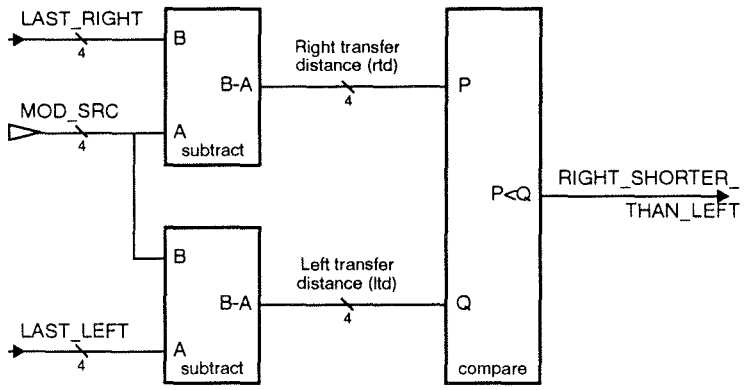


Figure 9.30. Compare Paths

In our example, LAST_LEFT is equal to 11 and LAST_RIGHT equal to 6. With MOD_SRC equal to 8, this gives a rtd of 14 and ltd of 13. The left transfer path is thereby the shorter and RIGHT_SHORTER_THAN_LEFT will be equal to 0.

9.4.2. Select Transfer Path

After computing and comparing the two alternative paths, one of the paths is selected. Selection is done on the basis of the length of the paths and which Ring bus segments are available. The "Select Transfer Path" section consists of four blocks and is assembled as shown in Figure 9.31.

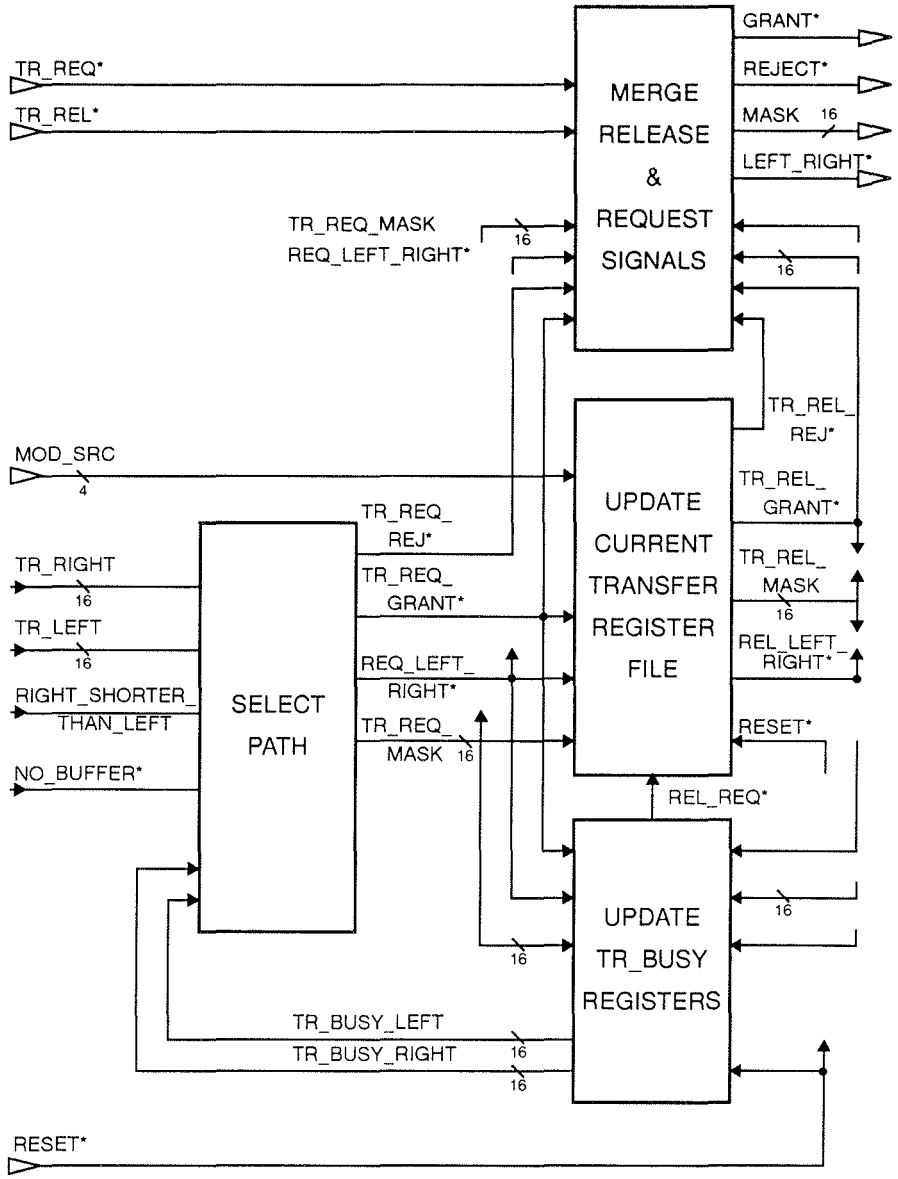


Figure 9.31. Select Transfer Path

Signal descriptions

TR_REQ_GRANT*

TRansfer REQuest GRANT (low). Asserted when the requested transfer is granted.

TR_REQ_REJ*

TRansfer REQuest REJect (low). Asserted when the requested transfer is rejected.

TR_REQ_MASK(0-15)

TRansfer REQuest MASK. This 16 bit mask shows which modules to participate in the requested transfer, from the source to the last destination module (both inclusive). Values:

0 : The module is not to participate in the transfer.

1 : The module is to participate in the transfer.

REQ_LEFT_RIGHT*

REQuested transfer LEFT or RIGHT. Tells whether the requested transfer being granted is to be performed as a left or right transfer. Value:

0 : Right transfer

1 : Left transfer

TR_REL_GRANT*

TRansfer RELease GRANT (low). Asserted when the transfer release is serviced.

TR_REL_REJ*

TRansfer RELease REJect (low). Asserted when the transfer release can not be serviced.

TR_REL_MASK(0-15)

TRansfer RELease MASK. The mask shows which modules were used by the transfer and now are released. Values:

0 : The module was not used by the transfer.

1 : The module was used by the transfer.

REL_LEFT_RIGHT*

RELeased transfer LEFT or RIGHT. Tells whether the transfer being released was a left or right transfer. Value:

0 : Right transfer

1 : Left transfer

REL_REQ*

RELease or REQuest. Tells whether the operation now being executed is a transfer release or a transfer request. Value:

0 : Transfer request

1 : Transfer release

TR_BUSY_LEFT(0-15)

TRansfer BUSY LEFT. This 16 bit mask shows which Ring bus left direction transfer segments (modules) are already in use by one or more currently active transfers. Values:

0 : Ring bus segment (module) is available for transfer.

1 : Ring bus segment (module) is not available for transfer.

TR_BUSY_RIGHT(0-15)

TRansfer BUSY RIGHT. Status for currently active right transfers, analogue to TR_BUSY_LEFT. Values:

0 : Ring bus segment (module) is available for transfer.

1 : Ring bus segment (module) is not available for transfer.

Functional description

The purpose of the "Select Transfer Path" block is twofold: First, in case of a transfer request, to select one out of the two alternative transfer paths (TR_RIGHT and TR_LEFT) computed by the "Compute & Compare Transfer Paths" (C&C) block. The selection is based on the comparison done by the C&C block (RIGHT_SHORTER_THAN_LEFT) as well as the idle/ busy status of the Ring bus segments (TR_BUSY_RIGHT and TR_BUSY_LEFT). The Ring bus status is stored in a register file in the "Update TR_BUSY Registers" block.

Secondly, the transfer release mechanism is also contained in this block. When a particular transfer is to be released (there may be several transfers active simultaneously), the transfer must be identified in some way. This is done by applying the address of its source module. The appropriate TR_BUSY register (left or right) must then be updated clearing (de-allocating) all segments used by the transfer. Because the identity of the segments to be cleared is not explicitly presented by the release procedure as an input parameter, they (that is, the transfer mask) must be stored within the arbitration mechanism for as long as the transfer remains active. This is done in the "Current Transfer Register File", contained in the "Update Current Transfer Register File" block. Together with the transfer mask, a single bit telling whether a transfer is a left or right direction transfer is also stored.

Both transfer requests and -releases are acknowledged by the arbitration mechanism by an asserted GRANT* or REJECT* signal. In case of a transfer release, a REJECT* is issued if the transfer, identified by its source module (MOD_SRC) is unknown to the arbitration mechanism. For the sake of symmetry and possible external use, the stored transfer mask (MASK) and left/ right direction bit (LEFT_RIGHT*) are output on transfer release as well as transfer request operations. The merging of release and request acknowledges and other output signals are done by the "Merge Release & Request Signals" block.

We will now take a look at the blocks in greater detail:

9.4.2.a. Select Path

Because nothing is known about future transfer requests, the most intelligent criterion to apply when selecting a transfer path is simply to select the shortest one. The "Select Path" block should therefore implement the following algorithm:

```
Select_Path()
{
    if (buffers available)
    {
        if (both paths are available)
            select the shortest
        else if (one path is available)
            select the one available
        else
            reject transfer request
    }
    else
        reject transfer request
}
```

Figure 9.32. Select_Path algorithm

If necessary buffers are unavailable, this is told by the state of signal NO_BUFFER*. A path (left or right) is available if *all* necessary Ring bus segments, as specified by the transfer masks (TR_LEFT and TR_RIGHT) are idle (TR_BUSY_LEFT, TR_BUSY_RIGHT). An implementation of the "Select Path" block can therefore be as shown in Figure 9.33.

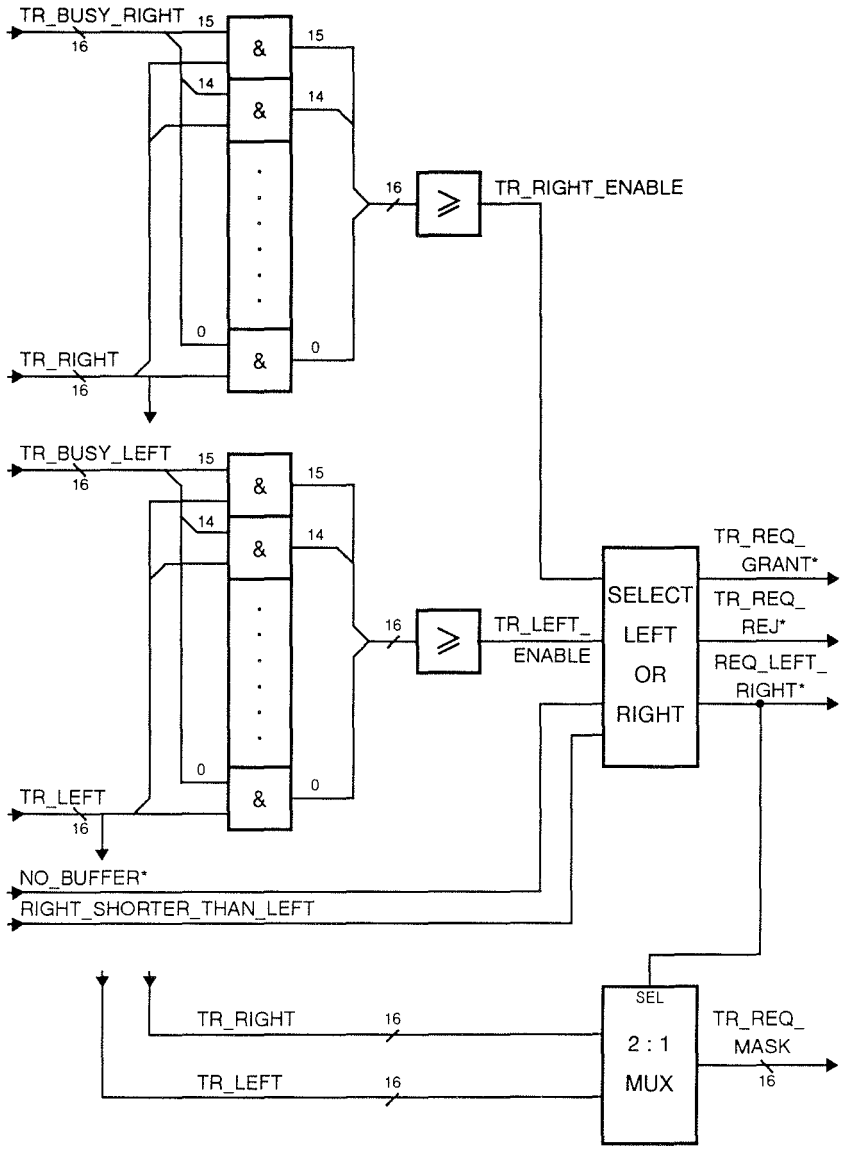


Figure 9.33. Select Path

The "Select Left or Right" control block contains the following logic:

```
TR_REQ_GRANT*
    = !(( TR_RIGHT_ENABLE #  TR_LEFT_ENABLE)
    &      NO_BUFFER*)

TR_REQ_REJ*
    = !((!TR_RIGHT_ENABLE & !TR_LEFT_ENABLE)
    #     !NO_BUFFER*)

REQ_LEFT_RIGHT*
    = !(  TR_RIGHT_ENABLE &  RIGHT_SHORTER_THAN_LEFT)
```

Figure 9.34. "Select Left or Right" control block

The REQ_LEFT_RIGHT* signal multiplexes the mask of the selected path onto the TR_REQ_MASK signal lines.

9.4.2.b. Update TR_BUSY Registers

The "Update TR_BUSY Registers" block consists itself of three blocks, one for each of the two TRBUSY registers and a control block. This is shown in Figure 9.35.

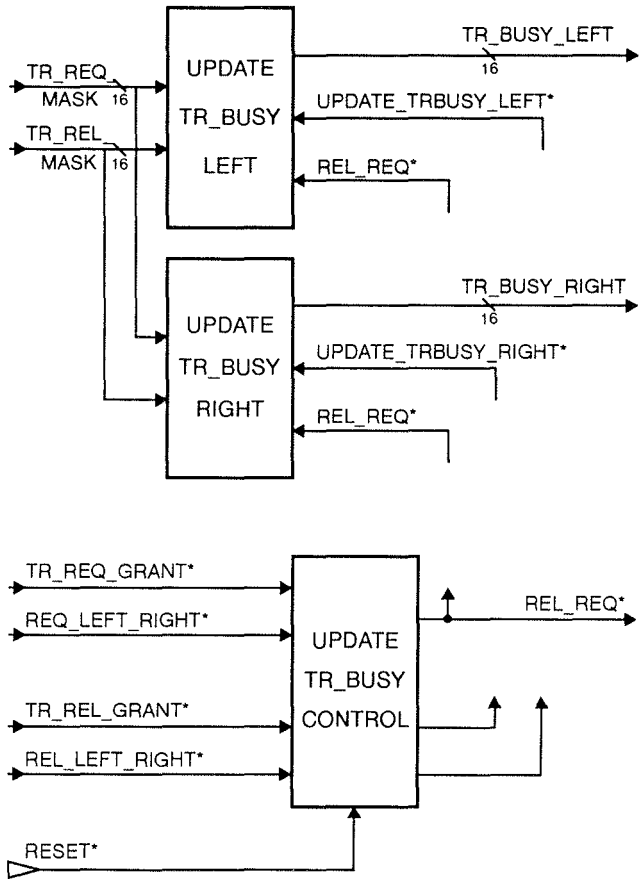


Figure 9.35. Update TR_BUSY Registers

Update TR_BUSY Control

This is a simple decoding of the input signals and implements the following logic:

```
REL_REQ*
    =     !TR_REQ_GRANT*

UPDATE_TRBUSY_LEFT*
    = !((!TR_REQ_GRANT* &  REQ_LEFT_RIGHT*)
    #    (!TR_REL_GRANT* &  REQ_LEFT_RIGHT*)
    #     !RESET)
```

```
UPDATE_TRBUSY_RIGHT*
    = !((!TR_REQ_GRANT* & !REQ_LEFT_RIGHT*)
    #    (!TR_REL_GRANT* & !REQ_LEFT_RIGHT*)
    #     !RESET)
```

Figure 9.36. "Update TR_BUSY Control" logic

Update TR_BUSY Right

In addition to the TR_BUSY_RIGHT register itself, this block contains the logic necessary to individually set (on request) and clear (release) each bit in the register. This is done by a read-modify-write mechanism implementing the logic

```
new_value(i)
    = old_value(i) & !TR_REL_MASK(i);    (i = 0-15, clear)

new_value(i)
    = old_value(i) #  TR_REQ_MASK(i);    (i = 0-15, set)
```

The TR_REQ_MASK is provided by the "Select Path" block, while the corresponding release mask TR_REL_MASK is supplied by the "Current Transfer Register File", indexed by the address of the source module.

Upon initialization, the TR_BUSY_RIGHT register must be reset (all bits cleared) by asserting the input signal RESET*. This forces the output of the 2:1 multiplexer to zero and then pulses the UPDATE_TRBUSY_RIGHT* signal.

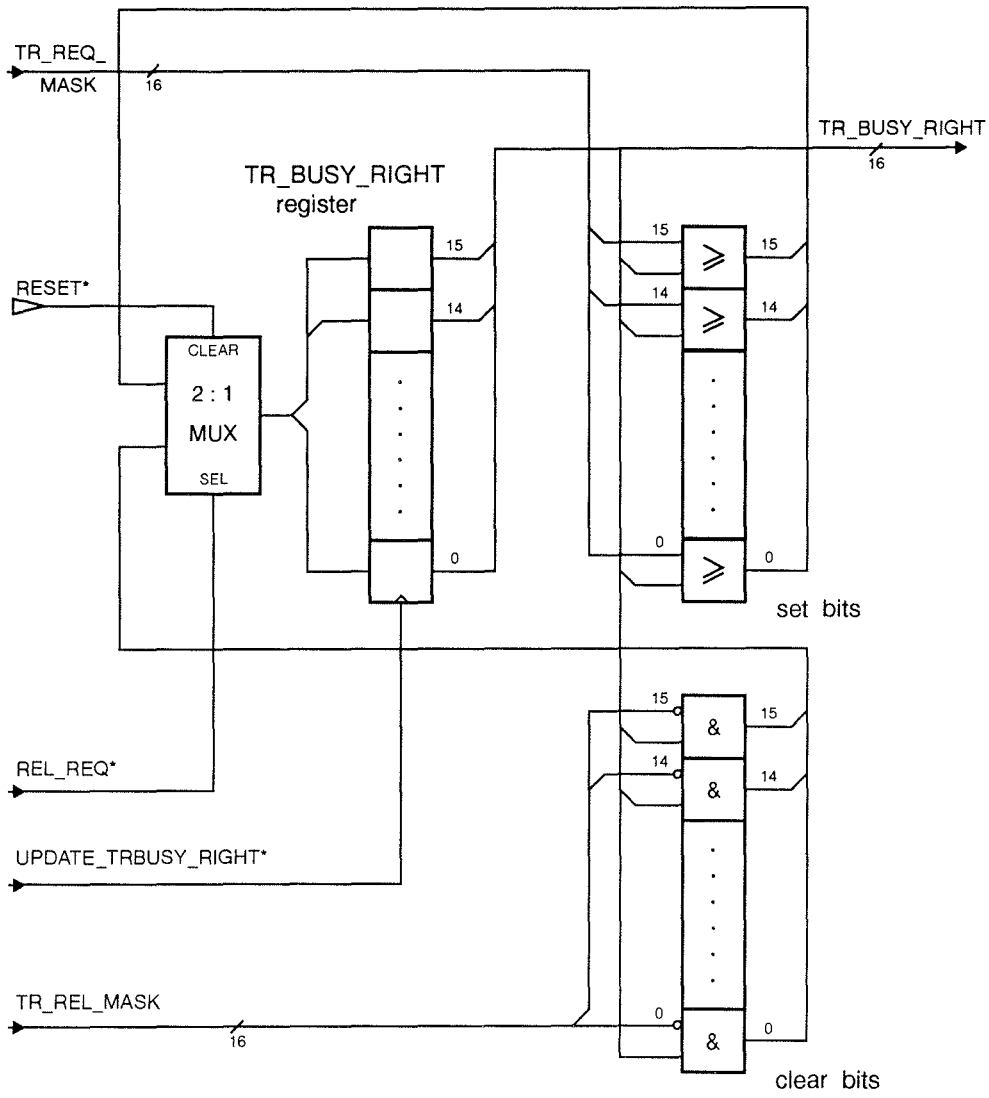


Figure 9.37. Update TR_BUSY Right

Update TR_BUSY Left

This block is identical to the corresponding TR_BUSY Right block.

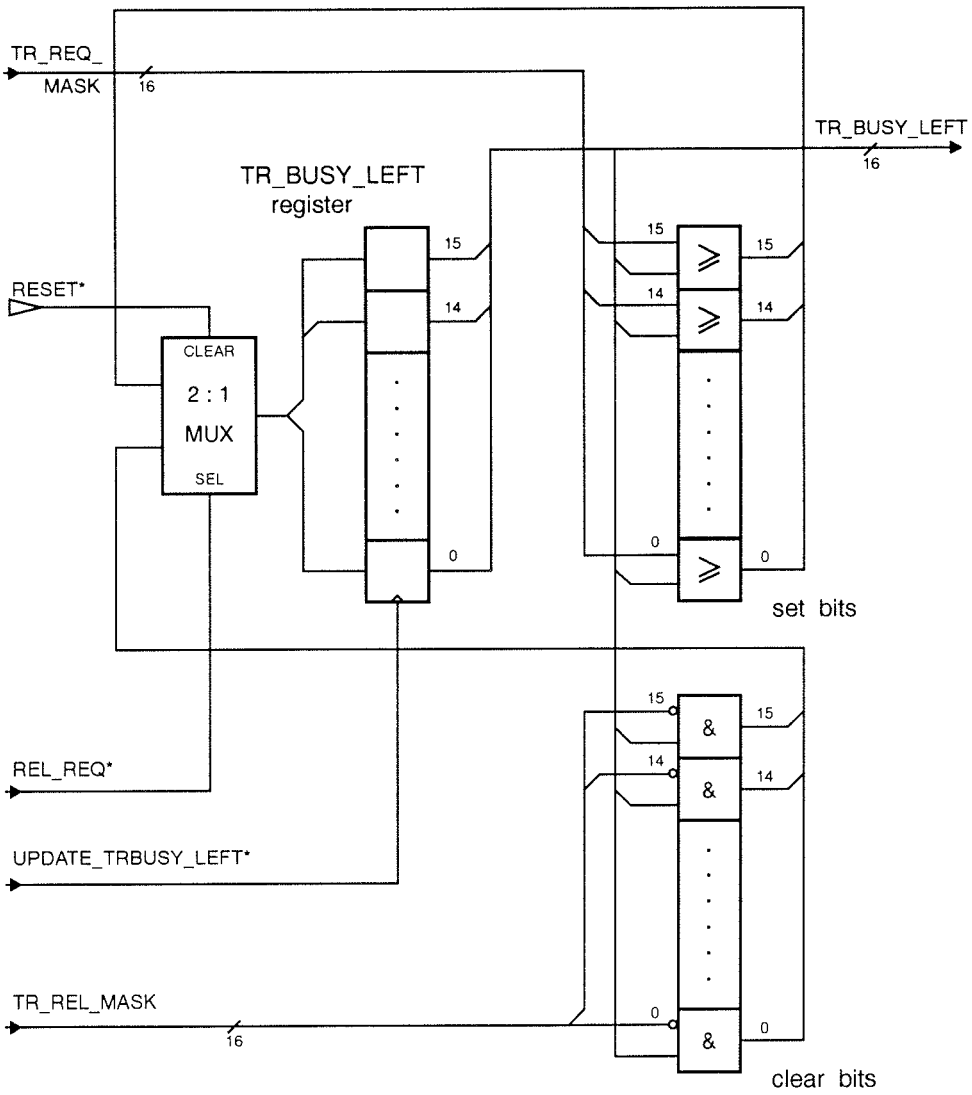


Figure 9.38. Update TR_BUSY Left

9.4.2.c. Update Current Transfer Register File

The "Current Transfer Register File" contains 16 entries with 17 bits each (a 16 bit mask and a single left/ right direction bit). Each time a transfer request is granted, the transfer mask (TR_REQ_MASK) and the bit telling whether it is a left or right transfer (REQ_LEFT_RIGHT*) are stored into the register entry with address equal to the source module address (MOD_SRC).

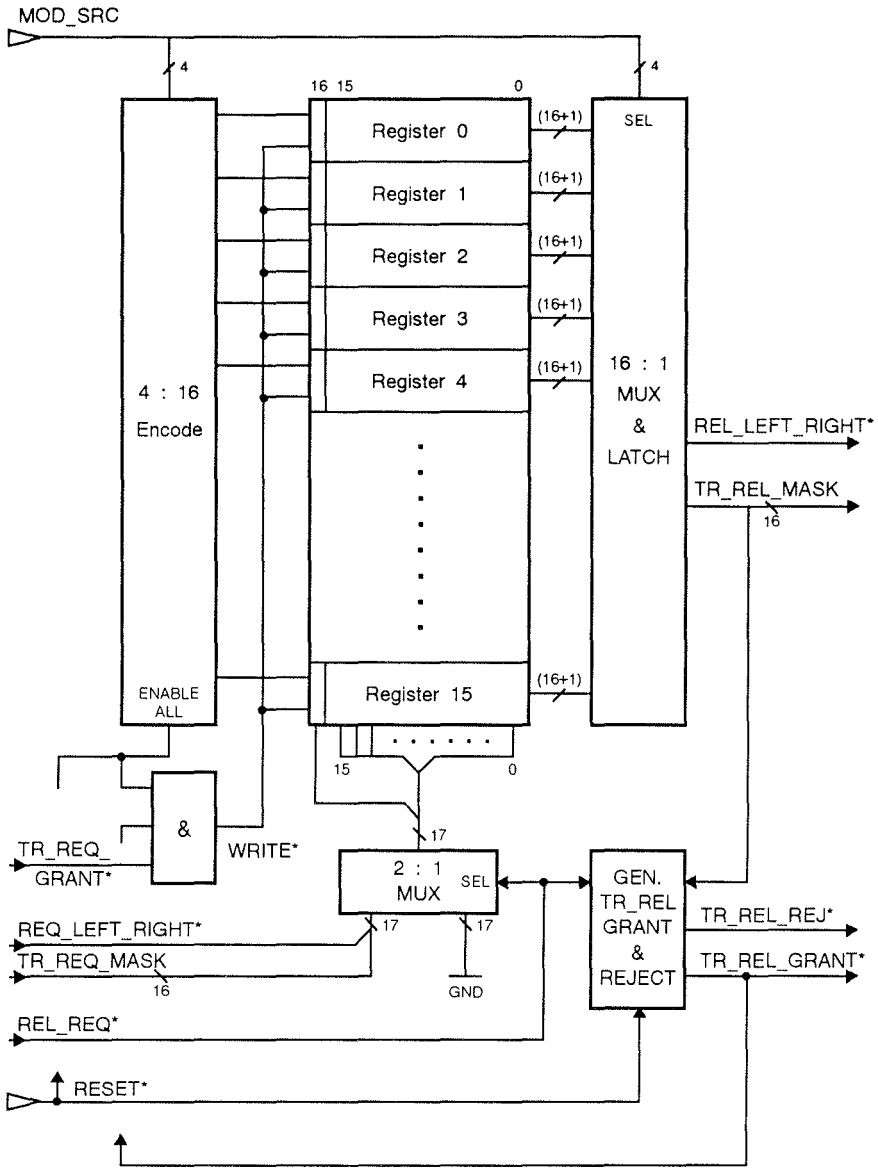


Figure 9.39. Update Current Transfer Register File

When a transfer is released, the 17 bit register entry addressed by the MOD_SRC input parameter is output and latched onto the TR_REL_MASK and REL_LEFT_RIGHT* signal lines, connected to the "Update TR_BUSY Registers" block. After latching, the entry is cleared by resetting all 17 bits to zero. The transfer release operation is either granted or rejected. This is done on the basis of the value of TR_REL_MASK:

```
TR_REL_REJ*
     = ! (REL_REQ* & (TR_REL_MASK == 0))

TR_REL_GRANT*
     = ! (REL_REQ* & (TR_REL_MASK != 0))
```

If all bits in TR_REL_MASK is equal to 0, this means that no active transfer is registered on the source module with address equal to MOD_SRC. Because the transfer is not registered, it can neither be released. The attempted transfer release operation must therefore be rejected.

The entire register file is cleared by asserting the RESET* signal: All outputs of the 4:16 encoder then goes active, thereby enabling all register entries for the writing of new data. The data input is set to all zeroes by selecting the appropriate input of the 2:1 data input multiplexer and the WRITE* signal line is then asserted, thereby writing all zeroes into the register file.

9.4.2.d. Merge Release & Request Signals

As already described, the request and release operations are generating their own acknowledge (GRANT* or REJECT*) and data output signals (MASK and LEFT_RIGHT*). These must be multiplexed into a single set of signals, constituting the output interface of the arbitration mechanism. This multiplexing is done by the "Merge Release & Request Signals" block.

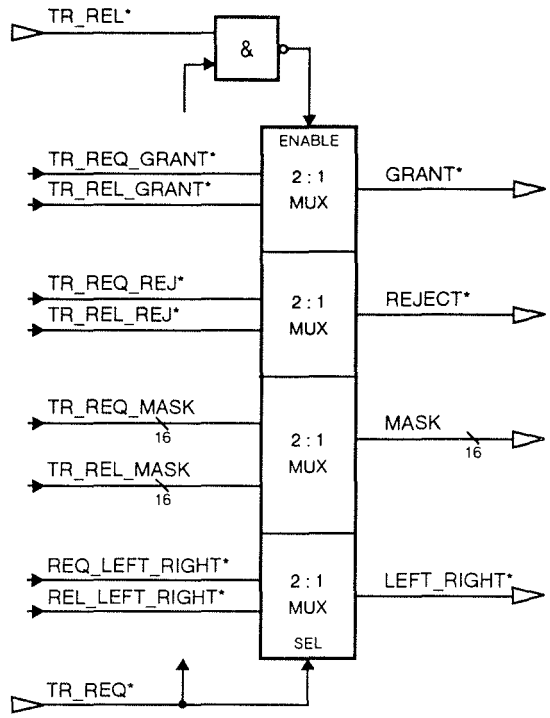


Figure 9.40. Merge Release & Request Signals

9.5. Timing considerations

Beyond the scope presented in the beginning of CHAPTER 9., however, is a detailed solution of *timing* relationships between the functional blocks constituting the Ring bus arbitration mechanism. This issue has thus so far not been discussed. Let us, however, go through some general remarks concerning timing considerations before leaving the arbitration mechanism.

Except from the propagation line blocks searching for the last destination modules ("Find Last Module on Right Transfer" and do. left), all blocks have a fixed, data-independent delay from input to output. Basically, there are then two approaches to ensure correct timing between the blocks:

Synchronous timing

In this case, each block must include an output register, clocked by a clock global to all modules contained in the arbitration mechanism. All output signals generated by the block must be clocked through this output register, before connected to the input of the next block. The clock period must be *greater than the input-to-output delay of the block having the largest delay*, plus the necessary setup and delay times in conjunction with the register operation itself. If the delay of one (or a few) blocks are prohibitively large, thereby slowing down the whole arbitration mechanism, inter-mediate registers within this (these) blocks may be inserted. To be able to

notify a block whenever its input is valid, and thereby is ready to start processing, each block must generate a "VALID" signal, to be connected to the "ENABLE" input of the succeeding block.

With the synchronous timing scheme, a grant/ reject acknowledge will be issued a fixed number of clock cycles after the transfer request/ release command has been presented to the arbitration mechanism.

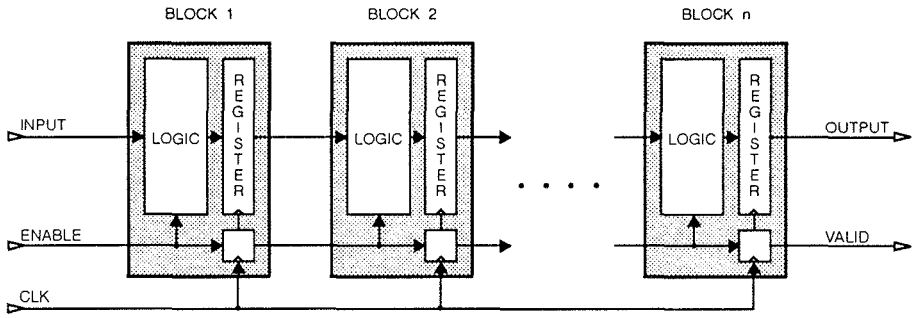


Figure 9.41. Synchronous inter-block timing

Asynchronous timing

Like the synchronous approach, asynchronous timing is based on pairs of VALID and ENABLE signals, allowing a ready result from one block to enable this result to be processed by the next block in the chain. In this case, however, the VALID to ENABLE delay will not be implemented by the means of clocked registers but by a dedicated *delay element*, contained in each block. The delay of this element must be greater than the delay of the logic part of the block, allowing for the necessary setup and delay times when transferring data from one block to the next.

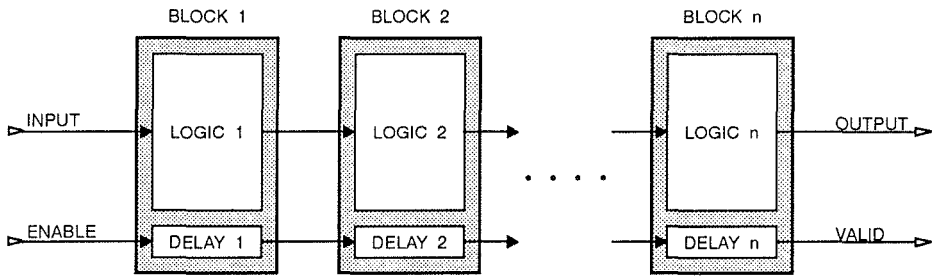


Figure 9.42. Asynchronous inter-block timing

As far as blocks having data-dependent delays are concerned, the simple (and brute force) solution would be simply to assume the worst-case condition. In our case, the two propagation line blocks, although connected in parallel, will then contribute to approximately half of the total delay of the arbitration mechanism alone (asynchronous case). A synchronous implementation without intermediate registers would simply be prohibitive, enforcing this worst-case delay onto every block. Therefore, the design of the propagation line blocks do in fact contain the ENABLE/ VALID mechanism, described together with the block logic.

What is best, synchronous or asynchronous timing? The *synchronous* approach is conceptually the simplest and results in a very "clean" design, but requires more hardware to implement. The *asynchronous* design, at the other hand, can be tuned to optimum performance and requires less hardware, but is regarded as "more difficult" to deal with by most designers. As far as the ENABLE/ VALID mechanism included in the two data-dependent delay blocks are concerned, this is best suited for asynchronous timing. However, by including inter-mediate registers, the synchronous scheme can also be used.

CHAPTER 10.The display system

The final stage in an ultrasound processing pipeline is the display system, presenting the data acquired by the ultrasound transducer to the user (operator). The main task of the display system is of course to present an image as crisp and clear as possible, making it easier to interpret the data. However, another aspect of the display system is just as important: To provide visual feedback and guidance to the user in operating the system when carrying out the examination. The display system must therefore serve two functions, as an *image data presentation* device as well as an *operator interface*. As we will see during the following discussions, these two functions will each have their specific set of requirements as far as the underlying display system architecture is concerned.

10.1. Display system requirements

As already discussed in the introductory part of this thesis, the data acquisition phase and the data interpretation phase of an ultrasound examination will to a much larger extent be separated in future clinical environments than it is today. Probably, they will also be carried out by different people: The actual examination by an ultrasound technician while physician interpret the data and make the diagnosis. Used in this mode of operation,

> *the purpose of the ultrasound instrument's own display system is to provide guidance during data acquisition.*

This in contrast to the data interpretation phase, which will have its own set of display system requirements. For one, processing and display are done off line and is therefore not subject to the real-time restriction as imposed by the ultrasound data acquisition rate. Furthermore, coming features as feature extraction capabilities, tissue characterization, trend analysis and the simultaneous display of multiple image sequences will require high-end graphic workstation capabilities to provide satisfactory results.

Limiting ourselves to the acquisition instrument's display system, however, the requirements can be somewhat lowered. Let us take a look at some of the key issues in this respect:

10.1.1. Display dynamics

If we define display dynamics as "change in the image as displayed on the screen", this change may have two reasons:

- The change in the image data itself. As new data is acquired, the displayed image will change with it.
- The change of any parameter, initiated either automatically or triggered by the system operator, changing the transformation from the set of acquired ultrasound data to the display of the same data.

Although the effect of a change (that is, the displayed image) in some cases may *look* similar (for instance changing the TGC-setting, causing the data to change, compared to changing the compress/ reject transformation parameter), the mechanism behind it is totally different and impose their own requirements as far as providing satisfactory display dynamics are concerned.

The most important aspects when designing a display system is to ensure

> *that the characteristics of the display dynamics of the application are fully supported by the underlying display system architecture.*

This means that the display system should not only take parameters as the amount of data to be displayed and the rate by which they arrive into account, but also how these data are organized and how the update is actually done. As already discussed, the implication of the term "real time" is that the display system itself does not limit the performance of the total system. This should solely be determined by the rate by which it is possible to acquire the ultrasound data. To be able to come up with a more precise definition of the real-time requirement, we will now introduce the following terms:

data acquisition time

> The time between successive acquisitions of 1 unit of data of a display component. The size of a unit will depend heavily on the component in question: A full 2D frame for a 2D tissue or flow image, a vector for a M-mode image and a single data sample for a trace.

display frame time

> Equal to the rate by which the display (screen) is refreshed. Usually equal to 1/50th or 1/60th of a second.

Referring to the previously made distinction between the two types of display dynamics, caused by data and parameter change, respectively, the real-time requirement can now more precisely be defined as:

> *In case of a data change, the change should be effective in the image as displayed on the screen within one data acquisition time measured from the point of time the new data was presented to the display system.*
>
> *In case of a parameter change, the change should be effective in the image as displayed on the screen within one display frame time measured from the point of time the new parameter was presented to the display system.*

In other words: Upon receiving new data, the display system should be able to receive and process those data by whatever rate they are generated. When a change is initiated, affecting image data already stored in the display system, the change should be effective on the next display frame. This also implies that the execution of the change should be synchronized to the screen output.

10.1.2. 2D image

A 2D image as displayed on the screen will consist of a tissue 2D image component and an optional flow image component. Each image component will in turn consist of a number of beams, determining the angular distribution of data. Finally, each beam contains a number of samples, determining the radial (depth) distribution of the ultrasound data. Depending on the particular type of transducer being used, the true geometrical shape of the image will either be like a circle sector or rectangular.

A rectangular image is the result when using a linear array transducer, having a large number of discrete transducer elements organized, as the name says, as a linear array. The pattern by which the data are acquired will in this case have the form of a rectangular grid. To display such an image on a raster scan display, organized as a number of vertical columns by a number of horizontal rows, the only geometrical transformation necessary is therefore a rescaling to fit the image data into the space available on the screen.

In case of a sector shaped image, however, we face a totally different approach to the problem of image display because the acquisition pattern is very different from the raster scan display pattern. As already explained, the image data are acquired, transferred and stored as a number of beams, each consisting of a number of samples. In system memory, the most effective way of storing these data will be in a rectangular fashion as shown in Figure 10.1.

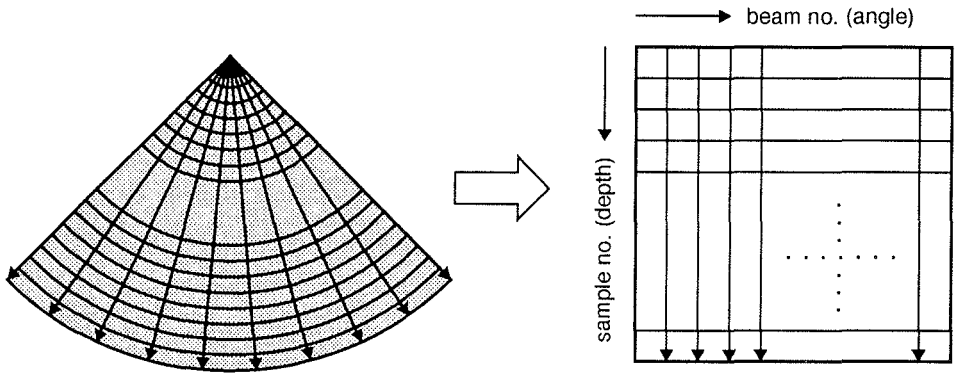


Figure 10.1.Polar data storage format

Because the sector shaped image represents the greater challenge to the display system compared to the rectangular image, the discussion will from now on be limited to the sector image. By changing the appropriate transformation tables, a display system handling real-time sector images can easily be set up to display rectangular images also.

The process of displaying the data must then re-construct the original image as shown in the left part of Figure 10.1. The value of each and every screen pixel p(x,y) encompassed by the outline of the sector must therefore be computed, based on the stored sample values. As shown in Figure 10.2.and Figure 10.3., the number of sample values "behind" each screen pixel will depend on the distance from the centre of the sector to the pixel being computed.

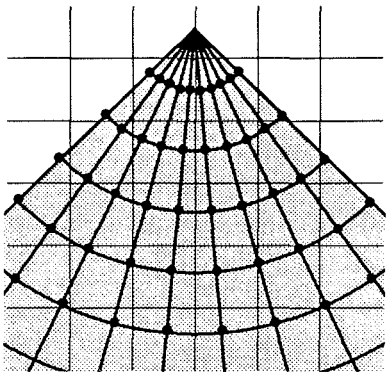


Figure 10.2. Near-field sample distribution

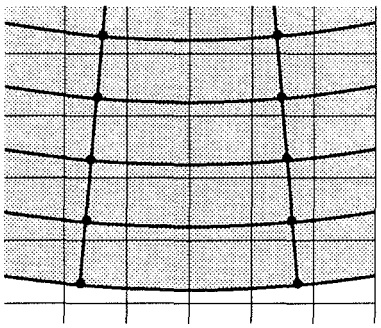


Figure 10.3. Far-field sample distribution

Because all beams are originating from the same point (the transducer element), every screen pixel near the centre will be covered by a number of sample values (Figure 10.2.). In theory, disregarding noise and the fact that the organ being imaged (the heart) is constantly moving, all these samples should have (approximately) the same value. An obvious solution would therefore be to compute the average value of a number of these samples to obtain the screen pixel value.

As far as pixel values distant from the centre are concerned (Figure 10.3.), the approach of the problem is opposite: The samples are spread so far apart that the position of the samples do not coincide with the positions of the screen pixels. In addition, not every pixel position will have its "own" sample value, the pixel values corresponding to these intermediate pixel positions must therefore be computed by some sort of interpolation between the sample values available. One approach to this is to use bilinear interpolation.

Bilinear interpolation

The principle of bilinear interpolation is shown in Figure 10.4.

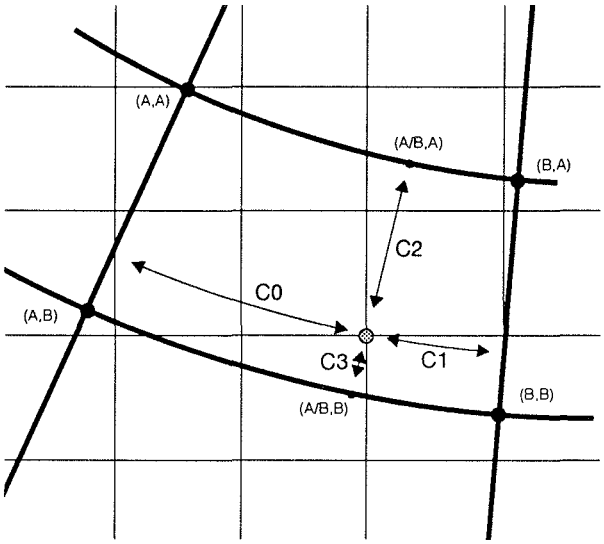


Figure 10.4. Bilinear interpolation

With bilinear interpolation (also called box interpolation), the pixel value in question (from now on called the *refresh pixel*) is computed by interpolating its 4 nearest neighbours ((A,A), (B,A), (A,B) and (B,B)). The first step is, through *coordinate transformation*, to compute the image buffer addresses (beam and sample number) of the 4 neighbour samples. In addition to the x,y-coordinate of the refresh pixel, this transformation is also a function of parameters as the sector's location (offset) and angular and radial size (scale). By using the addresses generated by the coordinate transform, the 4 neighbour sample values are then read from the image buffer. In addition to the sample addresses itself, the coordinate transformation process also computes the 4 interpolation coefficients C0 to C3. These coefficients express the distances from the position of the refresh pixel to the 4 image sample positions along the directions as indicated in Figure 10.4. Compensated for any embedded scaling factor, the sum of these coefficients will always be 1.0.

By interpolation between the 4 neighbouring samples, the value a sample (theoretically) would have had, if taken exactly at the refresh pixel position, can now be computed. This is done in two steps: First, the sample values from the same depth A and B, but laying on an imaginary beam going through the refresh pixel position is computed.

$$(A/B,A) = (A,A)*C0 + (B,A)*C1$$
$$(A/B,B) = (A,B)*C0 + (B,B)*C1$$

Then, by interpolating these intermediate sample values in a second step, the value of the imaginary sample in the refresh pixel position can be computed:

$$P(x,y) = (A/B,A)*C2 + (A/B,B)*C3$$

Having the implementation of such a scheme in mind, the following observations can be made:

OBSERVATION: Regardless of the actual position of the refresh pixel, the 4 sample values required for the bilinear interpolation will always be

- one even beam/ even sample
- one even beam/ odd sample
- one odd beam/ even sample
- one odd beam/ odd sample

As a consequence, for optimum performance the image buffer can therefore be organized as 4 separate banks, with each bank containing all sample values in each of the above mentioned categories (odd/ even beam, odd/ even sample). This allows parallel access to all 4 required sample values, independent of the actual refresh pixel position.

OBSERVATION: By setting all coefficients C0 to C3 equal, the outcome of the bilinear interpolation will be the average value of the 4 neighbour sample values.

In this way, the same mechanism can therefore be used for near-field as well as far-field pixel value computations.

OBSERVATION: The scanning process required to compute all pixel values inside a given sector image is easiest implemented by successively scanning the x,y coordinates of the image to be displayed. The alternative, scanning the original (raw) data in a beam/ sample fashion, requires a two-pass solution: First the value corresponding to the refresh pixel position most "closely" matching (according to some criteria) each 4 sample value set must be computed. In a second pass, the intermediate values in the sector far field can then be filled in as required.

In a raster scan display system, the x,y-based scanning process is already inherent in the system, for generating output to the screen. Therefore, if a sufficiently high bandwidth can be supported, the most effective way to implement the bilinear interpolation would be to do it "on the fly" when generating refresh pixel output to the screen. This is possible in an object-oriented display system where pixel output is generated directly from the image buffers without going through a centralized frame buffer.

10.1.3. M-mode

The M-mode image, with several hundreds of vertical vectors sliding horizontally across the screen, presents quite another approach to the problem of image updating: On each update, the entire M-mode image is shifted one vector to the left, resulting in the "oldest" vector falling off the left edge of the screen while a new vector is inserted at the right edge.

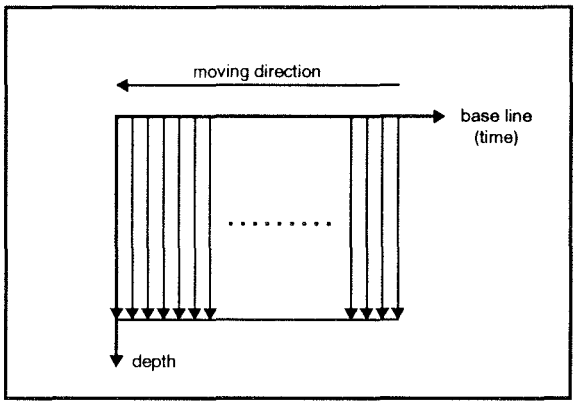


Figure 10.5. M-mode image

Implemented on a conventional frame buffer based system, on every update, the entire M-mode data matrix must be physically moved one column to the left to make room for the new vector on the right. In contrast, in a display system specially designed for this kind of application, vector data will be organized as a ring buffer with length equal to the total number of vectors contained in one M-mode image as displayed on the screen. On update, the data belonging to the oldest vector is replaced by the new vector data, and the pointer identifying the left-most vector on the screen is incremented by one. That is, only the data which actually is to be replaced (one vector out of several hundreds) is affected, the rest remains unchanged. Needlessly to say, this gives a far better performance than the physical move brute force approach.

The display of M-mode data should be supported by a dedicated display module. The design of such a module should be fairly straight-forward and will therefore not be discussed any further in this thesis.

10.1.4. Traces

To support clinical features as ECG and phono, the display system of a medical ultrasound diagnostic instrument must be able to generate trace curves. A trace curve is a one -dimensional curve where a signal's value is displayed as a function of time. Like M-mode, the trace curves are sliding to the left, dumping the oldest data off the left edge while adding new data, entering the image on the right edge. The most general approach to storing trace data would be store the data as a sequence of x/y coordinate pairs. In most cases, however, a fixed sampling rate (i.e. increment along the x (time) axis) can be assumed, making it necessary to store only the y (value) coordinate component.

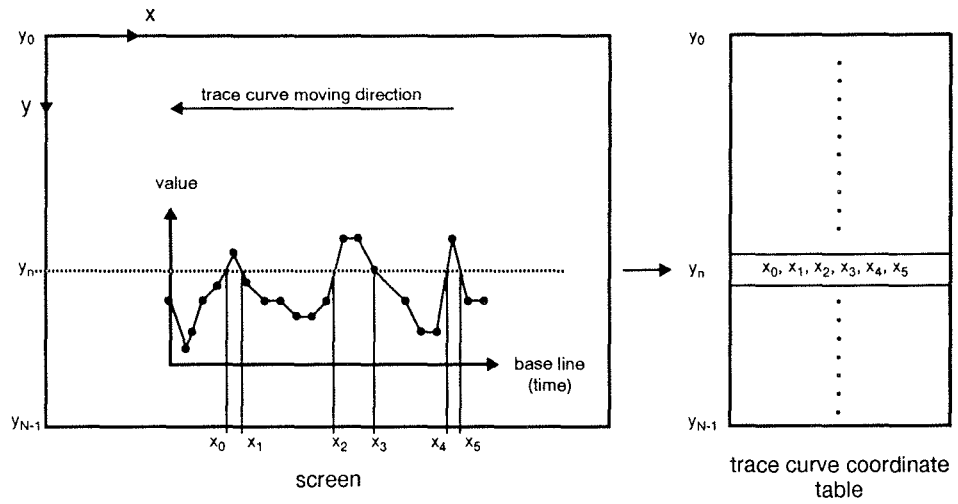


Figure 10.6. Trace curve display

The original, fixed x-increment trace coordinate sequence can then be reorganized into a datastructure based on and sorted by increasing screen y (value) components according to the screen refresh pattern (top-left to bottom-right). This is shown in the right part of Figure 10.6.. With the input taken from this datastructure, special hardware can now be designed to generate pixel output directly from the trace data "on the fly" with refresh pixel output to the screen. The required pixel filling feature illustrated in Figure 4.4. can easily be implemented as a "sample-and-hold" mechanism using the current x-value until the previous (next) x-value is encountered. However, because of the comparatively small amount of data involved, the task of generating the traces can easily be handled by the systems graphic processor, responsible for generating all other graphics to be displayed on the screen. No special-purpose hardware for trace generation is therefore to be included in the display system.

10.2. Display system architectures

For computer graphics and image processing applications ([Newman 1979], [Hearn 1986], [Foley 1990]), there are two basically different approaches available for implementing display systems: Raster scan and random scan displays.

In a **raster scan** display system, the display screen can be viewed upon as a raster consisting of a number of (vertical) columns and a number of (horizontal) rows. Every displayable point (picture element, *pixel*) on the screen are identified by its unique column (x) and row (y) address pair. Output to the screen is generated by scanning the screen raster in a regular fashion, row by row, left to right, starting in the upper left and ending up in the lower right corner. The pixel values from which the screen is refreshed are either stored in a centralized, dedicated memory module called a *frame buffer* or is distributed over several modules in the system.

A **random scan** display system does not have a pixel memory, storing every pixel value of the display screen. Instead, the displayed image is stored as parameterized graphic commands in a *display list.* Examples of such commands are instructions for displaying vectors and text strings. The actual display as seen on the screen is generated by that the list of commands contained in the display list is continuously executed by the system's display processor, in an endless loop. To avoid a flickering display, the display screen must be (re)generated at least 30 to 60 times pr. second [Foley 1990], which sets a limit to the length of the display file. Random scan display systems are also called *vector display* systems.

Due to the principle of storing the display as parameterized commands rather than pixel maps, random scan display systems are not suited to our kind of application where the data to display has the form of 2 -dimensional arrays of data values. In the rest of this thesis, the discussion will therefore be restricted to *raster scan* display systems. As already indicated, raster scan display systems can further be divided into two groups, depending on whether they are based on a centralized frame buffer memory or not.

10.2.1. Frame buffer based display systems

In a frame buffer based display architecture, the displayed image is stored in a dedicated memory module called a *frame buffer*. This is shown in Figure 10.7.

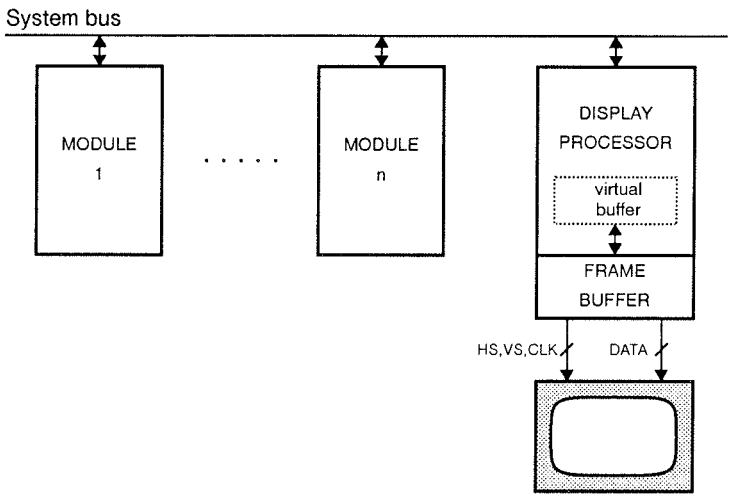


Figure 10.7. Frame buffer based display architecture

All screen refresh is done from the *pixel map* stored in the frame buffer. As far as frame buffer updating is concerned (that is, writing new data into the frame buffer), this is handled by a *display processor*. If a module wants to display data, these data must therefore first be transferred to the display processor via the system bus, formatted as a *raster primitive*. A raster

primitive may either have the form of a graphic command similar to the display list commands found in random scan systems, or it may be an array of data to be loaded into the frame buffer (and thereby displayed) at a specified location.

To increase buffer update speed, the function of the display processor can be distributed over a number of Processing Elements (PEs), with each PE responsible of its own part of the frame buffer. Determined by the position it will have on the screen, a raster primitive is processed and the buffer updated by one or more PEs. Several systems are using variants of this technique ([Torborg 1987], [Akeley 1988], [Runyon 1987], [Potmesil 1989]). Developed into its extreme, each pixel could be equipped with its own local processing capability as in the *Pixel-Planes 5* machine [Fuchs 1989].

The display is usually refreshed at a rate of 50 to 70 frames pr. second by scanning the frame buffer row by row, left to right, from top to bottom. To be able to manipulate the image data "on the fly" as they are displayed on the screen without actually altering the contents of the frame buffer, the data values stored in the frame buffer data are mapped through a table called the *Video Look-Up Table* (VLUT) before they are displayed on the screen. Functions as histogram equalization, data tresholding and image double buffering can be implemented simply by manipulating the contents of the VLUT. The length (number of entries) of the VLUT is determined by the number of frame buffer planes (that is, the number of bits used to represent each pixel value), while its width is determined by the number of colour shades the display must support. Typically, each primary colour (Red, Green, Blue) is displayed using 8 bit each (256 shades), giving a total VLUT width equal to 24 bits.

As already explained, screen refresh is done independently of frame buffer updating. However, because the frame buffer is a global resource, to be shared by the refresh mechanism (reading data out from the frame buffer) and the display processor (writing data into it), its access and use must be regulated. This can be done in several ways, depending on the technology by which the frame buffer memory is implemented. In practice, there are two alternatives: Static RAM (SRAM) or Video RAM (VRAM).

10.2.1.a. Static RAM frame buffer

Currently (October 1991), state-of-the-art SRAM technology can offer 1 Mbit memory chips organized as 128k*8 or 256k*4 with 25 ns random access time. With such chips, large, high-bandwidth frame buffers can be implemented. As far as access sharing is concerned, three different solutions to this problem is possible: Double buffering, split-phase access and screen retrace update.

Double buffering

In a double buffered system, there are two totally symmetric frame buffers. While the screen is refreshed from one buffer (the refresh buffer), the display processor writes new data into the other buffer (the update buffer). When the update is finished, the two buffers exchange roles and the screen is then refreshed from the newly updated buffer while the other buffer is available to the display processor for further update.

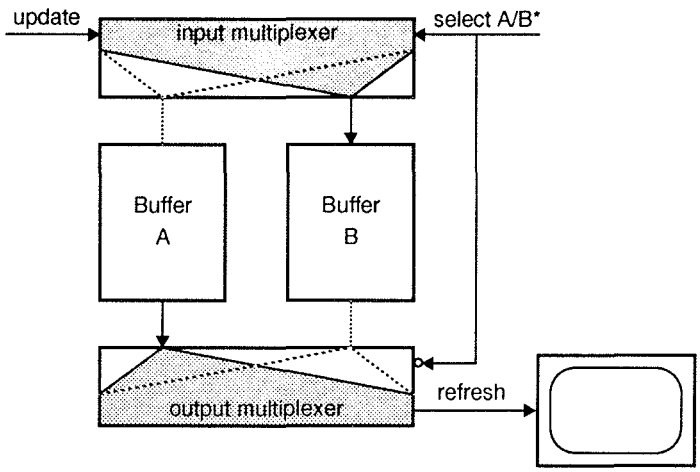


Figure 10.8. Double buffering

The double buffering scheme has the following properties:

- Buffer update and screen refresh is totally invisible to and independent of each other, both in terms of the speed by which the update is done and the time it takes.
- Because the amount of memory required is actually doubled compared to the other schemes, this approach is very resource demanding. In addition to the extra amount of memory, the data and address paths must also be doubled and multiplexed to implement full symmetry between the two buffers.

Double buffering can in practice only be used in situations where the entire contents of the frame buffer is reloaded on each update. In case of a partial update, the update buffer must first be initialized to the state of the refresh buffer, requiring either a shadow memory of some form (e.g. display list) or a physical copy of the refresh buffer to be made. In the latter case, a third type of access must be accommodated (refresh, update, copy), creating a more complex problem than the one we originally started with.

Split-phase access

With split-phase access, every other access cycle is allocated to screen refresh (refresh cycle) and frame buffer update (update cycle), respectively.

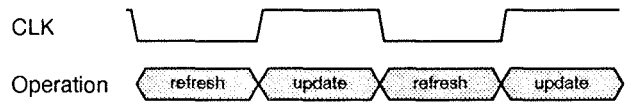


Figure 10.9. Split-phase access

Key features of the split-phase access approach are:

- Simple implementation compared to the double buffer scheme. The required multiplexing of the refresh and update address and data paths can easily be done by controlling the output enable mechanism of the respective drivers/ transceivers.
- In a medium-grade display system, the bandwidth necessary to refresh the screen will typically be in the area of 40 Mpixels pr. second (peak rate), or in other words, 1 new refresh pixel value each 25 ns. With frame buffer access time equal to 25 ns, in each refresh cycle two refresh pixel values must therefore be accessed/ produced in parallel to support the required bandwidth.

Screen retrace access

Depending on physical display characteristics as screen resolution and refresh rate, 25 to 30% of the time is used for screen retrace, horizontal and vertical ([Perez 1988], [Foley 1990]). During this period of time, the refresh mechanism is not accessing the frame buffer and buffer update can therefore freely be done without interference from the refresh process.

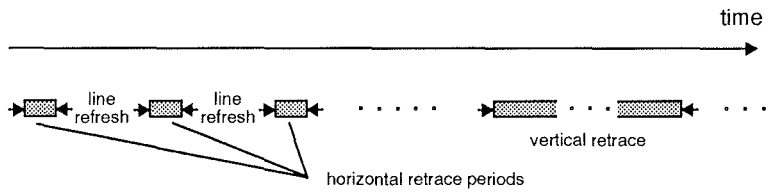


Figure 10.10. Screen retrace access

This approach to the problem has the following advantages and disadvantages:

- As for the split-phase access scheme, controlling the output enable mechanism of the data and address drivers will do the multiplexing job, resulting in a simple implementation.
- The full bandwidth of the frame buffer memory is available for screen refresh.
- Only a limited time is available for frame buffer update.

The significance of the latter point depends on the rate and pattern by which data enters the display system. If the average data rate is less than the sum of the retrace periods, the scheme is in principle feasible. However, a large portion of the time available for frame buffer update will be concentrated to the vertical retrace period. A large intermediate buffer (e.g. FIFO) may therefore be necessary to accommodate data received outside this period.

10.2.1.b. Video RAM frame buffer

A Video RAM (VRAM) is a Dynamic RAM (DRAM) containing a serial output port in addition to the usual random access port. Data from all columns (typically 512) are loaded in parallel into the serial port shift register and can then be clocked out serially at high speed (up to 40 Mhz). Compared to an ordinary DRAM's access times of typically 200 and 55 ns (random access and fast page mode), the serial output port benefits system performance in two ways:

- **Increased output bandwidth.** The possible data output rate is considerably higher (at least 100%) than what is achievable with a standard DRAM, even when used in the fast page mode.
- **Increased average input bandwidth.** Except from the parallel load operation of the serial output port, the random access port can be accessed totally independent of screen refresh. Most of the time (in the order of 95%) is therefore available to the display processor for frame buffer update. The *peak* input bandwidth compared to a standard DRAM is however unchanged.

Therefore, even when utilizing the fast page mode, input bandwidth may not suffice for some applications. Therefore, another high speed serial port has been added ([Bursky 1990], [Wilson 1990]). This port can be used either for data input or data output, making it possible either to double the output bandwidth or to do high speed frame buffer update. The two serial ports have their own clock and can therefore operate in total asynchronism with each other. One possible application of the second serial port is therefore to synchronize an incoming pixel stream to the screen refresh rate.

Although doubling the peak and average input bandwidth compared to the fast page mode of a standard DRAM, the same limitation applies to the serial input port as it does to fast page mode access: Data must be accessed one row at a time. The *random access* time is still in the order of 200ns. For applications having an irregular frame buffer update access pattern, making it impossible to utilize the fast access modes, this may be too slow. However, this problem of large frame buffer access times can in some situations be alleviated by using a virtual buffer.

Virtual buffer

The primary motivation for a virtual buffer memory is to achieve increased performance by exchanging the large and slow frame buffer memory for a smaller and faster virtual buffer memory to. All display processor access of the frame buffer will then be performed through the virtual buffer, which in turn is loaded into the frame buffer by using some high speed, block transfer mode. A random, low speed access pattern prescribed by the display processor algorithm is thereby converted to regular, high speed frame buffer updates. The virtual buffer can then be reused to build another part of the screen display. In this way, the whole image can be constructed, one piece at a time.

As described by [Gharachorloo 1989], virtual buffers can be used in two distinct ways: to support sweep algorithms and as a pixel cache.

Sweep algorithms makes one pass over the entire image and assigns the virtual buffer to successive portions of the image in turn. For each assignment, the virtual buffer is cleared, and all primitives to be output to the portion of the screen covered by the virtual buffer's current position are rasterized and written into the virtual buffer. The contents of the virtual buffer is then copied into the frame buffer, and the virtual buffer is moved to another portion of the screen, repeating the same procedure until the entire image is updated.

Needless to say, to obtain maximum performance, the primitives must be sorted in a way that allows all primitives corresponding to the same virtual buffer to be rasterized together. The size of the virtual buffer is a trade-off between cost and speed for a large buffer compared to the increased transformation and clipping overhead associated with a small buffer. Depending on the characteristics of the particular application in question, there are several virtual buffer organizations possible. Due to their application specific properties, they will, however, not be discussed any further here. An example of a sweep algorithm virtual buffer is shown in Figure 10.11.

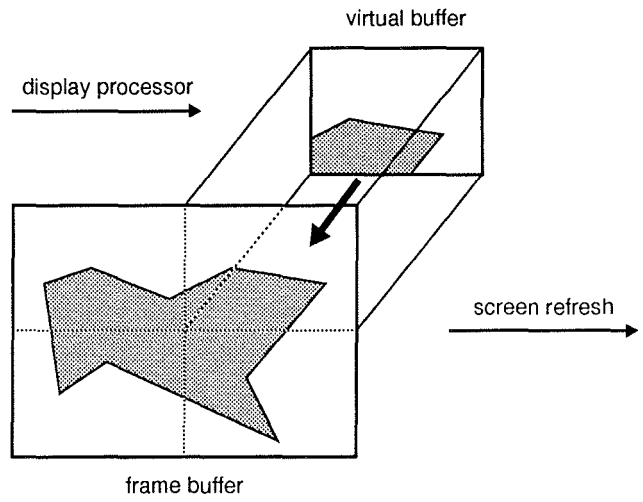


Figure 10.11. Sweep algorithm virtual buffer

The virtual buffer shown in Figure 10.11. is in size equal to one forth of the frame buffer and can be positioned to cover each of the four frame buffer quadrants in turn. On each position, (preferably) all primitives to be displayed in this quadrant is rasterized by the display processor and written into the virtual buffer before the virtual buffer in turn is loaded into the frame buffer. As far as screen refresh is concerned, this is done directly from the frame buffer.

As far as virtual buffers used as **pixel caches** are concerned, this is equivalent to the use of caches to speed up main memory access. Unlike the sweep algorithm approach, the resolution by which the pixel cache can be positioned in the frame buffer is greater than the size of the cache itself. In other words, the pixel cache can take overlapping positions. During the rasterization of a scene, the contents of the cache can be exchanged with the contents of the frame buffer many times. One system using the pixel cache scheme is the Stellar GS1000 graphic engine [Apgar 1988], integrating the pixel cache with the main memory cache for increased flexibility.

10.2.2. Object oriented display systems

In an object oriented display system, there is no dedicated frame buffer memory containing a complete pixel map of the displayed image. The total image is instead assembled from a number of *image components*. Rather than having a centralized, dedicated display processor, the task of updating and refreshing those components on the screen is distributed over a number of *display modules*. During display refresh, the pixel values are transferred in real time over a dedicated *pixel bus* from the display modules to the display controller, and further to the display screen. The outline of an object oriented display system is shown in Figure 10.12.

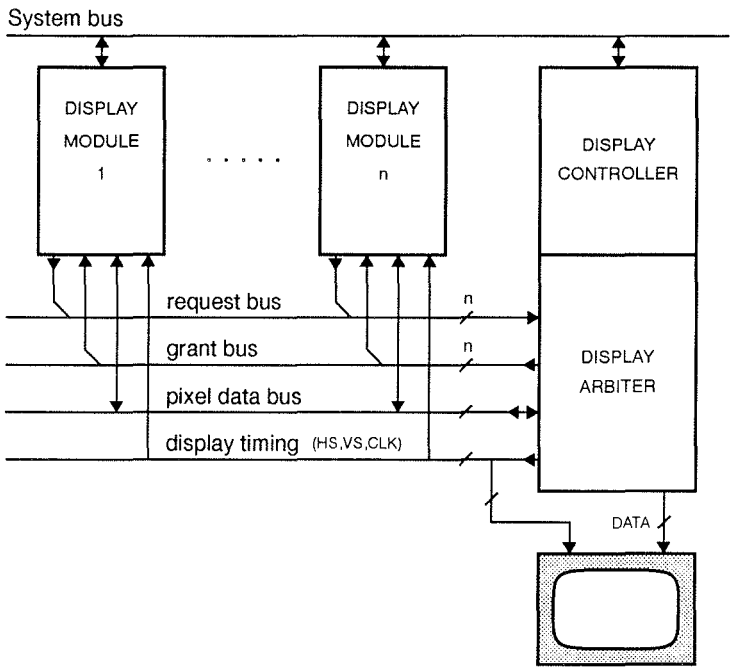


Figure 10.12. Object oriented display system

As far as refresh timing is concerned, every display module must be in total synchronism. Therefore, the screen timing signals as horizontal and vertical sync as well as the pixel clock must be distributed to all modules taking part in the display refresh. Based on these signals, each

display module is able to maintain its own pair of X- and Y-counters, at any time keeping track of the position of the pixel being refreshed. Because the display screen in an object oriented display system is a shared resource, global to several modules, its access and use must be regulated. This is done by a functional module within the display controller called the *display arbiter*, according to a

10.2.2.a. Display request/grant protocol

Each image component to be displayed on the screen has a specified position, size and shape, defining the component's *active area*. When the refresh pixel position is inside the component's active area, the display module signals this to the display arbiter by asserting a *display request* line. To make the display arbiter able to distinguish requests from different display modules, each display module will have its own request line, all connected to the display arbiter.

Because the entire screen area in principle is to each individual display module's disposal, conflicts may occur in case of overlapping active areas. Such conflicts will be resolved by the display arbiter, based on a *display priority table*. Multiple requests can then be resolved and a *display grant* issued to one and only of the requesting modules. After receiving the grant signal, the granted module is allowed to output its pixel value onto the pixel bus.

Display arbitration is in this way achieved on a pixel-by-pixel basis.

Like frame buffer based systems, display data to be output to the screen are mapped through a Video Look-Up Table (VLUT) before they are displayed. However, due to the distributed organization of the object oriented system, the VLUT may in this case be implemented as a two stage mechanism: In addition to the global VLUT located on the display controller, mapping the data received from every display module, each display module may in turn contain its own VLUT. The local VLUT will map the pixel data before they are output to the pixel bus.

Compared to a frame buffer based system, an object oriented display system has the following characteristics:

- Better dynamic properties. Because no physical movement of data is involved, image components can easily be moved around within the total screen area simply by changing the corresponding display module's X- and Y-pointers, determining the position of the image component on the screen.
- Better depth arbitration dynamics. By manipulating the contents of the display priority table, the mutual priorities (e.g. foreground, background) of the image components can easily be changed

To achieve similar performance with a frame buffer based system, each image component must have had its own set of frame buffer planes, facilitating depth arbitration by manipulating the contents of the Video Look-Up Table (VLUT). However, this approach is prohibitive already for a small number of image components due to the amount of memory required. Alternatively, there must be included an additional number of dedicated frame buffer planes to support functions as clipping, display modes and overlays [Rhoden 1989].

- Local VLUT support.

- Instead of having one, more or less general purpose, display processor, the hardware of the individual display modules can be tailored to the properties of the image component(s) they support.
- As a consequence of this, on-the-fly processing directly on the refresh pixel stream to the display is more feasible with an object oriented system than it is with a frame buffer based system.
- Unlike a frame buffer system, the refresh pixel stream has to be transferred over a bus connecting several modules. In addition, display arbitration must be supported at the same rate. This may limit the display's bandwidth, or in other words, the display's resolution.

The bandwidth limitation may however, to some extent be alleviated by pipelining the "request/ grant/ output data" cycle as shown in Figure 10.13.

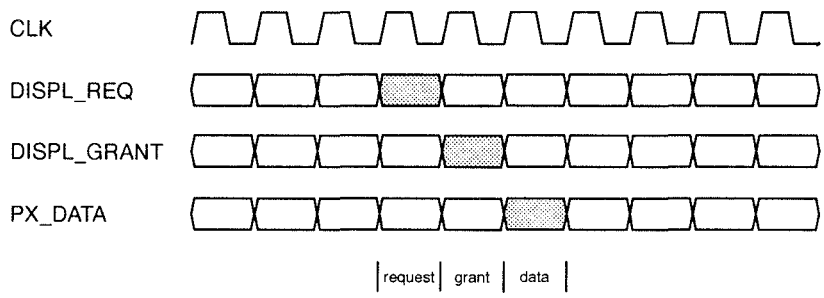


Figure 10.13. Pixel bus pipelining

An alternative approach to increasing the bandwidth would be to widen the pixel bus, permitting several pixel values to be transferred simultaneously. An inherent implication of this latter approach is that the resolution of display arbitration is reduced to the number of pixels transferred in parallel over the pixel bus. As far as display modules representing blocks of data are concerned, this reduction of arbitration resolution is of no practical importance. For text and graphics, however, arbitration should be done on a pixel-by-pixel basis. The problem can be solved by placing the text/ graphics generator on-board the same physical module as the display controller. Text and graphics data can then be merged into the pixel stream just before output to the VLUT with one pixel resolution. However, all things taken into consideration, it is in the author's opinion nevertheless fair to say that high resolution requirements are easier supported by a frame buffer based system than with an object oriented display system.

10.2.3. Hybrid systems

A hybrid display system as defined here is an object oriented display system where one (or more) of the display modules contains a frame buffer. The data output from the frame buffer is merged with the display data from the other display modules according to some pre-defined priority assignment. Typical application examples of such local frame buffers are for text and graphics support.

10.2.4. Multiple window support

The display layout of a medical ultrasound diagnostic instrument will vary with its current mode of operation: 2D imaging, Colour Flow, M-mode, combined modes, replay of data already acquired and stored etc. To be able to accommodate the different modes, at any time utilizing the available screen area in the best possible way, *multiple windows* should be supported. In a multiple window display system, windows can be created, moved, resized and closed at need.

Due to its distributed nature, object oriented display systems are well suited to support multiple windows. The simplest way to implement a multiple window system is to let each window be handled by its own display module. Each module could then, within the framework of a set of pre-defined restrictions, freely manage its own window. The only action affecting several windows would be a change in the windows' mutual priority, done by changing the contents of the display priority table.

However, often this "one module - one window" approach is not optimal. Two variations of the scheme is possible:

10.2.4.a. One module - several windows

If the display image contains several windows of the same type, it will many times be possible to share hardware resources between those windows. In such cases, one module must be capable of supporting more than one window.

An example of this can be the display of multiple 2D image windows. If the scan conversion is done "on the fly" during screen refresh, the different windows' use of the bilinear interpolation mechanism with its related hardware would be exclusive in time and can therefore be shared among the windows.

10.2.4.b. One window - several modules

If one window is to contain several image components, each component having different display characteristics, it could be advantageous to let the different image components, belonging to the same window, be handled by different display modules. Each module could then be tailored to one type of image components. Consequently, the support of each window is then distributed over several display modules.

As far as 2D image windows are concerned, they must in addition to an image component contain a graphics component. The latter would include the window outline, depth markers, any textual description of the data currently being displayed in the window as well as necessary graphic markers for measurement and analysis support. However, in sum this not enough to justify one graphic generator dedicated to each window, it should be shared between several windows. In this way, a set of 2D image windows would then be handled by two different display modules, one image module and one graphic module.

From a system point of view, the problem of keeping window descriptions consistent between the two modules now arises. Or, in other words,

how to guarantee that the different image components belonging to the same window, generated by different modules on the basis of the same set of display parameters, behave as they were one image component, generated by one module?

To ensure that the entire screen is refreshed on the basis of a consistent set of display parameters, display parameter update should be synchronized to the screen vertical retrace. Referring to the example of distributing the image and graphics components in 2D image windows, it would be most unfortunate if the two components did not move or scale together. Principally, it is two ways of ensuring that this will not happen:

Update synchronism

If updating the display parameters is done in such a way that

- the display parameters on all (involved) display modules are updated at the same time.

and

- they are made effective on all (involved) display modules as soon as they are received,

the required synchronization is embedded in the point of time by which the new display parameter set is transferred to the display modules.

The approach of update synchronism has the following implications:

- The distribution of display parameters must be done on a dedicated bus, guaranteed to be available when it is needed.
- A broadcast protocol should be used, ensuring that all modules receive the same message at the same time.
- Synchronization to screen vertical retrace can be done by the point of time by which the display parameters are distributed.

The obvious solution, meeting all three requirements listed above, would be to broadcast the set of display parameters to be updated from the display controller to the display modules by utilizing the pixel bus during the vertical retrace period. The pixel bus is in this period idle, and the synchronization to vertical retrace is automatically achieved. To make the display modules able to detect display parameters being output on the pixel bus, a dedicated signal line (DP_VALID) must be used. This is shown in Figure 10.14.

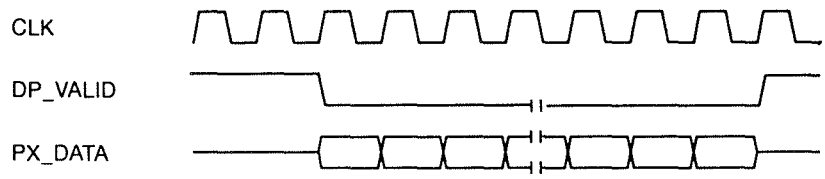


Figure 10.14. Display parameter broadcast

Execution synchronism

Execution synchronism must be applied when it can not be guaranteed that all display modules will receive the new set of display parameters simultaneously or within a specified interval of time. This would be the case if a shared bus without broadcast capabilities is used for transferring the display parameters.

This scheme will have the following consequences to system design:

- A handshake mechanism to make the display modules able to signal when they have received and effectuated the new set of display parameters must be included.
- Each module must individually synchronize the effectuation of the new display parameters to the screen vertical retrace.

As a conclusion, the method of update synchronization is, when it can be implemented, by far the one to prefer.

10.2.4.c. Window identification

In addition to the issue of synchronization, the distribution of image components over several modules raises another question:

> *How is the identity of the window currently being displayed communicated between these modules during screen refresh?*

Basically, there are two alternative approaches for doing this:

Identification by position. If a window's identity is implicit in the window's position on the screen, no explicit tagging of the window's identity is required. A prerequisite for this scheme to work is that display parameter updating can be synchronized as previously discussed.

Identification by tagging. In this case, the pixel bus is augmented by a *window tag*, identifying the window whose data is currently on the bus.

The important difference between the two schemes, is that identification by tagging allows overlapping windows: If multiple windows are to be supported by a single module, e.g. one scan converter generating image data for four different windows, window arbitration must take

place *before* the data are output onto the pixel bus. With overlapping windows, position alone is no longer enough to determine which window the pixel currently being refreshed belongs to. Of course, the window select & clip mechanism can be duplicated on the two modules. A much better way of taking care of this problem is to use identification by tagging by including window tag lines in the pixel bus signal lines. With a maximum of four windows, two tag lines are needed.

Window tagging schemes are also used in frame buffer based systems. In that case, a small number of bitplanes are reserved for tag storage, identifying the window (drawing process) to which the pixel value having the same location in the frame buffer belong. By including the window tag in the (pixel) value input to the Video Look-Up Table (VLUT) during screen refresh, each window can then have its own look-up table [Voorhies 1988]. This is also feasible in an object oriented display system.

10.2.5. Image buffer timing requirements

One of the most important parameters when specifying any kind of data buffering system is its *access time*. However, in many applications, another parameter has often greater influence on the buffer's bandwidth (which this is actually all about), namely the *access pattern*.

The access pattern determines how (or if) the buffer can be parallelized in order to increase its performance. From the discussions carried out in preceding sections of this chapter, the following conclusions can be made:

- The image data should be organized as 4 separate banks according to the least significant bit of their sample and beam addresses (even/ odd beams, even/ odd samples).
- The scan conversion procedure should preferably by done directly on-the-fly during screen refresh due to the scanning mechanism required.
- As far as the issue of buffer update/ screen refresh sharing is concerned, this should preferably be solved by that the buffer is updated during the screen retrace periods.

As a coarse estimate, the retrace time are accounting for 25 to 30% of the total refresh cycle and is mainly concentrated to the vertical retrace period. Being restricted to do buffer update during the retrace periods only, large buffers (FIFOs) may therefore be necessary to accommodate cases where large amounts of data are received immediately after a vertical retrace period has elapsed. To achieve the *average* buffer update rate required, assuming a 25% retrace period and that the number of data samples to be updated is equal to the number of pixels to be refreshed from the same buffer, *peak* buffer update rate should be at least 3 times the pixel refresh rate. This can be achieved by utilizing the 4 bank organization originally determined by screen refresh requirements also for buffer update. Image data can then be loaded into the image buffer 4 samples at a time, which will be enough to accommodate the average buffer update bandwidth required.

However, to avoid the need for large FIFO buffers, the image buffer update time should be more evenly distributed over the entire display cycle. Instead of restricting the buffer update time to the retrace periods, this can be obtained by augmenting the update time available to encompass the *total time* the buffer's data is not displayed on the screen.

In addition to the retrace periods, this definition will also include the time when the screen is refreshed from another source than the buffer in question. For a multiple window system, the maximum size of each window can without losing to much functionality be limited to an area (or actually a width) somewhat less than the total width of the screen. Sticking to the assumption of a 4-parallel input path and disregarding the horizontal retrace period,

> *a maximum window width equal to 80% of the total screen width will ensure that image buffer update will keep up with screen refresh on a line-by-line basis.*

In other words, a FIFO deep enough to store the number of image data samples received from the Ring bus during *one* scan line time will be sufficient. An important assumption for this to work is that the rate by which the FIFO can be unloaded (into the buffer) is greater than the rate by which it is loaded from the Ring bus.

What still is left to decide, is the implementation of the image buffer. As discussed earlier in conjunction with frame buffer based systems, there are two alternatives when designing large, high-speed buffers. Video RAMs (VRAM) or Static RAMs (SRAM).

Video RAM

The use of Video RAMs (VRAM) for image buffer implementation can unfortunately in practice be excluded from the very beginning by the mere fact that we want to do on-the-fly scan conversion, during screen refresh. The reason for this implication is that the unlinear access pattern required by the polar-to-rectangular scan conversion algorithm precludes the use of regular (and high speed) access mechanisms as the VRAM's serial output port and the fast page mode. This leaves the random access port, having an access cycle time in the order of 150 to 200 nanoseconds, which is at least 6 times to slow to support the refresh bandwidth required (25 ns pr. pixel). Even though the possibility of multiplexing the required number of banks into a single, high speed data stream feeding one bilinear interpolation mechanism is theoretically feasible, it would lead to a very complex and costly implementation.

Additionally, the problem of slow access must also be dealt with on the image buffer's input side. With an access time of 200 nanoseconds, the maximum rate by which 512 by 512 (beams, samples) images can be loaded into the image buffer will be 20 images pr. second. If we for a moment disregard the fact that the image buffer's total bandwidth must be shared between input and output, this is at least a factor of two too slow to comply with the real-time requirement of the total system. Fortunately, the effective access time as far as buffer update is concerned can be reduced in two ways:

- By splitting the buffer into separate banks. Ideally, this splitting should support *both* data input from the Ring bus as well as refresh data output. Although coinciding requirements can be found, the even/ odd sample

partitioning will for instance be suitable in both cases, conflict of interests will probably arise.

- Utilizing the fast page access mode. Because image data are transferred beam by beam, they can be loaded into the buffer page by page. Alternatively, the second serial port of the triple port VRAM can be used.

However, all things taken into consideration, a VRAM (DRAM) implementation of the image buffer is in the author's opinion not a optimum solution when scan conversion is to be performed in real time during screen refresh.

Static RAM

As far as the implementation of screen refresh scan conversion is concerned, state-of-the-art SRAM technology offering 1 Mbit/ 25 nanosecond random access time memories is representing a perfect match for available 40 Mhz coordinate transformation and bilinear interpolation chips. The usual SRAM disadvantages of high cost, high power dissipation and low density compared to VRAMs are of course still valid, but at a level not preventing an actual implementation. The image buffer is therefore to be implemented by the use of Static RAM memory.

CHAPTER 11.An actual display system design

In the previous section, the properties of different display system architectures were discussed. A comparison of the alternative architectures, based on the requirements of a medical ultrasound diagnostic instrument, gave the following coarse outline of a suitable display system architecture:

- Object oriented system architecture.
- Multiple, overlapping window support. Window identity tagged by dedicated signal lines on the pixel bus.
- One module - several windows: To optimize utilization of expensive hardware resources, expensive in terms of cost, board real estate as well as backplane slot usage.
- Several modules - one window: Consistency is ensured through broadcasting display parameters on the pixel bus during the vertical retrace period.
- Image data scan conversion is performed on-the-fly during screen refresh.
- Static RAM based image buffer. Buffer update is done when the buffer is not supplying refresh data to the screen (that is, outside the window's active area as well as during the horizontal and vertical retrace periods).
- To support bilinear interpolation, the image buffer is organized as 4 individually accessible banks. The partitioning is done according to the two-bit combination of the least significant bit of the beam and sample addresses, respectively (even/ odd sample, even/ odd beam).

As far as the graphics part of the display system is concerned, the requirements will be relatively modest, measured in terms of the number and vectors and characters which must be (re)generated every second. Compared to the capacity of a state-of-the-art graphics controller together with a high-speed frame buffer, an implementation satisfying the performance and display dynamics requirements imposed by the rest of the system will represent no problem. The graphics part of the display system will therefore not be discussed any further in the remaining part of this thesis.

For the 2D image part, however, responsible for supplying image data to the display screen by doing multiple window scan conversion of ultrasound data in real time, no off-the-shelf hardware can offer satisfactory performance. In the last section of this thesis, a relatively detailed design of a 2D image part, complying with the above listed set of specifications, is therefore presented.

11.1. Screen resolution

Other than the less specific term "medium to high grade resolution", the resolution of the display screen has not yet been specified. Starting with being a a bit more precise, a resolution in the order of 512 by 512 pixels (medium) up to 1024 by 1024 pixels (high) is what we are aiming at.

The reason for not bringing the screen resolution into the discussion until now, is that in the design of a display system, there will often be a conflict of interest: Between the always present demand, or wish, for a resolution as high as possible on one side and other display system properties on the other side. An example of such a trade-off is high resolution vs. good dynamic characteristics: The former is best served by a frame buffer based system while the latter is easier achieved in an object oriented system. As far as our specific application is concerned, we have focused on the dynamic properties of the system, the resolution must then be determined within the framework of that decision. However, the resulting resolution must of course satisfy the need of the application, otherwise the system must be redesigned so that need is met. Some element of *iterative processing* will therefore often be involved in the development of a system as described in this thesis.

From the display system parameters specified, the following boundary conditions exists as far as the screen resolution is concerned:

- Maximum window size: 512 by 512 pixels.
- Horizontally, the window size should be less than 80% of the total length of a scan line to allow sufficient time for update (section 10.2.5.).

With 512 pixels being equal to 80%, the total length of each scan line must therefore be at least 640 pixels. To make additional room for "secondary" display components as patient id-data, Colour Flow maps and TGC-settings located beside the image windows, 800 pixels pr. line is suitable as an initial selection. It is also a standard figure as far as horizontal display resolutions are concerned (PC VGA). Sticking to the defacto standard of an aspect ratio equal to 4:3, assuming quadratic sized pixels, this means a vertical resolution of 600 scan lines. To get a flicker-free display, a refresh rate equal to 60 Hz non-interlaced is chosen.

- Refresh rate equal to 60 Hz non-interlaced is chosen.The question is however: How do these figures comply with what is practically realizable using available technology?

With 800 by 600 pixels, 60 frames pr.second refresh rate and a retrace overhead factor equal to 1.3, pixel values must be transferred over the Pixel bus during screen refresh at a rate of 27 ns pr. pixel. This represents an almost exact match to 25 ns. SRAM chips and 40 Mhz coordinate transform and bilinear interpolation chips. As far the Pixel bus is concerned, 40 Mhz may be too fast because arbitration is to be done on a pixel-by-pixel basis, implying that the transceivers must be turned on and off at the same rate. By transferring the pixel values two by two, the necessary bandwidth is reduced to 20 Mhz, which will represent no problem to a bus with a small number of closely spaced modules. The conclusion is therefore:

The display system as specified in the preceding chapter can, by using current state-of-the-art technology, be implemented with a screen resolution equal to 800 pixels by 600 lines. The refresh rate is assumed to be 60 Hz, non-interlaced.

Architecture outline

An image display system architecture, conforming to the key features listed, is shown in Figure 11.1.

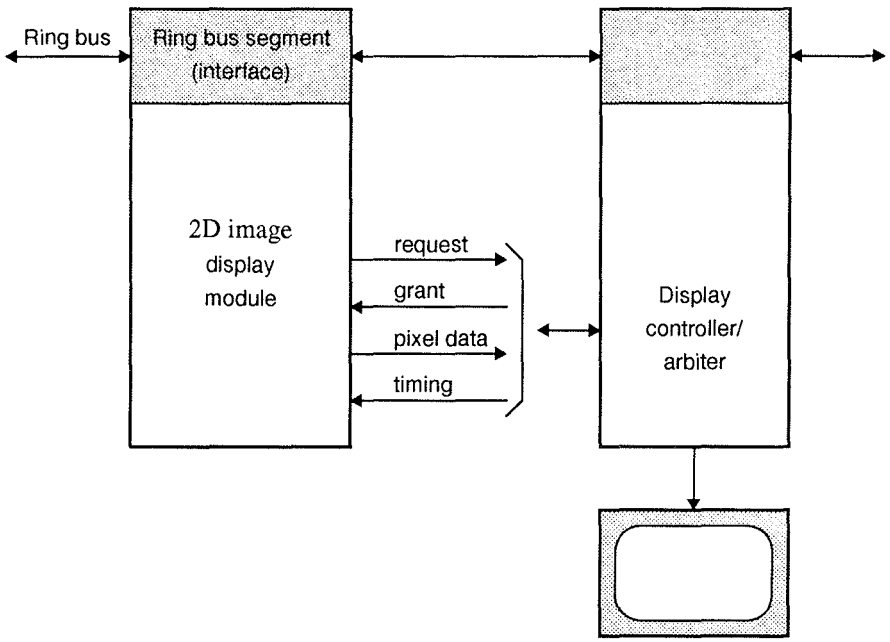


Figure 11.1. Image display system architecture

According to the discussion carried out in section 10.1., the display of the 2D image components must be carried out by a dedicated display module, capable of supporting up to a maximum of 4 simultaneous windows. Traces and other vector and textual image display information are generated and displayed by the display controller/arbiter module. In addition (not shown in Figure 11.1. and not discussed any further), a module for the display of M-mode data is needed.

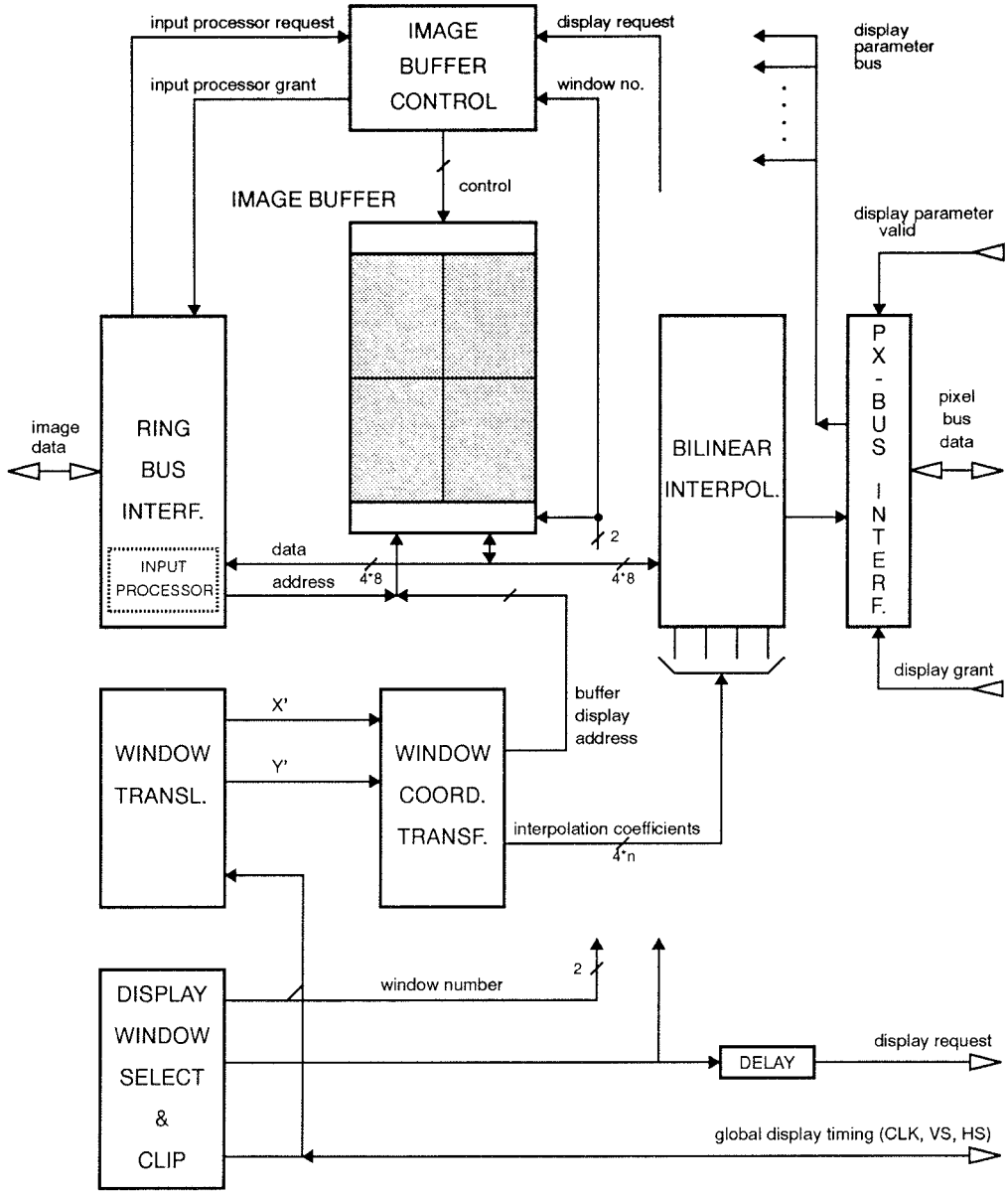


Figure 11.2. 2D display module architecture

The heart of the 2D display module is the *Image Buffer* block. The image buffer contains 4 separate window buffers, to be loaded with 2D ultrasound data via the *Ring Bus Input* block. Access to the Image Buffer is regulated by the *Image Buffer Control* block, ensuring that access conflicts between Ring bus buffer update on one side and display refresh on the other side are resolved. When simultaneous update and refresh requests do occur, the refresh request will always have priority.

For display refresh, the Image Buffer address is set up by the *Window Coordinate Transform* block. The buffer display address is calculated on the basis of the X' and Y' coordinates received from the *Window Translation* block, maintaining the local X and Y coordinates for all 4 windows. Input to the Window Translation block is a 2 bit value identifying the window currently being refreshed. This is computed by the *Display Window Select & Clip* block, on the basis of the global display timing parameters generated by the display controller. Whenever the position of the pixel being refreshed (the refresh position) is within one of the 4 windows, a display request is issued to the display controller module and the Image Buffer Control block. However, to accommodate for the number of cycles it takes to compute the refresh pixel value, the display request issued to the display controller is delayed a number of clock cycles through a *Delay* block compared to the request issued to the Image Buffer Control. The actual length of the delay is determined so that the computed pixel value is ready at the time the display grant signal is received from the display controller. To maintain the windows' absolute position on the screen, the delay between the two requests must be compensated by reducing the windows' horizontal position value accordingly. In this way, the display request will be issued the necessary cycles in advance to have the computed pixel value ready on time. The only practical implication of this is that the windows can not be positioned at the extreme left edge of the screen.

Along with the display request, the 2 bit window number is presented to the Image Buffer Control, causing the 4 samples required to compute the value of the refresh pixel to be output to the *Bilinear Interpolation* block. The required 4 interpolation coefficients are generated by the Window Coordinate Transform block. The coefficients can be regarded as being the fractional part of the result of the coordinate transform computation.

Finally, the refresh pixel value is output to the *Pixel Bus Interface* block, multiplexing two succeeding 8 bit pixels into one 16 bit value. If no other display module requests the Pixel bus with higher priority than the image display, the display controller then issues a display grant signal to the image display and the pixel data is output onto the Pixel bus.

As far as display parameter broadcast is concerned, this is performed on the Pixel bus during the screen vertical retrace period. The presence of valid data is signalled to the display modules by the means of a dedicated "display parameter valid" signal line. On board each module, the parameter data must be distributed by an internal (Display Parameter) bus. The display parameters are transferred as *packets*, containing a header identifying the address of the module(s) to receive the packet as well as the specific type and amount of data contained in the packet. Since the detailed implementation of this scheme will be module as well as application dependent, it is therefore not discussed any further in this document.

We will now take a look at the functional blocks contained in the image display architecture in greater detail. To ease interpretation of the schematics, all blocks implementing an *algorithm* of any kind (processing, table look-up etc.) is shadowed.

11.2. Ring Bus Interface

The Ring Bus Interface as shown in Figure 11.3. is connecting the Ring bus to the Image Buffer. Both directions of transfer are supported, the input of (unprocessed) image data to the Image Buffer as well as the readback of (processed) image data from the Image Buffer to the Ring bus.

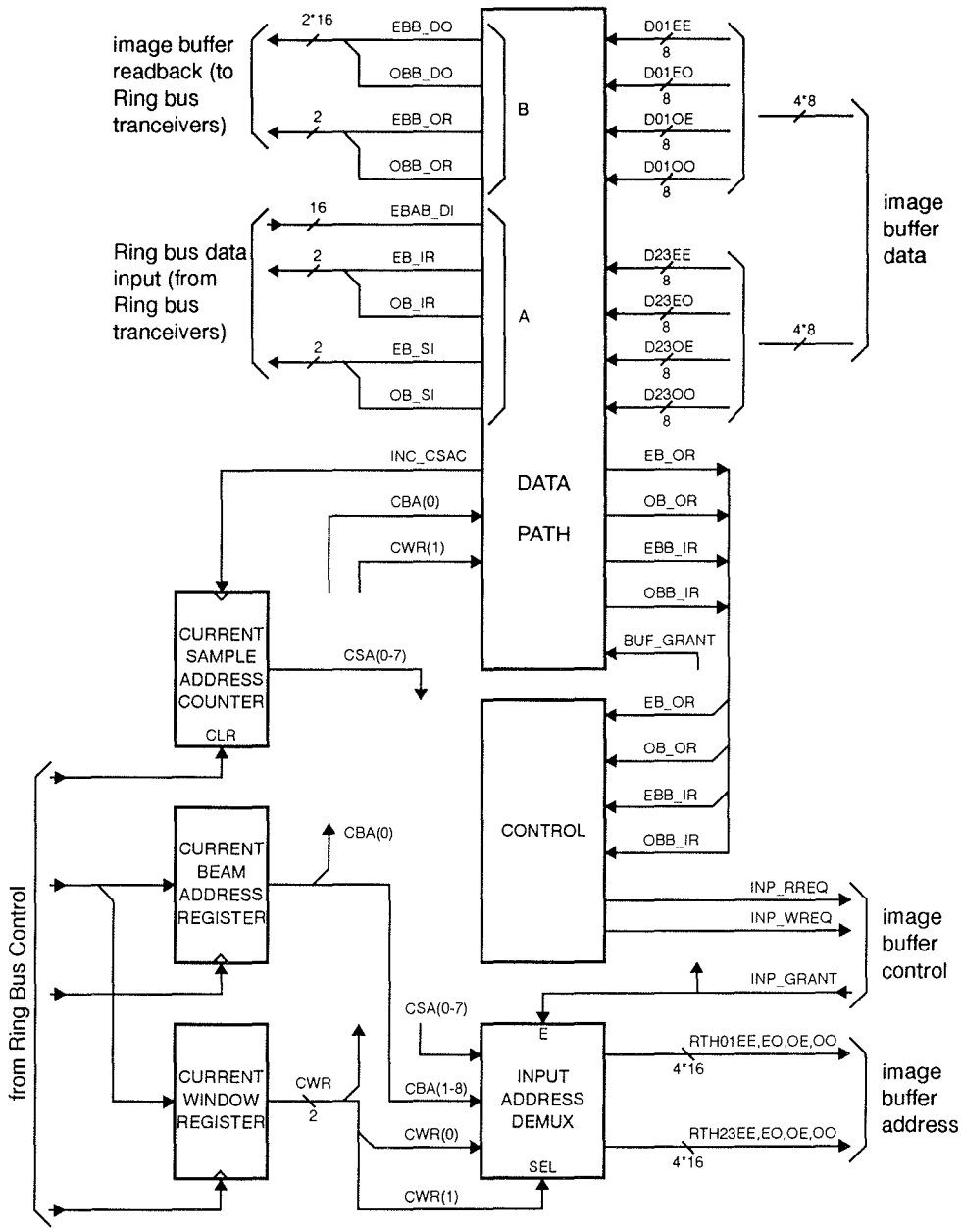


Figure 11.3. Ring Bus Interface

The Ring Bus Interface can logically be divided into two main sections, a data path and an address path. The *Data Path*, essentially containing a number of FIFOs and a large look-up table, will be described in detail on the coming pages and is therefore not discussed any further here.

The function of the *address path* is to provide the necessary addressing while writing data into (or reading data back from) the Image Buffer. As already discussed in detail in chapters 9 and 10, data are transferred on the Ring Bus formatted as packets, each packet containing one or two beams of data. As far as the *sample address* is concerned, it is therefore implicit in a data elements relative position within a packet. A counter being cleared at the beginning of each packet (beam) and incremented for each sample value being read or written will therefore do (*Current Sample Address Counter*).

The *beam address* of the data to be loaded, however, must be contained in the packet header. The module's *Ring Bus Control* block (not described in this document) will decode the packet header and load the beam address information into the *Current Beam Address Register*. With two beams contained in each packet, the Ring Bus Control must also detect the transition between the two beams, upon which the Current Sample Address Counter must be cleared and the Current Beam Address Register incremented. As far as the number of the window in which the data are to be stored (displayed) is concerned, this may either be included as a part of the packet header or stored as a display parameter. In both cases, a two bit register (*Current Window Register*) is necessary, from which the window identity can be distributed to the rest of the module.

Due to the mechanism chosen for sharing access to the Image Buffer, the *Input Processor*, contained in the Data Path block, must issue a request (INP_RREQ (read) or INP_WREQ (write)) whenever it wants to access the Image Buffer. The request is generated by the *Control* block and issued when data are ready to be written into the buffer (EB_OR (Even Beam - Output Ready), OB_OR) or read from the buffer (EBB_IR (Even Beam port B - Input Ready), OBB_IR). Because of the memory chip organization, the two buffer banks, containing windows 0/1 and 2/3, respectively, can be accessed independently. Which bank being requested, 0/1 or 2/3, is signalled through the state of the most significant bit of the Current Window Register, CWR(1). The request is granted (INP_GRANT asserted) on the very first cycle the requested bank is not needed for screen refresh.

A complete buffer address is assembled by the sample (pair) address CSA(0-7), the beam address except the least significant bit (CBA(1-8)), and the least significant bit of the window number (CWR(0)), selecting either the lower (windows 0 or 2) or the upper (windows 1 or 3) half of the memory chip. The address is assembled and multiplexed onto the 4 paths of the appropriate Image Buffer address bus, RTH01 or RTH23, depending on the most significant bit of the window number, CWR(1). The 4 paths of each address bus corresponds to each of the 4 memory chips needed to store the data for one window. The address assembly/ demultiplexing is done by the *Input Address Demux* block. Upon receiving an input grant, the assembled buffer address is driven onto the bus.

11.2.1. Data Path

In addition to the bidirectional port connecting the Data Path block to the Image Buffer, the Data Path block has two other ports, A and B, Through these ports, data input (A) and data readback (B) are performed.

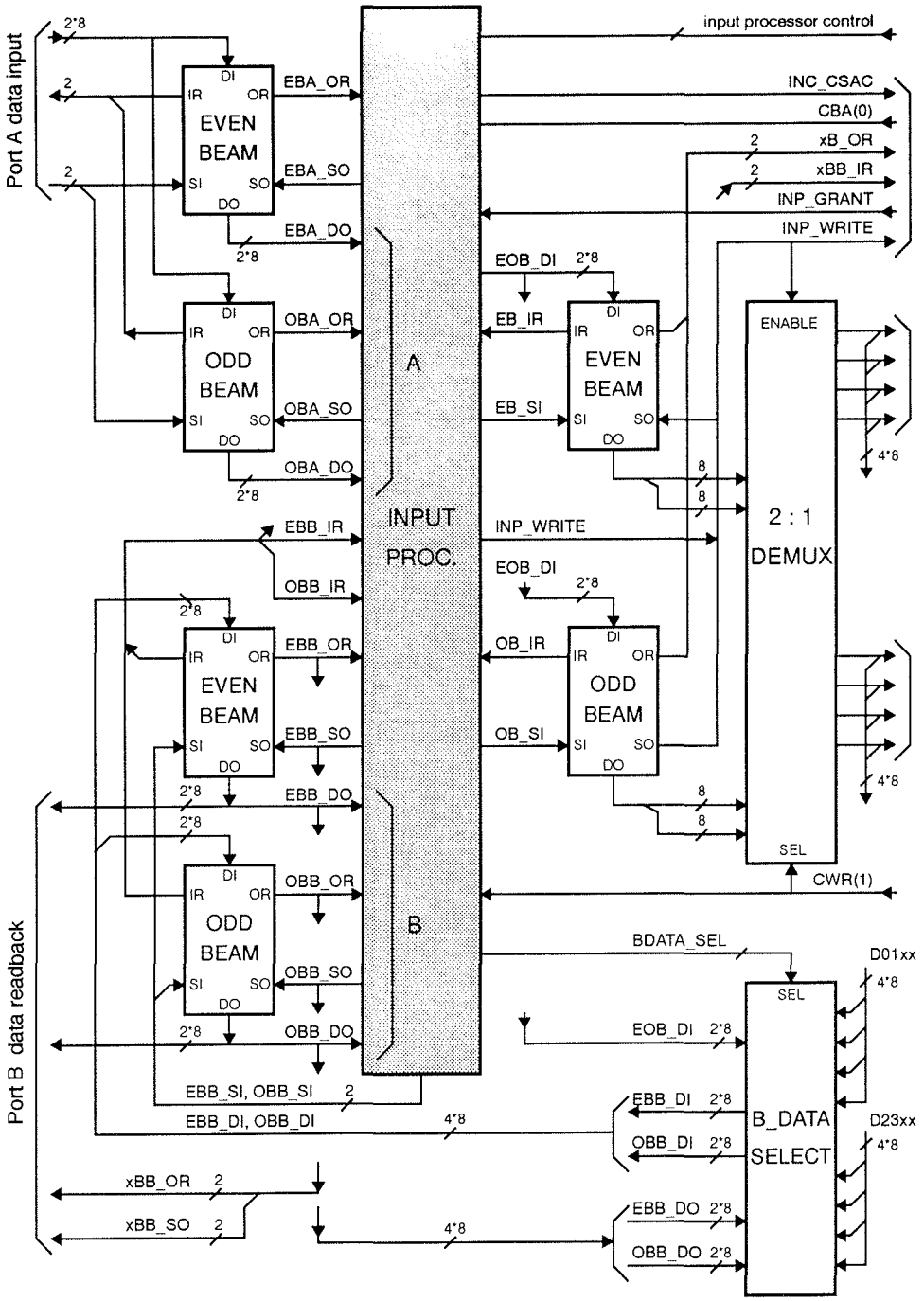


Figure 11.4. Data Path

Port A

By using the A port, image data are transferred from the Ring Bus and into the *Input Processor* block. The port contains two FIFOs, loaded by even beam and odd beam data, respectively. Each FIFO is 16 bits wide to accommodate sample pairs, in accordance with the format by which 8 bit samples are transferred two by two over the Ring Bus. The depth should be at least 256 16 bit values, large enough to contain one full beam of data. The control interface towards the Input Processor is a set of Output Ready (xxx_OR) signals, telling when new data can be read from the FIFO and a set of Shift Out (xxx_SO) signals, causing data to be shifted out of the FIFO. In the same way, the control interface towards the Ring Bus is a set of Input Ready (xxx_IR) and Shift In (xxx_SI) signals.

Port B

While the A port is used for data input to the Input Processor from the Ring Bus, port B handles data readback from the Image Buffer. This may be used for two purposes:

- The readback of processed data from the Image Buffer out on the Ring Bus.
- Making it possible to generate new data to be loaded into the Image Buffer as a function of the data received from the Ring Bus and the (old) data already stored in the Image Buffer.

As far as the first point is concerned, the Image Buffer is normally the last module in the processing chain. However, in principle, it is nothing preventing the input processing features of the image display system to be used as a sort of generic processing facility, allocating one (or more) of the 4 Image Buffer windows as intermediate storage. By manipulating the appropriate display parameters, the display of this (those) window(s) may be turned off, permitting the buffers to be used as general purpose data buffers. After processing, the data can then be read back onto the Ring Bus and transferred to other modules for further treatment. Eventually, the data are transferred back to the display system into *another window* of the Image Buffer to be displayed.

Another actual use of data readback from the Image Buffer is for test purposes. As a general rule, or at least a desired goal of any design, no part of a module should be hidden from inspection through software.

The aspect of modifying the image data with data already stored in the Image Buffer is however the most important, at least seen from an image processing point of view. Due to the use of FIFOS for intermediate storage of the readback data by which the new data are modified, the *location* in the Image Buffer from which data are read back is totally independent of the location into which the (processed) new data are to be loaded. This fact has the following implications:

- By modifying the data with data from the *same window* as well as the *same location* within the window, a time recursive filtering operation is performed. This makes the displayed image less susceptible to sudden variations in the data values (e.g. noise).
- By using data from another window but from the same location, data from two different images can be combined into one new image. One application

of this mode of operation will be to compare the two images, visualizing the result as the difference between them.

- Instead of image data, the other window may be loaded by a full 2-dimensional set of position dependent coefficients.

The source of the readback data is determined by the *B_Data Select* block (selecting window 0/1 or 2/3) in conjunction with the contents of the Current Beam Address Register and the Current Window Register contained in the Ring Bus Interface block.

If the coefficients are constant with respect to the angular (beam) coordinate of the image, it is however not necessary to dedicate an Image Buffer window for storing these coefficients. Instead, the port B FIFOs may be loaded directly from the Ring Bus by propagating the data received on the xBA_DO data lines straight through the Input Processor, out on the EOB_DI lines and then back to the port B FIFO input port on the xBB_DI lines, through the B_Data Select block. During Input Processor operation, the coefficients are then circulated from the port B FIFO output ports back to their input ports via the xBB_DO to xBB_DI path in the B_Data Select block. In this way, the port B FIFOs will contain all coefficients, while the 4 Image Buffer windows are available for image data. Because separate paths are provided for the two port B FIFOs, different sets of coefficients for even and odd beam data are supported.

The port B input port is interfaced to the Input Processor block by the two sets of Input Ready (xBB_IR) and Shift In (xBB_SI) signals, permitting (readback) Image Buffer data to be loaded into the FIFOs under the control of the Input Processor. Coefficient loading is performed in the same way. As far as the port B FIFO output ports are concerned, they will in some cases be controlled by the Ring Bus Interface (data readback to Ring bus) and in some cases the Input Processor (input processing). The corresponding Output Ready (xBB_OR) and Shift Out (xBB_SO) signals must therefore be connected to both the Input Processor and the Ring Bus Interface.

As data are read from the input port FIFOs, they are processed by the Input Processor and written into the output port FIFOs. The process of storing these data into the Image Buffer, however, must due to the access sharing mechanism with screen refresh be granted on an access-by-access basis by the Image Buffer Control. The appropriate buffer request signal is generated by the Control block in the Ring Bus interface: A write request INP_WREQ whenever there is a pending buffer write (xB_OR active), a read request INP_RREQ in case of a pending buffer read (xBB_IR active).

Depending on the value of the Current Beam Address Register's least significant bit (CBA(0)), data from the even beam or odd beam FIFOs are selected for processing. To fully utilize the buffer access bandwidth, however, data are not actually written into the Image Buffer before both output FIFOs (even and odd beam) contains data. Data are therefore always written (and read) 4 samples at a time (odd/ even beam, odd/ even sample).

When the buffer request can be serviced, the INP_GRANT signal is asserted. In case of a buffer write, INP_WRITE then goes active, causing data to be written into the buffer and the output FIFOs to be popped, preparing for the next write operation. By asserting INC_CSAC, the Current Sample Address Counter (CSAC) is then incremented. CSAC is supplying the least significant bits of the complete buffer address.

The data to be written into the Image Buffer must be directed onto the right data bus according to which window (0/1 or 2/3) they are to be written into. This is accomplished by the 2:1 demultiplexer block, controlled by the Current Window Register's most significant bit (CWR(1)).

11.2.1.a. Input Processor

The task of the Input Processor is to transform the received image data according to some defined function before the data are written into the Image Buffer. Due to the FIFOs used on input as well as output, the processing speed of the input processor is not related either to the rate by which the data are received (Ring Bus) or to the rate by which they are to be loaded into the Image Buffer. The implementation of the Input Processor itself can therefore freely be chosen without regards to processing speed. This of coarse, with the implied restriction that the average bandwidth of the Input Processor at least must be large enough to keep up with the rate by which data are entered from the Ring Bus. If not, it would represent a limitation to the overall system performance. In the design presented in this thesis, a lookup table implementation is used, having the advantage of being both simple and fast. More complex input processing algorithms may however require other solutions.

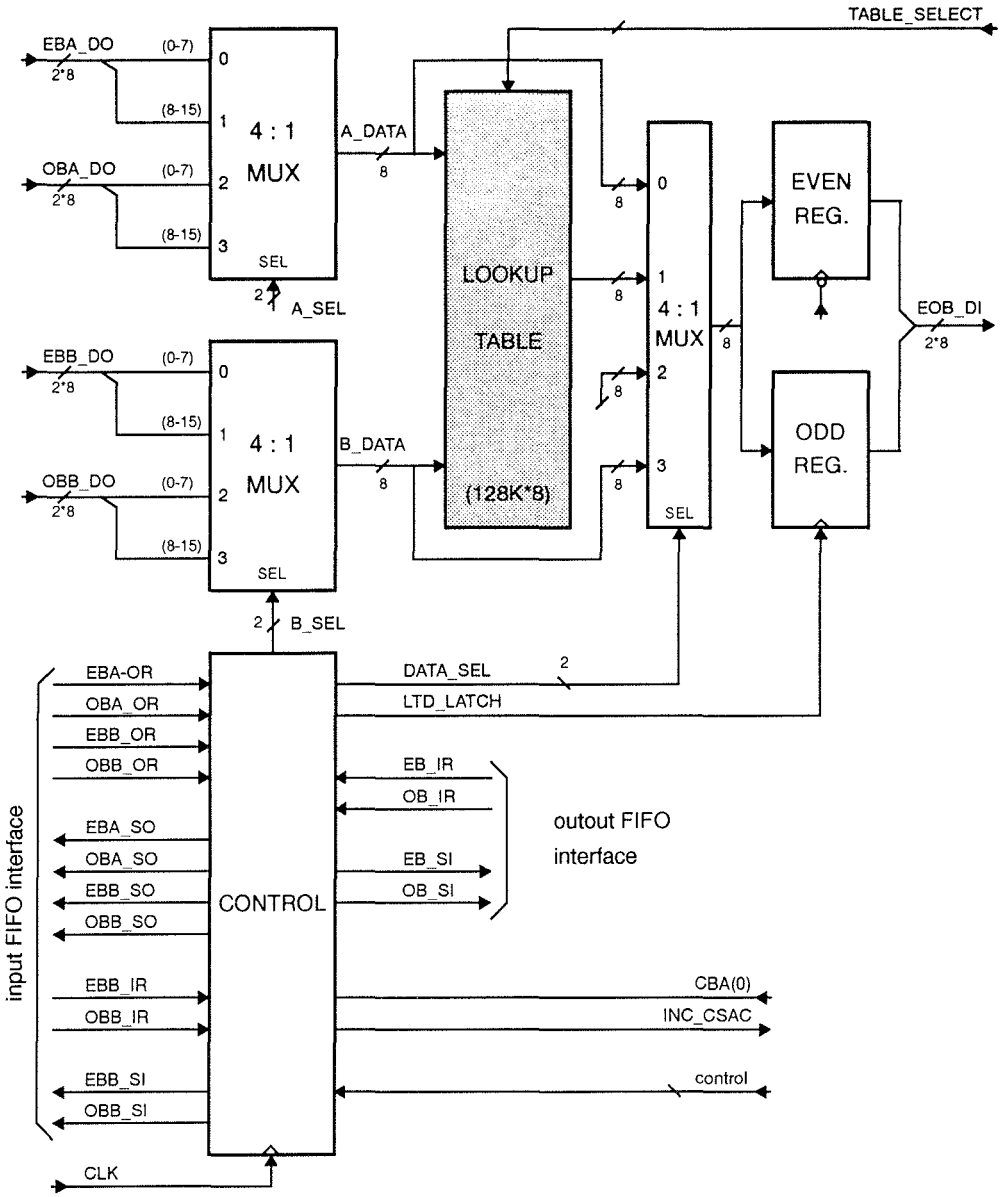


Figure 11.5. Input Processor

As shown in Figure 11.5., the transformation performed by the Input Processor has two input operands: The data value to be transformed (A) and a second value (B). As explained in the previous section, the B value may be

- a multiplication coefficient.

- a data sample from the previous frame in the same image sequence (as to which the A value belongs).
- a data sample from a different image sequence.

Input to each one of the A and B operand 4:1 multiplexers are a complete even/ odd beam, even/ odd sample 4 value set. Which one of the 4 values being used is determined by the A_SEL and B_SEL select signals. The most significant bit of the A_SEL and B_SEL signals, selecting even beam or odd beam data, will be equal to the Current Beam Address least significant bit CBA(0), while the least significant bit of A_SEL/ B_SEL will alternate between logic 0 and 1.

The actual transformation is defined by the contents of the lookup table. By using a larger memory than actually required by the size (i.e. number of bits) of the operands, several table sets can be stored and dynamically selected by the TABLE_SELECT signal(s).

The lookup table design as shown in Figure 11.5. is producing one data value at a time. To form an even/ odd sample pair, the data values must therefore be assembled two by two before they are written into the data path output FIFOs. This is done by the Even and Odd registers, latched on the rising and falling edges of the LTD_LATCH signal, respectively.

Input to the Even and Odd registers are selected by a third 4:1 multiplexer, controlled by the 2 bit DATA_SEL signal. The following options are available:

0	:	The "A" value, making it possible to feed the A value straight through the Input Processor. Via the B_Data Select block in the Data Path, the A value may then be routed back to the B input port FIFOs.
1	:	Lookup table output. This is the normal setting during Input Processor operation.
2	:	All bits grounded. To be used for FIFO and buffer clear operations.
3	:	The "B" value. Actually a spare input, but may be included to make the Input Processor symmetrical with respect to the two operands.

The Input Processor is interfaced to the Data Path input FIFOs through the xBx_OR (Output Ready) and xBx_SO (Shift Out) signals (buffer write operation), and the xBB_IR (Input Ready) and xBB_SI (Shift In) signals (buffer read). On each access to the buffer, the INC_CSAC signal is pulsed to make the Current Sample Address Counter to point to the next location.

11.3. Image Buffer

The Image Buffer is assembled by 8 Static RAM chips as shown in Figure 11.6. Each chip is containing 128 kBytes of data.

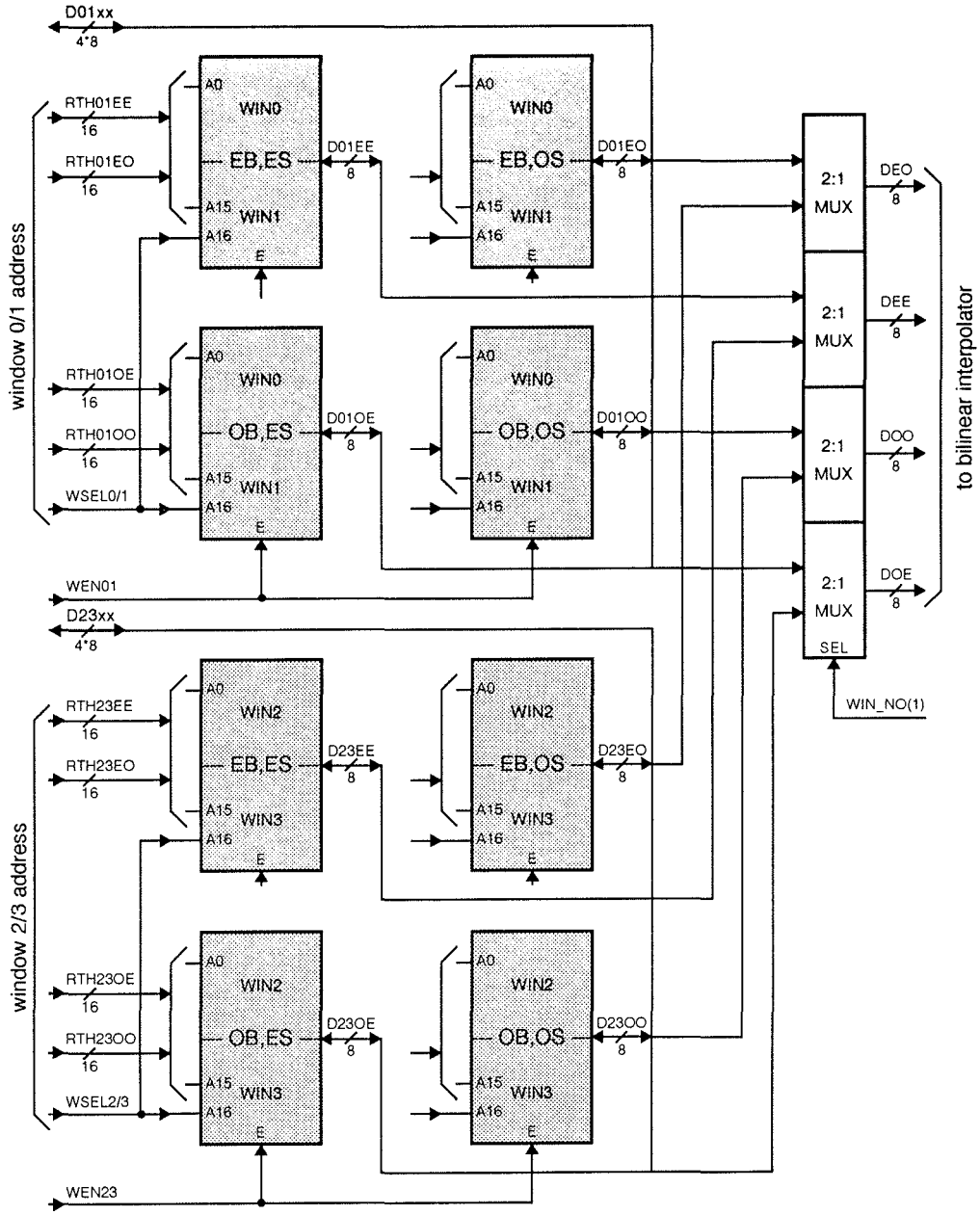


Figure 11.6. Image Buffer

Each of the 4 possible windows is limited to 512 by 512 8 bit sample values. To achieve maximum performance from the bilinear interpolation mechanism, the even/ odd beam, even/ odd sample values are stored in separate memory chips. The maximum amount of data pr.

window pr. chip is therefore 64 kByte, each chip containing data from two windows. As we will see later, the bilinear interpolation mechanism requires that each of the 4 memory chips containing data for the window in question can be independently addressed. Each chip must therefore have its own set of address lines, in the schematics named RTHxxEE, RTHxxEO, RTHxxOE and RTHxxOO. Which one of the two windows contained in each chip being accessed is selected by the most significant chip address line (WSEL0/1 and WSEL2/3, respectively). Needless to say, to support parallel read-out of data from the 4 chips, each chip must also have its own data bus (DxxEE, DxxEO, DxxOE and DxxOO).

To support 4 windows, two such two window banks must therefore be used, each bank containing 4 memory chips as described in the previous paragraph. Each of the two banks, containing data for windows 0/1 and 2/3 respectively, must also have its own set of address and data lines. The are named RTH01xx/D01xx and RTH23xx/D23xx in the schematics. Refreshing the screen with data from a window contained in one bank can therefore take place in parallel with updating one of the windows contained in the other bank. Each bank must therefore have its own output enable signal (WEN01 and WEN23).

The Image Buffer 4 sample data buses D01xx and D23xx are connected to the B_Data Select block contained in the Data Path block (Input Processor access), and the 4 2:1 multiplexers on the Image Buffer output, connecting the Image Buffer to the Bilinear Interpolation block. Output to the Bilinear Interpolation block is selected determined by the most significant of WIN_NO, the 2 bit value identifying the window from which the screen currently is refreshed.

11.4. Image Buffer Control

The task of the Image Buffer Control block is to regulate access to the Image Buffer. Two independent processes are competing for access, using their own set of dedicated control signals as shown in Figure 11.7.

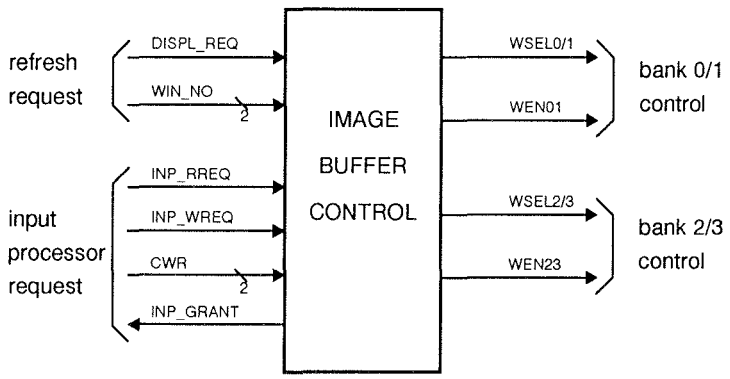


Figure 11.7. Image Buffer Control

Display refresh access (read) is requested by asserting the DISPL_REQ signal. The window (0 to 3) from which the refresh data is going to be read is identified by the two bit WIN_NO signal. Because the refresh process *always* will be granted access to the buffer the next cycle, an image buffer display grant signal is not necessary.

Input Processor access (read or write) is requested through the assertion of the INP_RREQ or INP_WREQ signal, the buffer window being requested is identified by the two bit CWR (Current Window Register) value. As soon as the request can be accommodated (that is, the first cycle when there is no conflict with display refresh), signal INP_GRANT is asserted.

Corresponding to the two types of requests, two pairs of control signals are generated: WSELx/x, selecting one of the two windows contained in each memory chip, and WENxx, connected to the memory chips' output enable. In case of simultaneous Input Processor read and display requests (to different banks), both banks will be enabled.

11.5. Display Window Select & Clip

Before describing the implemented mechanism for window selection and clipping in detail, we will first go through the basic philosophy behind the design:

11.5.1. The basic philosophy

Within the framework of the rest of the display system, the design goal for the window selection and clipping mechanism was to come up with a solution having the following features:

- To support 4 independent, possibly overlapping windows.
- No restrictions on window position and size other than the maximum size of 512 by 512 pixels.
- A change in a window's position and/or priority (depth) should be synchronized to and effective on the next refresh frame.
- The amount of hardware required should be minimized.

The latter two points requires that the amount of data necessary to process and/ or move in conjunction with a window operation is kept to a minimum. This prohibits the use of a complete map of the screen, for each pixel showing which window it belongs to. Instead, the windows must be parameterized.

To describe a window's position and size, four parameters are needed: The position of the upper left corner (Xmin, Ymin) and the position of the lower right corner (Xmax, Ymax). An example showing the position parameters of two non-overlapping windows A and B is shown in Figure 11.8.

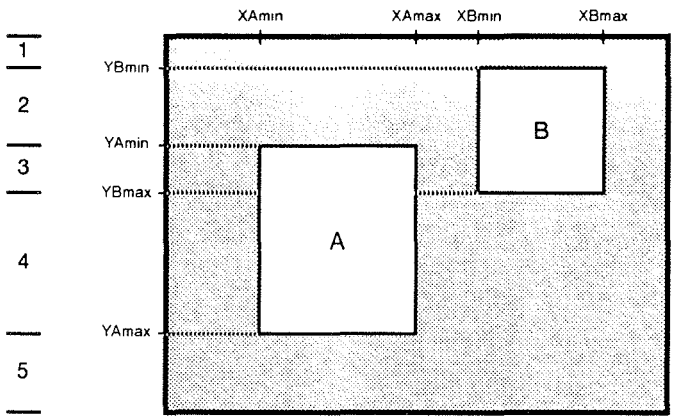


Figure 11.8. Window position parameters

According to their vertical position, the screen area from top to bottom can be divided into 5 parts:

Y > YBmin(1)

No window active.

YBmin > Y > YAmin(2)

Window B active between XBmin and XBmax.

YAmin > Y > YBmax(3)

Both windows active: Window A between XAmin and XAmax, window B between XBmin and XBmax.

YBmax > Y > YAmax(4)

Window A active between XAmin and XAmax.

Y > YAmax(5)

No windows active.

If a refresh position monitor was set up to continuously compare the position of the pixel currently being refreshed against the window parameters, its output signal would look something like the diagrams shown in Figure 11.9. for the 5 different parts of the screen area.

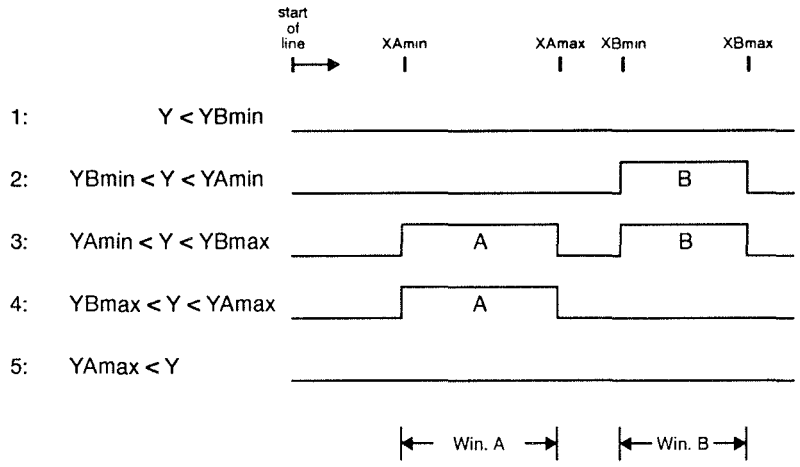


Figure 11.9. Refresh position monitor output

Depending on the output of the refresh position monitor,

- a display request to the display controller module would be generated whenever the refresh position was inside *any* of the defined windows.
- the output of the position monitor could directly be used as a pointer into the Image Buffer to select the window currently being refreshed.

A more comprehensive example is illustrated in Figure 11.10., showing 4 differently sized, non-overlapping, partly overlapping and fully overlapping windows. Some of the window parameters are common to more than one window (e.g. y=375 is common to windows 0, 1 and 3).

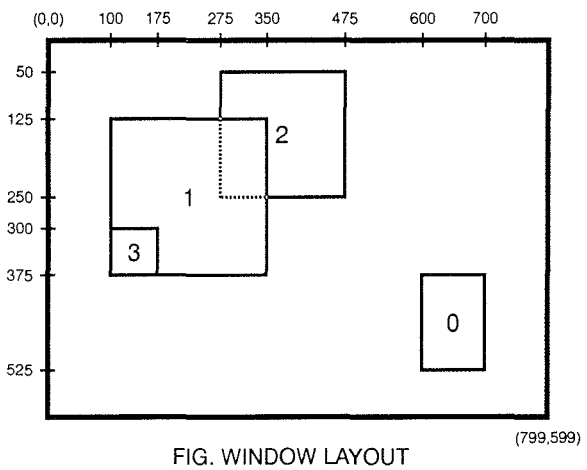


Figure 11.10. 4 window screen layout

As far as systemizing the window parameters into a table is concerned, there must be one entry for each row having a different set of column parameters compared to the preceding row. With 4 possible windows, and 2 entries for each window (top and bottom), the table will have a maximum of 8 row entries. The maximum number will be used if all windows have their horizontal (top and bottom) lines at different y- positions. This table, containing the numbers of the rows on which a window is entered or left, is called the *Row Table.*

Correspondingly, each row entry will need a maximum of 8 column entries, to accommodate the positions of the left and right edges of 4 windows. In total, an 8 by 8 entry table is needed to store the diagonal corner coordinates of all 4 windows. This table is called the *Column Table.* The screen row number corresponding to each row in the Column Table is stored in the Row Table.

Each table entry, in the Row Table as well as in the Column Table, is a value *pair.* The first element of each pair contains the (row or column) position, the second which window(s) are involved. Because more than one window may be positioned at the same coordinate, the window identity can not be decoded into a single 2 bit value but must be represented by a 4 bit mask "xxxx", with each bit position 3 to 0 corresponding to the window with the same number. Each of the 4 bits will have the following interpretation:

0 - Screen position is outside window "n".

1 - Screen position is inside window "n".

As far as the position elements of the table entries are concerned, they may be given as either absolute or relative values. If the positions are expressed as absolute screen position values, a counter/ register/ comparator mechanism is required to detect when the refresh position is equal to a window position as specified in the Row or Column table.

By reloading the counter when detecting a window edge with the distance to the next window edge, the comparators may be omitted. By counting the counter down, all it takes is to detect when it reaches zero. The positions must then be given as *incremental values*, relative to the previous window edge. This approach requires less hardware than the absolute value method, and it is therefore preferred in this design. The Column and Row tables for the screen layout from Figure 11.10. is shown in Figure 11.11. Because several windows have one or more edges in common, the tables are only partly filled.

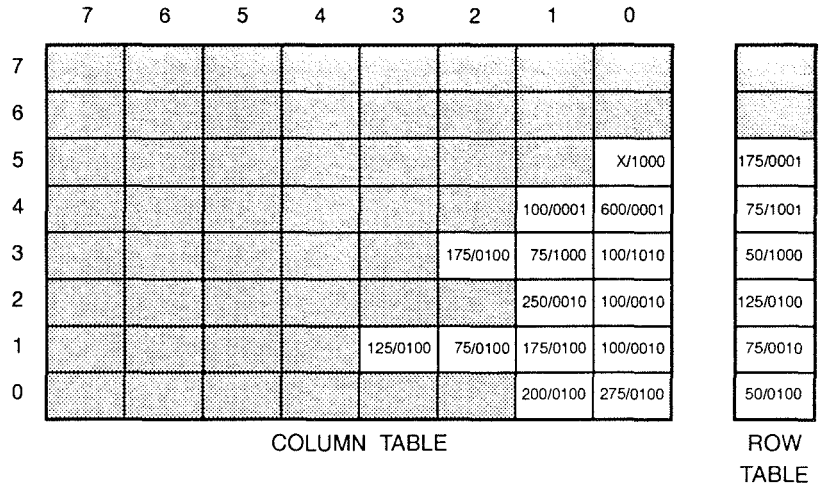


Figure 11.11. Column and Row tables

In addition to the tables defining the windows' position and size, a table defining their mutual display *priority* is needed. Because the windows are allowed to be overlapping, a window may be partially or fully covered by other windows. This situation is resolved through the Priority Table, containing one entry for each possible window combination. To support 4 windows, 16 entries are therefore needed. In accordance with the display layout as shown in Figure 11.10., the window priorities are window 3 (highest), 1, 0 and 2 (lowest). The contents of the Priority Table for this set of priorities is shown in Figure 11.12.

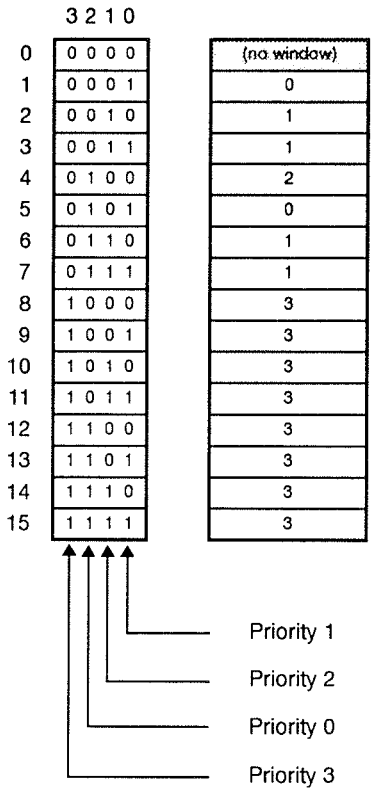


Figure 11.12. Priority Table

After this introduction to the basic philosophy behind the display window select & clip mechanism, we will now take a look at a possible hardware implementation of the scheme:

11.5.2. Display Window Select & Clip

The select and clip mechanism is built around the (Column, Row and Priority tables, together with the necessary electronics to support them (Figure 11.13.).

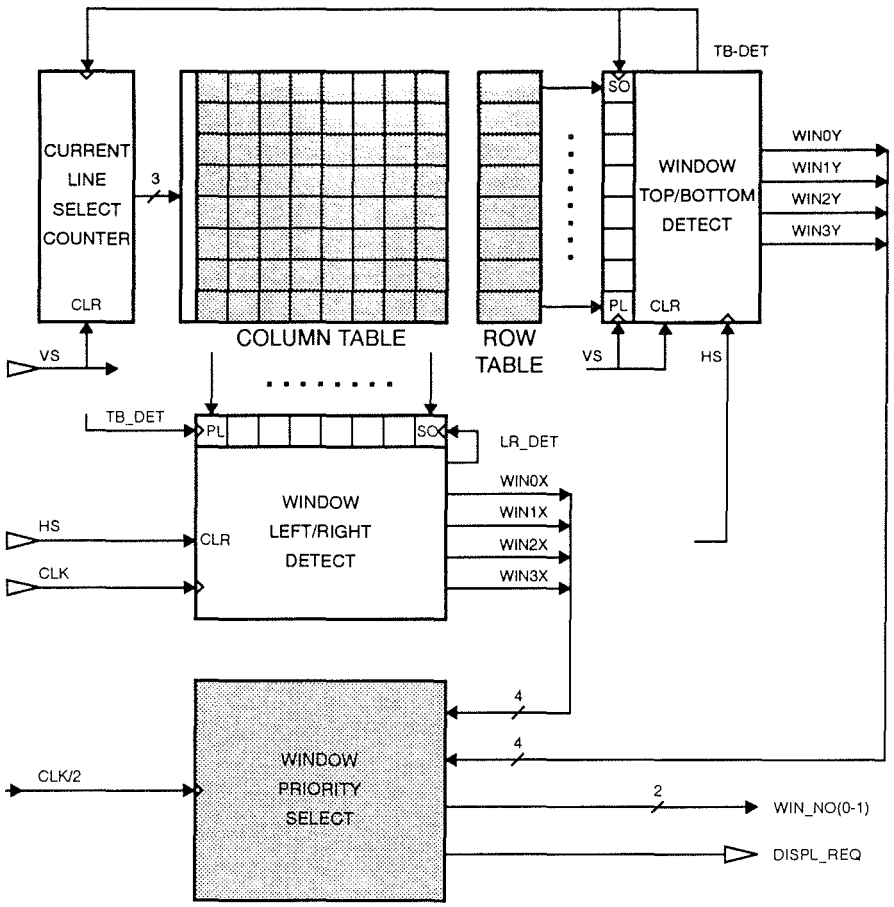


Figure 11.13. Display Window Select & Clip

Theory of operation

As already described, the Column and Row tables will consist of sequences of incremental values, each value giving the distance from one edge to the next. The distance is either measured as a number of scan lines (Row Table) or as a number of pixels along a scan line (Column Table). The two tables contain the same number of entries, with each entry being a single value in the Row Table and a vector in the Column Table.

The select & clip mechanism is resat by the screen vertical sync (VS) signal. The entire Row Table is then loaded into an 8 element shift register in the Window Top/ Bottom Detect block (WTBD), and the pointer to the current entry of the Column Table (Current Line Select Counter) is cleared, making the first entry of the Column Table current. The WTBD counter is loaded with the first entry of the Row Table, and for each scan line, the counter is decremented by the screen horizontal sync signal HS. When the counter eventually runs out, the screen

refresh has reached the top edge of the upmost window. The WTBD block signals this event by asserting the TB_DET signal (Top/Bottom-DETected). Through the shift register mechanism, the WTBD counter is then loaded with the next entry of the Row Table.

By the asserted TB_DET signal, the Window Left/ Right Detect block (WLRD) is loaded by the first entry (vector) of the Column Table, and the WLRD counter by the first value in that vector. This value contains the distance (number of pixels) from the screens left edge to the first windows left edge. The WLRD counter is counted down by the pixel clock CLK, and when it expires, it means that the vertical (left or right) edge of a window is reached. The signal LR_DET (Left/ Right Detected) is then asserted, causing the WLRD shift register to load the next value contained in the same Column Table vector into the counter.

Each time a counter (WTBD or WLRD) expires, the corresponding 4 window enable signals (WINiY or WINiX) are set equal to the mask value accompanying the value by which the counter was loaded. The position of the current *scan line* (that is, the scan line currently being refreshed) vs. the vertical positions of the 4 windows 0 to 3 are reflected through the state of the signal lines WINiY, generated by the Window Top/ Bottom Detect block (i = 0-3):

0 : The (vertical) position of the current scan line is outside window *i*, that is either above its top edge or below its bottom edge.

1 : The (vertical) position of the current scan line is inside window *i*, that is between its top and bottom edge.

In the same way, the Window Left/Right Detect block WLRD signals whether the *pixel* currently being refreshed is lying between any of the 4 windows' side (left and right) edges. This is reflected through the state of the signal lines WINiX (i = 0-3):

0 : The (horizontal) position of the current pixel is outside window *i*, that is either to the left of its left edge or to the right of its right edge.

1 : The (horizontal) position of the current pixel is inside window *i*, that is between its left and right edge.

By combining the corresponding WINiX and WINiY signals, it can then be determined whether the refresh pixel position is inside window *i* or not. This is done by the Window Priority Select block. If the refresh pixel position is inside *any* of the defined windows, the DISPL_REQ signal is asserted, signalling to the display controller that the Image Buffer module wants to supply the refresh data for this pixel. The identity of the window to supply the data is to be coded into the two bit value WIN_NO. If more than one WINi signal pair (i=0-3) is active simultaneously, there is overlapping windows at this position. The conflict is resolved by the Priority Table contained in the Window Priority Select block, presenting the winning window's identity on the WIN_NO lines.

As shown in Figure 11.13., the select & clip mechanism consists of three major components: The two detect blocks and the Window Priority Select block.

11.5.3. Window Top/Bottom Detect

The Window Top/Bottom Detect block (WTBD) consists of an 8 element shift register, a down counter and 4 toggle flip-flops as shown in Figure 11.14.

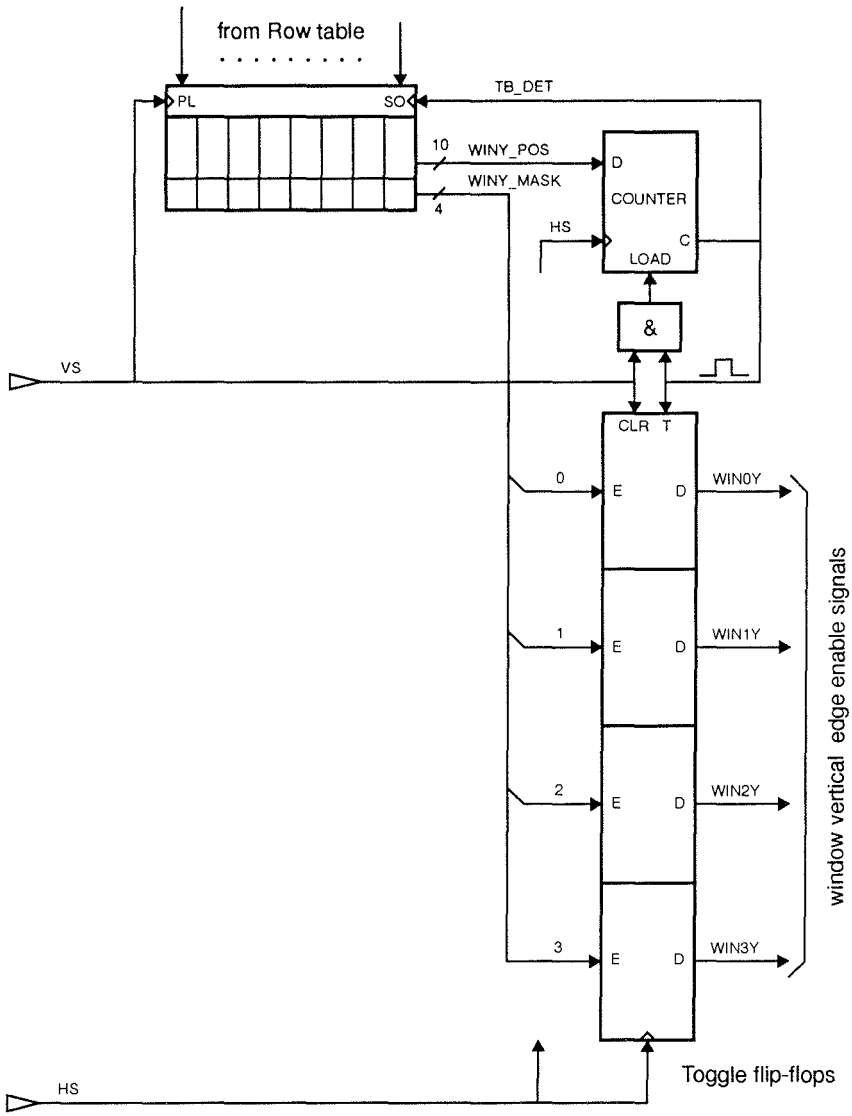


Figure 11.14. Window Top/Bottom Detect

On vertical sync (VS), the entire Row Table is loaded into the shift register, the flip-flops are cleared and the first value of the Row Table is loaded into the counter via the 10 bit WINY_POS lines. On every horizontal sync (HS), the counter is decremented. This goes on until the counter reaches zero. A high pulse is then generated on the counter's carry (C) output (signal TB_DET),

connected to the flip-flops' toggle (T) input. All flip-flops enabled by its corresponding WINY_MASK bit, contained in the Row Table, are then toggled. This causes the flip-flop outputs (D), clocked by the horizontal sync signal HS, to switch to a logic 1 (inside window) or logic 0 (outside window), depending on their previous state.

Signal TB_DET is also connected to the shift output (SO) terminal of the shift register, causing the next WINY_POS/ WINY_MASK value pair to appear on the shift register output. The counter is then loaded with the WINY_POS value, while the WINY_MASK will enable the flip-flop toggling the next time the counter expires.

The flip-flop outputs will all be equal to a logic 0 at the top of the screen (initial condition, cleared by VS) and at the bottom of the screen (provided that the Row Table has been correctly programmed).

11.5.4. Window Left/Right Detect

The Window Left/Right Detect block (WLRD) is in construction identical to the Window Top/ Bottom Detect block.

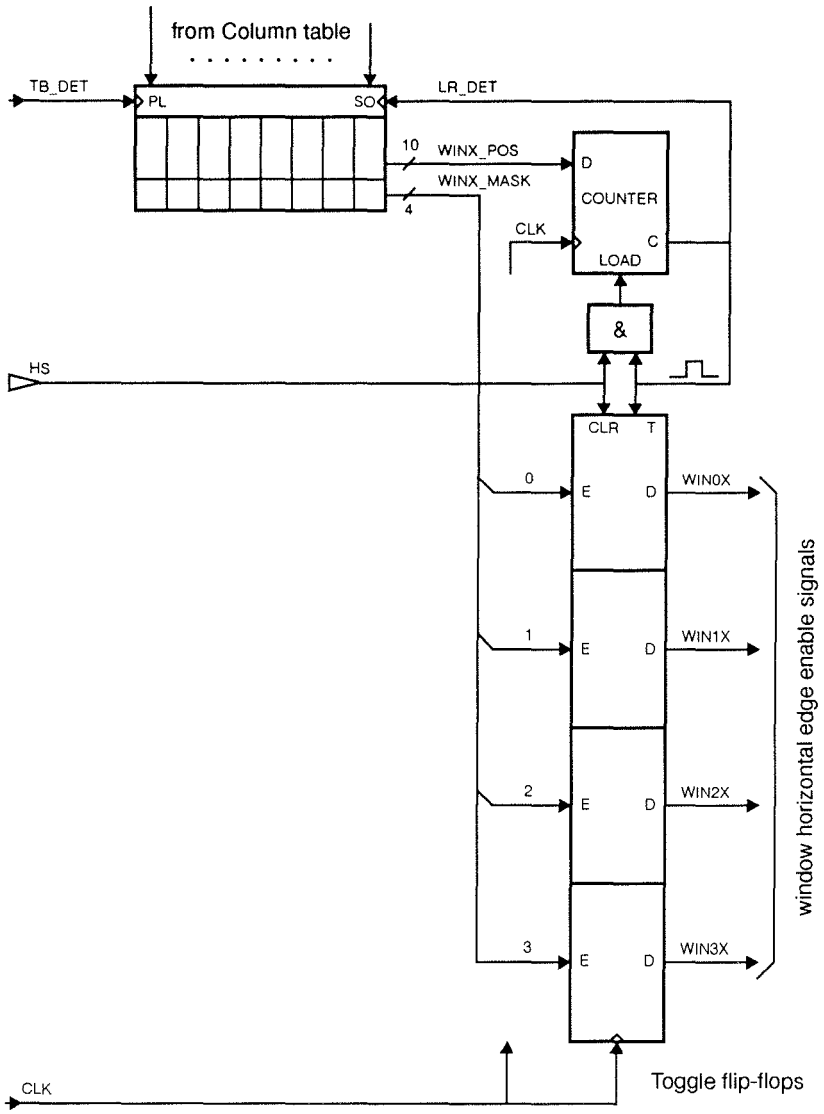


Figure 11.15. Window Left/Right Detect

The signals by which the block is controlled is however different: Each vector entry contained in the Column Table describes the screen layout along a *scan line*. The counter and the 4 toggle flip-flops must therefore be *cleared* by the horizontal sync (HS) signal, *clocked* by the pixel clock CLK signal. Every time the WLRD counter runs out, signal LR_DET (Left Right_Detected) is asserted and the counter is loaded with the next value of the vector. Before starting to refresh the next scan line, signal HS is asserted and the WLRD counter is reloaded with the first value of the *same* vector.

When a new window top or bottom edge is encountered, the WTBD counter runs out, signal TB_DET (Top Bottom_Detected) is asserted and the shift register is loaded with the *next* vector from the column table. On the next horizontal sync (HS) signal, the first value of this vector is loaded into the WLRD counter.

11.5.5. Window Select

The task of the Window Select block is, on the basis of the 4 pairs of window enable signals generated by the WTBD and the WLRD detect blocks, to select *one* of possible several enabled windows for display.

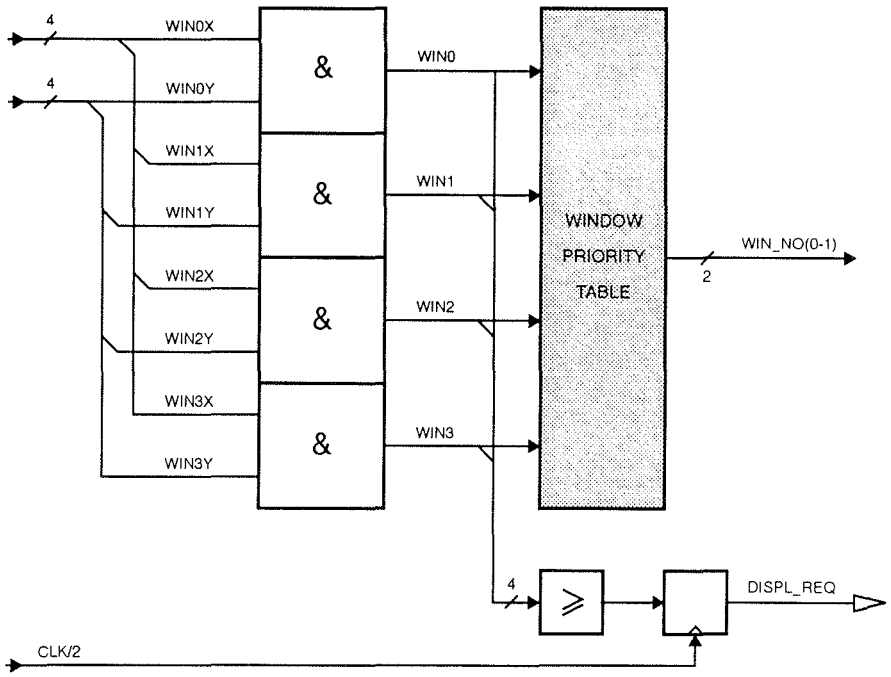


Figure 11.16. Window Select

The windows' mutual priority is defined in the Window Priority Table, addressed by the 4 composite window enable signals WIN0-WIN3. WIN0-WIN3 are generated by 4 dual-input AND-gates, combining the 4 pairs of window enable signals WINiX, WINiY (i=0-3) produced by the detect blocks.

If at least one window is enabled, a display request (DISPL_REQ) is issued to the display controller. The DISPL_REQ signal is generated by the 4-input OR-gate/ register combination. Because the pixel values are transferred two by two on the Pixel bus, the resolution of the display arbitration is equal to two pixels. The DISPL_REQ register must therefore be clocked by half the pixel clock, CLK/2.

11.6. Window Translation

The Window Translation block consists of 4 pairs of counters, one X- and one Y-counter for each window. The X-counters are cleared on the horizontal sync signal HS and clocked by the pixel clock signal CLK. The Y-counters are cleared on vertical sync VS and clocked by horizontal sync HS.

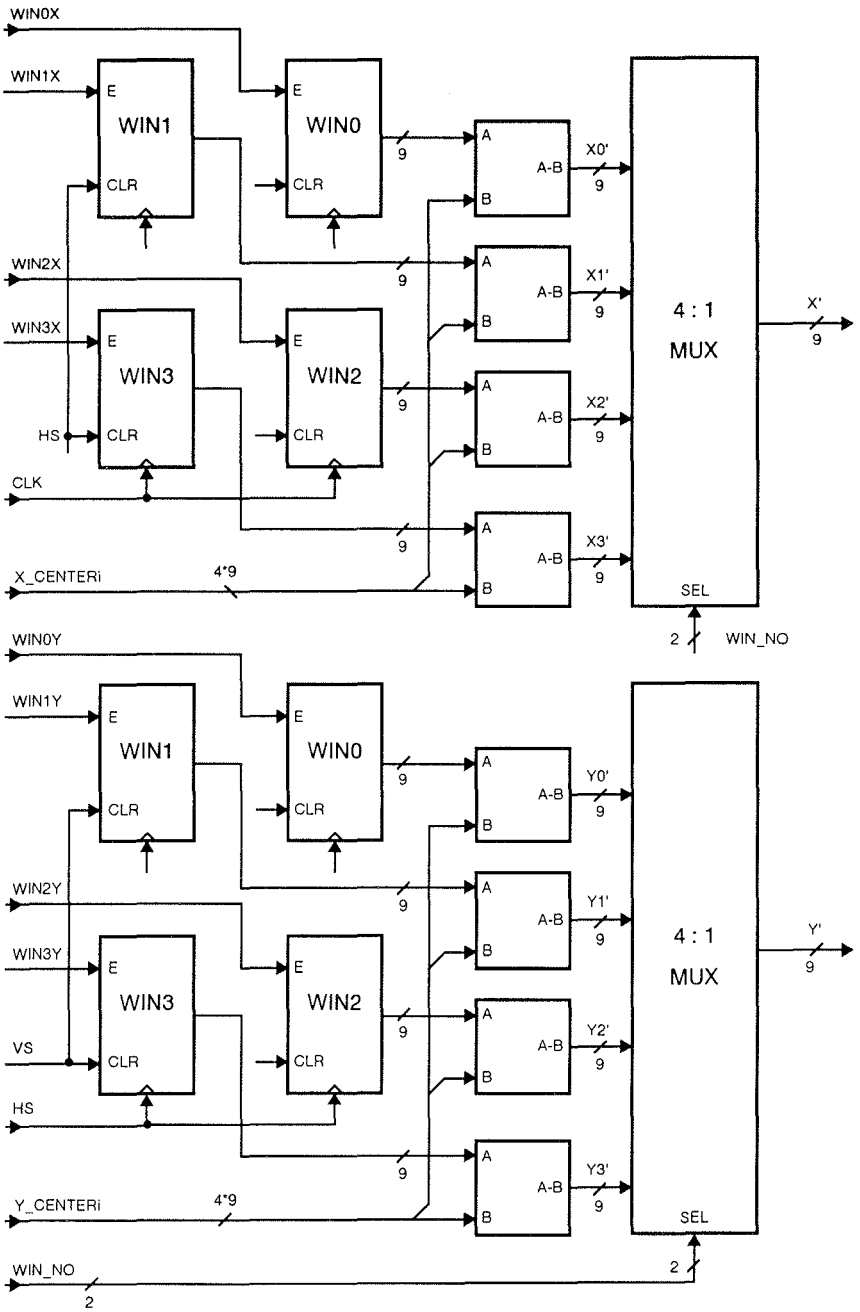


Figure 11.17. Window Translation

Each counter is enabled by the corresponding window enable signal WINiX or WINiY (i=0-3), generated by the Window Left/ Right Detect block or the Window Top/ Bottom Detect block, respectively. The 9 bit output values of a counter pair will therefore at any time during screen refresh be equal to the

> *X/Y position of the refresh pixel position, measured in the window's local coordinate system with the window's upper left corner as origo.*

However, the rectangular coordinates to be input to the Coordinate Transform block, performing the rectangular to polar conversion, must be given relative to the *centre of the circle sector*, not to the position of the window itself.

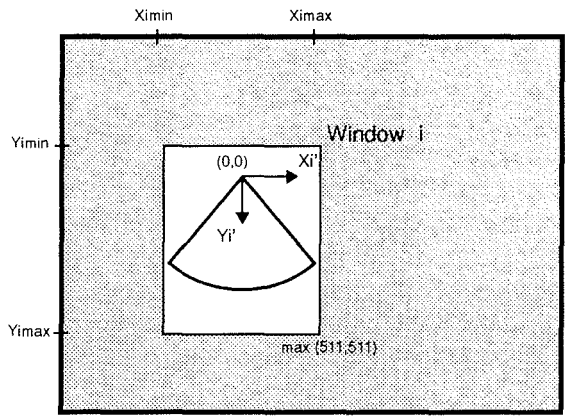


Figure 11.18. Coordinate system origo

Before output to the Coordinate Transform, the coordinate values of the image (sector) centre must therefore be subtracted from the values output by the counters. To be able to display the image at any position within the window, the subtraction is done be a subtracter (one for each coordinate, giving a total of 8 subtracters) instead of coding a limited number of selected image positions into a look-up table. The windows' origo coordinates (X_CENTERi and Y_CENTERi) are either stored in a register integrated with the B port of the subtracter or supplied on dedicated lines as shown in Figure 11.17. Output from the subtracters are the 4 coordinate pairs Xi'/Yi', giving the refresh pixel position relative to the image display position (sector centre) in each of the 4 windows.

At any pixel position on the screen, only one window at a time can be displayed. The Coordinate Transform block may therefore be shared between the four windows. The window currently being displayed is identified through the 2 bit value WIN_NO, generated on the basis of the contents of the Window Priority Table, contained in the Window Select block. WIN_NO selects one of the 4 coordinate pairs Xi'/Yi' through the two 4:1 multiplexers, producing the X'/Y' final output of the Window Translation block.

11.7. Window Coordinate Transform

The function of the Window Coordinate Transform block is to map the rectangular coordinate pair X'/Y' into a (polar) beam/ sample address pair directly to be used for addressing the data contained in the Image Buffer.

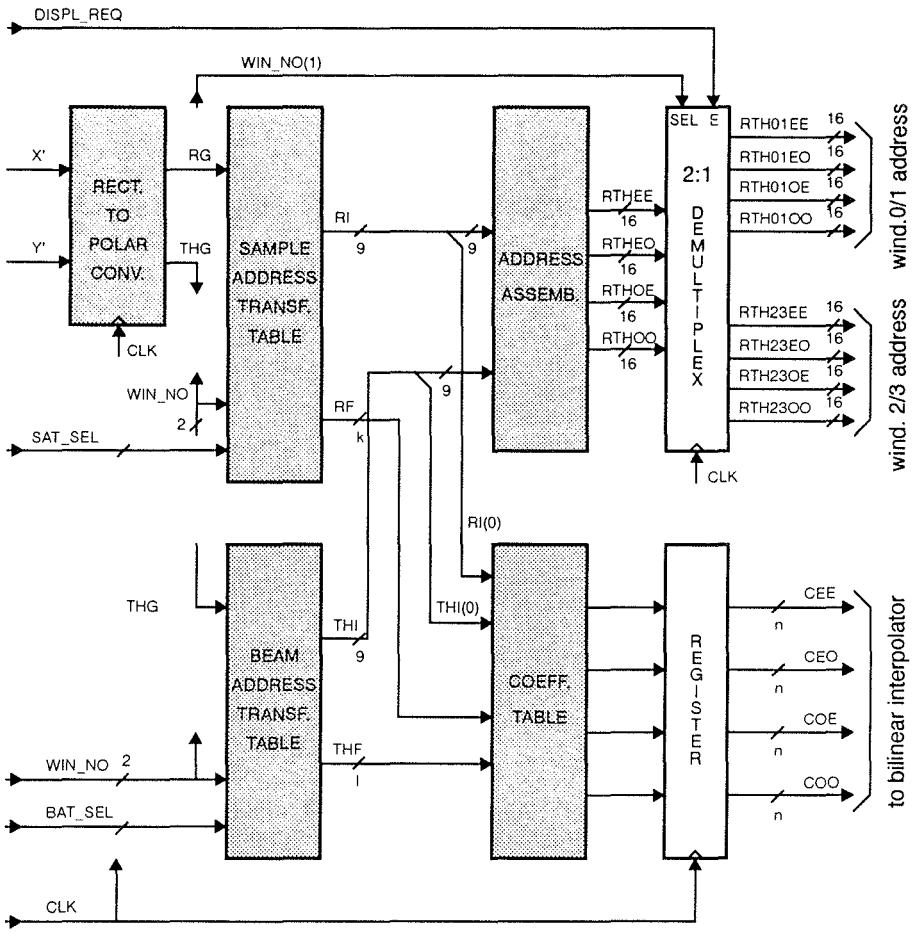


Figure 11.19. Coordinate Transform

The first step in this process is, by applying pure geometric conversion formulas, to map the rectangular coordinates X'/Y' into a corresponding set of bipolar values RG and THG:

```
RG  =   SQRT(X'**2 + Y'**2)
THG =   ARCTAN(Y'/X')
```

From the geometrical RG/THG coordinates the corresponding beam and sample address of the pixel to be refreshed can be computed. In that computation, the format of the image data must also be considered: The number of beams and the value of the sector angle as far as the beam address is concerned, and the number of samples and the value of the sector depth to compute a specific sample's address. Irrespective of the total number of beams and samples an image contains, it is stored in the Image Buffer within the corresponding window's local address space from location (0,0) as shown in Figure 11.20.

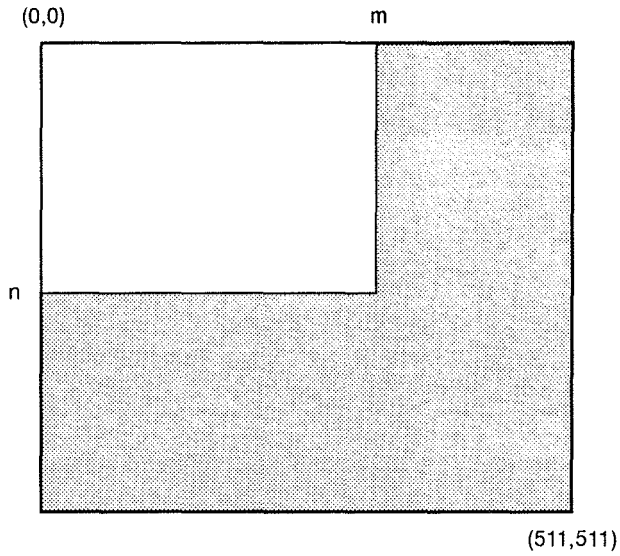


Figure 11.20. Image data storage

This mapping is done by the Beam Address Transformation Table and the Sample Address Transformation Table, respectively. Sector scaling is done through the contents of the transformation tables. Each window must have its own set of tables, the 2 bit window identity WIN_NO is used to select the particular table set.

To obtain a fastest possible response to image (re)scaling, requested by the system operator, it would be advantageous if transformation tables corresponding to several scale factors could be (pre)computed and stored in the table. When the operator requests image rescaling, the transformation tables corresponding to the scaling factors next to the current choice can be computed. Then, when the operator issues the next scaling request, the appropriate transformation table is (hopefully) already computed and stored. It can thereby be made effective simply by altering the value on the table select lines, SAT_SEL (Sample Address Transformation table - SELect) or BAT_SEL (Beam Address ..) for the sample and beam address transformation tables, respectively. To avoid transferring large amounts of table data over the Ring bus (or alternatively the Pixel bus), the Image Buffer module should include a processor capable of computing the necessary table data. If necessary to achieve the required update bandwidth, the transformation tables must be implemented by a double buffering scheme.

The output values of the two transformation tables can each be regarded as consisting of an integer part (RI, THI) and a fractional part (RF, THF). For better to explain the coordinate transform process, we will first take another look on the figure showing the principle of bilinear interpolation:

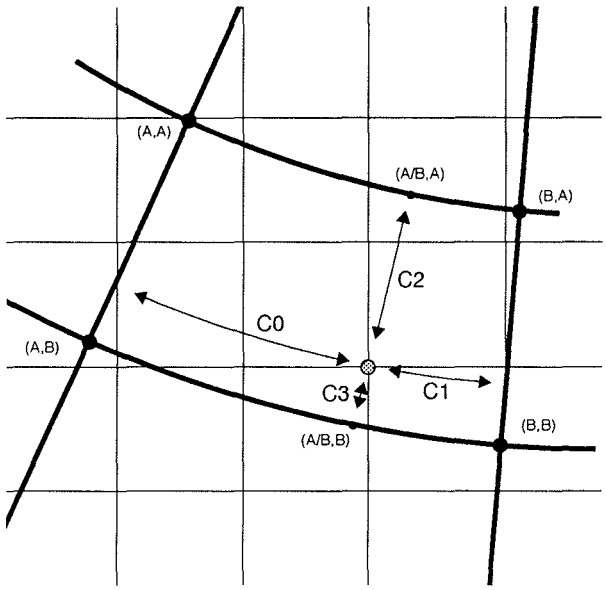


Figure 11.21. Bilinear interpolation principle

The **integer part** RI and THI is equal to

> *the sample and beam address of the image sample located immediately above and to the left of the position of the pixel to be computed (refreshed),*

indicated by a shadowed circle. This assumes an addressing scheme with beam 0 along the left edge and sample 0 at the top (centre) of the sector. The output of the transformation tables will then be equal to the address of sample (A,A) in Figure 11.21. To address a maximum number of 512 samples and/ or 512 beams, RI and THI must both be 9 bits values.

If the beam number of sample (A,A) is denoted **b**, and the sample number (within beam b) **s**, the address of the 3 other samples can be computed by adding 1 to the sample and/ or beam address of sample (A,A):

```
adr (A,A) = (b,s)
adr (A,B) = (b,s+1)
adr (B,A) = (b+1,s)
adr (B,B) = (b+1,s+1)
```

One of these addresses will be an even beam/ even sample (EE) address, one an even beam/ odd sample (EO) address, one an odd beam/ even sample (OE) address and one an odd beam/ odd sample (OO) address, corresponding to the partitioning of the image data over the 4 memory chips in the Image Buffer. But, which one of the 4 samples which is the EE sample, the EO sample, the OE sample and the OO sample will change as the position of the pixel being refreshed moves. With (A,A) being the even beam/ even sample address, the beam and sample addresses of the 4 samples will only differ in their least significant bit. In other cases, (A,A) is the odd beam/ odd sample address. Depending on the particular **b** and **s** values, the even and odd addresses may then differ in a number of (maximum all) bits. Therefore,

> *the EE, EO, OE and OO memory chips each need their own address path for sample addressing during bilinear interpolation.*

In the schematics, the four address paths are labelled RTHxxEE, RTHxxEO, RTHxxOE and RTHxxOO.

Each of the 4 sample addresses are output onto the appropriate address path as determined by the value of the least significant beam and sample address bits. The values on each of the 4 paths, being a complete 16 bit sample address, are put together according to the following algorithm:

```
RAA = RI;      THAA = THI
RAB = RI + 1; THAB = THI
RBA = RI;      THBA = THI + 1
RBB = RI + 1; THBA = THI + 1

for (xx = AA, AB, BA, BB) do
{
    if (THxx's LSB == 0)
        if (Rxx's LSB == 0)
            RTHEE = compute_RTH(Rxx, THxx)# EE
        else
            RTHEO = compute_RTH(Rxx, THxx)# EO
    else
        if (Rxx's LSB == 0)
            RTHOE = compute_RTH(Rxx, THxx)# OE
        else
            RTHOO = compute_RTH(Rxx, THxx)# OO
}

compute_RTH(R,TH)
{
    return (ISHFT(R,-1) + ISHFT((TH & 0x1fe),8))
}
```

The complete, composite 16 bits address is thereby assembled by taking the 8 most significant bits (out of 9) of the sample (R) and beam (TH) address. The sample part is put into the least significant byte of the 16 bit address while the beam part is put into the most significant byte.

As will be described in detail earlier, the Image Buffer is organized as two separate banks, containing data for windows 0/1 and 2/3, respectively. To allow that the update of one bank can be performed in parallel with refresh data being read from the other, each bank have its own set of address (and data) lines, labelled "01" and "23". The four composite address paths RTHxx supplying the refresh address must be directed onto the appropriate set of address lines RTH01xx or RTH23xx, depending on from which window (buffer) the screen is refreshed. The refresh window is identified by the 2 bit WIN_NO signal. Controlled by the most significant bit of WIN_NO, the address paths RTHxx are demultiplexed onto the right address lines.

Because the address lines are shared on a cycle-by-cycle (or actually a two-cycle-by-two-cycle basis) between buffer update and refresh, the output from the 2:1 address demultiplexer can only be enabled in cycles where refresh data is actually to be read from the buffer. This is signalled by the DISPL_REQ signal, controlling the demultiplexer's output enable.

The **fractional part** RF and THF of the output from the transformation tables is

> *a measure of the distance between the position of the image sample (A,A) and the position of the refresh pixel (shadowed circle), along the radial and angular dimensions, respectively.*

By normalizing the two distance values with respect to the distance between two neighbouring samples along the two dimensions, a set of 4 coefficients C0 to C3 can be computed (Coefficient Table). These coefficients will be equal to the weights by which the 4 individual sample values surrounding the refresh pixel ((A,A), (A,B), (B,A) and (B,B)) must be multiplied. By adding the outcome of these 4 multiplications, the mathematically correct value for the refreshed pixel is obtained.

Before output to the Bilinear Interpolator, the coefficient values C0 to C3 must be reorganized according to which of the 4 sample values (EE, EO, OE or OO) they correspond. This is determined by the least significant bits of the (integer parts of the) sample and beam buffer address, RI(0) and THI(0). The algorithm is equivalent to the one computing the RTHxx values.

11.8. Bilinear Interpolator

The Bilinear Interpolator can be implemented by an off-the-shelf, integrated circuit computing the weighted sum of 4 sample values (DEE, DEO, DOE and DOO) and 4 coefficients (CEE, CEO, COE and COO) into a single output value (IM_DATA).

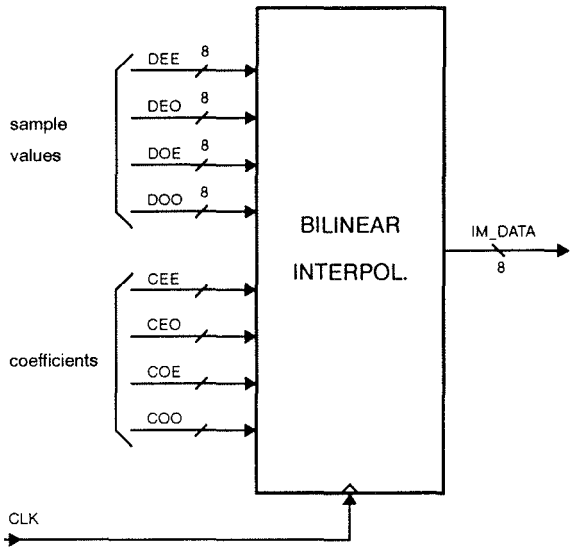


Figure 11.22. Bilinear Interpolator

Although the computation must be performed in real time (that is, 40 Mhz), producing one result every clock cycle, it may be internally pipelined. That is, the output from the Bilinear Interpolator may be a number of clock cycles delayed compared to when the input data were presented. As any other delay embedded in the display refresh path on board the Image Buffer module, it may easily be compensated for by adjusting the horizontal positioning of the windows.

11.9. Pixel Bus Interface

The Pixel Bus Interface block has two functions:

- to output image data onto the Pixel bus during display refresh.
- to receive any display parameters, broadcasted by the display controller during the display retrace periods.

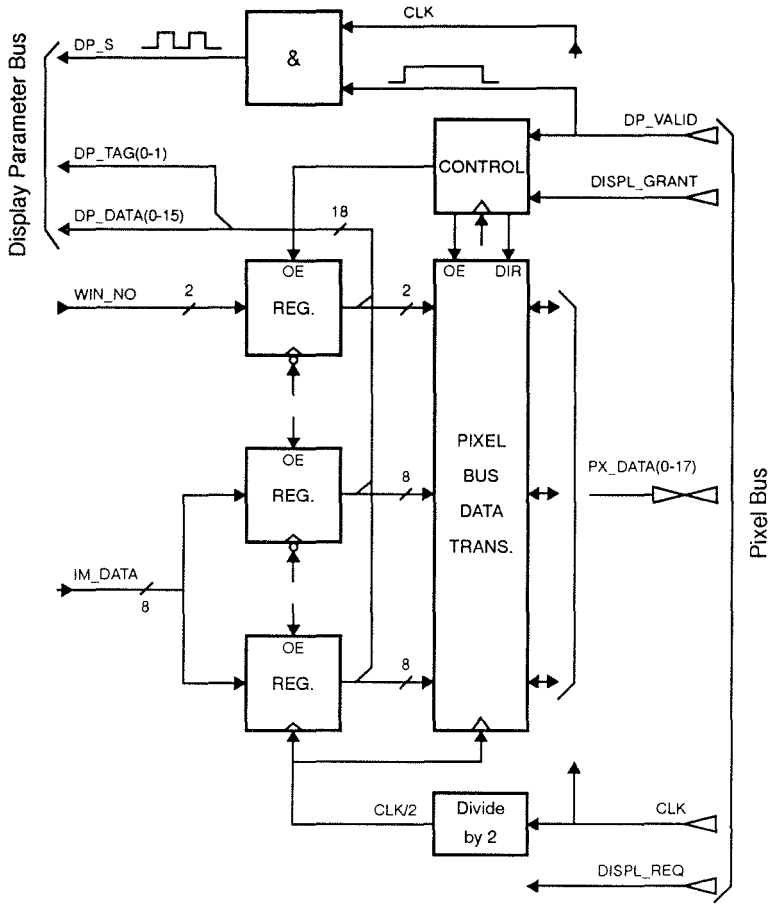


Figure 11.23. Pixel Bus Interface

As far as the **display refresh** function is concerned, pixel values are before output to the Pixel bus assembled two by two into 16 bit values. This is done by clocking a pair of registers on both edges of the CLK/2 signal. CLK/2 is generated by dividing the pixel clock signal CLK by two. In addition, the 2 bit value WIN_NO, identifying the window from which the refresh data are currently being read, must also be included in the Pixel bus. As earlier explained, this is necessary information for the display module responsible of generating the graphic overlays for the windows. In total, the Pixel bus will therefore be 18 bits wide. Pixel bus arbitration is controlled through a display request/ display grant mechanism, located on the display controller. Upon receiving a display grant (DISPL_GRANT), the 18 bit value contained in the Pixel Bus Data Transceiver is output onto the Pixel bus data lines PX_DATA(0-17).

To notify the display modules on **display parameter broadcast**, a dedicated signal line DP_VALID (Display Parameter_VALID) is included in the Pixel bus. DP_VALID is asserted whenever there is valid display parameter data present on the Pixel bus lines. By gating

DP_VALID with the Pixel bus clock signal CLK, DP_VALID is "chopped" into a sequence of contiguous pulses (DP_S, Display Parameter_Strobe), available as a control signal for loading the display parameters into a FIFO or RAM memory. The 18 bit display parameter values transferred over the Pixel bus are divided into a 16 bit data value (DP_DATA) and a two bit tag (DP_TAG). Each display parameter broadcast transfer will consist of one or more *packets*, each packet containing a header part and a data part. To make it possible for the display modules to synchronize to the start of a new packet, "start-of-packet" and "end-of-packet" are coded into the DP_TAG value. By decoding the DP_TAG lines together with the contents of the packet header, a display module will then be able to decide whether the packet is intended for itself or for another module.

References

Akely K. and Jermoluk T. (1988). High-Performance Polygon Rendering. *Proceedings from ACM SIGGRAPH '88 Conference* August 1-5, 1988, Atlanta, Georgia, pp. 239-246.

Andersen V.S. (1990). BASIS: Grunnleggende, men langtfra trivielt! (in norwegian). *PIKSEL'n* nr.4 - desember 1990 - side 8.

Andrews Warren (1989). Data transfer scheme breaks 40-Mbyte/s VMEbus barrier. *Computer Design*, September 1, 1989, p.58.

Andrews Warren (1989). 32-bit buses contend for designers' attention. *Computer Design*, November 1, 1989, p.78.

Andrews Warren (1990). Open buses broaden foothold at all levels. *Computer Design*, May 1, 1990, p.55.

Andrews Warren (1990). DEC's Turbochannel architecture: another contender in the open-bus war? *Computer Design*, June 1, 1990, p.46.

Andrews Warren (1991). Will performance win over sophistication in workstation busses? *Computer Design*, February 1, 1991, p.78.

Annaratone M., Arnould E., Gross T., Kung H.T., Lam M., Menzilcioglu O. and Webb J.A. (1987). The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1523-1537, December 1987.

Antonsson D., Danielsson P.E., Gudmundsson B., Hedblom T., Kruse B., Linge A., Lord P. and Ohlsson T. (1981). PICAP - A system approach to image processing. *Proceedings from the 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management.* Hot Springs, Virginia, Nov.11-13, 1981.

Apgar B., Bersack B. and Mammen A. (1988). A Display System for the Stellar (TM) Graphics Supercomputer Model GS1000 (TM). *Proceedings from ACM SIGGRAPH '88 Conference* August 1-5, 1988, Atlanta, Georgia.

Apple Computer, Inc. (1987). Designing Cards and Drivers for Macintosh II and Macintosh SE. From the series *'Inside Macintosh Library'*. ISBN-0-201-19256-X. Addision-Wesley Publishing Company, Inc.

Arvind and Nikhil R.S. (1987). Executing a Program on the MIT Tagged-Token Dataflow Architecture, pp.1-29. Source unknown.

Bain W.L. and Ahuja S.R. (1981). Performance analysis of high-speed digital buses for multiprocessing systems. *Proceedings from The 8'th Annual Symposium on Computer Architecture (IEEE).* May 12-14, 1981.

Baker and Watkins. (1967). A phase coherent pulse Doppler system for cardiovascular measurement. *Proceedings from the10'th Annual Conference on Engineering in Medicine and Biology*, vol.27, 1967.

Bommer W. and Miller L. (1982). Real-time two-dimensional color-flow Doppler flow imaging in the diagnosis of cardiovascular disease. *Journal of the American College of Cardiology*, 49:944, 1982 (abstract).

Borrill Paul L. (1989). High-speed 32-bit busses for forward-looking computers. *IEEE Spectrum*, July 1989, p.34.

Borrill Paul L. (1990). Futurebus+ : A tutorial. (Part 1). From *'VMEbus Computer Applications'*, a quarterly publication for the VMEbus usergroup society. Vol.4, no. 2, 06-90, pp. 25-34.

Borrill Paul L. (1990). Futurebus+ : A tutorial. (Part 2). From *'VMEbus Computer Applications'*, a quarterly publication for the VMEbus usergroup society. Vol.4, no. 3, 09-90, pp. 25-30.

Briggs F.A., Fu King-Sun, Hwang K. and Wah B.W. (1982). PUMPS Architecture for Pattern Analysis and Image Database Management *IEEE Transactions on Computers*, vol. C-31, no.10, October 1982.

Bursky D. (1990). Triple-Port Dynamic RAM Accelerates Data Movement. *Electronic Design*, May 24, 1990, p.37.

Cantoni V. and Levialdi S. (1983). Matching the Task to an Image Processing Architecture. *Computer Vision, Graphics, and Image Processing*, Vol.22, pp.301-309. 1983.

Cantoni, V. (1985). Classification schemes for image processing architectures. NATO ASI series, vol. F18 - *Computer architectures for spatially distributed data.* H.Freeman and G.G. Pieroni (editors), pp. 37-55. Springer-Verlag Berlin Heidelberg 1985.

Danielsson P.E. (1980). The time-shared bus - a key to efficient image processing. *Proceedings from the 5'th International Conference on Pattern Recognition*, Miami Beach, Florida, December 1-4, 1980.

Danielsson, P.E, and Levialdi, S. (1981). Computer Architectures for Pictorial Information Systems. *IEEE Computer*, pp. 53-67, November 1981.

Data I/O Corporation (1989). ABEL V3.1. January 1989.

DeJager Greg (1990). Push a 32-bit Micro Channel bus to the limit. *Electronic Design*, July 12, 1990, p.61.

Digital Equipment Coorporation (1990). TURBOchannel Hardware Specification. Order number: EK-369AA-OD-006. October 1990.

Dowden, T. (1990). *Inside the EISA Computers.* ISBN-0-201-52397-3. Addison-Wesley Publishing Company, Inc. February 1990.

Duncan, R. (1990). A Survey of Parallell Computer Architectures. *IEEE Computer*, pp. 5-16, February 1990.

Engbersen A.P.J. (1983). TOPPSY: A Time Overlapped Parallel Processing System. *Computer Vision, Graphics and Image Processing*, vol.24, pp.97-106, 1983.

Farrell E.P., Ghani N. and Treleaven P.C. (1979). A Concurrent Computer Architecture and a Ring Based Implementation. *Proceedings from the 6'th Annual Symposium on Computer Architecture (IEEE).* 1979.

Fish J.P. (1975). Multichannel, direction resolving Doppler angiography. *Second European Congress of Ultrasound in Medicine*, 1975 (abstract).

Foley J.D., van Dam A., Feiner S.K. and Hughes J.F. (1990). *Computer Graphics, Principles and Practice.* Second Edition. Addison-Wesley Publishing Company, ISBN 0-201-12110-7.

From *'Textbook of Color Doppler Echocardiography'* ed. by Navin C. Nanda. Lea & Febiger.

Flynn, M.J. (1966). Very High-Speed Computing Models. *Proceedings of the IEEE*, vol.54, no.12, pp. 1901-1909, December, 1966.

Flynn, M.J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, vol. C-21, no. 9, September 1972.

Fuchs H., Poulton J., Eyles J., Greer T., Goldfeather J., Ellsworth D., Molnar S., Turk G., Tebbs B. and Israel L. (1989). Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Proceedings from ACM SIGGRAPH '89 Conference* 31 July - 4 August, 1989, Boston, Massachusets. pp. 79-88.

Gaillat G. (1983). The Design of a Parallel Processor for Image Processing on-board Satellites: An Application Oriented Approach. *Proceedings from the 10'th Annual Symposium on Computer Architecture (IEEE).* Stockholm, 1983.

Gemmar P. (1982). Image correlation: Processing requirements and implementation structures on a flexible image processing system (FLIP). From *'Multicomputers and Image Processing'* edited by K. Preston and L. Uhr.ISBN 0-12-564480-9. Academic Press Inc.

Gharachorloo N., Gupta S., Sproull R.F. and Sutherland I.E. (1989). A Characterization of Ten Rasterization Techniques. *Proceedings from ACM SIGGRAPH '89 Conference* 31 July - 4 August, 1989, Boston, Massachusets, pp. 355-368.

Giacomo, J.D. (1990). *Digital Bus Handbook.* ISBN-0-07-016923-3. McGraw-Hill Publishing Company.

Glass L. Brett (1989). Inside EISA. *Byte*, November 1989, p.417.

Gotoh T., Sasaki S. and Yoshida M. (1985). Two image processing systems challenging the limits of local parallel architecture. Source unknown (CH2229 - IEEE conference 1985).

Guerra C. and Levialdi S. (1985). Computational models for image understanding. From *'Progress in Pattern Recognition 2'*, pp.39-56, L.N. Kanal and A. Rosenfeld (Editors). Elsevier Science Publishers B.V. (North-Holland), 1985.

Gurd J. and Watson I. (1980). Data driven system for high speed parallel computing - Part 1: Structuring software for parallel execution. *Computer Design,* p.91, June 1980.

Gurd J. and Watson I. (1980). Data driven system for high speed parallel computing - Part 2: Hardware design. *Computer Design,* p.97, July 1980.

Halstead Jr. R.H., Anderson T.I., Osborne R.B. and Sterling T.I. (1986). Concert: Design of a Multiprocessor Development System. *Proceedings from the International Symposium on Computer Architecture*. Tokyo, 1986.

Hanaki S. and Temma T. (1982). Template-Controlled Image Processor (TIP) Project. From *'Multicomputers and Image Processing'* edited by K. Preston and L. Uhr.ISBN 0-12-564480-9. Academic Press Inc.

Hasegawa M., Nakamura T. and Shigei Y (1981). Distributed Communicating Media - a Multitrack Bus - Capable of Concurrent Data Exchanging. *Proceedings from The 8'th Annual Symposium on Computer Architecture (IEEE)*. May 12-14, 1981.

Hatle L., Angelsen B. (1982). *Doppler Ultrasound in Cardiology*, Lea & Febinger

Hearn D. and Baker M.P. (1986). *Computer Graphics*. Prentice-Hall International, ISBN 0-13-165598-1.

Hewlett Packard (1990). *HP Vectra 486/25 Hardware Technical Reference Manual*. HP Part No. 5959-5094. January 1990.

Hindin Harvey J. (1985). Bus selection for 32-bit systems limited to two choices. *Computer Design*, September 15, 1985, p.23.

Hwang K. and Briggs F.A. (1984). *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, ISBN 0-07-031556-6.

IBM (1991). Micro Channel Architecture. Material received from IBM in February 1991. 72 pages.

IEEE (1987). *IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus*. ANSI/IEEE Std. 896.1-1987

Intel Corporation (1989/1). A Multibus II overview: Article Reprints and Technical Papers. From the series *'Real-time microcomputing'*.

Intel Corporation (1989/2). *MultibusII Hardware Concepts*.

Intel Corporation (1989/3). *MultibusII Software Concepts*.

Jagannathan R. and Ashcroft E.A. (1984). Eazyflow: A Hybrid Model for Parallel Processing. *Proceedings from the 1984 International Conference on Parallel Processing*, August 21-24, 1984,.

James D.V., Laundrie A.T., Gjessing S. and Sohi G.S. (1990). Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, pp. 74-77, June 1990.

Johnson, Barry W. (1989). *Design and Analysis of Fault Tolerant Digital Systems*. ISBN 0-201-07570-9. Addison-Wesley Publishing Company, Inc.

Jones, S.L.P. and Hardie M.S. (1991). A Futurebus Interface from Off-the-Shelf Parts. *IEEE Micro*, p. 38, February 1991.

Kahaner D. (1990). ETL Dataflow project 7 pages, July 2 1990. Posting retrieved from the USENET 'comp.research.japan' news group.

Kartashev S.I. and Kartashev S.P. (1979). A Multicomputer System with Dynamic Architecture. *IEEE Transactions on Computers*, vol. C-28, no.10, pp. 704-721, October 1979.

Kidode, M. (1983). Image Processing Machines in Japan. *IEEE Computer*, pp. 68-80, January 1983.

Kristoffersen K. (1985). *On the Processing of Doppler Signals in Ultrasonic Blood Velocity Measurements.* Dr.Techn thesis at the Division of Engineering Cybernetics, The Norwegian Institute of Technology, Trondheim.

Kruse B., Gudmundsson B. and Antonsson D. (1980). FIP - the PICAP II filter processor. *Proceedings from the 5'th International Conference on Pattern Recognition*, Miami Beach, Florida, December 1-4, 1980.

Lang T., Valero M. and Alegre I. (1982). Bandwidth of Crossbar and Multiple-Bus Connections for Multiprocessors. *IEEE Transactions on Computers*, vol. C-31, no.12, December 1982, pp. 1227-1234.

Lee R. (1989). Physical Principles of Flow Mapping in Cardiology. From *'Textbook of Color Doppler Echocardiography'* ed. by Navin C. Nanda. Lea & Febiger.

Linker D.T., Angelsen B.A.J., Torp H.G., and Samstad S.O. (1989). Digital Collection and Analysis of two-dimensional ultrasonic Doppler Flow Data. From *'Textbook of Color Doppler Echocardiography'* ed. by Navin C. Nanda. Lea & Febiger.

Lougheed Robert M. (1987). Advanced image-processing architectures for machine vision. *SPIE Vol.755 Image Pattern Recognition: Algorithm Implementations, Techniques, and Technology.* pp.35-51. 1987.

Luetjen K., Gemmar P. and Ischen H. (1980). FLIP : A flexible multiprocessor system for image processing. *Proceedings from the 5'th International Conference on Pattern Recognition*, Miami Beach, Florida, December 1-4, 1980.

Lyle Jim, Gutierrez Michelle (1991). SBus versus TurboChannel. *SunTech Journal*, February 1991, p.57.

Marrin Ken (1985). Bus differences more significant in principle than in practice. *Computer Design*, November 1, 1985, p.23.

Mead, C. and Conway, L.A. (1980). *Introduction to VLSI Systems.* ISBN-0-201-04358-0. Addison-Wesley Publishing Company, Inc.

Motorola Microsystems (1985). *VMEbus Specification Manual Revision C*, February 1985.

Mamekawa K., Kasai C., Tsukamoto M., Koyano A. and Omoto R. (1982). Imaging of blood flow using autocorrelation. *Ultrasound in Medicine and Biology*, 8:138, 1982 (abstract).

Newman W.M. and Sproull R.F. (1979). *Principles of Interactive Computer Graphics.* Second Edition. McGraw-Hill Publishing Company, ISBN 0-07-046338-7.

Perez R.A. (1988). *Electronic Display Devices.* TAB Professional and Reference Books, ISBN 0-8306-2957-2.

Peterson C., Sutton J. and Wiley P. (1991). iWarp: A 100-MOPS, LIW Microprocessor for Multicomputers. *IEEE Micro*, p. 26, June 1991.

Potmesil M. and Hoffert E.M. (1989). The Pixel Machine: A Parallel Image Computer. *Proceedings from ACM SIGGRAPH '89 Conference* 31 July - 4 August, 1989, Boston, Massachusets. pp. 69-78.

Preston, K. Jr. (1983). Cellular Logic Computers for Pattern Recognition. *IEEE Computer*, pp. 36-46, January 1983.

Rhoden D. and Wilcox C. (1989). Hardware Acceleration for Window Systems. *Proceedings from ACM SIGGRAPH '89 Conference* 31 July - 4 August, 1989, Boston, Massachusets, pp. 61-67.

Runyon S. (1987). AT&T goes to 'Warp speed' with its graphics engine. *Electronics*, July 23, 1987, p.54.

Saponas T.G. and Crews P.L. (1980). A Model for Decentralized Control in a Fully Distributed System. *Proceedings from Distributed Computing, COMPCON Fall 80 (the 21'th IEEE Computer Society International Conference)*, September 23-25, 1980, Washington.

Sasaki S., Gotoh T., Satoh T. and Iwase H. (1985). High speed pipeline image processor with a modifiable network. *Proceedings of the First International Conference on Supercomputing Systems: SCS 85*. St. Petersburg, Florida, Dec.16-20, 1985.

Satomura S. (1957). Ultrasonic Doppler method for the inspection of cardiac functions. *Journal/ Acoustical Society of America*, vol.29, pp.1181-1185, 1957.

Sawkar P.S., Forquer T.J. and Perry R.P. (1983). Programmable modular signal processor - a data flow computer system for real time signal processing. *Proceedings from the 1983 IEEE International Conference on Parallel Processing*, August 23-26, 1983.

Sha L., Rajkumar R. and Lehoczky J.P. (1991). Real-Time Computing with IEEE Futurebus+. *IEEE Micro*, June 1991, p.30.

Spragins J., Jafari H. and Lewis T. (1979). Some simplified performance modeling techniques with applications to a new ring-structured microcomputer network. *Proceedings from the 6'th Annual Symposium on Computer Architecture (IEEE)*, 1979.

Srini V.P. and Shriver B.D.(1980). Abstract dataflow protocol for communication in distributed computer systems. *Proceedings from Distributed Computing, COMPCON Fall 80 (the 21'th IEEE Computer Society International Conference)*, September 23-25, 1980, Washington.

Srini V.P. (1985). A Fault-Tolerant Dataflow System. *IEEE Computer*, pp.54-68, vol.18, March 1985.

Srini V.P. (1986). An Architectural Comparison of Dataflow Systems. *IEEE Computer*, pp.68-88, March 1986.

Sun Microsystems, Inc. (1989). *The SBus Specification, Revision A*. September 1989.

Skillicorn, D.B. (1988). A Taxonomy for Computer Architectures. *IEEE Computer*, pp. 46-57, November 1988.

Sutherland, I.E. (1989). Micropipelines. *Communications of the ACM*, pp. 720-738, vol.32, June 1989.

Taub D.M. (1984). Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus. *IEEE Micro*, August 1984, p.28.

Taub D.M. (1987). Improved Control Acquisition Scheme for the IEEE 896 Futurebus. *IEEE Micro*, June 1987, p. 52.

Texas Instruments, Inc. (1990). *NuBus Interface Products Data Book.*

Torborg J.G. (1987). A Parallel Processor Architecture for Graphics Arithmetic Operations. *Proceedings from ACM SIGGRAPH '87 Conference* July 1987, pp. 197-204.

Torp H. (1990). *Signal Processing in Real-time, two dimensional Doppler Color Flow Mapping*. Thesis submitted to The Norwegian Institute of Technology, div. of Engineering Cybernetics for the degree of Dr.Techn. Trondheim 1990 (preliminary issue).

Treleaven P.C., Brownbridge D.R. and Hopkins R.P.. (1982). Data-Driven and Demand-Driven Computer Architecture. *ACM Computing Surveys*, Vol.14, No.1, pp.93-143, March 1982.

Turley James L. (1990). The indivisible RMW cycle: Or is it? *The Supermicro Journal*, no.5, summer 1990, p.6.

Turrel Don (1989). VME64: Double your VMEbus speed. From *'VMEbus Computer Applications'*, a quarterly publication for the VMEbus usergroup society. Vol.3, no. 4, 12-89, pp. 31-32.

Vernon M.K and Manber U. (1988). Distributed Round-Robin and First-Come-First-Serve Protocols and Their Application to Multiprocessor Arbitration. *Proceedings from 15'th International Symposium on Computer Architecture*, 1988, Honululu, pp.269-277.

Viitanen J., Vanni P., Salo J. and Saarinen J. TAMIPS - A parallel processor for machine vision. *Source unknown.*

Voorhies D., Kirk D. and Lathrop O. (1988). Virtual Graphics. *Proceedings from ACM SIGGRAPH '88 Conference* August 1-5, 1988, Atlanta, Georgia, pp. 247-253.

Vranesic Z.G., Stumm M., Lewis D.M. and White R. (1991). Hector: A Hierarchically Structured Shared-Memory Multiprocessor. *IEEE Computer*, January 1991, p.72.

Warren Andrews (1990/1). Futurebus+ spec completed - almost. *Computer Design*, February 1, 1990, p. 22.

Warren Andrews (1990/2). Bridging today's busses to Futurebus. *Computer Design*, February 1, 1990, p. 72.

Warren Carl. (1990). Micro Standards: The Scalable Coherent Interface. *IEEE Micro*, pp.80-82, June 1990.

Watson I. and Gurd J. (1982). A Practical Data Flow Computer. *IEEE Computer*, p.51, February 1982.

Whang Min-Hur, Kua Joe (1990). Join the EISA Evolution. *Byte*, May 1990, p.241.

White George (1989). A Bus Tour. *Byte*, September 1989, p.296.

Wilson R. (1990). DRAM vendors address increasing specialization. *Computer Design*, December 1, 1990, p.63.

Yalamanchili, S., Palem, K.V., Davis, L.S., Welch, A.J., and Aggarwal, J.K. (1985). Image Processing Architectures: A Taxonomy and Survey. *Progress in Pattern Recognition 2*, L.N. Kanal and A. Rosenfeld (editors), pp. 1-37. Elsevier Science Publishers B.V. (North-Holland), 1985.

Zhang X. (1991). System Effects of Interprocessor Communication Latency in Multicomputers. *IEEE Micro*, p. 12, April 1991.

A. Modern digital buses

From economical as well as many other reasons, it will be advantageous to base the design on an already accepted and in-use bus standard. Looking at today's situation, there are in that case several to choose from:

- VME bus.
- Multibus II (IEEE 1296).
- Micro Channel Architecture (MCA).
- Extended Industry Standard Architecture (EISA).
- NuBus.
- SBus.
- TURBOchannel.
- Futurebus+ (IEEE 896).
- Scalable Coherent Interface, SCI (IEEE 1596).

A.1. VME-bus

In 1981, Motorola, in collaboration with Mostek, Signetics/Phillips and CSF Thompson of France, announced the VMEbus as nonproprietary standard in the public domain. The VMEbus specification was not a made from scratch, it evolved from VERSAbus, the proprietary Motorola bus standard used in their MC68000 family systems in the late seventies.

Today, VMEbus is by far the most widely used 32 bit bus system, specially for industrial applications, with more than 300 board and system manufacturers. Key features:

- Non-multiplexed data (16/32 bit) and address (23/32 bit) lines.
- Asynchronous operation
- Standard version fits in a single 96 pin DIN connector, extended version (32 bit data and 31 bit address) uses a second connector.
- Multiple address spaces due to a 6 bit address modifier code tagging each transaction.
- Theoretical bandwidth: 40 Mbytes/second (32 bit data).
- Centralized 4-level arbitration.
- Seven interrupt lines with vectored interrupt acknowledge.

A.1.1. Data transfer

VMEbus data transfers can be done in units of bytes, words (16 bit) or long words (32 bits). In addition to ordinary read and write, a read-modify-write cycle is included, making it possible to implement semaphore operations in a multiprocessor system.

For efficient transfers of contiguous blocks of data, block read and write transfers are provided. Only the start address is initially transferred over the bus, it is then the addressed slave's responsibility to generate addresses with a local counter for the individual data units within the

block. Block transfers are not allowed to cross 256 byte boundaries, the block transfer maximum length is therefore also 256 bytes. Except from this, the addressed slave gets no advance information about the actual length of the block transfer.

A special kind of data transfer is the *address-only* cycle. As the name implies, only an address is transferred (set up on the address lines), no data is presented on the data lines. This is the only VMEbus transfer which does not require a response (handshake) of any kind. The application of this feature is system specific and is not defined in the VMEbus specification.

Due to the asynchronous bus protocol using an open-collector, active low handshake line (DTACK*), broadcast operations is impossible to implement and therefore not a part of the VMEbus specification. To implement a handshaken broadcast operation where the master wait for the slowest slave module, an inverse handshake line is required (*release-when-ready*).

A.1.1.a. VME64

In an ordinary VMEbus block transfer, the address lines will be idle after the initial start address setup. By using these lines for transferring data, a total of 63 lines (32 data and 31 address lines) are available for data transfer in a 32 bit VMEbus system. The 64'th line is obtained by using the LWORD* control line, usually used for signalling whether the current transfer is a 32 bit (long word) transfer or not. In the 64 bit mode, this signal therefore has no meaning as a control signal. Because the timing of a VME64 block transfer is compatible with an ordinary 32 bit block transfer, the VME64 is compatible with conventional VMEbus hardware and software. In this mode, the theoretical VMEbus bandwidth capacity is 80 Mbytes pr. second. The maximum length of a VME64 block transfer is 512 bytes.

The VME64 transfer mode was first presented by Performance Technology (Rochester, NY). Later, Harris has developed its own semi-synchronous version stretching maximum performance even higher. VME64 is expected to be included as a part of the VMEbus standard in the next revision.

A.1.2. Arbitration

The VMEbus uses a centralized, four-level arbitration scheme, each level having its own bus request/ bus grant signal line pair. Each potential master module is connected to its request bus line through an open-collector interface, several modules can therefore use the same bus arbitration level. The four bus grant lines are daisy-chained from slot to slot. In case of simultaneous bus requests on the same level, the module located in the slot nearest to the arbiter module (slot 1) will therefore have priority.

Three types of arbitration philosophies are described by the VMEbus specification:

Prioritized arbitration assigns the bus according to a fixed priority scheme where each of the four levels has a priority in the range of 3 (BREQ3*, highest) to 0 (BREQ0*, lowest).

Round robin arbitration assigns the bus on a rotational priority basis where each of the four levels in turn has the highest priority.

Single level arbitration only accepts requests on one bus request line (BR3*). Used in small system with few potential bus masters.

If there is a pending bus request on a level with a higher priority than the current bus master, the arbitration module will inform the current bus master about this by asserting a dedicated signal line (BCLR*, Bus CLeaR). This is a request to the master to relinquish bus mastership at "an appropriate stopping point". No timeout limit is included in the VMEbus specification, an asserted BCLR* is therefore more to be regarded as a polite request than an order.

A.1.3. Interrupt handling

The VMEbus includes 7 level-sensitive dedicated interrupt lines. Due to the open-collector interface, several modules can be connected to the same interrupt line.

The interrupt lines are serviced by one or more interrupt handlers. Each handler services an exclusive subset of the seven interrupt lines. When detecting an interrupt, the interrupt is serviced by the appropriate interrupt handler by executing an interrupt acknowledge cycle. The 3 least significant address lines are then used for signalling the interrupt number being serviced, while a daisy-chained interrupt acknowledge signal assures that only one module responds to the interrupt acknowledge cycle (in case of several simultaneous interrupts on the same interrupt level). The responding interrupter module responds to the acknowledge cycle by presenting an 8 bit interrupt vector on the data lines D7-0, identifying the cause of the interrupt.

A.1.4. Multiprocessing facilities

Except from the already described read-modify-write data transfer mode, facilitating semaphore like operations, the VMEbus provides no specific support for multiprocessing applications.

A.1.5. System configuration

The VMEbus does neither support geographical addressing, nor control and status registers for module configuration.

Reference: [Motorola Microsystems 1985].

A.2. Multibus II (IEEE 1296)

The Multibus II is, as indicated by its name, a multiple bus architecture. Three bus structures are defined exclusively by Multibus II. These are the Parallel System (iPSB), the Local Bus Extension (iLBX II) and BITBUS. Two buses, the iSBX I/O Expansion bus and the Multichannel DMA (Direct Memory Access) I/O bus are carried over from the Multibus I architecture.

The main bus in the Multibus II architecture is the Parallel System Bus (iPSB). This is the Multibus II system highway, conveying data and control information between the various bus modules. In the rest of this document, the Multibus II discussion will be restricted to the Parallel System Bus. The term "Multibus II" will accordingly be used as a synonym for the same bus. Key features:

- Synchronous operation, 10 Mhz system clock.

- Multiplexed 32 bits data and address lines.
- Data transfers as 8, 16, 24 or 32 bit items.
- Supports up to 20 bus masters.
- Maximum bus capacity: 40 Mbyte pr. second (burst transfer).
- Message passing protocol included in the bus specification.
- Geographical addressing.

A.2.1. Data transfer

Data transfer operations on the bus can be divided into three phases:

In the first phase, the **request** phase, a bus module (the requester) emits a request for an operation. The requested operation is specified by 10 status/ control lines SC(9:0)*. These lines tell whether the operation is a read or a write, the data size, which address space to use and whether to perform bus locking or not. Additionally, two lines are parity for the other eight. The status/ control lines must be decoded before the I/O interface can interpret the address used for the transaction. The request phase last for a single clock cycle and is followed by the

reply phase. In this phase, the actual data transfer takes place between the requester and the addressed slave (the replier). The replier is responsible for generating status information out on 5 of the 10 status lines SC(9:0)*. This five wire handshake consists of a single line ready signal, telling when it is ready so that the requester may proceed with the cycle three status lines to indicate the failure or success of the operation. Additionally, parity for the 4 lines must be generated. In case of a multiple byte transfer, the requester and the replier will remain in the reply phase until all bytes are transferred.

Finally, when the transfer is terminated, the **exception** phase may be entered. This is the case if the requester detects any reply errors from the replier. Most transactions will be error-free, and the exception phase will not be entered. The Multibus II protocol is a very restrictive protocol, and the specification details numerous state diagrams to track and control the requester-replier operation.

A.2.1.a. Burst transfer

In case of a multiple byte (burst) transfer, the handshake protocol is executed each transfer cycle. The replier's I/O interface does not know in advance the length of the burst transfer, this is signalled by the requester through the state of one of the status/ control lines. However, the message passing protocol, build on top of the data transfer mechanism will give advance information about the length of the transfer.

A.2.2. The message passing protocol

The Multibus II message passing protocol is aimed to remove the task of passing messages between intelligent boards from each board's CPU, and delegate this to the bus interface logic. To ensure efficient utilization of bus bandwidth and reduce the amount of board real estate needed to implement the bus interface logic, a dedicated VLSI chip called the Message Passing Coprocessor (MPC) is designed.

The MPC's major features the support of inter-processor communication.This is implemented by two types of messages: Unsolicited and solicited messages.

A.2.2.a. Unsolicited messages

Unsolicited messages are used as a signalling mechanism between boards, replacing traditional system interrupts. They have the following characteristics:

- They are short, with a fixed length of 32 bytes allowing up to 28 bytes of user data.
- They can arrive unexpectedly.
- Consequently, no buffer allocation or sender/ receiver synchronization is necessary previous to the message transfer.

Unsolicited messages can be any of five types: General interrupt, broadcast interrupt, buffer request, buffer grant or buffer reject. An unsolicited message has a format specific to the actual message type.

A.2.2.b. Solicited messages

Solicited messages are used to transfer large amounts of data. They have the following characteristics:

- Variable length, up to 16 Mbytes.
- Buffer space must be allocated in advance (through the exchange of unsolicited buffer request/ buffer grant messages).

The actual data transfer is done as a series of solicited packets on the bus, each packet containing 32 bytes of data. This splitting is done to ensure that arbitration contests for the bus can be done at least every 1 microsecond. Otherwise, long message transfers could cause other (and possibly higher prioritized) Multibus bus masters to be unacceptably delayed.

A.2.3. Arbitration

The Multibus II uses a distributed, Parallel arbitration mechanism, executed in Parallel with data transfer: All boards request use of the bus through a common bus request line BREQ*. In case of several simultaneous requests, the bus is granted to the numerically highest board as identified by its 6 bit priority value, output on lines ARB(5:0). The priority value consists of a 5 bit software assignable identity code and a high priority bit. Priorities can therefore be changed dynamically while the bus is operating.

Two arbitration algorithms are supported: Fairness and high priority. The selected algorithm is signalled through line ARB5.

The **fairness** mode is used for data transfer and is "polite". That is, if the bus is busy when a board is about to issue a bus request, the board will hold the request back until the bus is free. Due to the maximum packet size of 32 bytes, this will never take more than 1 microsecond. In case of simultaneous requests, a board who has been granted (and used) the bus, will not request the bus again until all other requesters are serviced.

The **high priority** mode is used for interrupts and is "impatient": A board issuing a high priority bus request is then allowed to "barge" into an already ongoing bus arbitration cycled and be guaranteed the next access to the bus. If more than one board issues a high priority bus request, the boards will be processed in numerical order according to their priority value.

A.2.4. Interrupt handling

Unlike most other buses, Multibus II does not have dedicated interrupt lines, but implements "interrupts" by the already described mechanism based on unsolicited packets in combination with high priority bus arbitration. Two types of interrupt packets are implemented: General interrupt, issued to a specific board, and broadcast interrupt, to all boards. In almost all cases, this interrupt mechanism will give a latency of less than 1 microsecond.

A.2.5. Multiprocessing facilities

The Multibus II multiprocessing facilities are implemented solely within the frame of the message passing concept: No explicit mechanisms for usual features like bus locking and broadcasts are provided.

A.2.6. Interconnect address space

The concept of the *interconnect address space* is aimed to solve three major problems:

- Board identification.
- System configuration.
- Diagnostics.

The interconnect address space consists of a special set of registers located on each board in a Multibus II system. The interconnect address registers are always dual-ported. That is, they can be accessed from the local on-board CPU as well as from other Multibus boards. The registers can be divided into three groups:

The **board identity** registers are read-only and contain formatted information about that specific board such as board type, manufacturer, its revision level, what optional components are installed and other board-specific information.

Board configuration registers are read/write registers that allow the system software to set the configuration of many of the hardware options on the board. They therefore replaces the traditional jumper fields and facilitates the board (and thereby the whole system) to be dynamically reconfigurable.

The **diagnostic** registers are used for starting, stopping and status reporting of self-contained diagnostic routines on the board, otherwise known as built-in-self-tests (BIST).

The interconnect address space is accessed by a geographical addressing scheme: A complete 16 bit interconnect address is formed by a board slot identity value and a register offset, addressing a specific register on the board located in a specific slot.

Reference: [Intel 1989].

A.3. Micro Channel Architecture (MCA)

When time was due for IBM to extend the PC AT bus to 32 bit, they chose, from technical as well as policy reasons, to design a bus *not* compatible with its predecessor. Today, this bus is being used predominantly in the IBM PS/2 product line. Its name is MCA, the Micro Channel Architecture. Key features:

- Asynchronous data transfer protocol, not related to a system clock.
- Non-multiplexed 32 bits address and data buses.
- Separate address spaces for I/O (64 Kbyte) and memory (4 Gbyte).
- Central arbitration control supporting up to 16 devices to compete for the bus.
- Direct Memory Access (DMA) procedure supporting multiple DMA channels with burst capability.
- Level-sensitive interrupts with interrupt sharing on all levels.
- Theoretical transfer rate up to 20 Mbyte/second.

A.3.1. Data transfer

MCA uses a synchronous data transfer protocol where waitstates can be inserted on all transfers. Three signals are vital to the data transfer protocol: The controlling master tells that the address- and status lines are valid by asserting the *Address Decode Latch* (ADL) signal. The *Command* signal CMD is used by the master to signal valid data on data lines (write cycle) or request for data (read cycle). To make a slow slave able to extend the transfer cycle, a handshake signal *CarD CHannel ReaDy* CD CHRD is provided.

A transfer cycle is always initiated by that the master outputs the address and type of transfer (memory/IO, read/write) on the address and status lines, respectively. The ADL signal is then asserted telling the slaves that they may latch and decode these lines. If the slave is capable of servicing the access within a certain time specified by the MCA specification, it does so and the data is transferred by what is called a **basic transfer cycle**.

However, if the addressed slave is unable to finish the cycle within the specified time limit, the transfer cycle can be extended by the slave by asserting the handshake signal CD CHRDY. There are two types of extended transfer cycles, they are distinguished by the time CD CHRDY is unasserted to its normal (inactive) state.

A **synchronous extended cycle** occurs when CD CHRDY is released within a 30 ns after the leading edge of CMD. This causes the cycle to be extended by 100 ns or 1 waitstate. In case of a read cycle, the slave provides the data within a specified time from CMD.

If one waitstate is not sufficient, the slave may assert CD CHRDY up to a maximum of 3 microseconds. No restrictions are then applied to the release of CD CHRDY, it will be fully asynchronous to the CMD signal. This is called an **asynchronous extended cycle**.

A.3.1.a. DMA transfer

The DMA controller executes single-data transfers unless the addressed DMA slave requests a burst transfer by pulling the *BURST signal low. Note that a burst transfer is requested by the DMA slave, *not* initiated by the DMA controller. The controller will execute burst transfer until the slave unasserts the BURST* signal.

The DMA transfer will use basic, synchronous extended or asynchronous extended transfer cycles depending on the capabilities of the participating DMA slave.

Termination of the DMA transfer is indicated by the controller by asserting the Terminal Count (*TC) signal during the last data transfer. A DMA transfer may be disrupted by another device on the bus. This is done by the use of the *PREEMPT signal (see the "Arbitration" paragraph).

A.3.2. Interrupt handling

The interrupt mechanism supports a total of 11 level-sensitive, active low interrupt lines. The lines are shared by all slots, they must therefore be driven by open-collector drivers. To be able to identify the source of a signal on a specific interrupt line, each card capable of generating interrupts must maintain an "interrupt asserted" bit, readable by software on the interrupt controller. This bit must explicitly be cleared by the interrupt service routine.

A.3.3. Arbitration

The MCA implements a combination of a centralized and a distributed arbitration scheme:

A *Central Arbiter* is located on the system card and is responsible for controlling the arbitration cycle. That is, issuing a bus grant (ARB/*GRANT low) in response to a bus request (*PREEMPT low) from a MCA card. The requesting card signals its own priority by driving their 4 bit priority code onto the lines ARB(3:0). These lines are shared among all MCA bus masters and can be driven independently through open-collector drivers. The *Local Arbiter* on each MCA card is therefore responsible for comparing the state of the ARB(3:0) lines with its own priority vector, withdrawing the request if any higher prioritized card requests the bus.

A.3.3.a. Preemption

Usually, bus masters must arbitrate before each bus transfer. However, drivers requiring multiple data transfers will by asserting the BURST* signal keep the bus until BURST* is released (all transfers are complete). A bursting device may also stop a transfer if a second device requests the bus by asserting PREEMPT*. A bursting device may not ignore an asserted PREEMPT* more than 7.8 microseconds. If it does so, the Central Arbiter will forcibly take the bus away from the bursting device by raising the ARB/GNT* signal.

A.3.3.b. Fairness mode

A programmable fairness feature ensures each device a fair share of the channel time: If a device completes a bus transfer while other bus requests are active, this device is not allowed to request for the bus again until all other active requests have been serviced. All arbitrating devices will therefore be serviced in order of priority before the same device can regain control of the bus.

A.3.4. Multiprocessing facilities

Except from the more conventional mechanisms as far as arbitration and interrupt handling are concerned, the MCA has no specific support for synchronization and communication between multiple devices on the bus. By the use of other MCA mechanisms, however, this limitation can to some extent be overcome:

Broadcast and multicast can be implemented by mapping several devices into the same address space. Every MCA slot has its own CD CHRDY line. The individual CD CHRDY lines are gated together on the MCA system card, producing the signal CHRDYRTN. This signal will be high if all the individual slot lines are high, i.e all cards are ready. In this way, the controlling master is able to monitor all slaves participating in the transfer. The slowest slave will then determine what transfer cycle to use (basic, synchronous or asynchronous extended). Note that because the data lines are driven with three-state and not open-collector drivers, several devices can not drive the data lines at the same time. Broadcall (i.e. multiple source read) can therefore not be implemented.

By using the BURST* signal, bus locking can be obtained. Each transfer within a burst transfer can, by programming the status/control signals S0 and S1 accordingly, independently be a memory or I/O access, read or write. Test-and-set and read-modify-write mechanisms can then be implemented as burst transfers.

A.3.5. Geographical addressing

Each MCA slot is connected to the MCA system card (slot) via a unique line called CarD SETUP (CD SETUP). When this line is activated, a specific channel card is selected and access to the card's configuration data space is obtained.

A.3.6. System configuration

To eliminate the tedious, and often erroneous, task of installing jumpers and toggling switches, their functions are replaced by a set of registers called the *Programmable Option Select* registers, POS.The POS registers are accessible through the geographical addressing scheme, and make it possible for the MCA system controller to poll each card to determine the card's characteristics and write configuration data to it. Due to this "programmable jumper" approach, cards can be dynamically relocated and reconfigured while the system is running. Also, each card stores a unique identification number in the POS registers so that the system can identify each card.

Reference: [IBM 1991].

A.4. Extended Industry Standard Architecture (EISA)

As already discussed, IBM's new 32 bit PC bus (the Micro Channel Architecture, MCA) is not compatible with its predecessor. Due to the vast number of available boards built for the "old" bus, this opened an attractive possibility for other computer manufacturers: To define a 32 bit enhancement of the old bus while maintaining downward compatibility with the old PC, XT and AT bus structures. The result of this effort became EISA, developed and supported by a consortium of over 50 leading manufacturers, led by a core of nine members: Wyse, AST Research, Tandy, Compaq, Hewlett-Packard, Zenith, Olivetti, NEC and Epson (mnemonically *watchzone*, also called "the gang of nine"). Key features:

- Fully synchronous operation with an 8 Mhz system clock.
- Non-multiplexed 32 bit data and 32 bit address lines.
- Peak burst transfer rate equal to 33 Mbyte/second.
- 15 interrupt lines.
- Switchless autoconfiguration.

A.4.1. Data transfer

When designing the EISA bus, the main objective was to make it compatible through the whole range of personal computers from the basic PC up to the 32 bit EISA machines. Therefore, the bus protocol must be fully adaptive as far as bus width is concerned: Any bus master can access any slave in the system, even if their bus widths differ. To make the addressed slave able to inform the bus master which bus width it supports, dedicated signal lines are used. When incompatibility between the master and the slave occurs, the EISA bus controller intervenes into the cycle through a mechanism called *cycle translation*. The transfer requested by the master is then converted into transfers supported by the slave, e.g. a 32 bit transfer can be executed as four separate 8 bit transfers. This mechanism is fully invisible and transparent to the master.

Data are transferred in 8, 16 or 32 bit units. Even if the protocol is synchronous (no handshake), slow slaves may by asserting the signal line EXRDY insert waitstates to get more time to finish the current transfer.

Burst transfers are also supported as 8, 16 or 32 units. Before a burst transfer can take place, the bus master and addressed slave exchanges information about their burst transfer capabilities ensuring that both parties support this transfer. This is done through two dedicated signal lines, MSBURST* (driven by master) and SLBURST* (slave). End-of-transfer is signalled by the signal line T-C (Terminal Count). In case of a write burst (master to slave), this line is asserted by the master, on a read burst by the slave.

A.4.2. Arbitration

The EISA bus implements an arbitration scheme centralized around the bus controller: Each EISA slot is connected to the bus controller by its own bus request/ bus acknowledge (MREQx*/ MACK*) signal line pair. Additionally, DMA transfers can be requested (and

acknowledged) on 7 channels, each channel having its own request and acknowledge signal pair. All DMA channel lines are connected to all slots, as a bus. Finally, the refresh controller has its own bus request line for memory refresh (REFRESH).

The memory refresh controller, the highest priority DMA controller and the EISA bus masters compete for bus mastership through a three-way rotating arbitration scheme. Among the EISA bus masters, priorities can be either fixed or rotated (round robin). This scheme ensures a fair amount of access to the bus both for the bus masters and the refresh controller. As far as the DMA channels are concerned, however, it is possible for a low priority DMA channel to be starved for use of the bus.

A.4.3. Interrupt handling

The EISA provides a total of 15 shared, level-sensitive interrupt lines (note, however, that ISA boards must have exclusive use of interrupt lines they are connected to). To identify the source of a generated interrupt, the interrupt controller responds to an interrupt by executing two interrupt acknowledge cycles. On the second cycle, an interrupt vector is read from the data lines D(7:0).

A.4.4. Multiprocessing facilities

To ensure exclusive access to the bus during indivisible operations (e.g. test-and-set, read-modify-write), a bus lock mechanism is supported through a dedicated signal line LOCK*. By asserting this signal, the current bus master is guaranteed that no other board will be able to access the bus until LOCK* is released.

Broadcast and broadcall is not supported.

A.4.5. System configuration

Like a number of other digital buses, EISA supports a geographic addressing scheme where boards can be addressed by the location of the slot in which the board resides instead of its memory address. Each board contains, in a fixed geographically addressed location, information about the board and the resources located on the board (ports, memory etc.). Together with programmable switches and configuration registers instead of hardwired jumpers this allows for system configuration totally through software control.

References: [Dowden 1990] and [Hewlett Packard 1990].

A.5. NuBus

NuBus was the first standard 32-bit bus, developed by a research group at MIT, Massachusetts Institute of Technology in the late 70'ties. The commercial rights to NuBus was first purchased by Western Digital and finally by Texas Instruments. Despite this heavy industrial support, no significant commercial products emerged with NuBus until Apple Computer selected the bus for its second generation MacIntosh computer, the MacII, in 1987. Later, the bus was also adopted by Steve Jobs for his NeXt computer. Key features:

- Synchronous operation based on a 10 Mhz system clock. The clock has a 75% duty cycle, both clock edges are used for timing reference.

- System architecture independent of any particular microprocessor family.
- Multiplexed 32-bits address and data bus.
- Handshakes on all data transfers, slow slaves may insert waitstates.
- Maximum theoretical block transfer capacity: 37.5 MBytes/second.
- Virtual as well as physical interrupts.
- Distributed, Parallel arbitration scheme based on a fairness algorithm to prevent starvation.
- Geographical addressing.

A.5.1. Data transfer

The NuBus supports single data as well as block transfer transactions, read and write. Handshake (acknowledge) from the responding slave is required for all transfers.

A.5.1.a. Single data transactions

A single data transaction conveys one data item (32 bit word, halfword or byte). Which part of the addressed word to transfer, is told by the two least significant address bits AD1,0, the type of transfer by the transfer mode/status bits TM0,1. All transfers are unjustified: No matter the transfer mode used to access it, a specific byte of data is always conveyed on the same signal lines.

Each single data transaction must be handshaked by the addressed slave by asserting an acknowledge signal. At the same time, the slave also outputs status information on the transfer mode/ status lines, signalling whether the transaction was successful or an error (bus error, timeout etc.) occurred.

Due to the handshake protocol, a transaction can take any number of cycles up to a system specific timeout.

A.5.1.b. Block data transactions

A block data transaction consists of a start cycle specifying the start address, followed by multiple data cycles moving data to or from sequential address locations on the selected slave, and an acknowledge cycle. Allowed block lengths are 2, 4, 8 and 16 words, the actual block length is signalled by the lower address lines during the start cycle. Only 32 bit NuBus word transactions is supported in this mode. Referred to the block data bus master, the block transfer can be both read and write.

Acknowledge is given by the responding slave for each word transfer within the block as well as for the block as a whole. The (final) block acknowledge supplies status information telling whether the block transfer was successful or not.

A.5.2. Interrupt handling

NuBus supports two types of interrupt mechanisms, virtual (memory mapped) interrupts and physical interrupts associated with dedicated bus signals.

A.5.2.a. Virtual interrupts

The virtual interrupts are implemented as write transactions into an area of (local) memory monitored by the NuBus board. Any address range on the board can be defined as its interrupt space. This allows interrupts to be posted to individual boards. Additionally, it allows the interrupt priorities to be software specified by memory mapping the priority level.

No dedicated bus lines or protocols are needed to implement virtual interrupts.

A.5.2.b. Physical interrupts

Each slot is through an open collector wired-OR interface connected to a common interrupt line /NMREQ. To determine the source of an interrupt, the system interrupt controller must poll each slot (board) capable of generating the interrupt.

In addition to the two described interrupt mechanisms, the Apple MacII computer implements a third scheme: Each slot has its own dedicated /NMRQ line, connected to the slot where the interrupt controller (i.e. the CPU board) is located. The need for the individual poll of potential interrupt sources is thereby eliminated.

A.5.3. Arbitration

NuBus implements a Parallel, distributed arbitration scheme: Each NuBus slot is assigned its own unique 4 bit identification (ID) number, the slot number (for further details, see the "Geographical addressing" paragraph). When a NuBus master wants to use the bus, it signals this by asserting the dedicated /RQST line along with its ID number on the arbitration bus lines /ARB3-0. The boards are connected to the bus request line as well as the arbitration bus lines through open collector interfaces. That is, several boards are allowed to drive these lines simultaneously. The board(s) requesting the bus, will during the arbitration period continuously compare the state of the arbitration bus lines /ARB3-0 to its own 4 bit ID number. According to a specified algorithm, boards with non-matching IDs will then withdraw their requests and the highest priority board (i.e. the board with the highest ID (slot) number) will eventually remain as the bus winner.

However, to prevent board starvation and to distribute bandwidth more evenly than a strict priority arbitration scheme will give, a fairness arbitration policy is implemented: All boards participating in an arbitration contest are serviced before any new (and possibly higher) priority boards is allowed to compete for the bus.

A.5.4. Multiprocessing facilities

To facilitate synchronization between bus modules, some sort of indivisible operations must be implemented. NuBus supports two types of locking schemes to obtain this: Bus locking and resource locking.

A.5.4.a. Bus locking

If the NuBus is the only channel or path to access a board, indivisible operations on that board can be ensured by locking this channel. That is, to prevent other bus masters from getting access to the bus until the current bus master has finished its operations on that board. This is done by that the (current) master does not withdraw its bus request after it has won the arbitration, but simply continues to request for the bus. This will prevent other masters from getting access to the bus until the current bus master withdraws its request.

A.5.4.b. Resource locking

If a resource, which can be a board as well as a module on a board, can be accessed through other channels than the NuBus, it is not enough to lock the NuBus itself. This is the case for a dual ported RAM module located on a NuBus CPU-board, the RAM module is accessible from the NuBus as well as from the local processor bus. Therefore, the lock must be implemented on the resource itself.

NuBus accomplishes this by a special *resource-lock-cycle*. The addressed resource is then locked to NuBus (i.e. it can not be accessed by other than the NuBus), until the lock is explicitly released by a similar operation called a *null-cycle*. Both the resource-lock and the null-cycles belong to a set of special NuBus cycles called *attention-cycles*.

A.5.4.c. Broadcast/broadcall

This is not supported by NuBus. Because handshake from the addressed slave is required on each transfer, an inverted, open collector acknowledge line would there be needed to implement a multi destination/ multi source protocol.

A.5.5. Geographical addressing

This is accomplished by each slot having four identification signals /ID3 to /ID0, hardwired to logic high or low on the backplane. This 4 bit code, unique for each slot, serves two purposes:

- It is used as the 4 bit ID priority code during arbitration.
- By substituting 4 of the NuBus address signals by the 4 ID signals in the address decoding logic on each board, each slot can be mapped into its own unique block in the memory address space. Depending on which address signals are substituted, two such schemes are supported: The *slot space* allocating 16 MByte to each slot and the *super slot space* allocating 256 MByte to each slot.

Through these mechanisms, a NuBus board can be accessed solely based on its slot location on the backplane.

References: [Apple 1987] and [Texas 1990].

A.6. SBus

SBus is developed by Sun Microsystems for use in their RISC-based workstations whereas SPARCstation 1 was the first. SBus is, unlike the other buses treated in this survey, not primarily intended to be a general backplane bus, but rather a direct chip-level interconnect scheme for the use on a CPU mother-board.

The main design goal was that high speed IO-devices, such as FDDI and Ethernet, could rely on the same high-performance, low-latency access to the memory that is available to the central processor. This IO-devices could then be implemented without large private buffers. Further, it was desirable to allow for a system clock of 20 to 25 Mhz, because this range would be compatible with fast-page-mode cycle times of 1- and 4-Mbit dynamic RAM's.

To achieve these goals, three main principles have been used as guidance to the SBus design:

- Synchronous operation with all timing related to the rising edge of the clock. Allowed clock speeds are the range of 16.67 to 26 Mhz.
- Active drive. All bus signals are actively driven to their inactive states before the drive is removed. Most buses simply remove the drive while the signals still are active and thereby let the termination network pull the signals to their inactive state.
- No driver overlap. Except from the open-collector (interrupt) lines, no signal is driven by the same source during the same cycle.

Other key features:

- Supports a memory management mechanism called *address translation*. For each transaction, a 32 bit virtual address is mapped into a 28 bit physical address by the system controller.
- 32 bit data path multiplexed with the 32 bit virtual address path.
- Dynamic bus sizing on a transfer by transfer basis.
- Seven interrupt lines, allowed to be asynchronous with the system clock.
- Geographical addressing.
- Peak burst transfer rate: 80 Mbyte/second with a 25 Mhz system clock.

A.6.1. Data transfer

A complete SBus, or DVMA (Direct Virtual Memory Addressing) cycle, can be divided into two distinct phases:

The **translation cycle**. A 32 bit virtual address is placed onto the multiplexed address/ data bus for exactly one cycle. The system controller translates the virtual address into a 28 bit physical address, then this physical address is placed onto the lines PA(27:0) and the Address_Strobe* signal is asserted.

The **slave cycle**. Data is now transferred between the master and the selected slave. The slave has up to 255 clock cycles to perform the requested transfer and issue an acknowledgement on the acknowledge lines Ack(2:0)*.

A.6.1.a. Bus sizing

A special SBus feature is the concept of dynamic bus sizing. This allows a master to initiate a word or half-word transfer to a slave device without regard to whether the slave supports a transfer size that large or not.

When the selected slave responds to a transfer by indicating bus (re)sizing through the code placed on the acknowledge lines, the requested transfer is automatically splitted into two or four bus cycles. Each bus cycle is then transferring a byte or half-word, depending on the slave acknowledgement.

A.6.1.b. Burst transfer

By using burst transfer, the master can transfer 2, 4, 8 or 16 words (32 bit) to the selected slave at a rate of one word pr. cycle. The length of the burst transfer is indicated to the selected slave through the data size lines Size(2:0). The slave must acknowledge each word transfer, failing to do so will insert waitstates until a implementation dependent timeout is reached.

> *Only 32 bit word transfers is allowed in burst mode, dynamic bus sizing is not supported.*

A.6.2. Arbitration

SBus arbitration is done by the arbitration module, centrally located on the system controller. Each SBus master has its own Bus Request (BR*) and Bus Grant (BG*) signal pair. A bus master indicates to the system controller that it wants the bus by asserting its BR* line, it is granted the bus when the system controller asserts the requesting bus master BG* line. Arbitration is done for each transaction on the bus, except from atomic operations.

Arbitration priorities are system specific. To ensure a certain amount of fairness among competing SBus masters, however, the most recent bus master is not allowed to use the bus again until all other masters who have requested the bus have been served. Within this constrain, bus requests need not necessarily be processed in chronological order.

A.6.3. Interrupt handling

Interrupts are supported by seven open-collector interrupt lines IntReq(7:1)*, by which SBus slaves asynchronously can signal the interrupt handler located on the system controller. By convention, IntReq(7)* is the highest priority interrupt, IntReq(1)* is the lowest.

Any SBus slave may assert one or more of the interrupt lines at any time. Upon asserting an interrupt, the slave must set a bit in an internal register, readable by the system controller, indicating that the slave is generating an interrupt at that level.

To identify the source of a generated interrupt, the system controller must therefore check all possible interrupt sources until it finds one with that particular bit set.

A.6.4. Multiprocessing facilities

Although SBus is not intended as a general purpose backplane bus, several multiprocessing capabilities are supported:

A.6.4.a. Bus locking

The SBus will be locked to the current bus master as long as the master keep its bus request BR* line asserted. BR* is released when the master receives bus grant for the very last cycle. If the accessed resource is not accessible through other channels than the SBus, this mechanism will ensure atomic operation on that resource.

A.6.4.b. Broadcast

Each SBus slot is connected to its own SlaveSelect* (Sel*) line, asserted whenever that slot is accessed as a slave. This allows individual and Parallel access of all slots, regarding the physical address presented on the SBus address lines PA(0:27) as a local address within each slot. Therefore, a completely individual and selective broadcast mechanism could have been implemented based on that scheme.

But: As mentioned earlier, each transfer requires acknowledgement on the Ack(2:0)* lines. These lines are not shared (open-collector) lines, broadcast is therefore *not* supported by SBus. That is, only one slave may be selected at any time.

A.6.4.c. Address translation

Because each master is identified to the SBus controller through which Bus Request* (BR*) line it is connected, separate translation tables for each master can be maintained.

A.6.5. Geographical addressing

The SBus is a geographically addressed bus. It is the responsibility of the system controller to decode the virtual address presented by the master into the appropriate SlaveSelect* signal and the physical address PA(27:0). Because the SlaveSelect* signal is unique for each SBus slot, the physical address can be regarded as local to the slot. The slaves therefore do not need to know their place in the global address space.

A.6.6. System configuration

For system (re)configuration purposes, each SBus board is self-identifying. Located at physical address 0, with size typically in the range of 4 to 128 Kbytes, each board has a PROM. This PROM contains board identification information in addition to optional driver software. This will allow the board to be used as a boot device or a display device during booting.

Reference: [Sun 1989].

A.7. TURBOchannel

The TURBOchannel is a high performance communication channel developed by Digital Equipment Cooperation (DEC) for use in their new generation, RISC-based workstations. A TURBOchannel system consists of one system module and some number of option modules. Key features:

- Synchronous, asymmetrical IO-channel. It is asymmetric in the sense that the system module must participate in all transfers, transfers can not be performed directly between two option modules.
- System clock in the range of 12.5 to 25 Mhz, selected to be compatible with fast-page-mode access of the fastest DRAMs.
- Multiplexed 32 bits address/data bus.
- Max. burst transfer performance: 100 Mbyte/second.
- Synchronous handshake, slow option modules may insert waitstates.

A.7.1. Data transfer

The TURBOchannel supports two types of data transactions, I/O and DMA (Direct Memory Access).

A.7.1.a. I/O transactions

An I/O transaction is used when the system module performs a single word read or write to an option module. Data is then addressed and accessed as a 32 bit word, with a 4 bit "byte enable mask" permitting selective byte write within the word. The addressed option module is acknowledging the requested operation by two handshake signals, indicating whether the operation was successful or not.

I/O modules are addressed geographically. That is, every option module is connected to the system module by its own dedicated "option module select" line SEL*. The transaction address presented by the system module on the multiplexed address/ data bus can therefore be regarded as local to the selected option module. The address range for I/O transactions is 512 Mbyte, with 4 address/ data bus lines used for the byte enable mask.

A.7.1.b. DMA transactions

DMA transactions are option module read or write onto the system module. They are block transfers in units of 32 bit words only, no byte enable mask occupies address signals on the bus and the address space is therefore larger than for I/O transactions, 16 Mbyte. The transfer can be of any length from 64 words up to an implementation-defined limit, but must, however, always be a power of 2. To obtain maximum performance, data is transferred by one word each cycle. Once the DMA transaction is established, a new data word must therefore be supplied (read) and accepted (write) each cycle. There is no handshake mechanism available at a word-by-word basis within a DMA transaction.

The length of the transfer is not signalled before the transfer starts, but as an "end-of-transfer" signal from the option module during the last word of the block transfer. Together with the missing handshake possibility within the transfer, this requires the system and option module capabilities as far as block transfers are concerned to be closely matched.

Due to the speed, DMA transactions are usually done between FIFO's or directly to/from high speed dynamic RAMs, utilizing the fast-page access mode. In that case, the DMA transactions must not cross the byte boundaries of the DRAMs.

A.7.1.c. Broadcast

Broadcast transfers are not supported neither by the I/O nor the DMA transaction modes. This is in correspondence with the design philosophy behind the TURBOchannel: To be a high performance point-to-point communication link (like a port) rather than a general purpose multiprocessor bus.

A.7.2. Arbitration

The TURBOchannel is no symmetric, multimaster bus, the arbitration mechanism is therefore very simple.

As far as I/O transactions are concerned, these are initiated by the system module. There is only one system module in the system, and no arbitration is therefore necessary. However, the addressed option module may be occupied with a DMA transfer thereby being unable to service the I/O transfer request from the system module. This will be signalled by the option module by asserting a dedicated signal line (CONFLICT*).

In contrast, DMA transactions may be requested by any one of the option modules. Two signal lines are provided for channel arbitration, Read REQuest* (RREQ*) and Write REQuest* (WREQ*). They are processed by the system module, granting access by asserting the signal line ACKnowledge* (ACK*). Every option module has its own set of WREQ*, RREQ* and ACK* lines. The option module indicates the length of the block be the number of cycles that it continues to assert signals WREQ* or RREQ* after signal ACK* is asserted by the system module.

The system module is in full control of which option module requests the channel for what operation. Which arbitration philosophy (in terms of fairness, priority etc.) to use, is therefore solely up to the system module and is implementation dependent. The TURBOchannel hardware specification does not prescribe anything specific as far as this is concerned.

A.7.3. Interrupt handling

Every option module has its own level-sensitive interrupt (INT*) line. Once asserted, the option module may not release INT* until the interrupt condition is dismissed by software. Slot priority is not a part of the TURBOchannel specification but is implementation dependent.

A.7.4. System configuration

System configuration is supported by a ROM on every TURBOchannel option module. This ROM contains formatted information about itself and various board-specific parameters describing the option module. It may also contain additional option module firmware.

Reference: [Digital Equipment Corporation 1990].

A.8. Futurebus+ (IEEE 896)

In 1983, the *IEEE P896 Futurebus Working Group* was formed, aimed to develop innovative technology and protocols for a scalable multiprocessor bus. As work progressed, it draw attention from other bus specification work groups which partly joined (Rugged Bus, 1988) and partly supported The Futurebus+ work by basing their own bus extensions on the Futurebus+ specification. This is the case for VITA (VME International Trade Association, 1988) and Multibus Manufacturers Group (1989). The work has also been strongly influenced by the US Department of Defence's (Pentagon) selection of Futurebus+ as the basis for all future US Navy mission critical computers (1988).

According to Dr. Paul Borrill, chairman of the IEEE Futurebus Committee and director at SUN Microsystems:

> *"Futurebus+ takes its name from its goal of being capable of the highest possible transfer rate consistent with the technology available at the time the modules are designed, while ensuring compatibility with all other modules designed to this standard before and after. The plus sign (+) refers to the extensible nature of the specification, and the hooks provided to allow further evolution to meet unanticipated needs of specific application architectures."*

Key features:

- Scalable data bus widths from 32 up to 256 bits.
- Transfer protocols supporting single data transfers as well as high speed block transfers, handshaked or not handshaked.
- Multiplexed address/data lines.
- Fully distributed arbitration scheme.
- Parity protection on all lines.
- Multiple level of bus locking and mutual exclusion support.

A.8.1. Data transfer

Each Futurebus+ data transaction, regardless of the selected transfer mode, consists of three subsequent phases:

During the opening **connection** phase, a connection is established between a master and one or more slaves. All slaves take part in the connection phase, which is a fully handshaken broadcast operation. In the connection phase, the master provides information about the following data transfer to the slaves (transfer protocol, data size etc.). It also allows the slave(s) to return status information to indicate their response to the requested operation.

The connection phase is followed by the **data transfer** phase. In this phase, the actual data transfer is done between the master and its connected slaves.

The transaction is terminated by a **disconnection** phase, the connection between the master and the connected slaves is then broken. Breaking the connection must not be confused with relinquishing bus mastership: After terminating a transaction, the master module may retain mastership of the bus and begin new transactions with the same, or other slaves on the bus, without having to perform a new bus arbitration.

For the data transfer phase, the Futurebus+ specification supports two fundamentally different data transfer protocols, compelled and non-compelled.

A.8.1.a. Compelled mode

The compelled data transfer protocol is a basically asynchronous protocol where the selected slave(s) is compelled (obliged) to provide a response before the master is allowed to proceed to the next transfer.

The Futurebus+ specification supports three compelled transaction types: Address-only, single address and block transactions. The three transaction types are distinguished by the number and the direction of the transferred data.

In the **Address-Only** transaction, an **address** is transferred from the master to one or more slaves. There is no subsequent data transfer phase, hence no data are transferred. A typical example of use of this type of transactions are system event messages broadcasted to a number of modules (slaves).

To transfer one or more items of data to a single address, the **Single Address** transaction should be used, It allows the flexibility for each transfer to be in a different direction, useful implementations of this feature can therefore be operations like read-modify-write and write-read-verify. If the Futurebus is the only path by which the addressed slave can be accessed, these operations will be indivisible.

The **Block Transfer** transaction mode is highly efficient and provides the maximum performance, but restricts the transfers to the same direction - either all read or all write. In this mode, only the start address is set up via the bus, the addressed slave(s) must keep track of the current address (if necessary) by incrementing its local address counter or load the data into a FIFO. The length of the block transfer is not told to the slave(s) prior to the transfer, but is determined by the actual length of the transfer. A slave may, however, return an *end-of-data* status to the master to indicate that the slave will be unable to accept the next requested data transfer. Because the addressed slave(s) does not know the length of the block transfer in

advance and the unaddressed slaves do not track block transfers in which they do not participate, a block transfer should never cross module (slave) boundaries. In that case, data intended for other than the first module will be lost.

These three transaction modes may be orthogonally combined with three different handshake modes, single-slave, broadcast and three-party, yielding a total of 9 possible transfer protocols.

Single-Slave Handshake mode. Mostly, slaves will have a unique address range in the system and only one slave will then participate in the transaction. The data transfer will then proceed at a speed that is limited by only the master and the slave itself.

Broadcast Handshake mode. If several slaves are mapped into the same area of memory, all these slaves will participate in the transaction, and thereby respond together, when this area is accessed by the master. To make handshaking from multiple slaves possible, an additional inverted acknowledge signal line (AI*, address Acknowledge Inverse) is included. This handshake mode is used with broadcast (write) and broadcall (read). On broadcall, several slaves drive the data lines simultaneously. This is possible due to a "open-collector like" connection to these lines.

Three-party Handshake mode. This type of transaction is used in copy-back caching systems when data in the selected slave may be incorrect and a caching module is responsible for providing the correct data. A three-party transaction involves one master and two slaves and can be of two types, intervention and reflection.

An intervention occurs when an unselected slave (in a single-slave transaction) inhibits the selected slave and replaces the selected slave by itself for the rest of the transaction. Intervention must be done during the address transfer phase of the transaction.

Alternatively, the unselected slave can become a reflecting slave, causing the data on the bus to be written (reflected) into the selected slave itself. Even though the master may be reading data from the reflecting slave, the selected slave interprets the transaction as a write transfer, and stores the data.

Intervention and reflection can be combined: First, the intervening slave provides the master with correct data, then it updates the selected slave (the data is already on the bus) by reflecting the data into the slave.

A.8.1.b. Non-compelled mode

For high-speed block transfers, broadcasts and broadcalls, an optional non-compelled transfer mode is supported. As indicated by the name, no handshake from the selected slave(s) are required in this mode. The data placed on the bus is not made synchronous to a system clock, but synchronized by an own sync signal, always emitted by the same module emitting the information (data) signals. This mode of operation is called source-synchronization. Because there is no return handshake, this is truly a packet-type transfer mechanism. All transfers must be of fixed length, and status is returned only at the end of each transaction. All modules supporting this mode may monitor all data being transferred over the bus.

A.8.2. Arbitration

Basically, Futurebus+ uses the same distributed parallel contention arbiter scheme as Multibus II and NuBus. No central arbitration controller is involved, modules compete for the bus by presenting a unique arbitration number onto the 6 bit open-collector arbitration bus. In case of simultaneous requests from other modules, all modules not having the highest arbitration number will detect a mismatch between their own number and the number on the bus lines and eventually withdraw their request. In this way, the module with the highest arbitration number will gain bus mastership and no central controller is needed to resolve arbitration conflicts. The arbitration operation takes place on dedicated signal lines and is therefore executed in Parallel with data transfers.

As far as arbitration is concerned, Futurebus+ modules can be of two types, *fairness* and *priority*. The type of module is distinguished by the most significant bit in its arbitration number. Fairness modules have this bit set to 0 while priority modules have this bit set to 1. Thus priority modules always have higher arbitration numbers than fairness modules, a priority module will therefore always win an arbitration where the two classes are competing against one another. All arbitration numbers must be unique, to ensure this it is recommended to use the 5 bit geographical address as the 5 least significant bits of the arbitration number.

To avoid starvation of modules with low arbitration numbers, a fairness module relinquishing bus mastership is not allowed to compete for the bus again until no other modules are seeking control. This is the reason for using the term "fairness". Priority modules are not subject to this restriction, they may request for the bus whenever they like. All modules may dynamically reclassify themselves as fairness or priority modules, depending on the current operation. Ordinary data transfers should use the fairness mode while the exchange of synchronizing information could use the priority mode.

A unique feature of the Futurebus+ arbitration scheme is its optional idle bus approach, used when there is only a single board requesting mastership. In the idle-bus mode, the system uses the address/data lines to signal a request when the bus is idle. This minimizes latency for a module to access a memory subsystem. If two or more masters are detected during this request phase, the scheme automatically defaults to the parallel contention arbiter method.

A.8.2.a. Arbitration messages

A full Futurebus+ backplane consists of 21 modules. With a 6 bit arbitration number, this means that there will be 22 unique numbers "left-over" after assigning each module its own fairness and priority arbitration number. Because all modules can observe all requests on the arbitration bus, these spare numbers can be utilized as an efficient way of sending "messages" to other modules in the system, the message being the number presented on the arbitration bus. The interpretation of these messages are system specific, the only one defined in the Futurebus+ specification are number 63 (all ones), signalling power fail. The arbitration procedure sending an arbitration message is performed exactly like an ordinary bus arbitration, the only difference being that the module issuing the message aborts the arbitration instead of taking bus mastership when it has won the arbitration contest.

In this way, it is possible to communicate with other modules, for instance for synchronization purposes, without using dedicated lines or using the system data bus.

A.8.3. Interrupts

The Futurebus+ has no dedicated interrupt lines. Instead, two other mechanisms are used for signalling events. The first is the already described arbitration, or emergency, messages. These are conceptually not unlike ordinary shared interrupt lines: When first triggered, the interrupt signal can be observed by all modules while the sender remains anonymous. No data is included in an emergency message, the only piece of information is the emergency (arbitration) number itself.

The other mechanism is the targeted interrupt, that is, the interrupt is targeted for a specific destination (module). These are sent like ordinary write transactions, but to a specific address on a single module. Futurebus+ reserves 32 locations in each module's *Control and Status Register* (CSR) space, addressed by the module's slot location (geographical addressing), for this purpose.

A.8.4. Bus locking

In a single bus system where the modules are accessible only through their Futurebus+ interface, indivisible operations are automatically guaranteed because the current master module has always exclusive control over the bus. It can therefore carry out whatever indivisible operations necessary before releasing the bus.

With a multiple bus system, however, it is not sufficient to lock the bus to ensure indivisibility, but the module (slave) itself must be locked to prevent accesses via other channels than the Futurebus+. This is done by a dedicated Futurebus+ lock (*LK) signal: A slave accessed with this signal set, is not allowed to accept accesses via other channels until it is accessed again with the lock signal resat, or the master which issued the lock relinquishes bus mastership.

A.8.5. Geographical addressing

The Futurebus+ address space is divided into two areas. The majority of the address space is ordinary memory address space, accessible through the various types of available data transactions. The top 32 Mbytes, however, are reserved for system control and configuration purposes and is called the *Control and Status Register* space (CSR).

This block of 32 Mbytes is in turn divided into 256 blocks of 128 Kbytes each, called *bus space.* These blocks are primarily intended for use in systems connecting multiple Futurebus+ crates via bus repeaters. One block is then assigned to each crate by a unique number. Of the 256 bus space blocks, two are of special interest: Bus space 0 maps into the CSR of every backplane, allowing system-wide broadcasts of control information. On the other hand, bus space 255 always maps into the local backplane, making it un-necessary to assign a backplane number in case of a single crate system.

Each bus space block is in turn divided into 32 *module space* blocks of 4 Kbytes each. These 32 blocks are addressed through the Futurebus+ geographical addressing scheme: Connected to each slot of the backplane is a 5 bit switch, providing a unique, location-determined address for

that slot. Every slot in a Futurebus+ crate will then have a geographic address in the range of 1 to 21 (the maximum number of slots in one crate), by which its CSR module block can be accessed. As for bus space blocks, module space block 0 maps into the CSR of every module and is reserved for broadcast transactions (read or write) to the equivalent locations of every module.

Each module space block contain module identification information, 32 event registers used for targeted interrupts and 32 addressable switches. The addressable switches are used to control some basic module functions and replaces the more conventional jumper fields used for this purpose. This allows dynamic module (re)configuration without powering the system down.

Reference: [IEEE 1987], [Taub 1984].

A.9. SCI (IEEE 1596)

In the very near future, the limiting factor to performance in a multi-node, parallel processing system will not be the processing capacity of the individual nodes itself, but the rate by which they are able to communicate. To support a shared memory system consisting of 100 M-Flops processors, each processor (node) must be capable of transferring data at a rate of at least 1 GByte pr. second. Given the physics (distributed capacitance and the speed of light) and the once-at-a-time nature of a backplane bus, obviously other communication schemes must be investigated.

In November 1987, an IEEE working group was initiated to process these thoughts further into a specification for a high performance communication system. The group was named *Superbus* and was formed under the IEEE Computer Society's Microprocessor Standards Committee. Most of the members joining the group came from other high-speed bus design activities as Fastbus (IEEE 960) and Futurebus (IEEE 896). The work in the Superbus group, which eventually was leading to the Scalable Coherence Interface (SCI) specification, had the following design goals:

- Support transfer rates of 1 GByte pr. second with the capability of scaling upwards to meet future needs.
- Allow complete interconnection among all types of system modules: Processors, memory and I/O nodes.
- One up to an "infinite" number of nodes.
- Provide for modular growth, thus reducing processor and system cost.

To achieve these goals, performance far beyond that of buses and backplanes was needed. Rather than using bused backplane wires, SCI therefore employs a point-to-point interconnection technology based on uni-directional, optical fibre links. Data are transferred as packets using a requester/ responder transaction scheme, handshaked by a packet protocol. Due to the problem of routing broadcasts efficiently in a complex network, the concepts of broadcast transactions and eavesdropping third parties are abandoned. Instead, the cache coherence protocols are based on directed point-to-point transactions. Other key features are:

- Up to 64 K processor, memory or I/O nodes.
- Address width: 64 bits.

- Data width: 64 bits (logical) and 16 bits (physical).
- Multiplexed protocol.
- Driver technology: ECL.

References: [Warren 1990], [James 1990] and [Borrill 1989].

B. Parallel contention arbitration

The method of parallel contention arbitration has been very popular during the last few years and is adopted by a number of major bus standards like MultibusII, NuBus and Futurebus. Its first applications, however, was in the IEEE Standard 696 - Interface Devices and in the US Department of Energy's Fastbus.

Background

In a multi-master bus system, the modules connected to the bus can be divided into two categories, masters and slaves. As the name indicates, *slaves* only play a passive role during bus transfers, they are incapable of initiating bus transactions themselves. A typical example of a slave module is a memory board, being written to or read from as a result of an initiative taken by another module. This other module will be a bus *master*. Several modules (from now on called Potential Master modules, PMs) being connected to the same bus may have master capabilities, a mechanism is therefore needed to ensure that any conflicts due to simultaneous requests are resolved in an appropriate manner. This is the task of the system's *arbitration* mechanism. Such a mechanism can be implemented in many ways, distinguished by parameters as

- Centralized or distributed control.
- The number of priority levels.
- The type of arbitration protocol.

One possible implementation is the *parallel contention arbitration* scheme described here.

Functional description

In a parallel contention arbitration system, each potential master is connected to the bus by the following open-collector, active low signal lines:

ARBITRATE

The arbitration qualification (strobe) line. When asserted, it tells all PMs that an arbitration procedure is in progress.

ARB(0 - (n-1))

A number of lines eventually signalling the identity of the PM winning the arbitration contest. Values:

0 - unasserted (high voltage).

1 - asserted (low voltage).

Every PM is allocated one or several unique n-bit arbitration numbers, *arb(0 - (n-1))*. When a PM wants to take control of the bus, it asserts the ARBITRATE signal line and outputs its arbitration number *arb* onto the bus arbitration lines *ARB*. The basic algorithm is then as follows:

> *All PMs taking part in the arbitration contest will now compare, bit by bit, the value of its own arbitration number arb against the state of the arbitration bus lines ARB. If, for any bit k, arb(k) is equal to 0 while the corresponding bus line ARB(k) is equal to 1, a PM will withdraw all its lower order bits (arb(0) to arb(k-1), both inclusive) from the arbitration bus lines ARB(0) to ARB(k-1).*

When the ARB bus lines eventually settle, there will remain one and only PM module, having an exact match between its own arbitration number arb and the state of the ARB bus lines. This module is the winner of the arbitration contest. As shown by [Taub 1984], the maximum delay to reach a steady state for a system consisting of 2^n PMs is n/2 end-to-end bus propagation delays, plus a small amount of delay for the monitoring logic. One possible implementation of the described algorithm is shown in Figure B.1., taken from [Taub 1984]'s description of the arbitration mechanism in the IEEE 896 Futurebus. An absolute demand as far as the design of the arbitration mechanism is concerned, is that it is purely combinatorial. That is, it must contain no feedback paths. The reason for this is that the value present on the arbitration lines at any time must depend only on the bits being applied to them and not on any past history.

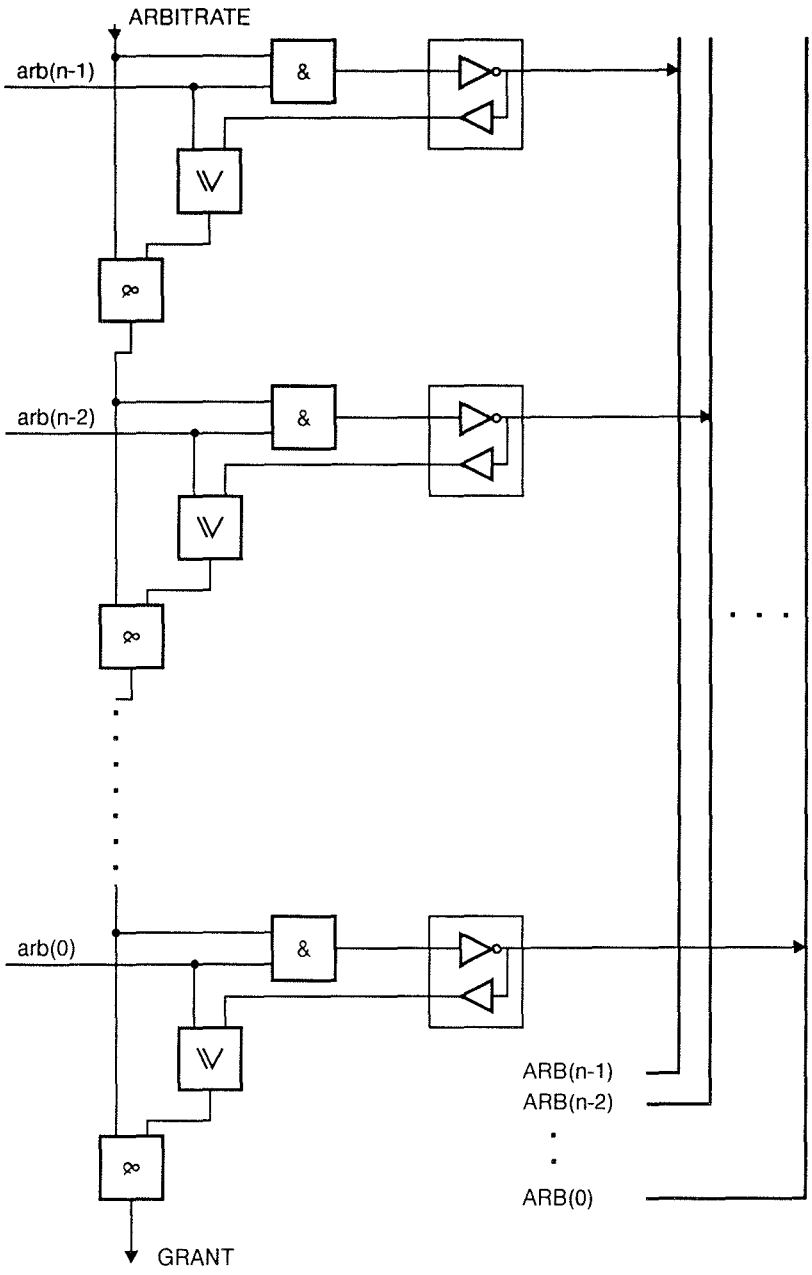


Figure B.1. Arbitration mechanism

Arbitration protocols

With a n bit arbitration number, there is a total of 2^n-1 valid arbitration number combination (the algorithm will not work with an arbitration number equal to "0"). This pool of numbers can either be allocated as a single number to each module, alternatively may each module be assigned several numbers, distinguished by the value of the most significant arbitration number bit(s). Because bus access is granted strictly on basis of the numerical values of the arbitration numbers involved in the arbitration contest, low/ high priority arbitration may be implemented by varying the higher order arbitration number bit(s).

An obvious danger with an arbitration protocol as described, is that frequent requests from high numbered (priority) modules may cause starvation of lower numbered modules. To avoid this, the bus modules are often made subject to the following restriction as far as issuing several, subsequent bus requests is concerned: A module winning the arbitration contest is not allowed to issue a new request until all requests issued within the same *batch* as its previous request have been serviced. A request batch is here defined as the period of time from the arbitration process is initiated by the first request being issued to an idle bus until the arbitration contest is terminated and the winner is taking control of the bus.

Compared to a more conventional, centralized arbitration scheme, parallel contention arbitration has the following advantages:

- It requires very few wires on the bus to carry out the arbitration algorithm.
- Priority scheduling of urgent requests can easily be integrated with mechanisms for fair scheduling of non-priority requests.
- The state of the arbiter is available and can be monitored by all modules on the bus.

In addition to being useful for diagnosing system failure, the latter point also makes it possible to use the arbitration bus for the broadcast of *emergency messages*. This is done by reserving a subset of the total pool of arbitration numbers to be interpreted as message rather than module identifiers. When an arbitration procedure is performed using one of the reserved numbers, the module issuing the number (e.g. the system controller) is not requesting the bus. Instead, the "arbitration procedure" is actually a message being broadcast to other modules connected to the bus arbitration lines, with the information contents of the message laying in the specific arbitration number by which the arbitration procedure is performed.

C. Ring bus system nomenclature

A nomenclature explaining all vital words and expressions used throughout the ring bus specification part of this thesis is presented. The included items are alphabetically sorted, as well as organized according to their subject, listed following a more or less logical sequence within each subject group. The subjects themselves are alphabetically listed. An overview of the main system components and how they are related are given in Figure C.1..

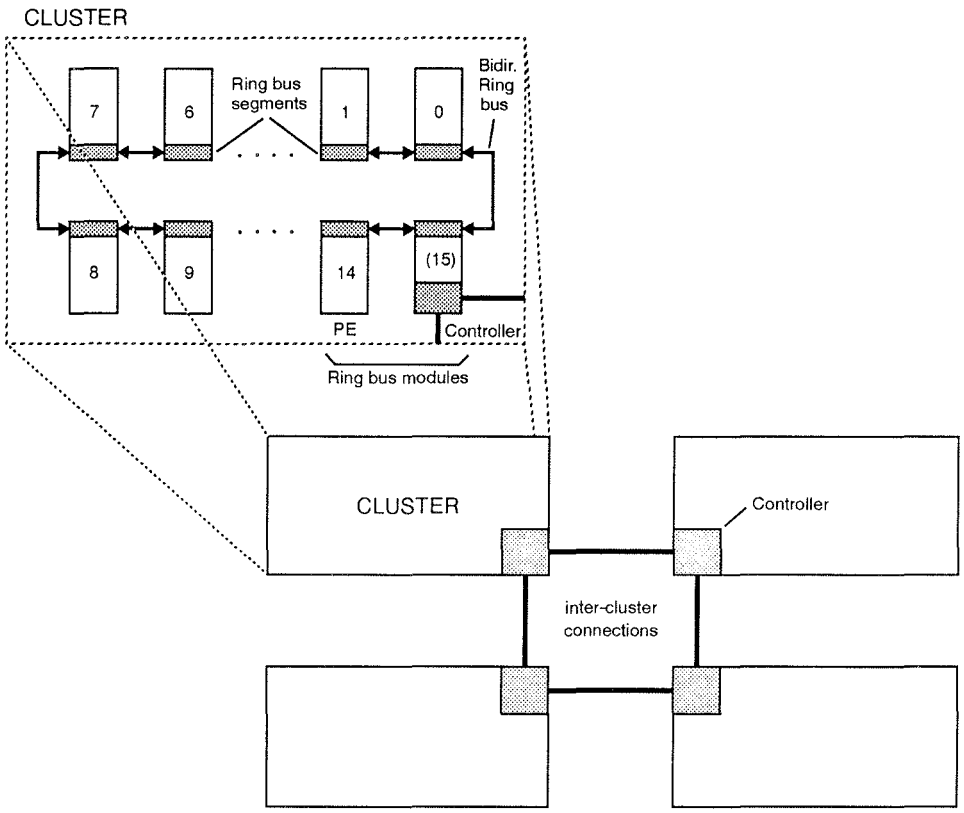


Figure C.1. Main system components

C.1. Alphabetical index

```
SOCP                          : Transfer entities
SODP                          : Transfer entities
Segment busy/ready            : Ring bus terminology
Segment select                : Ring bus terminology
Segment unselect              : Ring bus terminology
Slave controller              : Inter-cluster communication
Source module                 : Ring bus terminology
Tag                           : Transfer entities
Total system                  : System components
Transfer grant                : Ring bus terminology
Transfer path                 : Ring bus terminology
Transfer reject               : Ring bus terminology
Transfer release              : Ring bus terminology
Transfer request              : Ring bus terminology
Transfer request message      : Ring bus terminology
Wrap-around transfer          : Ring bus terminology
```

C.2. Listed by subjects

ADDRESSING ISSUES:

PE address

A PE's (local) address corresponds to its position along the ring bus and will be in the range of 0 to 14 (both inclusive). The same address is used for both left and right ring bus transfers.

Local

Address within the cluster.

Remote

Another cluster (than in which the source module is located).

controller address

The controller's (local) address is always 15.

cluster address

The total system can have up to 16 clusters, the cluster address will therefore be in the range of 0 to 15.

mask

A 16 bit value where each bit corresponds to the module or cluster with address equal to the bit's position (0 to 15) within a 16 bit word. All 16 modules in a cluster, alternatively all 16 clusters in the system can then be enabled/ disabled through a single 16 bit value.

INTER-CLUSTER COMMUNICATION:

Inter Cluster bus

Bidirectional link connecting two neighbouring clusters.

master controller

The controller currently in control of the inter cluster bus.

slave controller

The controller currently not in control of the inter cluster bus.

INTRA-CLUSTER COMMUNICATION:

Cluster Control bus

Shared bus connecting all ring bus modules. Consists of three buses: The Arbitration bus, the Address bus and the Module Control bus. Additionally, a number of related signal lines is also a part of the cluster control bus.

arbitration

The process of selecting one out of (potentially) several modules simultaneously competing for a common resource of some kind or another.

Arbitration bus

Signal lines implementing the arbitration mechanism used for getting access to the Address bus.

emergency message

5 bit, high priority broadcast message transferred on the Arbitration bus. Typical use are for signalling urgent events as power fail and system reset.

Address bus

Used for transferring the address information needed to set up a Ring bus transfer.

Module Control bus

Each PE module is connected to its local controller through its own dedicated signal line. This line is bidirectional and is used by the controller for giving commands to as well as reading status from the individual PEs. The current function of this line is determined by the state of three associated shared signal lines.

RING BUS TERMINOLOGY:

bus

A number of signal lines functionally to be regarded as an entity, transferring control or data information.

Ring bus

Interconnection network built from unidirectional, point-to-point bus links. Each ring bus module has an input port and an output port, the output port of module "n" being connected to the input port of module "n+1". Coming to the end, the output port of the last module is looped back and connected to the input port of the first.

bidirectional ring bus

Double ring bus system, permitting simultaneous transfers in both directions, left and right.

right transfer direction

Transferring data from ring bus module "n" to module "n+1".

left transfer direction

Transferring data from ring bus module "n" to module "n-1".

wrap-around transfer

A left direction transfer where the address of the last destination module is greater than the address of the source module, or a right direction transfer where the address of the last destination module is less than the address of the source module. That is, when the ring is visualized as a linear array of modules 0 (extreme left) to N-1 (extreme right), with modules 0 and N-1 connected together to form a ring, a wrap-around transfer will "go around" the ring transferring data between modules 0 and N-1 in either direction.

non wrap-around transfer

Transfers not including the transfer of data between modules 0 and N-1, in any direction.

Ring bus module

Processing Element (PE) or controller.

Ring bus segment

The mechanism for transferring Ring bus data from a module's input port to its output port. Each Ring bus module has two Ring bus segments, one for left transfers and one for right transfers.

segment busy

The Ring bus segment is currently taking part in a Ring bus transfer. This transfer can be a pure forwarding of data received from the preceding Ring bus module, or, if the module is selected as destination for the data, the data is copied into the module's local buffer before it is transferred to the next module. In both cases the segment is "busy" as far as other transfer requests are concerned.

segment ready

If a segment is not busy, it is ready (to take part in a Ring bus transfer).

segment select

The process of allocating a ready segment to a requested transfer, moving the segment to the busy state.

segment unselect

The process of de-allocating a busy segment, moving it to the ready state.

module busy

A module is busy if it is unable to receive new data. This will be the case if the module is still occupied with processing (old) data and no more local buffer space is available to buffer incoming new data.

module ready

If a module is not busy, it is ready (to receive new data).

NOTE. Beware of the difference between segment busy and module busy: *Segment busy* is related to Ring bus transfers and means that the module's Ring bus interface (left or right) is blocked as far as participating in new Ring bus transfers are concerned. *Module busy*, however, means that the module is unable to accept new data for local processing. The module's Ring bus (left and right) segments, however, may well be ready and available for data forwarding to other Ring bus modules.

transfer path

All Ring bus segments involved in performing a particular Ring bus transfer. All these segments (left or right) will be busy during the entire transfer.

neighbouring module

The next module in the transfer path, that is, the module to the immediate left or right of the module in question, depending on the direction of transfer.

source module

The module requesting and supplying the data in the transfer.

destination module

A module receiving (and possibly forwarding) data in the transfer.

last destination module

The destination module being the last along the transfer path. The last destination module removes the data from the Ring bus after storing it in its local memory, it does not forward the data to its neighbouring module.

transfer request

To perform a data packet transfer to one or several destination modules, a Ring bus module must first request the controller to use the Ring bus.

transfer request message

Message transferred on the Address bus requesting permission to perform a Ring bus data packet transfer to a specified set of Ring bus (destination) modules.

transfer grant

Permission issued by the controller to perform the requested transfer.

transfer reject

Permission to perform the requested transfer is denied by the controller.

pending transfer

A transfer requested, but not yet granted nor rejected.

pending transfer queue

The list of pending transfers.

transfer release

When the data packet transfer is finished, the transfer must be released. All Ring bus segments being allocated to the transfer are then unselected, thereby going from the busy to the ready state.

SYSTEM COMPONENTS:

PE

Processing Element. A module doing some kind of processing of the image data. The term *processing* is here to be interpreted in the broadest possible way, including various types of image filtering, image reformatting, statistical computations as well as feature extraction type of operations.

controller

A dedicated module executing control and service functions for the PEs in the cluster.

cluster

A collection of up to a maximum of 15 PE modules and one controller module. The PEs and the controller are interconnected by two unidirectional ring bus systems. One ring bus permits transfers in the right direction, the other in the left direction.

total system

A collection of up to 16 clusters.

local cluster

The cluster in which the source module of a data transfer is located.

remote cluster

Another cluster than in which the source module of the data transfer is located. The remote cluster is the destination of the data transfer.

local controller

The local cluster's controller.

remote controller

The controller of the remote cluster being the destination of the data transfer.

TRANSFER ENTITIES:

tag

Two extra bits accompanying the 16 bit Ring bus data values. The tag is used to identify the type of packet and to distinguish the last word in the packet from the rest of the packet.

packet

An entity of data to be transferred on the Ring bus. Each packet consists of a header part, containing information about the packet, and a data part. There are two types of packets, data and control packets.

local packet

Packet addressed to one or several destinations, all residing within the local cluster.

remote packet

Packet addressed to one or several destinations, all residing within a remote cluster.

data packet

Packet containing relatively large quantities of image or "general purpose" data. To transfer a data packet, access to the Ring bus must first be requested and granted by the controller.

SODP

Start-Of-Data-Packet. First word in data packet. Not explicitly tagged, but must be decoded by the module when receiving the first word tagged as a data packet.

EODP

End-Of-Data-Packet. Last word in data packet. Explicitly tagged.

control packet

Packet containing a small quantity of data intended for module synchronization and control. The transfer of control packets on the Ring bus does not require permission from the controller.

SOCP

Start-Of-Control-Packet. First word in control packet. Not explicitly tagged, but must be decoded by the module when receiving the first word tagged as a control packet.

EOCP

End-Of-Control-Packet. Last word in control packet. Explicitly tagged.

header

First part of packet, containing information about the packet. Depending on whether the packet is a local or a remote packet, the header consists of a local header only or a remote and a local header.

local header

Contains destination address information for addressing within the local cluster.

remote header

Contains destination address information for addressing a remote cluster. The remote header is removed by the remote controller before forwarding the packet onto the remote cluster Ring bus.

TRANSFER MODALITIES:

local broadcast transfer

Data or control packet transfer from one Ring bus module to all other Ring bus modules within the same cluster.

remote broadcast transfer

Data or control packet transfer from one Ring bus module to all other Ring bus modules in another cluster than in which the sending module is located.

global broadcast transfer

Data or control packet transfer from one Ring bus module to all other Ring bus modules in all clusters.

local multicast transfer

Data or control packet transfer from one Ring bus module to several other Ring bus modules (but not all) within the same cluster.

remote multicast transfer

Data or control packet transfer from one Ring bus module to several other Ring bus modules (but not all) in another cluster than in which the sending module is located.