Communications for Next Generation single chip computers[*]

David R. Smith[#] and Douglas Chan

State University of New York at Stony Brook

Abstract

It is the thesis of this report that much of what is presently thought to require specialized VLSI functions might instead be achieved by combinations of fast general purpose single chip computers with upgraded communication facilities. To this end, the characteristics of applications of this nature are first surveyed briefly and some working principles established. In the light of these, three different chip philosophies are explored in some detail. This study shows that some upgrading of typical single chip I/O will definitely be necessary, but that this upgrading does not have to be complex and that true multiprocessor-multibus operation could be achieved without excessive cost.

I. Example applications

Without doubt two of the most important computer applications of the coming decade will be graphics and speech processing. In turn, two principal characteristics of these applications are that the computation is often divisible into modular components and that they are often real time driven. Hence the requirement for speed which also points in the direction of specialized hardware. Examples may be found in the recent literature:

1. In a recent graphics processor [3], 12 copies of a specialized chip in combination are proposed to process graphic images for rotation, skew, translation, and various kinds of scaling and clipping. The computation rates of the component modules do not seem incompatible of the performance expected of next generation single chip computers.

2. A recent text-to-speech system [4], employs two general purpose processors feeding into specialized hardware. At the anticipated three character per second input rate, the second processor communicates data in packages of between 2 and 130 bytes every 10 milliseconds.

[*] Supported by General Instrument Corporaton.
[#] Visiting E.E. Dept., Stanford University, Spring 1981.

3. As can be seen from a recent survey, word recognition and continuous speech recognition systems [5], while still largely in the laboratory stage, are clearly developing along modular lines. The acoustic analysis module of one system [6] (itself done by FFT and divisible into modules), passes samples of 14 functions every 10 milliseconds on to the phonetic analysis module. Another [7], using software modules on a last generation mainframe, quotes ratios of cpu time to speech time of the following modules in series: Signal analysis 3:1, Spectral similarity 2:1, Region definition 2:1, Boundary placement 1:1.

It is to be noted that the information flow in all the above systems is unidirectional. As a contrast to this, a very common structure with the present generation single chips is to employ them as the slaves in a master-slaves configuration, eg. inside an intelligent terminal. In this case the communications are sometimes bidirectional. Another recent example of the master-slaves bidirectional configuration [8] uses special purpose processors to scan superimposed codes of a data base index.

All these applications are of course an illustration of the fact that parallel processing applications are emerging first in the fixed purpose realm. Although the ones mentioned here do not constitute a large number, they perhaps serve to give some idea of the features desirable in single chip computers which would serve them.

II. Working principles

In this section we set down what seem to be reasonable principles to be drawn at the outset concerning inter-chip communications in this context and some of the reasons for prefering them.

1. The processors will manipulate data in 16 bit words. In single chip computers the 16 bit word length will play for the next several years, and is sufficient for digital representation of analog quantities.

2. The number of pins per package is strictly limited and will not greatly differ from current practice.

3. Communications will be assumed to be word parallel. This is somewhat a choice of convenience if other considerations permit. Parallel ports are required anyhow for uniprocessor applications, and while serial communication could still be an added alternative, a decision to employ parallel communication will avoid the necessity for repetitive conversions through on-chip UART devices. In addition the use of the parallel ports would avoid further disparity in the estimated two orders of magnitude which separates the speed of communications internal and external to the chip [9].

4. There is no need of an address bus. Present designs for multibus architectures (eg. Multibus, Versabus, etc) are based on the arbitration of complete parallel bus structures, including the address bits [eg. 10].

In our context however this does not seem to be necessary. This is because we expect each communication (of parameters or results) between components of a modular computer structure to consist typically of a message or packet incorporating a number of words. This is shown in the applications described above. It has also been our experience with simulated applications for the Stony Brook Multicomputer, and in fact incorporated into the design of the kernel [11]. So if a whole package of words is going to a single destination, then an address bus would lie idle for all data words after the first. A destination address may as well be incorporated in the package header.

5. Bus connectivity: The next question to be addressed is how many distinct communication ports should emerge from a computer node. Clearly one is sufficient (eg. Ethernet), but in the present context not very interesting since the consequent restriction to a single time shared bus limits the concurrency possible. Performance is one of our concerns. If a condition were made that the processor internal bus must be available at the pins for testing purposes, then a minimum of 2 ports per node would be necessary, since distinct processor busses could not be directly interconnected. We will assume here that this is not the case,- that internal chip logic can be arranged so that the processor internal bus is available for testing at chip reset time, but after that a mode change can be effected which transforms that port into a buffered communication facility.

What then should be the connectivity of a node? Hardware for the X-tree project [12] is being constructed with 5 ports per node, although this would impact the pin resources of standard single chip packages. Also it is not clear that such a port multiplicity is necessary to be able to construct the network topologies considered in the X-tree literature, or to accommodate the bus loading of typical applications. This is because in fact, communication ports can be configured with appropriate control signals so as to be shared, and the apparent connectivities of modular computer graphs seen in the literature does not have to correspond to the number of bus ports coming off the chip. For example, the Stony Brook Multicomputer [11], though normally drawn as a graph having up to 6 edges to a node (fig. 1a), would, if implemented with single chip computers, be most conveniently partitioned so as to use only 3 port types per node (fig 1b). Simulation studies of typical applications [13] did not show bus loading as one of the limiting performance factors of the network.

Let us therefore consider the number of busses emerging per node. If this is two, then the maximum communication concurrency of the whole network is equal to the number of computing nodes (n), and occurs when they are all strung out in a simple pipeline as in figure 2. This assumes the use of buffered or DMA type ports able to operate independently of the CPU, otherwise, with program controlled ports, the maximum would be n/2. Of course it would also be less if some busses were incident on more than two nodes. Suppose now that the number of busses incident on a node is three. Then as far as the bus hardware is concerned the communication concurrency of the network might exceed the number of computing nodes. But

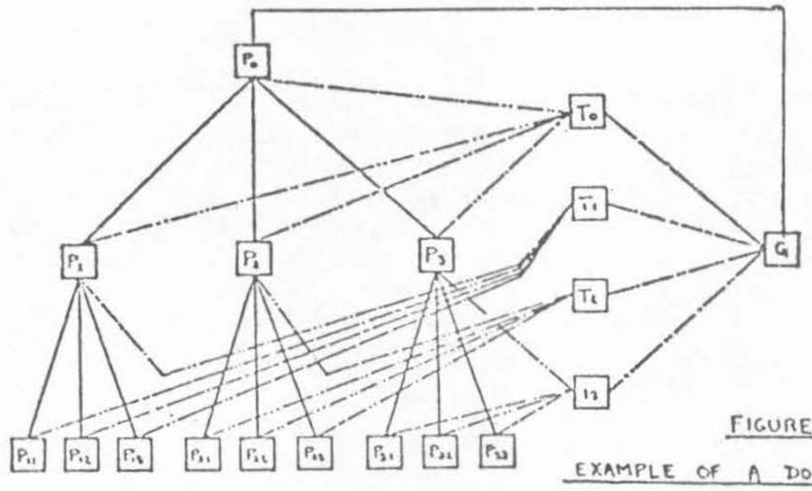


FIGURE 1(a)

EXAMPLE OF A DOUBLE TREE NETWORK (SUSB)

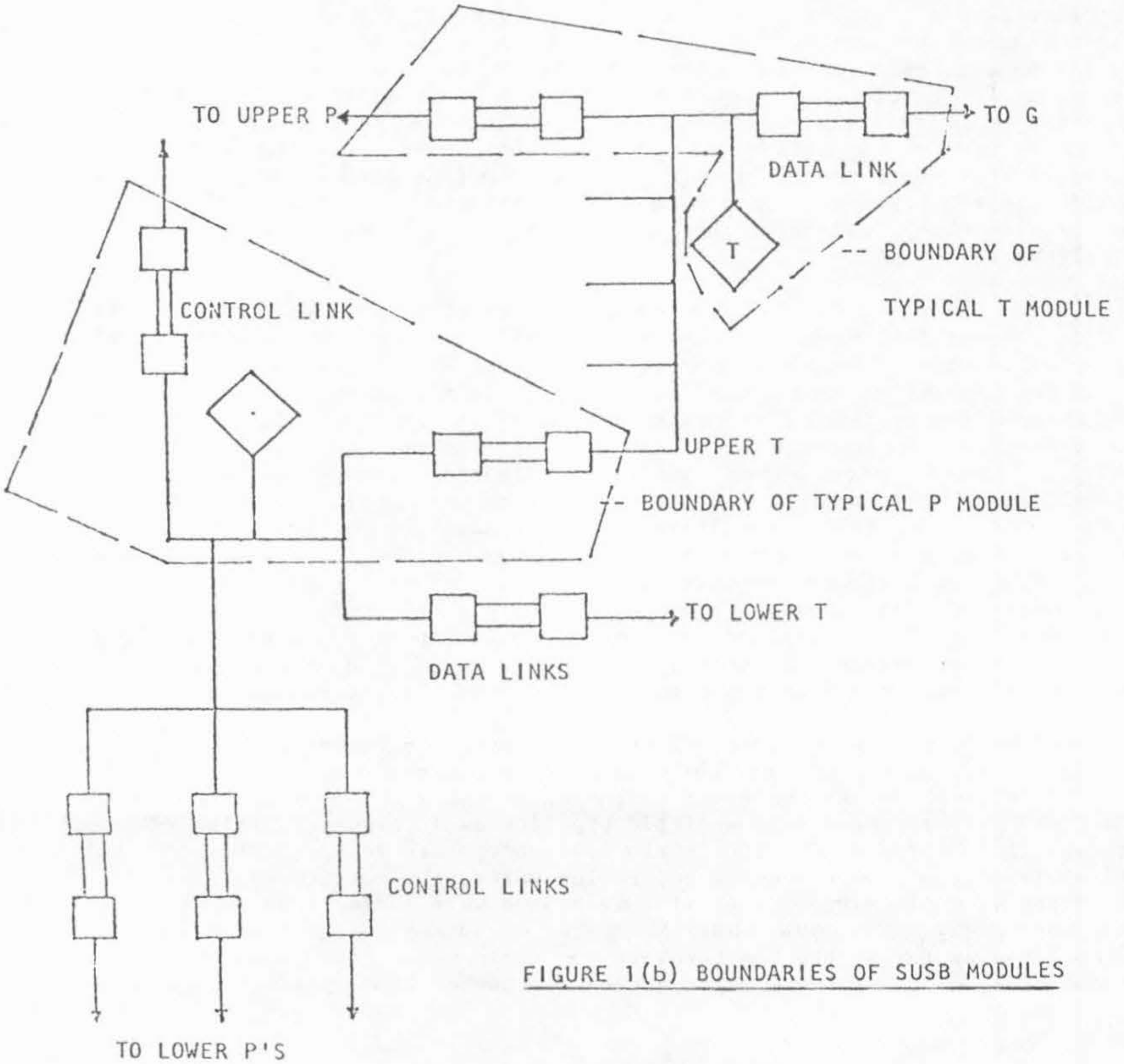


FIGURE 1(b) BOUNDARIES OF SUSB MODULES

unless the node computation is something rather trivial, it seems doubtful that results would be produced at a sufficiently rapid rate to justify anything like such a high communication/computation ratio. This supports the argument, already almost dictated by current pin limitations, that the number of bus ports should not exceed two. In paragraph 7 below, directionality arguments lead us to the conclusion that the number of ports should be exactly two.

6. Speed-up by pipelining and synchro-parallelism: For the internal structure of processors it is well known that there are two important recourses for when it is desired to achieve a computation rate wich exceeds the speed of the hardware available. These have been termed 'pipelining' and 'synchro-parallelism' [14]. They are illustrated in the context of n computer chip nodes in figures 2 and 3.

In the first, the serial combination, each computing node performs a part of the total computation and passes the intermediate results on to the next stage. In the simplest arrangement, the constituent computations all take the same amount of time, T, to complete. Then although a single computation still takes time nT, the results of repeated computations are streamed out at time intervals of T. If the constituent computations take differing amounts of time, then the technique may still be effective, but handshaking controls must be provided to make the faster components wait, and the overall results will be produced at intervals T_{max} corresponding to the slowest.

In the second, (fig 3), a parallel combination utilizes a staggered computation with input parameters and output results sequenced on shared busses. In the simplest form the constituent modules could be doing identical computations, or similar computations with different internal coefficients, and would therefore all take the same amount of time T. Again the effective computation rate is speeded up n times, in this case one result being produced every T/n secs. In a more complex case, the computations might take differing amounts of time, and then again the computation must be controlled to make the faster ones wait. In a more complex case still, serial networks of parallel stages might be indicated to more closely match the speeds of the pipeline stages, or alternatively, parallel networks of serial combinations, etc.

The important point is that the communication facilities that we propose should accommodate these likely to be encountered cases.

7. Bidirectional vs unidirectional ports: From the fixed applications literature we conclude that unidirectional communications are those most called for, although bidirectional communications are an important minority which must be accommodated. However, the bidirectional facility would be complex for the message based type busses we have outlined so far. It would require some type of semaphore system to control the

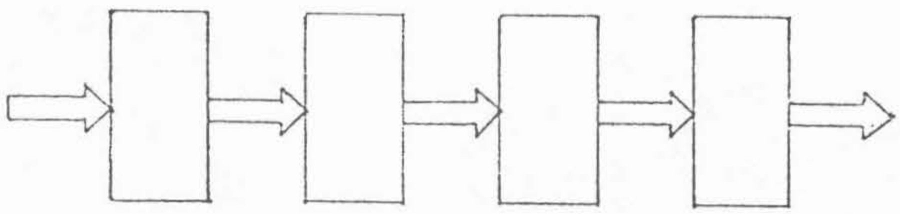


FIGURE 2 PIPELINE OR SERIAL CONNECTION

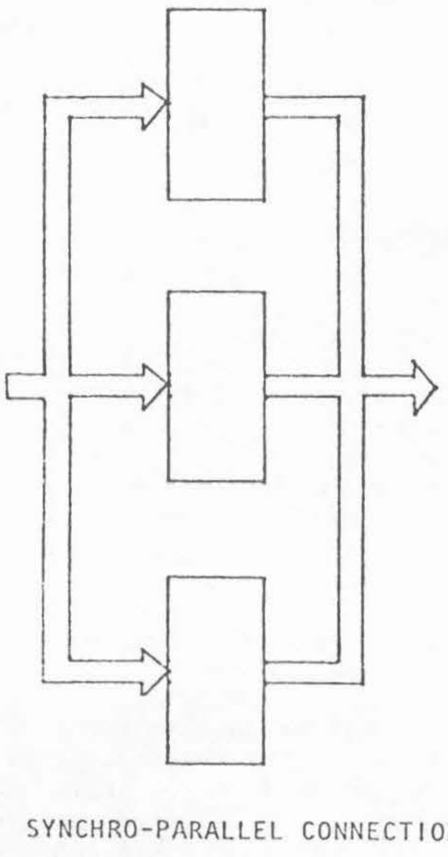


FIGURE 3 SYNCHRO-PARALLEL CONNECTION

intended bus master and direction of data transfer and to resolve collisions. Once established, communications on a fixed route and direction might proceed efficiently, but the protocol to change these would be a burden in terms of pins and time. The resulting overhead would then lie unused on what seems to be the majority of unidirectional communication applications. The alternative would be to employ a unidirectional transfer protocol on each port which was as simple in design as possible, as long as it could be shown that bidirectional applications could still be serviced (at the cost perhaps of using one or two extra chips). This will be the approach adopted here.

III Design for upgraded single chip computer,- version 1

A block diagram of a proposed design for an upgraded single chip computer is shown in figure 4 and will be used as a first model.

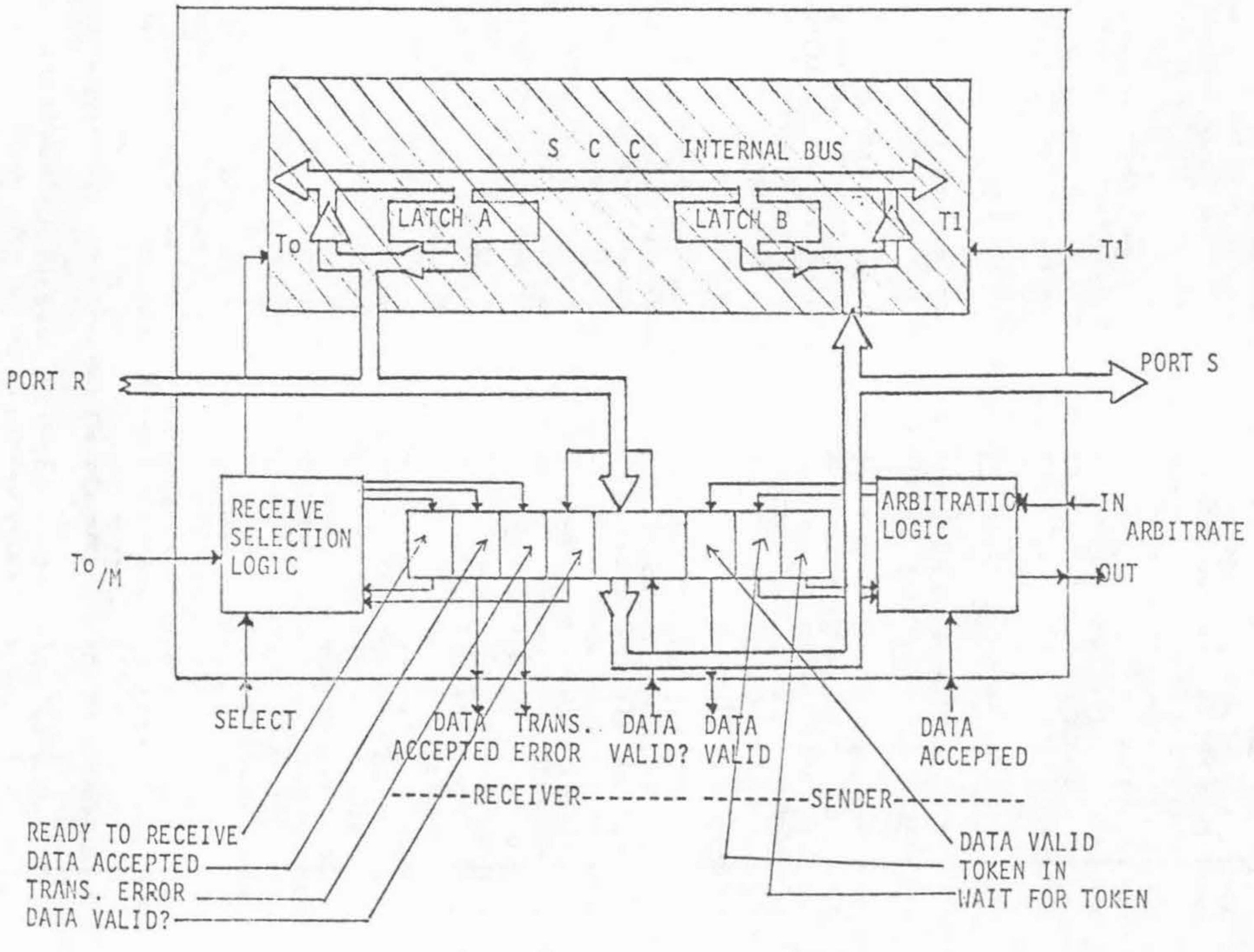
The large shaded block in the interior labelled SCC corresponds in function to a single chip computer of existing design. This will have the advantage that the design and debugging of this part has already been done and hopefully, partially amortized. As shown in our diagram we assume only that this part has two bidirectional I/O ports labelled A and B, and two interrupt inputs T0 and T1.

The additional logic circuit functions which give the new chip its upgraded capability are shown in the surrounding unshaded area of figure 4. Besides all the original pin connections to the SCC (which are led straight through), there are 8 additional pins for the added functions. This would increase the typical current 40 pin package to 48 pins which is not regarded as excessive. The added logic has three sections. Going from right to left in the figure, these are the arbitration control, control-status register, and the receive selection logic. The internal I/O ports are connected through to the external I/O ports S and R (for send and receive). Since the internal ports are bidirectional, while the external ports are unidirectional, we can utilize the remaining two functions to read and write the new control-status register with the existing SCC instruction set as shown. The need for the new arbitration logic arises because we are here absorbing the functions of the separate bus arbitration chip of existing commercial designs. It will operate on a simple daisy chain principle and be used chiefly for arbitration of the send functions. The receive selection logic is responsible for enabling the communciation pathway leading to this chip from the send port of another. When the module is not so selected, its receiver port and handshake lines should be in the high impedance state. When the READY FOR RECEIVING flag is set, and the SELECT input is active, the receiver selection logic will interrupt the SCC using pin T_0, which will invoke the attention of the communications kernel.

In two matters concerning this design we have been a little conservative, but these could presumably be adjusted later as the technology permits. Current instruction sets typically include a repeat function, which as applied to I/O, could be capable of streaming words at

UPGRADED SINGLE CHIP MICROCOMPUTER

FIGURE 4



an exceedingly rapid rate. Although we envisage multiple computer chips in close proximity, say on the same board, we are unsure that this rapid rate could be correctly synchronized over a variety of parallel and serial connections, and have therefore opted for the usual double handshake by word (see the data valid and accepted pins in figure 4). Secondly, it would be entirely feasible even now to insert FIFO buffers between our internal and external ports in each direction. This might be particularly effective in combination with the new 'repeat I/O'instructions, and if the message sizes were standardized. We will examine a configuration similar to this in a later section.

Communication between modules

Figure 5 shows the format of the message block. The first word contains an eight bit field which carries addressing information about the receivers, followed by eight redundant check bits. This is similar to a scheme suggested by R.B. Kieburtz for the Stony Brook Multicomputer. It is to provide message synchronization in the absence of input and output control pins allocated to this purpose. The kernel of the receiver will output on its ERROR pin in the event of communication error or if it gets out of message synchronism with its sender. On receiving this through its $T_{O/M}$ interrupt input, the kernel of the sender will restart with a message header. The second word contains source address and message length fields. The whole message can also be protected with a longtitudinal check with errors reported on the same pin if desired.

The most straightforward communication method supported by the above modules is illustrated by the bus connection in figure 6.

All the modules which have their send ports connected to the bus will have their ARBITRATION IN and OUT pins strung in a simple cyclic chain. As shown an active pulse must be inserted into the chain at reset time to start things off. The arbitration logic in each module is responsible to check for the receipt of a pulse (token) at its input. If the WAIT FOR TOKEN flag is set by the kernel, then the TOKEN-IN bit will be set. Else the token will be passed on immediately by the hardware to the next module. This will make possible the simple sequencing of the synchro parallel connection of figure 3 (WAIT FOR TOKEN flags always set), or in a more asynchronous situation, a rapid round robin determination of which module is requesting service and is next in line.

On the receiving side, the select line input of each receiver is connected from a different line of the data bus and all the DATA ACCEPTED lines of receivers attached to this bus are connected together in WIRED-AND fashion. This shown in figure 7. When quiescent, this output is in the high impedance state. However, when a receiver is selected it changes this output to logic true or false, depending on its READY FOR RECEIVING flag. To the sender, its DATA ACCEPTED? input will not be true until all the receivers it has selected are ready. This makes possible the following communication modes to up to eight receivers; PUBLIC BROADCAST: all receivers in the subset designated by the sender receive the message;

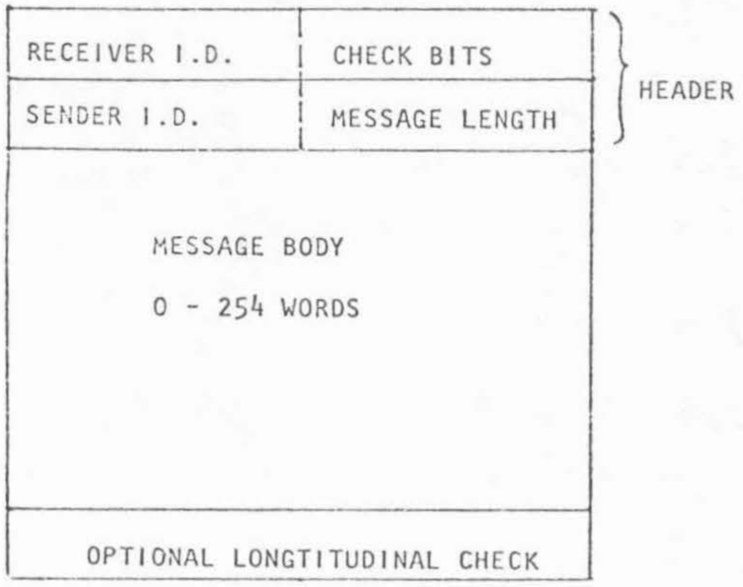


FIGURE 5 MESSAGE FORMAT

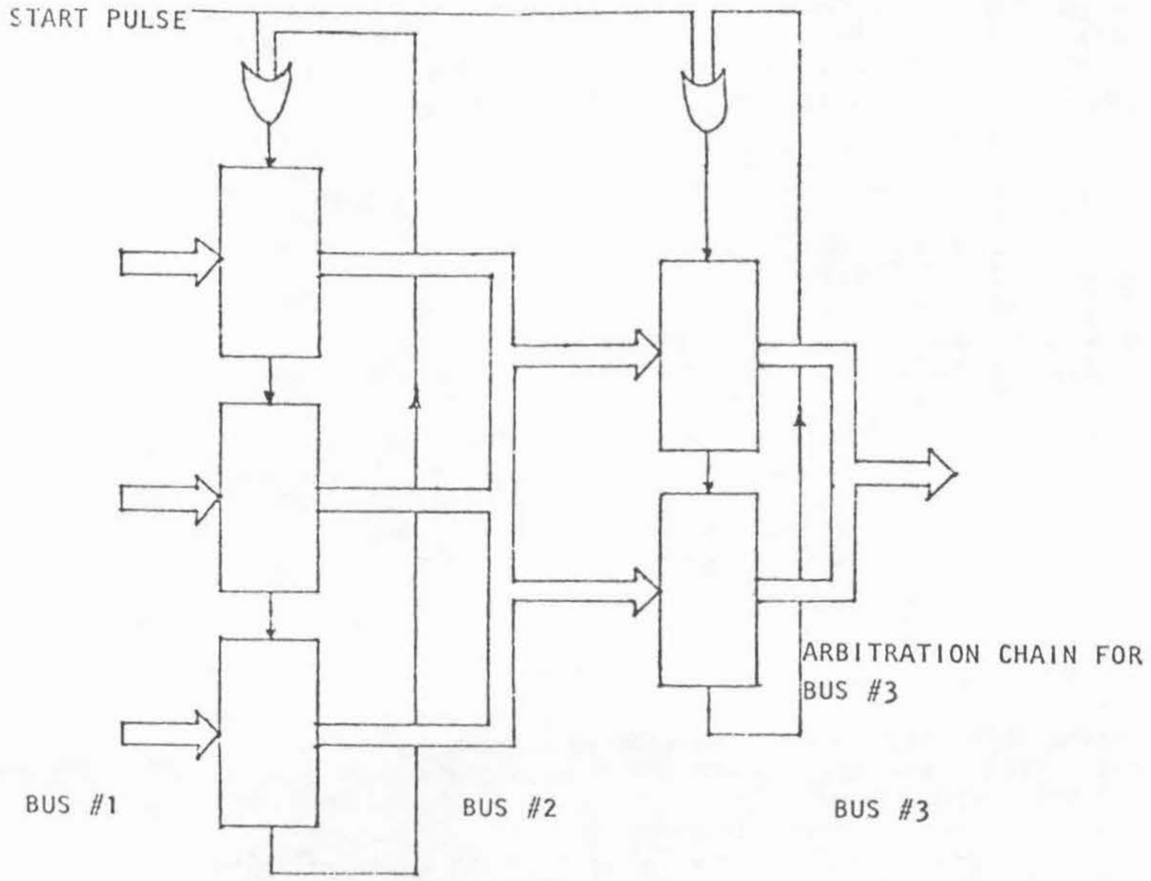


FIGURE 6 ARBITRATION CONNECTIONS

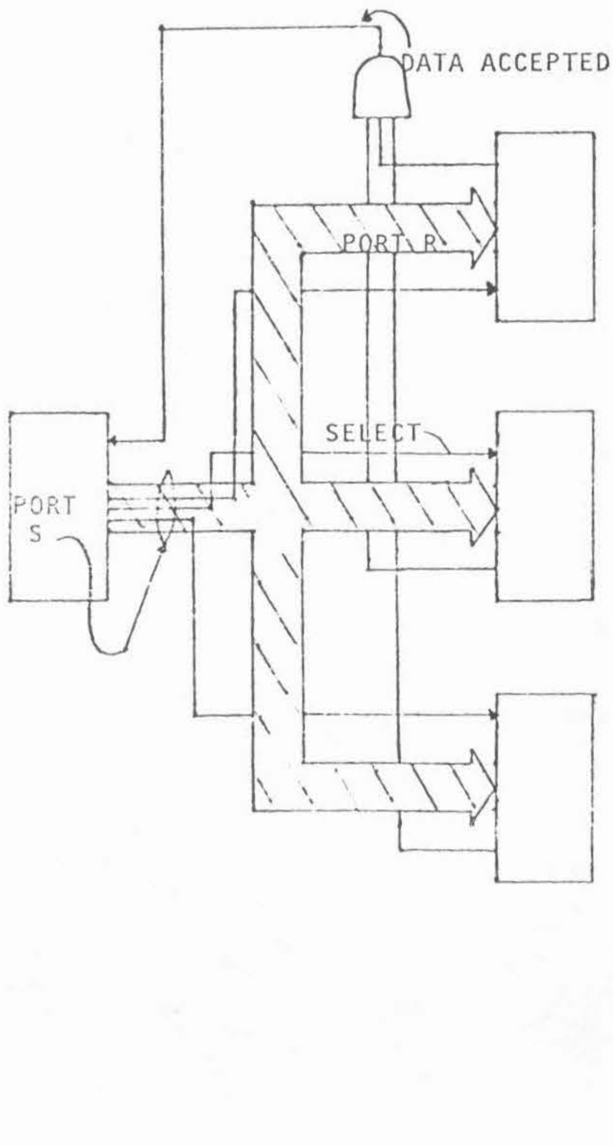


FIGURE 7

SELECT AND HANDSHAKE

CONNECTIONS

FIRST READY: the next receiver to be ready will receive the message. For both of these cases the receiver selection is mediated by the header word of the communication message block. In the FIRST READY case however only the rightmost bit in the header would be initially set and shifted one bit left after each timeout until either a ready receiver is found or all the receivers have been exhausted. In the latter case the kernel would then abort that transmission attempt and go to sleep until the next cycle of the daisy chain token. Thus FIRST READY is just a multiple application of the PUBLIC BROADCAST with a subset size of one.

In applications in which larger fan-outs are called for, this may be obtained by simply connecting the chips in the form of a tree. In this way one extra layer of 8 chips could achieve a fanout of 256 and so on. This is the same technique commonly used in multiplexer and decoder trees.

A detailed listing of the send and receive protocol sequence is given in appendix 1.

Multi-computer configurations.

Clearly the design we have outlined will be most efficient when used in simple serial or parallel unidirectional structures as in figures 2, 3, and 6. However, if bidirectional communication is desired, it can also be achieved as shown in the completely connected mesh network of figure 8. In this network, the receive port of each module is connected to every other send port, and vice versa. The disavantage of the complete connection is that a single bus and arbitration chain must encompass all modules. This implies that only one communication at a time can occur in the network. For increased communications concurrency a partially connected network would be more attractive such as that shown in figure 9. Such networks are truly multibus, with some unidirectional sections and some bidirectional.

Finally, as an important special case we draw attention to the common master slaves connection with bidirectional information transfer. As shown implemented with our single chip design in figure 10, this is achieved by using extra chips. In some cases, such as a master slave configuration with multiple slaves, it may be necessary for the slaves to identify themselves when sending to the master. Of course this could most easily be accomplished if the slaves had different programs in their ROMs including their own ID's. If the slaves had identical ROM contents however identification could still be achieved without an address bus as follows: At reset time an initializing routine in each slave could run a counter until the intial receipt of the arbitration token. It could then associate an address to itself based on the value of this count, and use this aferwards in the sender ID field of the packet headers (see figure 5).

IV. Pure Firmware Model

Here we examine whether the communication objectives set out above could be achieved without the extra specialized on-chip hardware. For this purpose we assume again the single chip computer with the two parallel bidirectional ports.

[illegible] ECTION

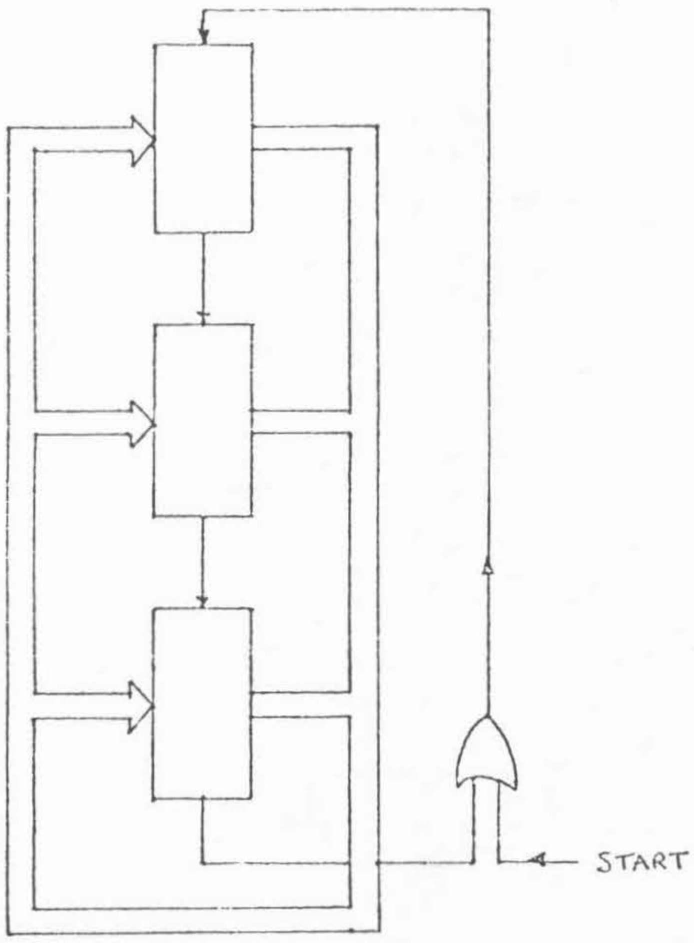


FIGURE 8 COMPLETELY CONNECTED NETWORK

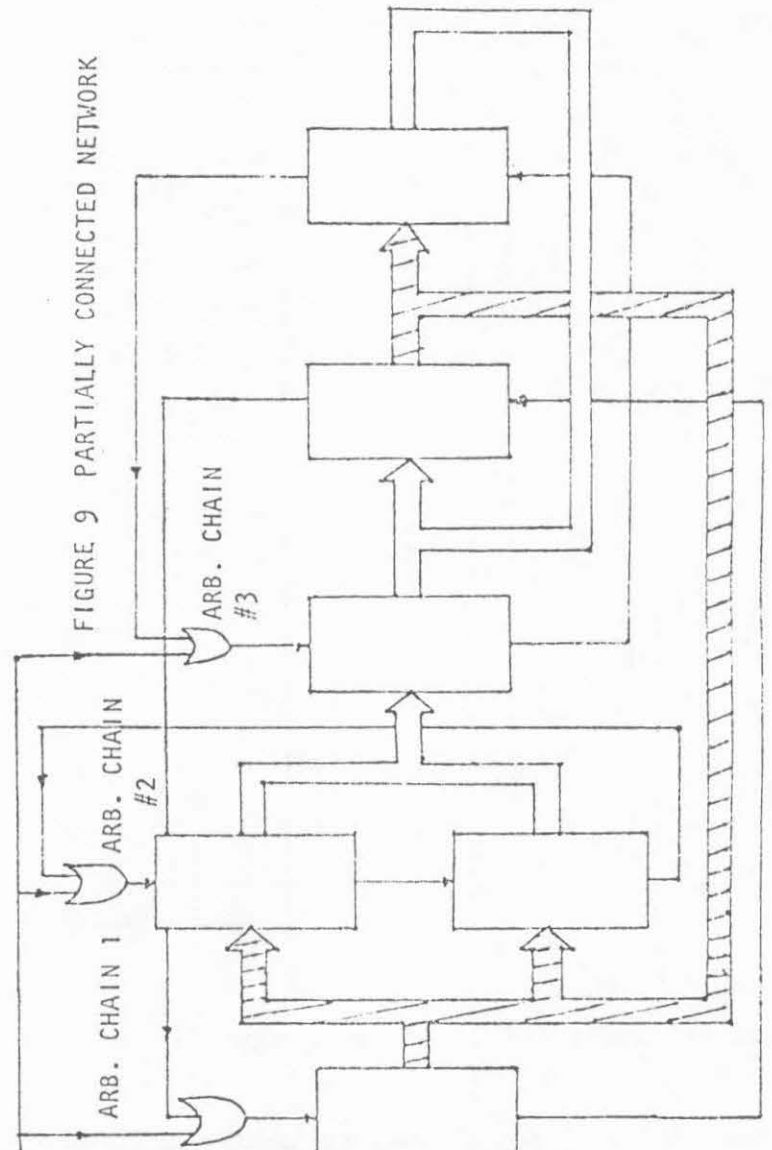


FIGURE 9 PARTIALLY CONNECTED NETWORK

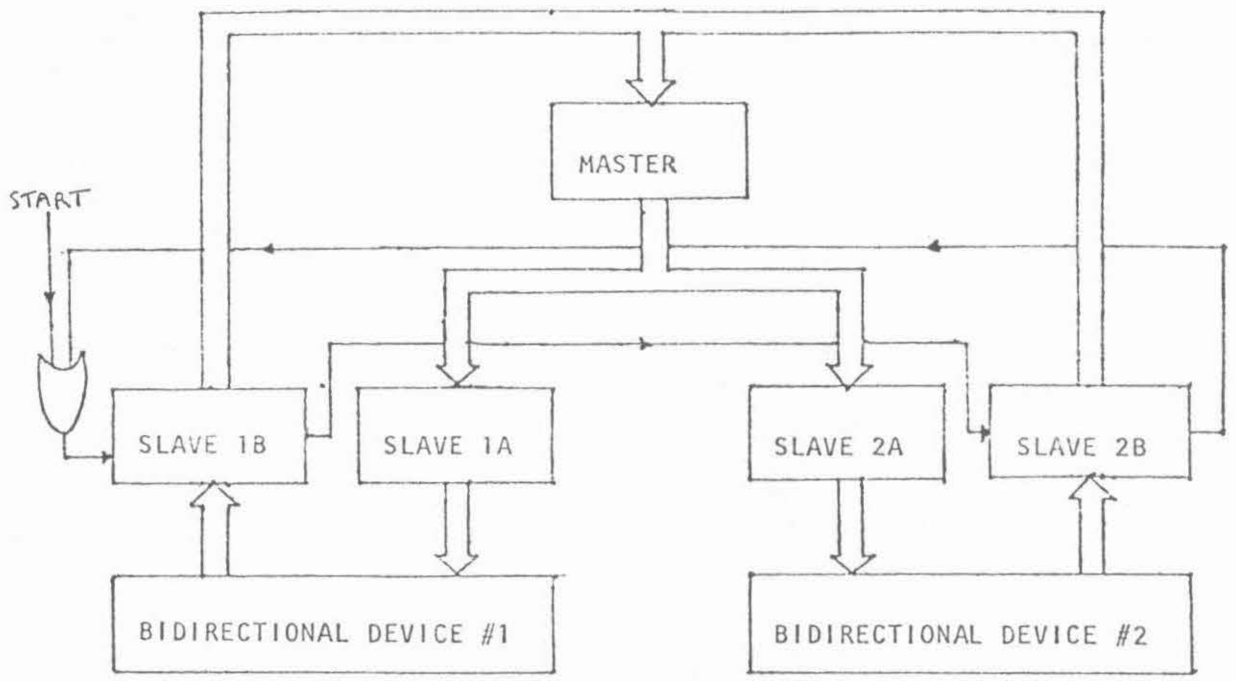


FIGURE 10 MASTER - SLAVE CONTROLLER

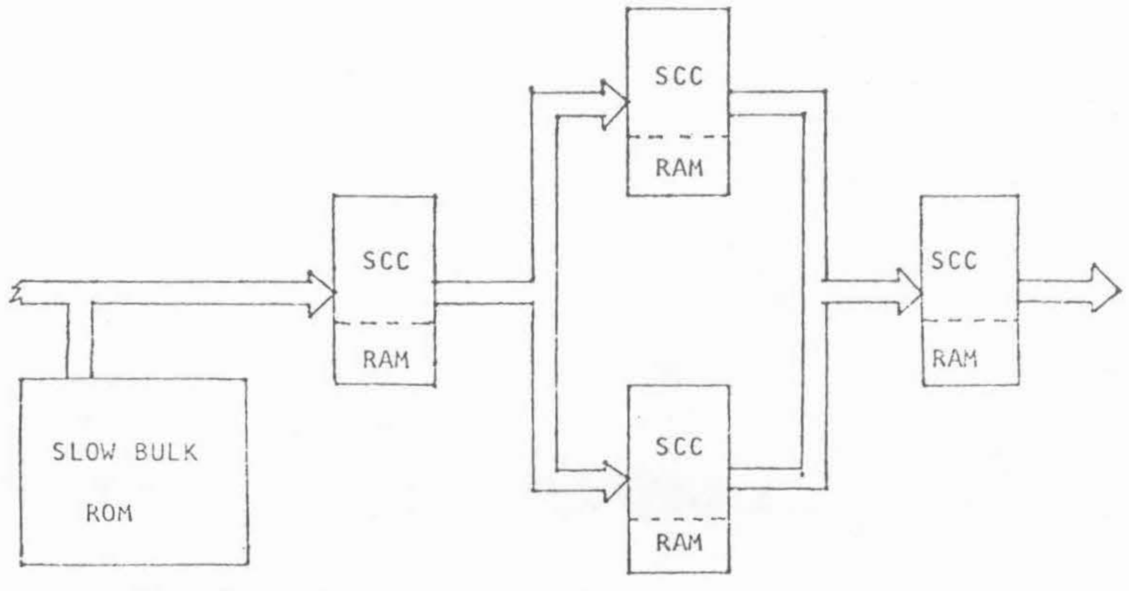


FIGURE 11 DOWNLOADING NETWORK

Without the dedicated hardware, communication protocols will necessarily be more complicated. As in other multicomputer environments we have to consider the questions of communication bus arbitration, module address selection, and the direction of information flow. Figure 12 shows our proposed utilization of the two ports to perform multicomputer communications. In the basic single chip computer, both ports A and B are bidirectional. For our present purpose port A would be the communication data bus and port B would be the control bus. Arbitration would be done in a fixed priority approach and only one-to-one module communication would be allowed.

Communications between modules would proceed in three phases. During the first phase, arbitration of the communication bus would be performed. After a new bus master is assigned, the second phase is entered and the master would select the module with which it wanted to communicate. In the third phase, communications between the master and the selected module would be performed word by word on a handshaking basis. All modules would participate in the first and second phase whereas only the master and the selected module will be involved in the third phase. Detailed description of the protocol can be found in appendix 2.

The size of the communications control program will be about 300 words. This estimation is based on the simple instructions used in section V to estimate the size of the arbitration program for the dedicated arbitrator. The timing of the arbitration process depends on the number of modules present. For example, if there were four modules on the bus, the time needed for the arbitration would be around 30 microseconds, assuming expected cycle times of next generation single chips. The address selection process would take at least another 3 microseconds. The timing of the message transmission would depend on the size of the message packet.

The advantage of this protocol is that no hardware modification is necessary. However, since only one communication bus can be available, no pipeline or synchro-parallel configurations can be achieved. Also the arbitration and address selection processes described above would have to be emulated by software and all modules would have to participate whether they were parties to that communication or not. This would result in a large amount of time used up in communications control. These must be accounted severe restrictions of the pure firmware case and an indication for hardware support of some kind.

V. Upgraded Chip with FIFO and Dedicated Arbitration.

In this section we will examine a third case in which it is assumed that the technology would permit the inclusion on the chip of a first-in first-out buffer store to assist in the communication of the message packet. Since arbitration could now take place at the packet rather than the word level, the timing requirements are slowed to the point where they could be handled by a software rather than a hardware process, - in fact on an identical chip, specialized by its ROM program for arbitration control.

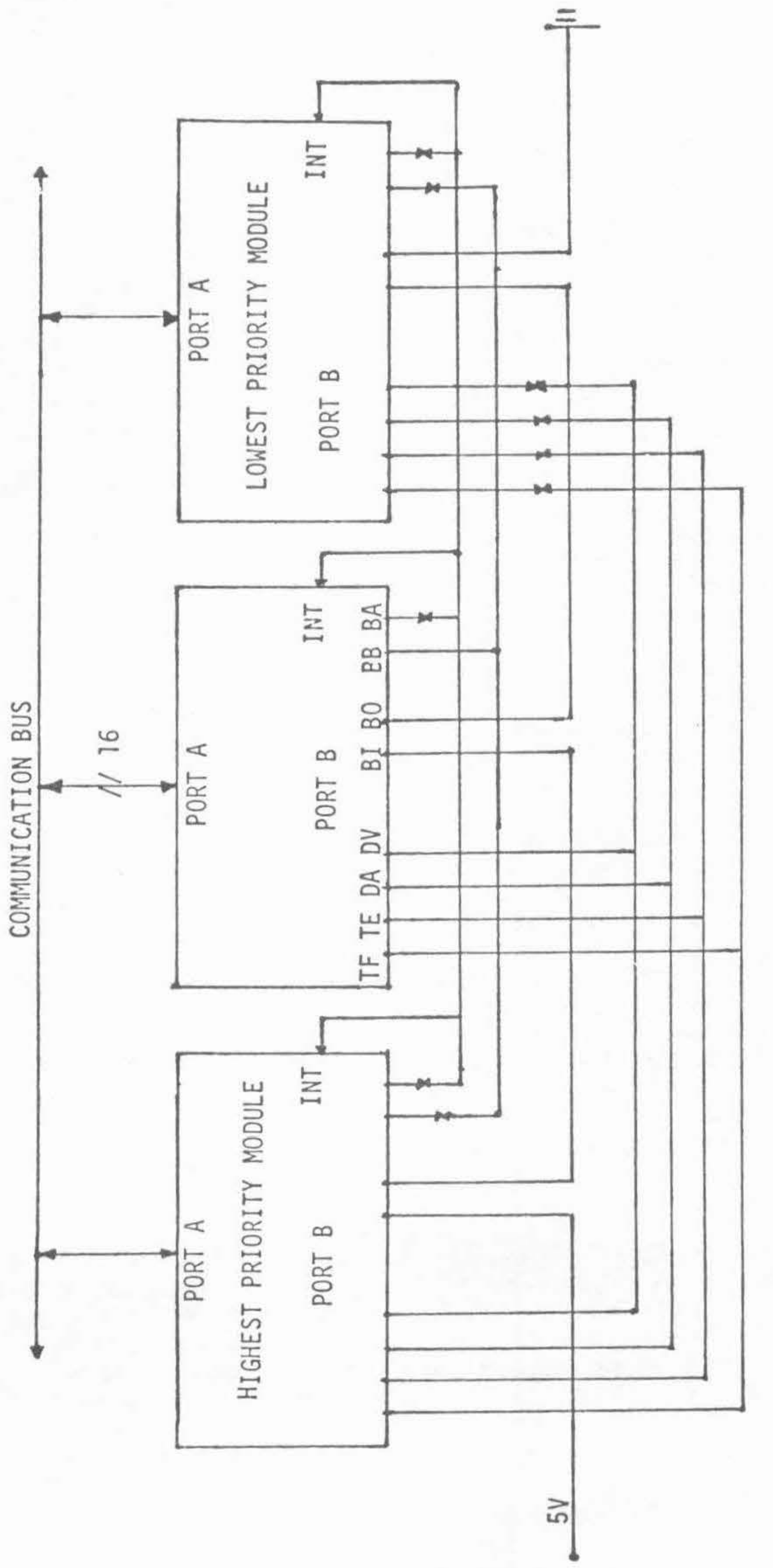


Figure 12 - PURE FIRMWARE MODEL

Otherwise we make similar assumptions as before, namely word parallel 16 bit communications and a 48 pin package limitation. Again one of the interrupt inputs will have to be multifunctional and during multicomputer mode serve as a TRANSMISSION-END signal input.

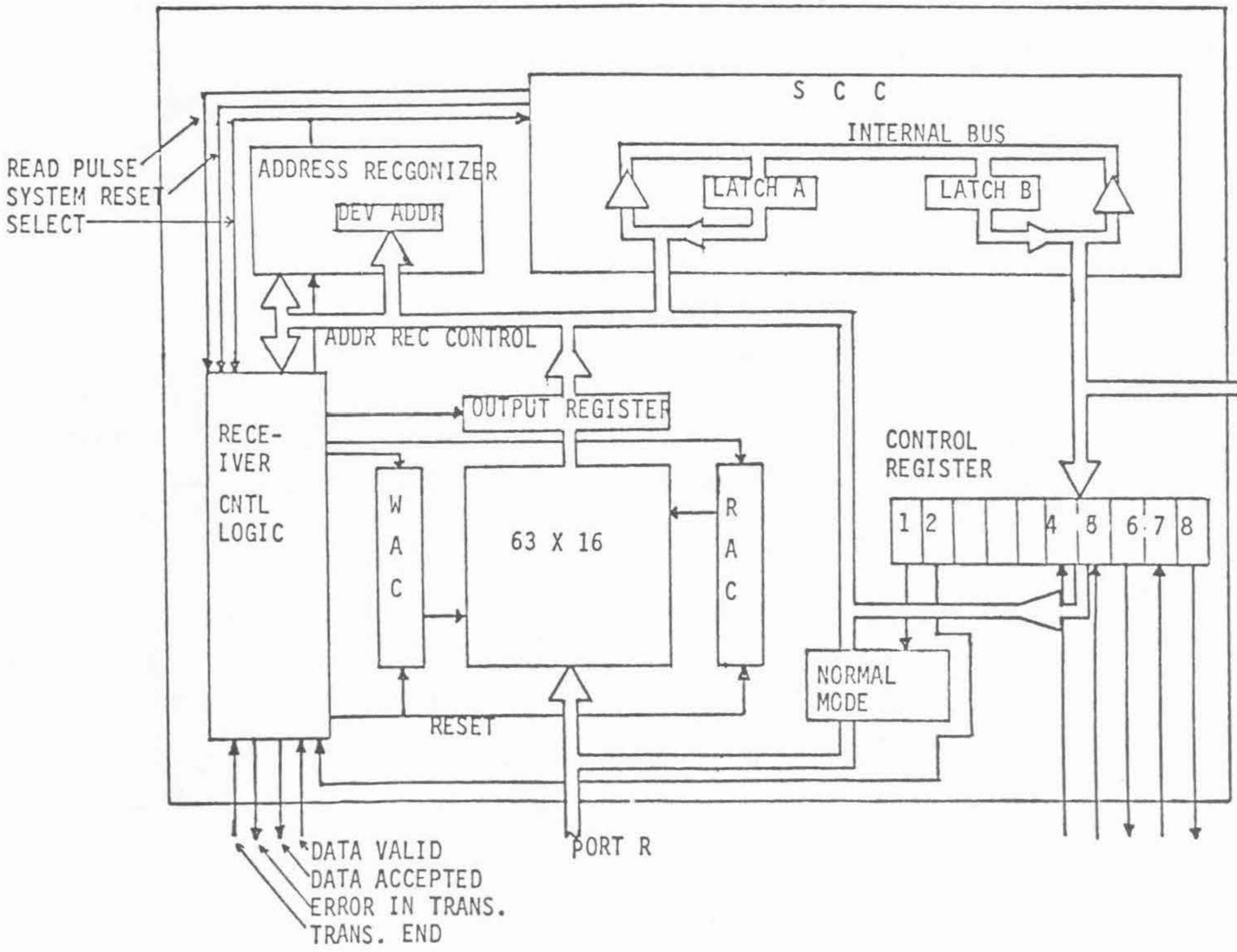
Figure 13 is the block diagram of the upgraded single chip microcomputer with FIFO. The module has two ports, namely port S for message transmitting and port R for message receiving. Both ports can be changed to bidirectional ports by software. The block labelled 'SCC' at the upper right corner is a dual port computer. To the left of SCC is the address recognizer hardware which is used to perform address comparison. If an address match occurs, it interrupts the SCC to input the message in the FIFO. If the device is not selected, the FIFO will be cleared for a new message packet. At the lower right corner is the receiver control hardware whose function is to control the address recognizer hardware and FIFO, and to generate receiver handshaking signals. In the lower center is the receiver FIFO and its control register. The structure of the FIFO is similar to current industry FIFO's. The input data is written into the storage array in a location specified by the Write Address Counter (WAC). The current output word is automatically available at the output register. After the current word of data is used, the next output word is read from the storage array at the location specified by the Read Address Counter (RAC). To the right of the FIFO hardware is the control register for the upgraded hardware. Part of the control register's contents is used as handshaking signals for transmitting messages.

Only one FIFO is placed on the receiver side in this scheme with the message words separately handshaken across under direct program control of the transmitter. This of course would be slower than if the message packet were transmitted between two FIFO's under direct hardware mediation. However we are using the FIFO here pricipally to simplify the control and we felt that two intermediary FIFO's would be more complex. The single FIFO scheme would still be faster than our first case because of the simpler synchronism -both of the communicating software processes do not have to attend at the same time. The receiver module can therefore poll the FIFO later and bring in the message with the fast repeat-I/O instruction. Thus in an N processor multibus system a bus concurrency close to N might be approached instead of being limited to N/2 as before. The detailed communications protocol for this case is given in Appendix 3.

Use of upgraded microcomputer as bus arbitrator

As we proposed in previous sections, our message communications scheme will be in packet format. There is a considerable amount of time lap between different packets being transmitted, which implies that the arbitration process can be achieved by software instead of dedicated hardware. In this section, we are going to examine the possibilities of using the same upgraded single chip computer as our communication bus arbitrator. Firstly, different arbitration schemes are discussed. Secondly, the size and speed of the arbitration program will be studied.

Figure 13 UPGRADED SINGLE CHIP MICROCOMPUTER WITH FIFO



For use as an arbitrator, our upgraded computer will be operated in normal mode, which means it will have two bidirectional ports: A and B. Port A will be used as bus request port and port B will be used as bus granted port. Figure 14 shows how local communication buses are connected to the arbitrators. More than one arbitrator can be connected together hierarchically to produce a larger arbitration system. Figure 15 shows a possible arbitration system.

Two different arbitration schemes could be implemented: Fixed Priority and Round-Robin systems. Pin 16 of the arbitrator bus request's port could be used to indicate which mode has been set, and pin 16 of the arbitrator bus grant's port used to indicate that the current bus master has released the bus. The fixed priority mode is entered when pin 16 of the arbitrator bus request's port is set to logic high. In this mode, pin 1 will have the highest priority and pin 15 will have the lowest priority. When the current bus master releases the bus by lowering its bus request line, arbitration phase is entered and the request with highest priority will be honored. Round-robin mode is entered when pin 16 of the arbitrator's receiver is set to logic low. In this mode, all pins will have the same priority and requests will be honored in a circular fashion. By combining the two arbitration schemes we could implement a hierarchy arbitration system as in figure 15. Modules having the same priority level are connected to the same arbitrator which implements the Round-robin scheme, where the arbitrators themselves are connected to a master arbitrator which implements the fixed priority scheme. With this hierarchical parallel structure, we could connect any number of modules together and still have minimal delay time. Both round-robin and fixed priority algorithms have the same structure. While a current bus master is using the bus, the bus arbitrator will continuously perform next bus arbitration. As soon as the current master has finished, a new bus master can be assigned. With this approach, delay time can be minimized.

Arbitration program outlines can be found in appendix 4. The sizes of both programs are about the same. Since we have to incorporate both programs into the program memory, their total size is less than 200 words. Worst case arbitration delay time is when there is no pre-arbitration done, in this case the arbitration delay time will be less than 30 microseconds. When there is pre-arbitration done, which is the average or best case, the timing will be less than 4 microseconds. With this arbitration delay time, we concluded that the size of the FIFO should be efficiently be 64 words, since the time the transmitter takes to transmit one packet should be greater than one arbitration period.

VI. Conclusion and Discussion.

Of the three schemes examined we believe the pure firmware model has been shown to have inefficiencies which might cancel out the advantages of the multiprocessor operation in the type of applications envisaged. If technology and marketing considerations would permit the 48 pin package then the inclusion of upgraded communication hardware would result in a more efficient system, and if a FIFO could be included, still

Figure 14. LOCAL BUS CONNECTION SCHEME

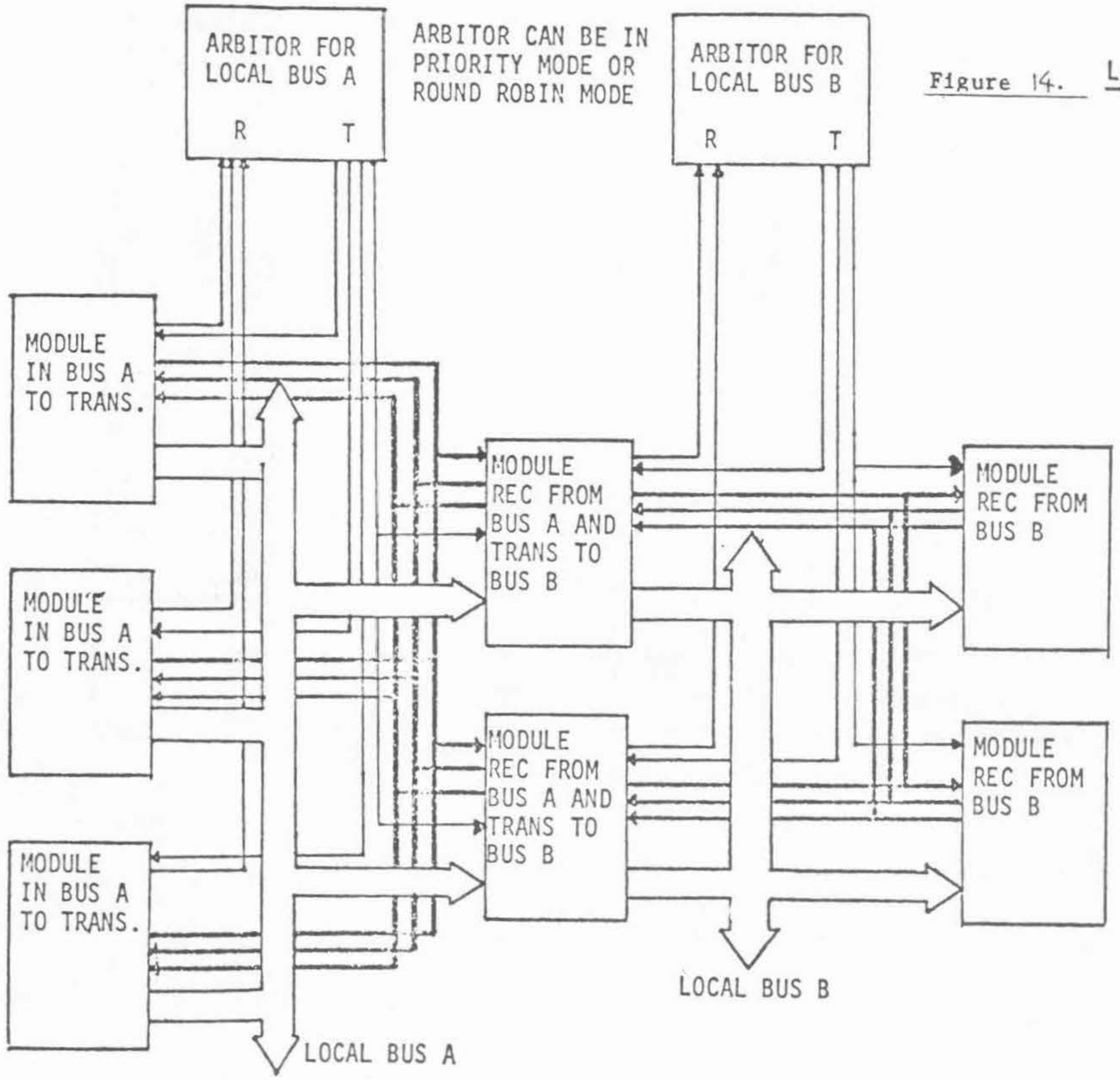
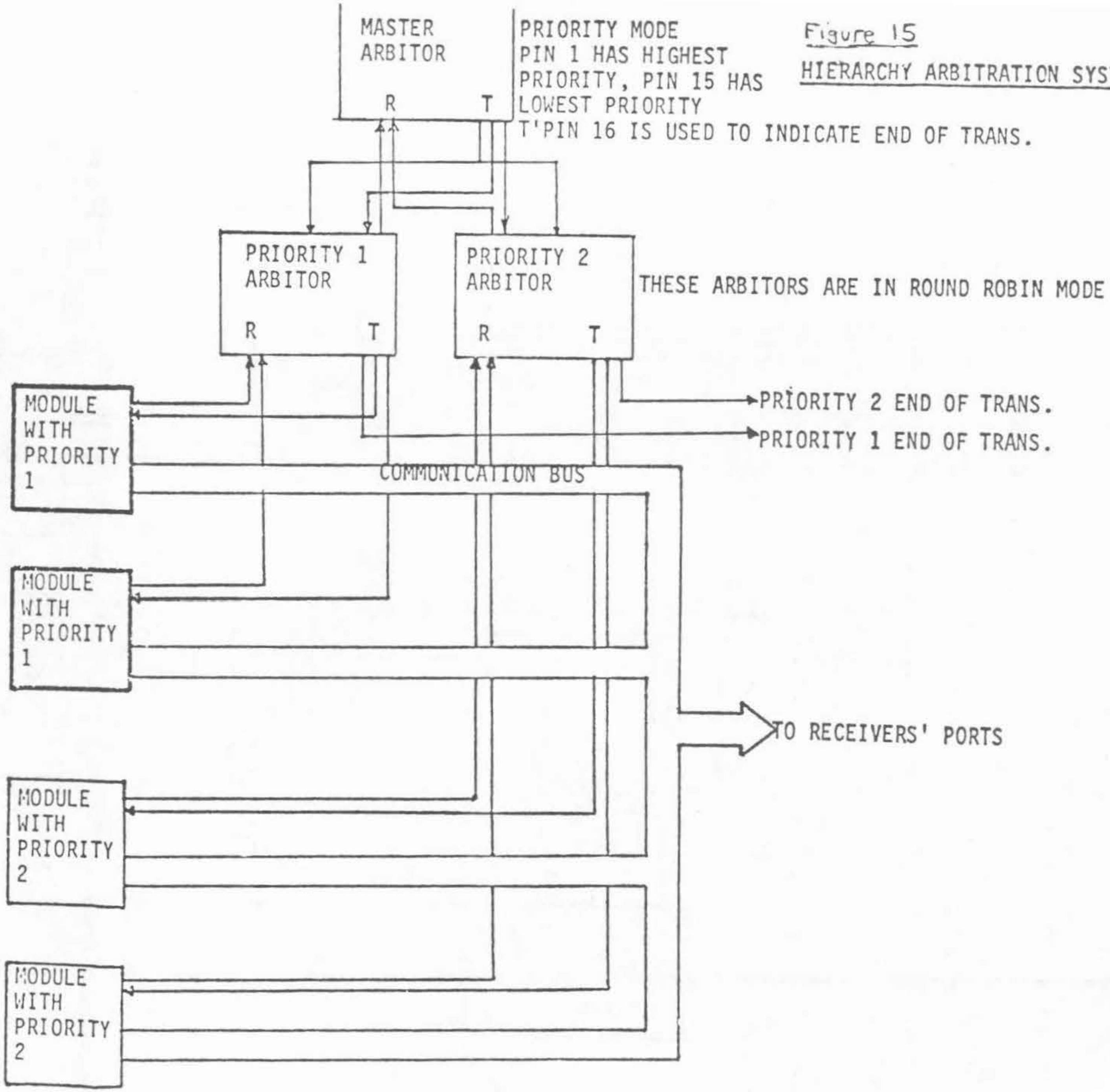

ARBITOR FOR LOCAL BUS A
R
T
ARBITOR CAN BE IN PRIORITY MODE OR ROUND ROBIN MODE
ARBITOR FOR LOCAL BUS B
R
T
MODULE IN BUS A TO TRANS.
MODULE IN BUS A TO TRANS.
MODULE IN BUS A TO TRANS.
MODULE REC FROM BUS A AND TRANS TO BUS B
MODULE REC FROM BUS A AND TRANS TO BUS B
MODULE REC FROM BUS B
MODULE REC FROM BUS B
LOCAL BUS A
LOCAL BUS B

Figure 15

HIERARCHY ARBITRATION SYSTEM



better. Realistically, these options may not be attractive in the first generation of 16 bit single chip computers since the communications software of two to three hundred words would be an appreciable fraction of the whole program memory space available. However, with each year that passes after that, with a doubling of program space, these schemes should become increasingly attractive.

In conclusion, what we have advocated here for the single chip computer can be viewed as a low cost alternative to the more expensive multiprocessor-multibus architectures which have been proposed for the current family of multichip microcomputers. It has also been oriented towards the area of fixed rather than general purpose applications. In this context, the development of a firmware kernel in ROM to remove most of the burdens of the communications from the users is clearly of crucial importance. Such a kernel would clearly have differences with kernels now being developed for large experimental general purpose multicomputers (such as [11]), since it should be oriented towards simplicity and speed efficiency.

An important property of the proposed chips is the flexibility of use and low cost. In this way the idea of employing extra chips of the same kind in a design can be easily entertained. Examples of this have been seen here for increasing communications fanout and for adding bidirectionality to the links. This perhaps may point the way to the use of such chips as this as general purpose system building blocks, much as discrete logic gates and programmed logic arrays have been used in previous technologies.

Looking slightly further ahead, we can see that one of the factors raising the costs of these multicomputer applications would be the different application software modules which would have to be burnt into chips in different parts of the network (parallel arms, serial arms). In other words we would like our standardized modules to be exactly alike in all respects. Eventually this might be achieved as shown in figure 11, using a system of downloading the applications software at power-on time into modules based on RAM rather than ROM. Here is where a hierarchical memory might be indicated in each chip with a large dense dynamic RAM backing up a small static RAM cache. This would also add new functions to the kernel, with perhaps a bootstrap version in each module bringing in the applications software and then the running kernel.

References

[1] Moore, G.E., "Are we ready for VLSI?", Proc. Caltech. Conference on Very Large Scale Integration, Jan. 1979, pp 3-14.

[2] Mead C., & L. Conway, Introduction to VLSI systems, Addison-Wesley, 1980.

[3] Clark J., "VLSI geometry processor for graphics", Computer J., July 1980, pp 59-68.

[4] Caldwell J., "Real time text to speech using custom VLSI and standard micro-computers, proc. Spring Compcon 80, p43.

[5] Dixon L.R., & T.B. Martin, Automatic speech and speaker recognition, IEEE press, 1979.

[6] Medress M.F. et al, "A system for the recognition of spoken connected word sequences, ibid, pp 238-247.

[7] Dixon L.R. & H.F. Siverman, "The 1976 acoustic processor MAP", IEEE Trans ASSP-25, (Oct 77), pp 367-379.

[8] Roberts C.S., "An associative/parallel processor for partial match retrieval using superimposed codes", Proc. 7th Symp. Computer Architecture, (April 1980), pp 218-227.

[9] Patterson D.A., & C.H. Sequin, "Design considerations for single chip computers of the future", IEEE Trans Vol. C-29:2 (Feb 1980), pp 108-116.

[10] 8086 family user manual, Intel Corp, 1980.

[11] Sadayappan P. et al, "An operating system kernel for a hierarchical computer", Proc. Fall Compcon 80, Sept 23-25 1980.

[12] Sequin C.H., "message switching circuits for multimicrocomputers", Proc. Spring Compcon 80, pp 328-334.

[13] Harris, J.A., & D.R. Smith, "Simulation experiments of a hierachical multicomputer" Proc. 6th Symp. Computer Architecture, 1979, pp .

[14] Chen T.C., "Overlap and pipeline processing", Ch. 9 in Introduction to Computer Architecture, H.A. Stone, ed., SRA assocs., 2nd ed., 1980.

APPENDIX 1

Transmission:

** The Kernel sets up the message block in its RAM. It will clear the READY-FOR-RECEIVING flag in the selection logic and set the WAIT-FOR-TOKEN flag in the daisy chain logic. It then continues processing and periodically polls the TOKEN-IN bit in the I/O register. When the daisy chain logic receives the token, it will check its WAIT-FOR-TOKEN flag. If the flag is set, it will set the TOKEN-IN bit in the I/O register and clear the WAIT-FOR-TOKEN flag.

** When kernel knows the TOKEN-IN bit is set, it will change its transmitter port and handshaking lines from high impedence to logic false. RTL1 interrupt line is also disabled. Afterthis, kernel will load latch A with first word of message and set the DATA-VALID out bit to true. (First word of message block is the bit pattern specifying which subset of receiver is chosen)

** The kernel will initiate a timer at this instance. If the timer has run out before the DATA-ACCEPTED IN bit is set, it will abort the message transfer and reset the appropriate bits and flags.

** When the kernel sees its DATA-ACCEPTED IN bit is set, it will acknowledge by clearing its DATA-VALID OUT bit.

** When the kernel see its DATA-ACCEPTED IN bit is cleared, it will send the rest of the message by handshaking. When transmitting the second word of the message, it will check the TRANSMISSION-ERROR IN bit. If the bit is set, it will clear the DATA-VALID OUT bit and check the last word transferred. As soon as the TRANSMISSION-ERROR IN bit is cleared by the receiver, the kernel will set the DATA-VALID OUT bit and re-transmit that word to the receiver. The kernel has control over number of retries and it will abort the message transfer if that limit has reached.

** When the kernel is finished with the transmission, it will clear the TOKEN-IN bit and set the READY-FOR-RECEIVING flag in the selection logic. After this, it will put its transmitter port and handshaking lines back to high impedence state.

** Upon the TOKEN-IN bit is cleared, the daisy chain logic will pass the token to the next module down the chain.

Receiving:

** The SELECT input line of the receiver will be in high impedence state during receiving and will be in logic false state when waiting for input.

** When the receiver's SELECT input is on, it will check the READY-FOR-RECEIVING flag. If the flag is set, it will change the DATA-ACCEPTED OUT line from high impedence to logic true. In addition, the receiver's port and handshaking lines are all changed to low impedence. If the READY-FOR-RECEIVING flag is not set, kernel will set the DATA-ACCEPTED OUT line from high impedence to logic false and DATA-VALID IN line to low impedence. When the DATA-VALID IN bit is reset, the kernel will reset the receiver's handshaking line back to high impedence state.

** The kernel will receive the rest of the message by handshaking with the transmitter. It will check for transmission error in the second word of the message. If error did occur, it will set the TRANSMISSION-ERROR OUT bit to true and discard that word. As soon as the DATA-VALID IN bit is cleared, it will clear the TRANSMISSION-ERROR OUT bit and waits for re-transmission of that word. When predetermined number of retries is over, kernel will abort the message receiving and reset its bits and flags.

** When the message receipt is completed, the kernel will put its receiver port and handshaking lines back to high impedence state. The SELECT input is reset to logic false.

APPENDIX 2 - Pure Firmware Model.

* When the system first boots up, all modules' BB, BA, DV, DA, TE and TF lines are not asserted. Except for the highest priority module's BI line is connected to 5V and the lowest priority module's BO line is connected to ground, all other modules' BI and BO lines are not asserted.

* When a module wants to gain access to the bus, first it has to check the BB (Bus Busy) line. If it is asserted, it has to wait for the current master to finish. If BB is not asserted, the module will assert its BA line. Since the BA line is connected to all modules' T1 interrupt line (including itself), all modules will enter an interrupt service routine which then begin the arbitration process.

* The module which asserted the BA line will reset it.

* All modules will initiate a count down timer for maximum arbitration process duration. When the count times out and the BB line hasn't been asserted, arbitration process will be terminated and all modules will resume their previous processes.

* Meanwhile all modules sample their BI (Bus arbitration In) input. When this signal is asserted for a module which did not request the bus, then that module will assert its BO (Bus arbitration Out) and wait for BB assertion or timer run out to occur. If this module was not the one which requested the bus, it will assert its BB line.

* As soon as the module gains control to the bus, it becomes the current bus master. It then puts a device address word on to the data bus and asserts its DV (Data Valid) line. When its DA (Data Accepted) line is asserted by the selected module, the master will

reset its DV line and returns to its calling process to perform the message communication. If the DA line doesn't assert when the timer run out occurs, then the arbitration process will be terminated and all modules will resume their previous processes.

* When a module's BB line is asserted by a new master, it will wait for the device address word to determine whether it is being selected. If a module is selected, it will assert its DA line. When the master resets its DV line, the selected module will enter a communication routine. Other modules will terminate the arbitration process and resume their previous processes.

* The DA lines are wired OR together, so only one to one module communication is allowed.

* Communication will be performed on word by word handshaking basis. At any time if the receiver module asserts the TE (Trans. Error) line, master will try to retransmit the last word for a predefined number of times. If still fail, communication will be aborted.

* When the communication is finished, the master asserts the TF (Trans. Finished) line and waits for DA line assertion. When its DA is asserted, it will reset its TF and BB line and the communication is concluded.

APPENDIX 3 - Upgraded Chip with FIFO

The modules all have distinct coded addresses. Modules will generate their addresses during the system boot up phase. When the system first boots up, all modules on a local bus will assert their BR (Bus Request) lines. Depending on the arbitration scheme used by the local arbitrator, individual modules will be granted the bus in succession and can generate their addresses based on the time elapsed.

Communications will proceed in two phases. During the first phase or bus arbitration phase, a module which wants to use the bus asserts its BR line and waits for its BG (Bus Grant) line to be asserted by the arbitrator. Once the BG line is asserted, the selected module will enter phase two which is the communication phase.

Since receiving is performed by hardware and buffered by a FIFO, we expect all the modules on the receiving side are ready to receive when the transmitter is ready to transmit the message packet. However, if the Receiving Buffer Full bit of the control register is set, the receiver module has to wait till it is clear before it can accept any more message. Handshaking signals are used throughout the communications, and the receivers' DA (Data Accepted) lines are wired AND together so as to ensure all the receivers are ready.

When the module gains access to the local communication bus, it will send out the message packet with a packet header word containing the coded destination address to the receiver modules. Depending on the coded address, message packets can be directed to an individual module or broadcasts to all of them.

At any time, if the transmitter's Error-in-Transmission line is asserted by one of its receiver modules, the transmitter will retransmit the last word up to a predefined number of times. If the error condition persists the transmitter will abort the transmission.

When the transmission is finished, the transmitter module will reset its BR line. The arbitrator will then reset the corresponding BG line and asserts its Transmission End line to all the receiving modules.

Once the Transmission End line is asserted by the arbitrator, all the receiving modules will check the packet header word and determine whether the message is intended for them. If the message is intended for a particular module its control logic will interrupt the SCC in order to ship the message from the FIFO to the SCC's memory. If the message is not directed to the receiver, the receiver control logic will clear the FIFO for another new message packet.

When a transmitter module is not the current bus master, its transmitter's handshaking lines are in high impedence state.

APPENDIX 4 - Arbitration programs

Listing of the arbitration program:

```
/* initialization phase, determine which arbitration scheme to use */

F-begin:                Load save-req with 000...000
                        Load temp with request word
                        temp <- temp logical AND 100...000
                        temp = 0?
                        (false) jump to f-wait-loop
                        (true ) jump to r-wait loop

/* enter Fixed Priority mode, look for bus request */

f-wait-loop:            load req-wd with request word
                        req-wd <- req-wd logical AND 011...111
                        req-wd = 0?
                        (false) jump to f-arb-1
                        (true ) jump to f-wait-loop
```

```
/* determine who gets  the bus               */

f-arb-1:               load arb-end with 010...000
                       load arb-curl with 100...000
f-arb-1loop:           left circular shift arb-curl 1 bit
                       temp <- arb-curl logical AND req-wd
                       temp = 0?
                       (false) jump to f-grant-bus
                       (true ) arb-end = arb-curl?
                       (false) jump to f-arb-1loop
                       (true ) jump to f-wait-loop

/* grant the bus to the highest priority requesting module */

f-grant-bus:           load bus grant word with arb-curl
                       load bus granted with 1
                       load save-req with 000...000

/* now start doing pre-arbitration while the bus is busy */

f-arb-2:               load req-wd-2 with request word
                       req-wd-2 <- req-wd-2 XOR arb-curl
                       req-wd-2 <- req-wd-2 logical AND 011...111
                       req-wd-2 = 0?
                       (false) jump to f-continuel
                       (true ) jump to f-arb-2
f-continuel:           load arb-end with 100...000
                       load arb-cur2 with 100...000

/* determine whether the bus is being used */

f-test:                bus-granted = 1?
                       (false) jump to f-arb-2loop
                       (true ) jump to f-poll

/* check to see whether the current master has finished if finished, and
there is an outstanding bus request, grant the bus. Otherwise continue
pre-arbitration process */

f-poll:                load temp with request word
                       temp <- temp logical AND arb-curl
                       temp = 0?
                       (false) jump to f-arb-2loop
                       (true ) load bus grant word with 100...000
                       load bus grant word with 000...000
                       load arb-curl with 000...000
                       load bus-granted with 0
                       save-req <> 0?
                       (false) jump to f-arb-2loop
                       (true ) load arb-curl with save-req
                       jump to f-grant-bus
```

```
/* perform pre-arbitration */

f-arb-2loop:              left circular shift arb-cur2 1 bit
                          arb-end = arb-cur2?
                          (false) jump to f-continue2
                          (true ) jump to f-arb-2
f-continue2:              temp <- arb-cur2 logical AND req-wd-2
                          temp = 0?
                          (false) jump to f-set-or
                          (true ) jump to f-test

/* save the request from pre-arbitration process */

f-set-or:                 load save-req with arb-cur2
                          bus-granted <> 1?
                          (false) jump to f-arb-2
                          (true ) jump to f-grant-bus


/* enter Round-Robin mode, look for bus request */

r-wait-loop:              load req-wd with request word
                          req-wd <- req-wd logical AND 011...111
                          req-wd = 0?
                          (false) jump to r-arb-1
                          (true ) jump to r-wait-loop

/* determine who should the bus be granted */

r-arb-1:                  load arb-end with 010...000
                          load arb-cur1 with 100...000
r-arb-1loop:              left circular shift arb-cur1 1 bit
                          temp <- arb-cur1 logical AND req-wd
                          temp = 0?
                          (false) jump to r-grant-bus
                          (true ) arb-end = arb-cur1?
                          (false) jump to r-arb-1loop
                          (true ) jump to r-wait-loop


/* grant the bus to the highest priority requesting module */

r-grant-bus:              load bus grant word with arb-cur1
                          load bus granted with 1
                          load save-req with 000...000

/* now start doing pre-arbitration while the bus is busy */

r-arb-2:                  load req-wd-2 with request word
                          req-wd-2 <- req-wd-2 XOR arb-cur1
                          req-wd-2 <- req-wd-2 logical AND 011...111
                          req-wd-2 = 0?
                          (false) jump to r-continue1
                          (true ) jump to r-arb-2
```

```
/* determine where to start pre-arbitration */

r-continue1:              bus-granted = 1?
                          (false) jump to r-new
                          (true ) load arb-cur2 with arb-cur1
                          jump to r-poll

r-new:                    load arb-end with 100...000
                          load arb-cur2 with 100...000
                          jump to r-arb-2loop

/* check to see whether the current master has finished if finished, and
there is an outstanding bus request, grant the bus. Otherwise continue
pre-arbitration process */

r-poll:                   load temp with request word
                          temp <- temp logical AND arb-cur1
                          temp = 0?
                          (false) jump to r-arb-2loop
                          (true ) load bus grant word with 100...000
                          load bus grant word with 000...000
                          load arb-cur1 with 000...000
                          load bus-granted with 0
                          save-req <> 0?
                          (false) jump to r-arb-2loop
                          (true ) load arb-cur1 with save-req
                          jump to r-grant-bus

/* perform pre-arbitration */

r-arb-2loop:              left circular shift arb-cur2 1 bit
                          arb-end = arb-cur2?
                          (false) jump to r-continue2
                          (true ) jump to r-arb-2
r-continue2:              temp <- arb-cur2 logical AND req-wd-2
                          temp = 0?
                          (false) jump to r-set-or
                          (true ) jump to r-test

/* save the request from pre-arbitration process */

r-set-or:                 load save-req with arb-cur2
                          bus-granted <> 1?
                          (false) jump to r-arb-2
                          (true ) jump to r-grant-bus
```

Note: save-req, temp, req-wd, arb-end, arb-cur1, bus-granted, req-wd-2, arb-cur2 can be registers or memory words.