To the Graduate Council:

I am submitting herewith a thesis written by Bryant E. Sorensen entitled "Design of a Digital Crossbar/Crosspoint Switch for TMS320C40 Digital Signal Processors." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Bruce Bomar

Dr. Bruce Bomar, Major Professor

We have read this thesis
and recommend its acceptance

L Montgomery Smith

Roy D Joseph

Accepted for the Council:

Lou Minkel

Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Bryant E Sorensen

Date November 3, 1993

DESIGN OF A DIGITAL CROSSBAR/CROSSPOINT SWITCH FOR TMS320C40 DIGITAL SIGNAL PROCESSORS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Bryant E. Sorensen

December 1993

ACKNOWLEDGEMENTS

I would like to thank Dr. Bruce Bomar, who gave many suggestions and much encouragement to this research project. I appreciate the time I had to dedicate to its timely completion. I would also like to thank Mr. Tom Tibbals of Sverdrup Technology, Inc., for the encouragement and support that made this project possible. Mr. Don Hervig has my thanks for loaning me the software used in the designs for all the time I needed.

I would most like to thank my wife, Marie, for coming with me to graduate school and supporting me while here. Her work in taking care of our family matters allowed me to concentrate on my studies while here.

This research was supported in part by Sverdrup Technology, Inc., under contracts A93W-03 and A93W-16 with The University of Tennessee Space Institute (UTSI). The work was also supported in part by the Air Force Office of Scientific Research Summer Graduate Research Program, Summer 1993.

ABSTRACT

The design of an eight-port digital crossbar/crosspoint switch is presented. An *N-port crossbar switch* is an interconnection device in a multiprocessor system which allows connection between pairs of data ports of different processors. There are a total of $\frac{N}{2}$ connections possible at a time, where N is the number of ports attached. A *crosspoint switch* allows any one of the ports to broadcast its data to as many of the other ports as desired, giving a possibility of N-1 connections. This design provides both these interconnections between eight communication ports of Texas Instruments' TMS320C40 (C40) Digital Signal Processors in a parallel processing system. The eight-port switch design proves the concepts necessary to expand to a larger switch, specifically a thirty-two port switch. The design is based on LSI Logic's Crossbar Switch VLSI chip and utilizes large programmable logic devices to handle data transfer handshaking, data bus direction changing, and configuration handshaking between the switch and the control unit. The control unit consists of a remote C40 connected to the switch through a memory expansion protocol, and the appropriate control software. The switch was tested for data path integrity and throughput rate. A data rate of 6.6 to 7.2 $\frac{\text{Mbytes}}{\text{sec}}$ through the switch was achieved using 40 Mhz C40s, demonstrating the switch's usefulness in a parallel processing system. Expansion to larger switches, specifically a thirty-two port switch, is also discussed.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The Texas Instruments TMS320C40 Digital Signal Processor (TI C40 DSP) has 6 high-speed communication ports (commports) on the chip. This presents a great advantage in parallel processing systems, as each processor can transmit and receive data from up to 6 other processors quickly. However, in large parallel processing systems that handle data acquisition and analysis, it is advantageous to have the following features for each processor [1]:

(1) Have communication with more than 6 other processors;

(2) Be able to change the organization of the network interconnections (network topology), specifically the commport connections, while the system is in use, through software control;

(3) Broadcast data from one processor to multiple other processors.

One way to provide some of these features in a parallel processing environment is through fixed C40 commport interconnection architectures. TI suggests a few possibilities in their C40 data book ([2], pp. 13-40 to 42). The challenge with the architecture approach lies with finding the optimal topology that minimizes time between ports, and creating the software to make each desired connection through a minimum-delay route. These fixed architectures, however, do not allow for changes in the topology while the system is in use, nor do they provide for one commport to be able to broadcast data to multiple other commports without sending the data repeatedly.

One way to provide all these desired features is a to use a crossbar/crosspoint switch. An *N-port crossbar switch* is a hardware interconnection device in a parallel

processing system which allows connection between pairs of data ports of the different processors attached to the switch. A total of $\frac{N}{2}$ connections are possible in the switch at a time, and these connections can be changed through software while the system is in use. A *crosspoint switch* allows any one of the ports to broadcast its data to as many of the other ports as desired, for a maximum of N-1 connections. The crosspoint connections are also reconfigurable while the system is in use.

Crossbar switches have been built for various uses [1] [3], but none are known which handle the C40 commports. Also, the crossbar switches described in the literature do not claim crosspoint switch capabilities.

This thesis describes the design of a crossbar/crosspoint switch used to connect the commports of the C40 processor. The crossbar and crosspoint switch capabilities are both included in the design, which is referred to as a crossbar switch. An 8-port switch (N = 8) was designed, built, and tested as proof-of-concept for larger switches, specifically a 32-port switch. The final goal of this design and test is to prove that a 32-port switch is feasible to build and use in a large parallel processing system. The 8-port size was used as the test because it provides more connections than just a processor itself with 6 commports, was more affordable to build, could be tested with just a few other processors, and could be scaled easily to create the 32-port switch.

A. Specifications of the Design

Each commport consists of twelve lines. There are eight data lines, two data handshake lines, and two bus ownership arbitration lines. Both the crossbar and crosspoint parts of the switch must be able to connect any two of the eight data busses in the system, with up to 4 connections in the switch for the crossbar and

7 connections for the crosspoint switch. Each connection must be made as quickly as possible so the system does not become bottlenecked.

The crossbar switch part must handle the handshaking to change the data bus direction. Each commport has a bus request line which it asserts when it wants ownership of the bus in order to transmit data, and a bus grant signal which it asserts when it has finished transmitting data and is ready to give up the bus in order to receive data. When asserted, these signals must be handled by the switch to change the bus direction of two connected commports within a certain time to avoid electrical damage to the DSP's.

The crosspoint part of the switch does not allow direction changes, but instead must distribute the data and data strobe from the broadcasting port to all the receiving ports, then logically combine the data acknowledges from the receiving ports to create one signal at the broadcasting port. This combination must only use those signals from the ports that are receiving data and ignore any signals from ports which are not involved in the broadcast.

B. Approach

The design is divided into four main parts:

(1) The data bus connection for both the crossbar and crosspoint switch;

(2) A bus ownership arbitration signal handler for the crossbar switch;

(3) A data bus handshaking arbiter for the crosspoint switch;

(4) A control unit and address decoding for the entire device.

Figure 1 shows a block diagram of the crossbar switch, with the four different parts marked on the diagram.

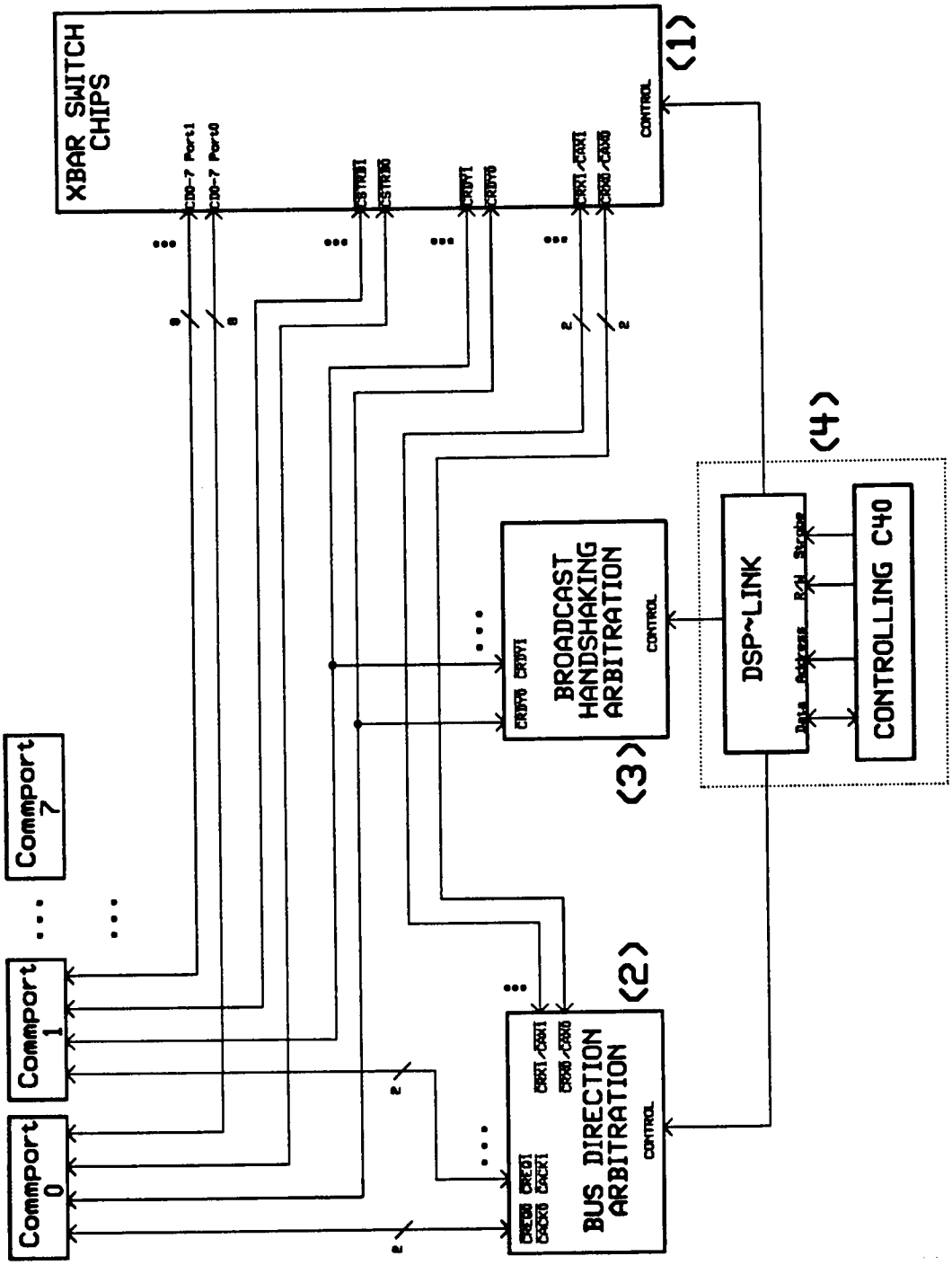


Figure 1 – Block Diagram of 8-Port Crossbar Switch

This thesis proposes original hardware designs to handle the data bus connections and the two handshaking arbitrations, and original software routines to control the system. The data bus connections are made by using LSI Logic L64270 64x64 Crossbar Switch VLSI chips in parallel. The bus arbitration handler and the data handshaking arbiter are implemented in large programmable logic devices (PLDs), the EPM5032 and the EPM5192 from Altera. The address decoder uses a standard 22V10 PLD. The control unit consists of a separate C40 processor with the software needed to configure and connect the switch. All the hardware for the switch was mounted on a VME-form printed circuit board, while the controlling C40 was connected via cable to the switch through DSP~Link [4], a memory expansion protocol interface.

C. Thesis Outline

Chapter II of this thesis gives background on the crossbar problem by illustrating a simple crossbar and crosspoint switch, then discussing the specific commport signals of the C40 to be switched in the crossbar. Next, the connection of the commport signals through the LSI Logic crossbar switch chips is discussed in Chapter III (Block #1 in Figure 1). The handling of the bus arbitration signals for the commports is the topic of Chapter IV (Block #2 in Figure 1), while Chapter V deals with the handling of the bus handshaking signals of the commports for broadcasts (Block #3 in Figure 1). Chapter VI explains the design and use of the controller for the crossbar switch, giving details on the hardware, software, and address decoding used (Block #4 in Figure 1).

The results of testing the switch both for functionality and data throughput rates are given in Chapter VII. Chapter VIII gives the conclusions of the design and

tests, a preview of possible future work on the crossbar switch and crossbar-type designs, and ideas on expansion to larger switches.

CHAPTER II

CROSSBAR SWITCH EXAMPLES AND COMMPORT BACKGROUND

A. Definition and Example of Crossbar Switch

A crossbar switch is a hardware bidirectional interconnection device in a multiprocessor or parallel processing system which allows connection between any two data ports of the different processors attached to the switch. There are a total of $\frac{N}{2}$ connections possible at a time, where N is the number of ports attached. For example, in an 8-port system, Port A can connect to any one of the other seven ports (taking up two ports), Port B can connect to any of the other 5, Port C can connect to one of the other 3 ports, and Port D can connect with the last port left. This gives a total of $\frac{8}{2} = 4$ connections in the switch.

The size of a crossbar is given as n x m, where n = number of data paths, and m = number of ports that can be reached through the switch [3]. The size can also be defined by n = number of input ports, and m = number of output ports of the crossbar [5].

A crossbar switch usually consists of a large multiplexer (mux) for each output port which selects an input port to connect to the output port (note that each port can consist of a single line or multiple lines which are switched together). Each output has access to each input in the switch. There is a memory register associated with each mux, which holds the address of the input port to be selected in the mux. Each mux also has its own address to signify when it is being configured. Thus, there are two sets of address lines – one for the mux and one for the input port selected in the mux. These two sets can be thought of as the output and input

addresses, respectively. The output of each mux can be put through a tri-state buffer which is enabled or disabled by a bit in its memory register.

An illustration might be helpful at this point. A diagram of an example 4x4 crossbar is shown in Figure 2(a). Suppose the user wants to connect input port 3 to output port 1. By placing the number 3 on the input address lines and number 1 on the output address lines, then accessing the crossbar (usually to strobe the data into the output mux's memory), the connection is made at the crossbar. This is illustrated in Figure 2(b).

It is also possible to connect the crossbar switch in what is known as 'bidirectional' mode [5]. In this mode, there are an equal number of input and output ports, and each input is associated with one output to become a single entity. The input and output are tied together and the output tri-state buffer is on when the port is used as an output and off when the port is used as an input. This cuts the number of ports in half, and since each port is bidirectional, the size of the switch is given by a single number N = number of ports. If we connect the example crossbar in this mode, there now are 4 bidirectional ports in the switch, as shown in Figure 2(c). There are still $\frac{N}{2}$ connections possible, though N is smaller.

A connection in this mode is very similar to the previous connection example. However, in the L64270 Crossbar Switch chip from LSI Logic used in this design, two access cycles must be used to make the connection. The first cycle disables the tri-state buffer of the port that is to become an input, while the second cycle makes the connection at the output mux. This is illustrated in Figure 3(a) and 3(b) with the Port3 - Port1 example in bidirectional mode.

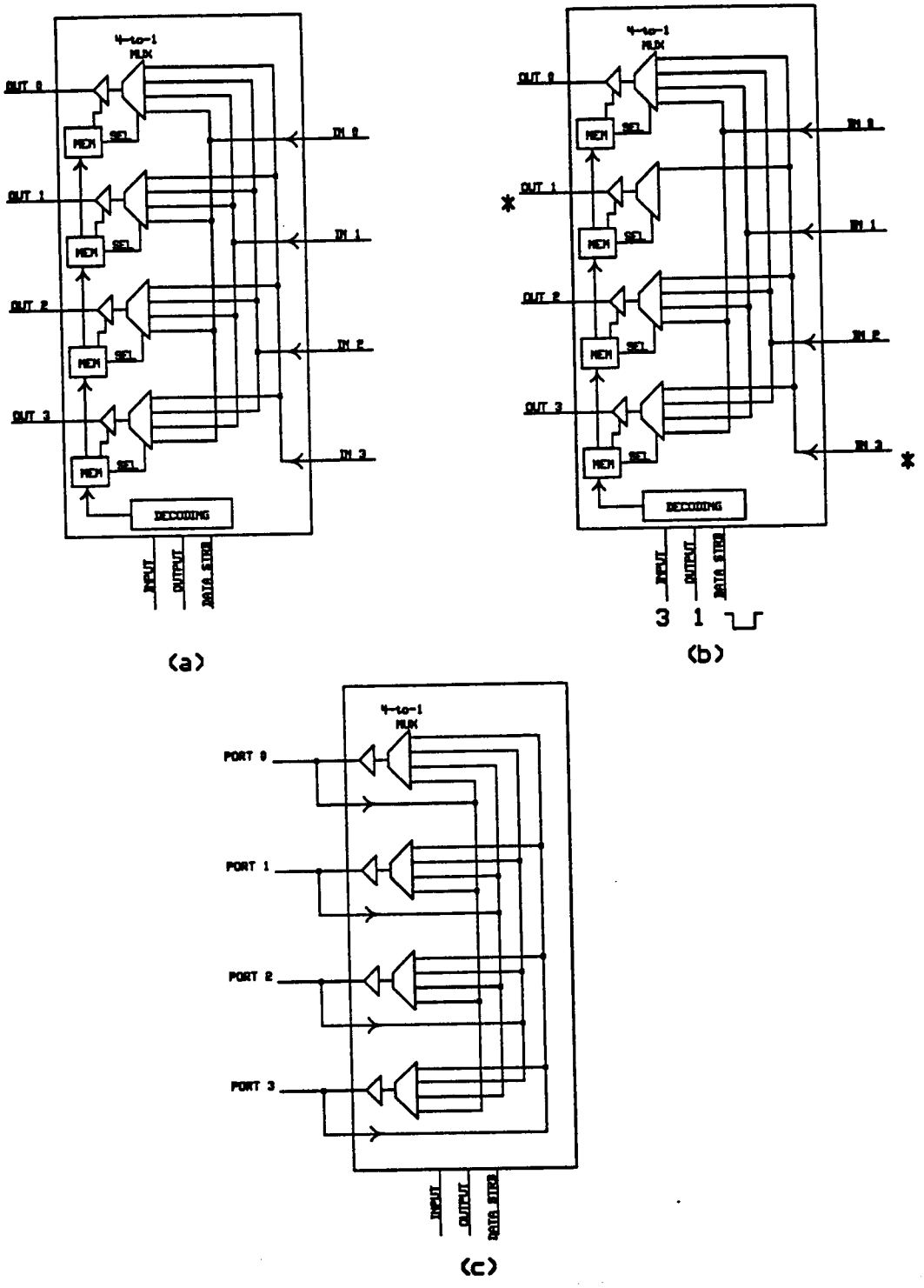


Figure 2 – Crossbar Examples (a) Example 4x4 Crossbar Switch (b) Example Connection (c) Bidirectional Mode

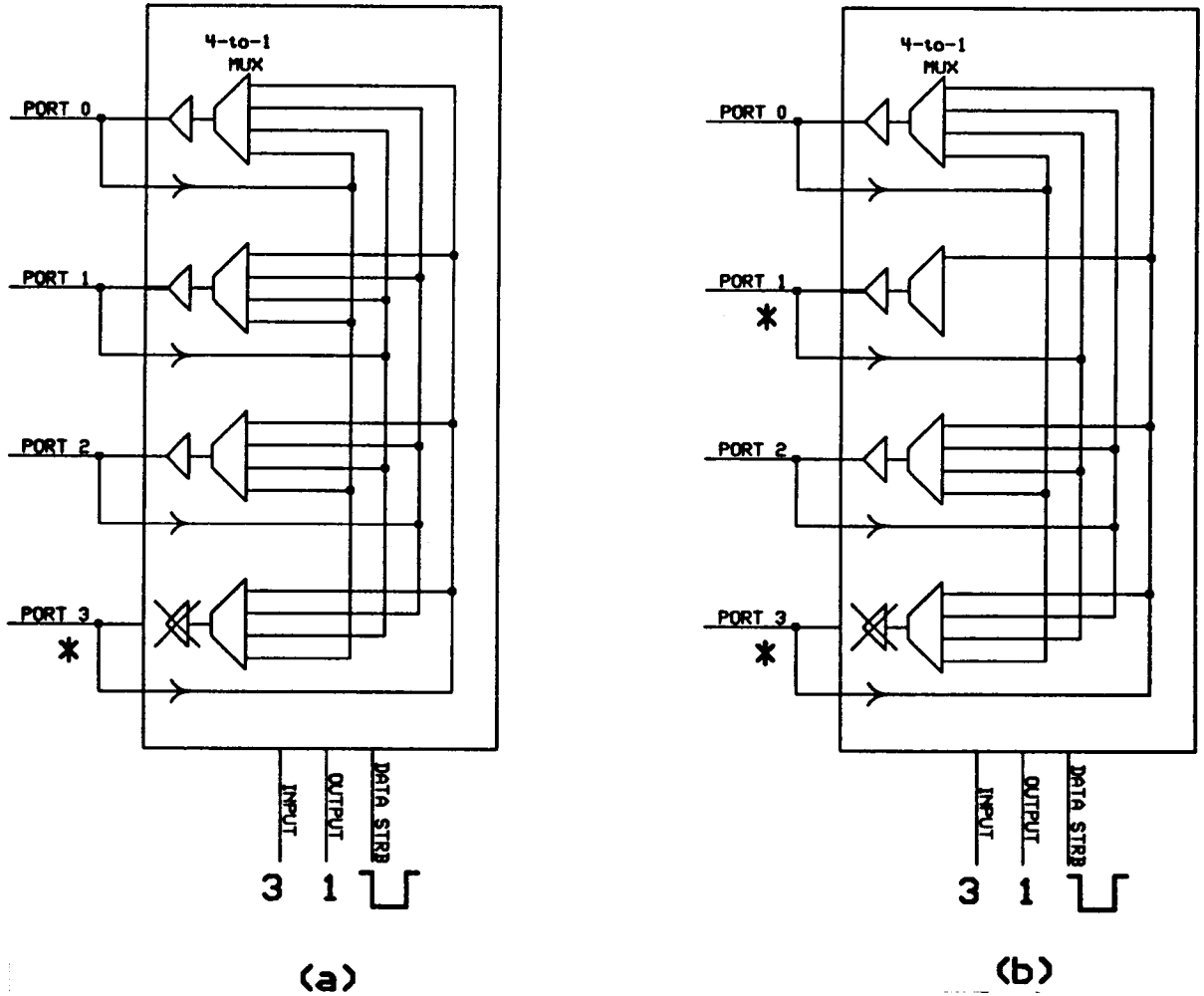


Figure 3 – Crossbar Bidirectional Connection

(a) 1st Cycle (b) 2nd Cycle

B. Definition and Use of Crosspoint Switch

A *crosspoint switch* allows any one of the ports attached to send its data simultaneously to as many other ports as desired. This is referred to as a 'broadcast'. The crosspoint switch can be implemented in the same device as the crossbar switch by selecting the same input port at multiple output muxes.

Another illustration shows this concept. Suppose in the 4x4 unidirectional switch, the user wants Port0 to broadcast to Port1 and Port3. This is accomplished by selecting 0 on the input address lines and 1 on the output address lines in the first cycle (Figure 4(a)), then again putting 0 on the input lines while specifying 3 on the output lines in the next access cycle (Figure 4(b)).

A similar approach is used configuring the bidirectional crosspoint switch. To make the Port0 to Port1 and Port3 broadcast, the user must first signal the switch to disable Port0's tri-state buffer, then make the Port0 - Port1 connection, and finally make the Port0 - Port3 connection. This is not illustrated as the concepts are the same as in previous examples.

C. Extension of Examples to Design

The design presented in this paper is an 8-port bidirectional mode crossbar and crosspoint switch integrated into one device. The entire switch system will be referred to as a 'crossbar switch' or just 'crossbar', and is abbreviated 'Xbar switch' or 'Xbar'. The 8 ports are commports from C40s, consisting of 12 lines each. These signals and the considerations in handling them are explained next.

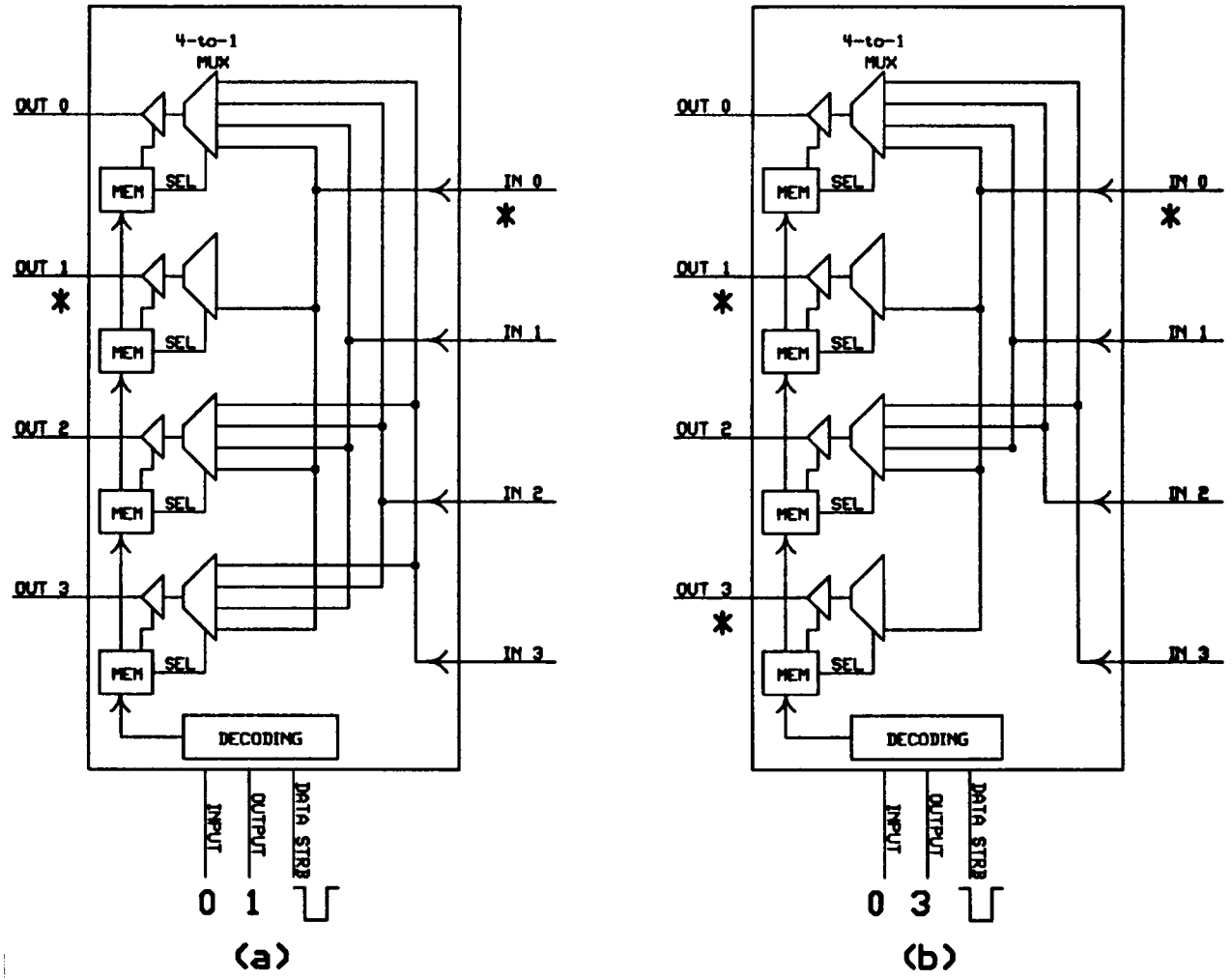


Figure 4 – Crosspoint Broadcast Connection

(a) Port0 - Port1 (b) Port0 - Port3

D. Communication Port Signals and Their Operation

The Texas Instrument TMS320C40 (C40) is a 32-bit floating point digital signal processor with the unique feature of having six communication ports (commports) on the chip. Each commport consists of twelve lines – eight data lines (CD0-CD7), a data strobe ($\overline{\text{CSTRB}}$), a data acknowledge ($\overline{\text{CRDY}}$), a bus request ($\overline{\text{CREQ}}$), and a bus acknowledge ($\overline{\text{CACK}}$). Each of these lines is bidirectional. The CD0-CD7, $\overline{\text{CSTRB}}$, and $\overline{\text{CACK}}$ lines are driven by the C40 when the commport is in output mode, with $\overline{\text{CRDY}}$ and $\overline{\text{CREQ}}$ being input lines. CD0-7, $\overline{\text{CSTRB}}$, and $\overline{\text{CACK}}$ are inputs, while $\overline{\text{CRDY}}$ and $\overline{\text{CREQ}}$ are outputs, when the commport is in the input state.

The $\overline{\text{CSTRB}}$ and $\overline{\text{CRDY}}$ lines provide common handshaking for the data and are referred to as data handshaking lines. $\overline{\text{CSTRB}}$ is driven low by the output port when valid data is present, and $\overline{\text{CRDY}}$ is asserted low by the input port when the data has been received. When the output port detects $\overline{\text{CRDY}}$, it removes $\overline{\text{CSTRB}}$, and another byte transfer can then take place. The input port removes its $\overline{\text{CRDY}}$ when $\overline{\text{CSTRB}}$ goes back high to ready itself for the next byte.

The C40 commports use the $\overline{\text{CREQ}}$ and $\overline{\text{CACK}}$ lines to arbitrate ownership of each data bus connection. Bus ownership is referred to as holding the bus 'token' and the bus direction changing is called 'token transfer' ([2] Chapter 8). A port which is in output mode owns the bus token. If an input port desires the bus, i.e. has data to output, it asserts $\overline{\text{CREQ}}$ low. When the output port detects $\overline{\text{CREQ}}$, it waits until it has finished transferring data or is idle. When ready, the output port then asserts $\overline{\text{CACK}}$ low, signaling that it relinquishes bus ownership. When the input port sees $\overline{\text{CACK}}$ low, it then drives $\overline{\text{CREQ}}$ back high. The positive-going edge of $\overline{\text{CREQ}}$ turns the commport lines around, i.e. from input to output and vice

versa.

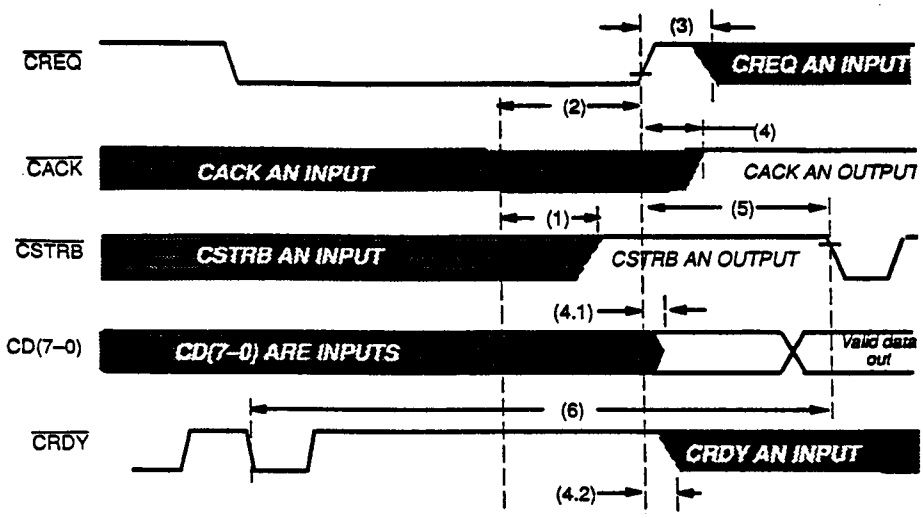
There is a short time after $\overline{\text{CREQ}}$ going high where $\overline{\text{CREQ}}$, $\overline{\text{CSTRB}}$, and $\overline{\text{CRDY}}$ are being driven high by both commports. This helps eliminate glitches in these lines. However, this overlap time, coupled with the fact that all the lines are changing direction, is cause for concern in the timing of the bus ownership transfer [6]. If one port gives up ownership too soon, glitches may occur in the data handshaking and bus arbitration lines. If a port is tardy in relinquishing ownership, one end may be driving the bus high at the same time as the other end is driving it low, which can cause damage to the C40 ([6] pp. 17-1,17-2 and [2] p. 8-15).

Timings for the commport bus ownership transfers are given in Figure 5 and Figure 6. Notice, however, that these are relative timings to the internal synchronizers of the C40's. As far as this author knows, no solid absolute timing is known for the bus token transfers. The timings given do provide some idea of what two commports would do if directly connected and this design tries to mimic the timing as nearly as possible.

E. Crossbar Considerations in Handling Commport Signals

On power-up, half of the C40's commports are automatically configured as output ports (commports 0, 1, and 2) and half as input ports (commports 3, 4, and 5) ([2] pp. 8-14 and 8-15). No two input or two output ports should be directly connected as this can damage the C40. The Xbar switch needs to provide isolation and go-between arbitration so that any two ports can be connected without damage.

Timing is a large problem in handling the bus arbitration. Bus turnarounds must be completed in such a way at both ends of a connection so that the two commports are not driving each other for significant lengths of time. This problem



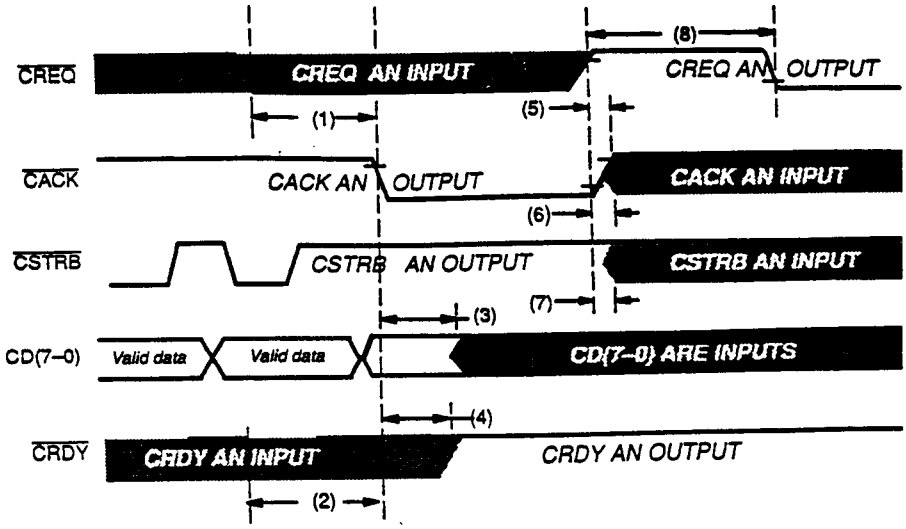
= When signal is an input (clear = when signal is an output).

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min†	Max†	Min†	Max†	
(1)†	$t_{d(AL-SO)T}$	$\overline{CACK}$ low to $\overline{CSTRB}$ change from input to a high-level output	0.5P+ 6	1.5P+22	0.5P+ 6	1.5P+ 22	ns
(2)†	$t_{d(AL-RQH)T}$	$\overline{CACK}$ low to start of $\overline{CREQ}$ going high for token request acknowledge	P + 5	2P + 20	P + 5	2P + 20	ns
(3)	$t_{d(RQH-RQI)T}$	Start of $\overline{CREQ}$ going high to $\overline{CREQ}$ change from output to an input	0.5P – 5	0.5P+13	0.5P – 5	0.5P+ 13	ns
(4)	$t_{d(RQH-AO)T}$	Start of $\overline{CREQ}$ going high to $\overline{CACK}$ change from an input to an output level high	0.5P – 5	0.5P+13	0.5P – 5	0.5P+13	ns
(4.1)	$t_{d(RQH-DO)T}$	Start of $\overline{CREQ}$ going high to CD(7–0) change from inputs driven to outputs driven	0.5P – 5	0.5P+13	0.5P – 5	0.5P+13	ns
(4.2)	$t_{d(RQH-RI)T}$	Start of $\overline{CREQ}$ going high to CRDY change from an output to an input	0.5P – 5	0.5P+13	0.5P – 5	0.5P+13	ns
(5)	$t_{d(RQH-SL)T}$	Start of $\overline{CREQ}$ going high to $\overline{CSTRB}$ low for start of word transfer out	1.5P – 8	1.5P+ 9	0.5P – 8	1.5P+ 9	ns
(6)	$t_{d(RL-SL)T}$	$\overline{CRDY}$ low at end of word input to $\overline{CSTRB}$ low for word output	3.5P+12	5.5P+48	3.5P+12	5.5P+ 48	ns

† These timing parameters result from synchronizer delays and are referenced from the falling edge of H1. The inputs (that cause the output-signal pins to change values) are sampled on H1 falling. The minimum delay occurs when the input condition occurs just before H1 falling, and the maximum delay occurs when the input condition occurs just after H1 falling.

Figure 5 – Commport Token Transfer to an Input Port

(From [2], p. 14-33 and 14-34)



= When signal is an input (clear = when signal is an output).

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)†	$t_{d(RQL-AL)T}$	$\overline{CREQ}$ low to start of $\overline{CACK}$ going low for token request acknowledge	P + 5	2P + 22	P + 5	2P + 22	ns
(2)†	$t_{d(RL-AL)T}$	$\overline{CRDY}$ low at end of word transfer out to start of $\overline{CACK}$ going low	P + 6	2P + 27	P + 6	2P + 27	ns
(3)	$t_{d(AL-CD)I}$	Start of $\overline{CACK}$ going low to CD(7–0) change from outputs to inputs	0.5P – 8	0.5P + 8	0.5P – 8	0.5P + 8	ns
(4)	$t_{d(AL-RO)T}$	Start of $\overline{CACK}$ going low to $\overline{CRDY}$ change from an input to output, high level	0.5P – 8	0.5P – 8	0.5P – 8	0.5P – 8	ns
(5)†	$t_{d(RQH-AQ)T}$	$\overline{CREQ}$ high to $\overline{CREQ}$ change from an input to output, high level	4	22	4	22	ns
(6)†	$t_{d(RQH-AI)T}$	$\overline{CREQ}$ high to $\overline{CACK}$ change from output to an input	4	22	4	22	ns
(7)†	$t_{d(RQH-SI)T}$	$\overline{CREQ}$ high to $\overline{CSTRB}$ change from output to an input	4	22	4	22	ns
(8)†	$t_{d(RQH-RQL)T}$	$\overline{CREQ}$ high to $\overline{CREQ}$ low for the next token request	P – 4	2P + 8	P – 4	2P + 8	ns

† These timing parameters result from synchronizer delays and are referenced from the falling edge of H1. The inputs (that cause the output-signal pins to change values) are sampled on H1 falling. The minimum delay occurs when the input condition occurs just before H1 falling, and the maximum delay occurs when the input condition occurs just after H1 falling.

Figure 6 – Commport Token Transfer From an Output Port

(From [2], p. 14-35 and 14-36)

is complicated even further by a lack of absolute timing specifications in the TI C40 data book. Suffice it to say that the timing constraints are a significant concern in switching the bus directions (section 8.7 of [2]).

In the broadcast mode, the output commport will have its $\overline{\text{CSTRB}}$ signal distributed to the receiving commports by the Xbar switch. However, all the $\overline{\text{CRDY}}$ lines from the input commports must be combined to produce a single $\overline{\text{CRDY}}$ at the output commport. The bus arbitration lines must be ignored since bus turn-around is not possible in broadcast mode.

As stated previously, the $\overline{\text{CREQ}}$ and $\overline{\text{CRDY}}$ signals flow in the opposite direction of the other commport signals. They must still connect to the same commport as the other 10 lines. However, at the Xbar level, when the 10 lines are specified as an input, these other two lines must be specified as an output, and vice versa. The next chapter explains how these connections are made through the Xbar switch.

CHAPTER III

COMMPORT LINE CONNECTIONS USING THE LSI LOGIC CROSSBAR SWITCH

A. Hardware Features of the Crossbar Switch Part

The actual connections in the Xbar switch are implemented using the LSI Logic L64270 64x64 Crossbar Switch chip. This chip allows any one of 64 outputs to be connected to any one of 64 inputs, to be set to a constant value (high or low), or to be tri-stated (disabled). Each output has a 64-to-1 mux and registers which hold the select line values to the mux as well as an enable/disable line for the output driver. The chip supports a bidirectional mode in which each input is connected to a corresponding output. This mode was described in Chapter II and will be used in this design.

The crossbar chip also allows for groups of inputs, outputs, or bidirectional lines to be treated as single entities, called busses. A bus on the crossbar chip must be a power of two in size, ranging from width 1 to width 64. When busses are to be used, all the lines associated with a bus are connected at once. For example, if input bus A, consisting of lines A.0-A.7 (8 lines), is to be connected to output bus B (also 8 lines wide and denoted by B.0-B.7), then after the connection is made, line A.0 is connected to B.0, A.1 to B.1, etc. This simplifies the connection process, as only one address for A and one for B are needed, as opposed to each separate line needing to be addressed for its connection. For example, if busses are to be 4 lines each, there are only 16 addresses needed, one for each bus ($4 \frac{\text{lines}}{\text{bus}} * 16$ busses $= 64$ lines). This design will use 8 eight-bit busses on each crossbar chip.

The crossbar chip must be initialized once following reset. The same control

lines are used for both initialization and for making subsequent connections. During initialization, the mode of the chip (bidirectional or not) and the bus width (1 to 64) are specified. For a connection, the input and output addresses corresponding to the lines or busses to be connected are strobed into the appropriate memory locations.

The crossbar chip has 19 control lines which carry this initialization and connection (configuration) information. These 19 lines are:

1. CI.0-CI.7 – Control Input. *Initialization*: CI.0-CI.5 specify the bus width, CI.6 specifies the mode (bidirectional or not), and CI.7 is not used. *Connections*: CI.0-CI.5 are used to give the input bus address, while CI.6 and CI.7 are held low.
2. REGADR.0-REGADR.6 – Register Address. *Initialization*: REGADR.6 is asserted to indicate initialization is taking place, while REGADR.0-REGADR.5 are 'don't cares'. *Connections*: REGADR.0-REGADR.5 are used to specify the output bus address, while REGADR.6 is 0.
3. $\overline{\text{WE}}$ – Write Enable. This signal is used to strobe both initialization and connection data into the crossbar memory.
4. BDCYC1 – Bidirectional Cycle 1. *Connections only*: The bidirectional mode requires two accesses, or cycles, to the crossbar chip per connection. The first cycle is used to disable the driver of the output associated with the input to be connected (remember the two are connected in bidirectional mode). The second cycle strobes the input bus address into the select line registers of the 64-to-1 mux or muxes associated with the output or outputs. If BDCYC1 is high, the crossbar chip assumes the first cycle is being made, while BDCYC1 low indicates the second cycle is in

progress. BDCYC1 is not used in initialization.

5. BNKLDI – Bank Load. *Connections only*: If BNKLDI is low during connection cycles, the connection data is held in an auxiliary memory bank. As soon as BNKLDI goes high, this data for all connections is transferred simultaneously to the individual registers of the outputs. BNKLDI is ignored during initialization.
6. $\overline{\text{OE}}$ – Output Enable. If high, all output drivers on the crossbar chip are disabled. If low, each output driver is individually enabled or disabled, according to its own register.

For further information on these signals, see page 187 of [5].

B. Initialization and Configuration of the Crossbar Chip

To initialize the crossbar chip, REGADR.6 is asserted and the bus width to be used is specified on CI.0-CI.5. CI.6 is asserted to indicate bidirectional mode is desired, while REGADR.0-REGADR.5 and CI.7 are don't cares during the initialization. With the desired values on CI.0-CI.6, $\overline{\text{WE}}$ is pulled low to enter the information into the crossbar chip, as shown in Figure 7.

After the crossbar chips are initialized, connections are made in the bidirectional mode by making two access cycles to the chips. During the first cycle, BDCYC1 is held high while specifying the output bus address on REGADR and the input bus address on CI, and $\overline{\text{WE}}$ is pulsed low. This first cycle disables the tri-state buffers of the output bus. In the next cycle, BDCYC1 is taken low, REGADR and CI remain the same, and $\overline{\text{WE}}$ is pulsed low again. This cycle makes the desired input/output connection. If desired, BNKLDI can be held low during both cycles, then pulled high any time after completion to latch the configuration. This allows

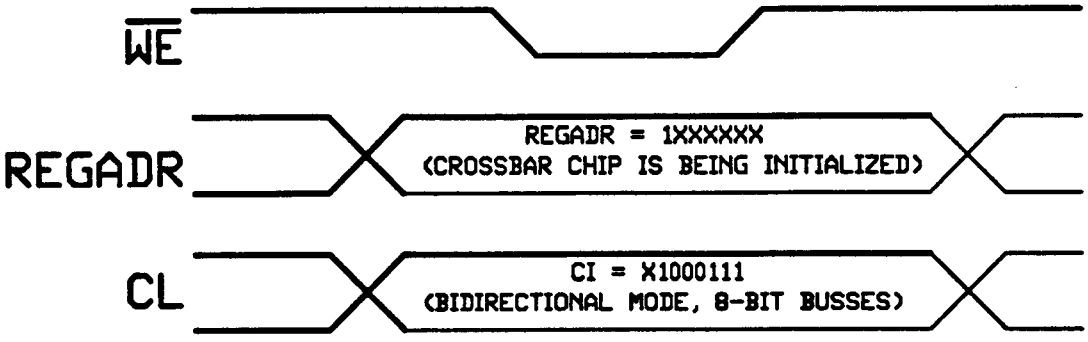


Figure 7 – Crossbar Chip Initialization Sequence

for multiple connections specified at the crossbar to be latched in at the same time. If BNKLDI is high, the connection information goes directly to the outputs' registers. Figure 8 shows the connection sequence with BNKLDI being used to latch the connection in after it is completely specified.

C. Parallelization of the Crossbar Chips

There are two problems in using these chips to switch more than 64 commport lines. First, as per the LSI Logic application note ([5] p.195), it can be shown that to fully interconnect L lines requires 2^m chips, where $m = \lceil \frac{L}{64} \rceil$. If the commports in the 8-port system were to be fully interconnectable, then since $L = (8 \text{ ports}) * (12 \frac{\text{lines}}{\text{port}}) = 96$ lines and $m = \lceil \frac{96}{64} \rceil = 2$, $2^2 = 4$ crossbar chips would implement the system. This is a decent figure for the 8-port Xbar switch, but would present a problem in expanding to larger switch sizes. For example, in a 32-port system, $L = 384$, $m = 6$, meaning $2^6 = 64$ chips are required, which would be

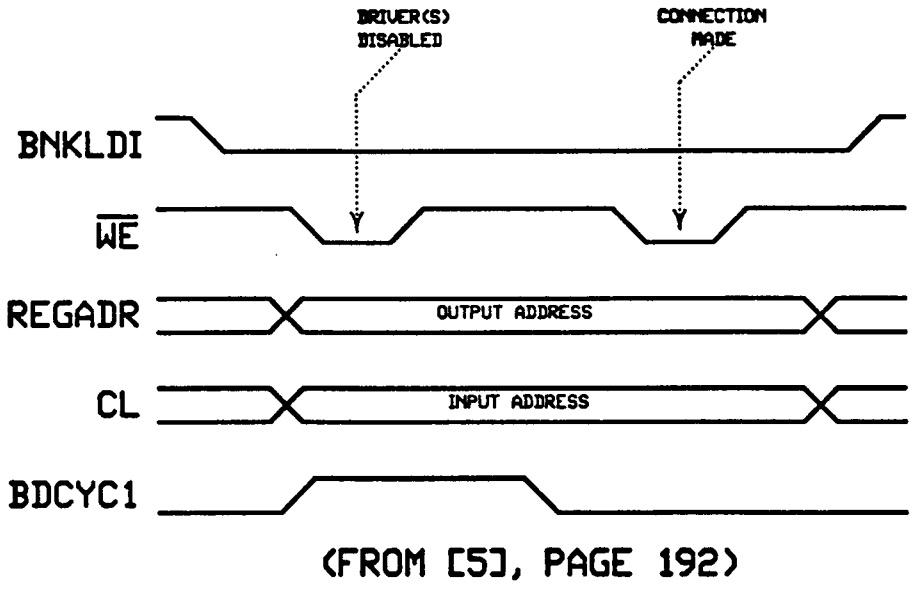


Figure 8 – Crossbar Connection Configuration Sequence

impractical. Another problem arises because the $\overline{\text{CREQ}}$ and $\overline{\text{CRDY}}$ signals on each commport go opposite to the other signals. This normally would require separate crossbar chips with separate configuration commands, increasing both the size of the switch and connection time. However, as will now be discussed, it is possible to parallel connect the control lines for all the crossbar chips. Parallelization of the crossbar chips in this way allows for simplification both in the number of chips used and the time it takes to configure them.

There are two steps used to achieve this simplification through parallelization. The first step consists of splitting up the commport signals into 3 groups – the data lines CD0-CD7, $\overline{\text{CSTRB}}$ and $\overline{\text{CACK}}$, $\overline{\text{CREQ}}$ and $\overline{\text{CRDY}}$ – and switching each group through its own crossbar chip. The second step is to require that each chip receive the same control signals at the same time, with one modification for the chip which handles $\overline{\text{CREQ}}$ and $\overline{\text{CRDY}}$ which allows these two lines to flow in the opposite direction of the other 10 lines.

In this grouping of signals, the 8 data lines from each commport are the largest group being switched through the crossbar together, thereby mandating that 8-bit busses be used for all the crossbar chips if their control lines are to be parallel connected. This means, for the data line crossbar chip, that all 64 crossbar lines are used (8 busses $* 8 \frac{\text{lines}}{\text{bus}} = 64$ lines). The other two groups only have 2 signals from each commport, and therefore do not use their chips to the fullest. However, in larger systems, these other two groups will use their chips more fully, especially in a 32-port switch. Because these groups do not use the entire bus, the signals must be placed at the same position on every bus to ensure they will be connected correctly. For example, $\overline{\text{CREQ}}$ from Port0 must occupy the lowest bit of the first bus, $\overline{\text{CREQ}}$ from Port1 must occupy the lowest bit of the second bus, etc.

As for the control signals, notice that if two busses are connected in one chip, and the same control signals are sent to another chip, the busses at the same addresses on the second chip will be connected in exactly the same manner as the first chip. Thus the data lines, $\overline{\text{CSTRB}}$, and $\overline{\text{CACK}}$ can all be connected between two commports (or more during broadcasts) at the same time by using the exact same control signals on both chips. However, notice that if two ports have a input/output relationship for the data, $\overline{\text{CSTRB}}$, and $\overline{\text{CACK}}$ lines, they will have an *output/input* relationship for the $\overline{\text{CREQ}}$ and $\overline{\text{CRDY}}$ lines – exactly switched. The bus addresses at the chip are still the same, just the direction has changed.

Thus, if the input address control lines of the CD0-CD7 and $\overline{\text{CSTRB}}/\overline{\text{CACK}}$ crossbar chips (the CI lines) are sent to the output address lines of the $\overline{\text{CREQ}}/\overline{\text{CRDY}}$ crossbar chip, while the output address lines of the first two chips (the REGADR lines) are sent to the input address lines of the third chip, it will switch the bus relationship from input/output at the CD0-CD7 and $\overline{\text{CSTRB}}/\overline{\text{CACK}}$ crossbar chips to output/input at the $\overline{\text{CREQ}}/\overline{\text{CRDY}}$ chip.

Both REGADR.6 and CI.6 are asserted during initialization and deasserted during connections, while during initialization REGADR.0 through REGADR.5 are don't cares and CI.0-CI.5 specify the bus width to be used. During configuration, REGADR.0-REGADR.5 specify the output bus of the crossbar to be connected to the input bus, and the input bus is addressed by CI.0-CI.5. Thus, by switching these two control signal groups bit for bit at the $\overline{\text{CREQ}}/\overline{\text{CRDY}}$ chip from the data and $\overline{\text{CSTRB}}/\overline{\text{CACK}}$ chips, a crossbar bus which is an input for CD0-CD7 and $\overline{\text{CSTRB}}/\overline{\text{CACK}}$ (specified by CI at these two chips) will be an output bus for the $\overline{\text{CREQ}}$ and $\overline{\text{CRDY}}$ signals (specified by REGADR at the $\overline{\text{CREQ}}/\overline{\text{CRDY}}$ chip), and vice versa. The switching of these two sets of crossbar control signals and the splitting up of the commports signals to different crossbar chips is shown in Figure 9. Notice that CI.7 is a don't care during initialization and a 0 otherwise, so it can be permanently grounded.

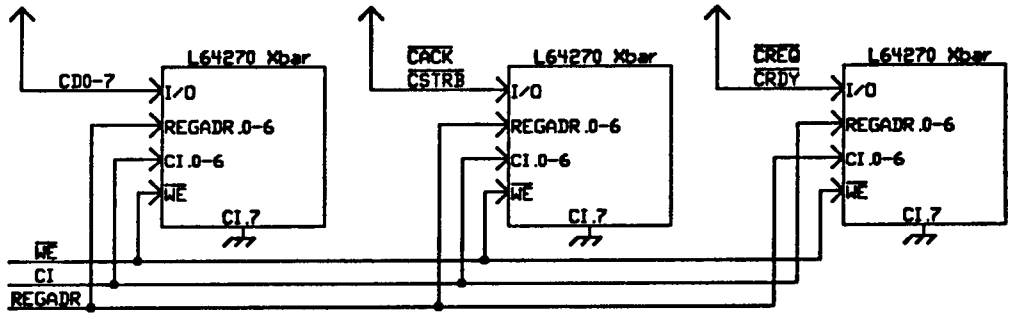


Figure 9 – Crossbar Chip Utilization

It is the symmetry of the CI and REGADR signals during connection sequences that allows for this bit-for-bit change between the two sets. Notice that the fact that REGADR.0-REGADR.5 are don't cares during initialization, and that CI.6 and REGADR.6 are equal at all times, allows for the change to be made at all times, not just during connections. If this ability to switch the REGADR and CI lines was not available, it would be necessary to configure the opposite-going crossbar chip separately by address decoding and separate connection sequences.

This implementation uses only three crossbar chips as compared to the four required in the calculations above. It will be shown later in Chapter VIII that a 32-port system can be accomplished with only 6 crossbar chips in this manner, as compared to the 64 chips that would be required as calculated above. This setup also allows only similar signals to reach each other (data lines to data lines, strobes to strobes, etc.). In addition, by handling the control signals in this manner, minimum configuration time is achieved. It requires at least two cycles to connect any two lines or busses, and by making the connections in parallel, an entire commport-to-commport connection is achieved in two cycles.

CHAPTER IV

HANDLING OF BUS ARBITRATION SIGNALS $\overline{\text{CREQ}}$ AND $\overline{\text{CACK}}$

The $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handling unit provides bus ownership arbitration between commports to ensure that desired connections can be made correctly without damaging the C40 commports. This allows the parallel processing system to be configured in an arbitrary topology and to modify the topology while the system is operating. The arbitration signal handling is done by a state machine associated with each port which keeps track of the state of the port and provides necessary signals to the port and to other state machines to arbitrate bus ownership correctly.

A. State Machine Considerations and Design

The main purpose of the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler is to receive from and provide to the commports the bus arbitration signals necessary to configure the Xbar as desired. The handler can be thought of as an intermediary which can request and grant bus ownership tokens, or merely pass them between connected commports.

For example, suppose Port0 and Port 1 are presently output ports and it is desired to connect the two with Port0 as output to Port1 as input. In this case, both Port0 and Port1 initially hold tokens. After the handler receives this configuration request from the controller, it must assert $\overline{\text{CREQ}}$ to Port1 to request Port1's token. The state machine then waits for Port1 to return $\overline{\text{CACK}}$, thereby releasing its token. The handler then removes $\overline{\text{CREQ}}$ to put Port1 in the desired state, and signals the controller to go ahead with the specified connection. Port0 holds the token for the connection after it is made.

A similar situation exists when an input port is to be changed to an output port. In this case, the handler must wait for the port to request a token and then grant the request to the port before giving the go-ahead to the control unit to make the connection.

For normal token transfers between two already connected ports, the state machine merely passes the request on after assuring that the control unit has also detected the request and is making the necessary changes at the crossbar chips to turn the direction around. The state machine then waits for the acknowledge to return and passes it on to the commport. By waiting for the control unit to detect the request before allowing it to pass on, the Xbar can make the changes and have them ready to be latched in (by use of BNKLDI at the crossbar chips) by the time the state machine has finished passing on the acknowledge to the commport. The direction change in the Xbar is then latched in as quickly as possible after the commports change direction, thereby avoiding the glitch or electrical damage problems.

The state machine for a given commport has no indication of what other state machine it must communicate with *a priori*. Therefore, there are accessory signals used which are passed on from the state machine to the crossbar chip network where the connections can be made. These signals are called $\overline{\text{CRX}}$ and $\overline{\text{CAX}}$ (the 'X' at the end identifies them as signals passed through the Xbar chips), which correspond to $\overline{\text{CREQ}}$ and $\overline{\text{CACK}}$ respectively. Pull-up resistors are used on $\overline{\text{CREQ}}$, $\overline{\text{CACK}}$, $\overline{\text{CRX}}$, and $\overline{\text{CAX}}$ to eliminate switching glitches and provide a default HIGH state. An example connection of two commports through the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler is shown in Figure 10.

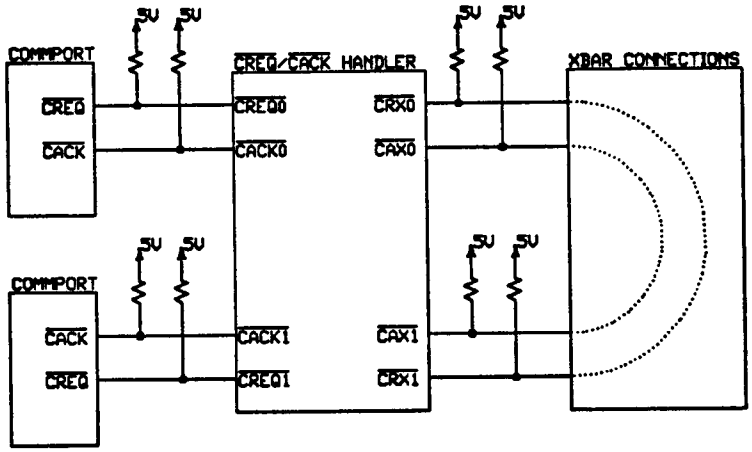


Figure 10 – $\overline{\text{CREQ}}/\overline{\text{CACK}}$ Connection Example

B. Implementation of the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ Handler

Each port has its own state machine to handle its $\overline{\text{CREQ}}$ and $\overline{\text{CACK}}$ signals. Figure 11 gives a flow diagram of this state machine. All eight state machines are connected with proper address decoding which indicates if the associated port is to be configured as input or output. For the eight state machines, three lines of input address and three of output address are needed. These lines come from REGADR.5,4,3 and CI.5,4,3, respectively (remember that an input commport is an output at the crossbar chips, and vice versa). There are also two other input lines, 'O' and 'I', which indicate the state of each commport when it is first switched through the Xbar (the state machine keeps track of the state afterwards, but does not know the port's state initially). If 'O' is asserted when the port's state machine is addressed by output address lines, the port is in the output state already.

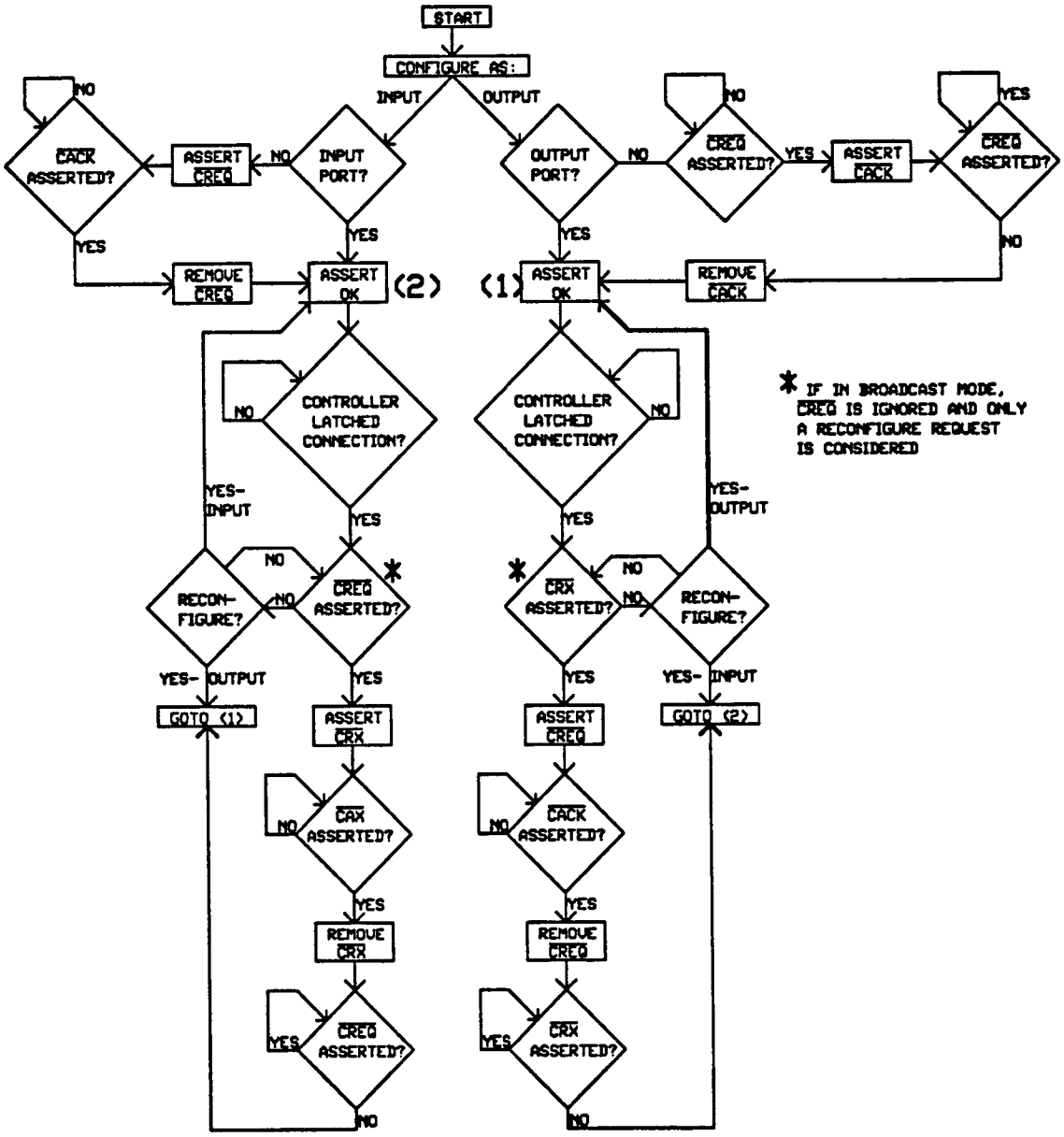


Figure 11 – $\overline{\text{CREQ}}/\overline{\text{CACK}}$ State Machine Flow Diagram

If not asserted, it is in the input state and must be turned around to achieve the desired configuration. Similarly, if 'I' is asserted when the port is to be an input, the state machine need not do anything; however, if 'I' is not asserted, the state machine must allow the port to become an output before signaling readiness.

The latching signal used at the crossbar chips (BNKLDI) is also brought into the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler. This allows the state machine to detect when a connection has been or is being made. Two other lines, designated as CONFR and BRDCST, indicate to the state machines when a new configuration is desired and when the Xbar is in broadcast mode, respectively. More will be said on these in Chapter VI on decoding. The state machines use a 20-Mhz clock signal for state transitioning.

Each state machine provides a handshake signal ('OK') to the control unit. This signal is asserted when the commport has been placed in the correct state and the port is ready to be connected. The control unit reads this signal from the handler by asserting 'RDOK' at the handler, and assures the correct OKs are asserted before latching in a connection in the Xbar. Figure 12 shows the block diagram of the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler.

It was necessary to implement this $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler on a large PLD because of the register-intensive nature of the design. An ALTERA EPM5192GC-1 in the 84-pin PGA package was used.

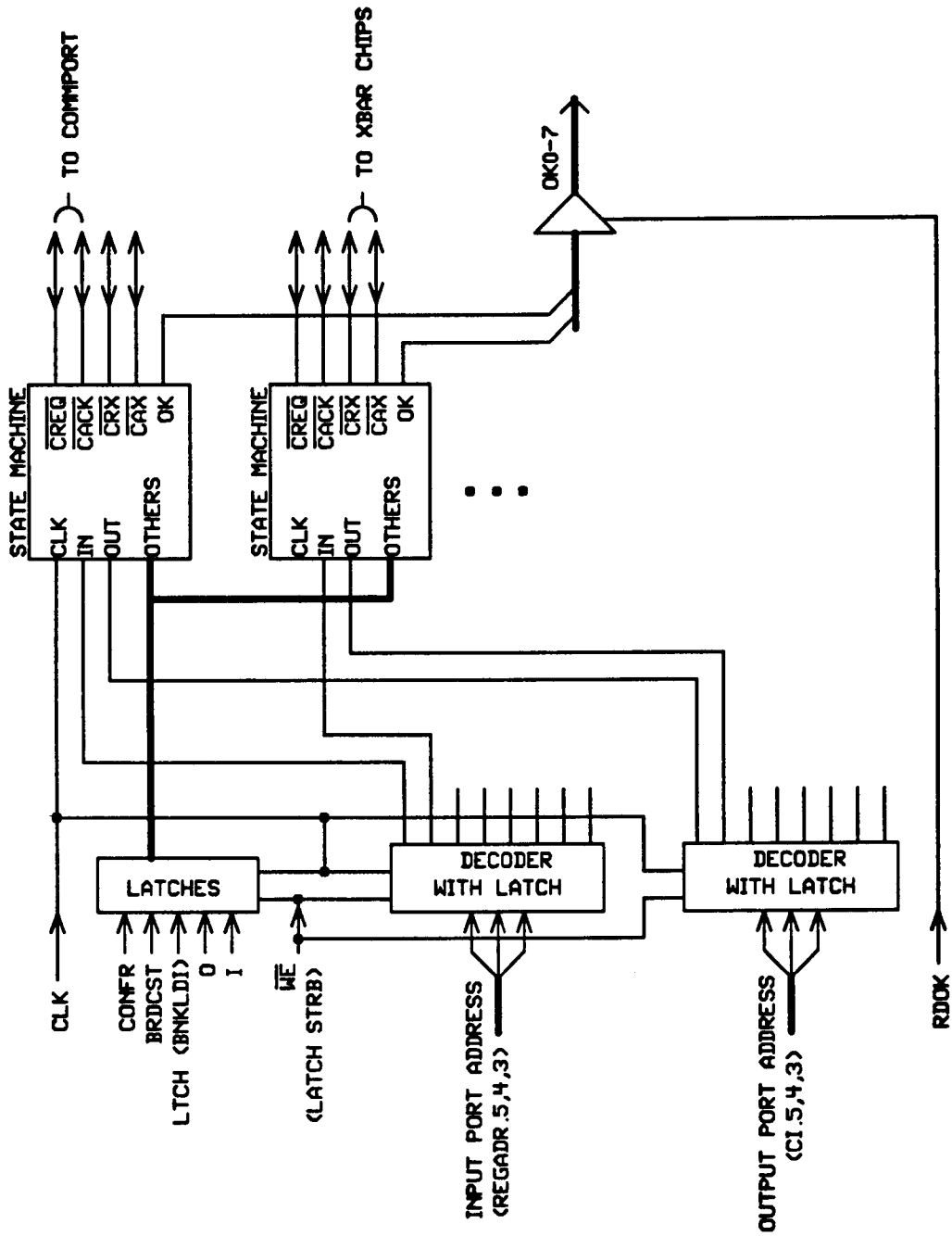


Figure 12 – Block Diagram of $\overline{\text{CREQ}}/\overline{\text{CACK}}$ Handler

CHAPTER V

BROADCAST ARBITRATION OF $\overline{\text{CRDY}}$ SIGNALS

A. Handling of Commport Signals to Accomplish Broadcasts

The C40 commports can accomplish a one-to-many broadcast by sending the data and data strobe signal from the broadcasting port to all the receiving ports, then logically ORing all the return $\overline{\text{CRDY}}$ signals together to produce the $\overline{\text{CRDY}}$ at the broadcast port. The logical OR assures that all receiving ports have given the ready signal before asserting $\overline{\text{CRDY}}$ to the broadcast port, according to the equation:

$$\overline{\text{CRDY}}(\text{broadcast}) = \overline{\text{CRDYa}} + \overline{\text{CRDYb}} + \overline{\text{CRDYc}} \ldots$$

which is logically equivalent to (by DeMorgan's Theorem):

$$\text{CRDY(low)} = \text{CRDYa(low)} \bullet \text{CRDYb(low)} \bullet \text{CRDYc(low)} \bullet \ldots$$

Furthermore, during the broadcast, the $\overline{\text{CREQ}}$ and $\overline{\text{CACK}}$ signals are ignored, so that bus turnaround is not possible. This idea is presented on page 13-37 of [2] and is shown in Figure 13.

The crosspoint part of the Xbar switch allows implementation of this idea. It was decided to allow only one port to broadcast at a time in this system. The broadcasting port's data and strobe signals are routed through the Xbar to as many of the other seven Xbar outputs as desired. This accomplishes a buffered broadcast of the data, as each crossbar chip output has its own driver.

It is also necessary to selectively OR the $\overline{\text{CRDY}}$ lines from the receiving

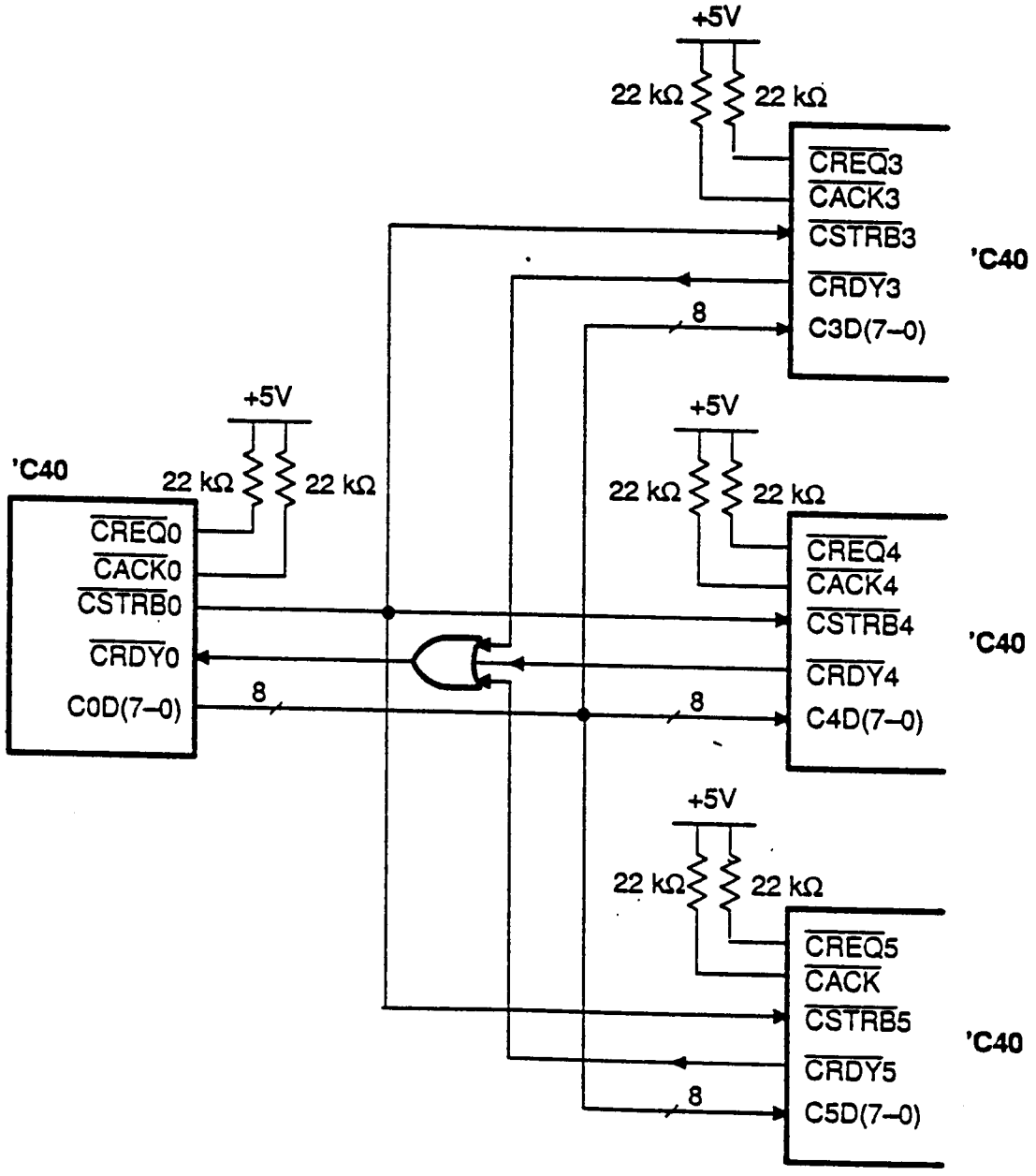


Figure 13 – Broadcasting From One C40 to Many C40s

(From [2] p. 13-37)

commports to create the $\overline{\text{CRDY}}$ at the broadcast port. $\overline{\text{CRDY}}$ signals from commports not involved in the broadcast must be ignored. The configuration is not known beforehand, so the selection and ORing of signals must be done in the system. The $\overline{\text{CRDY}}$ arbitration unit handles the selective ORing of these signals.

B. The $\overline{\text{CRDY}}$ Arbitration Unit

The $\overline{\text{CRDY}}$ arbitration unit ORs together any desired combination of $\overline{\text{CRDY}}$s from the commports to produce a $\overline{\text{CRDY}}$ at the specified broadcasting commport. This is done by enabling the $\overline{\text{CRDY}}$s that are to be used, masking out the unused $\overline{\text{CRDY}}$s, and driving the broadcasting port's $\overline{\text{CRDY}}$ as an output from the arbitration unit.

This $\overline{\text{CRDY}}$ arbitration unit consists of eight blocks, one for each port. Figure 14 shows the basic block. The block generates the $\overline{\text{CRDY}}$ for its associated port when it is broadcasting, and masks the port's $\overline{\text{CRDY}}$ either to enable it if the port is to receive broadcast data or to disable if the port is not to receive broadcast data.

When the port is to broadcast, the tri-state buffer attached to the $\overline{\text{CRDY}}$ signal is enabled. The tri-state enable signal is held in a flip-flop and is set when the ADR signal for the block is high and a strobe is given on $\overline{\text{WE}}$.

When the port is not broadcasting, its $\overline{\text{CRDY}}$ is ANDed with a masking signal to create $\overline{\text{CR}}$. The masking signal is held in the flip-flop which is clocked by $\overline{\text{WE}}$ and has the DATA line coming into it. If DATA is clocked in high, the $\overline{\text{CRDY}}$ signal is enabled, meaning $\overline{\text{CR}} = \overline{\text{CRDY}}$ and the port is receiving the broadcast. If the port is not receiving the broadcast (its $\overline{\text{CRDY}}$ is to be ignored), then DATA is clocked in low, which means the $\overline{\text{CR}}$ signal will always be low.

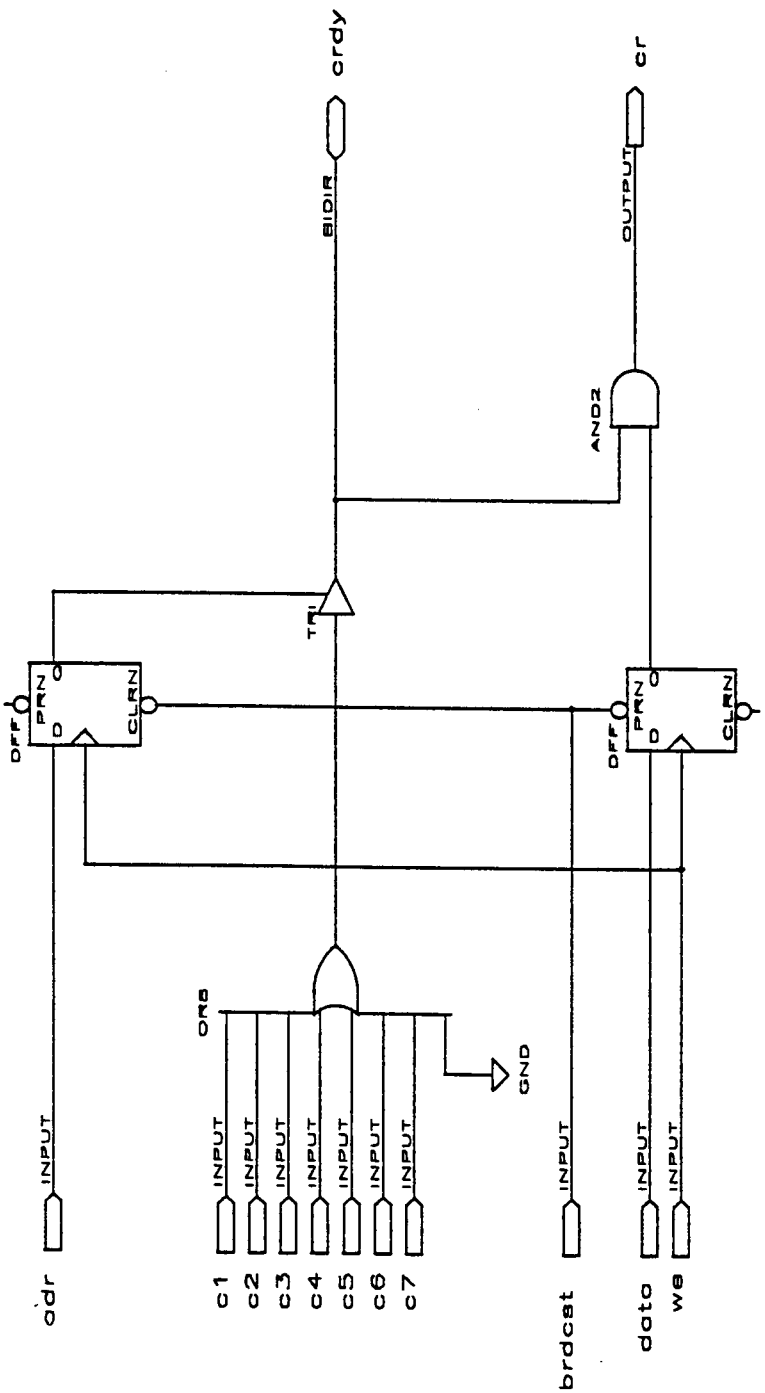


Figure 14 – $\overline{\text{CRDY}}$ ORing Block

The $\overline{CR}$ signals from all other ports (C1-C7) are brought into the block and ORed together. If a $\overline{CR}$ signal is disabled, it will always be low and therefore have no effect on the output of the OR gate (since A + 0 = A). This ORing then produces the desired $\overline{CRDY}$ signal for the broadcast port.

The BRDCST signal into the block disables all the tri-state buffers (the enable flip-flop is cleared) and keeps all the $\overline{CR}$s high (since the $\overline{CRDY}$s are high when not in use and the masking flip-flop is set high) when it is not asserted low. The $\overline{CR}$s must be held high till the $\overline{CRDY}$ arbitration unit is configured, so that when the broadcast port's $\overline{CRDY}$ driver is enabled, the $\overline{CRDY}$ to the broadcast port starts in the high state and does not give a false low strobe, as it might if any of the $\overline{CR}$s were low. When BRDCST is asserted high, meaning the Xbar is in broadcast mode, the flip-flops can be clocked with the desired values.

The eight ports' blocks are combined on one PLD (the Altera EPM5032DC-15) to create the Xbar switch's $\overline{CRDY}$ arbitration unit, shown in Figure 15. Each block's ADR signal is created by decoding the lines ADR0-ADR2, and the block's DATA signal comes from the one of the 8 lines DATA0-DATA7. The $\overline{WE}$ and BRDCST signals are common to all blocks.

C. Configuration of $\overline{CRDY}$ Arbiter with Example

To configure the $\overline{CRDY}$ arbiter, the broadcasting port's address is placed on lines ADR0-ADR2, and the bits corresponding to the receiving ports are set in the data byte DATA0-DATA7. If the n^{th} DATA bit is set, the n^{th} port is a receiver. The control unit assures that BRDCST is asserted, then pulses the $\overline{WE}$ line to latch in the configuration.

As an example, suppose that Port2 is to broadcast to Ports 0, 1, 5, and 7. The

control unit would set ADR = 010b (binary) = 2, which is Port2's address, and set the DATA byte to 10100011b. Note that the 0th, 1st, 5th, and 7th bits are set, corresponding to the ports which will receive the broadcast. The control unit then pulses the write signal to the $\overline{\text{CRDY}}$ arbitration unit, which enables Port2's $\overline{\text{CRDY}}$ output driver, while Ports 0, 1, 5, and 7 have their $\overline{\text{CR}}$ signals enabled. Port2's $\overline{\text{CRDY}}$ will now only be asserted low when all four of the receiving ports assert their $\overline{\text{CRDY}}$ signals low. Since the other ports (ports 3, 4, and 6) are not used, their $\overline{\text{CR}}$ signals remain constantly low and do not affect the ORing of all seven $\overline{\text{CR}}$s at Port2.

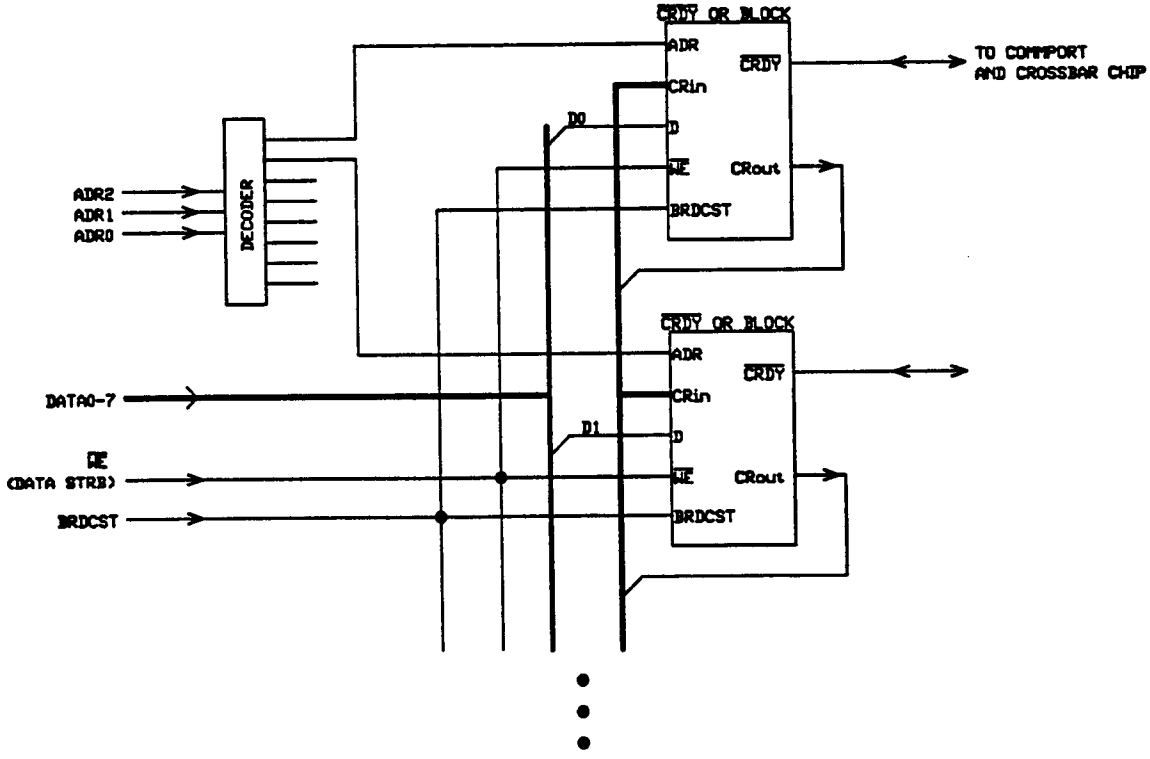


Figure 15 – $\overline{\text{CRDY}}$ Arbitration Unit Block Diagram

CHAPTER VI

CONTROL UNIT AND ADDRESS DECODING OF CROSSBAR SWITCH

A. Control Unit Purposes and Hardware

The control unit has several tasks. First, it initializes the crossbar chips. Second, it configures the Xbar switch by passing connection information addressed to the correct parts on the Xbar board. Next, it reads the OK signals from the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler to assure the ports are ready to be connected, and then latches in the connections at the Xbar. If not in broadcast mode, the controller then monitors the $\overline{\text{CREQ}}$ lines to check for bus turnaround requests. If a $\overline{\text{CREQ}}$ is asserted, the controller sends a new connection command to the Xbar and again checks for the correct OK signals before latching in the connection. If in broadcast mode, the controller disables the $\overline{\text{CRDY}}/\overline{\text{CREQ}}$ crossbar chip so the $\overline{\text{CRDY}}$ arbiter can properly function. The control unit must be ready in all cases to reconfigure the Xbar if so requested by the parallel processing system. A flow diagram of the controller is given in Figure 16.

The control unit of the Xbar consists of an off-board C40 with address lines, data bus, and read/write signals brought onto the Xbar board through a DSP~Link interface. DSP~Link is a memory expansion protocol interface consisting of 15 address lines, 16 data lines, a data strobe, and a read/write signal (R/W). For detailed information on DSP~Link, the reader is referred to [4]. These signals are brought from a PC-host C40 board to the Xbar with cable and buffered on the Xbar board.

The off-board C40 is used for a couple of reasons. First, it is compatible with

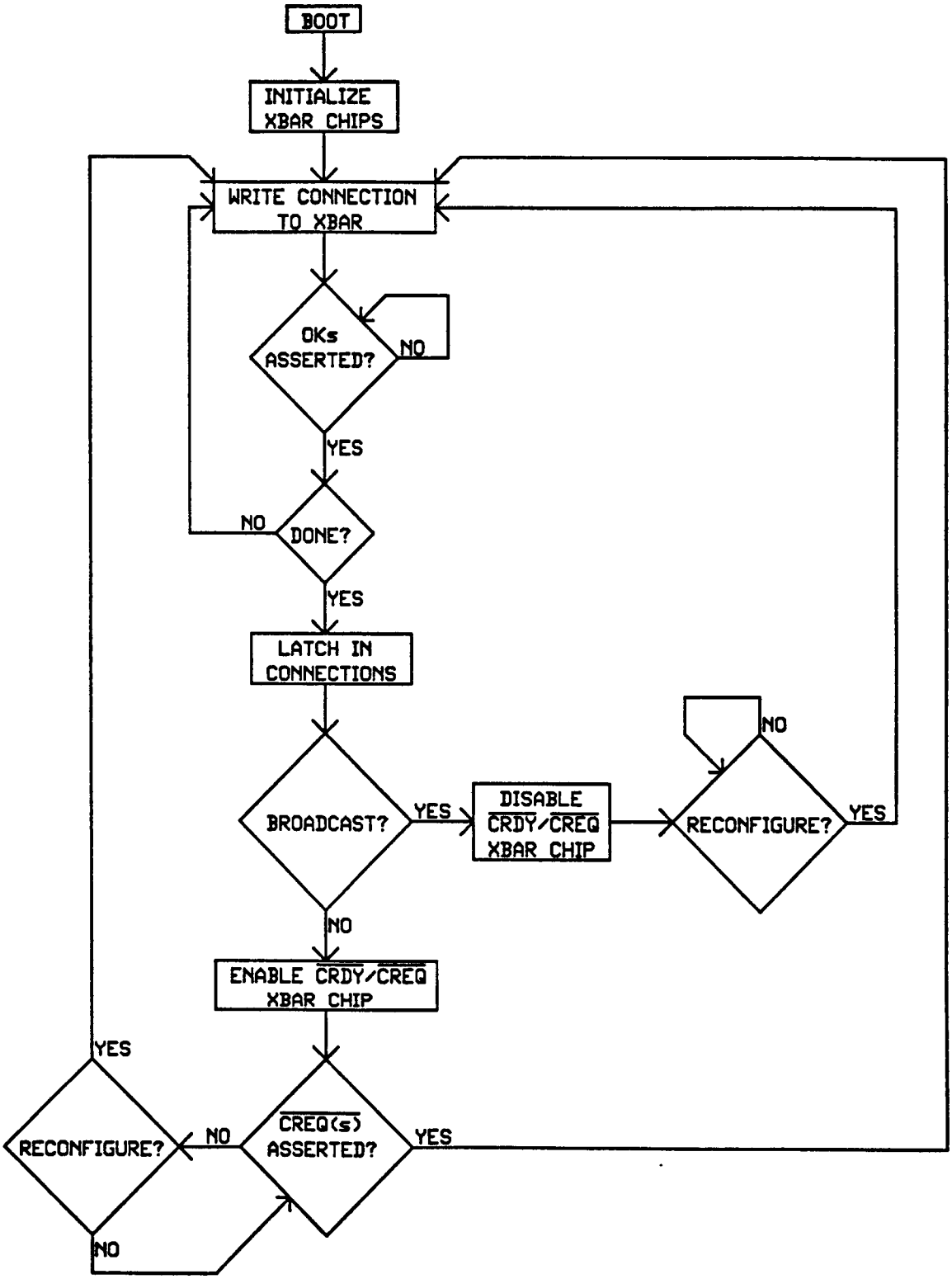


Figure 16 – Flow Diagram for Xbar Control

the C40s being switched through the Xbar, and can use its commports to monitor the other C40s in the network. Second, DSP~Link is a standard interface available on a variety of C40 host boards, and is of a convenient size, both in address lines and data bus width. This means that other platforms with DSP~Link or comparable interfaces can be used to control the Xbar switch. The alternative was to put a C40 on the Xbar switch board, which would have cost much more in size, power, layout, etc., although it would have saved in configuration time and monitoring speed.

B. Decoding for Xbar

The control unit must access eight different parts of the Xbar and provide control signals to these different parts. The controller must write to the crossbar chips, the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler, and the $\overline{\text{CRDY}}$ arbiter. It must also read the OK lines from the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler and the $\overline{\text{CREQ}}$ lines. The controller writes to a certain address to enable the $\overline{\text{CRDY}}/\overline{\text{CREQ}}$ crossbar chip and reads from that address to disable it. There is also an address that the controller reads to latch the connections at the crossbar chips while writing the latch signal to the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler.

Address lines from DSP~Link are also used to create the signals BDCYC1, BNKLDI, CONFR, BRDCST, and ADR0-ADR2. These lines occupy the DSP~Link address lines A0-A6, respectively, and are used by the crossbar chips, the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler, and the $\overline{\text{CRDY}}$ arbiter. Address lines are used because they are always in a determined state and not tri-stated between accesses as the data lines can be ([2] Chapter 9). Lines A7, A8, and R/W decode the 8 necessary Xbar locations as shown in Figure 17(a). Other memory location uses are in Figure 17 (b) and (c).

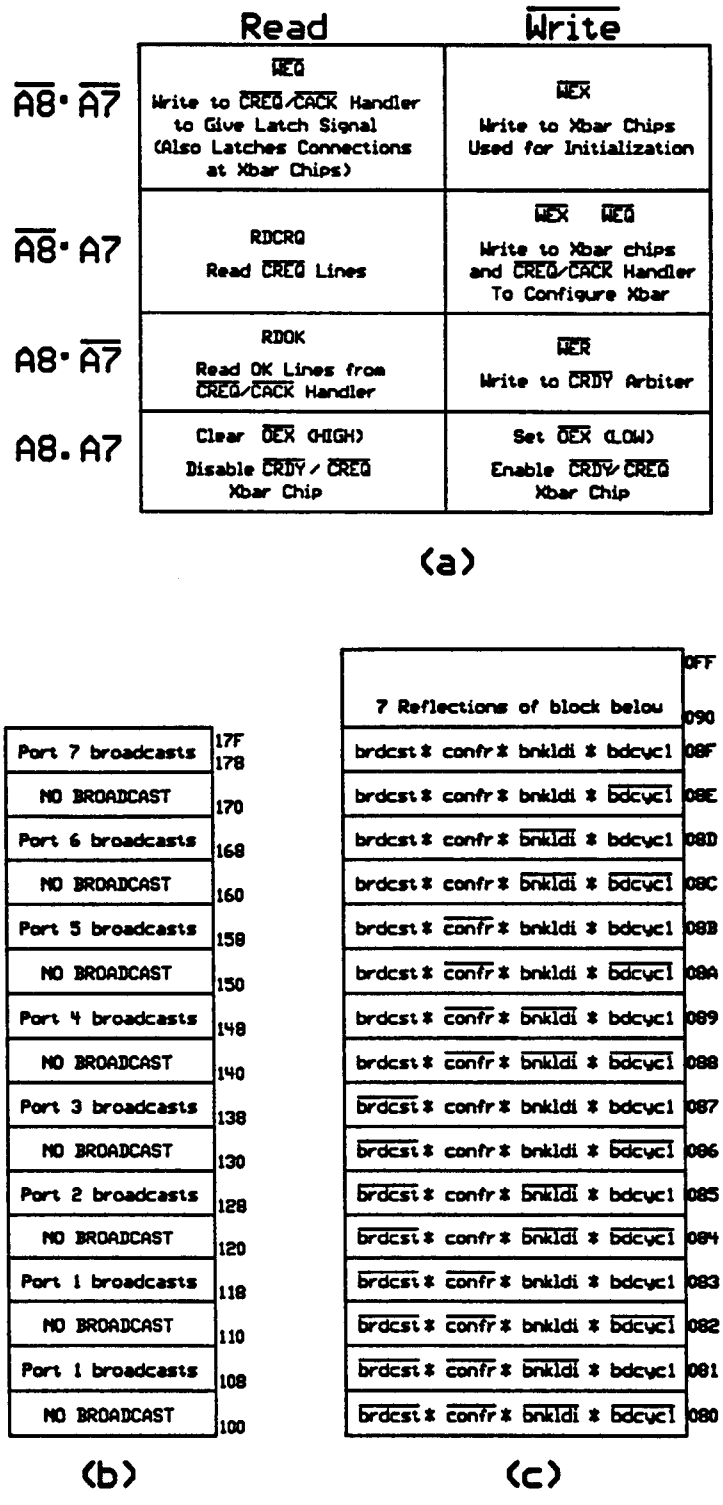


Figure 17 – Xbar Address Decoding (a) Basic Decoding (b) $\overline{\text{CRDY}}$ Access (c) $\overline{\text{CREQ}}/\overline{\text{CACK}}$ and Crossbar Access

By using address lines for control signals, the controller must write to or read from different addresses to accomplish desired tasks. For example, to initialize the Xbar, the controller can write to 000x (addresses are given in hex as offsets from the start of DSP~Link in the C40's memory). To write connection information to the crossbar chips and the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler in non-broadcast mode, the controller writes to 081x for the first cycle, and 080x for the second. At 081x, CONFR and BRDCST are not asserted, while BDCYC1 is high and BNKLDI low. At 080x, CONFR, BRDCST, and BNKLDI remain the same, but BDCYC1 goes low to indicate the second configuration cycle. To latch the connection, the controller reads from 002x, which brings BNKLDI high and strobes it into the $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler. In broadcast mode, the controller can write to 108x to indicate that Port0 is broadcasting. In this case, ADR = 000b, while BRDCST is asserted and the correct location at the decoder is accessed to write to the $\overline{\text{CRDY}}$ arbiter. The decoder was implemented in a standard 15-nsec 22V10 PLD.

C. Data Line Usage

The data lines from DSP~Link are used as shown in Figure 18. During initialization of the crossbar chips, the data lines are used exclusively for the REGADR and CI purposes, as explained in Chapter III. For configuration, they are used to specify the input and output port of a connection in the Xbar. The state of these two ports is also given by the 'O' and 'I' lines. The lower byte of the data bus is used to specify which ports are receiving the broadcast, as explained in Chapter V. The controller can also read the OK lines, which reside in the upper byte of the two-byte word, or the $\overline{\text{CREQ}}$ lines, residing in the lower byte.

D. Possible Timing Problems

DSP~Link timing constrains the strobe signal $\overline{\text{IOE}}$ to be held low for a minimum of 120 nsec with a 10 nsec minimum setup and hold time for the address and R/W lines. The data on a read or write has to be valid for 30 nsec minimum before the $\overline{\text{IOE}}$ line goes high at the end of an I/O access [4]. Because of the slowness of this data strobe (4.17 Mhz as compared with the commports maximum rate of 16 Mhz), it is possible that the configuration will bottleneck the system, with turnarounds being requested before the connections are latched in. To avoid this, the user of the switch will need to ensure that the controlling C40 sends a signal to the other C40s in the network indicating the connections have been made, and the other C40s will need to wait until the signal is received before sending commport data. This will guarantee the commports are not set in motion until after their connections have been made in the Xbar switch.

$\overline{\text{CREG}}/\overline{\text{CACK}}$		I2	I1	I0				I		O2	O1	O0				O
Xbars	RG.6	RG.5	RG.4	RG.3	RG.2	RG.1	RG.0		CI.6	CI.5	CI.4	CI.3	CI.2	CI.1	CI.0	
$\overline{\text{CRDY}}$									U7	U6	U5	U4	U3	U2	U1	U0
Reads	OK7	OK6	OK5	OK4	OK3	OK2	OK1	OK0	$\overline{\text{CR7}}$	$\overline{\text{CR6}}$	$\overline{\text{CR5}}$	$\overline{\text{CR4}}$	$\overline{\text{CR3}}$	$\overline{\text{CR2}}$	$\overline{\text{CR1}}$	$\overline{\text{CR0}}$

RGx = REGADRx
I0-I2 = Input Port in Connection (at $\overline{\text{CREG}}/\overline{\text{CACK}}$ handler)
O0-O2 = Input Port in Connection (at $\overline{\text{CREG}}/\overline{\text{CACK}}$ handler)
Ux = Use Port x - Indicates Port x will receive broadcast
OKx = OK Lines to Read from $\overline{\text{CREG}}/\overline{\text{CACK}}$ chip
$\overline{\text{CRx}}$ = $\overline{\text{CREG}}$x for Reads from $\overline{\text{CREG}}$ Buffer Chip

I = Input Port Status
O = Output Port Status

Figure 18 – Control Words Template

CHAPTER VII

TESTS OF XBAR AND RESULTS

A. Comments on Testing Procedures

The Xbar was tested using a PC-hosted 40 Mhz C40 board from Spectrum Signal Processing as a controller. The commports from this C40 were used in the Xbar so the data could be checked by the controller. The Spectrum-provided TI emulator/debugger software was used to access the C40 from the PC. A separate power supply was used for the Xbar board. To test a piece of code, the debugger was entered and then the power to the Xbar was turned on. A software reset was then given to the C40 in case any glitches gave a 'fake' bus request or data strobe to a commport during power-up of the Xbar board (as was observed at times). The code was then loaded onto the C40 from the debugger and appropriate breakpoints set, especially at the ending ERROR and STOP loops. The program counter of the C40 was then set to the start of the code and the 'run' command was issued. If the C40 broke at the STOP loop, the test was successful, while a break at the ERROR loop indicated problems. All the tests described in this chapter broke at the STOP loop.

B. Test of One-One Connections

The Xbar was first tested to ensure that a single connection could be made and that the data could be sent through the Xbar connection uncorrupted. The following code fragment is an example of what the controlling C40 would use to make the connection and to check it. The code used here is assembly language for the C40 DSP. In this code fragment, C40 commports 0 and 3 are connected.

Commport0 is an output port at power-up, while Commport3 is an input. They maintain these states throughout this test. Note that the controller first initializes the Xbar and enables the $\overline{\text{CRDY}}/\overline{\text{CREQ}}$ crossbar chip.

```
        LDA @CRDYEN,AR0          ; /CRDY Xbar enable address
        STI R0,*AR0              ; enable the /CRDY Xbar with dummy write
        LDA @XBARI,AR0           ; Xbar initialization address
        LDI 8e8eh,R0             ; Xbar initialization control word
        STI R0,*AR0              ; initialize Xbars
        LDA @RDOK,AR6            ; address to read OK's
        LDA @LATCH,AR4           ; latch signal for connections
        LDA @CYCLE1,AR5          ; Xbar address, 1st cycle of configure
        LDI @CONNECT,R3          ; connection command word
        STI R3,*AR5-(1)          ; make the first connection cycle
        STI R3,*AR5++            ; 2nd connection cycle, keep BNKLDI low
testok1:
        AND *AR6,R6,R1           ; read and mask OK lines
        SUBI R6,R1               ; check if OK's are asserted
        BNZ testok1              ; keep checking if not both asserted
        LDI *AR4,R4              ; latch in connection
; Load control registers for commports
        LDA @comm_port0_ctl,AR0  ; Load comm port 0 cont reg addr
        LDA @comm_port0_otp,AR1  ; output FIFO addr
        LDA @comm_port3_ctl,AR2  ; Load comm port 3 control reg. address
        LDA @comm_port3_inp,AR3  ; input FIFO address
        LDA @DUMP,AR4            ; dump for data read from commport
        LDI 7FFFh,R1             ; Output word setup
        LDI 0100h,R0             ; Mask for bit 8 of CPCR - output level
        LDI 1E00h,R2             ; Mask for input level
OUT0:
        TSTB *AR0,R0             ; Check if O/P FIFO is full
        BNZ OUT0                 ; If yes, check again
        STI R1, *AR1             ; write value to O/P, port 0
IN3:
        AND *AR2,R2,R3           ; check input level of port 3
        BZ IN3                   ; keep checking if input FIFO is empty
        LDI *AR3,R3              ; load value from I/P, port 3
        STI R3,*AR4++            ; store in memory
        SUBI 1,R1
        BNZ OUT0                 ; loop until output 32k
```

```
    LDA @DUMP,AR4          ; go back and check the data
    LDI 7FFFh,R1
CHECK:
    SUBI *AR4++,R1,R2      ; check if equal
    BNZ ERROR
    SUBI 1,R1
    BZ STOP
    BR CHECK

STOP   BR   STOP

ERROR  BR   ERROR
```

As shown, 32767 words of 32 bits each were sent through this connection. The program terminated at the STOP loop, showing that no errors were found. This satisfactorily proved that the correct connection was made at the Xbar.

Some concern has been expressed over the possibility of data corruption in commport connections if $\overline{\text{CSTRB}}$ remains asserted low too long at the receiving commport. This is discussed in [6], along with a hardware solution to shorten the $\overline{\text{CSTRB}}$ pulse at the receiver. However, this problem was not observed in the test, neither for the one-to-one connections nor for broadcast connections, even with the maximum possible delay of 50 nsec through the crossbar chips (25 nsec each way) [5]. If in the future this problem does present itself, implementation of the pulse-shortener should be sufficient to solve it.

C. Test of Connection Bus Direction Changing

Next, the Xbar was tested to ensure that the state machines in the $\overline{\text{CREQ}}$/ $\overline{\text{CACK}}$ handler would respond correctly in arbitrating a change in the bus ownership between two connected commports. The original commport connection was made as per the code in Part B, connecting Commport0 as output to Commport3 as

input. The following code fragment demonstrates the change of bus ownership. Commport3 will have data to output and assert its $\overline{\text{CREQ}}$. The controller detects the request and makes the new connection with Commport3 as output and Commport0 as input.

```
    LDI @REVERSE,R7     ; Load the reverse connection command

; make change in direction by outputting to port 3

OUT3:
    TSTB *AR2,R0        ; check if o/p buffer full
    BNZ OUT3            ; if so, check again
    STI R3,*AR3         ; write out to port 3, initiate change
    LDI 08h,R1          ; mask for /CREQ lines
CHKCRQ:
    ANDN *AR5,R1,R5     ; check for /CREQ asserted low
    BZ CHKCRQ           ; keep checking if not asserted
    STI R7,*AR5-(1)     ; make the first connection cycle
    STI R7,*AR5++       ; 2nd connection cycle, keep BNKLDI low
testok2:
    AND *AR6,R6,R1      ; check for OKs asserted
    SUBI R6,R1          ; check if OK's are asserted
    BNZ testok2         ; keep checking if not both asserted
    LDI *AR4,R4         ; latch in connection
```

The connection was then checked for integrity with data now being sent in the opposite direction of that in Part B. The program had similar ending statements, with ERROR and STOP loops. Again, the program branched to the STOP loop, showing that the direction of the connection was successfully switched.

D. Test of Initial Connection Bus Arbitration

The Xbar was also tested to see if a connection could be made between two ports that needed to change their state before the connection could be made. The

routine used is similar to that of Part C, except the data was sent to Commport3 before the connection was made. The connection command word showed 'O' and 'I' as not asserted and requested Commport3 as output be connected to Commport0 as input. The controller then checked for the OKs, made the connection upon receiving the OKs, then checked the data path. Again, data was transferred without corruption.

E. Test of Reconfiguration Ability

The Xbar was then tested for the ability to make an initial set of connections, receive a reconfiguration request, set each port to the desired new state, then make the new set of connections. Two connections were made similar to that in Part B. The data paths were checked and found satisfactory. The controller then requested that two new connections be made with the following commands.

```
LDA @CONFRQ,AR7  ; load reconfigure request address
LDI @RCONN1,R3   ; 1st reconfiguration
LDI @RCONN2,R4   ; 2nd reconfiguration
STI R3,*AR7      ; make the 1st reconfigure request
STI R4,*AR7      ; make the 2nd reconfigure request
```

The controller then checked all 4 OK lines and when they were asserted, made the new connections with the following commands. The connections were immediately latched in since the OKs had already been received.

```
STI R3,*AR5-(1)  ; make the first connection
STI R3,*AR5++
STI R4,*AR5-(1)  ; make the second connection
STI R4,*AR5++
LDI *AR4,R4      ; latch in connection
```

Again, the data paths were tested and found to be uncorrupted.

With these tests, the Xbar was shown to function properly in all aspects of unidirectional mode (one-to-one connections). Initial bus arbitration, direction changing, and reconfiguration capabilities were all demonstrated. The connections were shown to be made properly without corruption in the data paths.

F. Broadcast Capability Test

The capability to broadcast data was then tested in the Xbar. Three connections were made according to the following code. A control word is also sent to the $\overline{\text{CRDY}}$ arbiter and the $\overline{\text{CRDY}}/\overline{\text{CREQ}}$ crossbar chip is disabled.

```
      LDA @BCYCLE1,AR0 ; 1st connection cycle address
      LDA @LATCH,AR1   ; latch of connections address
      LDA @RDOK,AR2    ; address of OK lines
      LDA @CRDYCNF,AR3; address of /CRDY chip, port 2 broadcasting

      LDI @CONN1,R7    ; set up 1st connect address
      STI R7,*AR0-(1)  ; 1st connection cycle write
      STI R7,*AR0++(1) ; 2nd connection cycle write
test1:
      AND R6,*AR2,R7   ; read the OK lines
      SUBI R6,R7       ; check if correct ones are asserted
      BNZ test1        ; keep checking if not OK

      LDI @CONN2,R7    ; set up 2nd connect address
      STI R7,*AR0-(1)
      STI R7,*AR0++(1)
test2:
      AND R6,*AR2,R7
      SUBI R6,R7
      BNZ test2

      LDI @CONN3,R7    ; set up 3rd connect address
      LDI @CRDYCNTL,R1 ; load /CRDY chip control byte
      LDA @CRDYEN,AR4  ; address to disable /CRDY Xbars for broadcast
```

```
; make the 3rd connection
      STI R7,*AR0-(1)
      STI R7,*AR0++(1)
test3:
      AND R6,*AR2,R7
      SUBI R6,R7
      BNZ test3
; latch in all three connections
      LDI *AR1,R0          ; dummy read to latch in connections
; disable /CRDY Xbar
      LDI *AR4,R0          ; perform dummy read,disable /CRDY Xbar
; output control word to /CRDY arbiter
      STI R1,*AR3          ; write control word to /CRDY chip
```

All three data paths were checked for integrity over 16 Megawords. No errors were detected by the controlling C40, thereby proving the Xbar broadcast capability. The potential problem mentioned in Part B did not appear in this test. This is reassuring, especially since one would expect the broadcast data transfer time to be longer than that of a one-to-one connection (since multiple $\overline{\text{CRDY}}$s have to return for the output port to see it in broadcast mode).

The functionality of the Xbar is thus demonstrated, both for the crossbar and crosspoint parts of the switch. The data paths through the switch demonstrated integrity in both modes, while the data handshaking and bus arbitration handling proved to be sufficient.

G. Data Throughput Rate Test

Finally, the Xbar was tested for maximum data throughput rate. Two ports were connection as in Part B, with a cable going from the commports to the connectors on the Xbar board. Each cable is approximately 45 cm in length. This gives a total of 2 x 45 cm = 90 cm of cable.

Spectrum Signal Processing, in [7], discusses their measurements of data throughput for various lengths of cable, using both 40 Mhz and 50 Mhz C40s. They state that the first byte of a transmitted word (4 bytes) is synchronized to the output processor's 'H1' clock, which is an internal clock running at one-half the processor's clock frequency. Because of this synchronization, transfer time for consecutive words is a multiple of the output processor's H1 clock period. The fastest a 40 Mhz C40, like the one used in these tests, can send a word in a consecutive word transfer is five H1 clock cycles at 20 Mhz, or 250 nsec, corresponding to 16 $\frac{\text{Mbytes}}{\text{sec}}$ ([7] p. 3). If more delay through cable or other components is added, the synchronization skips clock cycles. Thus the transfer rates fall on a finite set of values, as shown in Table 1 (see [7] p. 4). This 90 cm length of cable is fairly long and tends to give more of a 'worst-case' data rate. Better data rates should be achieved for shorter cable lengths.

Table 1 – Throughput Rates for 40 Mhz C40 Commports

H1 Cycles (@ 20 Mhz)	Transfer Time, 1 word = 4 bytes	Throughput Rate Mbytes/sec
5	250 nsec	16.0
6	300 nsec	13.33
7	350 nsec	11.43
8	400 nsec	10.0
9	450 nsec	8.89
10	500 nsec	8.0
11	550 nsec	7.27
12	600 nsec	6.67

Spectrum predicts the maximum throughput for 1.0 meter of cable with a 40 Mhz C40 at 11.43 $\frac{\text{Mbytes}}{\text{sec}}$. This corresponds to 7 H1 clock cycles. The delay through

the Xbar should add about 200 nsec per word. This comes from two delays for each byte, one for the $\overline{\text{CSTRB}}$ and one for $\overline{\text{CRDY}}$ to return, each specified at 25 nsec maximum, with 4 bytes per word; or 4 bytes $* 2 \frac{\text{delays}}{\text{byte}} * 25 \frac{\text{nsec}}{\text{delay}} = 200$ nsec. This should add about 4 extra H1 clocks cycles, since each cycle corresponds to 50 nsecs. The data rate through the Xbar should thus be 7.27 $\frac{\text{Mbytes}}{\text{sec}}$, corresponding to 11 H1 clock cycles.

To test the actual throughput rate, the following code was used after making the connection. This routine fills the output commport's FIFO, waits for it to empty half-way, fills it again, then reads half of the input port's FIFO. This is repeated the desired number of times.

```
        LDA @comm_port0_ctl,AR1     ; Load the commport control addresses
        LDA @comm_port0_otp,AR2
        LDA @comm_port3_inp,AR3
        LDHI 0100h,AR4              ; Set up loop counter
        AND3 0EFh,*AR1,R9           ; set bit 4 of commport0 control low
        STI R9,*AR1
        LDHI 01234h,R0              ; Set up word to output
        OR 5678h,R0
        STI R0,*AR2                 ; Output the word 8 times to
        STI R0,*AR2                 ; fill up output buffer of
        STI R0,*AR2                 ; commport0
        STI R0,*AR2
        STI R0,*AR2
        STI R0,*AR2
        STI R0,*AR2
        STI R0,*AR2
WAIT1   LSH -8,*AR1,R1              ; Wait for output buffer to
LOOP    BC WAIT1                    ; empty half way (4 words)
        STI R0,*AR2                 ; Fill output buffer up again
        STI R0,*AR2                 ; by sending 4 more words
        STI R0,*AR2
        STI R0,*AR2
        LDI *AR3,R1                 ; Input 4 words from commport3
        LDI *AR3,R1
```

```
        DBD AR4,LOOP            ; Decrement the counter
        LDI *AR3,R1
        LDI *AR3,R1
        LSH -8,*AR1,R1          ; Check commport0's output buffer again
STOP    BR STOP
```

The data rate was measured through the Xbar by timing how long it took to transfer 256 Mbytes of data. The measured throughput ranged from a low of **6.6** $\frac{\text{Mbytes}}{\text{sec}}$ to a high of **7.2** $\frac{\text{Mbytes}}{\text{sec}}$. This corresponds closely to 12 and 11 H1 clock cycles, respectively. This means the crossbar chips and the Xbar board added about 4 H1 clock cycles to the word transfer time for the best case, or 5 cycles for the worst case. This closely matches the predicted data rate of 7.27 $\frac{\text{Mbytes}}{\text{sec}}$ for the best case, while it appears some extra delay can manifest itself, causing an extra clock cycle to be added for the worst case.

CHAPTER VIII

CONCLUSIONS, EXPANSION, AND FUTURE RESEARCH

A. Conclusions

The Xbar switch design of this thesis can be used in a parallel processing system based on the Texas Instruments TMS320C40 Digital Signal Processor. The switch has demonstrated both unidirectional and broadcast connection capabilities. Data can be sent through the switch without corruption, at least for the length of cable used to test it. Reasonably high data transfer speeds in the range of 6.6 $\frac{\text{Mbytes}}{\text{sec}}$ to 7.2 $\frac{\text{Mbytes}}{\text{sec}}$ through the Xbar switch have been achieved. Expansion of the design ideas to switch sizes of 32 or 64 ports is also possible, with only a slight increase in the hardware complexity of the switch.

The size of the switch was greatly reduced by placing the crossbar chips in parallel and by switching different groups of signals through separate chips. This not only reduced the size, but also increased the switch configuration speed by minimizing the number of cycles needed to make a connection.

The data handshaking and bus arbitration signals satisfactorily provide accurate data transmission, broadcast capability, and proper commport state manipulation in the switch. The data handshaking is correctly switched through the Xbar for unidirectional connections and is fast enough so that the problem of data corruption at the receiving commport did not appear in the tests of the Xbar switch. The correct $\overline{\text{CRDY}}$ signals are combined in the data handshaking arbiter so that broadcasts can occur uncorrupted. The bus arbitration handler satisfactorily creates, passes, and receives the necessary token transfer signals to manipulate the commports into the desired states.

The control and address decoding parts of the Xbar switch were also shown to work correctly. By using certain address lines for control signals, the Xbar switch can be initialized, configured, and monitored by accessing different addresses in the controlling C40's memory. DSP~Link was shown to perform satisfactorily as well, though in the future, if expansion is desired, an on-board C40 is suggested to speed up the control processes. Software routines which perform the correct functions have been developed and explained in this thesis, and form the basis of a in-system Xbar switch control software package.

B. Expansion to Larger Switches

This Xbar switch can easily be adapted for systems containing 16, 32, or 64 commports. It is more efficient to use system sizes that are powers of two, because the crossbar chips switch busses that are powers of two in size. These sizes could be implemented using the ideas presented in this thesis, with only a linear increase in hardware. For systems larger than 64 ports and up to 128 ports, each line would need to be connected in a 4-chip subsystem to be able to access all the other similar lines ([5] p. 195). This means that a 128-port system would require $(4\ \frac{\text{chips}}{\text{line}}) * (12\ \text{lines}) = 48$ crossbar chips, which is impractical.

A 64-port system would use the 64x64 crossbar chips to their fullest potential. In a 64-port system, each line of the commport would go to a different crossbar chip. This means that each line would be switched by itself and have access to all other 63 similar lines of the other commports. The two opposite-flowing lines ($\overline{\text{CREQ}}$ and $\overline{\text{CRDY}}$) would have their own separate chips and could easily be programmed to flow opposite of the other commport lines, just as they do in the 8-port system. This means that only 12 crossbar chips would be needed to implement the system,

one for each commport signal.

The 32-port system would be the simplest expansion. This is because the C40 controlling the system has a 32-bit data bus, and the crossbar chip requirement has not yet begun to expand at an exponential rate. It would require 6 crossbar chips each working with 2-bit-wide busses, according to:

$$\frac{12\ \frac{\text{lines}}{\text{commport}}}{2\ \frac{\text{lines}}{\text{bus}}} * \frac{32\ \text{commports}}{32\ \frac{\text{busses}}{\text{crossbar chip}}} = 6\ \text{crossbar chips}$$

Since each $\overline{\text{CREQ}}/\overline{\text{CACK}}$ handler works independently, it would require four of them to implement the 32-port system. The $\overline{\text{CRDY}}$ unit would need to expand so that each $\overline{\text{CRDY}}$ signal comes from 31 others ORed together. This could be implemented on one EPM5192 PLD chip from Altera. More buffer/driver chips would be needed for the control signals to the crossbar chips and the other PLDs, as well as more commport headers and the necessary pull-up resistors.

The main differences in software for a 32-port system come from outputting a full 32-bit word to the $\overline{\text{CRDY}}$ chip for broadcasts, and needing to check all 32 bits for reads of the $\overline{\text{CREQ}}$ or OK signals . The configuration commands would be the same, except more of the lower bits for REGADR and CI at the crossbars would be used since the busses to be switched are smaller (more ports are present, requiring more bits in the address lines to specify). This capability is already in place. The controlling C40 would need to be on the Xbar switch board to fully use the 32-bit data bus, which would also mean the configuration time for one connection would be faster. Even though there would be more connections to make, the turnaround response would be much faster since the on-board C40 can monitor the $\overline{\text{CREQ}}$ and OK signals more rapidly.

C. Future Work on Xbar

Other Xbar switch architectures and hardware options could be explored to find those which allow for larger switches and cut down the delay through the Xbar switch.

One possibility for a new architecture basis is the new 1024-pin analog interconnect chip from Aptix Corporation [8]. This chip is field-programmable and claims to make true analog connections which can handle voltages from 0 to 5 volts. Right now, this chip comes with its own development system and board onto which the elements to be connected can be mounted. A schematic is entered in software on a host PC, translated into chip-readable code, then downloaded to the Aptix chip, which then makes the connections quickly. The slowdown of this system now is the entering of a new schematic every time new connections are to be made. If a software system were developed so that a list of configuration commands could be downloaded very quickly to the chip, this interconnect element might be able to respond quickly enough to satisfy the Xbar switch demands. The chip certainly has the large advantage of having 1024 interconnecting pins, allowing for a 85-commport switch in a very small space. Also, no handshaking management or pull-up resistors would be needed since the connections are bidirectional.

As suggested in the introduction, another possibility for a 'crossbar'-type system is to use different C40 commport interconnection architectures. TI suggests many possibilities for different architectures in their C40 data book ([2] pp. 13-40 to 42), which can be expanded or improved. Future work might concentrate on finding and proving the optimal architecture that minimizes time between ports, and creating the software to make each desired connection through a minimum-delay route.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Atac, R. et. al., "Crossbar Switch Backplane and Its Application", IEEE Transactions on Nuclear Science, Vol. 36, No. 1, Feb. 1989, pp. 726-730.

[2] "TMS320C4x User's Guide", Texas Instruments Corporation, May 1991.

[3] "Solid-State Crossbar Switch", NASA Technical Briefs, Fall 1983, p. 16.

[4] "DSP˜Link Technical Application Note", Issue 1.32, Spectrum Signal Processing Inc., Burnaby, B.C., Canada, Nov. 1991.

[5] "Digital Signal Processing (DSP) Databook", LSI Logic Corporation, Sep. 1991, pp. 186-199.

[6] Larson, K. and Patterson, J. "Designing with TMS320C40 Comm Ports: Part 1", TMS320 DSP Designer's Notebook, Number 17, Texas Instruments Corporation, February 1993.

[7] "Considerations for C40 Communications Port Data Transfer", Application Note, Spectrum Signal Processing Inc., Burnaby, B.C., Canada.

[8] "FPIC^{TM} AX1024D (FPIC/D) Field Programmable Interconnect Component", Preliminary Data Sheet, v1.0, Aptix Corporation, Schaumburg, IL, August 1992.

VITA

Bryant E. Sorensen was born April 4, 1968, in Bismarck, North Dakota. He attended grade and middle school in Greybull, Wyoming, then moved to Preston, Idaho to finish his last three years of high school at Preston Senior High. After graduating as valedictorian in 1986, he attended Utah State University in Logan, Utah, for one year before taking a two-year leave to serve as a proselytizing missionary for the Church of Jesus Christ of Latter-Day Saints. He finished his B.A. in Mathematics, Computational Option, in May 1992, graduating Summa Cum Laude. He was inducted into the Phi Kappa Phi Honor Society while attending USU. He then moved to Tullahoma, Tennessee, to obtain his Master of Science degree in Electrical and Computer Engineering at The University of Tennessee Space Institute. This was completed in December of 1993.

He is married to the former G. Marie Adamson of Ogden, Utah, and is the father of Kristina Marie Sorensen.