IEEE STANDARD 696 INTERFACE GUIDE

VERSION 2.1

180 Ridge Road
Cimarron, CO 81220

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

While DOS/65 can use controllers other than the IEEE Standard 696, Tarbell controller, use of the IEEE Standard 696 bus is a very low cost approach. In any event the 6502 to IEEE Standard 696 interface is easily established and the peculiar changes required for the floppy interface are easy. The following discussion assumes that the user does not already have a KIMSI or BETSI and thus must build the interface from scratch. Those users with existing interfaces should note the approach used and consider whether modifications are required to your interfaces.

NOTE

This document uses an asterisk as a suffix to denote an inverted signal. For example GATE* means that the signal GATE* is at the logic zero level (0V for TTL) when GATE is true.

SECTION 2 - ADDRESS BUS

The address bus interface is the simplistic of all. If no DMA capability is needed, than the interface shown in Figure 2-1 will be perfectly adequate as long as any IEEE Standard 696 I/O boards used only the low eight address lines as the port address. Luckily, the Tarbell controller falls in that category. The addition of DMA capability will require that the control signals for the buffers be lifted from ground and tied to an appropriate control bus line as shown in Figure 2-1. If you do not foresee a need for DMA then do not even consider it. The extra complexity is not big but in today's world every cent counts and more importantly every wire-wrap (or whatever) pin is just one more chance to make a mistake. All further discussion will assume DMA is not needed.

The octal buffers shown (81LS95) were selected because of the convenience of the 20 pin package. Other buffers such as the many members of 8097/8T97/74367 family could also be used and may provide better bus drive power than the 81LS95's provide. I have, however, experienced no problems in driving my terminated 18 slot Vector Graphic mother board even when loaded with 12 cards (5-RAM (56K), 1-ROM, 1-Video (80 x 24), 1-Programmable Character Generator, 2-Tarbell Controller, 1-DAC, 1-I/O).

The restriction that all IEEE Standard 696 I/O cards use only the lower eight address bits to select the I/O port is a real problem. For those of you not familiar with the 8080, it has a separate 256 port I/O address space in addition to the 64K memory space. As will be seen later, the control signals tell the cards on the bus which address space is being used at any one time. The problem is not that there is a separate address space - that we can handle with the control signal interface. The problem stems from the fact that when operating in the I/O space the 8080 places the port number on both the upper and lower 8 bits of the address bus. The users of the IEEE Standard 696 bus are thus free to pick either the high or low 8 bits (or a combination) of the address bus when decoding the port address on a peripheral card. The fix for this problem for the 6502 user is simple. Since we will establish a "separate" I/O space in our interface (to be discussed in the control section) it is a simple matter to echo the "port number" on the low and high 8 bits of the IEEE Standard 696 address bus whenever the I/O space is addressed. The circuitry for this modification to the address buffering scheme is shown in Figure 2-3. In this figure, as in Figure 2-1, each buffer really represents an 8 bit buffer.

The two signals IO and IO* are generated in the control interface and will be discussed later. They are complementary and serve to drive the A8 through A15 IEEE Standard 696 bus lives with 6502 A0 through A7 for I/O operations and with 6502 A8 through A15 for memory operations.

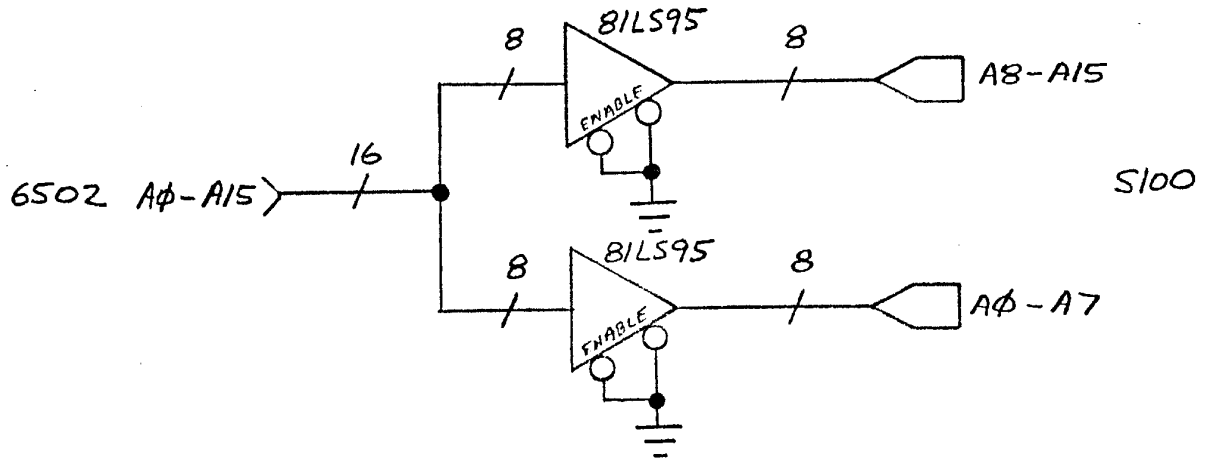

Figure 2-1. Simple 6502 to IEEE Standard 696 Address Bus Interface

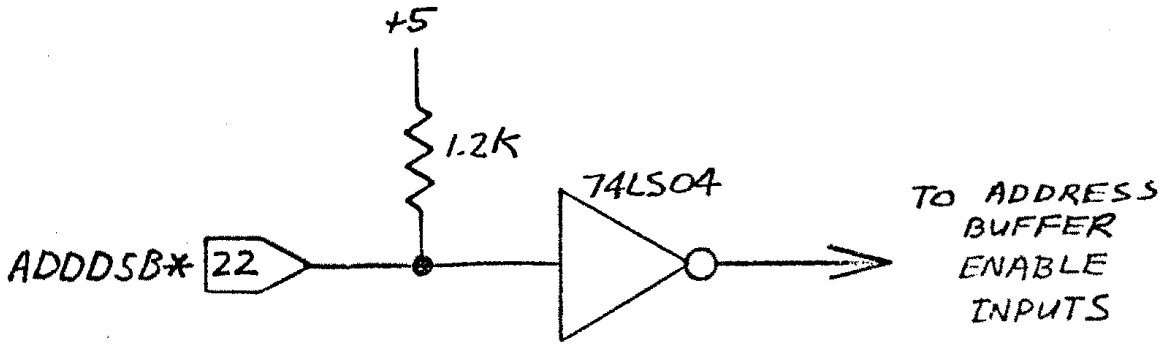

Figure 2-2. Typical Address DMA Modification

One small additional point. The control interface will use the A8 through A15 address signals to decode the I/O space and to control data bus buffer enabling. Those signals are the buffered outputs of the 6502 not the IEEE Standard 696 A8 through A15 bus signals. Thus in Figure 2-3 they are the output of the buffer labeled with an asterisk (*).

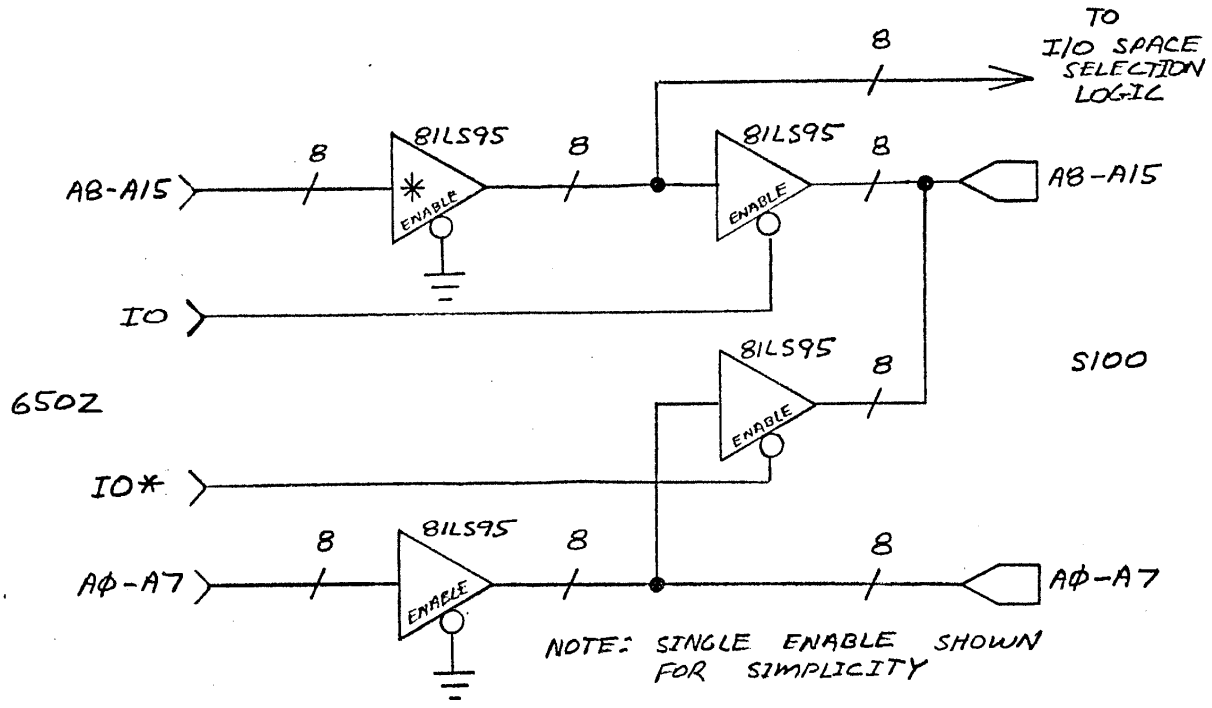

Figure 2-3. 6502 to IEEE Standard 696 Address Bus Interface

SECTION 3 - DATA BUS

The data bus interface is no more complex than the address bus interface but can be a bit tricky. In its simplest form the interface looks like Figure 3-1. In most cases that approach will not work. The reason is simple. Figure 3-1 assumes that any 6502 read operation is a read from the IEEE Standard 696 bus and hence the octal buffer from the unidirectional, IEEE Standard 696 DI bus is enabled. That is true if and only if no device other than the 6502 CPU is connected to the left side of the interface. If the left side of the interface is connected to a more complex system, such as the expansion connector of a SYM-1 or KIM-1, then conflicts will result. The reason for the conflicts is simple. Those systems have either memory or I/O devices mounted on the CPU board which must be addressed and read from. If one attempts to read from, for example, the SUPERMON ROM on a SYM when the circuitry of Figure 3-1 is used, the DI buffer will be enabled and the 6502 data bus thrown into chaos. The solution is to only enable the DI buffer when no conflicts exist. The buffer modification required is shown in Figure 3-2 and the control logic needed is discussed in the following section.

Note that as was the case for the address bus, DMA compatibility has not been provided.

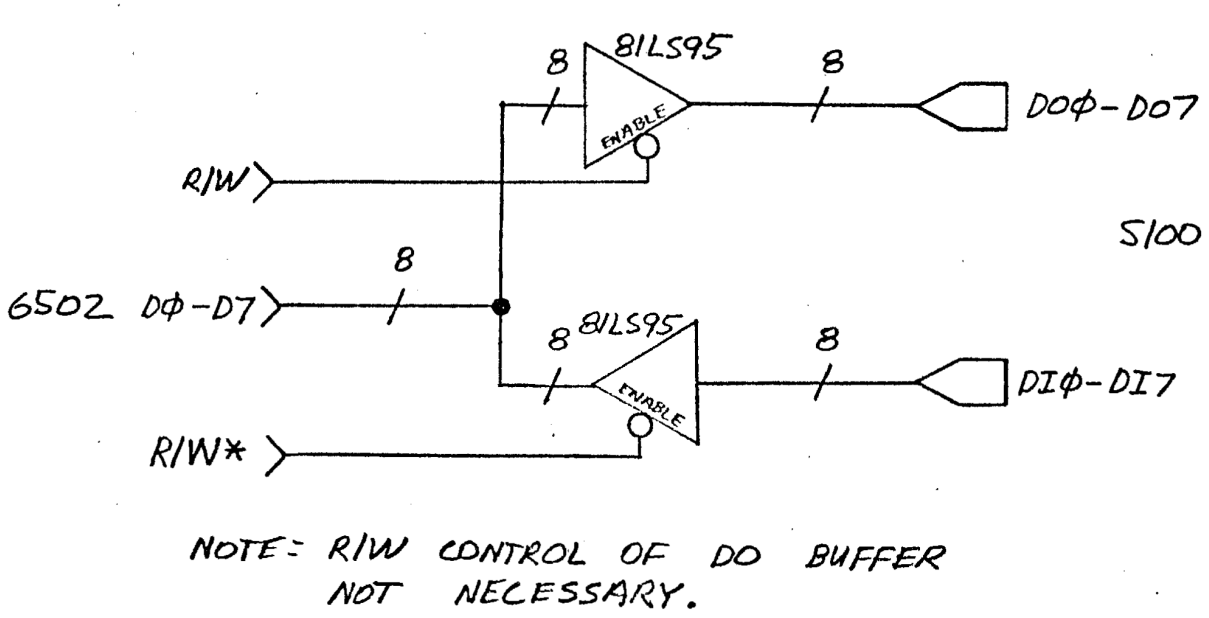

Figure 3-1. Simple 6502 to IEEE Standard 696 Data Bus Interface

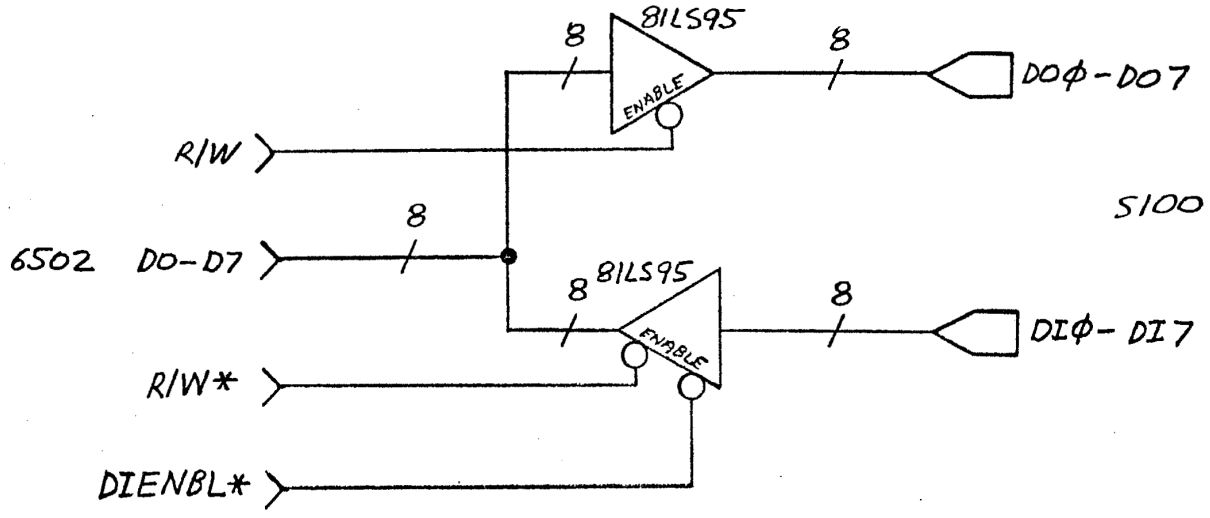

Figure 3-2. 6502 to IEEE Standard 696 Data Bus Interface

SECTION 4 - CONTROL BUS

Now the fun part. First, however, a significant disclaimer is in order. No single 6502 to IEEE Standard 696 control bus interface will enable one to use all IEEE Standard 696 cards without modification.

In some cases no practical modifications will solve the problem either since the offending card may have been designed specifically for the somewhat unique (at least from a 6502/6800 point of view) 8080 timing characteristics. What does work? All I can point to is my experience which says that the following cards worked with no modifications.

- Vector Graphic 8K RAM
- Godbout 16K ECONORAM IV
- Tarbell FDI (but not the bootstrap)
- SSM PB-1 ROM Board and Programmer (Read Only-cannot program)

Minor modifications to the following cards were required to get them working. Interestingly some of those mods were also necessary to get the cards to work reliably with my Ithaca Audio Z-80 CPU board - but that is another story!

Objective Design VDI (80 x 24) video board

Objective Design PCG

While one might consider use of special timing circuits to more closely match the 8080 timing, such an approach is probably excessively complex. If one-shots are used rather than synchronous logic, the circuitry may also be somewhat touchy from a noise or adjustment point of view. Synchronous designs are clean but usually require a higher frequency clock (typically 2 to 16 times the CPU clock) which just is not available on many systems.

Now that the disclaimer is out in the open let's face the problem. The most significant aspect of the problem is that the IEEE Standard 696 bus uses very many control signals - many more in fact than it needs to use. Like the I/O port problem discussed above, the large number of control signals available left the IEEE Standard 696 card designer free to pick and choose. As a consequence, our interface must generate as many as possible. First let me divide the signals into the following categories:

1. Must Have
2. Probably Must Have
3. Probably Do Not Need
4. Do Not Need

As Table 4-1 indicates, the number of control signals which probably are needed is much

more than the usual 6502 1, 2, R/W, SYNC, RDY, IRQ*, and NMI* signals.

The logic used to generate the majority of the control signals is shown in Figure 4-1 and Figure 4-2.

The logic equations which describe those signals are:

IEEE Standard 696	6502
PDBIN	R/W
Φ2	Φ2
PWR*	(Φ2 and R/W*)*
MWRITE	Φ2 and R/W* and IO*
SINP	R/W and IO
SOUT	Φ2 and R/W* and IO
SMEMR	R/W and IO*

The only confusing aspect of those equations should be what the variables IO and IO* mean. As mentioned earlier, these two signals are complementary. IO is true when the IEEE Standard 696 I/O space is to be addressed. It is generated by assigning a fixed page of 256 bytes within the 64K address space of the 6502 as an I/O block. Thus if any address within that block is addressed by the 6502, the IEEE Standard 696 bus is told to connect the I/O address space to the bus for input (SINP) or output (SOUT). In general that page should be located as high as possible or in some otherwise unused corner of the 6502 address space. In my home built system I use E000 through E0FF which leaves 56K of address space below that page for contiguous memory.

The method of generation of IO and IO* is relatively simple. One approach which can be used is shown in Figure 4-3. It has the advantage of only using two IC's and is easily reconfigured to accommodate relocation of the I/O block. The two magnitude decoders detect the page number (in this case $E0) assigned as the IEEE Standard 696 I/O page. Changing the page only requires changing of the comparison value. If a dip plug is used or if wire-wrap connections are used the changes are very easy. Remember that the address inputs are the buffered 6502 outputs, not the signals on the IEEE Standard 696 bus.

The next group of signals to be discussed are the reset or reset-like signals. Most IEEE Standard 696 boards either do not use these or use only one of the three. The Tarbell controller is unique since it uses all three. The circuitry in Figure 4-4 should work if your systems generates a good clean

TABLE 4-1. IEEE Standard 696 CONTROL SIGNALS					
SIGNAL	PIN	MUST HAVE	PROBABLY MUST HAVE	PROBABLY DO NOT NEED	DO NOT NEED
XRDY	3	X or PRDY/72			
VI0	4			X	
VI1	5			X	
VI2	6			X	
VI3	7			X	
VI4	8			X	
VI5	9			X	
VI6	10			X	
VI7	11			X	
STADSB*	18				X (DMA ONLY)
CCDSB”	19				X (DMA ONLY)
UNPROT	20			X	
SS	21				X
ADDDSB*	22				X (DMA ONLY)
DODSB*	23				X (DMA ONLY)
Φ2	24			X	
Φ1	25			X (TIE TO +5V)	
PHLDA	26			X	
PWAIT	27	X			
PINTE	28			X	
SM1	44			X (SYNC)	
SOUT	45	X			
SINP	46	X			
SMEMR	47	X			
SHLTA	48			X	
CLOCK	49			X	
SSWDSB*	53			X	
EXTCLR*	54	X			
SXTRQ*	59				X
SXTN*	61				X
RFSH	66			X	
PHANTOM*	67			X	
MWRITE	68	X			

TABLE 4-1. IEEE Standard 696 CONTROL SIGNALS					
SIGNAL	PIN	MUST HAVE	PROBABLY MUST HAVE	PROBABLY DO NOT NEED	DO NOT NEED
PS*	69			X	
PROT	70			X	
RUN	71			X	
PRDY	72	X or XRDY/3			
PINT*	73			X (INTERRUPT ONLY)	
PHOLD*	74			X	
RESET*	75	X			
PSYNC	76			X (TIE TO +5V)	
PWR*	77	X			
PDBIN	78	X			
SINTA	96			X	
SWO*	97			X	
SSTACK	98				X
POC*	99		X		

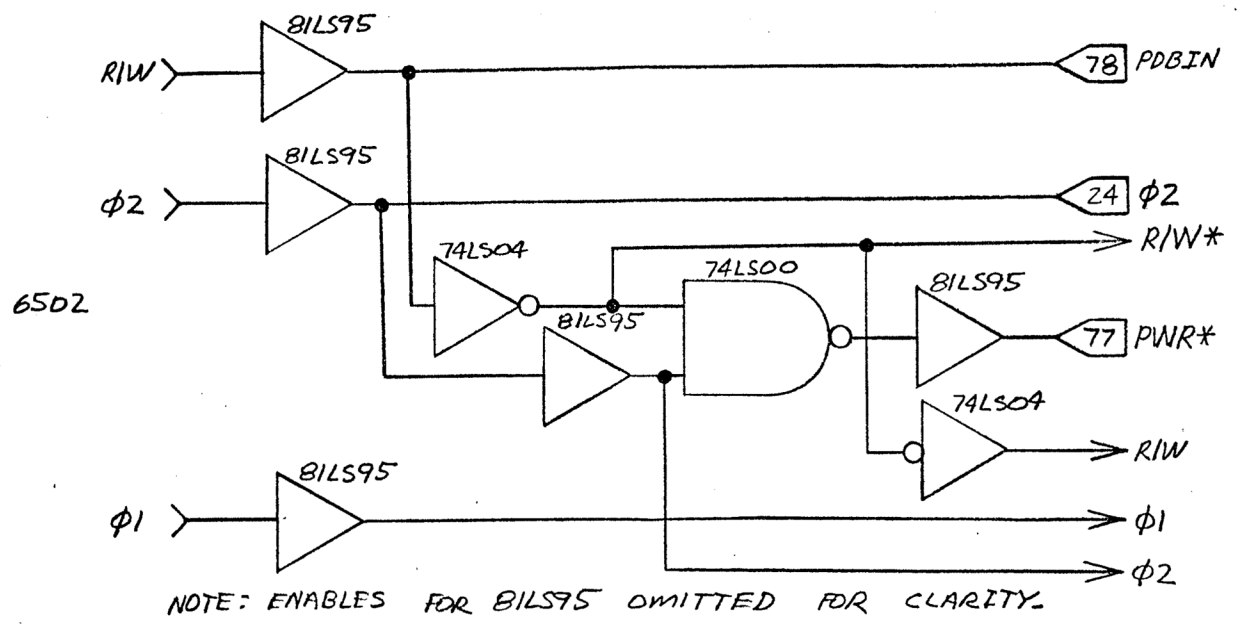

Figure 4-1. Primary Control Signal Outputs

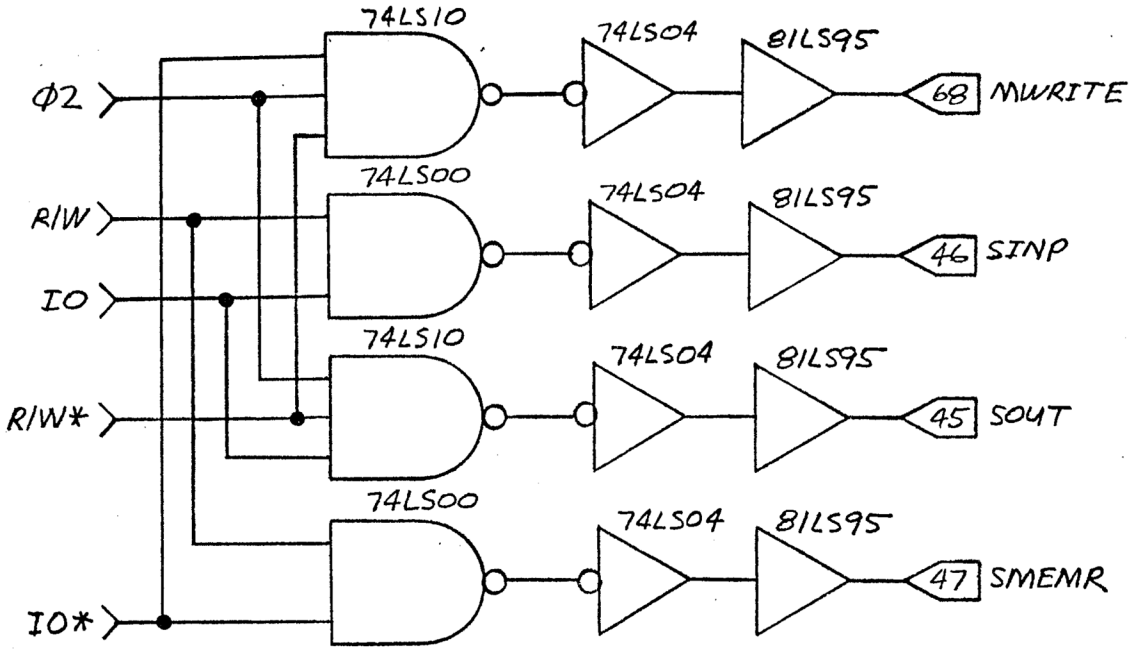

Figure 4-2. Secondary Control Signal Outputs

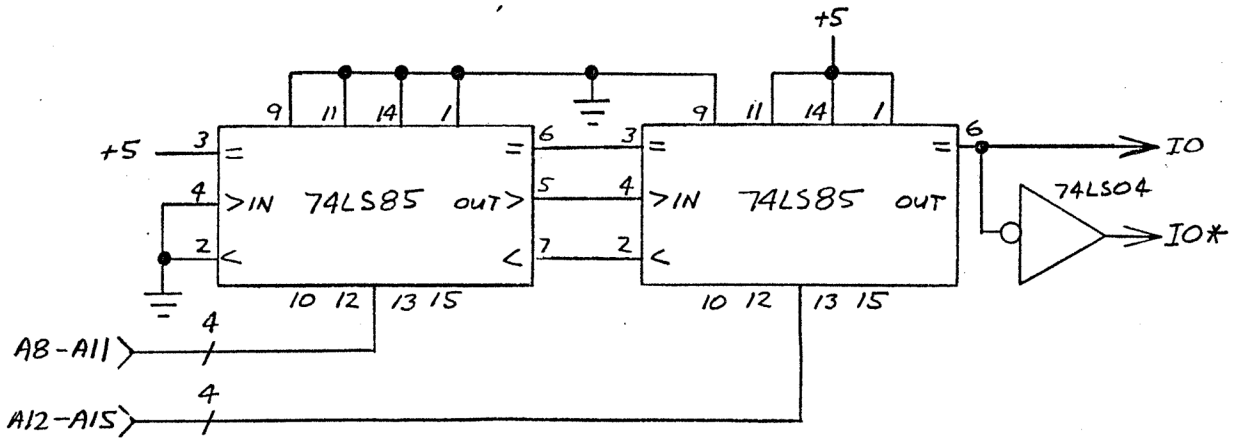

Figure 4-3. IO Decode Logic (For E0xx)

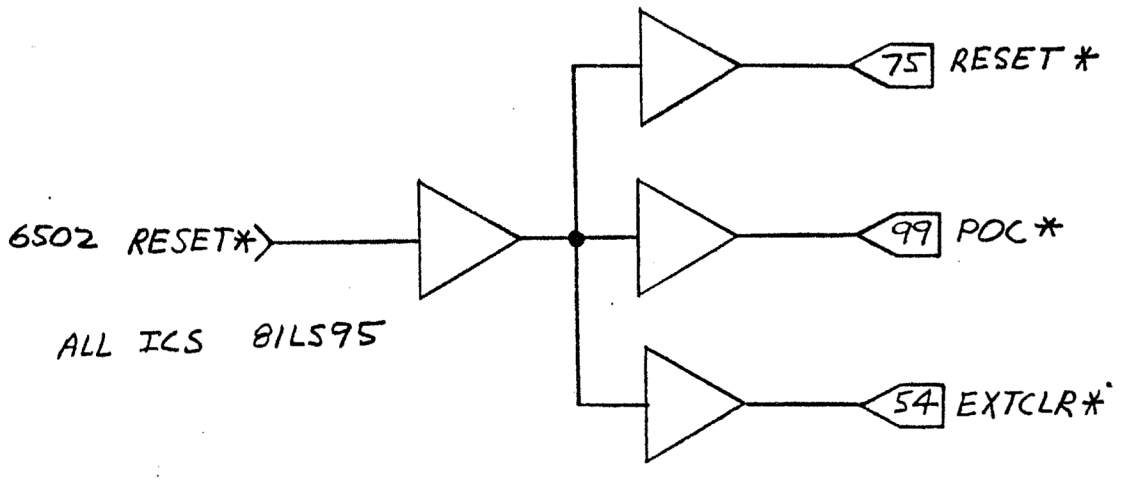

Figure 4-4. IEEE Standard 696 Reset Logic

RESET* signal. If you are wondering why three separate drivers are used, the answer is simple. If each IEEE Standard 696 bus line is terminated, tying three terminated lines to one bus driver probably would not work since the driver couldn't handle the load. If the IEEE Standard 696 bus is not terminated a single driver should work ok! An alternate scheme would be to drive only the RESET* line and tie the others to +5 volts. This works for the Tarbell controller since the three lines are effectively "or'ed" on the controller.

Probably the most important and the most interesting part of the interface is the logic which allows the controller to halt the CPU until the next data transfer can occur without conflict. As it turns out, the Tarbell controller has one fortuitous design characteristic resulting from use of the Western Digital 1771 controller chip - the CPU need only "halt" during read cycles. That is indeed a good thing since the 6502 cannot be halted during a write cycle. Without getting too deep into the 1771 and the Tarbell logic, this condition is true since the critical timing is paced by two outputs of the 1771. These two outputs (DRQ and INTRQ) tell the CPU when the next transfer (read or write) can be accomplished or when an operation has been completed. The CPU tests those signals be executing a read of one specific port. If the controller is not ready when that port is read, the IEEE Standard 696 ready line (either 3-XRDY or 72-PRDY, depending upon a jumper selection on the Tarbell board) is pulled down to a logic zero.

It would seem to be an easy matter then to connect that IEEE Standard 696 bus line to the 6502 RDY line and complete the interface. That is an easy approach but such a direct connection will not work, I know - I tried it that way. The reason stems from the design of the 6502 and is not obvious due to rather poor detail in the 6502 documentation. The bottom line is this - the RDY line into the 6502 cannot change at just any random point in the CPU clock cycle. If it changes at some points in that cycle, the 6502 will crash. Unfortunately, the 6502 spec does not tell us exactly when such a

change can legally take place. The safest approach is to use the trailing edge of the 1 clock or the leading edge of 2 as the gate. The logic of Figure 4-5 accomplishes just that. Other flip-flops could be used with suitable logic changes.

One more aspect of the control interface remains. Recall that Figure 3-2 shows a need for a signal which can enable or disable the DI buffer as a function of what is enabled in the CPU board. Some systems may include the necessary decoding and it is thus merely a question of routing those signals to the interface and then creating the DIENBLE* signal. As an example of how to generate DIENBLE*, Figure 4-6 shows an example for the SYM-1.

The key to generating DIENBLE* is to know what addresses are used by RAM, ROM, or I/O on the CPU board. For the basic SYM-1 the conflicts which must be eliminated are the SUPERMON ROM which resides in the region $8000 to $8FFF, the I/O and System RAM which is scattered throughout the region $A000 to $AFFF, and that portion of System RAM which is echoed in the region $F800 to $FFFF. As the notes in Figure 4-6 indicate, the addition of other ROMs to the SYM-1 will require additional blocks to be disabled. For systems other than the SYM-1 similar logic can be used.

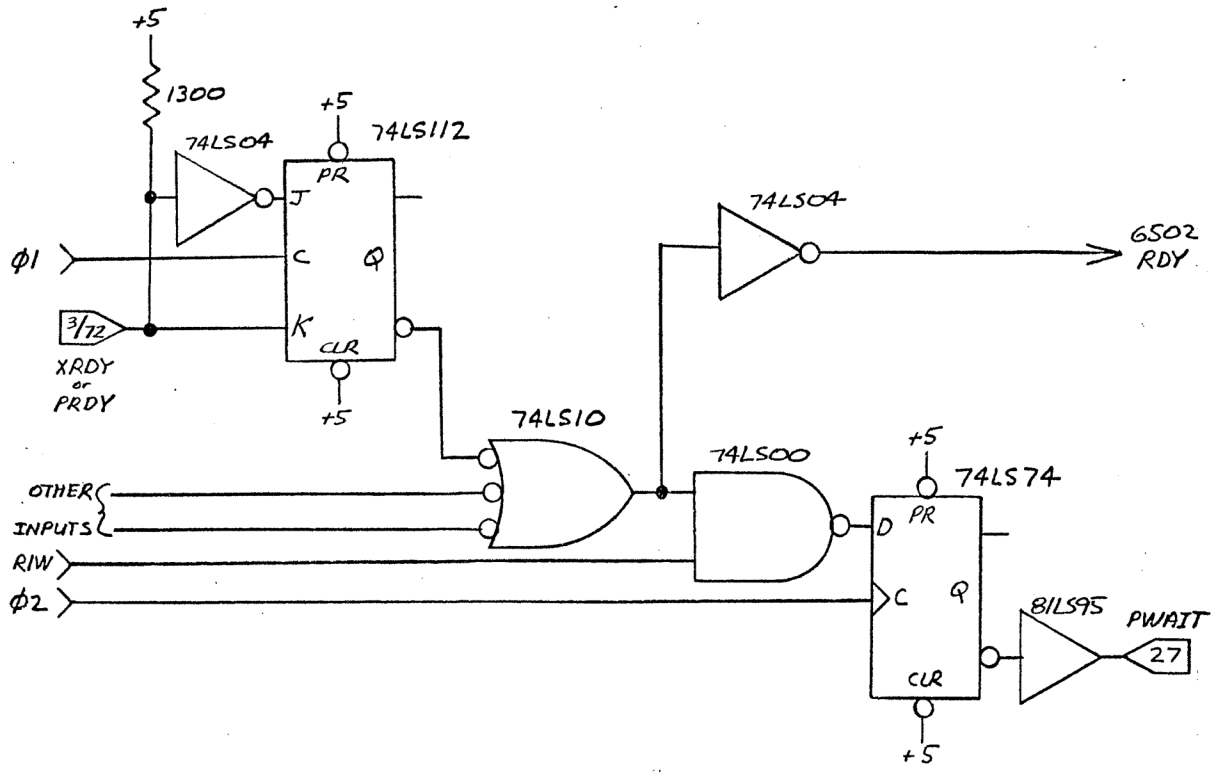

Figure 4-5. 6502 RDY Logic

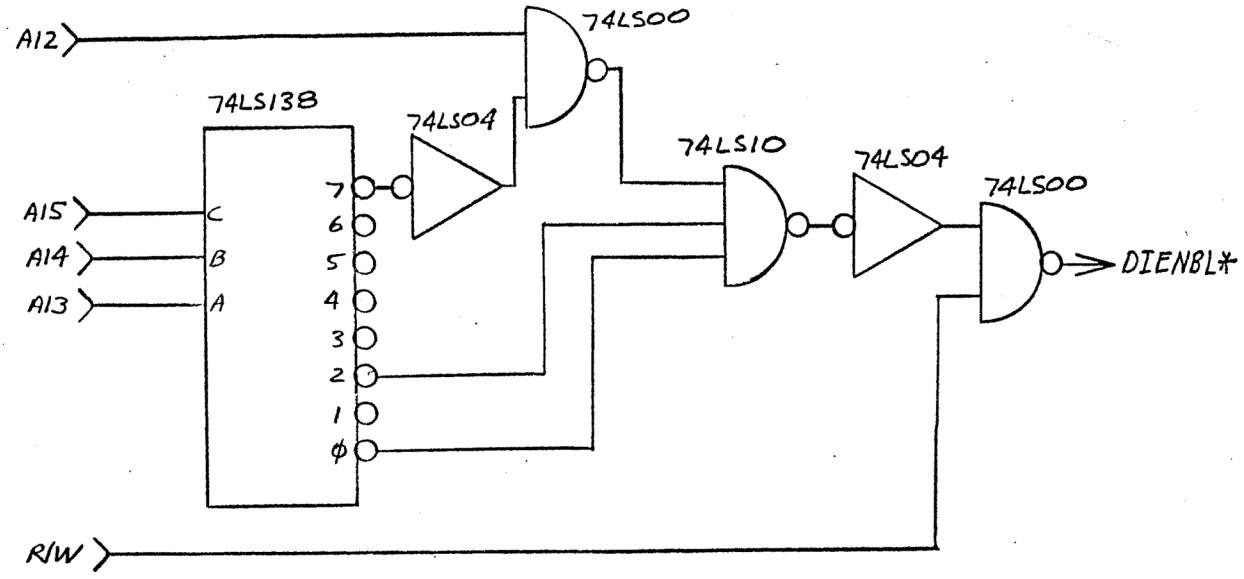

Figure 4-6. Typical SYM-1 DIENBLE* Logic

SECTION 5 - FINAL HARDWARE NOTES

Finally a few words for the purists who may have looked at the 1771 data sheet and concluded that it can not be interfaced to the 6502 without altering the system timing. The 1771 specification says that it needs at least 150ns of data hold time during write cycles after the enable signals return to the false state. The 6502 is only guaranteed to provide 30ns. Based on those numbers it should not work. The signal delays through the various gates and buffers between the 6502 and the 1771 do not resolve the dilemma of why it works, so I am left with the following, possibly incorrect but plausible explanations:

1. Some early 1771's need (on a worst case basis) as much as 150 ns.
2. Later 1771's typically need much less than 150 ns.
3. Typical 6502's provide more than 30 ns hold time. (The 6502 spec says typical is 60 ns.)

Does all that mean I was just lucky and that another combination of 6502's and 1771's would not work? While that is a possibility I strongly suspect that later 1771's (especially the -01 versions) do not need anywhere near 150 ns even in a worst case situation. Moreover I have used three different 6502s (including one very early date code chip) and all three worked perfectly.