3 Barnacles on the ISA Bus

When I first planned these projects, '386SX system boards ran at about 25 MHz. One month later, the slowest boards you could buy started at 33 MHz. Now, a few years later, 166 MHz Pentium boards are sliding toward obsolescence as the long awaited MMX CPUs arrive.

Even the rate of change is increasing!

Curiously, though, higher CPU performance has little to do with your firmware's I/O bandwidth. Just like barnacles on a ship's hull, a myriad of "PC-compatible" boards hold down the maximum speed of the ISA I/O bus. Yes, you can design a faster bus, but customers may not beat a path to your door.

The acronym ISA stands for "Industry Standard Architecture" which, in turn, is a euphemism for "IBM PC/AT Clone". Stripped of the hype, what you're getting is design dating from the late 1970's, tweaked several times over the years, and intended for a simple desktop microcomputer. If you demand blazing I/O performance, these barnacles are not for you.

The PCI (Peripheral Components Interconnect) bus used on all Pentium and Pentium Pro systems eliminates many of the restrictions found on the ISA bus. Previous attempts at improving the ISA bus, such IBM's own Microchannel and the EISA (Enhanced ISA) bus, simply didn't have the right combination of timeliness, broad industry support, performance, and affordability. PCI, today's bus of choice for high-performance I/O, is certainly here to stay. Unfortunately, designing and building PCI bus I/O device requires far more technology, engineering knowledge, and test equipment than is reasonable for anyone other than a full time hardware designer.

But, as we'll see, the good old ISA bus runs entirely fast enough for many, if not most, embedded controller applications. In this chapter, I'll take a look at I/O performance as we begin constructing the Firmware Development Board that serves as the basis for the rest of the projects in this book. As with the firmware in the previous two chapters, I regard covering the fundamentals as more important than presenting a complete design in one shot.

The Intel and AMD CPU data books and Web sites mentioned in the Sources appendix provide many of the grim details required to actually design a system board, starting with a handful of chips. Fortunately for us, that level of detail is irrelevant because the PC Compatibility Barnacles define precisely what the CPU and its support chips must do to be *PC Compatible*. We can see the CPU only

through several layers of glue circuitry that must operate properly to keep all the various boards and programs working.

For our purposes, the ISA bus cycle definitions in Solari's *AT Bus Design* and *ISA and EISA Theory and Operations* books provide enough information to construct a useful ISA bus board. I used those books as the basis for much of the logic and timing information in these projects, although you should refer to the Bibliography appendix for more information on many other related books.

However, it behooves you to verify that these designs will actually work in your system: even when you follow a recipe exactly, you must still taste the soup before dishing it out. I can only hit the high spots here and trust that you will verify the detailed numbers for your own circuits.

I still recall making Oriental Pepper Soup from a recipe that called for 3 T of black pepper. All present agreed it was plenty hot enough, but perhaps three *teaspoons* would have sufficed. Could a "t" have turned into a "T" by mistake?

One Byte at a Time

The simplest ISA bus I/O operation occurs while writing a single byte to an output port. For simplicity, I'll use the target system's parallel printer port at address 0378 to illustrate the process, although it applies to any byte I/O operation. Connect the LED circuit shown in Chapter 1 to watch the parallel port's output in action.

Each iteration of the code snippet in Listing 1 produces two pulses on Bit 0 of the parallel port. With a 33 MHz system clock and a '386SX CPU, the first pulse lasts

Listing 1
This chunk of the porttime.c Micro-C program writes a pair of pulses to the parallel printer port. The first pulse uses C-level functions, while the second pulse shows how to use inline assembler to reduce execution time. You'll find the program and source code in this chapter's subdirectory.

```
while (!chkch()) {
   disable();
   outp(PortAddr,0x01);              /* rising edge                    */
   outp(PortAddr,0x00);              /* falling edge                   */
asm {
   MOV   DX,PortAddr
   MOV   AL,#$01
   OUT   DX,AL
   MOV   AL,#$00
   OUT   DX,AL
}
   enable();
}
```

3 µs and the second is only 1 µs. You should see a dim glow from the LED as the pulses repeat every 31.5 µs with a 13% duty cycle.

Although *x*86 CPU bus interfacing circuitry can be complex (just read the data book, yikes), the ISA I/O bus operates in a fairly straightforward manner. I connected a logic analyzer to the bus and took Photo 1 to illustrate this point.

The top trace is **BCLK**, the 8.33 MHz Bus CLocK. The output operation begins with the rising edge of **BCLK** just before **BALE** (Bus Address Latch Enable), the second trace, goes high. **BALE** falls when the I/O address lines have settled. The third trace, the negative-active I/O Write signal, **-IOW**, clocks the data into the parallel port's output latch.

The port's output data, shown on the bottom trace, actually appears at the pin one **BCLK** cycle after **-IOW** ends. Evidently the integrated I/O chip uses **BCLK** to update its internal latches. As we'll see later, a latch built from discrete TTL chips presents its results immediately after the rising edge of **-IOW**.

Photo 1
This is the logic analyzer record of a single 8-bit write to port 0378. The output operation begins with the rising edge of BCLK just before BALE goes high and ends when -IOW goes high to latch the output data into the port. The integrated I/O chip used in this ISA bus system delays the actual output one additional BCLK cycle.

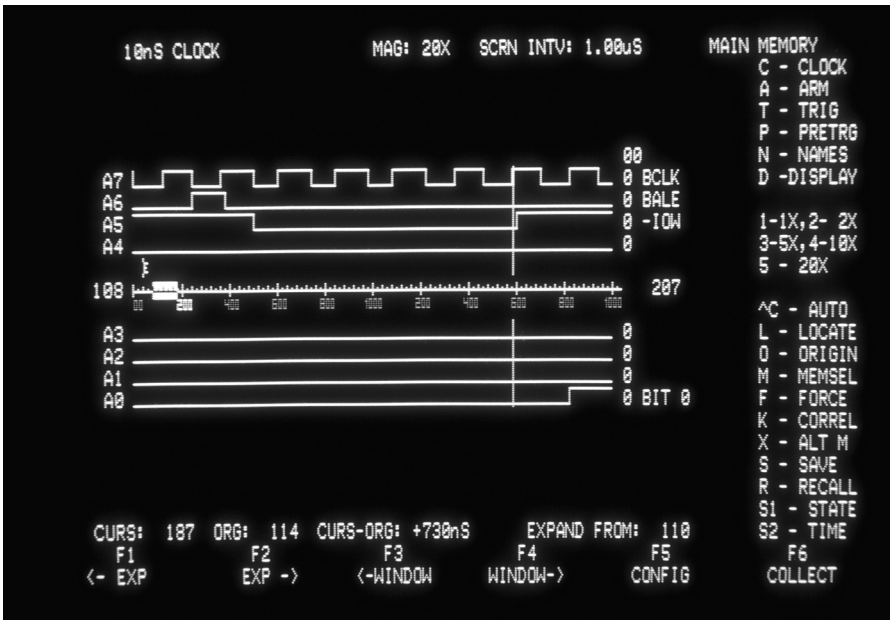
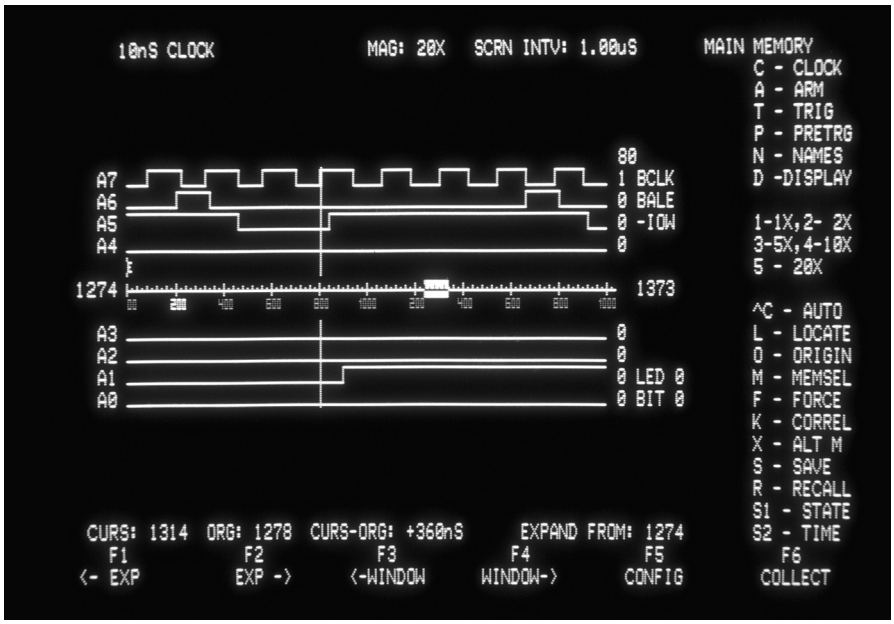


Photo 2
This photo records a single 16-bit write to the LED display on the Firmware Development Board. The operation is complete in three BCLK cycles and the data appears at the latch output immediately after -IOW rises.



Not counting the internal latch delay, the whole output operation takes six 120 ns bus clock cycles. The first two cycles set up the address and data, while the remainder provide enough time for the I/O device to accept the data. Byte reads require a similar amount of time, during which the device provides the data to the CPU in response to the **-IOR** bus signal.

The rather leisurely pace of a single-byte write (or read) may come as a surprise, but the nominal ISA bus cycle requires 720 ns. This ensures compatibility with older ISA bus boards designed for the original 4.77 MHz 8088 PC, while forming a real stick in the spokes for faster CPUs that can gnaw through dozens of instructions in that time.

Unfortunately, we have no way to speed up single-byte I/O cycles. The ISA bus includes a Synchronous Ready signal that can force no-wait memory accesses, but the system board ignores **-SRDY** during I/O operations. You will find some

references referring to **-SRDY** as **-ENDXFR** or **-NOWS**, but, whatever the name, pin B08 simply doesn't help.

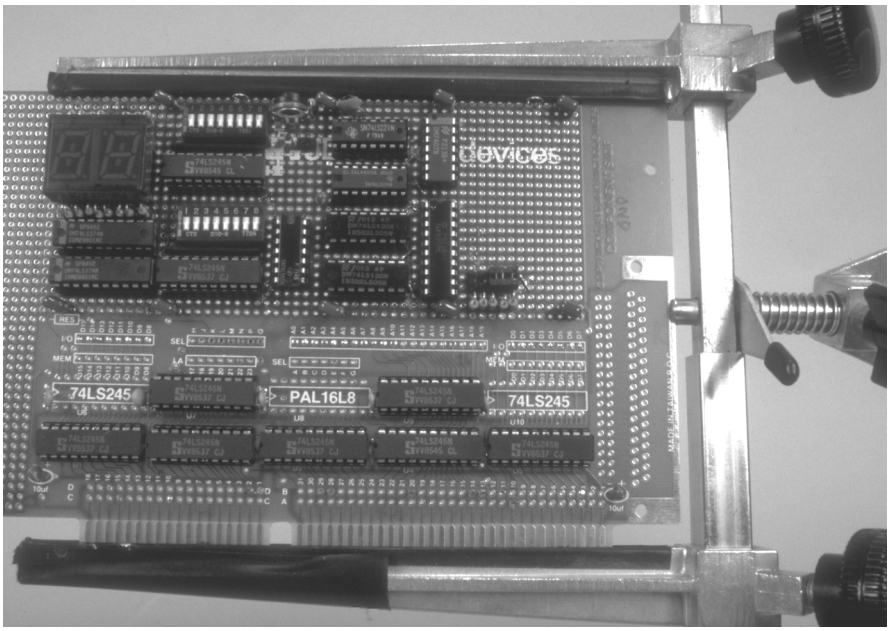
IBM's PC AT designers widened the PC's I/O bus to 16 bits and, because none of the older 8-bit PC boards could respond on the new lines, the new cycles could run considerably faster. Photo 2 shows that writing a 16-bit value takes only three bus cycles from the first rising **BCLK** edge until the data appears in LED bit 0.

That's better, but we must first build some 16-bit hardware…

ISA Board Basics

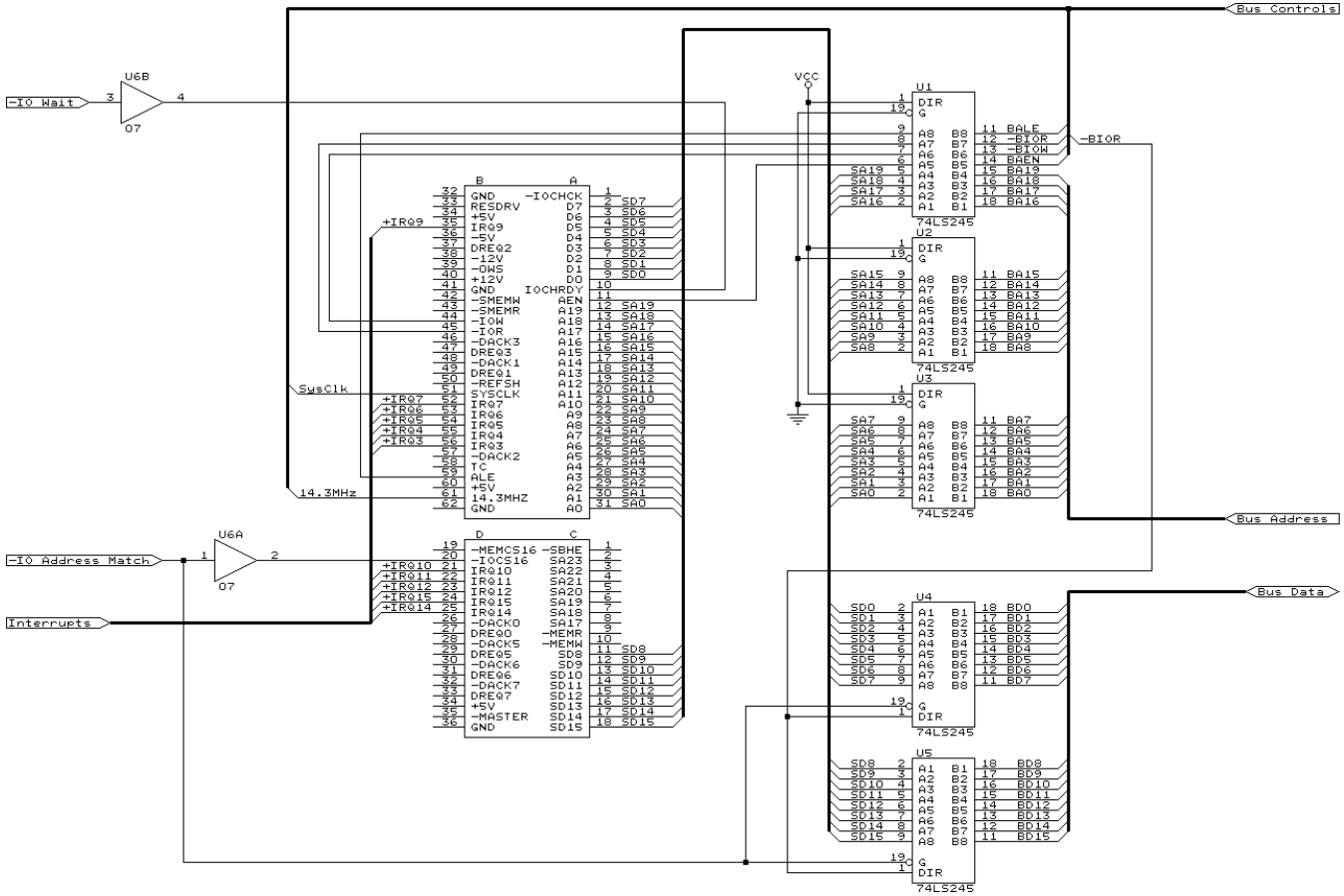
Photo 3 shows the end result: a 16-bit PC/AT prototype board with a handful of ICs, a pair of DIP switches for inputs, and a two-digit LED display for output. Despite using discrete chips, well over half of the board remains empty. We will have plenty of room for the rest of our circuitry, never fear.

Photo 3
I built the Firmware Development Board on a JDR Microdevices prototyping board, with the bus buffers arranged parallel to the connectors in the etched-circuit area and the address decoding and timing ICs near the top edge. The circuits you'll see in the rest of the book depend on the functions provided by these ICs, DIP switches, and LEDs.



Schematic 1

The ISA bus signals can drive a maximum of two (or perhaps, four) LSTTL loads per slot. The LS245 transceivers on the address and control lines are hard-wired to drive signals to the board, while the address decoding circuitry controls the direction of the data line buffers. The system board provides pullup resistors for the IOCHRDY and -IOCS16 lines.



When you examine the Firmware Development Board's schematics, you'll be struck by the absence of "high tech" circuitry. I've deliberately used standard SSI gates and MSI decoders rather than PALs, GALs, PEELs, or other programmable logic devices. While those devices offer significant reductions in board space and wiring complexity, they tend to obscure the mechanics of what is really a fairly simple process.

Regardless of which programmable device I might pick, the majority of you folks won't have access to the programmer required to produce it. But you can implement these garden-variety circuits with whatever programmable logic devices you might have available. There have been enough articles about programmable logic devices in various sources that you should have no trouble getting from here to there.

The FDB (that's "Firmware Development Board") circuitry for this chapter divides neatly into three parts: a set of buffers to isolate the board from the I/O bus, address decoding and timing circuitry, and the LED and switch interfaces. The first two account for most of the logic; the remaining four ICs handle the actual I/O bits.

Buffering the Bus

Schematic 1 shows the FDB's ISA bus buffer circuits. The Original IBM PC AT bus specifications allowed up to two low-power Schottky TTL loads on each board, which the LS245 buffers handle quite nicely. The EISA bus spec allows up to twice that load on each board, but, if you are mixing logic families, you must calculate the actual currents for both high and low logic states. You should also consult the PC/104 signal loading specs, should your embedded project take that bus. In any event, keep your board's bus loading to the absolute minimum.

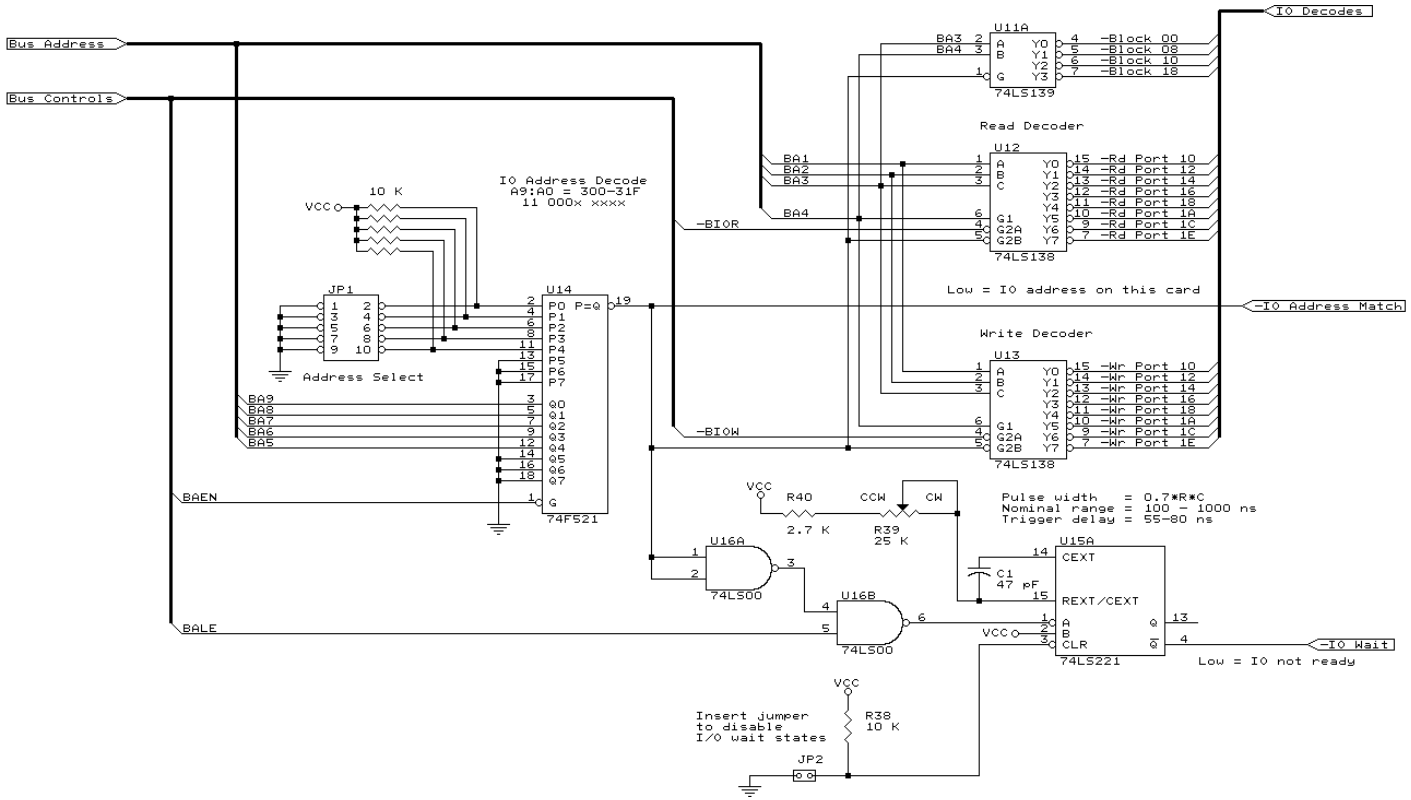
For example, in your own ISA bus designs, you may want to omit the buffers on some lines because you know the exact loading. In fact, you might think you can get away with excessive loading on your custom board in a specialized target system with, say, only two other ISA bus boards… that road is fraught with peril, as "just one more board" may push the bus loading over the limit. Rule of thumb: conservative design eliminates sleepless nights.

Notice that the three LS245 transceivers used for the address and control lines are always enabled and wired to send signals in only one direction. You may use LS541 buffers instead, but I prefer one less part number, particularly when LS245s are faster and less expensive. Check *your* suppliers for the latest prices and availability.

The signals on the inboard side of the LS245 drivers bear a **B** prefix to indicate that they are buffered from the bus. I prefix low-active signals with a minus sign, but as

Schematic 2

This I/O address decoder recognizes I/O port accesses to the FDB and activates the hardware corresponding to that port. Because the FDB contains only 16-bit ports (by definition), the card-selected signal activates the -IOCS16 signal shown in Schematic 1 for all addresses on the board. The firmware must avoid 16-bit accesses to odd addresses, although 8-bit accesses will work under conditions we'll explore in Chapter 4. You may replace the 74F521 with a 74LS688 if you note that the latter chip has a much longer propagation delay and will affect other circuits.



I mentioned in Chatper 0, you'll find that other references use leading or trailing asterisks, front or back slashes, overbars, and even more peculiar notations.

So much for standardization…

The two LS245 buffers on the data lines are enabled only when the address decoding circuitry detects an I/O access to a port on the FDB. When that happens, **-BIOR** determines the signal direction: the buffers drive data to the board, except, of course, when the CPU reads data from the board.

There are only two other signals leaving the board: **-IOCS16** tells the ISA bus interface that this board can handle 16-bit I/O accesses and **IOCHRDY** can stretch those cycles to accommodate slow devices. It may seem strange to talk about slowing the cycles when we're investigating faster I/O, but I'll explain this in Chapter 4.

Both of the control outputs use 7407 open collector drivers, because every ISA bus board can drive those signals. The PC system board includes the rather low value pullup resistors that each ISA I/O board must pull low to activate the signals.

Access Control

Schematic 2 should be more interesting, because the logic actually does something useful. These few ICs determine the I/O addresses of the devices on the FDB and control the bus access timing.

Although an *x*86 CPU can access 65536 (exactly 64×1024) I/O ports, ISA bus I/O boards decode only address bits **A0** through **A9**. As with the 8088's 1 MB of RAM, the 1024 ports defined by those ten bits seemed like a lot at the time.

Over the last decade, various manufacturers have assigned most of those 1024 I/O addresses to one device or another. Before you hardwire your new logic, you should check the references to see who else might be using *your* address. Often, you can overlay a device you're sure you (or your users) will never install. Be very careful when you produce a commercial product, though: those old widgets out there lie in wait for fatal assumptions like that!

The Firmware Development Board's default address range is 0300 through 031F, which corresponds to the ports on the original IBM prototype board. Just in case your target system already has something at those addresses (you aren't trying this in your *host* PC, are you?), a set of jumpers defines the address range. If you change the board's I/O address, you must modify and recompile the sample code, as we operate well below the level of Plug-and-Play hardware.

The CPU and the ISA bus can handle all combinations of 8-bit and 16-bit accesses to even and odd addresses, but each ISA board must include some logic to help out. Because we have complete control over the firmware, I decided to stay with 16-bit accesses at even addresses. It certainly simplifies the logic and, as you'll see in the next few chapters, does not mean the board cannot handle 8-bit devices.

U14, a 74F521 comparator, XORs address bits **BA5** through **BA9** with five jumper inputs and activates the **-IO ADDRESS MATCH** line when they coincide. That signal drives **-IOCS16** to enable the faster 16-bit accesses. That signal marks the only difference between 8-bit and 16-bit bus cycles.

Notice that U14 also activates **-IOCS16** during memory accesses when bits **BA5-BA9** match the jumper settings. That's the way it's *supposed* to work: the CPU's side of the ISA bus interface ignores spurious **-IOCS16** signals during memory accesses.

If you verify the circuit timings, you may be able to replace the 74F521 comparator with a 74LS688 comparator. Because 74LS circuits have significantly more delay than 74F, calculate and measure the changed timings carefully to ensure that the circuits we'll add in later chapters will work properly.

U12 and U13, the two LS138 decoders, combine U14's output with four more address bits to identify each of the sixteen 16-bit ports in the board's address range. The **-BIOR** and **-BIOW** signals enable the decoders, whose outputs indicate both address selection and bus timing.

U11A, half of an LS139 decoder, combines the comparator's output with two more address bits, making each output active for four consecutive 16-bit ports. This chip will come in handy later on when we add ICs with several internal addresses; the chip select signals can come directly from this decoder.

In a real design, you would use a programmable logic device to replace all four of these ICs. In effect, the PAL (or whatever) would implement a custom decoder with only the outputs your circuitry uses. Should you have access to a PAL programmer, feel free to adapt what you see here.

For boards that must handle both 8-bit and 16-bit accesses, consult the references to make sure you cover all the possibilities. That type of logic is hard to do with discrete gates and is tailor made for a PAL, just two of the reasons I avoided it. In truth, though, the design you see here is just another example of a tradeoff between hardware and firmware.

The remaining circuit, U15A, takes a step in the wrong direction. The LS221 one-shot creates wait states to slow down the faster 16-bit I/O cycles. It's not essential for the circuits in this chapter, but I decided to include an analog wait state generator. This way, you can vary the delay continuously and observe the resulting wait states, rather than be stuck with fixed delays from a digital delay line.

Essential I/O

Schematic 3 shows two I/O devices which (just barely) demonstrate that the FDB works. The LEDs provide visible indicators for sixteen output bits and the DIP switches control sixteen input bits. Both ports live at address 031E, presenting the firmware with write-only LEDs and read-only switches.

You may use sixteen discrete LEDs instead of the two-digit, seven-segment display I picked. The firmware in upcoming chapters will display error codes and status indications on these digits, but your application may be different.

Driving LEDs directly from an LS374 latch is, admittedly, unusual. Each latch output bit can sink up to 24 mA at normal logic levels, making the 10 mA for each segment entirely acceptable. The digits have separate common anode connections, both tied to the +5 V supply and current limiting resistors in series with each LED.

Note that LS374 outputs cannot *source* more than a few milliamps each, so they cannot drive common cathode LED digits.

Stepwise Construction

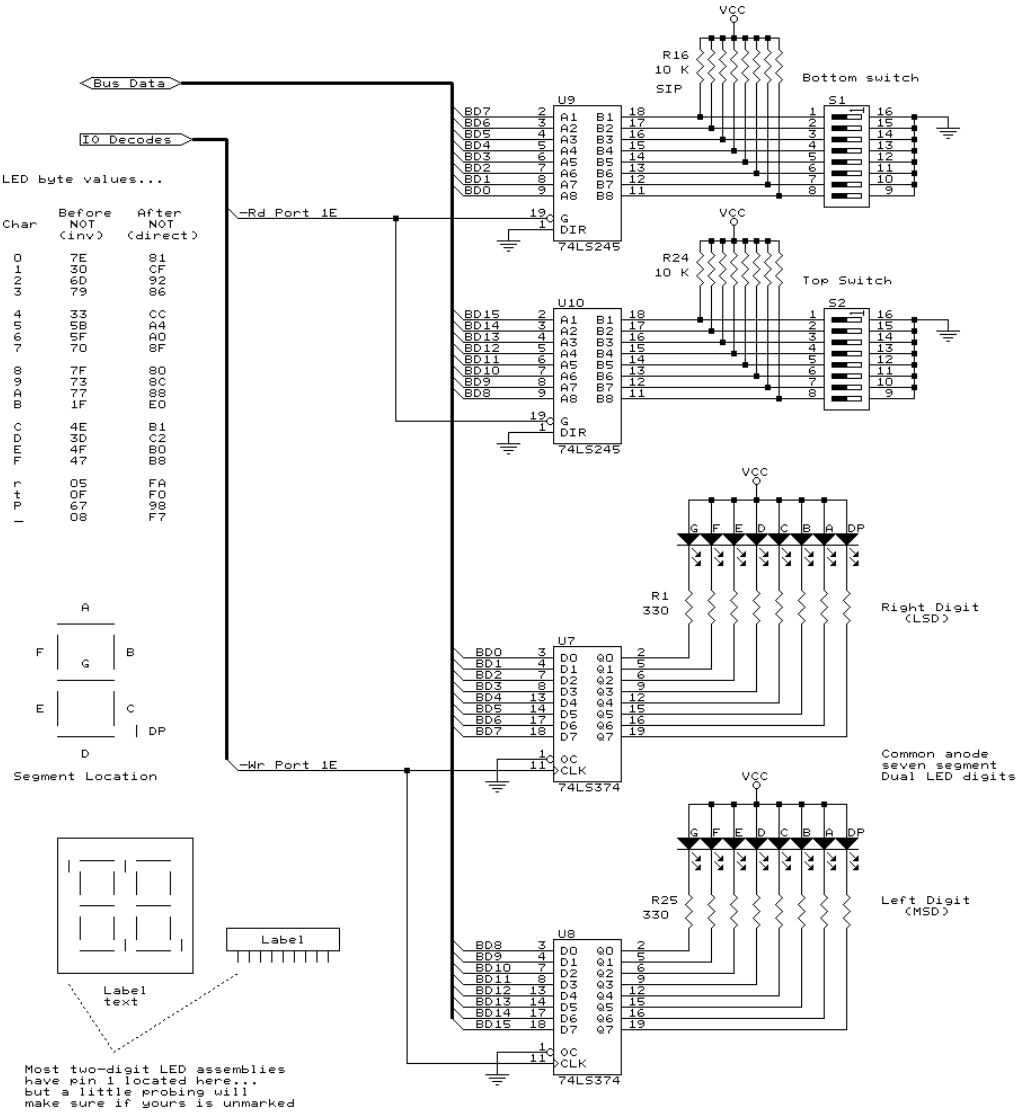
Reading about all this is one thing, but experience remains the best teacher. Fire up those soldering irons!

I used a JDR Microdevices prototype board as the basis for the circuitry shown in this chapter. Although the board etching includes a complete PAL-based bus interface, I decided to present a slightly different and somewhat more general design. As a result, my board has several cuts and adds that match it to the schematics shown here and in the Schematics appendix.

You can also use a bare board, without etched circuitry, similar to those made by Vector, but I *strongly* recommend ground and power planes covered by a good solder mask. Boards with etched buffer circuits reduce the number of wires you must cut, strip, and solder enough to make them worthwhile, even if you don't use their address decoding circuitry.

Schematic 3

The I/O circuitry shown here is the bare minimum required to verify that the FDB's buffers and address decoders are working correctly. The LEDs may be either common anode, seven-segment digits as shown or discrete diodes. The tables show the digital output values that produce some recognizable characters. You can use similar circuitry to build digital inputs and outputs that interface with the real world.



One of the big advantages of designing a plug-in board for a PC is that you can use the computer to debug the thing. I wrote PortTest.C to exercise each section of hardware as I built it. Proceeding step-by-step may not be as challenging as debugging the whole affair at once, but the job gets done faster with less hassle.

The code in Listing 2 should give you an idea of how to proceed. We start by exercising the LED digits with a visible test pattern, so you can tell at a glance whether the hardware works correctly. That's a debugging technique you won't find in the manuals, by the way…

You will, however, need an oscilloscope or logic analyzer to verify some of the other tests. For example, checking the F521 address comparator requires writing to addresses for devices on the board as well as for ones elsewhere in the address space. The only way to see those pulses (or the lack of them) is with an oscilloscope or well-handled logic probe. Of course, if your wiring and chips work properly, then you won't need to debug anything.

In any event, check out the code in this chapter's subdirectory before you begin construction. If you build the hardware in the same order as the tests, you should have no problems.

Listing 2

Writing firmware to exercise new hardware simplifies the debugging process. This section of PortTest.C blinks the LED digits, then lights each segment in turn. Because the firmware drives the LEDs directly from digital outputs, without intervening seven-segment decoders, it can control each segment independently. Simply watching the digits can reveal short-circuits or missing wires without any other test equipment.

```
case '3' :
   putstr("Single bit chasing through LED\n");
   putstr(" segment order is DP, then ABCDEFG\n");
   do {
      outpw(BOARD_BASE+LED_OFFSET,0xFFFF);       /* all segments off      */
      WaitTicks(10);
      outpw(BOARD_BASE+LED_OFFSET,0x0000);       /* all segments on       */
      WaitTicks(10);
      outpw(BOARD_BASE+LED_OFFSET,0xFFFF);       /* all segments off      */
      WaitTicks(10);
      outp(SYNC_ADDR,0x01);                      /* sync on LPT port      */
      for (Counter = 0x8000; Counter; Counter >>= 1) {
         outpw(BOARD_BASE+LED_OFFSET,~Counter);  /* bit low = on          */
         WaitTicks(7);
         Option = chkch();
         if (Option) {
            break;
         }
      }
      outp(SYNC_ADDR,0x00);
      WaitTicks(10);
   } while (!Option);
   break;
```

Delayed I/O

The final section of PortTest.C exercises the LS221 wait state generator I mentioned earlier. Although the LS374 and LS245 ICs can keep up with normal 16-bit I/O rates, they also work with slower accesses. This makes them ideal experimental subjects: nothing can go wrong!

Most of the time you want I/O to run as fast as possible, but, once in a while, you must interface a device that cannot keep up with the normal bus rate. The **IOCHRDY** bus signal holds the key to slowing things down.

If **IOCHRDY** remains high (*I/O Channel Ready*) during an access, the ISA bus interface will complete the cycle normally. However, if it goes low (*I/O Channel **not** Ready*) the interface provides additional time. A pull up resistor on the CPU board holds the line active (*Ready*), allowing designers to entirely omit that circuitry from I/O boards that don't use the signal.

The board must pull **IOCHRDY** low within 76 ns after the leading edge of **-IOW** or **-IOR** during 16-bit accesses. The bus interface samples **IOCHRDY** on each rising **BCLK** edge and extends the **-IOR/IOW** signals by whole 120 ns, **BCLK** cycles.

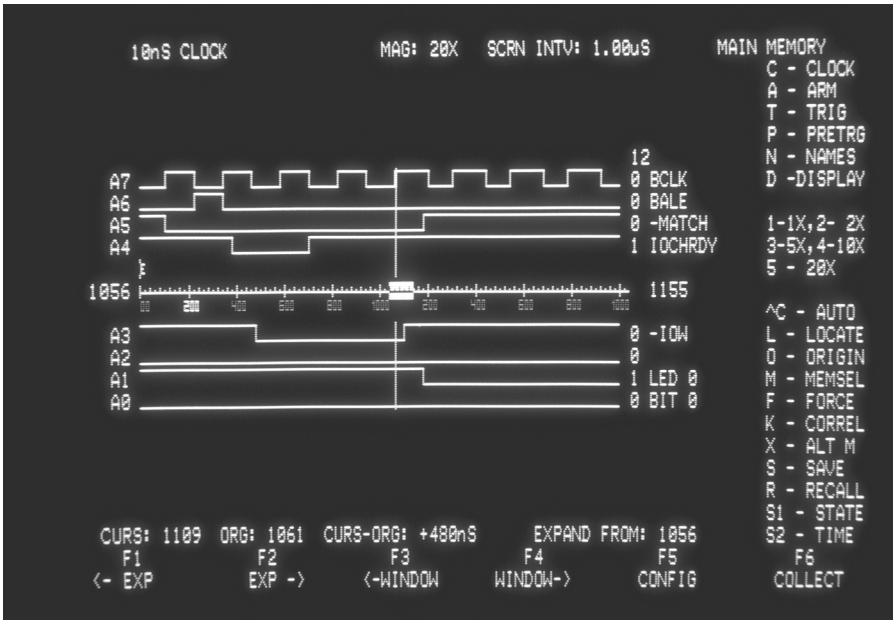
The LS221 has a maximum trigger delay of about 80 ns, far too slow to fire it directly from the **-IOW/IOR** signals. Instead, I opted for a genuine kludge: it's triggered from the falling edge of **BALE**, when the I/O address comparator signals a match. Because the addresses don't stabilize during the early part of the **BALE** pulse, the LS221 may trigger on F521 comparator glitches that occur while the address settles.

It turns out that this doesn't matter for our present purpose, because the only effect is to slow down a bus cycle that should run at full speed. Because there is no data loss and we're examining delays in detail, I decided to overlook the glitches in the interest of producing an easily variable delay for this chapter.

Photo 4 shows a 16-bit write with one wait state. Compare that with Photo 2 to see how **IOCHRDY** affects the results. Of course, you can't see the slowdown by eye, because the additional delay amounts to only 120 ns… those LEDs certainly won't look much different, even to a well-trained eye.

However, the wait state calibration routine in PortTest.C provides a way to adjust the number of wait states without a scope or logic analyzer. The test loop shown in Listing 3 counts the iterations completed in each 54.9 ms BIOS timer tick. As you increase the number of wait states by adjusting R39, the trimpot controlling the LS221 one-shot, the number of iterations decreases.

Photo 4
If the IOCHRDY line is low for at least 76 ns after the -IOW pulse starts, the ISA bus circuitry adds wait states to the 16-bit access. This write cycle takes four complete BCLK cycles due to the added wait state. Compare this cycle with the one in Photo 2.



Listing 3
Adding wait states to I/O accesses reduces the bus speed, allowing slower I/O devices to work properly. This test loop from PortTest.C displays the number of iterations completed in a single 54.9 ms BIOS timer tick. Increasing the number of ISA bus wait states decreases the number of iterations.

```
while (!chkch()) {
   outp(SYNC_ADDR,0x01);              /* sync on printer port              */
   NextTick = WaitTicks(1) + 1;       /* lock on next edge                 */
   Counter = 0;
   do {
      outpw(BOARD_BASE,Counter++);
   } while (peekw(0x0040,0x006C) != NextTick);
   outp(SYNC_ADDR,0x00);
   printf("Count %5u\r",Counter);
}
```

The components I chose for the LS221 limit the **IOCHRDY** pulse between 100 and 1000 ns, which should be enough for most devices. **IOCHRDY** must not be held low more than 15.6 µs during any access to prevent interference with the system's RAM refresh signals. We're safe from that problem.

The number of iterations obviously depends on the CPU speed. A 33 MHz '386SX completes 8080 (no, I'm not making that number up) iterations with no additional wait states. Remember to install the jumper that disables the LS221 before making this measurement, as component tolerances may produce a pulse long enough to trigger a wait state, even with the trimpot fully counterclockwise.

Simple math: 54.9 ms divided by 8080 iterations tells you that each iteration takes about 6.79 µs. I counted 28 instructions in the assembler listing, indicating that the CPU trundles along at 4.1 million instructions per second or about 243 ns per instruction. Each instruction lasts for some multiple of the 30 ns CPU clock, implying about eight clock cycles per instruction.

Removing the jumper and adding one wait state drops the count to 7970 iterations and 6.89 µs per loop. The difference, 100 ns, seems somewhat less than the 120 ns you'd expect.

I think the precise difference works out to 90 ns, three CPU cycles, due to the instruction prefetch queue being empty when the CPU encounters the `OUT` instruction. Because the CPU flushes the queue when it `CALL`s the `outpw()` routine and the instructions that get the address and data from the stack are fairly fast, the CPU actually goes idle during part of the nominally no-wait-state `OUT` instruction. The added wait state partially overlaps the prefetch delay for the next instruction, resulting in less lost time than the book indicates.

Adding a second wait state produces 7830 iterations or 7.01 µs per loop. The difference here, exactly 120 ns, suggests that the prefetcher got itself back on track and had the next instruction ready on time.

I'll leave it to you to continue adding wait states and calculating the results. The bottom line, even in this contrived example, shows that each additional wait state adds less than 2% to the overall loop time. The CPU spends so little time in I/O operations compared to the other calculations in the loop that an additional 120 ns, just four CPU cycles, is no big deal.

Performance measurement on the complex CPUs found in the *x*86 series is not at all trivial, but this simple example shows that if you *really* care about the results, you must make careful measurements. We'll see further examples in upcoming chapters, but for background reading you should get a copy of Abrash's *Zen of Assembly*

Language or his more recent *Zen of Code Optimization* and commit the lessons found therein to heart.

Of course, a digital delay line is the correct way to produce a short delay when you know precisely how long it should be. Simply send **-IOR/-IOW** through the delay line, XNOR the delayed output with the input, and drive **IOCHRDY** with the result. The pulse goes low when **-IOR/-IOW** goes low and returns high when the delayed signal arrives. You must include address decoding and so forth, but the output becomes a fast, low-delay, well-controlled blip that lasts just long enough for your I/O device.

The Digi-Key catalog includes a selection of digital delay lines that run $10-$20 in onesie-twosie quantities, depending on the delay, number of taps, and precision. You can see why I picked a one-shot for this adjustable delay, but I don't want you using an LS221 in your design just because you saw it here! Rummage through your catalogs and see what's available to produce the exact delays you need.

Release Notes

Refer to the code diskette for all the test programs. As before, I used Micro-C and you can run the `HEX` files on your target system with the `MON86` debugger or by copying the `BIN` files on a diskette with the appropriate `BootSect` loader from Chapter 1.

You can also copy the `BusLabel` file on the other side of this page and paste it on your ISA bus extender to simplify your ISA bus signal probing. The raw ASCII equivalent in `BusLabel.asc` will work with your favorite word processor, if you set the printer's line spacing at 0.1 inch to match the ISA bus connectors.

Bus Labels

This label for your bus extender identifies the ISA bus connector pins. The pins are exactly 0.1 inches apart, with 0.4 inches between the two connectors. Beware of copiers that slightly enlarge or reduce their images without warning.

	B - Bottom	A - Top
01	Ground	-IOChCk
02	ResDrv	SD 7
03	+5 V	SD 6
04	IRQ 9	SD 5
05	-5 V	SD 4
06	DRQ 2	SD 3
07	-12 V	SD 2
08	-End Xfer	SD 1
09	+12 V	SD 0
10	Ground	IOChRdy
11	-SMemW	AEN
12	-SMemR	SA 19
13	-IO W	SA 18
14	-IO R	SA 17
15	-D Ack 3	SA 16
16	DRQ 3	SA 15
17	-D Ack 1	SA 14
18	DRQ 1	SA 13
19	-Refresh	SA 12
20	SysClock	SA 11
21	IRQ 7	SA 10
22	IRQ 6	SA 9
23	IRQ 5	SA 8
24	IRQ 4	SA 7
25	IRQ 3	SA 6
26	-D Ack 2	SA 5
27	TC	SA 4
28	BALE	SA 3
29	+5 V	SA 2
30	14.4 MHz	SA 1
31	Ground	SA 0

	D - Bottom	C - Top
01	-MemCS16	-SHBE
02	-IO CS 16	LA 23
03	IRQ 10	LA 22
04	IRQ 11	LA 21
05	IRQ 12	LA 20
06	IRQ 15	LA 19
07	IRQ 14	LA 18
08	-D Ack 0	LA 17
09	DRQ 0	-MemR
10	-D Ack 5	-MemW
11	DRQ 5	SD 8
12	-D Ack 6	SD 9
13	DRQ 6	SD 10
14	-D Ack 7	SD 11
15	DRQ 7	SD 12
16	+5 V	SD 13
17	-Master	SD 14
18	Ground	SD 15