TTL Implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization

H. G. Dietz, T. Muhammad, and T. Mattox

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

December 2, 1994

Supported by ONR Grant No. N0001-91-J-4013 and NSF Grant No. CDA-9015696
Donations from IBM, Texas Instruments, AMD, and Intel
Equipment loans from Digital Equipment Corporation and IBM

Abstract

PCs and workstations offer execellent performance per unit cost. Thus, for computational tasks where a single machine is not sufficient, the possibility of treating multiple machines as a parallel system seems very attractive. However, conventional communication hardware mechanisms yield communication and synchronization latencies that make only the largest-grain parallelism efficient. For example, typical LANs accessed through UNIX have latencies measured in milliseconds; even the fastest parallel supercomputers have latencies measured in tens to hundreds of microseconds. Thus, for any of these systems, the minimum effective parallel grain size is measured in thousands of floating point operations.

Rather than accepting these delays, we have developed new hardware that very efficiently implements barrier synchronization and aggregate communication operations. This new hardware is called PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization. Although the various different versions of PAPERS differ somewhat in the operations they support, each PAPERS unit is capable of performing any basic synchronization or aggregate communication operation with a total UNIX-process to UNIX-process latency of just a few microseconds.

This paper details the public domain design of TTL_PAPERS, which Purdue University and the authors provide strictly on an "as-is" basis, without warranty or liability. TTL_PAPERS differs from the other PAPERS units in a variety of ways, however, all these differences focus on making TTL_PAPERS implementable with a minimum of hardware development resources. The 4-processor TTL_PAPERS design discussed in this paper uses just 8 TTL chips, can be wired and powered using parts readily available at Radio Shack or other electronic supply stores, and fits neatly into a box that is easily constructed using standard sized lumber.

1. Theory of Operation

When one first hears of TTL_PAPERS, it is very difficult to accept that such cheap and simple hardware can implement very effective synchronization and communication operations for parallel processing. This is because the traditional views of parallel and distributed processing rest on a set of basic assumptions that are incompatible with the concept:

- Conventional wisdom suggests that the operating system should manage synchronization and communication, but even a simple context switch to an interrupt handler takes more time than TTL_PAPERS takes to complete a typical synchronization or communication. All interactions with the TTL_PAPERS hardware are I/O port accesses made directly from the user program; there are no OS modifications required and no OS call overhead is incurred.
- Communication operations are characterized primarily by latency (total time to transmit one object) and bandwidth (the maximum number of objects transmitted per unit time). The hardware and software complexity of most interaction methods results in high latency; high latency makes high bandwidth and large parallel grain size necessary. In contrast, TTL_PAPERS is very simple and yields a correspondingly low latency. Providing low latency allows TTL_PAPERS to work well with relatively fine-grain parallelism, but it also means that relatively low bandwidth can suffice.
- A typical parallel computer is constructed by giving each processor a method of independently performing synchronization and communication operations with other processors; in contrast, TTL_PAPERS interactions between processors are performed as aggregate operations based on the global state of the parallel computation, much as in a SIMD machine. This SIMD-like model for interactions results in much simpler hardware and a substantial reduction in software overhead for most parallel programs (as was observed in the PASM prototype [SiN87]). For example, message headers and dynamically-allocated message buffers are not needed for typical TTL_PAPERS communications. It is also remarkably cheap for the TTL_PAPERS hardware to test global state conditions such as `any` or `all`.

Thus, TTL_PAPERS does not perform any magic; it simply is based on a parallel computation model that naturally yields simpler hardware and lower latency. TTL_PAPERS is not really a network at all, but a special-purpose parallel machine designed specifically to provide low-latency synchronization and communication, taking full advantage of the “loosely synchronous” execution models associated with fine-grain to moderate-grain parallelism.

1.1. Synchronization

The only synchronization mechanism implemented by TTL_PAPERS is a special type of fine-grain barrier synchronization that facilitates compile-time scheduling of parallel operations [DiO92] [DiC94].

Hardware barrier synchronization was first proposed in a paper by Harry Jordon [Jor78], and has since become a popular mechanism for coordination of MIMD[1] parallel processes. A barrier synchronization is accomplished by processors executing a `wait` operation that does not terminate until sometime after all PEs have signaled that they are `wait`ing. However, while building the 16 processor PASM (PArtitionable Simd Mimd) prototype in 1987 [SiN87], we realized that the hardware enabling a collection of conventional processors to execute both MIMD and and instruction-level SIMD[2] programs was actually an extended type of barrier synchronization mechanism. Generalizing this barrier synchronization mechanism resulted in several new classes of barrier synchronization architectures, as reported in [OKD90] [OKD90a]. Unlike other PAPERS implementations, the TTL_PAPERS barrier mechanism provides only a subset of these features; however, these features are sufficient to enable conventional processors to efficiently implement fine-grain MIMD, SIMD, and VLIW[3] execution [CoD94a].

The TTL_PAPERS barrier mechanism also provides an effective target for compile-time instruction-level code scheduling [DiO92].

1.2. Communication

In addition to high-performance barrier synchronization, TTL_PAPERS provides low-latency communication. This mechanism is not equivalent to a shared memory nor a conventional message-passing system, but has several advantages for loosely synchronous communication. Basically, it trades asynchrony for lower latency and more powerful static routing abilities.

As a side-effect of a barrier synchronization, each PE can place information into the TTL_PAPERS unit and get information from whichever processor, or group of processors, is selected. Thus, the sender only outputs data; it is up to the receiver to know where to look for the data that the sender has made available to it. In TTL_PAPERS, we further simplify the hardware by extending this concept so that all processors must agree on and contribute to the choice of what data each processor will have available. Compared to conventional networks, this method allows less autonomy for each processor, but yields simpler hardware, better performance, and more powerful access to global state.

The most basic PAPERS and TTL_PAPERS communication operation is a multi-broadcast that sends to each processor a bit mask containing one bit from each processor. This is a very powerful mechanism for examining the global state of the parallel machine. An obvious application is for processors to “vote” on what they would like to do next. For example, it can be used to determine which processors are ready for the next phase of a computation or which processors would like to access a shared resource (e.g., to implement a balanced, conflict-free, schedule for

[1] MIMD refers to Multiple Instruction stream, Multiple Data stream; i.e., each processor independently executes it own program, synchronizing and communicating with other processors whenever the parallel algorithm requires.

[2] SIMD refers to Single Instruction stream, Multiple Data stream; i.e., a single processor with multiple function units organized so that the same operation can be performed simultaneously on multiple data values.

[3] VLIW refers to Very Long Instruction Word; i.e., a generalization of SIMD that allows each function unit to perform a potentially different operation on its own data.

access to an Ethernet). However, because any communication pattern is a subset of multi-broadcast, it can also be used to implement general communication “routing”.

In addition to one-bit multi-broadcast, some versions of PAPERS provide the ability to get four bits of data from any processor in a single operation. Using a unidirectional Centronics printer port, four bits is the theoretical maximum number of data bits that can be obtained by one port input operation. TTL_PAPERS implements four bit sends, and even more functionality, using simple NAND logic to combine signals from the processors. Operations directly supported include:

- Since TTL_PAPERS data is literally a four-bit wide NAND of data sent across all processors, computing a four-bit global NAND takes only one operation. Likewise, NAND can implement AND and OR functions by simply complementing the output or by complementing the inputs.
- To accomplish a four-bit broadcast from a single processor, all the other processors simply output 0xf — four 1 bits. Again, the result will be the complement of the data sent.
- To accomplish a one-bit multi-broadcast, each processor sends four bits of data such that processor *i*’s bit *i* is the data it wishes to send, and the other bits are all 1 values. For example, bit 2 will be 1 for processors 0, 1, and 3; thus, NANDing these signals together will result in a signal that is the complement of bit 2 from processor 2.

In summary, TTL_PAPERS provides almost no facility for autonomous communications, but does provide a very rich collection of aggregate communication operations based on global state.

1.3. Interrupts

To facilitate some level of asynchronous operation, some versions of PAPERS provide a separate interrupt broadcast facility so that any processor can signal the others. Such an interrupt does not really generate a hardware interrupt on each processor, rather, it sets a flag that each processor can read at an appropriate time. Although such a check can be made at any time, the two most obvious times are:

- When a barrier `wait` has taken an unexpectedly long time. This would indicate that one or more processors might have generated an interrupt, freezing the barrier state so that other processors would know to check for an interrupt.
- When the OS is invoked to perform a scheduling operation. Thus, gang scheduling and other OS functions can be implemented by having the OS on each processor use the interrupt facility to coordinate across processors.

At this time (November, 1994), the TTL_PAPERS interrupt facility is not used by the software. However, the hardware does provide all the necessary logic for generating an interrupt and providing a special “interrupt acknowledge barrier.”

1.4. Purpose

Unlike most research prototype supercomputers, TTL_PAPERS is a fully public domain hardware and software design intended to be widely replicated. It is hoped that the fine-grain capabilities of TTL_PAPERS in linking conventional computers will bring a qualitative change to the fields of cluster, network, and heterogeneous supercomputing.

The first public demonstration of TTL_PAPERS was at the International Conference on Parallel Processing, August 16-18, 1994. That unit differed from the one discussed here in that it lacked a parallel interrupt facility, but we were able to demonstrate the complete library using a cluster of four IBM 486DX/33 ValuePoint PCs under Linux. The second public demonstration was at the IEEE Supercomputing conference, November 14-18, 1994. There, we demonstrated the version of TTL_PAPERS discussed here; however, we didn't just use a cluster of PCs under Linux. We also demonstrated the system using four IBM PowerPCs under AIX and using four DEC Alpha workstations under OSF. Despite the fact that TTL_PAPERS is a very simple and cheap unit that communicates through the parallel printer port, these demonstrations have gone far toward convincing the research community that it isn't a toy. TTL_PAPERS is a very serious introduction to a new way of thinking about how processors in a parallel machine should interact.

The TTL_PAPERS units will continue to evolve and improve; since the first PAPERS was built in February 1994 [DiM94], we have constructed and tested six different types of PAPERS units. TTL_PAPERS systems will also serve as a software testbed for a prototype supercomputer based on the same barrier synchronization technology — CARDBoard, the Compiler-oriented Architecture Research Demonstration Board. CARDBoard differs from TTL_PAPERS in that it does not center on using PCs as processing elements, but simply uses PCs as hosts for boards that each incorporate four high-performance RISC processors along with a PAPERS-like synchronization and communication facility. Thus, CARDBoard will offer much higher performance than TTL_PAPERS, but requires much more complex and specialized hardware and software.

In addition to the authors of this paper, many students have been involved in the development of PAPERS and CARDBoard. These students include Y. Choe, T. Chung, W. Cohen, R. Fisher, S. Kim, G. Krisnamurthy, C. Sheldon, and J. Sponaugle.

2. PC Hardware

Although TTL_PAPERS provides very low latency synchronization and communication, it is interfaced to PCs using only a standard parallel printer port and is implemented with a minimal amount of external hardware. This section details the PC hardware involved in use of TTL_PAPERS.

Throughout the following description, we will distinguish between stand-alone PCs and PCs used as processors within a parallel machine by referring to the later as "PEs" — processing elements. The design presented here directly supports 4 PEs (PE0, PE1, PE2, and PE3), and we also detail how the design can be scaled to 8, 16, or 32 PEs. In fact, no significant changes are needed to scale the design to thousands of processors.

2.1. PE Hardware Interface

No changes are required to make standard PC hardware into a TTL_PAPERS PE. All that is needed is a standard parallel printer port and an appropriate cable. Although some of the PCs on the market provide extended-functionality parallel ports that allow 8-bit bidirectional data connections, many PCs provide only an 8-bit data output connection. Like the original PAPERS, TTL_PAPERS can be used with any PC; it uses only the functions supported by a standard unidirectional parallel port.

But if there is no parallel input port, how does TTL_PAPERS get data into the PC? The answer lies in the fact that the 8-bit data output port is accompanied by 5 bits of status input and 4 bits of open-collector input/output (which are sometimes implemented as output only) on two other ports associated with the 8-bit data output port. The way we use these lines, there are actually 12 bits of data output and 5 bits of data input.

All versions of both TTL_PAPERS and PAPERS use all 5 available input lines. However, the various versions differ in how many and which of output signals are used. Because the open-collector lines are generally not as well driven as the 8-bit data output lines and require access to a different port address, we generally use the open-collector lines only for signals that are modal, i.e., that change relatively infrequently and can have large settling times. The version of TTL_PAPERS discussed here uses 11 of the 12 available output lines, but only the 8-bit data output port is written in the process of normal interactions with the TTL_PAPERS hardware; the 3 other bits are used only for interrupt handling.

The pin/contact assignment for each of the lines used by TTL_PAPERS is given in Table 1. Table 1 lists the pin numbers as they appear on the PE's DB25 connector. We recommend directly connecting the cable's wires to the TTL_PAPERS circuit board; thus, it is very useful to know the color code used by the wires. Table 1 includes a wiring color code. However, we have found that cables from different manufacturers differ in some of the color assignments, so this color code is to be considered only as an accurate guide to how the first group of prototypes is wired.

Table 1: DB25 Parallel Port Pin Assignments

Pin #	Std. Name	Use In TTL_PAPERS	Wire Color
Pin 1	Strobe	P_NAK, not interrupt ack	Brown
Pin 2	D0	P_D0, bit 0 of output nybble	Red
Pin 3	D1	P_D1, bit 1 of output nybble	Orange
Pin 4	D2	P_D2, bit 2 of output nybble	Pink
Pin 5	D3	P_D3, bit 3 of output nybble	Yellow
Pin 6	D4	P_LG, light LED Green	Green
Pin 7	D5	P_LR, light LED Red	Light Green
Pin 8	D6	P_S0, strobe P_RDY to 0	Blue
Pin 9	D7	P_S1, strobe P_RDY to 1	Purple
Pin 10	Ack	P_I3, bit 3 of input nybble	Gray
Pin 11	Busy	P_RDY, toggling ready signal	White
Pin 12	PE	P_I2, bit 2 of input nybble	Black
Pin 13	SlctIn	P_I1, bit 1 of input nybble	Brown, White stripe
Pin 14	AutoFD	P_IRQ, interrupt request	Red, White stripe
Pin 15	Error	P_I0, bit 0 of input nybble	Red, Black stripe
Pin 16	Init	P_SEL, int/ready select	Orange, White stripe
Pin 17	Slct	*reserved*	Orange, Black stripe
Pin 18	Gnd	—	Pink, Black stripe
Pin 19	Gnd	—	Yellow, Black stripe
Pin 20	Gnd	—	Green, White stripe
Pin 21	Gnd	—	Green, Black stripe
Pin 22	Gnd	—	Blue, White stripe
Pin 23	Gnd	—	Purple, White stripe
Pin 24	Gnd	—	Gray, Black stripe
Pin 25	Gnd	—	Black, White stripe

2.2. PE Port Bit Assignments

Although the parallel port hardware is not altered to work with TTL_PAPERS, the parallel port lines are not used as they would be for driving a Centronics-compatible printer. Thus, it is necessary to replace the standard parallel port driver software with a driver designed to interact with TTL_PAPERS. Toward this end, it is critical to understand which port addresses, and bits within the port registers, correspond to each TTL_PAPERS signal.

There are three port registers associated with a PC parallel port. These registers have I/O addresses corresponding to the port base address (henceforth, called `P_PORTBASE`) plus 0, 1, or 2. Typically, `P_PORTBASE` will be one of 0x378, 0x278, or 0x3bc, corresponding to MS-DOS printer names `LPT1:`, `LPT2:`, and `LPT3:`. Check the documentation for your PC system to determine the appropriate `P_PORTBASE` value for the parallel port that you are using as the interface to TTL_PAPERS. As a general rule, most PCs use 0x378 for the built-in port, however, IBM PCs generally use 0x3bc. Workstations based on processors other than the 386, 486, or Pentium, e.g., DEC Alphas, also generally use 0x3bc; however, most of these processors map port I/O registers into memory addresses, so you will need to replace the `inb()` and `outb()` operations with accesses to the memory locations that correspond to the specified port register I/O addresses. For example, the PowerPC specification places I/O address 0x3bc at physical memory location 0x800003bc.

Table 2: `P_PORTBASE` + 0 Bit Assignments

Hex Mask	Use In TTL_PAPERS	Pin #
0x80	P_S1, strobe P_RDY to 1	Pin 9
0x40	P_S0, strobe P_RDY to 0	Pin 8
0x20	P_LR, light LED Red	Pin 7
0x10	P_LG, light LED Green	Pin 6
0x08	P_D3, bit 3 of output nybble	Pin 5
0x04	P_D2, bit 2 of output nybble	Pin 4
0x02	P_D1, bit 1 of output nybble	Pin 3
0x01	P_D0, bit 0 of output nybble	Pin 2

The bit assignments for the first port register, `P_PORTBASE` + 0, are listed in Table 2. Each bit in the register is identified by its hex mask value, use in TTL_PAPERS, and signal pin number. This register is used to send TTL_PAPERS both strobe and data values, as well as controlling a bi-color LED (red/green Light Emitting Diode) on the PAPERS front panel.

If a PC wants to mark itself as not participating in a group of barrier synchronizations, it should simply output 0xcf; this corresponds to setting P_S1, P_S0, P_D3, P_D2, P_D1, and P_D0 all equal to 1. Notice that, if a PC is not connected to one of the TTL_PAPERS cables, the TTL inputs will all float high, causing the missing PC to be harmlessly ignored in operations performed by the PCs still connected. In contrast, setting both P_S1 and P_S0 to 0 will ensure that all barrier operations halt. In normal operation, each TTL_PAPERS operation is triggered by toggling the P_S1 and P_S0 lines between (P_S1=1, P_S0=0) and (P_S1=0, P_S0=1); this can be done by simply exclusive-oring the previous output byte with (P_S1 | P_S0).

A PC sends data to TTL_PAPERS by setting the output nybble bits appropriately and toggling the strobe lines as described above. Performing this operation as two steps, first changing data bits then changing the strobe bits, can be done to increase the reliability of transmission.

Data lines should be given time to settle before a new ready signal is derived from the new strobe signals. Changing strobes and data simultaneously results in a race condition in which the data bits have only about 20 nanoseconds "head start"—a small enough margin for many systems to perform unreliably.

The P_LR and P_LG lines are simply used to control a bi-color LED to indicate the status of the PC relative to the currently executing parallel program. When TTL_PAPERS is not in use, both bits should be set to 0, yielding a dark LED. When a parallel program is running, the LED should be lighted green, which is accomplished by making P_LG=1 and P_LR=0. When a PC is waiting for a barrier, it should make its LED red by setting P_LG=0 and P_LR=1. It is also possible to generate an orange status light by setting both P_LG and P_LR to 1, however, this setting is used only rarely (as a "special" status indication).

Table 3: `P_PORTBASE` + 1 Bit Assignments

Hex Mask	Use In TTL_PAPERS	Pin #
0x80	P_RDY, toggling ready signal	Pin 11 (inverted)
0x40	P_I3, bit 3 of input nybble	Pin 10
0x20	P_I2, bit 2 of input nybble	Pin 12
0x10	P_I1, bit 1 of input nybble	Pin 13
0x08	P_I0, bit 0 of input nybble	Pin 15
0x04	*unused, but generally 1*	—
0x02	*unused, but generally 1*	—
0x01	*unused, but generally 1*	—

The second port register, `P_PORTBASE` + 1, was intended to be a status input register, but is used to receive data from TTL_PAPERS. Bit assignments for this register are given in Table 3. Because some bits in the port registers are used for active low signals when talking to a printer, these bits have values that are the opposite of the actual signal on the corresponding pin; the PAPERS signals are all defined in terms of the sense in which the register records them, but a note appears if the sense of the signal is inverted at the pin.

To enhance the portability of TTL_PAPERS to somewhat non-standard parallel printer ports, only these five bits are used as input: four bits of data and one bit to act as a ready line. Because 0x40 is the only bit that can be enabled to generate a true interrupt to the PC, earlier versions of PAPERS made P_RDY use 0x40 so that the PAPERS unit could generate a true hardware interrupt when a barrier synchronization had completed. However, this led to an inconvenient order for the bits of the input nybble, and we never found a good use for the true hardware interrupt (because interrupt handlers have too much latency), so the current arrangement makes P_RDY use 0x80. The new arrangement is superior not only because it keeps the bits of the input nybble contiguous, but also because the inversion of P_RDY is harmless, whereas the inversion of an input data line would require extra hardware or software to flip the inverted data bit (e.g., exclusive or with 0x80).

Note also that P_RDY is a toggling ready signal. The original PAPERS unit used software to "reset" the ready signal after each barrier synchronization had been achieved, thus requiring four port operations for each PAPERS synchronization. By simply toggling P_RDY when each new barrier is achieved, TTL_PAPERS can perform barrier synchronizations using just two port operations.

Table 4: `P_PORTBASE` + 2 Bit Assignments

Hex Mask	Use In TTL_PAPERS	Pin #
0x80	*unused*	—
0x40	*unused*	—
0x20	*unused*	—
0x10	*unused, but force 0*	—
bit 3	*reserved*	Pin 17 (inverted)
bit 2	P_SEL, int/ready select	Pin 16
bit 1	P_IRQ, interrupt request	Pin 14 (inverted)
bit 0	P_NAK, not interrupt ack	Pin 1 (inverted)

The third port register, `P_PORTBASE` + 2, is described in Table 4. It serves only one purpose for TTL_PAPERS: parallel interrupt support. Actually, for the reasons described earlier, TTL_PAPERS never generates a "real" interrupt to a PC. However, parallel interrupts provide a mechanism for managing the use of the TTL_PAPERS unit in a more sophisticated way, for example: providing a better TTL_PAPERS "check-in" procedure, facilitating abnormal termination of parallel programs, implementing a user-level parallel filesystem, and even gang scheduling and parallel timesharing of the TTL_PAPERS unit.

To cause a parallel interrupt, a PC simply sets P_IRQ to 1. However, other processors will not notice that a parallel interrupt is pending unless they explicitly check. This is done by changing P_SEL to 1, which causes the normal P_RDY (described above) to be replaced by an interrupt ready flag... until P_SEL is again set to 0. Thus, any PC can check for an interrupt at any time without interfering with the operation of other PCs; for example, while delayed waiting for a barrier, it is essentially harmless to check for an interrupt. To encourage PCs to check for an interrupt, the interrupting PC can set its P_S1 and P_S0 bits to 0 (see above), forcing barriers to be delayed. When all PCs set their P_NAK to 0, this simply acts to perform a special interrupt barrier. The extended interrupt functionality is implemented by sending an interrupt code as a side-effect of this special barrier.

3. TTL_PAPERS Hardware

Thus far, this document has focused on the way in which PC hardware interacts with TTL_PAPERS. In this section, we briefly describe the hardware that implements TTL_PAPERS itself. The TTL_PAPERS design has been carefully minimized to use just 8 standard TTL 74LS-series parts and to fi t on a single-layer circuit board. The result is a remarkably simple design that is inexpensive to build, yet fast to operate.

3.1. Logic Design

The logic design for TTL_PAPERS is logically (and physically) divided into three subsystems: the barrier and interrupt mechanism, the aggregate communication logic, and the LED display control. This section briefly explains how the required functionality is implemented by the board's logic.

3.1.1. Barrier/Interrupt Hardware

Although the DBM (Dynamic Barrier MIMD) architecture presented in [CoD94] [CoD94b] is far superior to the original DBM design as presented in [OKD90a], neither one is simple enough to be effi ciently implemented without using programmable logic devices. Thus, TTL_PAPERS uses a variation on the SBM (Static Barrier MIMD) design of [OKD90]. The primary difference between the previously published SBM and the TTL_PAPERS mechanism is that there are two barrier trees rather than one. The reason is simply that the published SBM silently assumed that the barrier hardware would be reset between barriers, essentially by an "anti barrier." In contrast, the use of two trees allows the hardware for one tree to be reset as a side-effect of the other tree being used, halving the number of operations needed per barrier. Both of these two trees are trivially implemented using a 74LS20, a dual 4-input NAND. The result is latched by setting or resetting a 1-bit register implemented using 1/2 of a 74LS74.

The interrupt logic is remarkably similar to the barrier logic, also using a 74LS20 and 1/2 of a 74LS74. However, there is a difference in the connection between these chips. Interrupt requests are generated by any PE, and acknowledged by all PEs, while barrier synchronizations are always implemented by all PEs. Thus, to invert the sense of the interrupt request output from the NAND, the interrupt logic uses the simple trick of clocking the 1-bit register rather than setting it asynchronously. This trick saves a chip and actually yields a slightly more robust interrupt mechanism, because the interrupt is triggered by an edge rather than by a level.

Finally, because there are not enough input bits for each PE, the above two latched values must be independently selectable for each PE. The obvious way to select between values is using a multiplexor; however, there is no standard TTL 74LS-series part that implements selection between two individual bits. Instead, we construct the muliplexer using two quad driver chips that have individually tri-stateable outputs. The 74LS125 and 74LS126 differ only in the sense of their enable lines, thus, using the same select line to control one output on each chip makes it possible to simply wire both outputs together. Only the selected signal will be passed; the other line's driver will be in the tri-state high impedance state.

Notice also that this logic trivially can be scaled to larger machines. Every 4 PEs will require a 74LS125 and a 74LS126. The 74LS74 remains unchanged, although a driver chip may be needed to increase fan-out of the outputs for more than 8 processors. The 74LS20 NAND gates simply get replaced by larger NAND trees. For example, for 8 processors, each 1/2 of a 74LS20 is replaced by a 74LS30 (8 input NAND). For 16 processors, chip count is minimized by implementing each NAND tree using a 74LS134 (12-input NAND) and 1/2 of a 74LS20, the outputs of which are combined using 1/4 of a 74LS32 (quad 2-input OR). A somewhat neater 16 processor board layout results from using 2 74LS30 and 1/4 of a 74LS32 for each of the NAND trees, and a 32 processor system can easily be constructed using 4 74LS30 and 3/4 of a 74LS32 for each NAND tree. If you have more than 32 machines, we strongly recommend building something fancier than a single SBM.

3.1.2. Aggregate Communication Hardware

Although other versions of PAPERS provide internal data latching and fancy communication primitives, TTL_PAPERS simply offers NANDing of the 4 data bits across the processors.

In general, there is nothing tricky about building these NAND trees; they look exactly as described above. However, each NAND tree has an output that must drive one line to each of the processors, and the use of relatively long cables (up to about 10 feet each) requires a pretty good TTL driver. For the case of 4 processors, it happens that the 74LS40 (dual 4-input NAND buffer) provides sufficient drive to directly power all 4 lines, although the signal transitions become somewhat slow (e.g., about 300 nanoseconds) as the cables get long. In a larger system, we recommend constructing the NAND tree as described above, and then using 74LS244 or 74LS541 octal drivers to increase the drive ability of the NAND outputs. With a typical cable, each driver within a 74LS244 or 74LS541 can drive up to about 4 lines.

3.1.3. LED Display Hardware

In the PAPERS prototypes, we have experimented with a variety of different status displays. However, by far the easiest display to understand was one using just a single bi-color LED for each processor. The color code is very intuitive: green means running, red means waiting (for more than two "clock" cycles), and black means not in use. The problem is that there is no trivial way to derive these color choices directly from the barrier logic, thus, the LEDs are explicitly set under software control.

There are a variety of different types of bi-color LEDs, however, the cheapest and most common have three-leads: power for each of the two colors and a common ground. Because TTL can sink more than it can source, we would rather have seen a common power input and separate grounds... oh well. The result is that each color of each LED must be fairly well driven, and it isn't a good idea to make the computer port directly drive the lights. Instead, a 74LS541 is used. For scaling, each 74LS541 can handle all the LED drive for 4 processors.

There is also a power LED on the TTL_PAPERS unit, and that actually yields a minor complication. To avoid confusion with the rest of the status display, the TTL_PAPERS power LED is blue. Although blue LEDs are really nice, they do not appear as bright as other colored LEDs. Consequently, we found that 1K Ohm resistors for driving the red/green bi-color LEDs just about matched the brightness of a 330 Ohm resistor for driving the blue power LED. Actually, green was slightly darker than red or blue with this drive, so perhaps the green LEDs should be driven through about 800 Ohms? In any case, you may find that your LEDs differ in brightness from ours, so some playing with resistor values may be appropriate.

It is also worth mentioning that, if you can't find appropriate bi-color LEDs, you can simply use red LEDs and ignore the green LED drivers. Likewise, if you can't find an appropriate blue LED for the power indicator, a green LED can be substituted.

3.2. Parts List

The complete parts list for constructing a TTL_PAPERS is given in Table 5. The total parts cost for a unit should be between $20 and $50, depending on the quantity and source for the parts. For example, the cables can be purchased for less than $2 each via mail order, but can be more than $20 each if purchased from some stores. We strongly recommend checking prices from several mail order sources for cables and other parts.

Along the same lines, if your computers do not have unused parallel printer ports, there is a very wide range in the pricing of ISA bus parallel port cards. We have found a $7 card via mail order that works fine; the $45 or more that you would spend for each card in some stores isn't likely to buy you any improvement in performance. In fact, if there is a speed difference between the card and the built-in port, it is harmless to use the slower port for a printer while the faster port is used for the TTL_PAPERS connection. Do *not* use a port-switcher or similar device to share a single port; TTL_PAPERS might not work with such devices.

Table 5: TTL_PAPERS Parts List

Used In	Label	Description	Quantity
Box		Red Oak 1x4 (3/4" by 3 1/2")	9" +
Box		Poplar 1x4 (3/4" by 3 1/2")	9 1/2" +
Box		Poplar scant board or Plywood (1/4" by 3 1/2")	11 1/4" +
Box		1 1/4" all-purpose screws	8
Cables		10’ 25-wire Male DB25 to Male DB25	2
Display	U8	74LS541, octal Buffer	1
Display	D2-D5	3-lead Bi-Color Red/Green LED	4
Display	D1	Blue LED	1
Display	R2-R9	1K Ohm resistor	8
Display	R10	330 Ohm resistor	1
Logic		Printed Circuit Board (see Figures 11-13)	1
Logic	U1, U2	74LS20, dual 4-input NAND	2
Logic	U6, U7	74LS40, dual 4-input NAND Buffer	2
Logic	U5	74LS74, dual D Flip-Flop with Preset & Clear	1
Logic	U3	74LS125, quad Tri-State Bus Buffer	1
Logic	U4	74LS126, quad Tri-State Bus Buffer	1
Logic	R1	1K Ohm resistor	1
Logic	C1	0.001 microfarad Capacitor	1
Power		9-12 volt, 200 ma AC to DC adaptor	1
Power		Power receptacle (to match above)	1
Power		Ground post (e.g., small screw eye)	1
Power	U9	7805, 5 volt regulator	1
Power	C3, C4, C6-C8	0.1 microfarad Capacitor	5
Power	C5	47 microfarad Capacitor	1
Power	C2	100 microfarad Capacitor	1

3.3. Construction

Constructing a TTL_PAPERS unit requires building two separate items: the box and the board. Because the board is directly wired to the cables (i.e., no connectors are used), and the box acts as a mount for the circuit board, LEDS, ground post, and cables, it is much easier to assemble the box first and then assemble the board. The following two sections detail exactly how the box and board can be constructed.

3.3.1. Building The Box

TTL_PAPERS is housed in a very simple hardwood case, but a lot of people ask us, "where did you get the custom boxes?" Well, we designed and built them. It isn't that difficult. We could use a standard metal or plastic case, but they just are not as nice. The wooden boxes also are cheaper because they can simply be drilled to create mounts for LEDs, cables, and power connector —mounting hardware is not needed.

The materials you will need to construct the TTL_PAPERS box are listed in Table 5. As specified, this constitutes between $5 and $10 worth of hardwood lumber; however, almost any hardwood, or even pine, can be substituted for the 1x4 Red Oak and Poplar. Red Oak was selected because it is very durable and yields a beautiful natural finish; Poplar was selected because it is cheap and also is very durable.

Throughout the construction, be sure to read, understand, and follow all safety instructions and be sure to wear safety glasses. Woodworking isn't particularly dangerous, but doing stupid things with power tools can lead to serious injuries.

Figures 1 and 2 show the front view and right side view of the completed box. Figures 3 through 10 show the individual pieces that are assembled to make the box, including the positions of rabbets, holes, and screw countersink holes. Although these figures were generated by `groff` and `pic` using the exact dimensions of the pieces, the processing through these tools and a postscript interpreter is likely to result in printed output that does *not* maintain the precise dimensions. Check the actual dimensions before you use these figures as templates. If the error is less than 1/16" over a 6" span, directly using the figures as templates is unlikely to cause serious problems in assembling the box.

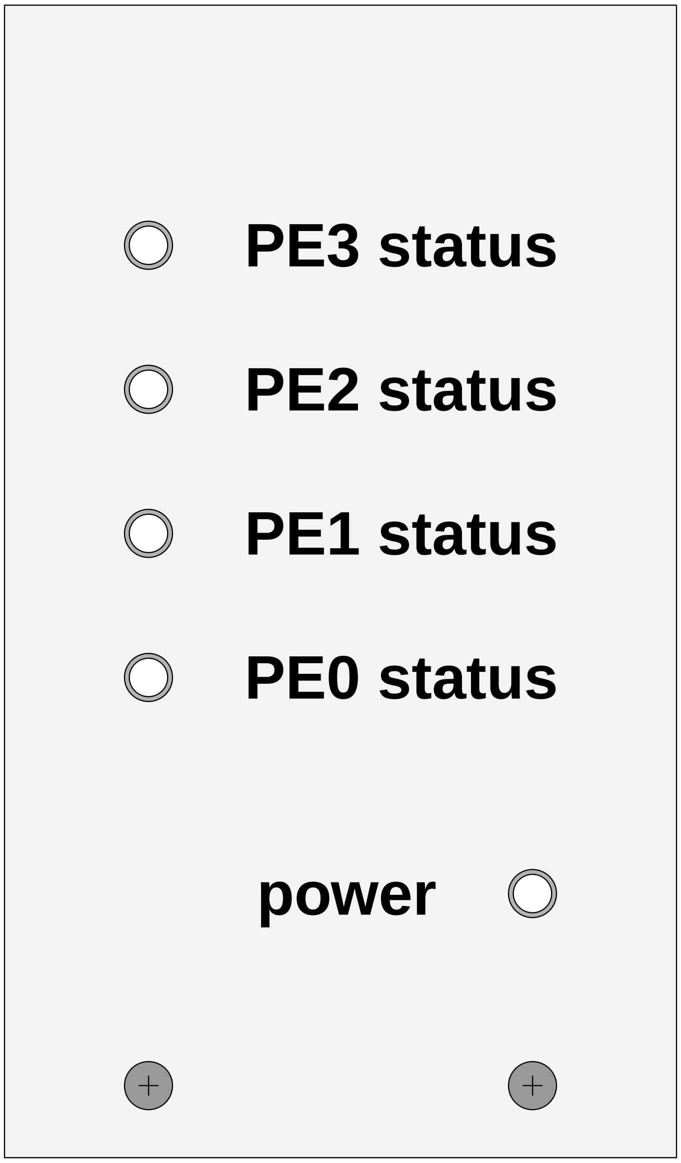

Figure 1: Assembled TTL_PAPERS Box (Front View) 3 1/2" x 6"

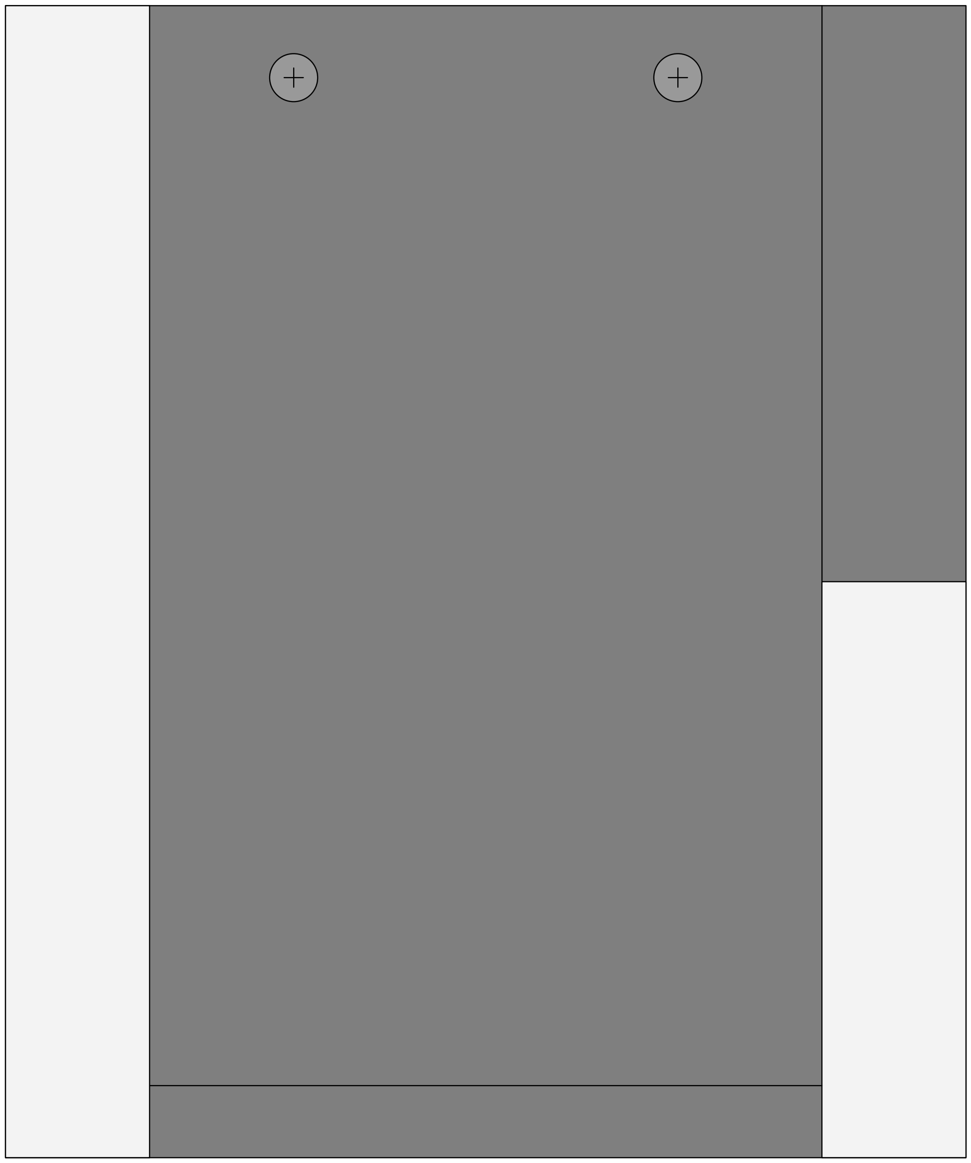

Figure 2: Assembled TTL_PAPERS Box (Right View) 5" x 6"

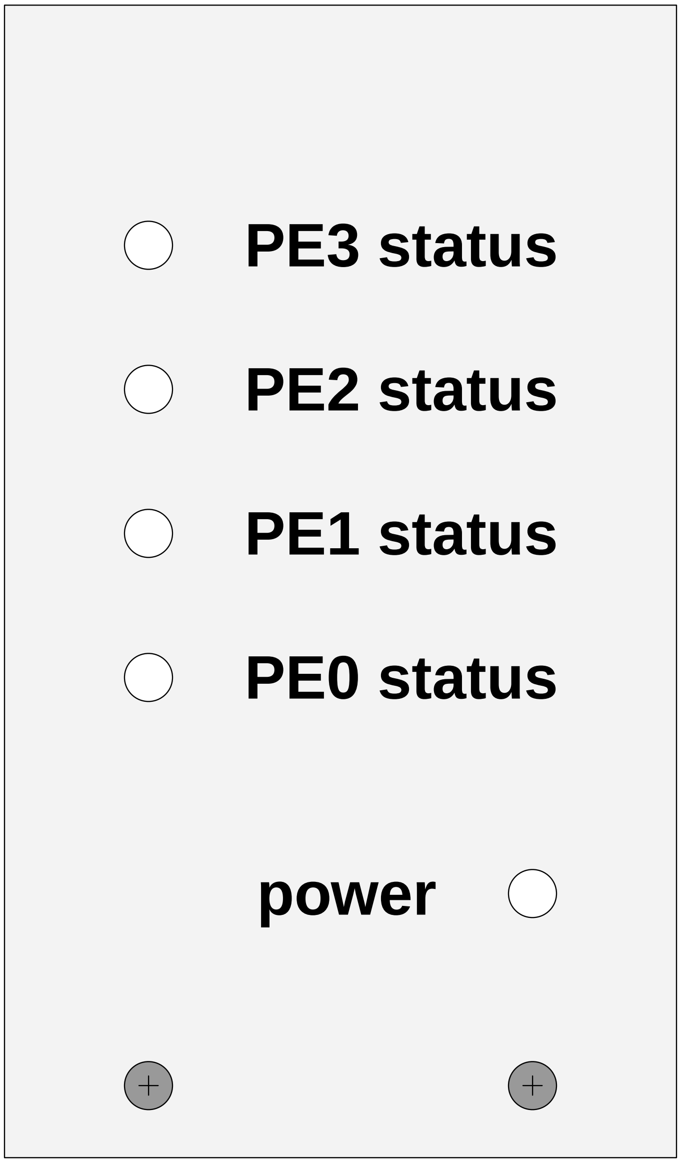

Figure 3: Front Panel 3 1/2" x 6" x 3/4"

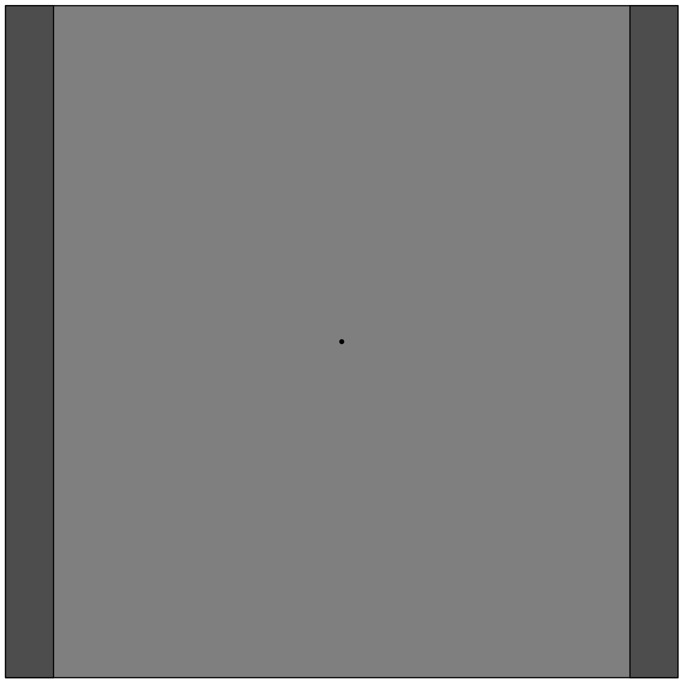

Figure 4: Base (Top View) 3 1/2" x 3 1/2" x 3/4", rabbets shaded

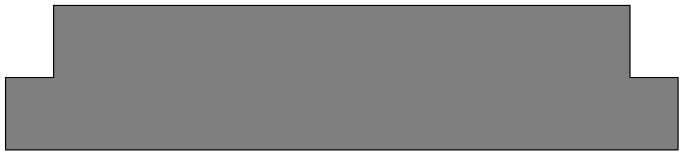

Figure 5: Base (Front View) 3 1/2" x 3 1/2" x 3/4"

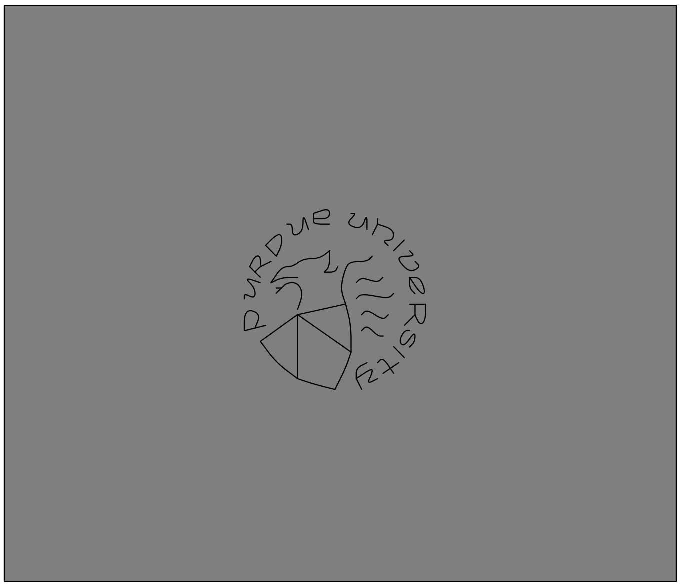

Figure 6: Cover Back 3 1/2" x 3" x 3/4"

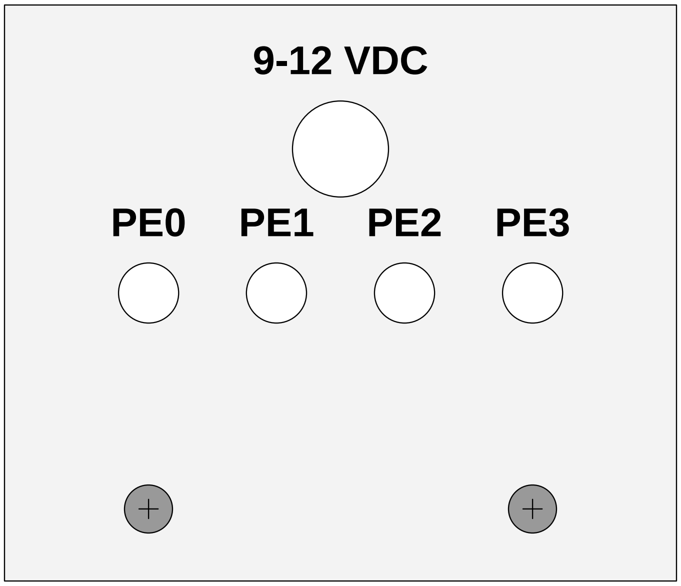

Figure 7: Base Back 3 1/2" x 3" x 3/4"

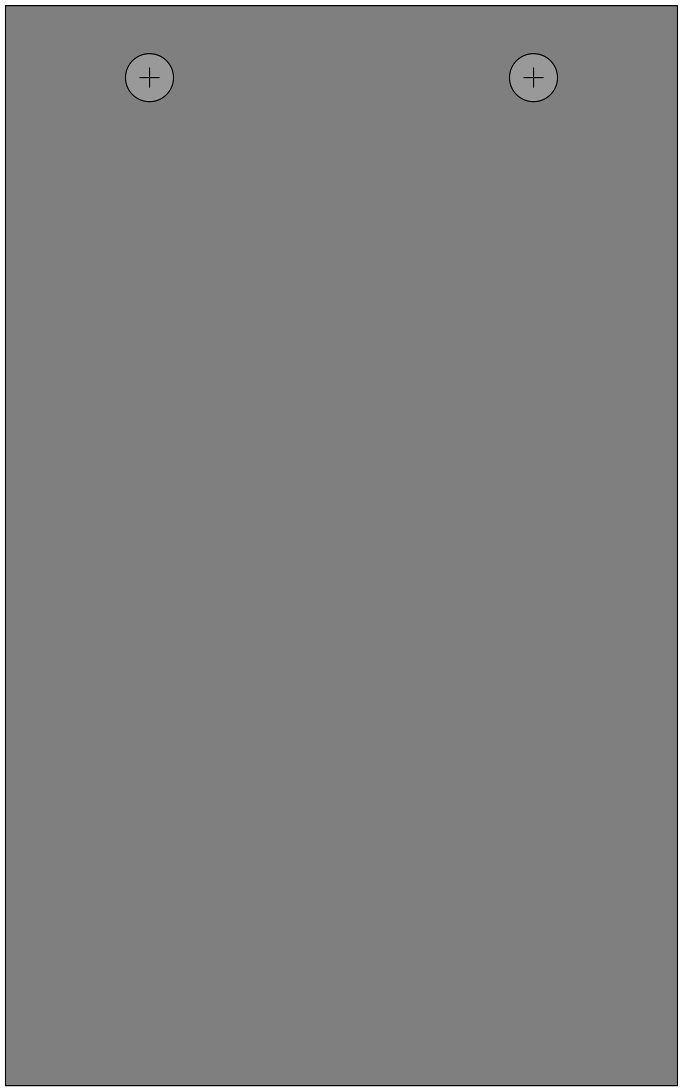

Figure 8: Cover Right Side 3 1/2" x 5 5/8" x 1/4"

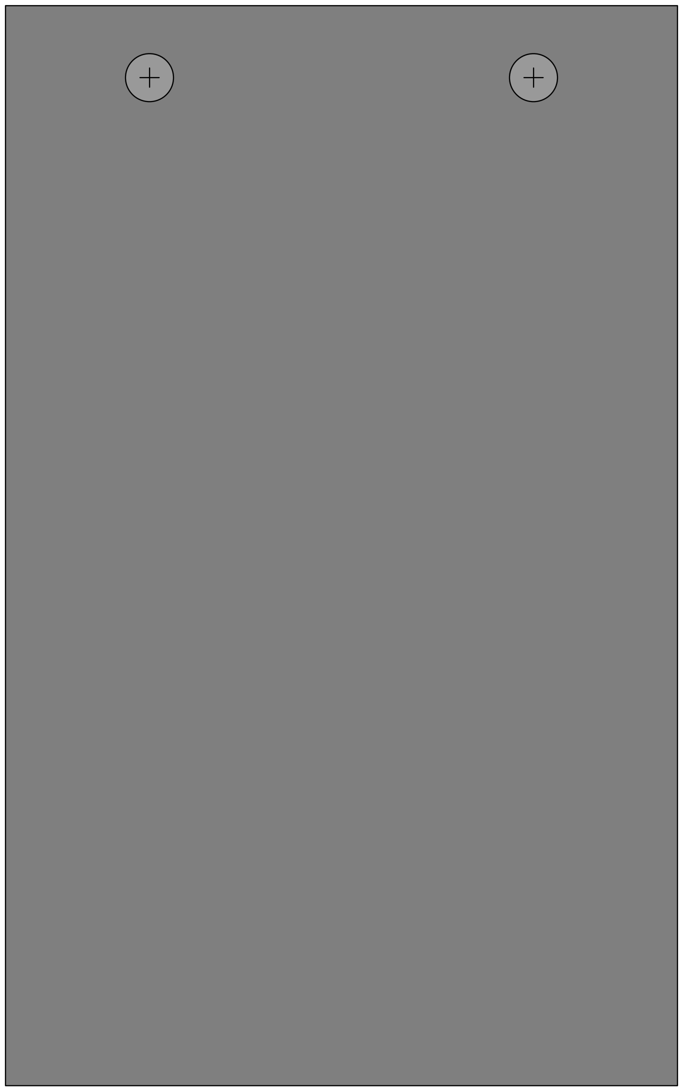

Figure 9: Cover Left Side 3 1/2" x 5 5/8" x 1/4"

Figure 10: Cover Top 3 1/2" x 3" x 3/4"

The following step-by-step instructions are for making a single TTL_PAPERS box. Recall that the parts list is given in Table 5 and that Figures 3 through 10 give templates for the individual pieces of wood after they have been cut to size. Read through all the steps before you begin construction. If more than one box is being built, it is generally easier and safer to perform step [2] before cutting the Poplar 1x4 to length.

[1] Using a miter box or some other crosscut saw, make the following 90° cuts. Cut the Red Oak 1x4 to make two pieces, one 6" long and one 3" long. Cut the Poplar 1x4 to make three pieces, two 3" long and one 3 1/2" long. Cut the poplar scant board to make two pieces, each 5 5/8" long.

[2] The 3 1/2" long piece of Poplar will become the base of the box (Figures 4 and 5). In order to prevent the box cover sliding from side to side, a rabbet is cut into each side of the base. This rabbet should be 3/8" deep and 1/4" wide. The best way to make these rabbets is to use a router table, table saw, or radial arm saw, although a router with a rabbeting bit will suffice. You may find it easier to cut the rabbets *before* cutting the Poplar 1x4 to the 3 1/2" length.

[3] You've now got a lot of drilling to do. A drill press is recommended, but a hand-held power drill can suffice. You can use the diagrams provided in this document as templates to align your holes, but you will need to determine the appropriate drill sizes based on the particular parts you are using. Always use a backer board and/or masking tape to reduce splintering when drilling. The holes to drill are:

- Five holes for the LEDs on the front panel (Red Oak 6" piece shown in Figure 3). These holes should first be drilled from the front so that the LEDs are a snug fit and cannot be pushed through the panel (there is a little "lip" at the base of each LED that is slightly wider and thus acts as a stop). Next, the panel is flipped and a slightly larger drill is used to widen each hole to a depth of about 1/2". The resulting holes will allow the LEDs to be inserted from the back of the panel so that they are flush with the front of the panel, with the lip on each LED firmly seated against the point where the hole narrows.
- Four holes for the cables on the back panel (Red Oak 3" piece shown in Figure 7). These holes should be drilled to precisely match the outside diameter of the cables being used, so that each cable has to be "worked" through.
- One hole for the power receptacle on the back panel (Red Oak 3" piece shown in Figure 7). These receptacles come in a wide variety of shapes and sizes, some of which are much easier to mount than others. The plastic receptacle we have used on many of the TTL_PAPERS boxes requires a very small hole that is partly squared-off — not a serious problem if you happen to have a mortising jig, but nearly impossible otherwise. We recommend using a circular metal receptacle that was designed for a panel mount with a locking nut; simply drill the hole to exactly match the size of the threaded portion of the receptacle. Later, when you are assembling the box, you will

be able to screw-in the receptacle, with the metal thread acting as a tap to cut a matching thread in the hardwood.

- Eight countersink holes for the screws that will hold the assembly together (two each on the Red Oak 6" shown in Figure 3, Red Oak 3" shown in Figure 7, and the Poplar 5 5/8" pieces shown in Figures 8 and 9).

[4] Having completed the above, the box is ready for assembly; however, it is generally easier to do some of the finish work on the parts before they are assembled. Thus, this is a good time to sand and paint the component parts. The Poplar parts (Figures 4 and 5, 6, 8, 9, and 10) are given a coat of black paint, the Red Oak parts (Figures 3 and 7) are given a coat of sanding sealer.

[5] Assemble the lower half of the box. Before assembly, install the ground post in the center of the base. Next, butter one end of the base Poplar 3 1/2" piece (Figures 4 and 5) with yellow carpenter's glue and clamp it flat near the edge of a table. Position the front panel Red Oak 6" piece (Figure 3) and screw it into the base. Rotate this unit, glue, and screw the back panel Red Oak 3" piece (Figure 7) onto the base.

[6] Assemble the cover of the box. The two side Poplar 5 5/8" pieces (Figures 8 and 9) ought to rest snugly, with the countersink holes at the top edge, in the rabbets between the front and back panels... but they probably don't quite fit. Use a power sander, plane, or joiner to shave them to a good fit. You may also have to fine-tune the length of these pieces so that they match each other. Next, take one of the Poplar 3" pieces (Figure 10) and glue and screw the side pieces to it much as you did for the lower half of the box in step [5]. Finally, to assemble the upper back piece of the cover, lay the box on its front panel with the cover in place and position, glue, and clamp the last Poplar 3" piece (Figure 6). Refer to Figure 2 to see the assembled cover. Because the cover is painted black, a multitude of minor alignment errors tend to be hidden....

[7] Once the glue has set, give the box a final sanding and finishing. Generally, you'll have to retouch the black at least where you had trimmed the side panels. If the box is likely to be handled often, we suggest giving the entire box a few coats of a satin finish clear polyurethane —this yields a nice, hard, finish that is both tougher and more consistent than if some parts were just painted black. One other issue to consider at this time is RF shielding; wood is not conductive, so you might want to use a conductive paint or to glue aluminum foil on the inside of the box. Because there are typically no signals over about 1MHz within the TTL_PAPERS unit, shielding should not be a major issue.

Notice that the completed box consists of two separate pieces; the cover is not permanently attached to the base. If desired, a strip of Velcro can be used to hold the two units together by the seam between the cover and base at the back of the unit. However, the interlocking fit of the cover and base makes a fastener unnecessary unless the box will be handled often.

3.3.2. Building The Board

All the circuitry for the complete TTL_PAPERS unit, including the logic, display components, and power conditioning, fits on a 2 1/2" by 4" single-layer circuit board. Aside from turning-off auto-routing and designing the board traces by hand, a few tricks were needed to accomplish this minor miracle:

- Had we used standard circuit-board-mount DB25 connectors for each of the cables, the board would have been much more expensive, larger, and would have required at least two layers. However, by bringing each wire from each cable separately and directly to the position where the signal is used on the board, a single layer suffices for all but one connection. A single jumper (marked J1) implements this connection.
- One of the chips, namely the LED driver, is turned sideways relative to the other chips.
- The display LEDs are mounted on the back side of the circuit board. In other words, the side that doesn't have the chips on it has the LEDs sticking up by about 3/4". Thus, when the board-mounted LEDs are aligned with and inserted into the corresponding holes in the back side of the front panel of the box, the front of the LEDs is flush with the front of the front panel when the back of the circuit board is flush against the back of the front panel. Soldering the LEDs onto the back side of the circuit board isn't very difficult because the leads are so long, and this LED mounting is very effective.

The result is a very inexpensive board that can be made using just the most basic etching materials (e.g., as found in Radio Shack) and a small drill press with which to drill the holes.

The board is fairly easy to build, but be very careful to follow the instructions, because some steps just cannot be done in any other order. For example, you cannot install the jumper (step 4) after populating the board (step 5). Likewise, you cannot connect the cables (step 7) and then pass them through the holes in the box (step 6). The construction procedure is:

[1] Using the 2X board image given in Figure 11, etch a 2 1/2" by 4" single-sided circuit board. The words "TTL PAPERS" should be etched such that they read correctly when viewed directly on the solder side of the board.

[2] Although Figure 13 provides a label mask that is suitable for use as a component-side silkscreen, the expense of silkscreening is not needed. You could do the silkscreen now, but we suggest simply using Figure 13 as a guide during assembly.

[3] Using the 2X drilling mask given in Figure 12 as a guide, drill the holes in the board. (If you will be manually drilling the holes, you might find it easier to combine Figures 11 and 12 to produce an negative hole mask that can be etched along with the board traces.)

[4] Remember the J1 jumper we mentioned before? Install this jumper before you install any other parts. One of the holes for this jumper is directly underneath an IC in the finished board, so you should use thin wire for this jumper, e.g., 30-gauge wire-wrap wire works fine.

[5] Populate the board. Figure 13 gives the component-side label mask, and both Figure 14 (the circuit diagram) and Table 5 (the parts list) cross-reference these labels with the components they represent. Because most of these chips cost less than sockets and TTL parts are relatively robust, all the ICs can be soldered directly to the board. Also insert the 7805, capacitors, and resistors as marked; note that resistor R1 should be inserted standing vertically rather than laying flat on the board. As mentioned earlier, the LEDs are installed about 3/4" high off the solder side of the board. Be sure that the orientation is correct for 7805, electrolytic capacitors, LEDs, and ICs -- 180° rotation might still fit, but it will not work for these parts.

[6] Prepare the cables. The cables are purchased as 10' cables with a male DB25 connector on each end, so the first step is to cut each of the two cables in half, yielding four cables that each have a male DB25 on one end and a clean cut on the other end. Pass the clean cut end of each cable through one of the tight-fitting holes you drilled in the back of the box. Now strip the cable sheath back about 6" on each cable, and cut the sheath and shielding away from the wires. Now is also a good time to strip each wire (and to tin it with solder, if you wish); strip the wires only about 1/8".

[7] Determine the correct cable color code. Each cable should contain 25 wires, one connected to each pin on the DB25 — if it doesn't, you cannot use the cable you have. Generally, these cables use color-coded wires, but there is no real standard for which color goes to which pin. Consequently, although Table 1 provides the color code that was used on the cables for the first TTL_PAPERS boxes we built, your cables probably differ by at least a few color assignments. Use a continuity checker to determine how your cable's color code differs from the one listed in Table 1, and make your own corrected version of Table 1. Using that table along with Figure 13, it is a simple matter to determine which wire carries each signal.

[8] Connect the cables to the ground post. Of the 25 wires in each cable, 8 are signal ground connections. We strongly recommend creating a single-point ground connection for all the cables and the power supply connection. Do this by collecting the eight signal ground wires from each cable, stripping the wires about 3/8", twisting them together tightly, and soldering each twisted group directly to the ground post.

[9] Connect the cables to the board. At this time, you have the somewhat daunting task of connecting the remaining wires from the cables to the appropriate (and apparently randomly located) spots on the board. This really is not as bad a problem as it seems, but *neatness* counts! We recommend that you connect one signal for all cables before moving on to the next signal, and further suggest that you loosely twist the last cable's wire around the same signal wires from the other cables. About 3 times around the other wires is sufficient to keep the wires together as a group.

Notice that, for most wires, the order of connection of the wires from the different cables (i.e., processors) does not matter; for example, the order of inputs to a NAND gate is irrelevant to the NAND operation. The cable (i.e., processor) number is used as a suffix on the label for the four signals that are order sensitive (i.e., SEL, RDY, LG, and LR). Because TTL_PAPERS doesn't use one of the signals, you should have four wires left unconnected when you are done; wrap these individually in electrical tape and tape the bunch to inside base of the box.

[10] Make the power and ground connections. Connect a 6" lead to the power receptacle's power and a 3" lead to the power receptacle's ground, then install the power receptacle in the box. Next, solder the 3" ground wire to the ground post in the base of the box. Next, solder a 4" ground wire from the ground post to the ground connection point on the circuit board. Finally, loosely twist the 6" power lead around the 4" ground wire you just connected, and solder the 6" power lead to the power input point on the circuit board.

[11] Complete the mounting of the board in the box. The board itself probably needs no mounting hardware; simply line-up the LEDs with the holes in the back of the front panel and press the board flat against the back of the front panel. The force of the wires against the board and the fit of the LEDs in their mounting holes will tend to hold the board in place, but a couple of pieces of Velcro or a small screw through each of the two mounting holes in the board will ensure that the board stays in place. Likewise, the heavy connections to the ground post essentially ensure that the cables will not pull out, but a small dab of super glue, or a thin wrapper around the cables just after they enter the box, will provide an even safer mount.

As for the box, you may find that some things become easier if you are building more than one TTL_PAPERS board at a time. For example, if you will be building several units, purchasing all the cables from one batch will allow you to use the same color code for all the boxes.

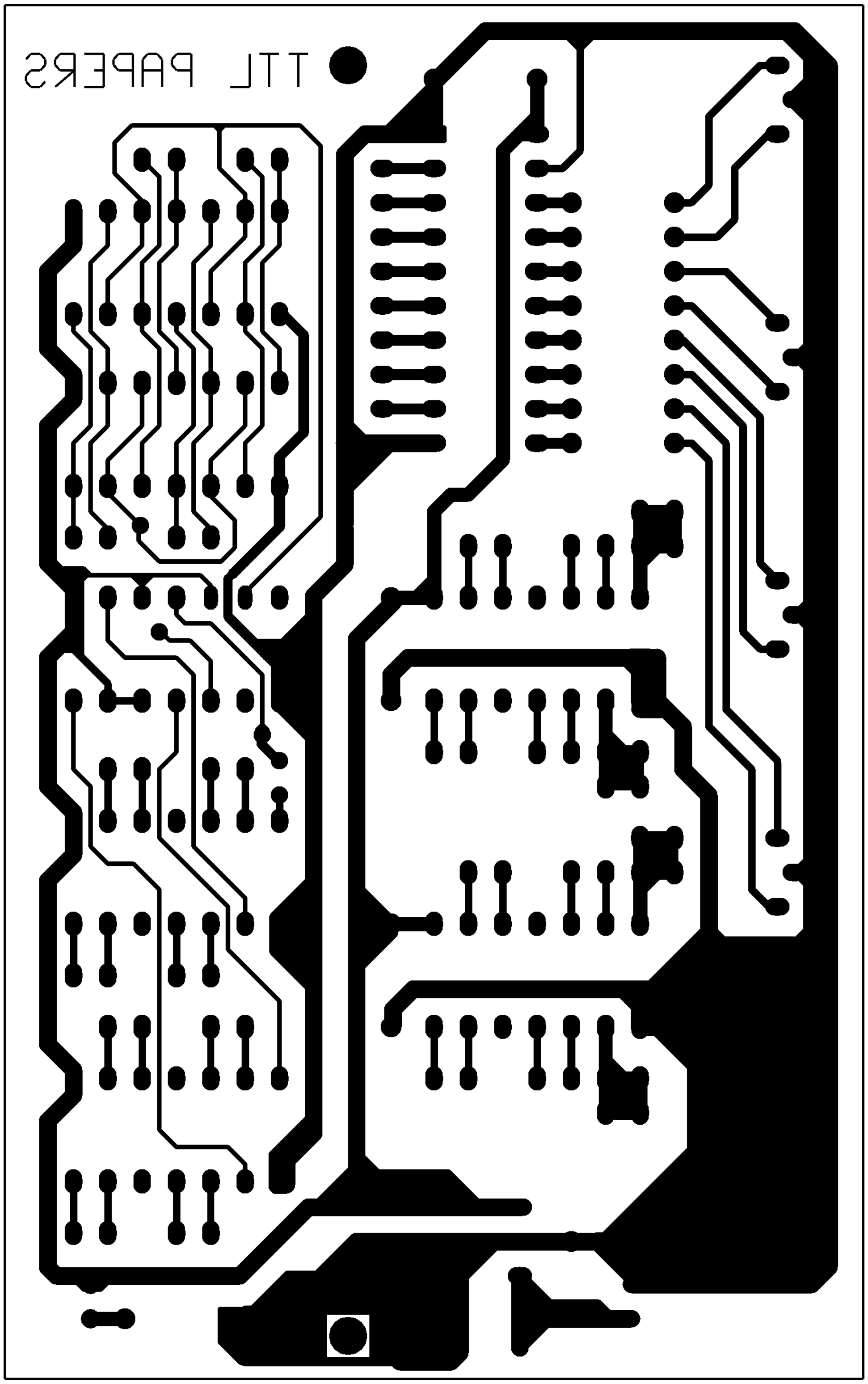

Figure 11: Circuit Board Traces, 2X Plot

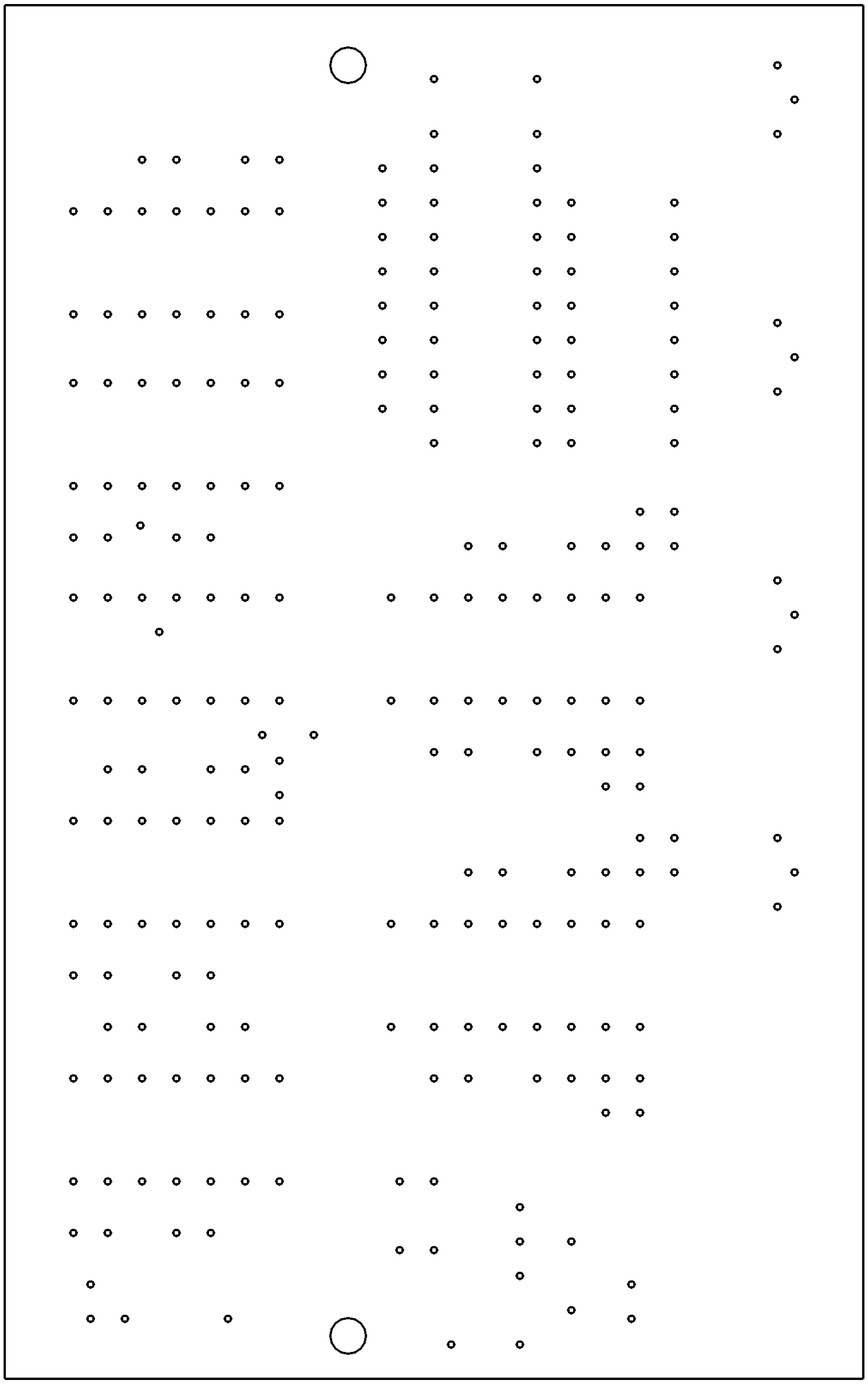

Figure 12: Circuit Board Holes, 2X Plot

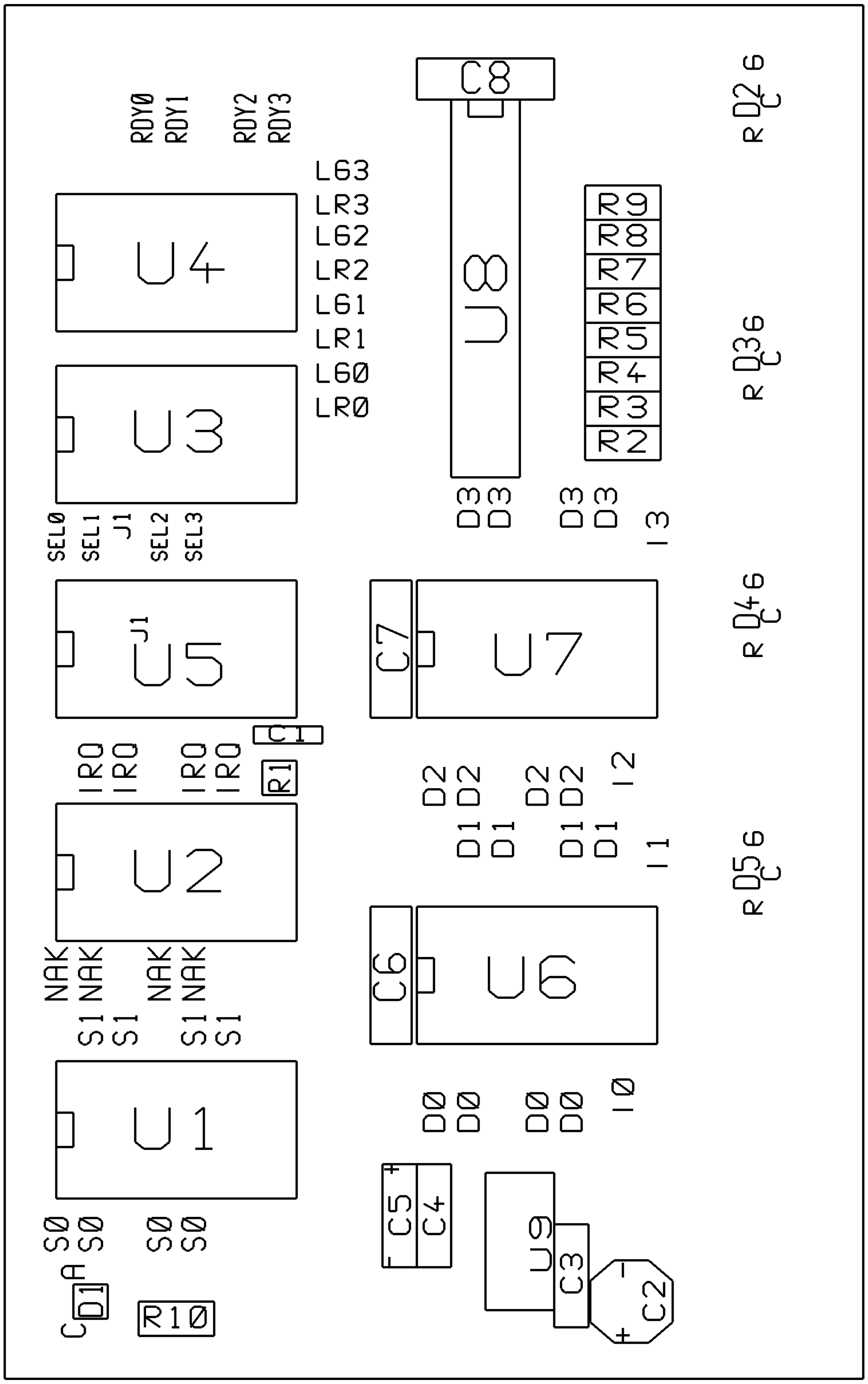

Figure 13: Circuit Board Labels, 2X Plot

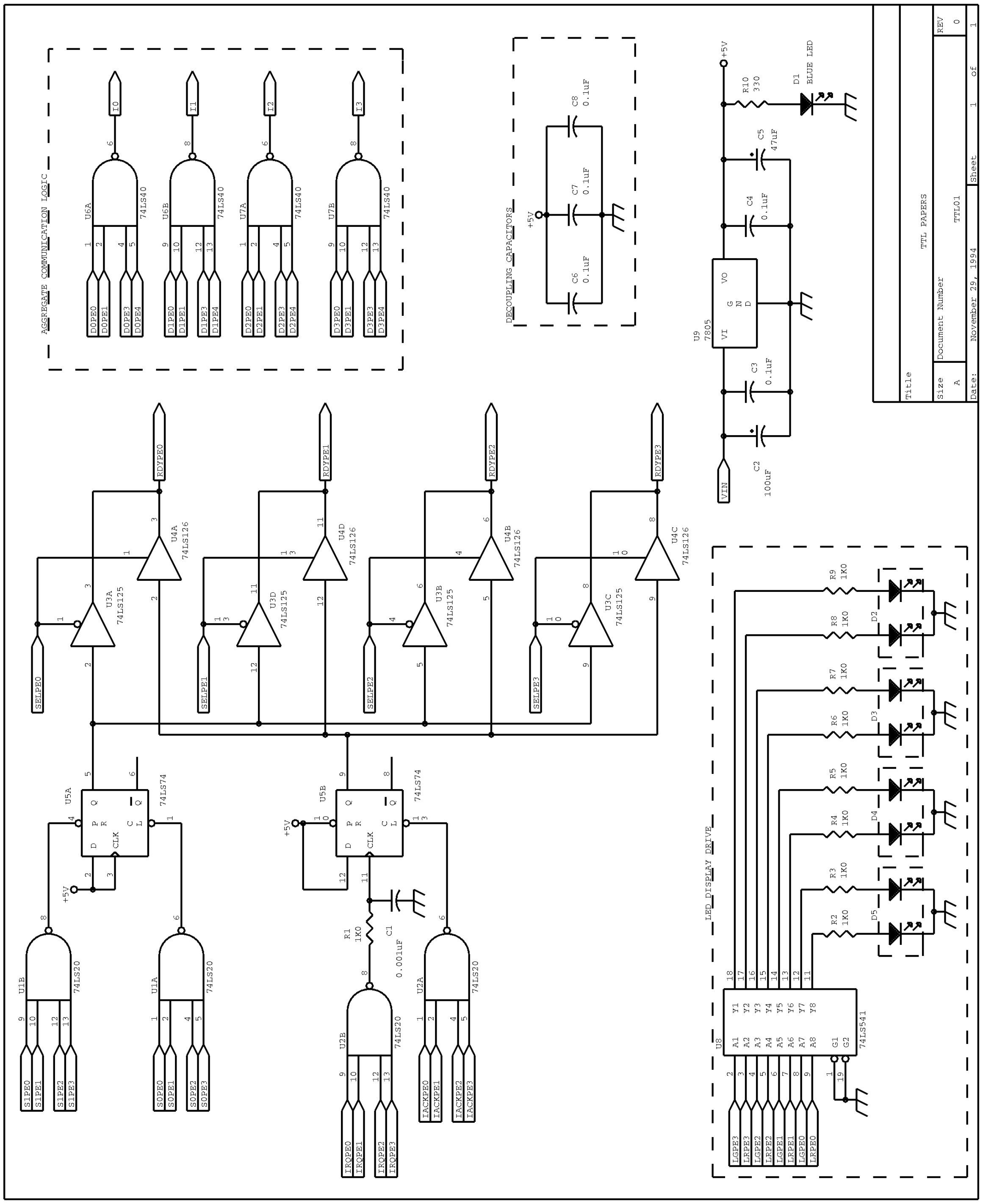

Figure 14: TTL_PAPERS Circuit Diagram

4. PAPERS Software

Although TTL_PAPERS will be supported by a variety of software tools including public domain compilers for parallel dialects of both C and Fortran [DiO92] [CoD94a], in this document we restrict our discussion to the most basic hardware-level interface. The code given is written in C (the ANSI C-based dialect accepted by GCC) and is intended to be run under a UNIX-derived operating system. However, this interface software can be adapted to most existing (sequential) language compilers and interpreters under nearly any operating system.

The following sections discuss the operating system interface, TTL_PAPERS port access, the basic barrier interface, and how NANDing is used to implement data communication.

4.1. Operating System Interface

Although it would certainly be possible to implement the TTL_PAPERS software interface as part of an operating system's kernel, typical latency for a minimal system call is between 5 and 50 times the typical latency for the TTL_PAPERS hardware port operations —layering overhead of a system call for each TTL_PAPERS operation would destroy the low latency performance. Thus, the primary purpose of the OS interface is to obtain direct user-level access to the ports that connect to TTL_PAPERS.

On older architectures, such as the 8080 and Z80, direct user access to I/O was accomplished by simply using the port I/O instructions or by accessing the memory locations that corresponded to the desired memory-mapped I/O device registers. Things are now a bit more complex. Using a PC based on the 386, 486, or Pentium processor, port I/O instructions are now protected operations. Similarly, processors like the DEC Alpha, PowerPC, and SPARC use memory mapped I/O, but physical addresses corresponding to I/O ports generally are not mapped into the virtual address space of a user process. None of these problems is fatal, but it can take quite a while to figure out how to get things working right, and you might have to write your own kernel driver to set things up.

Thus, although the parallel printer port can be directly accessed under most operating systems, here we focus on how to gain direct user process access to I/O ports on a 386, 486, or Pentium-based personal computer running either generic UNIX or Linux.

4.1.1. Generic UNIX

In general, UNIX allows user processes to have direct access to all I/O devices. However, only processes that have a sufficiently high I/O priority level can make such accesses. Further, only a privileged process can increase its I/O priority level —by calling `iopl()`. The following C code suffices:

```
if (iopl(3)) {
    /* iopl failed, implying we were not priv */
    exit(1);
}
```

But beware! This call grants the user program access to *all* I/O, including a multitude of unrelated ports.

In fact, this call allows the process to execute instructions enabling and disabling interrupts. By disabling interrupts, it is possible to ensure that all processors involved in a barrier synchronization act precisely in unison; thus, the average number of port operations (barrier synchronizations) needed to accomplish some TTL_PAPERS operations can be reduced. However, background scheduling of DMA devices (e.g., disks) and other interference makes it hard to be sure that a UNIX system will provide precise timing constraints even when interrupts are disabled, so we do not advocate disabling interrupts.

Even so, performance of the barrier hardware can be safely improved by causing UNIX to give priority to a process that is waiting for a barrier synchronization. This improves performance because if any one PE is off running a process that has nothing to do with the synchronization, then all PEs trying to synchronize with that PE will be delayed. The priority of a privileged UNIX process can increased by a call like:

```
/* set priority just below critical OS code */
nice(-20);
```

The argument to `nice()` should be a negative value between -20 and -1.

4.1.2. Linux

Although Linux supports the UNIX interface described in the previous section, it also provides a more secure way to obtain access to the I/O devices. The `ioperm()` function allows a privileged process to obtain access to only the specifi ed port or ports. The C code:

```
if (ioperm(P_PORTBASE, 3, 1)) {
    /* like iopl, failure implies we were not priv */
    exit(1);
}
```

Would obtain access for 3 ports starting at a base port address of `P_PORTBASE`. Better still, if the operating system is managing all parallel program interrupts, only the fi rst two ports need to be accessible:

```
if (ioperm(P_PORTBASE, 2, 1)) {
    /* like iopl, failure implies we were not priv */
    exit(1);
}
```

Because the 386/486/Pentium hardware checks port permissions, this security does not destroy port I/O performance; however, checking the permission bits does add some overhead. For a typical PC parallel printer port, the additional overhead is just a few percent, and is probably worthwhile for user programs.

4.2. Port Access

Although Linux and most versions of UNIX provide routines for port access, these routines often provide a built-in delay loop to ensure that port states do not change faster than the external device can examine the state. Consequently, the TTL_PAPERS support code uses its own direct assembly language I/O calls. The code is:

```
inline unsigned int
inb(unsigned short port)
{
    unsigned char _v;
__asm__ __volatile__ ("inb %w1,%b0"
        :"=a" (_v)
        :"d" (port), "0" (0));
    return(_v);
}

inline void
outb(unsigned char value,
unsigned short port)
{
__asm__ __volatile__ ("outb %b0,%w1"
        : /* no outputs */
        :"a" (value), "d" (port));
}
```

The basic TTL_PAPERS interface is thus defined in terms of the above port operations on any of three parallel printer port interface registers. The following definitions assume that `P_PORTBASE` has already been set to the base I/O address for the port, which is generally 0x378 on PC clones and 0x3bc for IBM ValuePoint PCs.

```
/*      Stuff concerning the regular output port...
*/
#define P_OUT(x) \
        outb(((unsigned char)(x)), \
             ((unsigned short) P_PORTBASE))
#define P_S1 0x80           /* Strobe RDY -> 1 */
#define P_S0 0x40           /* Strobe RDY -> 0 */
#define P_S  (P_S0 | P_S1)  /* Strobe Mask */
#define P_LR 0x20           /* LED Red */
#define P_LG 0x10           /* LED Green */
#define P_LY (P_LG | P_LR)  /* LED Yellow (Red + Green) */
#define P_LB 0x00           /* LED Black */
#define P_D3 0x08           /* Data Bit 3 Value */
#define P_D2 0x04           /* Data Bit 2 Value */
#define P_D1 0x02           /* Data Bit 1 Value */
#define P_D0 0x01           /* Data Bit 0 Value */

/*      Stuff concerning the input port...
*/
#define P_IN() \
        inb((unsigned short) (P_PORTBASE + 1))
#define P_RDY     0x80  /* Ready */
#define P_INT     0x80  /* INTerrupt */
#define P_I3      0x40  /* PE3 P_D bit */
#define P_I2      0x20  /* PE2 P_D bit */
#define P_I1      0x10  /* PE1 P_D bit */
#define P_I0      0x08  /* PE0 P_D bit */
#define P_IN_MASK 0x78  /* Mask for all the above bits */

/*      Stuff concerning the mode port...
*/
#define P_MODE(x) \
        outb(((unsigned char)(x)), \
             ((unsigned short) (P_PORTBASE + 2)))
#define P_SEL 0x04  /* INT/RDY SELect */
#define P_IRQ 0x02  /* Interrupt ReQuest */
#define P_NAK 0x01  /* Not interrupt AcKnowledge */
```

It is important to note that, even though the TTL_PAPERS hardware interrupt mechanism does not have to be used, it is necessary that any processor connected to TTL_PAPERS set the mode port so that the `P_RDY` bit, and not `P_INT`, is visible. This can be done by executing:

```
P_MODE(P_NAK);
```

As part of the TTL_PAPERS initialization code.

4.3. Barrier Interface

Logically, each barrier synchronization consists of two operations, signaling that we are at a barrier and waiting to be signaled that the barrier synchronization has completed. Although the TTL_PAPERS library generally combines these operations, here we discuss them as two separate chunks of code. The code for signaling that we are at a barrier is simply:

```
P_OUT(last_out ˆ= P_S);
```

This code just flips the sense of both strobe bits. Because `last_out` is initialized to have only one of the strobe bits high, this has the effect of alternating between `P_S0` and `P_S1`. Nothing else is changed, including the output data bits.

The code that actually waits for the barrier synchronization to complete involves a fair bit of logic, even though typically only a few instructions are executed. The reason that there is so much code has to do with three features of the TTL_PAPERS interface. The first complication is that the ready signal toggles, rather than always going to the same value. The second complication is that the status LEDs are software controlled. The third complication is that we want to check for an interrupt whenever a barrier is excessively delayed. The resulting code is something like:

```
/* Which condition am I waiting for? */
if (last_out & P_S0) {
    /* Waiting for P_RDY */
    if ((!(P_IN() & P_RDY)) && (!(P_IN() & P_RDY))) {
        /* Polled twice, make LED red */
        P_OUT(last_out ˆ= (P_LG | P_LR));
        /* Continue waiting */
        while (!(P_IN() & P_RDY)) CHECKINT;
        /* Ok, LED green again */
        P_OUT(last_out ˆ= (P_LG | P_LR));
    }
  } else {
    /* Waiting for not P_RDY */
    if ((P_IN() & P_RDY) && (P_IN() & P_RDY)) {
        /* Polled twice, make LED red */
        P_OUT(last_out ˆ= (P_LG | P_LR));
        /* Continue waiting */
        while (P_IN() & P_RDY) CHECKINT;
        /* Ok, LED green again */
        P_OUT(last_out ^= (P_LG | P_LR));
    }
}
```

In the initial version of the support library, `CHECKINT` is nothing —TTL_PAPERS interrupts are not used. However, we can easily check for an interrupt by defining `CHECKINT` as:

```
{
    /* Make P_INT status visible */
    P_MODE(P_SEL | P_NAK);
    /* Check for interrupt */
    if (P_IN() & P_INT) {
        /* Process the interrupt.... */
    } else {
        /* Restore P_RDY */
        P_MODE(P_NAK);
    }
}
```

4.4. NAND Data Communication

Although the TTL_PAPERS library provides a rich array of aggregate communication operations, all that the hardware really does is a simple 4-bit NAND as a side-effect of a barrier synchronization. However, this operation typically requires 5 port accesses rather than just the 2 port accesses used to implement a barrier synchronization without data communication. The extra port operations are required because:

- When a PE changes one or more of its output data bits, the TTL_PAPERS hardware NANDs in the new data, immediately changing the input data bits for all PEs. Thus, if one PE gets far enough ahead of another PE, it could change the data before the slower PE has been able to read the previous NAND result. If interrupts are disabled and an initial ordinary barrier synchronization is performed, then a block of data can be transmitted safely using static timing analysis alone to ensure that this race condition does not occur. Unfortunately, it isn't practical to disable interrupts under UNIX, so the safe solution is to follow each data transmitting barrier with an ordinary barrier that ensures all PEs have read the data before any PE can change the data. This adds 2 port operations.
- The TTL_PAPERS hardware is fed both data and strobe signals on the same port. Thus, data and strobe signals change simultaneously. As they race through the cables to the TTL_PAPERS logic and back out the cables, the TTL_PAPERS logic should give the NAND data at least a 20 nanosecond lead over the toggling of the `P_RDY` bit, but is 20 nanoseconds enough to ensure that the NAND data doesn't lose the race? Well, it depends on the implementation of the PC port hardware and the electrical properties of the cables...

which is another way of saying that the race isn't acceptable. An additional port operation is used to ensure that the data wins the race.

There are two possible ways to insert the extra operation. One way is to double the output operation, so that we first change the data and then toggle the strobe. This has the advantage that only PEs that are actually changing their data would need to insert the extra operation. If only early PEs change their data, we don't see any delay; however, the asymmetry of the PE operations can actually result in more delay than if the extra operation was always inserted. The other alternative is to resample the NAND data value after `P_RDY` has signaled it is present. This is somewhat more reliable than the doubling of the output operation, but the delay is always present for any PE that reads the data.

In any case, the result is that a barrier synchronization accompanied by an aggregate communication will take 5 port operations. For example, to perform a reliable NAND with our contribution being the 4-bit value *x*, we could first:

```
last_out = ((last_out & 0xf0) | x);
```

This sets the new data bits so that a barrier synchronization will transmit them along with the strobe. The next step is to perform a barrier synchronization, precisely as described in section 4.3. Having completed the barrier, we know that the NAND data should now be valid, so we resample it:

```
nand_result = P_IN();
```

Finally, a second barrier synchronization is performed to ensure that no PE changes its output data until after everyone has read the current NAND data.

Notice that *nand_result* is actually an 8-bit value with the NAND data embedded within it; thus, to extract the 4-bit result we simply use:

```
((nand_result >> 3) & 0x0f)
```

All the TTL_PAPERS library communication operations are built upon this communication mechanism. For example, ANY, ALL, and voting operations require just one such operation, or 5 port accesses. Larger operations, such as an 8-bit global OR, 8-bit global AND, or broadcast require 2 of the above transmission sequences, or 10 port accesses.

5. Conclusion

In this paper, we have presented the complete public domain design of the TTL_PAPERS hardware. This design represents the simplest possible mechanism to efficiently support barrier synchronization, aggregate communication, and group interrupt capabilities —using unmodified conventional workstations or personal computers as the processing elements of a fine-grain parallel machine. We do not view TTL_PAPERS as the ultimate mechanism, but rather as an introductory step toward the more general and higher performance barrier and aggregate communication engines that have been at the core of our research since 1987. The key thing to remember about TTL_PAPERS is not what it is, but rather *why* it is.

Why is TTL_PAPERS so much lower latency than other networks? Because it doesn't have a layered hardware and software interface. *Why* is the hardware so simple? Because it isn't a network; the fact that TTL_PAPERS communications are a side-effect of barrier synchronization eliminates the need for buffering, routing, arbitration, etc. *Why* is it useful? Because, although shared memory and message passing hardware is very common, the most popular high-level language and compiler models for parallelism are all based on aggregate operations —exactly what TTL_PAPERS provides. Further, barrier synchronization is the key to efficiently implementing MIMD, SIMD, and VLIW mixed-mode execution. *Why* didn't somebody do it earlier? We and others did. The problem is that tightly coupled design of hardware and compiler is not the standard way to build systems, so earlier designs (e.g., PASM, TMC CM-5) tended to use too much hardware and interface software, cost too much, and perform too poorly. *Why* is it public domain? Two reasons: (1) Purdue University didn't want to patent the basic mechanism back in 1987 and (2) we are primarily interested in the related compiler technology, and having more appropriate hardware makes our compiler work more valuable (i.e., more publishable).

Higher-performance versions of PAPERS are on the way. In fact, this TTL_PAPERS design is the sixth PAPERS design we have built and tested, and three of the other designs easily outperform it... but they use much more hardware. We are currently working on a smart parallel port card for the ISA bus that will roughly quadruple the performance of TTL_PAPERS without any change to its hardware. We are also pursuing versions of the higher-performance designs based on Texas Instruments FPGAs, and anticipate a PCI interface to future PAPERS units. Also watch for releases of various compilers targeting PAPERS. The email server at `papers@ecn.purdue.edu` will be the primary place for announcing and distributing future releases of both hardware and software.

References

[CoD94] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, *Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part I: Barrier Architecture,* Purdue University School of Electrical Engineering, Technical Report TR-EE 94-9, March 1994.

[CoD94a] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, *Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part II: Mode Emulation,* Purdue University School of Electrical Engineering, Technical Report TR-EE 94-10, March 1994.

[CoD94b] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors," *Proc. of 1994 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 93-96, August 1994.

[DiC94] H. G. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox, "Compiler Techniques For Fine-Grain Execution On Workstation Clusters Using PAPERS," *7th Annual Workshop on Languages and Compilers for Parallel Computing* (also to appear as a book chapter from Springer Verlag), Cornell University, August 1994.

[DiM94] H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox, *PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization*, Purdue University School of Electrical Engineering, Technical Report TR-EE 94-11, March 1994.

[DiO92] H. G. Dietz, M.T. O'Keefe, and A. Zaafrani, "Static Scheduling for Barrier MIMD Architectures," *The Journal of Supercomputing,* vol. 5, pp. 263-289, 1992.

[Jor78] H. F. Jordon, "A Special Purpose Architecture for Finite Element Analysis," *Proc. Int'l Conf. on Parallel Processing,* pp. 263-266, 1978.

[OKD90] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: static barrier MIMD (SBM)," *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 35-42, August 1990.

[OKD90a] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: static barrier MIMD (DBM)," *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 43-46, August 1990.

[SiN87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy," *Proc. of Second Int'l Conf. on Supercomputing,* pp. I 418-427, 1987.

Table of Contents