At about the same time, the 8086 asserts ALE high to enable the external address latches. It then sends out $\overline{BHE}$ and the desired address on the AD0–A19 lines. When the 8086 pulls the ALE line low, the address information is latched on the outputs of the external latches. After the 8086 is through using the AD0–AD15 lines for an address, it removes the address from these lines and puts the lines in the input mode (floats them). The 8086 then asserts its $\overline{RD}$ signal low. The $\overline{RD}$ signal going low turns on the addressed memory or port, which then outputs the desired data on the data bus. To complete the cycle the 8086 brings the $\overline{RD}$ line high again. This causes the addressed memory or port to float its outputs on the data bus. If the 8086 READY input is made low before or during T_2 of a machine cycle, the 8086 will insert WAIT states as long as the READY input is low. When READY is made high, the 8086 will continue with T_4 of the machine cycle. WAIT states can be used to give slow devices additional time to put out valid data. If a system is large enough to need data bus buffers, then the 8086 DT/$\overline{R}$ signal connected to these buffers will set them for input during a read operation or set them for output during a write operation. The 8086 $\overline{DEN}$ signal will enable the buffers at the appropriate time in the machine cycle.

8086 Bus Activities During a Write Machine Cycle

Now that we have analyzed the 8086 bus activities for a read machine cycle, let's take a look at the timing diagram for a write machine cycle in the right-hand side of Figure 7-1*b*. Most of this diagram should look very familiar to you because it is very similar to that for a read cycle.

During T_1 of a write machine cycle the 8086 asserts M/$\overline{IO}$ low if the write is going to be to a port, and it asserts M/$\overline{IO}$ high if the write is going to be to memory. At about the same time, the 8086 raises ALE high to enable the address latches. The 8086 then outputs $\overline{BHE}$ and the address that it will be writing to on AD0–A19. Incidentally, when writing to a port, lines A16–A19 will always be low, because the 8086 only sends out 16-bit port addresses. After the address has had time to pass through the latches, the 8086 brings ALE low again to latch the address on the outputs of the latches. Besides holding the address, these latches also function as buffers for the address lines. After the address information is latched, the 8086 removes the address information from AD0–AD15 and outputs the desired data on the data bus. It then asserts its $\overline{WR}$ signal low. The $\overline{WR}$ signal is used to turn on the memory or port that the data is to be written to. After the addressed memory or port has had time to accept the data from the data bus, the 8086 raises the $\overline{WR}$ signal line high again and floats the data bus.

If the memory or port device cannot accept the data word within a normal machine cycle, external hardware can be set up to pulse the READY input low each time that memory or a port device is addressed. If the READY input is pulsed low before or during T_2 of the machine cycle, the 8086 will insert a WAIT state after state T_3. Remember that during WAIT states the signals on the data bus, address bus, and control bus are held constant, so the addressed device has one or more extra clock cycles to accept the data from the data bus. If the READY input is made high before the end of the WAIT state, the 8086 will go on with state T_4 as soon as it finishes the WAIT state. If the READY input is still low just before the end of the WAIT state, the 8086 will insert another WAIT state. It will continue to insert WAIT states until READY is made high. The point here is that the 8086 can be forced to insert as many WAIT states as are necessary for the addressed device to accept the data.

If the system is large enough to need buffers on the data bus, then DT/$\overline{R}$ will be connected to the direction input on the buffers. During a write cycle, the 8086 asserts DT/$\overline{R}$ high to put the buffers in the transmit mode. When the 8086 asserts $\overline{DEN}$ low to enable the buffers, data output from the 8086 will pass through the buffers to the addressed port or memory location.

Work your way across the timing diagrams for the read and write machine cycles in Figure 7-1*b* until you feel that you understand the sequence of activities that occurs.

A Closer Look at the 8086

Figure 7-2, p. 168, shows a pin diagram for the 8086. You don't need to learn the detailed functions of all these pins. The main reason for showing you this is so that if you want to look at some of the 8086 signals with a scope or logic analyzer, you know which pins to connect to. We also want to make a few comments about some of the pins to give you a clearer idea of how an 8086-based microcomputer functions.

> NOTE: For reference, part of an 8086-data sheet showing all the pin descriptions is shown in Appendix A.

First, in Figure 7-2, find V_{CC} on pin 40 and ground on pins 1 and 20. Next, find the clock input, labeled CLK, on pin 19. As we showed you in the preceding sections, an 8086 requires a clock signal from some external clock generator to synchronize internal operations in the processor. Different versions of the 8086 have maximum clock frequencies ranging from 5 MHz to 10 MHz.

Now look for the address/data bus lines, AD0–AD15. Remember from the previous section that the 8086 has a 20-bit address bus and a 16-bit data bus and that the lower 16 address lines are multiplexed out on the data bus to minimize the number of pins needed. The 8086 sends out a signal called Address Latch Enable, or ALE, on pin 25 to strobe the external address latches. The upper 4 bits of the 20-bit address are sent out on the lines labeled A16/S3 through A19/S6. The double mnemonic on these pins shows that address bits A16 through A19 are sent out on these lines during the first part of a machine cycle, and status information, which identifies the type of operation being done in that cycle, is sent out on these lines during a later part of the cycle.

Having found the address bus and the data bus, now

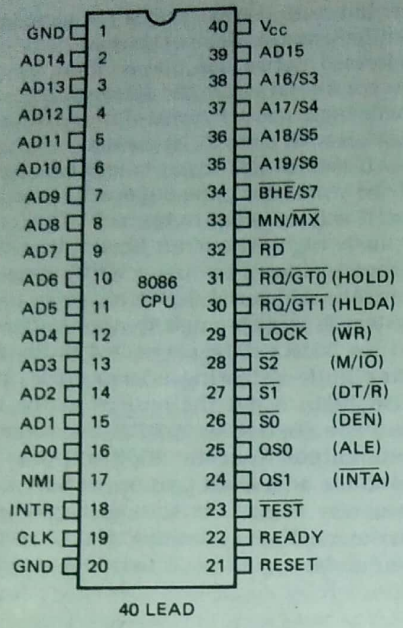


FIGURE 7-2 8086 pin diagram. (*Intel Corporation*)

look for the control bus signal pins. Pin 32 of the 8086 in Figure 7-2 is labeled $\overline{RD}$. This signal will be asserted low when the 8086 is reading data from memory or from a port. Pin 29 has the labels $\overline{WR}$ and $\overline{LOCK}$ next to it because it has two functions. The function of this pin and the functions of the other pins between 24 and 31 depend on the *mode* in which the 8086 is operating.

The operating mode of the 8086 is determined by the logic level applied to the MN/$\overline{MX}$ input, pin 33. If pin 33 is asserted high, then the 8086 will function in *minimum mode*, and pins 24 through 31 will have the functions shown in parentheses next to the pins in Figure 7-2. In minimum mode, for example, pin 29 will function as $\overline{WR}$, which will go low any time the 8086 writes to a port or to a memory location. The $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ signals form the heart of the control bus for a minimum-mode 8086 system. The 8086 is operated in minimum mode in systems such as the SDK-86 where it is the only microprocessor on the system buses.

If the MN/$\overline{MX}$ pin is asserted low, then the 8086 is in *maximum mode*. In this mode, pins 24 through 31 will have the functions described by the mnemonics next to the pins in Figure 7-2. In this mode, the control bus signals ($\overline{S0}$, $\overline{S1}$, and $\overline{S2}$) are sent out in encoded form on pins 26, 27, and 28. An external bus controller device decodes these signals to produce the control bus signals required for a system which has two or more microprocessors sharing the same buses. In Chapter 11 we discuss 8086 maximum-mode operation and show its use in multiple-microprocessor systems.

Another important pin on the 8086 is pin 21, the RESET input. If this input is asserted and then released, the 8086 will, no matter what it was doing, fetch its next instruction from physical address FFFF0H. At this address, then, you put the first instruction you want the microcomputer to execute after a reset or when the power is first turned on.

Finally, notice that the 8086 has two interrupt inputs, the *nonmaskable interrupt* (NMI) input on pin 17 and the *Interrupt* (INTR) input on pin 18. A signal can be applied to one of these inputs to cause the 8086 to stop executing its current program and go execute an interrupt procedure which takes care of the condition that caused the interrupt. You might, for example, connect a temperature sensor from a steam boiler to an interrupt input on an 8086. If the boiler gets too hot, then the temperature sensor will assert the interrupt input. This will cause the 8086 to stop executing its current program and go execute an interrupt-service procedure, which turns off the fuel supply to the boiler. A return instruction at the end of the interrupt-service procedure sends execution back to the interrupted program. In the next chapter we discuss interrupts further and show you how to write interrupt-service procedures.

Now we show you how to use a logic analyzer to observe and make measurements on microprocessor bus signals such as those we discussed in the preceding section.

USING A LOGIC ANALYZER TO OBSERVE MICROPROCESSOR BUS SIGNALS

Introduction

It is difficult to observe microprocessor bus signals with a standard scope because you can only look at two signal lines at a time. A logic analyzer such as the Tektronix 1230 shown in Figure 7-3 allows you to observe and make measurements on 16 to 64 signal lines at once. The least expensive version of the 1230 allows you to look at up to 16 signals at once, but expansion boards can be added to increase the number of input signal lines to 32, 48, or 64. Hewlett-Packard, Gould, and several other companies make comparable stand-alone logic analyzers.

Personal computers can be adapted to function as logic analyzers by installing plug-in units such as the

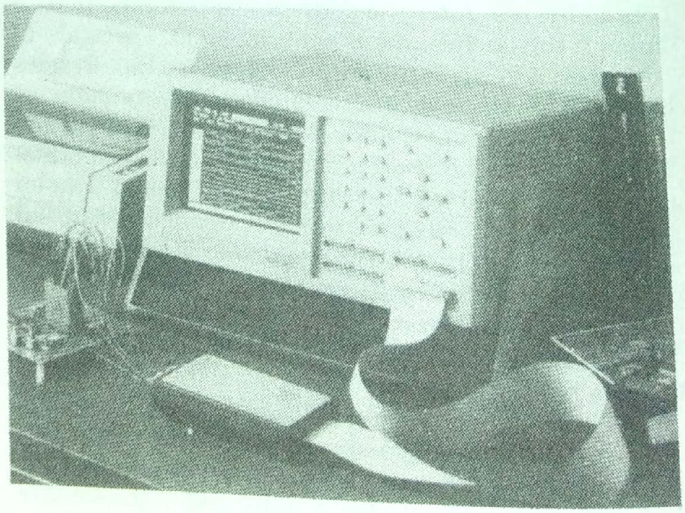


FIGURE 7-3 Tektronix 1230 logic analyzer. (*Courtesy of Tektronix Inc.*)

MicroCase Inc. μ Analyst 2000 or the Bitwise Designs, Inc. Logic 20.

One method of connecting signal lines to the analyzer inputs is with a *pod* and test clips such as those shown in front of the analyzer in Figure 7-3. Another method commonly used with microcomputers is a special cable with a plug which is inserted in the microprocessor socket on the circuit board. The microprocessor is plugged into a socket on top of the plug.

Before we describe how to make measurements with a logic analyzer, we will review the basic operation of a logic analyzer.

Review of Logic Analyzer Operation

Figure 7-4 shows a functional block diagram of a simple logic analyzer. Since logic analyzers are used to detect and display only 1's and 0's, a comparator is put on each input. The reference input of the comparator is set for the logic threshold of the devices in the system you are looking at. If you are looking at TTL or CMOS signals, for example, you set the threshold to 1.4 V. The comparators then make sure that the signals to the rest of the analyzer circuitry are clear-cut 1's or 0's.

The analyzer takes a "snapshot" of the logic levels on the data inputs each time it receives a *clock* pulse. The samples are stored in an internal RAM. Different analyzers store between 256 and 1024 samples for each input channel.

As shown by the block diagram in Figure 7-4, the analyzer can be clocked by an internally produced signal or some external signal. If you are using an analyzer to look at 8086 address and data lines, for example, you could use ALE as a clock signal. The analyzer will then take a sample each time the 8086 puts out an address and pulses ALE. The samples stored in the analyzer memory will then represent a sequence of addresses output by the 8086. As another example, you could clock the analyzer on the $\overline{RD}$ signal from an 8086. With this clock signal the analyzer will take a sample each time the 8086 does a read operation, so the samples stored in the analyzer memory will represent the sequence of data words read in from memory or from ports.

To make precise timing measurements with an analyzer, you use a clock signal from an internal, crystal-controlled oscillator. In this case the analyzer will take a sample each time a pulse from the internal clock oscillator occurs. If, for example, you choose an internal clock frequency of 50 MHz, the analyzer will take a sample every 20 ns. You can then determine the time between two events by counting the number of samples and multiplying the number by 20 ns.

If the analyzer is receiving either an internal or an external clock, it will be continuously taking samples of the input data and storing these samples in the internal RAM. A *trigger* signal tells the analyzer when to stop taking samples and display the samples stored in the RAM. As shown by the block diagram in Figure 7-4, you can use some external signal to trigger the analyzer, or you can use a *word recognizer* in the analyzer to produce a trigger signal. A word recognizer compares the binary word on the input signal lines with a word you set with switches or a keyboard. When the two words match, the word recognizer sends out a trigger signal.

Since the analyzer is continuously taking samples, you can set the analyzer for a *pretrigger* display, a *center trigger* display, or a *posttrigger* display. For an analyzer that displays 256 samples, pretrigger means that the display will show the 256 samples that were taken just before the trigger occurred. For center trigger mode, 128 samples taken before the trigger and 128 samples taken after the trigger will be displayed. Posttrigger mode

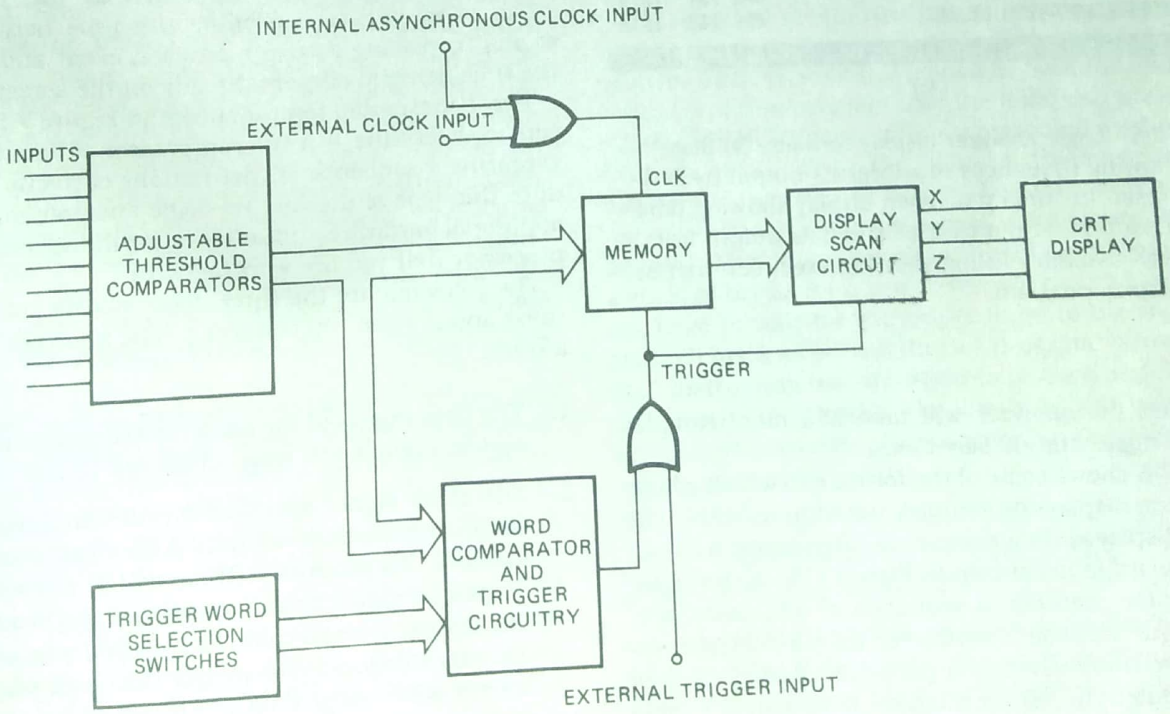
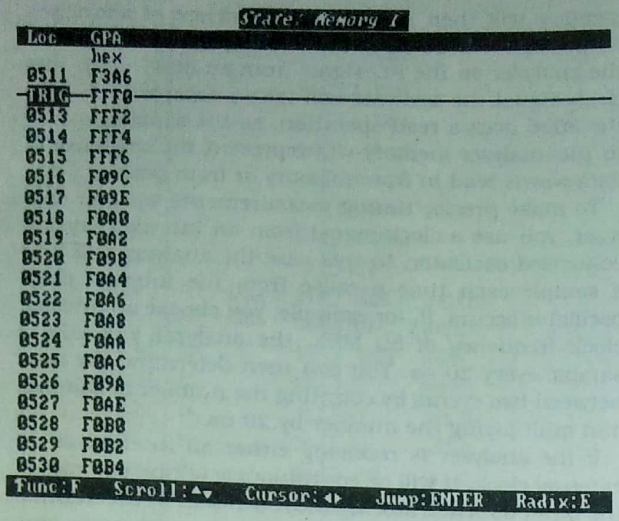
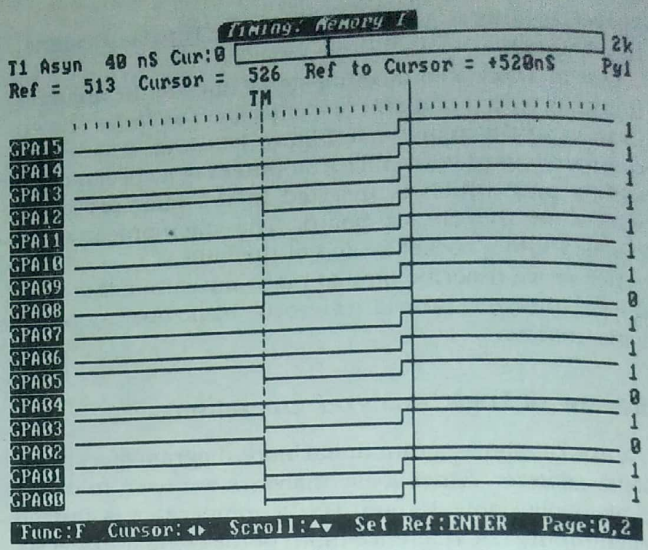


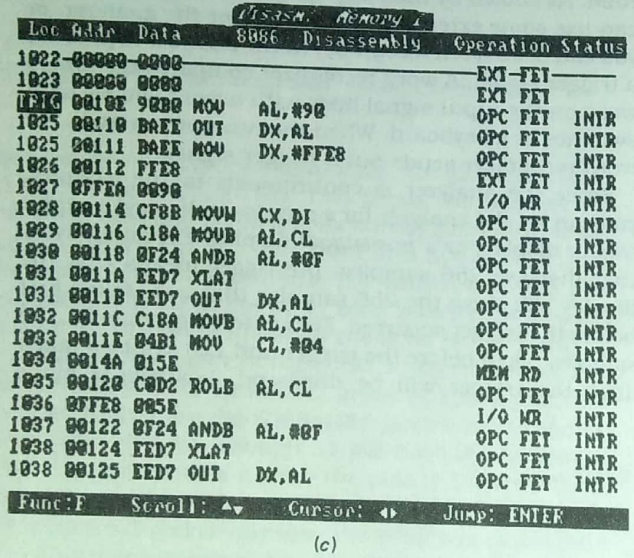
FIGURE 7-4 Block diagram of simple logic analyzer.



(a)



(b)



(c)

FIGURE 7-5 Logic analyzer display formats. (*a*) State listing showing sequences of addresses output by SDK-86 after reset. (*b*) Timing diagram display showing time between address output by 8086 and data output from RAM. (*c*) Disassembly listing showing execution part of SDK-86 display program.

means that the analyzer will take 256 more samples after the trigger and display them.

Figure 7-5 shows some of the formats in which a logic analyzer can display the samples stored in its RAM. The series of displayed data samples is often called a *trace*.

The *state table list* shown in Figure 7-5*a* is useful for observing, for example, a sequence of addresses sent out or a sequence of data words read in by a microprocessor. To determine whether a particular address line is shorted, you might tell the analyzer to display the table in binary so you can see the individual 1's and 0's.

However, a hexadecimal listing such as that in Figure 7-5*a* makes it easier to recognize if a microcomputer is putting out addresses in the right sequence. Some analyzers, such as the Tektronix 1230, allow you to take a series of samples from a functioning system, store these samples in a second memory in the analyzer, and then compare them with a series of samples taken from a nonfunctioning system. We have found this feature quite helpful in troubleshooting malfunctioning instruments which have poor documentation.

The *timing diagram format* shown in Figure 7-5*b* is most useful when making time measurements with an internal clock. As we mentioned before, you can measure times by simply counting the number of clock pulses between two events and multiplying by the time per clock pulse. Some analyzers, such as the Tektronix 1230, allow you to determine the time between two events by placing a cursor on each event and reading the time between cursors directly on the screen.

The *disassembly format* shown in Figure 7-5*c* allows you to determine if a microprocessor is fetching and executing a sequence of instructions correctly. To produce this type of display, the logic analyzer must have additional hardware and software for the specific microprocessor that you are working with.

The following are the three major points you have to think about when you connect a logic analyzer up to do a trace:

1. The data inputs of the analyzer are connected to the system signals you want displayed in the trace.
2. The clock signal specified for the analyzer tells it when to take data samples and store them in its memory. To produce a trace which shows the sequence of states that a system steps through, you usually use an external clock. When you are using an external clock, you specify the clock edge which occurs when valid data is on the data inputs. For making timing measurements you usually use the crystal-controlled internal clock.

3. The trigger specified for the analyzer tells it when to stop taking samples and display the set of samples stored in its memory. Usually you will use the internal word recognizer to trigger the analyzer when a specified word is present on the data inputs.

Now that you have an overview of logic analyzer operation, here are some specific examples of how you observe 8086 bus signals and timing. Exercises in the lab manual which goes with this book give still more detailed examples.

MAKING A TRACE OF A SEQUENCE OF ADDRESSES

The first step in using a logic analyzer to look at microcomputer signals is to decide what specific signals you want to look at and connect the analyzer data inputs to those signals. If you want to do a trace which shows the sequence of addresses that the 8086 outputs as it executes a test program, you connect the data inputs of the analyzer pod to the 8086 AD0–AD15 pins.

The next step is to decide what signal to clock the analyzer on. To make this decision, you look carefully at the 8086 timing waveforms in Figure 7-1*b* to find a signal edge which occurs when valid addresses are on the AD0–AD15 lines. One possible signal to use for clocking the analyzer is the 8086 CLK signal shown at the top of the waveforms in Figure 7-1*b*. This signal has a falling edge when the address is valid on AD0–AD15, but it also has falling edges when the lines are floating and when the data from or to memory is on the lines. In other words, if the analyzer is clocked on this signal, the trace will show a mixture of data, addresses, and garbage, which you have to sort out.

A better choice for an analyzer clock signal is the 8086 ALE signal, because this signal is present only when addresses are on the AD0–AD15 lines. To use ALE as a clock signal, connect the External Clock input of the analyzer to the 8086 ALE pin. To determine which edge of the ALE signal to clock the analyzer on, look closely at the 8086 timing waveforms in Figure 7-1*b*. At the time when the positive edge of the ALE signal occurs, the 8086 has not yet output the address, so clocking the analyzer on this edge will not grab the addresses. The falling edge of the ALE signal occurs when the address is solidly settled on the AD0–AD15 lines, so you should set the analyzer to clock on the falling edge of ALE.

The final step is to determine what to trigger the analyzer on. Since you want to make a trace of a sequence of addresses, the logical choice here is to choose an internal trigger and set the internal word recognizer to produce a trigger when the first program address is present on the data inputs. For example, if the first program instruction is in memory at 00100H, you would set the analyzer to trigger when this address is present. When you specify the trigger position, set the analyzer for "begin" so that the trace listing starts with the specified address. The example logic analyzer trace in Figure 7-5*a* shows this type of display.

MAKING A TRACE OF A SEQUENCE OF DATA WORDS

As a second example of using an analyzer to look at microcomputer signals, suppose that you want to do a trace which shows the sequence of data words read in from memory as the 8086 executes a test program. For this trace you connect the analyzer data inputs to the 8086 AD0–AD15 pins, because the data comes in on these lines.

To determine what signal to clock the analyzer on and which edge of that signal to specify, you again look closely at the 8086 timing waveforms in Figure 7-1*b*. From these waveforms you should see that the 8086 $\overline{RD}$ signal is asserted during a Memory Read operation, so this is an appropriate signal to connect to the analyzer's External Clock input. The rising edge of the $\overline{RD}$ signal occurs when valid data is on the data bus, so set the analyzer to clock on a rising edge.

Since you want a trace of the data words read in from memory by the 8086, you need to look at the test program to determine what to trigger the analyzer on. For this example, assume the simple test program shown here is entered in memory and run.

```
00100 EB HERE:JMP HERE  ; Endless loop which does nothing
00101 FE
00102 90        NOP     ; Just more words to fetch
00103 90        NOP
00104 04        ADD AL,55H
```

Since the 8086 has a 16-bit data bus, it can read in a word (2 bytes) at a time if the word starts on an even address. When reading in the code bytes for this program then, the 8086 will send out address 00100H and assert both A0 and $\overline{BHE}$. The byte containing EBH will come into the 8086 on AD0–AD7, and the byte containing FEH will come into the 8086 on AD8–AD15. The first data word read in from memory then is FEEBH, so this is the word you set the analyzer to trigger on.

When the trace is completed, it will show the sequence of words FEEBH, 9090H, and 0455H over and over. The only part of this program that the 8086 executes is the HERE:JMP HERE instruction represented by the codes EBH and FEH. While the 8086 is decoding the JMP instruction, however, it fetches the codes for the following instructions and stores them in its queue, ready to be used. This is analogous to the way a helper sets up a stack of bricks for a bricklayer, so the bricklayer does not have to wait for the helper to go to the truck and get each brick as needed. In this program, however, the JMP instruction tells the 8086 to go back and fetch the JMP instruction again. The words 9090H and 0455H are fetched from memory and stored in the 8086 queue, but they are never used.

USING A CLOCK QUALIFIER IN LOGIC ANALYZER MEASUREMENTS

In the preceding example we showed you how to produce a trace of the data words read in from memory by an 8086. Now suppose that you are executing a program which reads data words from memory and data words

from ports. If you simply clock the analyzer on the rising edge of the $\overline{RD}$ signal as you did for the preceding example, the trace will contain both the data words read from memory and the data words read from ports. You can use the M/$\overline{IO}$ signal to produce a trace which contains only the words read from memory or only the words read from ports.

Remember from our previous discussions that when the 8086 writes a word to a memory location or reads a word from a memory location, it will assert its M/$\overline{IO}$ signal high. When the 8086 writes a word to a port or reads a word from a port, it will assert the M/$\overline{IO}$ signal low.

To produce a trace of only the data words read from ports, you connect the $\overline{RD}$ signal to the External Clock input of the analyzer and connect the M/$\overline{IO}$ signal to an input on the analyzer labeled *Clock Qualifier*. The principle here is that if this input is used, the analyzer will respond to a clock signal only if a specified level is present on that input. For this example, you want the analyzer to take a sample on the rising edge of the $\overline{RD}$ if M/$\overline{IO}$ is low. Therefore, you will specify a low as the active level for the clock qualifier input. Depending on the analyzer, the active level for the clock qualifier input may be set in a menu, by a switch on the pod, or by connecting the qualifier signal to a specific input on the data pod.

To produce a trace of only the data words read from memory, you can clock on the rising edge of $\overline{RD}$, connect the M/$\overline{IO}$ signal to the Clock Qualifier input, and specify a high for the clock qualifier. The point here is that by carefully choosing the clock signal and the qualifier signal, you can usually produce a trace of just the data you want.

MEASURING MEMORY ACCESS TIME WITH A LOGIC ANALYZER

As shown in the 8086 timing waveforms in Figure 7-1*b*, one type of memory access time is the time it takes for a memory device to produce valid data on its outputs after an address is applied to its address inputs. With a little thought you can use a logic analyzer to measure the actual memory access time in a system.

The first step in this measurement is to enter and run a test program which reads from the desired memory device over and over. For this example, we will use the same program we used in the preceding example. To make it easy to refer to, we repeat it here.

```
00100 EB HERE:JMP HERE  ; Endless loop which does nothing
00101 FE
00102 90        NOP             ; Just more words to fetch
00103 90        NOP
00104 04        ADD AL,55H
```

The next step is to think about what signals to connect to the analyzer data inputs. To determine the time between a valid address from the 8086 and valid data from the memory device, you obviously need to look at the address/data lines. The number that you can trace and display depends on the particular analyzer you are using. The basic Tektronix 1230 analyzer will sample and display 16 channels in the timing mode which you use for this type of measurement. If you have an analyzer such as this, you can connect the analyzer data inputs to the AD0–AD15 pins on the 8086. The upper four address lines, A16–A19, do not change during the execution of this example program, so you don't need to look at them.

When making timing measurements with a logic analyzer, you almost always use the crystal-controlled internal clock to tell the analyzer to take samples so that you know the exact time between samples. For an SDK-86 board the memory access time for the RAM that contains the sample program will be around 100 ns. To get the best possible resolution for your timing measurement, then, you should set the analyzer clock period for the shortest time possible on your analyzer. The shortest period for the Tektronix 1230 with a 16-channel display is 40 ns per clock, so we will use this setting.

To choose the trigger word for this measurement, look again at the timing waveforms in Figure 7-1*b*. The address goes out on the data bus and later the data comes back in. Since the address is the first activity, you set the word recognizer in the analyzer to trigger on the first address that is sent out.

Once you do a trace, you can determine the memory access time by counting the number of sample points between the address of 0100H appearing on the bus and the data word of FEEBH appearing on the bus. Figure 7-5*b* shows an example of this type of display. If your analyzer has cursors, you can position one cursor at the time when the address becomes valid, position the other cursor at the time when the data becomes valid, and read the time difference between the two from the on-screen display.

Note that the resolution of this measurement is only 40 ns, because that is the time between samples. In other words, any changes that take place between sample points will not be shown in the display until the next set of samples is taken. On many analyzers you can specify a shorter sampling period if you reduce the number of signal lines being traced. With the Tektronix 1230, for example, you can use a sample clock with a period as short as 10 ns if you can get by with sampling only four signal lines. We usually start by doing a trace of, for example, all 16 lines, and then from the 16 we choose four which show the desired transitions. With just these four lines we can decrease the sample period to 10 ns and thereby increase the resolution of our measurement.

We obviously can't describe here all the ways to use a logic analyzer. If you have one, consult the manual for it to learn some of the finer points of its use. Also, the lab manual that is available for use with this book has some exercises to help you gain more skill with an analyzer. The point here was to show you how to use the analyzer as a "window" into what is going on in a system. By carefully choosing the signals you look at, the signal you clock on, and the word you trigger on, you can often solve difficult problems. For this reason, a logic analyzer is a valuable tool when developing a new microcomputer-based product.

Now that you know how to observe and make measurements on microcomputer bus signals, let's take a closer look at an 8086 system.

AN EXAMPLE MINIMUM-MODE SYSTEM, THE SDK-86

The previous sections showed how a clock generator, address latches, and data bus buffers are connected to an 8086 to form what we might call the minimum-mode CPU group. As shown in Figure 7-1*a*, this group of ICs generates the address bus, data bus, and control bus signals needed for an 8086 minimum-mode system. In this major section of the chapter we discuss how this CPU group is connected with ROM, RAM, ports, and other devices to form a system. The system we use for this discussion is the *Intel SDK-86 system design kit*, an 8086-based unit suitable for building the prototypes of small microcomputer-based instruments.

Figure 7-6 shows a photograph of an SDK-86 board. From the photograph you can see that, in addition to the microcomputer ICs, the board has a hexadecimal keypad, some 7-segment displays, and a large open area for adding more ROM, RAM, ports, or interface circuitry. A monitor program in ROM on the board allows you to enter, execute, and debug machine code programs using the onboard hex keypad or an external CRT terminal connected to the serial port on the board. The board comes with 2 Kbytes of RAM and sockets where you can add another 2 Kbytes. The board also has six 8-bit parallel ports which you can program to be inputs or outputs. To get a better idea of the hardware functions on the board and the devices used to implement these functions, let's look at the detailed block diagram of the SDK-86 in Figure 7-7, p. 174.

Whenever you are approaching a system that is new to you, it is a good idea to study the detailed block diagram of the system carefully before you start digging into the actual schematics. The schematics for even a small system such as this are often spread over many pages. Without the overview that the block diagram gives, it is very difficult for you to see how all the schematic pieces fit together.

The first parts to look at in Figure 7-7 are the 8086 CPU and the 8284 *clock generator*. Note that the 8284 has a 14.7456-MHz crystal connected to it. According to the data sheet for the 8284, the frequency of the crystal connected to the 8284 will be divided by 3 to produce the clock signal sent to the 8086. Therefore, the actual 8086 clock frequency for this board will be 4.915 MHz. Another clock signal called PCLK, which is also produced by the 8284, has a frequency of half the clock frequency, or in this case 2.45 MHz. This signal is used as a general-purpose clock signal throughout the system. The hardware RST signal and the RDY signal are also passed through the 8284 to synchronize them with the clock signal before they are sent to the

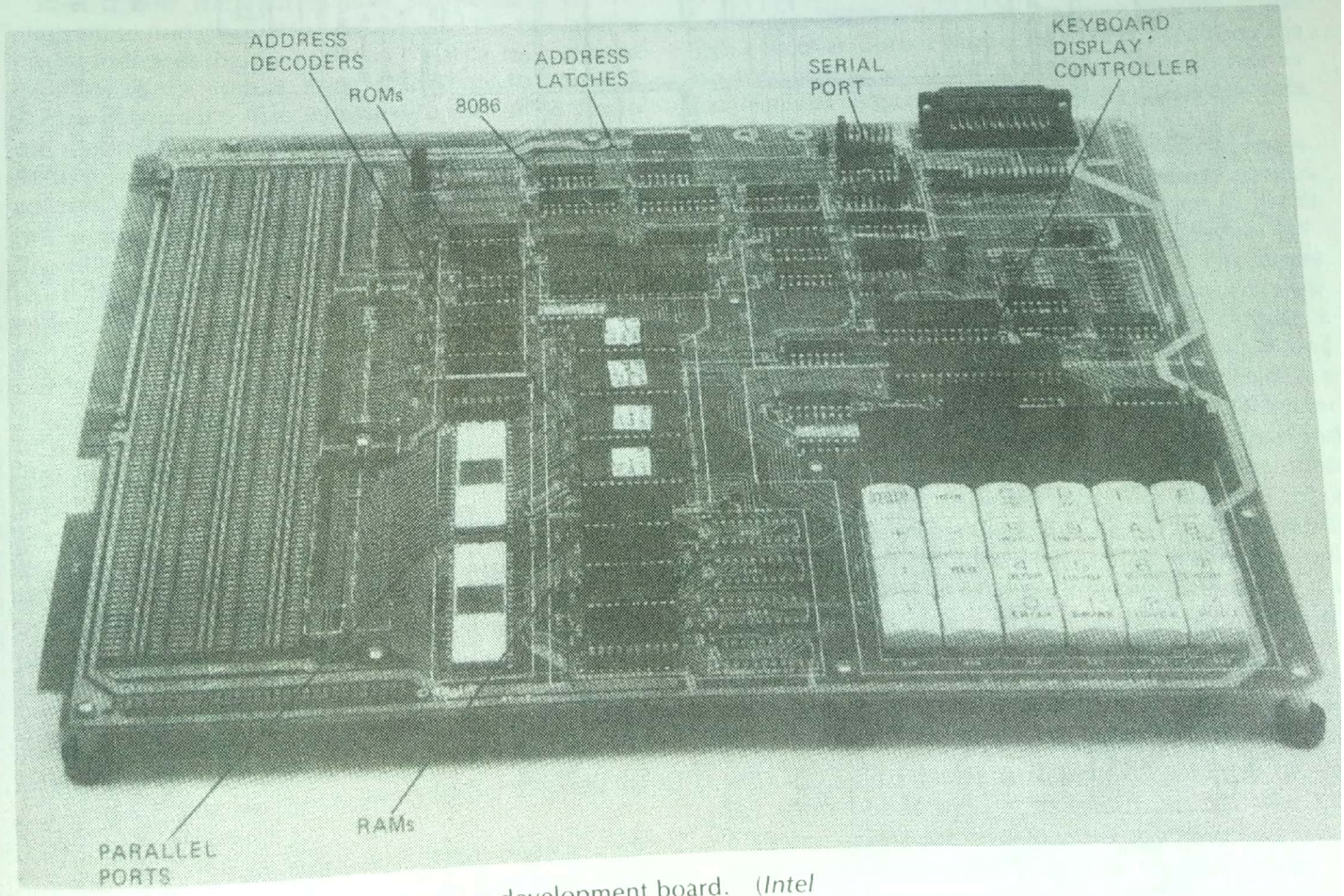


FIGURE 7-6 Intel SDK-86 microprocessor development board. (*Intel Corporation*)

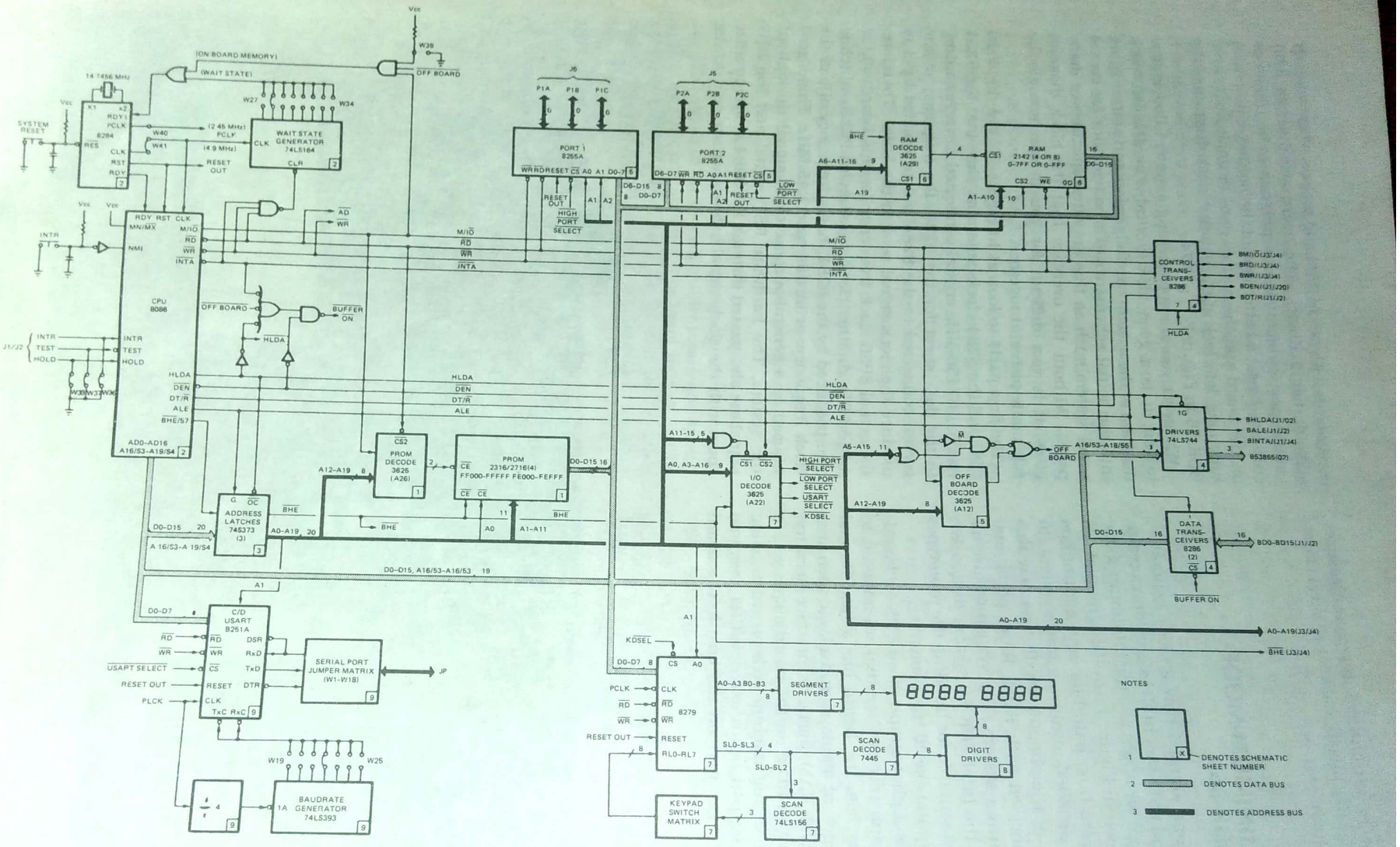


FIGURE 7-7 Detailed block diagram of SDK-86 board. The block diagram shown here and the schematics in Figure 7-8 are for the original Intel SDK-86 board. The SDK-86 board is now manufactured by University Research and Development Associates, Inc. (URDA), 4516 Henry Street, Suite 407, Pittsburgh, PA 15213. The URDA SDK-86 board is essentially the same as the original Intel board, except that it uses two 2732A EPROMs instead of 4 2716s, it uses 6264 static RAMs instead of 2142 static RAMs, and it uses discrete logic gates as address decoders instead of using 3625 PROMs. If you have an URDA board, consult the schematic for it to see the details of these sections. All functions and addresses on the URDA board are the same as those on the Intel board. (*Intel Corporation*)

8086. As you can see in Figure 7-7, considerable circuitry is connected to the RDY1 input so that several conditions can cause a WAIT state to be inserted in a machine cycle. The structure labeled W27 through W34 above the WAIT-state generator in Figure 7-7 represents wire-wrap pins which can be jumpered to specify the number of WAIT states desired in a machine cycle. We will discuss this in detail later.

By this time you may have noticed that the symbols for the 8284, 8086, and WAIT-state generator each have a small box containing a 2 in their lower right corner. This number tells you that the detailed schematic for these parts will be found on sheet number two of the set of schematics. Figure 7-8 on pages 176–184 shows the complete schematic set for the SDK-86 board, so you can check this out if you wish.

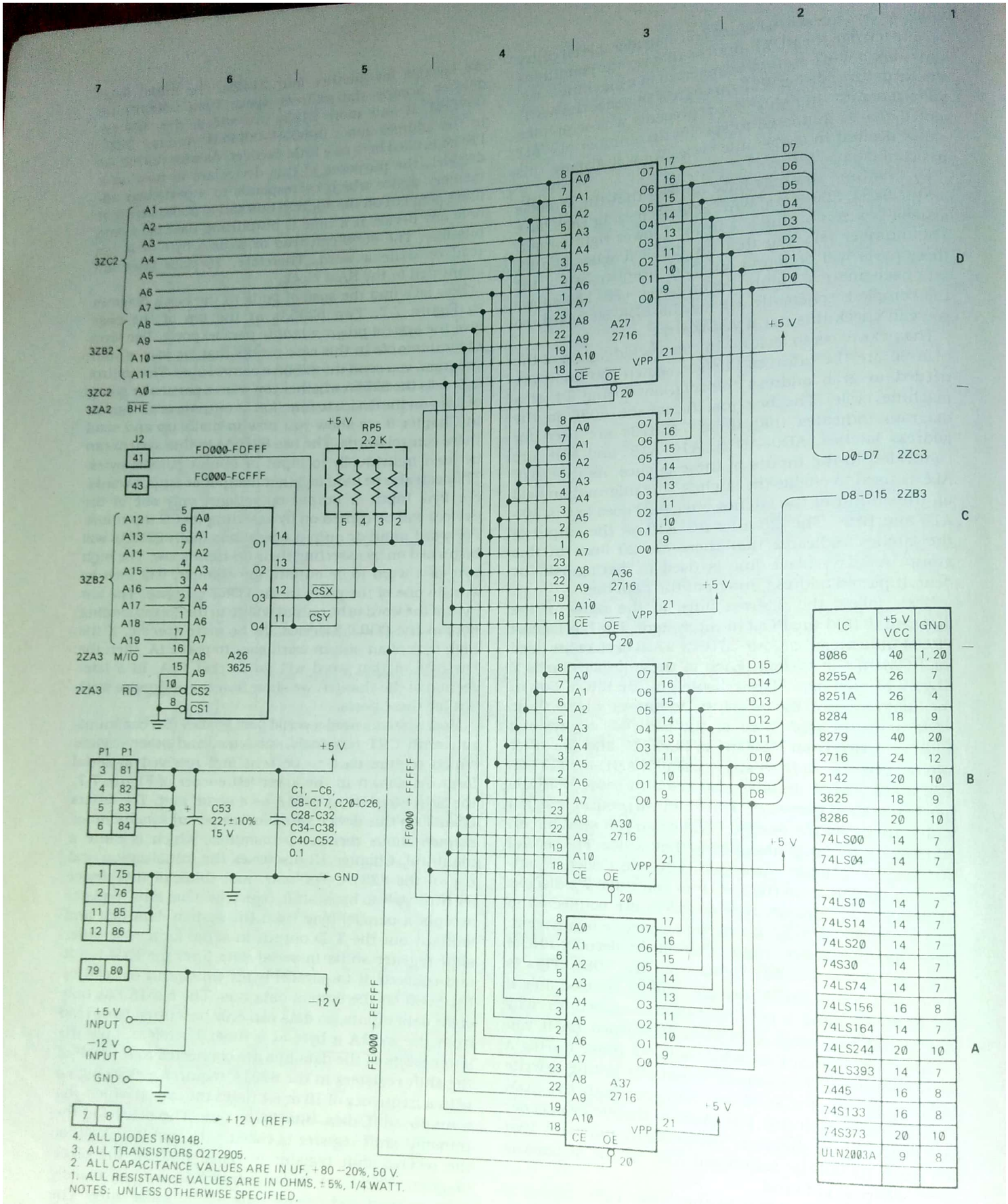
The next parts to look for in the block diagram of the SDK-86 are the *address latches*, which you know are needed to grab address information during T1 of a machine cycle. The box just below the 8086 in the diagram indicates that three 74S373s are used for address latches. AD0–AD15, A16–A19, and $\overline{BHE}$ are connected to the inputs of these latches. As expected, ALE is used to enable the latches. The information held on the output of the latches after ALE goes low is A0–A19 and $\overline{BHE}$. The /20 after A0–A19 on the output of the latches indicates that there are 20 lines in this group. A heavy black line is used to distinguish the demultiplexed address bus from the data bus.

Next, follow the address lines to the right on the diagram to find the ROM in the system. The box labeled PROM indicates that four 2316 or 2716 devices are used for ROM in the system. Each of these devices holds 2 Kbytes of memory. Also indicated in the PROM box in the diagram are the absolute addresses where these devices are located. Two of the EPROMs occupy the address space from FE000H to FEFFFH, and the other two occupy the address space from FF000H to FFFFFH. The 3625 PROM decoder connected to these EPROMs has two related purposes. The first is to produce a signal which turns on the desired ROM when you send out an address in the range assigned to that device. The second purpose is to make sure that only one device is outputting signals onto the data bus at a time. We discuss in detail later how address decoders are connected to give a desired address to a particular device in a system. Note that the enable input, $\overline{CS2}$, of the decoder PROM is connected to the $\overline{RD}$ signal from the 8086. This is done so that the PROM decoder will be enabled only if the 8086 is doing a read operation. Can you see why you would not want a ROM to be turned on if you accidentally sent out an address in its range during a write operation? The answer is that attempting to write to the outputs of a ROM can burn out both the ROM and the buffer outputs. The (A26) in the PROM decoder box of the block diagram, incidentally, indicates that the 3625 IC will be numbered A26 on the schematic sheet where it is found.

Follow the address bus to the upper right corner of the block diagram in Figure 7-7 to find how RAM is implemented in this system. The board comes with 2 Kbytes of static RAM contained in four 2142s, but there are sockets for another four 2142s. The initial four devices occupy the address space from 00000H to 007FFH. If four more 2142s are added, they will be in the address space 00800H–00FFFH. Another 3625 PROM is used here as a RAM decoder. As with the PROM decoder, the purposes of this device are to turn on a memory device which corresponds to a particular address sent out on the address bus and to make sure that only one device at a time is outputting data on a data bus line. The 8086 can read or write a byte or it can read or write a word. Therefore, 16 data lines are connected to the RAM block.

Now let's find the *system ports* in the block diagram in Figure 7-7. Two 8255As at the top of the page give the system *programmable* parallel ports. The term *programmable* in this case means that, as part of your program, you send the 8255A a *control byte*. The control byte tells the 8255A whether you want a particular group of lines on the device to function as outputs or as inputs. In Chapter 9 we show you how to make up and send these control words. The two 8255As in this system can be used individually to input or output parallel bytes. They can also be used together to input or output words. For byte input or output operations, only one of the devices will be turned on by asserting its $\overline{CS}$ input low. For word input or output operations, both 8255As will be turned on by asserting their $\overline{CS}$ inputs low. The high byte of a word to be output, for example, will then be sent to one of the ports in the PORT 1 device. The low byte of the word to be output will go to the corresponding port in the PORT 2 device. To be more specific, if the high byte of an output word goes to port P1A, then the low byte of that word will go to port P2A. In a later section of the chapter, we show how the addresses work out for these ports.

Most systems need a serial port so they can communicate with CRT terminals, modems, and other devices which require data to be sent and received in serial form. As shown in the lower left corner of Figure 7-7, the SDK-86 uses an 8251A as a *serial port*. The letters USART on this device stand for *universal synchronous/asynchronous receiver transmitter*, which is quite a mouthful. Chapter 13 discusses the initialization and use of the 8251A. For now, just think of this device as two back-to-back shift registers. One shift register accepts a parallel byte from the system data bus and shifts it out the TxD output in serial form. The other shift register shifts in serial data from the RxD input and converts it to parallel bytes which can be read by the 8086 on the system data bus. The 8251A has only eight data inputs, so data can only be written to or read from the 8251A a byte at a time. Therefore, only the lower 8 bits of the data bus are connected to it. Each of the shift registers in the 8251A requires a clock signal with a frequency of 16 or 64 times the rate at which you want to shift data bits in or out. The clock for the transmit shift register is called TxC, and the clock for the receive shift register is called RxC on the block diagram. These are tied together because you usually want to send and receive data at the same rates. The clock for these inputs is produced by dividing the 2.45-MHz PCLK signal from the 8284 clock generator. Wire-



IC	+5 V VCC	GND
8086	40	1, 20
8255A	26	7
8251A	26	4
8284	18	9
8279	40	20
2716	24	12
2142	20	10
3625	18	9
8286	20	10
74LS00	14	7
74LS04	14	7
74LS10	14	7
74LS14	14	7
74LS20	14	7
74S30	14	7
74LS74	14	7
74LS156	16	8
74LS164	14	7
74LS244	20	10
74LS393	14	7
7445	16	8
74S133	16	8
74S373	20	10
ULN2003A	9	8

4. ALL DIODES 1N914B.
3. ALL TRANSISTORS Q2T2905.
2. ALL CAPACITANCE VALUES ARE IN UF, +80 −20%, 50 V.
1. ALL RESISTANCE VALUES ARE IN OHMS, ± 5%, 1/4 WATT.
NOTES: UNLESS OTHERWISE SPECIFIED,

FIGURE 7-8 SDK-86 complete schematics; see also pages 177–184. Sheet 1 of 9. *(Intel Corporation)*

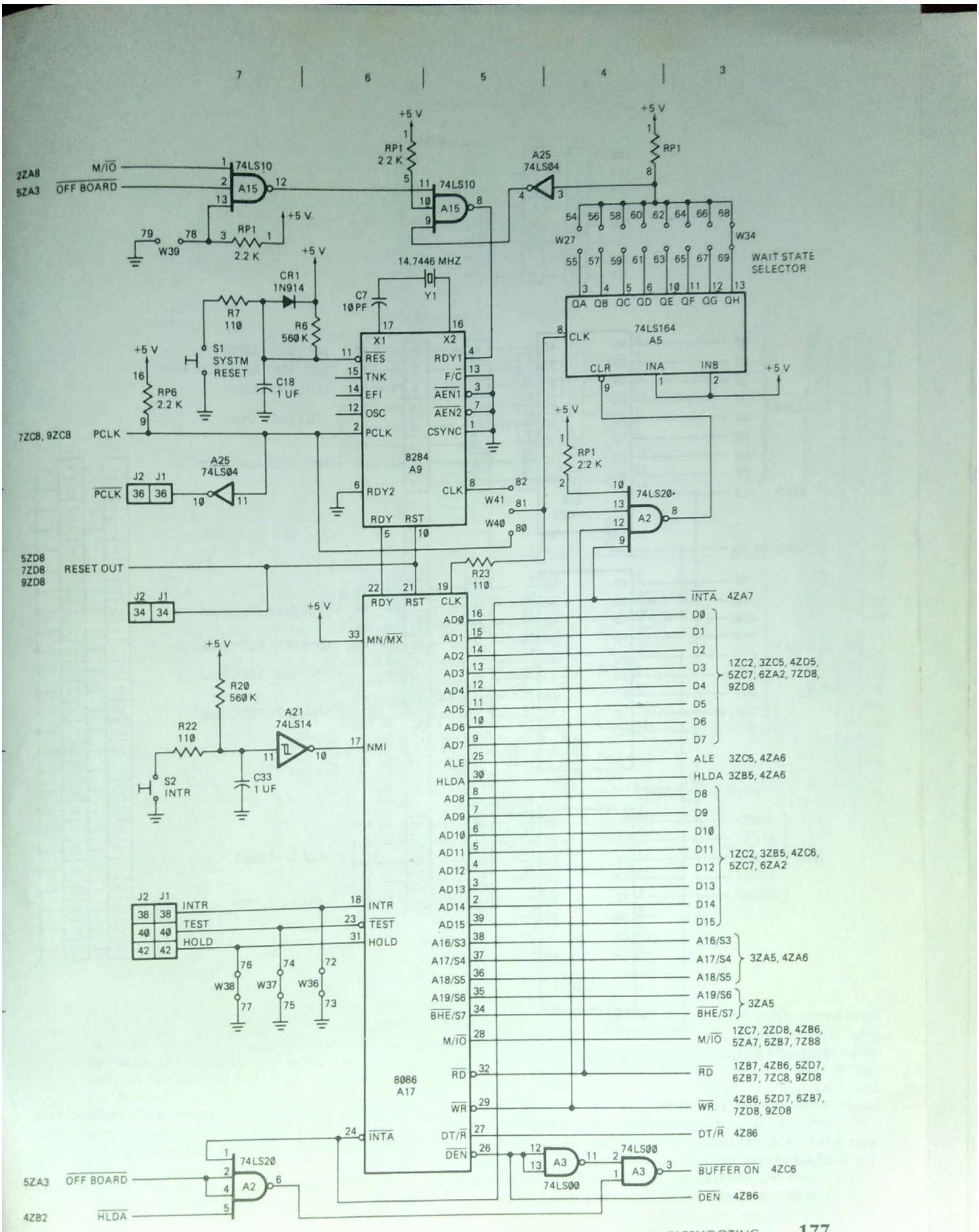


FIGURE 7-8 (*continued*) Sheet 2 of 9.

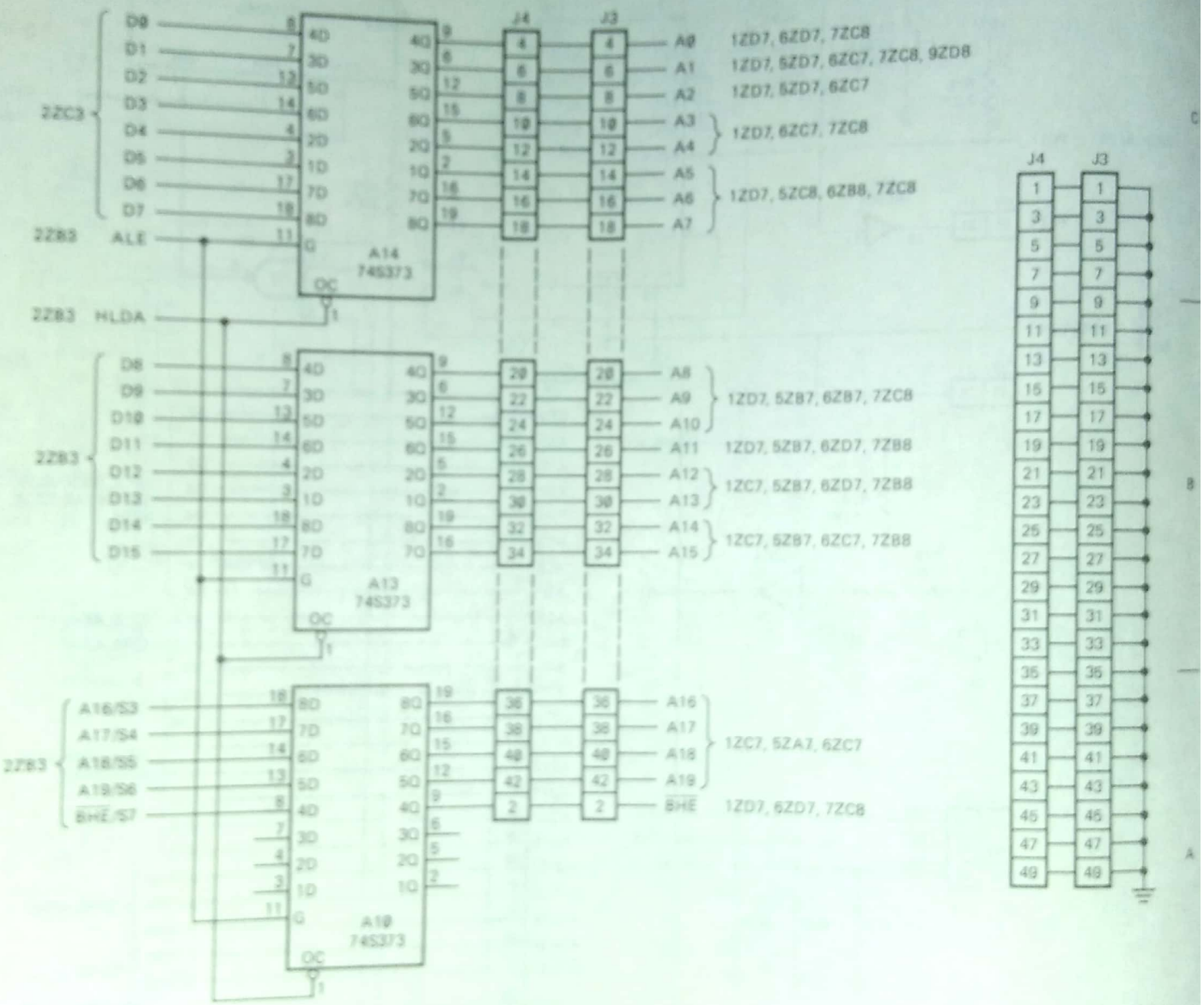


FIGURE 7-8 (continued) Sheet 3 of 9.

wrap jumper pins, W19–W25, allow you to select the desired TxC and RxC frequency from a divider chain in the 74LS393 *baud rate generator*. *Baud rate* is a way of specifying the rate at which data bits are shifted in or out of a serial device. Baud rate for a device such as the 8251A is defined as 1 over the time per bit. If the time per bit is 416 μs, for example, then the baud rate is 2400 baud. Common baud rates for serial data transmission are 300, 600, 1200, 2400, 9600, and 19,200.

The final port device to discuss here is the 8279 in the bottom center of the SDK-86 block diagram (Figure

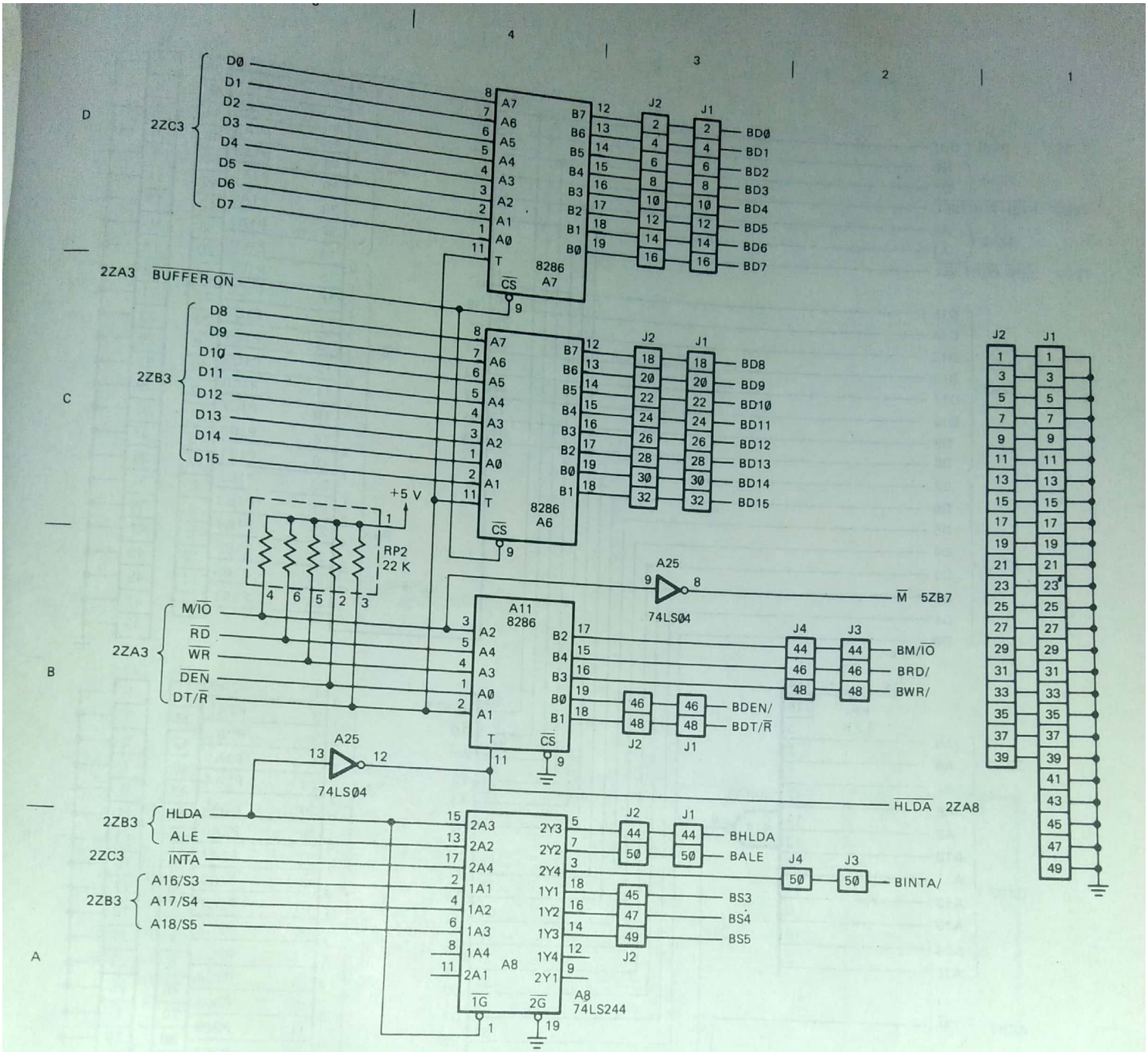


FIGURE 7-8 (*continued*) Sheet 4 of 9.

7-7). The 8279 is a *specialized input/output device* which has two major functions. The first function is to scan the hex keypad, detect when a key is pressed, debounce the signal from a pressed key, and store the code for the pressed key in an internal RAM, where it can be read by the 8086. The second major function of the 8279 is to refresh the multiplexed display on the eight 7-segment LED displays. Seven-segment codes for the digits to be displayed are sent to a RAM in the 8279. The 8279 then automatically sends out the code for one digit and turns on that digit. After a millisecond or so, the 8279 sends out the 7-segment code for the next digit and turns on that digit. The process is continued until all digits have been lit, and then the 8279 cycles back to the first digit again. In Chapter 9 we discuss in detail how you use an 8279. The main point for now is that this device takes care of scanning a keyboard and refreshing a display so that you don't have to do these operations as part of your program.

Now that you have an overview of the ports in this system, see if you can find in the block diagram the decoder which selects an addressed port. You should find the 3625 PROM labeled (A22) about in the center of the block diagram. Later we discuss how this device

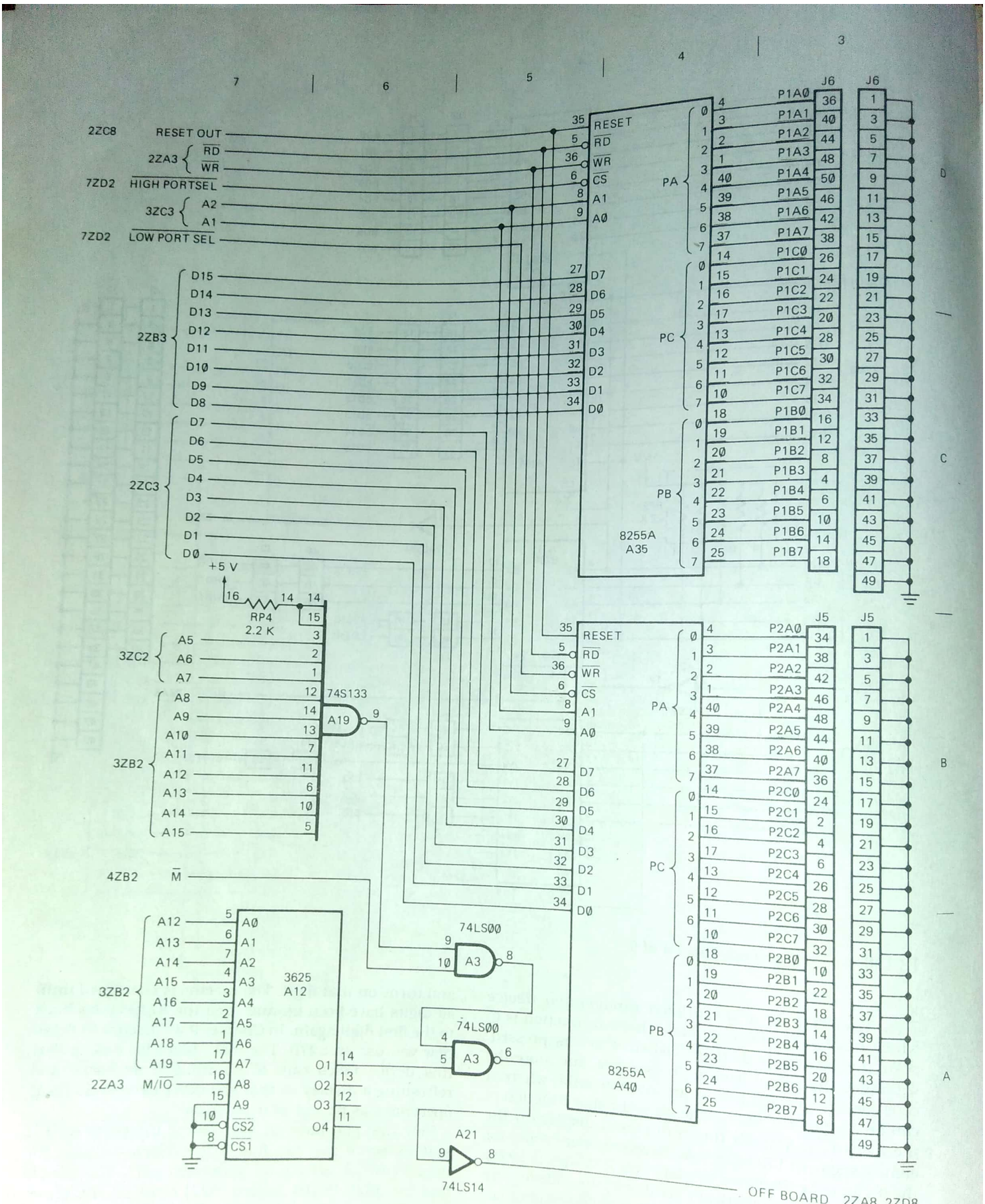


FIGURE 7-8 *(continued)* Sheet 5 of 9.

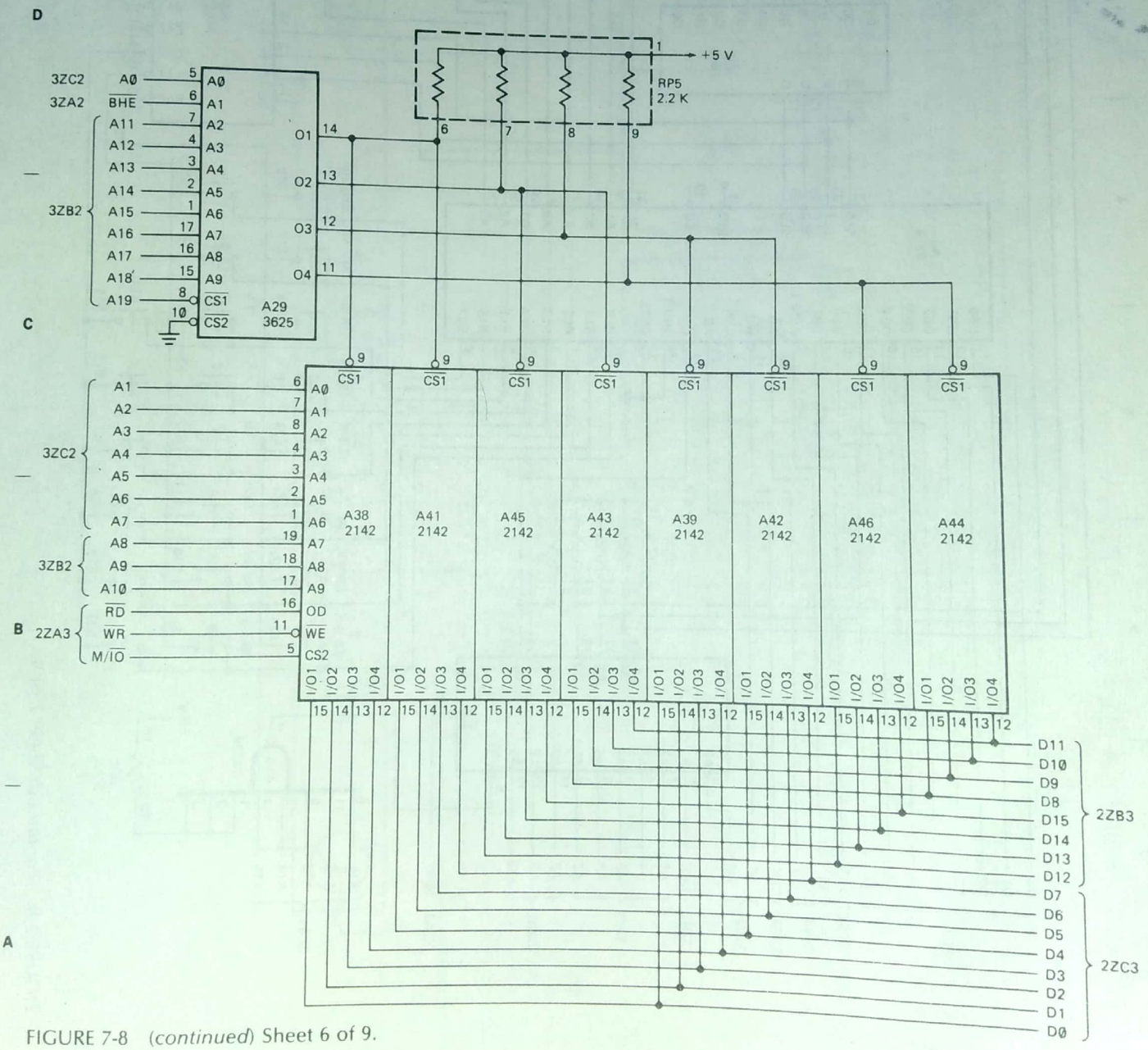


FIGURE 7-8 (*continued*) Sheet 6 of 9.

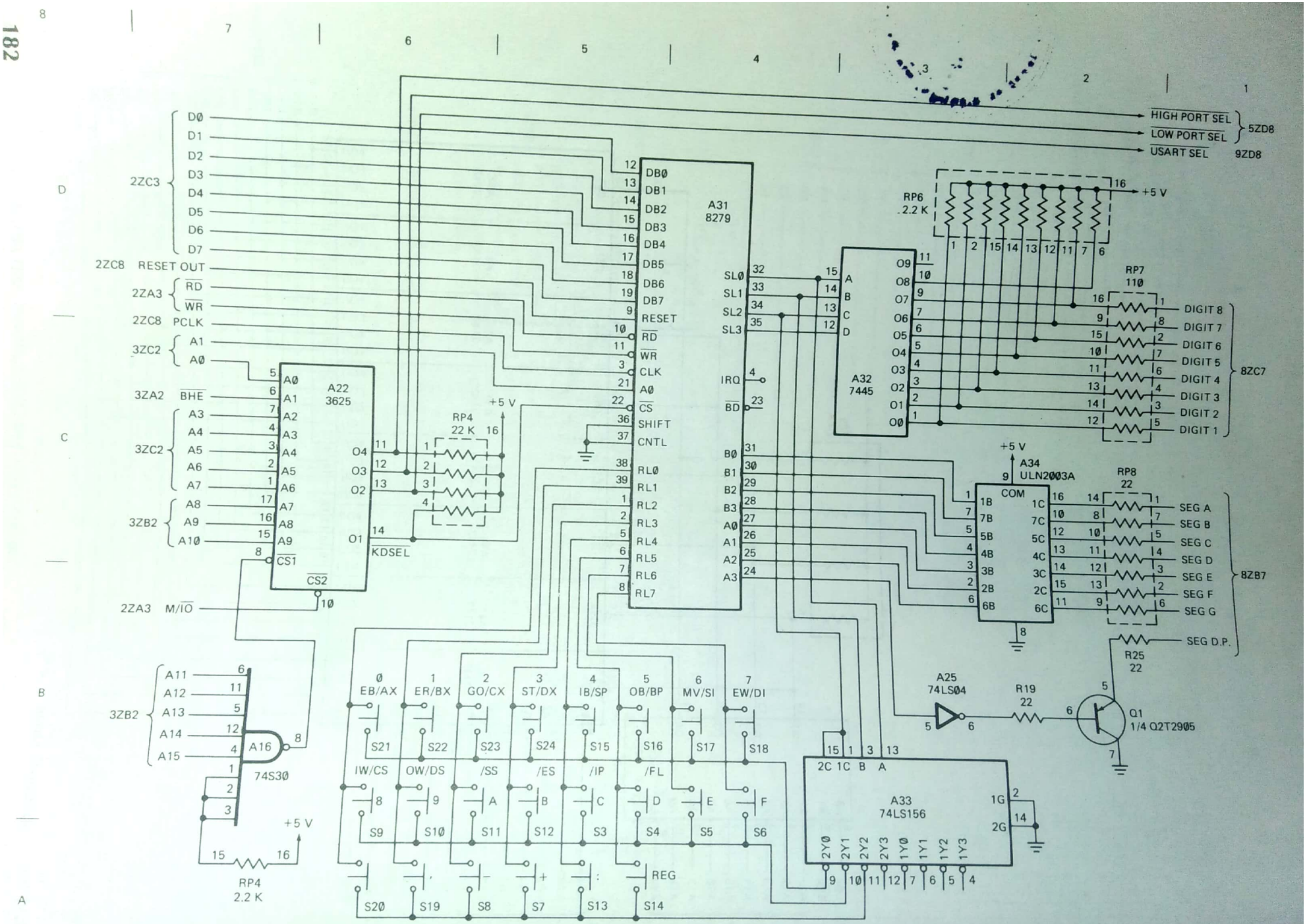


FIGURE 7-8 (*continued*) Sheet 7 of 9.

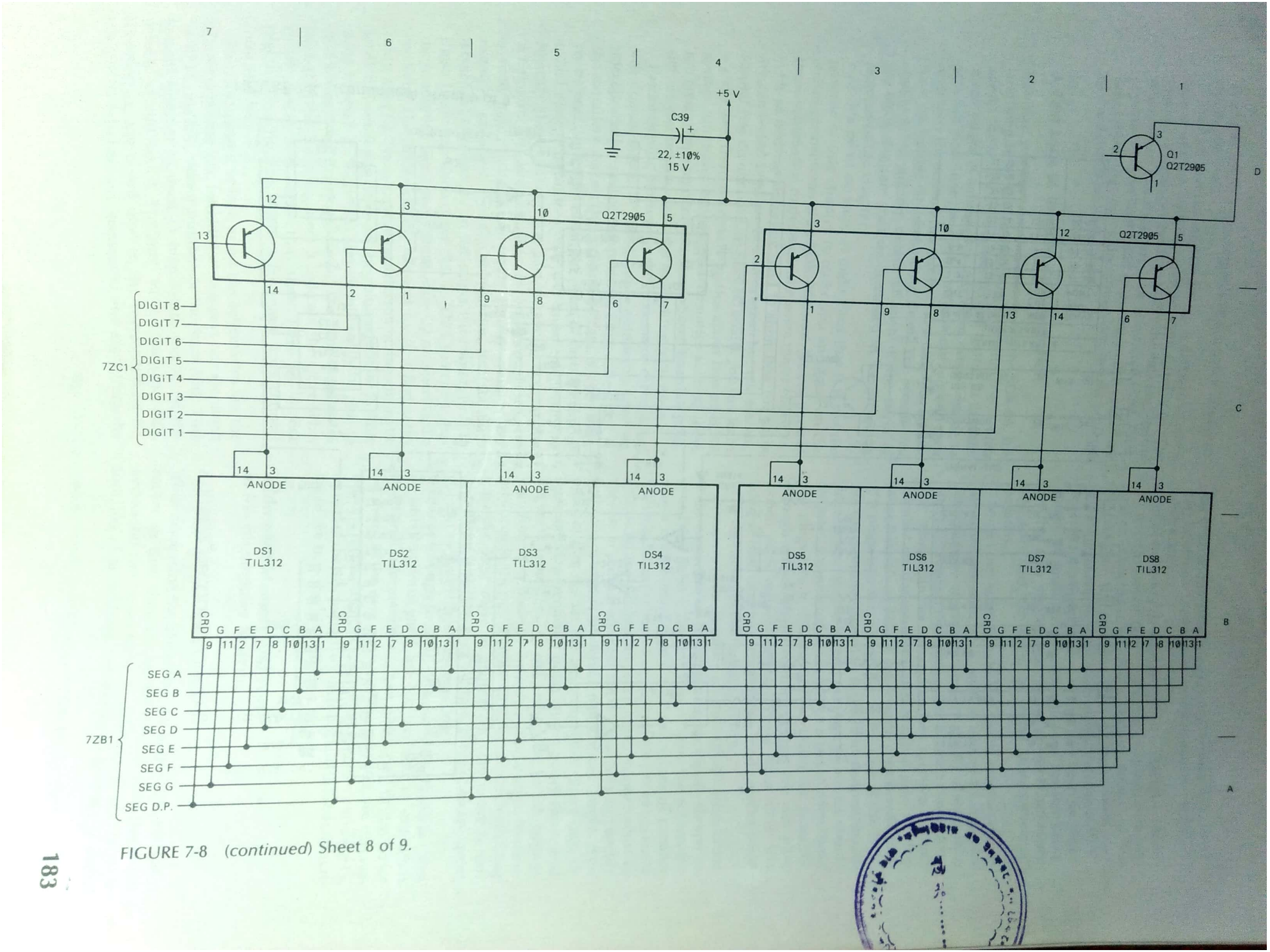


FIGURE 7-8 (*continued*) Sheet 8 of 9.

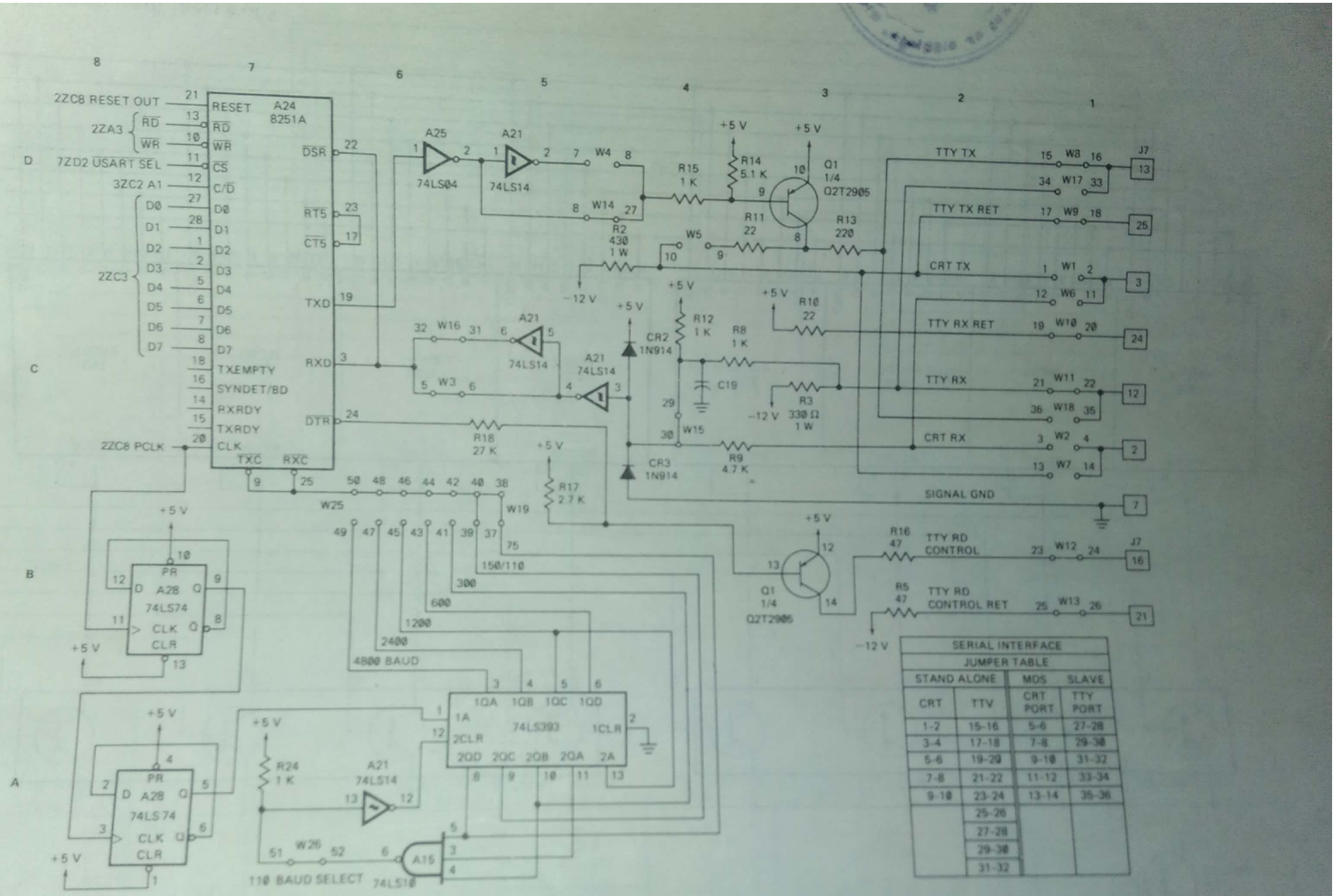


FIGURE 7-8 *(continued)* Sheet 9 of 9.

produces the port select signals from a port address sent out by the 8086.

The final parts of the SDK-86 block diagram to take a look at are the buffers along the right-hand edge. The purpose of these devices is to buffer the data and control bus lines so that they can drive additional ROM, RAM, or ports that you might add to the expansion area of the board. Note that the address lines are already buffered by the 74S373 address latches.

A First Look at the SDK-86 Schematics

Now that you have seen an overview of the SDK-86, the next step is to take a first look at Figure 7-8, which shows the actual schematics for the board. At first the many pages of schematics may seem overwhelming to you, but if you use the *5-minute freak-out rule* and then approach the schematics one part at a time, you should have no trouble understanding them. The schematics simply show greater detail for each of the parts of the block diagram that we discussed in the preceding sections of the chapter.

At this point we want to make clear that it is not the purpose of this chapter to make you an expert on the circuit connections of an SDK-86 board. We use parts of these schematics to demonstrate some major concepts, such as address decoding, and to show how the parts are connected together to form a small but real system. Even if you do not have an SDK-86 board, you can learn a great deal from these schematics about how an 8086 system functions. Multipage schematics such as these are typical for any microprocessor-based board or product, so you need to get used to working with them.

Before getting started on the next major concept, we will discuss some of the symbols commonly used on microprocessor system schematics. First, take a look at the numbers across the top and bottom of each schematic and the letters along the sides of each. These are called *zone coordinates*. You use these coordinates to identify the location of a part or connection on the schematic, just as you might use similar coordinates on a road map to help you locate Bowers Avenue. For example, on sheet 1 of the schematics, find the lines labeled A1 through A7 in the upper left corner. Next to these lines you should see 3ZC2. This indicates that these address lines come from zone C2 on sheet 3. To see what the lines actually connect to, first find schematic sheet 3. Then move across the row of the schematic labeled C until you come to the column labeled 2. This zone is small enough that you should easily be able to find where these lines come from. The zone coordinates next to these lines on sheet 3 indicate the other schematic sheets and zones that these lines go to. For practice, try finding where a few more lines connect from and to.

The next points to look at on the schematics are the numbers on the ICs. In addition to a part number such as 2716, each IC has a number of the form A36. This second number is used to help locate the IC on the printed circuit board. The number is commonly silk-screened on the board next to the corresponding IC. Usually IC numbers are sequential and start from the upper left corner of the component side of a board. There may be several 2716s on the board, but only one will be labeled A36.

In addition to ICs, another type of device often found on microprocessor boards is a *resistor pack*. You can find an example in zone C5 of schematic sheet 1. As you can see from the schematic, this device contains four 2.2-kΩ resistors. Resistor packs may physically be thin, vertical, rectangular wafers, or they may be in packages similar to small ICs. The advantages of resistor packs are that they take up less printed-circuit-board space and that they are easier to install than individual resistors.

Some other symbols to look at in the schematics are the structures with labels such as J2 and P1. You can find examples of these in zones C7 and B7 of schematic sheet 1. These symbols are used to indicate *connectors*. The number in the rectangular box specifies the pin number on the connector that a signal goes to. The letter P stands for *plug*. A connector is considered a plug if it plugs into something else. In the case of the SDK-86, the connector labeled P1 is the printed-circuit-board edge connector. The letter J next to a connector stands for *jack*. A connector is considered a jack if something else plugs into it. On the SDK-86 board the jacks J1 through J6 are 50-pin connectors that you can plug ribbon cable connectors into. These jacks allow the address bus, data bus, control bus, and parallel ports to be connected to additional circuitry.

One more point to notice on the SDK-86 schematics is the capacitors on the power supply inputs shown in zone B6 of sheet 1. As you can see there, the schematic shows a large number of 0.1-μF capacitors in parallel with a 22-μF capacitor. Most systems have *filtering* such as this on their power lines. You may wonder what is the use of putting all these small capacitors in parallel with one which is obviously many times larger. The point of this is that the large capacitor filters out, or *bypasses*, low-frequency noise on the power lines, and the small capacitors, spread around the board, bypass high-frequency noise on the power supply lines. Noise is produced on the power supply lines by devices switching from one logic state to another. If this noise is not filtered out with bypass capacitors, it may become great enough to disturb system operation.

Glance through the SDK-86 schematics to get an idea of where various parts are located and to see what additional information you can pick up from the notes on them. In the next section of this chapter, we discuss how microcomputer systems address memory and ports. As part of the discussion, we cycle back to these schematics to see how the SDK-86 does it.

Addressing Memory and Ports in Microcomputer Systems

ADDRESS DECODER CONCEPT

While discussing the block diagram of the SDK-86 board earlier in this chapter, we mentioned that the 3625 devices on the board serve as *address decoders*. One function of an address decoder is to produce a signal

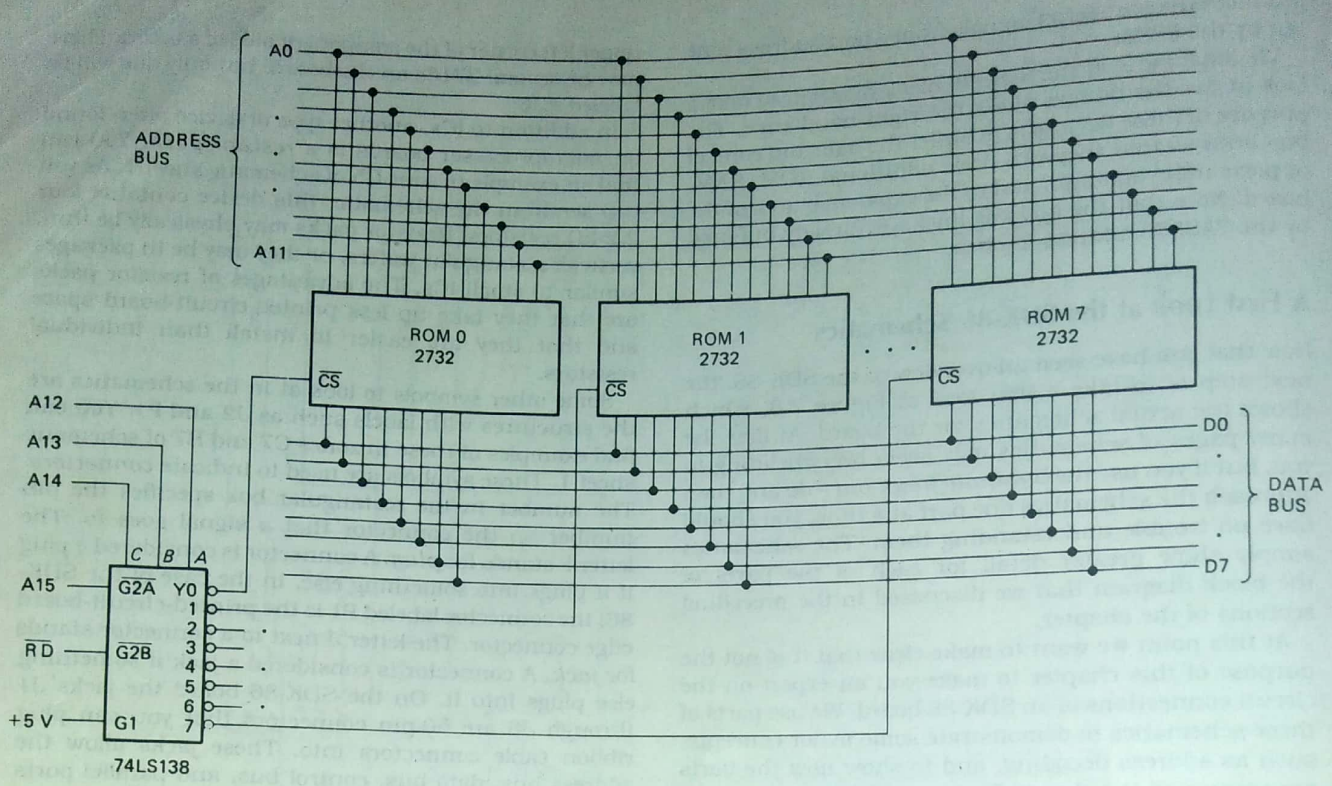


FIGURE 7-9 Parallel ROMs with decoder.

which enables the ROM, RAM, or port device that you want enabled for a particular address. A second, related function of an address decoder is to make sure that only one device at a time is enabled to put data on the data bus lines.

It seems that every microcomputer system does address decoding in a different way from every other system. Therefore, instead of memorizing the method used in one particular system, it is important that you understand the concept of address decoding. You can then figure out any system you have to work on.

AN EXAMPLE ROM DECODER

To start, look at Figure 7-9. This figure shows how eight EPROMs can be connected in parallel on a common address bus and common data bus. Just by looking at the schematic you can see that these EPROMs output bytes of data because each has eight outputs connected to the system data bus. The number of address lines connected to each device gives you an indication of how many bytes are stored in it. Each EPROM has 12 address lines (A0–A11) connected to it. Therefore, the number of bytes stored in the device is 2^{12} or 4096. If you have trouble with this, think of how many bits a counter has to have to count the 4096 states from 0 to 4095 decimal, or 0000H to 0FFFH.

Note that each 2732 in Figure 7-9 has a Chip Select, $\overline{CS}$, input. When this input is asserted low, the addressed byte in a device will be output on the data bus. The 74LS138 in Figure 7-9 makes sure that the $\overline{CS}$ input of only one ROM device at a time is low.

If the 74LS138 is enabled by making its $\overline{G2A}$ and $\overline{G2B}$ inputs low and its G1 input high, then only one output of the device will be low at a time. The output that will be low is determined by the 3-bit address applied to the C, B, and A select inputs. For example, if CBA is 000, then the Y0 output will be low, and all the other outputs will be high. This will assert the $\overline{CS}$ input of ROM 0. If CBA is 001, the Y1 output will be low and the ROM 1 will be selected. If CBA is 111, then Y7 will be low, and only ROM 7 will be enabled. Now let's see what address range these connections on the 74LS138 will give each of these ROMs in the system.

To determine the addresses of ROMs, RAMs, and ports in a system, a good approach in many cases is to use a worksheet such as that in Figure 7-10. To make one of these worksheets, you start by writing the address bits and the binary weight of each address bit across the top of the paper, as shown in the figure. To make it easier to convert binary addresses to hex, it helps if you mark off the address lines in groups of four, as shown. Next, draw vertical lines which mark off the three address lines that connect to the decoder select inputs (C, B, and A). For the decoder in Figure 7-9, address lines A14, A13, and A12 are connected to the C, B, and A inputs of the decoder, respectively. Then write under each address bit the logic level that must be on that line to address the first location in the first EPROM.

To address the first location in any of the EPROMs, the A0 through A11 address lines must all be low, so put a 0 under each of these address bits on the worksheet. To enable EPROM 0, the select inputs of the decoder must

		HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX EQUIVALENT ADDRESS
		2^{15} A15	2^{14} A14	2^{13} A13	2^{12} A12	2^{11} A11	2^{10} A10	2^{9} A9	2^{8} A8	2^{7} A7	2^{6} A6	2^{5} A5	2^{4} A4	2^{3} A3	2^{2} A2	2^{1} A1	2^{0} A0	
BLOCK 1	START	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0000
	END	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	= 0FFF
BLOCK 2	START	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 1000
	END	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	= 1FFF
BLOCK 3	START	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	= 2000
	END	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	= 2FFF
BLOCK 4	START	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	= 3000
	END	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= 3FFF
BLOCK 5	START	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 4000
	END	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	= 4FFF
BLOCK 6	START	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 5000
	END	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	= 5FFF
BLOCK 7	START	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	= 6000
	END	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	= 6FFF
BLOCK 8	START	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	= 7000
	END	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= 7FFF

DECODER ADDRESS INPUTS

FIGURE 7-10 Address decoder worksheet showing address decoding for eight 2732s in Figure 7-9.

be all 0's. Since address lines A14, A13, and A12 are connected to these select inputs, they must then all be 0's to enable EPROM 0. Write a 0 under each of these address bits on the worksheet. Since address line A15 is connected to the $\overline{G2A}$ enable input of the decoder, it must be asserted low in order for the decoder to work at all. Write a 0 under the A15 bit on your worksheet. Note that the $\overline{RD}$ signal from the microprocessor control bus is connected to the $\overline{G2B}$ enable input of the decoder. The decoder then will only be enabled during a read operation. This is done to make sure that data cannot accidentally be written to ROM. The G1 enable input of the decoder is permanently asserted by tying it to +5 V because we don't need it for anything else in this circuit.

You can now read the starting address of EPROM 0 directly from the worksheet as 0000H. The highest address in EPROM 0 is that address where A0–A11 are all 1's. If you put a 1 under each of these bits as shown on the worksheet, you can see that the ending address for EPROM 0 is 0FFFH. Remember that A12–A14 have to be low to select EPROM 0. A15 has to be low to enable the decoder. The address range of EPROM 0 is said to be 0000H to 0FFFH, a 4-Kbyte block.

Now let's use the worksheet to determine the address range for EPROM 1. EPROM 1 is enabled when A15 is 0, A14 is 0, A13 is 0, and A12 is 1. For the first address in EPROM 1, address lines A0 through A11 must all be low. Therefore, the starting address of EPROM 1 is 1000H. Its ending address, when A0 through A11 are all 1's, is 1FFFH. If you look at the worksheet in Figure 7-10, you should see that the address ranges for the other six EPROMs in the system are 2000H to 2FFFH, 3000H to 3FFFH, 4000H to 4FFFH, 5000H to 5FFFH, 6000H to 6FFFH, and 7000H to 7FFFH. In this system, then, we use address lines A14, A13, and A12 to select one of eight EPROMs in the overall address range of 0000H to 7FFFH. Some people like to think of address lines A14, A13, and A12 as "counting off" 4096-byte blocks of memory. If you think of the address lines as the outputs of a 16-bit counter, you can see how this works. The end address for each EPROM has all 1's in address bits A0–A11. When you increment the address to access the next byte in memory, these bits all go to 0, and a 1 rolls over into bits A14, A13, and A12. This increments the count in these 3 bits by 1 and enables the next highest 4096-byte EPROM. The count in these bits goes from binary 000 to 111.

AN EXAMPLE RAM DECODER

The system in Figure 7-9 contains only ROM. In most systems, you want to have ROM, RAM, and ports. To give you more practice with basic address decoding, we will show you now how you can add a decoder for RAM to the system.

Suppose that you want to add eight 2K × 8 RAMs to the system, and you want the first RAM to start at address 8000H, just above the EPROMs, which end at address 7FFFH.

To start, make a worksheet similar to the one in Figure 7-10. Addressing one of the 2048 bytes (2^{11}) in each RAM requires 11 address lines, A0 through A10. These lines will be connected directly to the address inputs on each RAM, so draw a vertical line on the worksheet to indicate this.

The three address lines A11, A12, and A13 will be used to select one of the eight RAMS, so write a 3-bit binary count sequence under these three columns in your worksheet.

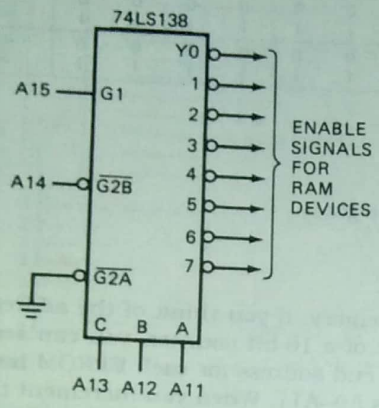
We want the RAM to start at address 8000H. For this address, A15 is a 1 and A14 is a 0, so mark these values in the appropriate columns in your worksheet. Your completed worksheet should look like the one in Figure 7-11*a*, p. 188. Now, let's see how you can implement this truth table with hardware.

Since you want to select one of eight RAM devices, you can use another 74LS138 such as the one we used for

HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX DIGIT				HEX EQUIVALENT ADDRESS	START OF BLOCK
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0		
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 8000H	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	= 8800H	2
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	= 9000H	3
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	= 9800H	4
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	= A000H	5
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	= A800H	6
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	= B000H	7
1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	= B800H	8

DECODER ADDRESS INPUTS

(*a*)



(*b*)

FIGURE 7-11 Address decoder. (*a*) Worksheet for eight 2-Kbyte RAMs starting at address 8000H. (*b*) Schematic for 74LS138 connections.

the EPROMs. You want to select 2048-byte blocks of memory, so address line A11 will be connected to the A input of the decoder, A12 will be connected to the B input of the decoder, and A13 will be connected to the C input of the decoder.

You want the block of RAM selected by the outputs of this decoder to start at address 8000H. For this, address A15 is high and A14 is low. The G1 enable input of the decoder is active high, so you connect it to the A15 address line. This input will then be asserted when A15 is high. You connect the A14 address line to the $\overline{G2A}$ input of the decoder so that this input will be asserted when A14 is low. Because you don't need to use it in this circuit, you can simply tie the $\overline{G2B}$ input of the decoder to ground so that it will be asserted all the time. Figure 7-11*b* shows the connections for this decoder. Note that you don't connect the 8086 $\overline{RD}$ signal to an enable input on a RAM decoder, because you want to enable the RAMs for both read and write operations.

From the worksheet or truth table in Figure 7-11*a*, you can quickly determine the address range for each of the RAMs. The first RAM will start at address 8000H. The ending address for this RAM will be at the address where bits A0–A10 are all 1's. If you put 1's under these bits on your worksheet, you should see that the ending address for the first RAM is 87FFH. For practice, work out the hexadecimal addresses for each of the other seven RAMs. When you finish, compare your results with those in Figure 7-11*a*. The eight RAMs occupy the address space from 8000H to BFFFH.

AN EXAMPLE PORT DECODER

Figure 7-12*a* shows how another 74LS138 can be connected in a system to produce chip select signals for some port devices. The truth table or address decoder worksheet in Figure 7-12*b* shows the system address which corresponds to each of the decoder outputs.

First, note that A15 and A14 must be high to enable the decoder, so these bits are 1's in the worksheet. Then notice that A13 and A12 must be low to enable the decoder, so these columns on the worksheet contain 0's. Finally, address lines A3, A4, and A5 are connected to the decoder select inputs, so we wrote a 3-bit binary count sequence in these columns in the worksheet.

Address lines A0, A1, and A2 will be connected directly to the port devices to address individual ports and control registers in the devices. This is the same idea as connecting the lower address lines directly to a ROM so that we can address one of the bytes stored there.

Address lines A6 through A11 are not connected to the port devices or to the decoder, so they have no effect on selecting a port. We don't care then whether these bits are 1's or 0's. As you will see, these "don't care" bits mean that there are many addresses which will turn on one of the port devices. To give the simplest address for each device, however, we assume that each of these don't care bits is 0. Write 0's under each of these bits on your worksheet. You should now see that the address C000H will cause the Y0 output of the decoder to be asserted. The address C008H will cause the Y1 output of the decoder to be asserted. Using address lines A3, A4, and A5 on the decoder select inputs, then, leaves eight address spaces for each port device.

To see that any one of several different addresses can select one of these port devices, replace the 0 you put under A6 on the first line of your worksheet with a 1. This represents a system address of C040H. A15 and A14 are 1's and A13, A12, A5, A4, and A3 are 0's for this address. Therefore, this address will also cause the Y0 output of the decoder to be asserted. You can try other combinations of 1's and 0's on A6 through A11 if you need to further convince yourself that these bits

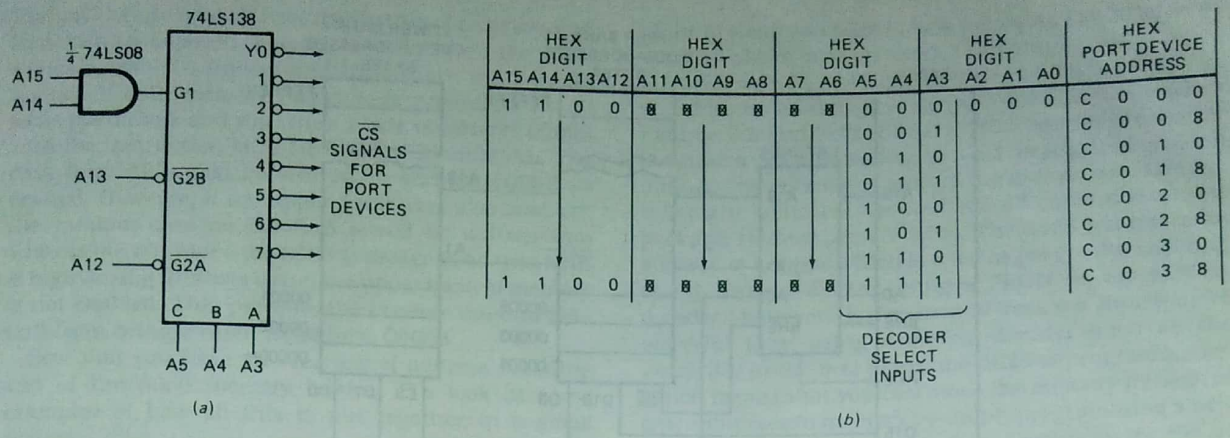


FIGURE 7-12 Adding a port device decoder. (*a*) Schematic for 74LS138 connections. (*b*) Address decoder worksheet.

don't matter when addressing ports. Again, we usually use 0's for these bits to give the simplest address.

Using a decoder which translates memory addresses to chip select signals for port devices is called *memory-mapped I/O*. In this system a port will be written to or read from in the same way as any other memory location. In other words, if this were an 8088 system, you would use an instruction such as MOV AL,DS:BYTE PTR 0C000H to read a byte of data from the first port to the AL register instead of using the MOV DX,0C000H and IN AL,DX instructions. The advantage of memory-mapped I/O is that any instruction which references memory can be used to input data from or output data to ports. In a system such as this, for example, the single instruction ADD AL,DS:BYTE PTR[0C000H] could be used to input a byte of data from the port at address C000H and add the byte to the AL register. The disadvantage of memory-mapped I/O is that some of the system memory address space is used up for ports and is therefore not available for memory.

You can use memory-mapped I/O with any microprocessor, but some microprocessors, such as those of the 8086 family, allow you to set up separate address spaces for input ports and for output ports. You access ports in these separate address spaces directly with the IN and OUT instructions. Having separate address spaces for input and output ports is called *direct I/O*. The advantage of direct I/O is that none of the system memory space is used for ports. The disadvantage is that only the specialized IN and OUT instructions can be used to input or output data.

In a later section of this chapter, we show how direct I/O is done with the 8086, but first we will discuss how the 8086 addresses memory.

8086 and 8088 Addressing and Address Decoding

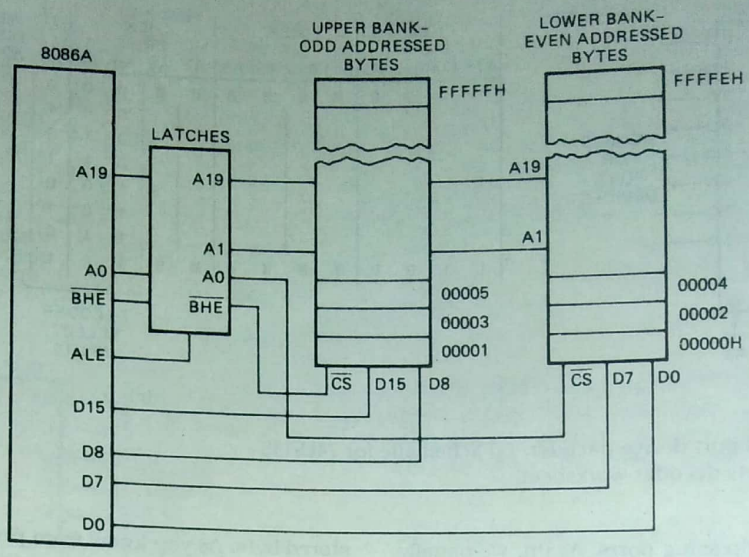
8086 MEMORY BANKS

The 8086 has a 20-bit address bus, so it can address 2^{20} or 1,048,576 addresses. Each address represents a stored byte. As you know from previous chapters, when you write a word to memory with an instruction such as MOV DS:WORD PTR[437AH],BX, the word is actually written into two consecutive memory addresses. Assuming that DS contains 0000, the low byte of the word is written into the specified memory address, 0437AH, and the high byte of the word is written into the next-higher address, 0437BH. To make it possible to read or write a word with one machine cycle, the memory for an 8086 is set up as two "banks" of up to 524,288 bytes each. Figure 7-13*a*, p. 190, shows this in diagram form.

One memory bank contains all the bytes which have even addresses such as 00000, 00002, and 00004. The data lines of this bank are connected to the lower eight data lines, D0 through D7, of the 8086. The other memory bank contains all the bytes which have odd addresses such as 00001, 00003, and 00005. The data lines of this bank are connected to the upper eight data lines, D8 through D15, of the 8086. Address line A0 is used as part of the enabling for memory devices in the lower bank. An addressed memory device in this bank will be enabled when address line A0 is low, as it will be for any even address. Address lines A1 through A19 are used to select the desired memory device in the bank and to address the desired byte in that device.

Address lines A1 through A19 are also used to select a desired memory device in the upper bank and to address the desired byte in that bank. An additional part of the enabling for memory devices in the upper bank is a separate signal called *bus high enable*, $\overline{\text{BHE}}$. $\overline{\text{BHE}}$ is multiplexed out from the 8086 on a signal line at the same time as an address is sent out. An external latch, strobed by ALE, grabs the $\overline{\text{BHE}}$ signal and holds it stable for the rest of the machine cycle, just as is done with addresses. Figure 7-13*b* shows you the logic level that will be on the $\overline{\text{BHE}}$ and A0 lines for different types of memory accesses.

If you read a byte from or write a byte to an even address such as 00000H, A0 will be low and $\overline{\text{BHE}}$ will be high. The lower bank will be enabled, and the upper bank will be disabled. A byte will be transferred to or from the addressed location in the low bank on D0–



(*a*)

ADDRESS	DATA TYPE	$\overline{BHE}$	A0	BUS CYCLES	DATA LINES USED
0000	BYTE	1	0	ONE	D0-D7
0000	WORD	0	0	ONE	D0-D15
0001	BYTE	0	1	ONE	D7-D15
0001	WORD	0	1	FIRST	D0-D7
		1	0	SECOND	D7-D15

(*b*)

FIGURE 7-13 8086 memory banks. (*a*) Block diagram. (*b*) Signals for byte and word operations.

D7. For an instruction such as MOV AH,DS:BYTE PTR[0000], the 8086 will automatically transfer the byte of data from the lower data bus lines to AH, the upper byte of the AX register. You just write the instruction and the 8086 takes care of getting the data in the right place.

Now, if the DS register contains 0000H and you use an instruction such as MOV AX,DS:WORD PTR[0000] to read a word from memory into AX, both A0 and $\overline{BHE}$ will be asserted low. Therefore, both banks will be enabled. The low byte of the word will be transferred from address 00000H to the 8086 on D0–D7. The high byte of the word will be transferred from address 00001H to the 8086 on D8–D15. The 8086 memory, remember, is set up in banks so that words, which have their low byte at an even address, can be transferred to or from the 8086 in one bus cycle. When programming an 8086, then, it is important to start an array of words on an even address for most efficient operation. If you are using an assembler, the EVEN directive is used to do this.

When you use an instruction such as MOV AL,DS:BYTE PTR[0001] to access just a byte at an odd address, A0 will be high and $\overline{BHE}$ will be asserted low. Therefore, the low bank will be disabled, and the high bank will be enabled. The byte will be transferred from memory address 00001H in the high bank to the 8086 on lines D8–D15. The 8086 will automatically transfer the byte of data from the higher eight data lines to AL, the low byte of the AX register. Note that address 00001H is actually the first location in the upper bank.

The final case in Figure 7-13*b* is the one where you want to read a word from or write a word to an odd address. The instruction MOV AX,DS:WORD PTR[0001H] copies the low byte of a word from address 00001 to AL and the high byte from address 00002H to AH. In this case, the 8086 requires two machine cycles to copy the two bytes from memory. During the first machine cycle the 8086 will output address 00001H, assert $\overline{BHE}$ low, and assert A0 high. The byte from address 00001H will be read into the 8086 on lines D8–D15 and put in AL. During the second machine cycle the 8086 will send out address 00002H. Since this is an even address, A0 will be low. However, since we are accessing only a byte, $\overline{BHE}$ will be high. The second byte will be read into the 8086 on lines D0–D7 and put in AH. Note that the 8086 automatically takes care of getting a byte to the correct register regardless of which data lines the byte comes in on.

The main reason that the A0 and $\overline{BHE}$ signals function

the way they do is to prevent the writing of an unwanted byte into an adjacent memory location when the 8086 writes a byte. To understand this, think what would happen if both memory banks were turned on for all write operations and you wrote a byte to address 00002 with the instruction MOV DS:BYTE PTR[0002],AL. The data from AL would be written to address 00002 as desired. However, if the upper bank were also enabled, the random data on D8–D15 would be written into address 00003. Since the 8086 is designed so that $\overline{BHE}$ is high during this byte write, the upper bank of memory is not enabled. This prevents the random data on D8–D15 from being written to address 00003.

Now that you have an overview of address decoding and of the 8086 memory banks, let's look at some examples of how all this is put together in a small system.

ROM ADDRESS DECODING ON THE SDK-86

Sheet 1 of the SDK-86 schematics in Figure 7-8 shows the circuit connections for the EPROMs and EPROM decoder. The 2716 EPROMs there are 2K × 8 devices. Two of the EPROMs have their eight data outputs connected in parallel to system data lines D0–D7. These two EPROMs then give 4 Kbytes of storage in the lower memory bank. The other two EPROMs have their data outputs connected in parallel to system data lines D8–D15 to give 4 Kbytes of storage in the upper bank of ROM.

Eleven address lines are needed to address the 2 Kbytes in each device. Therefore, system address lines A1–A11 are connected to all the EPROMs in parallel. Remember that A0 cannot be used to select a byte in the EPROMs because, as we described in the last section, it is used to enable or disable the lower bank.

A 2716 has two enable inputs, $\overline{CE}$ and $\overline{OE}$. In order for the 2716 to output an addressed byte, both of these enable inputs must be asserted low. The $\overline{CE}$ inputs of the two devices in the lower bank are connected to system address line A0, so the $\overline{CE}$ inputs of these devices will be asserted if A0 is low. The $\overline{CE}$ inputs of the two 2716s in the upper bank are connected to the $\overline{BHE}$ line. The $\overline{CE}$ inputs of these devices then will be asserted whenever $\overline{BHE}$ is asserted low. To summarize, then, the two devices labeled A27 and A36 form the lower bank of EPROMs and the two devices labeled A30 and A37 form the upper bank of EPROMs in this system. To see how the $\overline{OE}$ enable input of each of these devices gets asserted and to determine the address that each device will have in the system, you need to look next at the 3625 address decoder labeled A26 on sheet 1 of Figure 7-8.

A 3625 is a 1K × 4 bipolar PROM which functions as an address decoder, just as the 74LS138 performs in Figures 7-9 and 7-11. Since a 3625 has open collector outputs, a pull-up resistor to +5 V is required on each output. The dotted box around the four resistors on the schematic indicates the four are all contained in one package, resistor pack 5 (RP5). The 3625 translates an address to a signal which is used as part of the enabling of the desired device. Using a PROM as an address decoder, however, is for several reasons much more powerful than using a simple decoder such as the 74LS138. In the first place, the 3625 is programmable, which means that you can move the memory devices to new addresses in memory by simply programming a new PROM. Second, the large number of inputs on the PROM allows you to select a specific area of memory without using external gates. If, for example, you wanted the $\overline{G2A}$ input of a 74LS138 to be asserted if A11–A15 were all high, you would have to use an external NAND gate to detect this condition. With a PROM, you can just make this condition part of the truth table you use to burn the PROM.

Now, to analyze any address decoder circuit, first determine what signals are required to enable the decoder. The $\overline{CS1}$ enable input of the 3625 EPROM decoder is tied to ground, so it is permanently enabled. The $\overline{CS2}$ enable input is tied to the $\overline{RD}$ signal from the 8086, so that the decoder will only be enabled if the 8086 is doing a read operation. As explained previously, you don't want to accidentally enable a ROM if you send out a wrong address during a write operation.

The next step in analyzing a decoder circuit using a PROM is to consult the manufacturer's manual for the system. You have to do this because, for a PROM, the relationship between the inputs and the outputs cannot be determined directly from the schematic.

Figure 7-14 shows the truth table for the PROM from the SDK-86 manual. This truth table is just a compressed form of writing an address decoder worksheet such as those we used in the previous discussion of address decoding. From the truth table you can see that in order for the O1 output of the 3625 to be asserted low, M/$\overline{IO}$ has to be high. This is reasonable, since this decoder is enabling memory devices, not port devices. Also, address lines A12 through A19 have to be high in order for the O1 output of the PROM to be asserted low. Since the upper eight address bits must all be 1's for the O1 output to be asserted, the lowest address which

PROM INPUTS				PROM OUTPUTS				PROM ADDRESS BLOCK SELECTED
M/$\overline{IO}$	A14-A19	A13	A12	O4	O3	O2	O1	
1	1	1	1	1	1	1	0	FF000H-FFFFFH
1	1	1	0	1	1	0	1	FE000H-FEFFFH
1	1	0	1	1	0	1	1	FD000H-FDFFFH ($\overline{CSX}$)
1	1	0	0	0	1	1	1	FC000H-FCFFFH ($\overline{CSY}$)
ALL OTHER STATES				1	1	1	1	NONE

FIGURE 7-14 Truth table for an SDK-86 ROM decoder PROM (A26).