CPUville Z80 Computer Kit Instruction Manual

By Donn Stewart

Table of Contents

Introduction

The CPUville Z80 Computer Kit is an educational kit consisting of four parts: a computer kit, a bus display kit, a logic probe kit, and this instruction manual.

The kit is based on a small 8-bit computer system designed around the Zilog Z80[1] Central Processing Unit (CPU). This popular microprocessor has been used for years in small personal computer systems, and is still being used in embedded systems, such as those controlling appliances. It is easy to design with, yet has an extensive set of instructions, allowing sophisticated programming.

The CPUville Z80 computer is based on designs popularized by the book Build Your Own Z80 Computer by Steve Ciarcia[2]. This book, while out of print, is available for on-line viewing at Google Books. Other design ideas come from Z80 Microcomputer Design Projects by William Barden, Jr.[3], and from me tinkering around.

The original prototype system for this kit is described in detail on my website, http://cpuville.com/Kits/Z80-kits-home.html. The wire-wrapped system described there was translated almost exactly into a printed circuit board system that a hobbyist or student can solder together themselves. I used the open-source KiCad package to design the printed circuit board. The boards for the kit have been manufactured in the USA using lead-free technology by Advanced Circuits, Aurora, Colorado. The parts are all through-hole, that is, no surface-mount devices, so soldering is easy. While this project might be hard for a novice (there are over 500 pins to be soldered on the computer board), anyone with some soldering experience and patience should be able to complete it successfully.

The computer has 4096 bytes (4K[4]) of memory, divided into 2K erasable-programmable read-only memory (EPROM) and 2K random access memory (RAM). There are two input ports, which are small switches, that allow data entry one byte at a time. There are two output ports which display 8-bit output on light-emitting diodes (LEDs). The maximum clock speed is 2 MHz (megahertz, or million cycles per second).

The computer is designed to be paired with a display board that shows the activity on the computer system buses, which are the sets of parallel wires the parts use to communicate with each other. The computer has two clock speeds. The "fast" clock is 2MHz, and runs the system when you are using the computer normally. The slow clock is only a few Hz. When the computer is paired with the bus display, running it with the slow clock allows you to observe what is happening on the system buses. This makes a good classroom demonstration project. If a few cycles per second is still too fast, you can take a video of the computer running on the slow clock, and look at it frame-by-frame. In theory, the CPU can be single stepped, but I have had difficulty making this work reliably. Maybe I will get this to work in the next version of the kit.

The 2K of read-only memory is in a 2716 EPROM. This comes pre-programmed with some small test

1 Z80 is a registered trademark of Zilog, Inc.
2 Build Your Own Z80 Computer by Steve Ciarcia, 1981, Byte Books, McGraw-Hill, Peterborough, New Hampshire
3 Z-80 Microcomputer Design Projects by William Barden, Jr., 1980, Howard W. Sams & Co., Inc, Indianapolis, Indiana
4 When discussing binary addresses I will use the convention of "K" as shorthand for the number 1024, which is 2^10. This convention does not apply to the use of "K" for values of electronic components, where it means 1000.

programs, and with a program loader that allows the user to enter their own program into the 2K RAM, and execute it. There is also program code for running a serial interfact, which is available as a separate kit. A complete listing of the EPROM contents, and some example programs for entry into RAM, with comments, are included in this manual. The EPROM has a window over the chip to allow UV light in to erase it. I think this is educational, because it allows you to see how small the integrated circuit really is. The bulk of the part is packaging that makes it large enough for human hands to handle. The EPROM is socketed so you can remove it, and program it yourself if you have your own E/EPROM programmer. The Z80 and 2K RAM are also socketed, so if you get tired of this kit, you can take them out and make your own computer project.

The kit is expandable to some degree. A serial interface kit is available, that allows you to communicate with the computer using a PC running terminal emulation software. The on-board memory and input-output ports can be disabled by removing shorting blocks on two jumpers. This allows the hobbyist to create add-on boards with increased input/output ports and memory, or a memory-mapped display. Add-on boards can be connected to the computer using the bus connection sockets where the bus display board attaches. The advanced hobbyist will need to know that the Z80 interrupts and direct memory access signals have not been brought out to the bus connectors, so there is some limit to how fancy an expansion board can be. I might make an upgraded kit with these signals implemented in the future, but for now I am offering the simple kit as described here.

This is an educational kit. If you learn something from building the computer, then I have accomplished my purpose. The computer is not intended to be useful outside its educational purpose. In particular, it is not designed to control machinery or processes where failure might result in property damage or injury. It has no way of being connected to the Internet, unless you make your own connection hardware and software. These are projects for the future.

This manual has detailed instructions for assembling the logic probe, bus display and computer kits, including photographs to illustrate critical component placements. There are two sets of instructions for building the computer. One set is for those who just want to build it quickly, which can be done in a few hours. The other set, in the supplemental materials, is for a more “educational” assembly, where each section of the computer system is built one at a time, with the idea that a student would look at the function of each section in detail. When the computer is built by sections, the logic probe or bus display can be used to show that the section is functional before going on to build the next section. A full set of schematics with explanations is included. I hope you have some educational fun with this project!

--Donn Stewart, May 2012

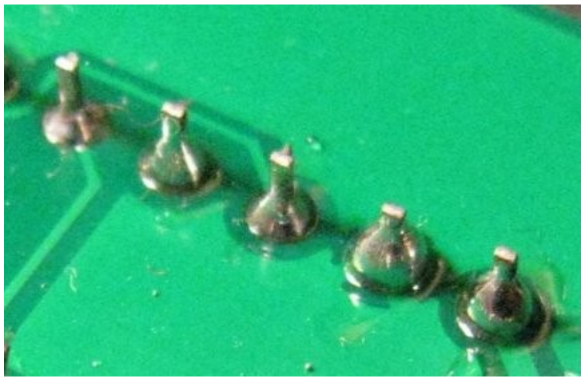
Building Tips

Thanks for buying a CPUville kit. Here is what you need to build it:

1. Soldering iron. I strongly recommend a 15 watt iron. You may use a 30 watt iron, but you will have to be a little more careful, and faster, to avoid damaging the parts or the board.
2. Solder. Use rosin core solder. Lead-free or lead-containing solder are fine. I have been using Radio Shack Standard Rosin Core Solder, 60/40, 0.032 in diameter. Use eye protection when soldering, and be careful, you can get nasty burns even from a 15-watt iron.
3. Tools. You will need needle nose pliers to bend leads. You will need wire cutters to cut leads after soldering, and possibly wire strippers if you want to solder power wires directly to the board. I find a small pen knife useful in prying chips or connectors from their sockets. A voltmeter is useful for testing continuity and voltage polarity. A logic probe is useful for checking voltages on IC pins while the computer is running, to track down signal connection problems.
4. De-soldering tool. Hopefully you will not need to remove any parts from the board, but if you do, some kind of desoldering tool is needed. I use a “Soldapullt”, a kind of spring-loaded syringe that aspirates melted solder quickly. Despite using this, I destroy about half the parts I take off, so it is good to be careful when placing the parts in the first place, so you don't have to remove them later.

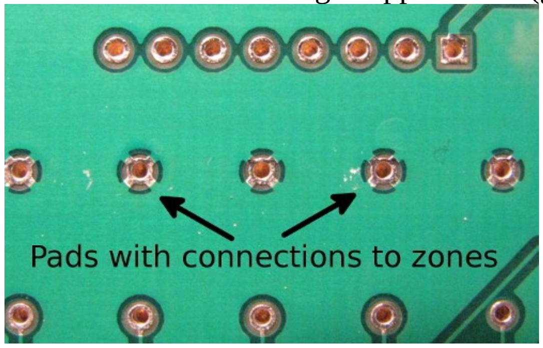
Soldering tips:

1. Before you plug in the iron, clean the tip with something mildly abrasive, like steel wool or a 3M Scotchbrite pad (plain ones, not the ones with soap in them).
2. Let the iron get hot, then tin the tip with lots of solder (let it drip off some). With a fresh coat of shiny solder the heat transfer is best.
3. Wipe the tinned tip on a wet sponge briefly to get off excess solder. Wipe it from time to time while soldering, so you don't get a big solder drop on it.
4. All CPUville kits have through-hole parts (no surface-mounted devices). This makes it easy for even inexperienced hobbyists to be successful.
5. The basic technique of soldering a through-hole lead is as follows:
   1. Apply the soldering iron tip so that it heats both the lead and the pad on the circuit board
   2. Wait a few seconds (I count to 4), then apply the solder.
   3. Apply only the minimum amount of solder to make a small cones around the leads, like this:



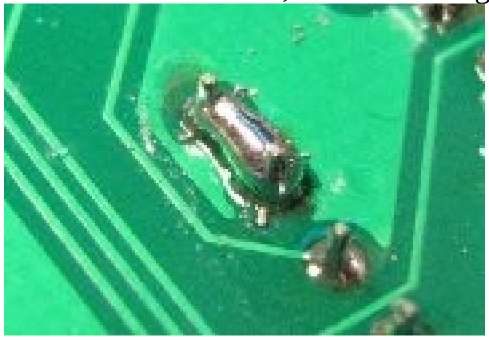
This is only about 1/8th inch of the 0.032 inc diameter solder that I use. If you keep applying the solder, it will drip down the lead to the other side of the board, and you can get shorts. Plus, it looks bad.

4. Remove the solder first, wait a few seconds, then remove the soldering iron. Pull the iron tip away at a low angle so as not to make a solder blob.
5. There are some pads with connections to large copper zones (ground planes) like these:



These require extra heat to make good connections, because the zones wick away the soldering iron heat. You might need a more powerful (30 watt) soldering iron. If all else fails, you can take a razor blade and cut one or two of the connecting traces. This should slow the escape of heat enough to solder.

6. The three main errors one might make are these:
   1. Cold joint. This happens when the iron heats only the pad, leaving the lead cold. The solder sticks to the pad, but there is no electrical connection with the lead. If this happens, you can usually just re-heat the joint with the soldering iron in the proper way (both the lead and the pad), and the electrical connection will be made.
   2. Solder blob. This happens if you heat the lead and not the pad, or if you pull the iron up the lead, dragging solder with it. If this happens, you can probably pick up the blob with the hot soldering iron tip, and either wipe it off on your sponge and start again, or carry it down to the joint and make a proper connection.
   3. Solder bridge. This happens if you use too much solder, and if flows over to another pad. This is bad, because it causes a short circuit, and can damage parts.

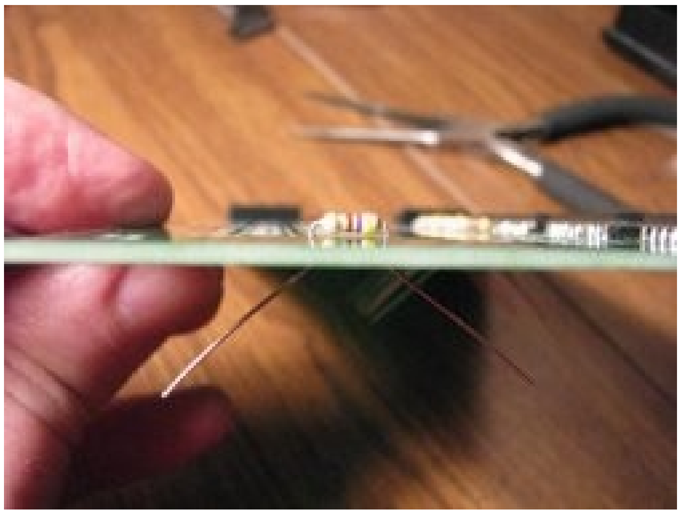


If this happens, you have to remove the solder with a desoldering tool, and re-do the joints.

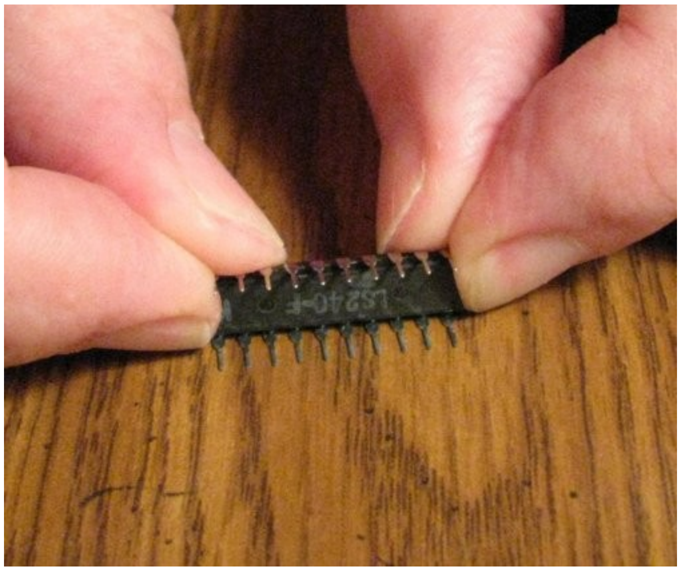
Other tips

1. Be careful not to damage the traces on the board. They are very thin copper films, just under a thin plastic layer of solder mask (the green stuff). If you plop the board down on a hard surface that has hard debris on it (like ICs, screws etc.) it is easy to cut a trace. Such damage can be fixed, if you can find it, but try to avoid it in the first place.

2. When soldering multi-pin components, like the ICs, it is important to hold the parts against the board when soldering so they aren't “up in the air” when the solder hardens. The connections might be OK, but it looks terrible. If you make a lot of connections on a part while it is up in the air it is very difficult to get it to sit back down, because you cannot heat all the connections at the same time. To prevent this, I like to solder the lowest profile parts first, like the ICs, because when the board is upside down they will be pressed against the top of the board by the surface of the table I am working on. Then, I solder the taller parts, like the LEDs, then the switches and capacitors. Sometimes, I need to put something beneath the component to support it while the board is upside down to be soldered, like a rolled-up piece of paper. Another technique is to put a tiny drop of solder on the tip of the iron, press the chip against the board with one hand, and apply the drop of solder to one of the leads. When the solder hardens, it holds the chip in place. Solder the other leads, then come back and re-solder the one you used to hold it. It is good to re-solder it because the original solder drop will not have had any rosin in it. The rosin in the cold solder helps the electrical connection to be clean.
3. The components with long bendable leads (capacitors, resistors, and LEDs) can be inserted, and then the leads bent to hold them in place:



4. You might have to bend the leads on ICs to get them to fit into the holes on the boards. Place the part on the table and bend the leads all at once, like this:



Bending the leads one-by-one or all together with the needle nose pliers doesn't work as well for some reason.

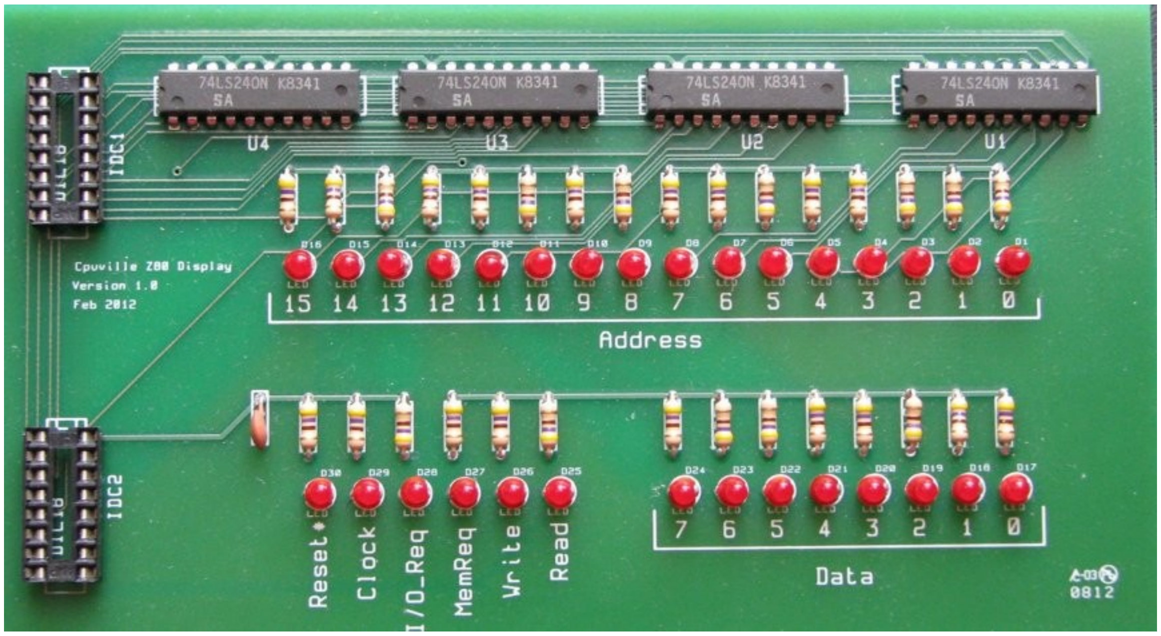
5. After you have soldered a row or two check the joints with a magnifying glass. These kits have small leads and pads, and it can be hard to see if you got the solder on correctly by naked eye. You can miss tiny hair-like solder bridges unless you inspect carefully. It is good to brush off the bottom of the board from time to time with something like a dry paintbrush, to get off any small solder drops that are sitting there. Also, hold the board up to the light, looking at the bottom. If you can see light coming through any of the holes, that means there is inadequate solder. The computer kit has over 500 connections to solder, and you will probably forget to do some. I have. Of course, the vias, the little plated holes where a printed circuit board trace goes from one side of the board to the other, do not need any solder, so they will stay open.

Building the Logic Probe

Please refer to the new logic probe instruction manual, which can obtained from this link:

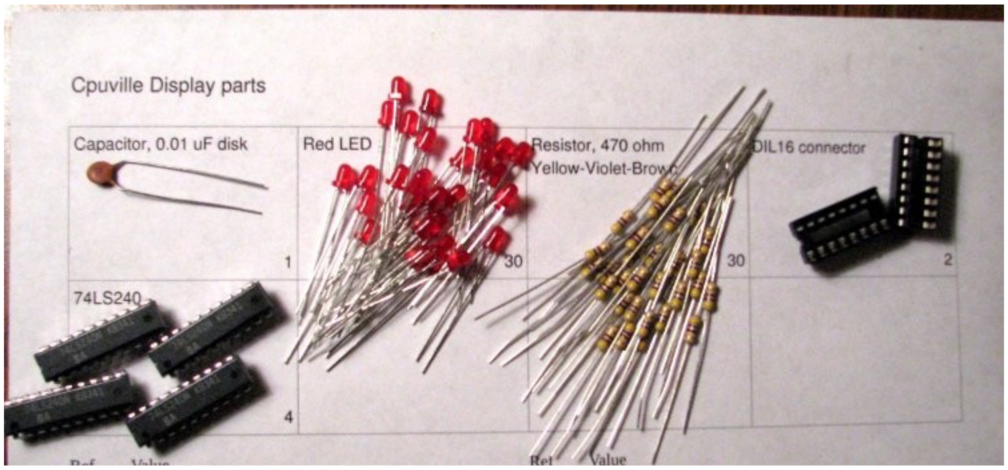
http://www.cpuville.com/Kits/logic-probe-instructions.pdf

Building the Display

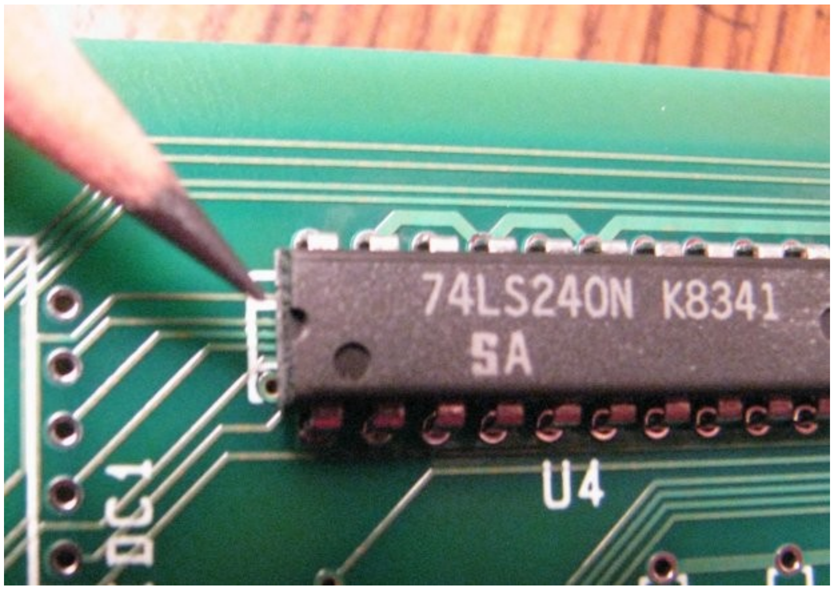


Next to the logic probe, the display is easiest to build. If you bought both the display and computer kit, build the display first.

1. Use the parts organizer sheet (in the Appendix) to count the parts, and get familiar with them.



2. Most of the parts need to be placed in the board in the proper orientation:
   1. LEDs: the cathode (the short lead) is the more negative of the two leads, and is marked by the flat side of the flange on the plastic LED body. The flat side – short lead goes toward the RIGHT (see the picture in the logic probe instructions).
   2. ICs: The LEFT-hand side of each IC has a little cut-out:

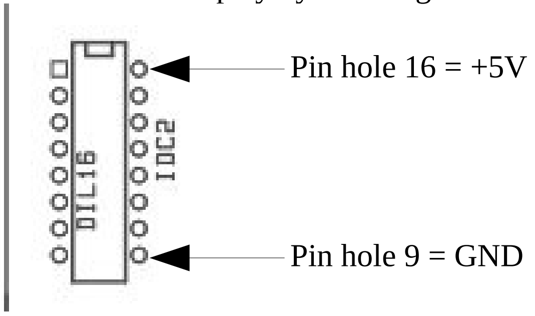


This makes sure that Pin 1 will be in the LEFT lower corner.

3. Resistors and ceramic capacitors (disks) do not have to be oriented.
4. The 16-pin connectors have no orientation, but there is a cut-out toward one end, I usually put this toward the top of the board.
5. There is no reason to put the parts on the board in any particular order. You should start with the low-profile parts first, then work up to the taller parts. This is because when you have the board upside down for soldering, the parts will sit flat against the top side. The parts from flattest to tallest: resistors, ICs, sockets for connectors, LEDs, capacitor.

The display is simple to build, but soldering all those resistors and LEDs can get tiring. Take your time and try to get the LEDs in so they stand up straight. What I have tried is to solder one lead of each LED, then turn the board over and try to straighten the LED bodies. Since the other lead is not soldered you can bend them a little. Do not use too much force, or you can break the LED body off the leads. They don't have to be perfectly straight. The LEDs supplied with the kit have a fairly wide viewing angle.

You can test the display by inserting solid-core wires into IDC2 socket pin holes 9 and 16:

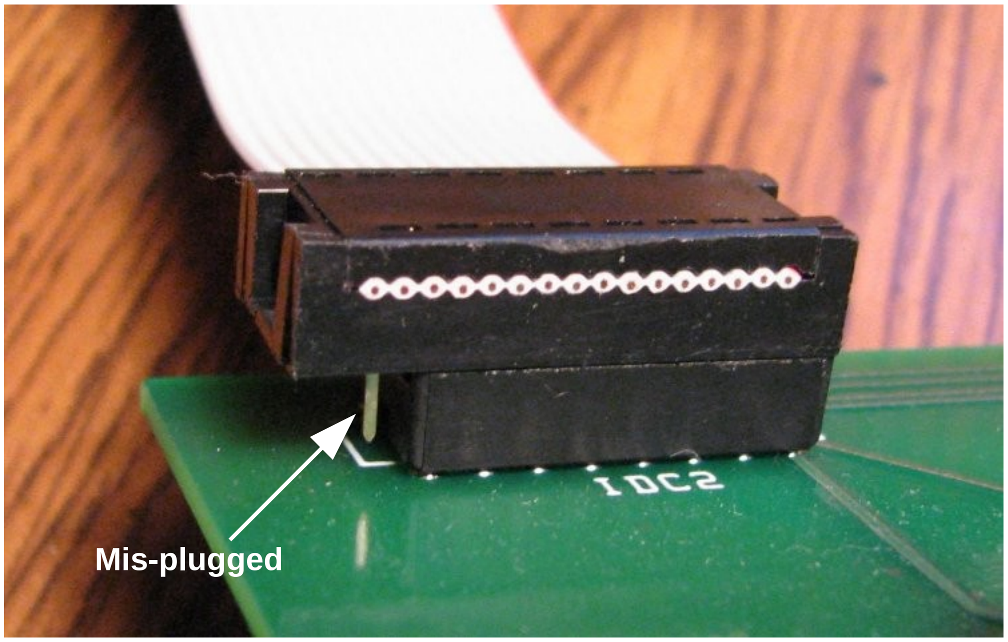


Connect +5V Regulated DC[5] to the wire in hole 16, and ground to the wire in hole 9. All the LEDs will light. Then, if you insert a third wire from the same circuit connected to ground into each other hole of

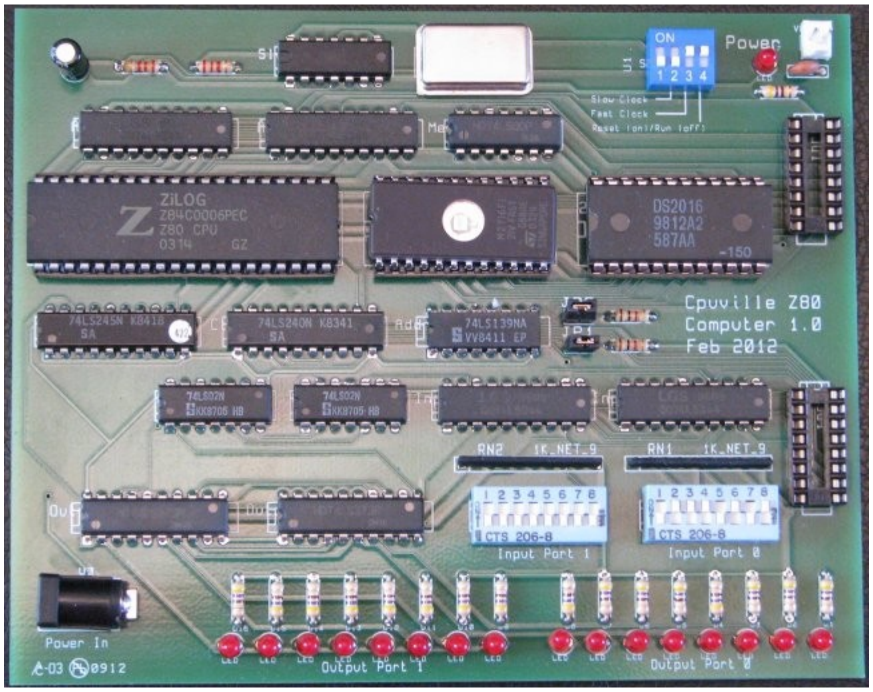
5 This project requires a +5V **regulated** DC power supply capable of at least 2000 mA (i.e., a 10 watt power supply). An unregulated power supply will not work properly and may damage the system.

the connectors, one at a time, the LED corresponding to that hole should turn off.

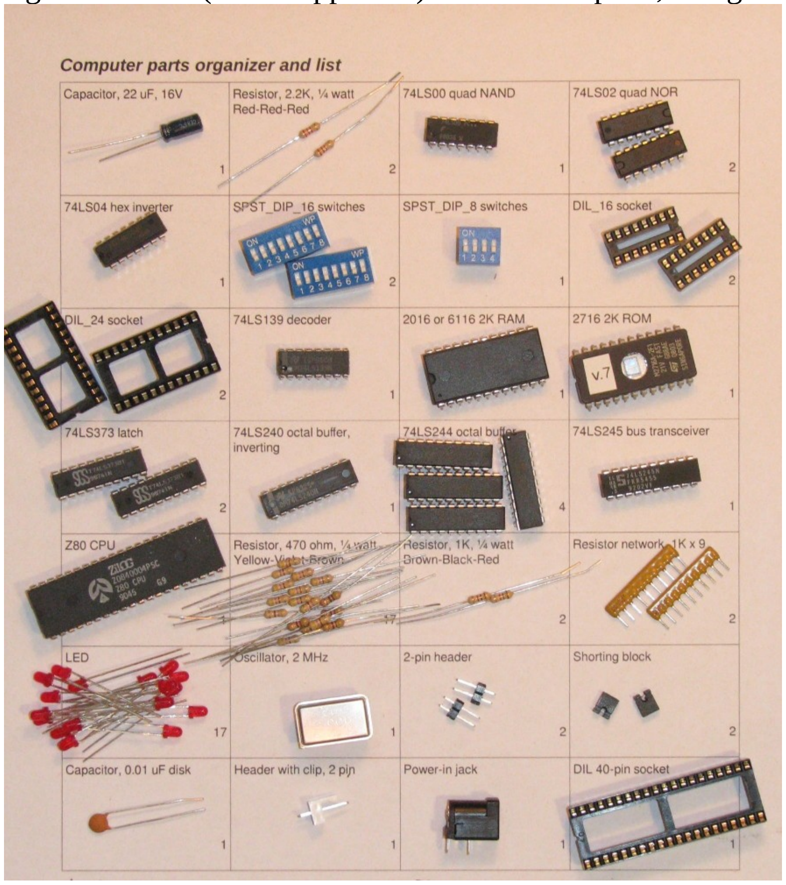
When you connect the display board to the computer with the ribbon cables, be very careful that the pins all go into holes. The connectors can be shifted one pin up or down, and still fit:
Make sure each pin goes into each hole.



Building the Computer



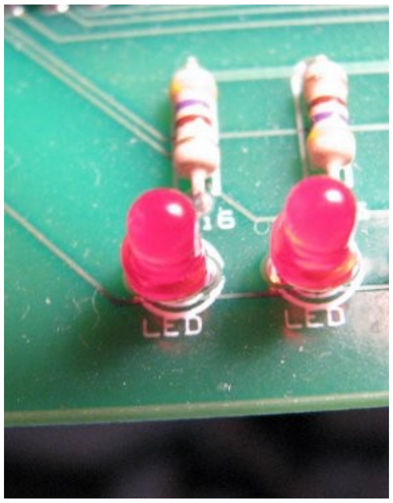
1. Use the parts organizer sheet (in the Appendix) to count the parts, and get familiar with them.



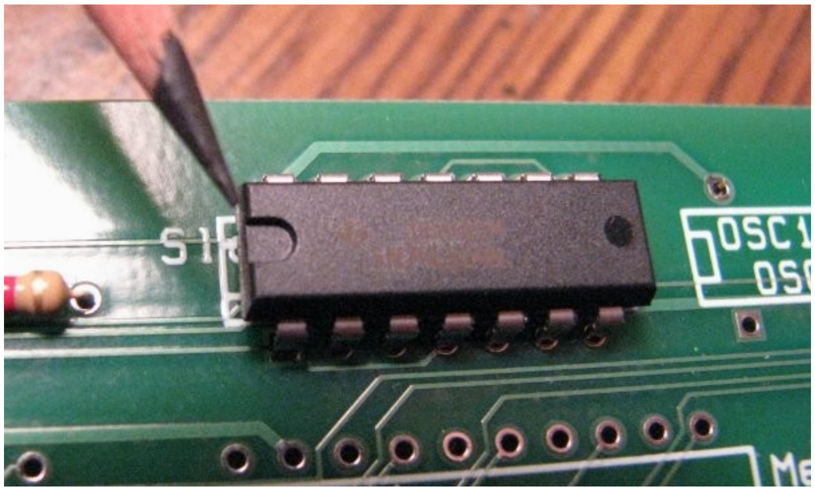
Computer parts organizer and list

Capacitor, 22 uF, 16V 1	Resistor, 2.2K, ¼ watt Red-Red-Red 2	74LS00 quad NAND 1	74LS02 quad NOR 2
74LS04 hex inverter 1	SPST_DIP_16 switches 2	SPST_DIP_8 switches 1	DIL_16 socket 2
DIL_24 socket 2	74LS139 decoder 1	2016 or 6116 2K RAM 1	2716 2K ROM 1
74LS373 latch 2	74LS240 octal buffer, inverting 1	74LS244 octal buffer 4	74LS245 bus transceiver 1
Z80 CPU 1	Resistor, 470 ohm, ¼ watt Yellow-Violet-Brown 17	Resistor, 1K, ¼ watt Brown-Black-Red 2	Resistor network, 1K x 9 2
LED 17	Oscillator, 2 MHz 1	2-pin header 2	Shorting block 2
Capacitor, 0.01 uF disk 1	Header with clip, 2 pin 1	Power-in jack 1	DIL 40-pin socket 1

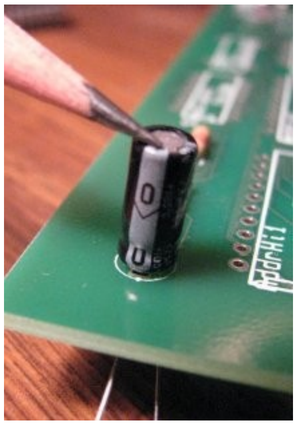
2. Most of the parts need to be placed in the board in the proper orientation:
    1. LEDs: the cathode (the short lead) is the more negative of the two leads, and is marked by the flat side of the flange on the plastic LED body. The flat side – short lead goes toward the RIGHT:



    2. ICs: The LEFT-hand side of each IC has a little cut-out:

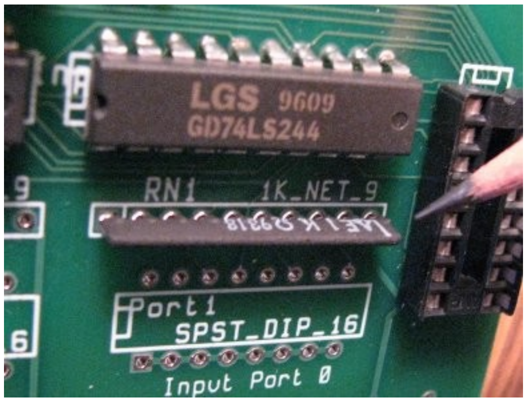


       This makes sure that Pin 1 will be in the LEFT lower corner.
    3. Electrolytic capacitors (little round can): The more negative lead is marked with a stripe. In this computer design, it doesn't really matter which side it goes in, but put it toward the LEFT anyway:



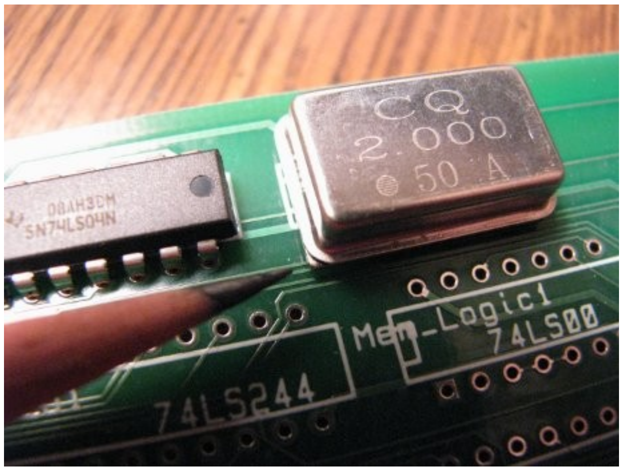
The small ceramic capacitor (disk) doesn't have any polarity.

4. IF YOU DON'T READ ANY OTHER INSTRUCTIONS READ THIS ONE. Resistor networks: these go in with the marked pin (the common pin) to the RIGHT:



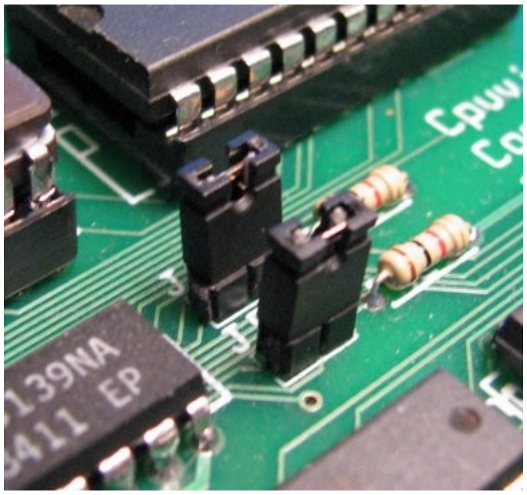
This placement is correct as shown in this photo. I made a mistake when I made the schematic module for this part, so you have to put it in backward, that is, with pin 1 to the RIGHT. I will fix this in a future version of the computer board.

5. The oscillator has a sharp corner and a dot that go to the LEFT.

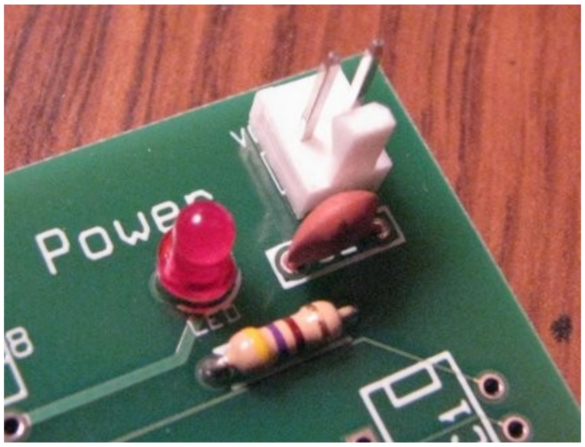


6. Resistors and ceramic capacitors (disks) do not have to be oriented.

7. The switches go in with “On” toward the top.
8. The two-pin headers for the jumpers are not oriented. However, be sure to put the shorting blocks on them before you try to run the computer:



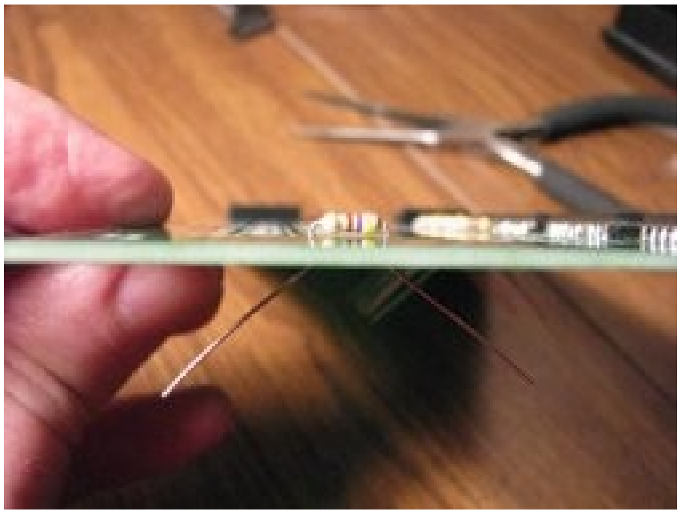
   The computer's on-board memory and input-output ports are disabled if the shorting blocks are removed.
9. The header in the right upper corner, for connection to the logic probe, is oriented with the white plastic tab toward the front of the board:



10. The 16-pin connectors have no orientation, but there is a cut-out toward one end, I usually put this toward the top of the board. Similarly, the 40-pin connector for the Z80 and the 24-pin connectors for the EPROM and RAM can go either way.

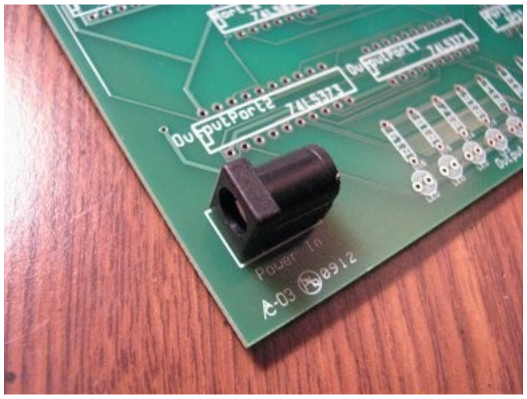
3. Once you are familiar with the parts and how they are oriented in the board you can start soldering. If you are interested in an educational building plan, see the supplemental material “Building by Sections”. If you just want to build it, then read on.
4. Review the section “Soldering Tips” before you start. There is no reason to solder the parts into the board in any particular order. Start with the low-profile parts first, then work up to the taller parts. This is because when you have the board upside down for soldering, the parts will sit flat against the top side of the board if you build from low- to high-profile. The parts from flattest to tallest: resistors, ICs, sockets for ICs and connectors, oscillator, LEDs, switches, headers (two-pin connectors), power-in jack, and electrolytic capacitor. You can bend the leads of the

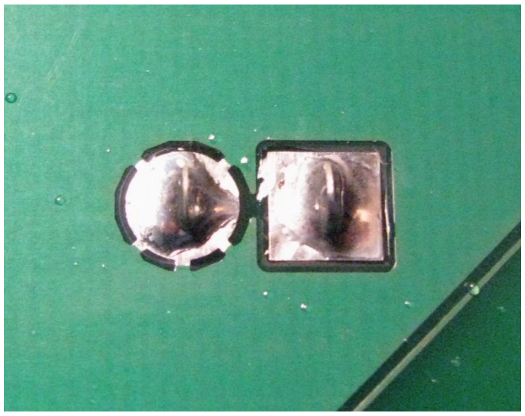
resistors and LEDs to help hold them while soldering.



Trim the leads off the resistors, LEDs and capacitors close to the board after you solder them. There is no need to trim the leads of the ICs, sockets, oscillator, or headers.

5. The power-in jack has tab connectors, but round holes. This is because to make slots would cost about $3.00 more per board, and slots are not necessary for a good connection. The tabs fit tight, you might have to apply a little pressure to get them to go in the hole. Then, just fill in the holes with solder:





6. Once you have finished making all the connections, inspect the board carefully to make sure you have not forgotten to solder any pins. Hold the board up to a bright light, looking at the bottom. Light will come through any open holes (of course the via holes, where circuit board traces go from one side of the board to the other, will be open, but all the holes with pins in them should be soldered shut). If you see open holes, solder them. Look for solder bridges. If everything looks good you are ready to do some test runs.

Binary, briefly

To test the computer you will need to know a little about binary numbers. Binary, or base-2, is the favored number system for computers because it is relatively easy to design circuits that have two stable states[6]. These states are 0 and 1, and in the computer you are building, are equivalent to 0 volts or ground (GND), and +5 volts or Vcc. These states are also called high and low, or clear and set depending on the situation.

Binary notation uses ones and zeros (1's and 0's) that are borrowed from the ordinary Arabic numerals. Each number is made up of a string of these numerals. The rightmost numeral occupies the one's place, same as in decimal notation. However, the next place to the left, instead of the ten's place, is the two's place. The next place is the four's place, and the eight's place is next to that. Each place in the number will be double the place to its right. The value of the number, as in decimal, is the sum of the value of each place:

Binary 1100 = (1 x 8) + (1 x 4) + (0 x 2) + (0 x 1) = 8 + 4 + 0 + 0 = decimal 12

Long binary numbers can confuse the eye, so there is a shorthand notation that is used to write them. This system is hexadecimal, or base-16, number system. Hexadecimal notation needs 16 numerals. It borrows its first 10 from the Arabic numerals used in the decimal system. Its last 6 are the letters A through F. Both upper and lower case can be used for the letter-numerals. There is a close connection between hexadecimal and binary notation. Here is the table:

6 Some early computers, such as ENIAC, used decimal numbers, and had circuits with 10 stable states.

Binary	**Hexadecimal**	**Decimal**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A or a	10
1011	B or b	11
1100	C or c	12
1101	D or d	13
1110	E or e	14
1111	F or f	15

The four-bit binary numbers in the table are called nybbles. Each nybble can be written as one hexadecimal numeral. Note the use of leading zeros in the nybbles. This is characteristic of the way numbers are written when working with computer systems at the hardware and machine code level, because we are dealing with number widths defined by hardware. The Z80 has 8-bit binary registers and data words, so we are usually dealing with two-nybble numbers called bytes. Typically, binary numbers are written with a space between the nybbles in a document like the one you are reading, to allow the eye a little relief from the long strings of 1's and 0's. Also, it is easy to convert the binary numbers to hexadecimal when they are written this way. For example, consider this 16-bit binary number:

1101 1010 0011 1001

By referring to the table above, you can quickly write this as a hexadecimal number:

DA39

In order to make clear which number system is being used, we will add some extra information to the numbers. The convention I will use in this manual, which is used widely, and in most assemblers, is to add a leading “zero x” (0x) to a hexadecimal number, like this:

0xDA39

Computer programmers like to use this notation, because the leading zero tells the parser that what follows is a number, not a word. Another convention is to use a trailing lower-case “h” for hexadecimal numbers:

DA39h

Binary numbers are sometimes designated with a trailing lower-case “b”, and decimal numbers with a lower-case “d”:

1101 1010 0011 1001b
55865d

The problem here is that b and d are both hexadecimal numerals, so we need to be a little careful. It is always the responsibility of the writer to make sure that the reader knows which number system is being used. I like to write out the words:

binary 1101 1010 0011 1001
decimal 55865

Once you understand binary and hexadecimal notation, the next challenge is to learn binary arithmetic in the computer environment, where numbers have defined widths because of the hardware (registers and memory locations) they are contained in. Addition in such an environment can lead to odd outcomes if the sum exceeds the maximum width of the register. For example, in the Z80 CPU:
Binary: Decimal equivalent:

```
 1111 1100      252
+0010 0101     + 37
 0010 0001       33
```

Got that? The bit that is carried-out cannot fit in the 8-bit register, so the result is not what you might have hoped for. Fortunately, the CPU keeps track of the carry-out in a one-bit register called the carry flag. We can check the carry flag after the addition operation, to make sure we are still in control of our arithmetic.

Binary subtraction, and negative binary number conventions, are particularly challenging. A full treatment of this subject is beyond the scope of this manual, but here is a brief treatment of signed 8-bit binary numbers.

The convention for signed binary numbers is that every number with a 1 in the leftmost position is a negative number. There are 128 possible 8-bit numbers with a 1 in the left-most place, and 256 possible 8-bit numbers in all (including zero), so an 8-bit byte can encode signed numbers from -128 to +127. The values of the negative numbers are determined by thinking of the 8-bit register as a kind of odometer. If you count backwards from +2, you get these patterns:

Binary:	Decimal equivalent:
0000 0010	+2
0000 0001	+1
0000 0000	+0
1111 1111	-1
1111 1110	-2
1111 1101	-3
.	
.	
.	
1000 0001	-127
1000 0000	-128
0111 1111	+127
0111 1110	+126

An easy way to create a negative number (remember it can only be -1 to -128 for an 8-bit byte) is by a method called two's-complement negation. Here is the method. We will create -2 for an example.

First, write the positive binary number you wish to negate:

```
0000 0010
```

Complement it. That is, change each 0 to a 1, and each 1 to a 0:

```
1111 1101
```

Now, add 1:

```
 1111 1101
+0000 0001
 1111 1110
```

That's it. Note that 1111 1110 binary is negative 2, looking at the “odometer” list above.

Does it act like -2? If we add it to a positive value, say +12, the result should be +10:

Binary: Decimal:

```
 0000 1100     + 12
+1111 1110     +(-2)
 0000 1010     + 10
```

So it behaves as it should. Adding a signed, 8-bit integer is used in Z80 assembly language in relative jump instructions, such as the DJNZ disp instruction.

Testing the Computer

The 2K EPROM contains a few simple programs designed to test the computer system (see the 2K EPROM listing in the Appendix for details). Apply +5V DC regulated[7] to the computer board, and set the Reset switch On (up). To access a program, you put the 16-bit binary starting address onto the input port switches while the computer is in reset (Reset switch On). Then, take the computer out of reset (turn the Reset switch Off). It will jump to the address location specified by the input port switches and start execution from there. These are the test programs and their starting addresses:

<u>Port reflector, address 0x001F (binary 0000 0000 0001 1111)</u>
This program gets a data byte from each input port, and displays it on each output port. If this works correctly, then you know that the input and output ports work, and that the Z80 CPU is communicating with the 2K EPROM properly. This program does not test the 2K static RAM.

<u>Simple counter, address 0x002A (binary 0000 0000 0010 1010)</u>
In this program, the Z80 increments the value in the 8-bit register A and displays the result on output port 0. The output port display will go from 0 to 255 (binary 0000 0000 to 1111 1111) over and over again. It is useful when the bus display board is attached to watch how the CPU operates the bus signals when the slow clock is on. With the fast clock the bus display and outputs are a blur, but if you reset the computer while this program is running, port 0 will display a random number between 0 and 255 decimal.

<u>Count to a million, address 0x0032 (binary 0000 0000 0011 0010)</u>
This program counts down 16 times by decrementing the A register, then increments the 16-bit register pair HL and displays the result on the output ports 1 and 0. The result is 16 x 65, 536 = 1,048,576 operations for a full cycling of the output. It is impressive to run this program with the slow clock, which seems to take forever to increment the output once, and compare that to the 2MHz clock, which goes through the whole count in a second or two. This gives a visible demonstration of the speed of the computer.

<u>Program loader, address 0x0046 (binary 0000 0000 0100 0110)</u>
This program takes bytes from input port 0 and puts them in RAM, one after the other. After you start the program, place a byte intended for input on the port 0 switches, then close and open the RIGHTmost switch on input port 1. When you close this switch, the byte is written into RAM, and the output port 1 lights all come on. When you close the switch, output port 0 will show the byte you have written, and port 1 will show the low-byte of the RAM address where the byte was place. Repeat this process for each byte in your program. When all the bytes have been entered, close the LEFTmost switch on input port 1. The program will then jump to the beginning of RAM and execute the program there. See the appendix for some program examples. For example, to load the short program RAM_test_1.asm, one would follow these steps:

1. With the computer in reset (Reset switch On), select the Fast clock, and put 0x0046 (binary bit pattern 0000 0000 0100 0110) on the input port switches.
2. Set the Reset switch Off. The computer will now jump to the address 0x0046 in ROM and start executing the code there, which is the Program Loader (see the ROM listing 2K_ROM_6.asm

7 This project requires a +5V **<u>regulated</u>** DC power supply capable of at least 2000 mA (i.e., a 10 watt power supply). An unregulated power supply will not work properly and may damage the system.

in the appendix).

3. The RAM_test_1.asm program has been assembled for you. On the listing, you can see the machine code bytes listed next to the RAM addresses in which they are to be placed. The first instruction in the program, `ld a,005h,` means “load the A register with the hexadecimal value 05”. This instruction is assembled to create the machine code 0x3e, 0x05. These bytes need to be placed into RAM addresses 0x0800 and 0x0801 in order for the processor to execute this instruction.
4. The program loader program will place bytes into RAM one after the other starting at address 0x0800 as you enter them into the input port 0 switches. You enter the bytes this way:
    1. Place the byte to be entered on the port 0 switches. The first byte is 0x3e. The binary bit pattern for this byte is 0011 1110.
    2. Close the RIGHTmost switch on input port 1. The output port 1 LEDs will all light up, signaling that the byte has been entered into RAM
    3. Open the RIGHTmost switch on input port 1. The output port 1 LEDs will show the lower byte of the RAM address where the program code byte was entered (binary 0000 0000 for the first byte), and the byte you entered will show on the output port 0 LEDs (bit pattern binary 0011 1110)
    4. Place the next byte to be entered (0x05, binary pattern 0000 0101) on the input port 0 switches.
    5. Close and open the RIGHTmost switch on input port 1 as before. The output port 1 LEDs with light when you close the switch, and then will display the low-order byte of the RAM address where this byte was entered, now binary 0000 0001. The byte 0x05, which is the code you entered, will be displayed on the port 0 LEDs
    6. Do this for all the bytes in the program.
    7. After you have entered the last byte (76h, binary pattern 0111 0110), close the LEFTmost switch on input port 1. This will cause the program loader program to jump to the first location in RAM (address 0080h) and execute the code it finds there. This will be the simple RAM test program you entered.
    8. This program puts the bytes 05h and 0Ah on output ports 0 and 1, respectively, then halts. The halt instruction causes the CPU to cease program execution until it is reset[8].
    9. If you reset the CPU, you can switch to the slow clock, and put the address 0x0800 (the start of RAM) on the input ports. When you take it out of reset, it will jump to the beginning of RAM, and execute the program you entered there. Then you can see what is happening on the buses during the halt state, with the slow clock running.

<u>Memory test, address 0x0074 (binary 0000 0000 0111 0100)</u>

This program places a byte value into each location in the memory, starting at the beginning of the 2K RAM, and reads it back. It tests to see if it gets the same value back. If it gets a different value, that is an indication either that the RAM is not working properly, or that the program has finished going through the RAM, and is now addressing the 2K EPROM, which cannot be written by the computer.

8 In the halt state, the CPU continues to run. It fetches instructions, and ignores them, executing no-operations (NOP, opcode 0x00) internally. After each instruction fetch, the CPU will execute a memory refresh cycle, which you can see on the bus display if you run the slow clock, or with a logic probe if the fast clock is selected. The memory refresh cycles are for systems with dynamic memory. Our system has static memory, so the refresh cycles are not needed. A refresh cycle can be identified by the MemReq signal coming on alone, without a Read or Write signal accompanying it, and by a low value on the Z80 pin 28. The lower 7 bits of the address bus can be used as a row address for refreshing dynamic memories during a refresh cycle.

The program displays the address on the output ports. If the resulting address is binary 0001 0000 0000 0000, which is equivalent to 4K, this is an indication that every address in the RAM is working properly. If it is some lesser value, that is a sign that all is not well.

Peek, address 0x008D (binary 0000 0000 1000 1101)
This program allows you to look at any address in memory and see what is stored there. You put the 16-bit address you want to “peek” into on the input ports, and the contents of that address location will be displayed on output port 0. For example, if you put the address 001Fh on the input port switches, the binary pattern 1101 1011, or DBh will be displayed on output port 0. This is the first machine code byte in the Port reflector program.

Poke, address 0x0099 (binary 0000 0000 1001 1001)
This program allows you to put (“poke”) a byte anywhere in RAM. It is similar to the program loader, except you have to put the address in for each byte you enter. After you start the program, place the high-order byte of the address on the input port 0 switches, then close and open the RIGHTmost switch on input port 1. Do the same for the low-order address byte next. The address will show on the output ports. Then, enter the data byte to be placed into that location. When you close the switch, this byte is written into RAM. When you open the switch after entering the byte, the program starts over so you can enter another byte if you want. You can use Peek to verify that the data byte was written. This program can only enter bytes into RAM (addresses 0800h to 0FFFh). It won't be able to store anything in ROM, because the ROM chip cannot be written while it is plugged into the computer. It needs a separate programmer with special timing signals and +25V to program.

If all this works, congratulations, you have built your own working Z80 computer!

Z80 Programming

The Z80 CPU, like all stored-program computer processors, gets its instructions from the system memory. These instructions are binary numbers that code for the operations the programmer wants the CPU to perform. Operation codes are called opcodes for short, and the set of these numbers is the machine code or machine language of the processor. Some operations will take additional numbers, or operands, which are 8- or 16- bit numbers, 8-bit port addresses, or 16-bit memory addresses.

Since the opcodes are just numbers, the Z80 designers created English-derived abbreviations and short words, called mnemonics, that are associated with the opcodes. These mnemonics, or “aids to memory”, help a human programmer write a program without having to continually look up every opcode. After the program is written, each mnemonic with its associated operands can be easily assembled into one machine language statement. That is why this type of programming language is called assembly language. Each processor has its own assembly language, which depends on the structure of the processor. You can easily assemble short programs “by hand” by referring to the opcode tables, but for long programs there are assembler programs that will do this for you.

To really learn Z80 assembly language programming you would need a semester course with a fat textbook[9]. This processor understands over 150 different instructions. You can get a complete table of instructions in the Z80 datasheet, or in the Z80 CPU Users Manual from Zilog, Inc. (http://www.zilog.com/docs/z80/um0080.pdf). There are excellent resources on the Internet for learning assembly language, and free assembler programs (see Resources). However, since most programs are written with only a subset of the whole instruction set, you can get started without much study. The function of many of the less-used instructions can be duplicated with a few of the common instructions.

I have decided to show here the instructions I have used in the programs included in this instruction manual, which are in the first part of the 2K ROM and in the example programs for loading into RAM. Program listings are in the Appendix. These programs use about 30 of the available Z80 instructions. You can write any program with this subset of instructions, or with even fewer if you want. The original stored-program computer had only 7 instructions, so 30 is plenty.

Before we introduce the instructions we need to introduce the processor. What exactly does it do? The Z80, like any computer processor, does one small thing at a time, very fast. The small things are simple operations (add, subtract, and logical operations like AND and OR), data movement, and program flow control that can respond to changes in conditions or the results of calculations. The operations are performed on data held in special locations inside the processor called registers. The Z80 has two sets of 14 registers, but we will only use a few of these. The ones we will use are A and B, and H and L[10]. Each of these registers holds one 8-bit number. The A register is the main register that is involved in operations and data movement. Whenever you want to get a number from memory to operate on, you need to load it from memory into the A register. Once it is in the A register, there are other instructions that allow you to copy this number to the other registers. Also, the A register will usually hold the result of an arithmetic or logical operation. For example, the instruction ADD A,B (opcode 1000 0000 binary,

9 I learned by studying Z80 Assembly Language Programming by Lance A. Leventhal, Osborne/McGraw-Hill, 1979, 619 pages

10 Register designations and mnemonics in Z80 assembly language can be either upper or lower case (case insensitive). This is not true of labels, however.

or 80h), will add the contents of the B register to the contents of the A register, with the result replacing the original contents of the A register. For this reason, the A register is also called the accumulator, in keeping with the original total-keeping registers of early computers like the ENIAC.

The Z80 has a series of one-bit registers, or flip-flops, each of which is set (made equal to 1), or not set (made equal to 0) by the operations as they are performed. These are the processor flags, and are used to make decisions about the program flow. For example, the JP NZ instruction ("jump if not zero", opcode 1100 0010 binary, or C2h) will cause the program to jump to a new address if the zero flag (Z for short) is not set by the previous operation, often a subtraction. The other flag used in the programs in this manual is the minus flag, or M for short, which is set (becomes 1) if the previous operation resulted in a negative number. A third flag, the carry flag, or C for short, is often used, but none of the programs in ther first part of the ROM use it. It is used in the second part of the ROM which contains the serial port instructions.

The H and L registers are special in that some instructions use the pair to designate a 16-bit address. For example, the instruction LD (HL),A (opcode 0111 0111 binary, or 77h) will place (load) the contents of the A register into the memory location indicated by the 16-bit value held in the H and L registers. The H and L stand for "high" byte and "low" byte of the address. The parentheses around HL indicate that this is will be treated as a memory address. So, if H is 08h, and L is 40h, then this instruction will place the contents of register A into memory location 0840h.

There is also a 16-bit Program Counter (PC) register, that holds the address of the next instruction to be fetched. The PC is set to 0x0000 when the computer is reset, and starts executing code from there once it is taken out of reset (made to run). The default behavior is that the PC is incremented by the number of bytes in the last instruction fetched, so program execution goes on one instruction after another, from low memory addresses toward high. The program flow control instructions (JP, or jump) alter this flow of execution by changing the PC. This causes the CPU to fetch instructions from new areas of memory, or "jump" to new code.

Here is the subset of instructions that are used in the first part of the ROM and in programs in this manual (with a few extras thrown in for completeness):

Arithmetic and Logical Operations				
Mnemonic	**Opcode**	**Operand**	**Operand**	**Meaning**
AND number	E6	number		Logical AND A and number, result to A
AND B	A0			Logical AND A and register B, result to A
OR number	F6	number		Logical OR A and number, result to A
OR B	B0			Logical OR A and register B, result to A
XOR number	EE	number		Logical exclusive-OR A and number, result to A
XOR B	A8			Logical exclusive-OR A and register B, result to A
CPL	2F			Logical complement A (change 1s to 0s, and 0s to 1s)
ADD[11] A,number	C6	number		Add number to A, result to A
ADD A,B	80			Add B to A, result to A
ADC A,number	CE	number		Add number and carry to A, result to A
ADC A,B	88			Add B and carry to A, result to A
SUB number	D6	number		Subtract number from A, result to A
SUB B	90			Subtract B from A, result to A
SBC A,number	DE	number		Subtract number and borrow from A, result to A
SBC A,B	98			Subtract B and borrow from A, result to A
CP number	FE	number		Subtract number from A, leave A unchanged (flags change)
CP B	B8			Subtract B from A, leave A unchanged (flags change)
INC A	3C			Increment the 8-bit value in A (add 1)
INC HL	23			Increment the 16-bit value in HL (add 1)

11 The “official” assembly language reference has “ADD A,number” for addition to the A register, but “SUB number” for subtraction from the A register. Most assembler programs will assemble “ADD A,number” and “ADD number” to the same opcode. Same with “SUB number” and “SUB A, number”.

Program Flow Control Operations				
Mnemonic	**Opcode**	**Operand**	**Operand**	**Meaning**
DJNZ disp	10	disp		Decrement B, jump disp[12] if not zero (operation used as a counter)
HALT	76			Stop CPU execution. Only interrupt can restart
JP (HL)	E9			Jump to the address indicated by HL
JP addr	C3	addr lo	addr hi	Jump to address (unconditional jump)
JP C,addr	DA	addr lo	addr hi	Jump to address if carry flag set (C = 1)
JP NC,addr	D2	addr lo	addr hi	Jump to address if carry flag cleared (C = 0)
JP Z, addr	CA	addr lo	addr hi	Jump to address if zero flag set (Z = 1)
JP NZ, addr	C2	addr lo	addr hi	Jump to address if zero flag cleared (Z = 0)
JP M,addr	FA	addr lo	addr hi	Jump to address if minus flag set (M = 1)
JP P,addr	F2	addr lo	addr hi	Jump to address if minus flag cleared (M = 0)

12 The displacement is an 8-bit signed value that is added to the program counter if the jump is executed, that is, if the zero flag is not set (NZ). After the DJNZ instruction is fetched, the program counter will be pointing to the address of the first byte of the next instruction. The DJNZ instruction is two bytes long. So, to jump back to the DJNZ instruction, the displacement value needs to be -2, or FE hexadecimal. See the discussion on the page following the operation table.

Data Movement Operations				
Mnemonic	**Opcode**	**Operand**	**Operand**	**Meaning**
LD (addr),A	32	addr lo	addr hi	Load memory location with contents of A
LD (HL),A	77			Load memory location indicated by HL with A
LD A,(addr)	3A	addr lo	addr hi	Load A with contents of memory location
LD A,(HL)	7E			Load A with memory location indicated by HL
LD A,B	78			Load A with contents of B
LD A,H	7C			Load A with contents of H
LD A,L	7D			Load A with contents of L
LD A,number	3E	number		Load A with an 8-bit number
LD B,A	47			Load B with contents of A
LD B,number	06	number		Load B with an 8-bit number
LD H,A	67			Load H with the contents of A
LD H,number	26	number		Load H with an 8-bit number
LD HL,number	21	num lo	num hi	Load HL with a 16-bit number
LD L,A	6F			Load L with the contents of A
LD L,number	2E	number		Load L with an 8-bit number
OUT (port),A	D3	port		Place the 8-bit contents of A onto an output port
IN A,(port)	DB	port		Place the 8-bit value from port into the A register

Some operations take one- or two-byte operands. The two-byte operands are read by the CPU in the order low-order byte first, then the high-order byte. In other words, the **low** order byte of a two-byte operand will be in the **low**er-memory address location, and the **high**-order byte will be in the **high**er memory address location. This is called little-endian byte order[13]. This is the meaning of the “addr lo” and “addr hi”, or “num lo” and “num hi” in the above table of operations. The program listings in the Appendix will show this clearly. Numbers can be either 8- or 16-bit. Memory addresses are 16-bit. Port addresses are 8-bit.

There is one strange operand above, labeled as “disp”, for the DJNZ instruction. This is a special instruction used for loops. This instruction decrements (decreases by one) the B register, and checks the zero flag. If the zero flag is not set, meaning the B register contains a non-zero value, the instruction uses the disp (displacement) value to calculate a jump relative to the current program counter. The disp value is a signed, 8-bit number. The jump works this way. After the DJNZ instruction and its operand are fetched, the program counter (PC) will be pointing to the memory location that is after the disp location. The displacement value in the DJNZ instruction in the Program Loader program, 0xFE, is negative two. When this is added to the program counter, it is brought back two locations, and will point to the DJNZ instruction again, making a loop that is used for a delay. The instruction keeps decrementing the B register until it is zero. Then, the zero flag is set (= 1), and the jump is not made. Instruction execution proceeds from the location after the disp value.

In addition to the opcodes, if you are using an assembler program, that program may use pseudo-opcodes called assembler directives. You can see the use of these in the program listings in the Appendix. The EQU directive makes a label equal to a value, so you can use the label as an operand in later instructions. The ORG directive tells the assembler where to start the memory addresses it will use. The DEFM directive (define message), puts the ASCII code for a string of alphanumeric characters into the memory. The similar DEFB directive (define byte) puts one or more bytes into the memory.

An assembly language program in written using a word processor, or piece of paper, in four columns. These are the labels, the opcode, the operand(s), and the comments. The opcode and operand(s) is the assembly language. Sometimes line numbers are added as another column, but are not necessary for small programs. As an example, we will write a simple program that takes an 8-bit number from each input port, adds them, and displays the result on the output ports.

```
Add_Program:   ld   a,00h          ;Clear outputs to start
               out  (0),a
               out  (1),a
Get_addends:   in   a,(0)          ;Get 8-bit addends
               ld   b,a            ;Store one in B register
```

13 See the Wikipedia article on endianness – http://en.wikipedia.org/wiki/Endianness

```
in   a,(1)          ;Get the other
add  a,b            ;Add them
out  (0),a          ;Output the result
ld   a,00h          ;Clear port 1 LEDs
out  (1),a
jp   nc,Get_addends ;All done if no carry
ld   a,01h          ;If carry, put 1 on port 1
out  (1),a
jp   Get_addends    ;Start again
```

Note the labels end with a colon (:). This tells the assembler program that the preceding characters are a label and not an instruction. The label itself, without the colon, is used as an operand in the jump instructions.

With the program written, we need to assemble it to create the machine code for the Z80 CPU. We can create a sheet for this with two columns to the left of the assembly language. These columns are for the memory addresses, and the machine code which will occupy the memory locations. In the listings in the Appendix, the assembler program I used has placed the machine code bytes for each instruction on one line, even if there are two or three bytes. But, perhaps it will be easier for hand assembly to place each byte on a separate line. We can help a little by making a table with the RAM memory addresses in it already. There is a blank table in the Appendix that you can print out and use to assemble programs. Here are the top few rows:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
0800				
0801				
0802				

The table can have as many addresses as we need, up to the full 2K of RAM. We are starting at address 0x0800 because that is the beginning of RAM in the CPUville Z80 computer. If we were using an assembler program we would tell it to start assembling machine code for address 0x0800 with the ORG 0x0800 statement.

To start, put in the first assembly language statement:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
0800		Add_Program:	ld a,00h	;Clear outputs to start
0801				
0802				

Now, look up the opcode for the "LD A,number" instruction in the opcode table on page xx and put it in the first location:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
0800	3E	Add_Program:	ld a,00h	;Clear outputs to start
0801				

Then put the operand for this instruction in the next memory location. The operand is an 8-bit number (a byte), in this case the port address 00h:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
`0800`	`3E`	`Add_Program:`	`ld a,00h`	`;Clear outputs to start`
`0801`	`00`			
`0802`				

In other words, the instruction LD A,00H is assembled into the machine code 3Eh, 00h. The label `Add_Program,` which is equivalent to 0800h, indicates where the program starts. It is not used in this assembly, but it is helpful to indicate the program start address to human eyes. Put in the next instruction, and then its opcode and operand in the next available memory locations:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
`0800`	`3E`	`Add_Program:`	`ld a,00h`	`;Clear outputs to start`
`0801`	`00`			
`0802`	`D3`		`out (0),a`	
`0803`	`00`			
`0804`				

Continue until you have assembled down to the first jump instruction:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
`0800`	`3E`	`Add_Program:`	`ld a,00h`	`;Clear outputs to start`
`0801`	`00`			
`0802`	`D3`		`out (0),a`	
`0803`	`00`			

0804	D3		out (1),a	
0805	01			
0806	DB	Get_addends:	in a,(0)	;Get 8-bit addends
0807	00			
0808	47		ld b,a	;Store one in B register
0809	DB		in a,(1)	;Get the other
080A	01			
080B	80		add a,b	;Add them
080C	D3		out (0),a	;Output the result
080D	00			
080E	3E		ld a,00h	;Clear port 1 LEDs
080F	00			
0810	D3		out (1),a	
0811	01			
0812	D2		jp nc,Get_addends	;All done if no carry
0813				
0814				
0815				

The label `Get_addends` is the operand of the JP NC instruction. By looking at the memory location corresponding to this label in the previously assembled code, we can see that this label points to memory address 0806h. We need to place this 16-bit value into the memory after the opcode for JP NC, with the **lower** byte of the operand in the **lower** byte of memory, and the **higher** byte of the operand in the **higher** byte of the memory (little endian notation) like this:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
0800	3E	Add_Program:	ld a,00h	;Clear outputs to start
0801	00			

0802	D3		out (0),a	
0803	00			
0804	D3		out (1),a	
0805	01			
0806	DB	Get_addends:	in a,(0)	;Get 8-bit addends
0807	00			
0808	47		ld b,a	;Store one in B register
0809	DB		in a,(1)	;Get the other
080A	01			
080B	80		add a,b	;Add them
080C	D3		out (0),a	;Output the result
080D	00			
080E	3E		ld a,00h	;Clear port 1 LEDs
080F	00			
0810	D3		out (1),a	
0811	01			
0812	D2		jp nc,Get_addends	;All done if no carry
0813	06			
0814	08			

Sometimes, if we are assembling code that jumps ahead, we will not know what the target address of the label will be. We can put placeholder bytes into the jump instruction operand locations until we have assembled up to the target, then go back and put in the proper values once we know what the address will be. Here is the finished assembly of the addition program:

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
0800	3E	Add_Program:	ld a,00h	;Clear outputs to start
0801	00			

0802	D3		out (0),a	
0803	00			
0804	D3		out (1),a	
0805	01			
0806	DB	Get_addends:	in a,(0)	;Get 8-bit addends
0807	00			
0808	47		ld b,a	;Store one in B register
0809	DB		in a,(1)	;Get the other
080A	01			
080B	80		add a,b	;Add them
080C	D3		out (0),a	;Output the result
080D	00			
080E	3E		ld a,00h	;Clear port 1 LEDs
080F	00			
0810	D3		out (1),a	
0811	01			
0812	D2		jp nc,Get_addends	;All done if no carry
0813	06			
0814	08			
0815	3E		ld a,01h	;If carry, put 1 on port 1
0816	01			
0817	D3		out (1),a	
0818	01			
0819	C3		jp Get_addends	;Start again
081A	06			
081B	08			
081C				

The same program has been assembled by an assembler program, and is listed at the end of the Appendix. Run the program loader, and enter the machine code bytes one at a time into RAM. When you execute the program, it should add the two one-byte addends on the input ports, showing sum on the port 0 LEDs, and the carry-out in the low-order bit of the port 1 LEDs.

Computers in General

A computer is a kind of universal information processor. The first computers were people. The kind of problem human computers worked on were iterative calculations that cannot be done with simple calculators. An example of this type of computation is a ballistics trajectory. A projectile, after it is fired from a gun, experiences several forces that determine its flight path through the atmosphere, and ultimately, where it lands. Some of the forces depend on its velocity. But, these forces also change its velocity, which changes the forces, and so on, meaning that one cannot just write a simple equation that describes the entire trajectory. The trajectory has to be broken down into small pieces, each a fraction of a second long. The forces, and the changes in velocity and position are calculated for each time interval, one at a time, starting from the initial position and velocity, until the projectile “lands”. These kinds of computations are called numerical integrations. Human computers would do them by hand on paper spreadsheets, using a mechanical calculator to do the additions, subtractions, multiplications, divisions, and square roots as required for each interval. They are not hard to do but are very tedious, and human computers get very fatigued with them, and often make simple mistakes that ruin the whole computation. They have a machine-like feel to them.

During World War II there was a great increase in the need for ballistics trajectory computations. Human computers could not keep up with the demand. The first electronic general-purpose computer, ENIAC (Electronic Numeric Integrator and Computer) was built for this purpose. However, it was created as an electronic copy of a paper spreadsheet, with a bank of machines called accumulators. Each accumulator was like one column of a spreadsheet. It would hold one variable, and alter the variable by adding, subtracting, multiplying or dividing with a variable held in another accumulator, according to the connections that had been made between them. If your computation needed more columns, you needed to add more accumulators. To program ENIAC, technicians had to re-connect the accumulators with plugs and wires in new patterns to match the computation being performed.

It was the ENIAC designers who realized that it would be much simpler to create a computer with a single accumulator, and then store the contents of the accumulator in a storage unit, what we call computer memory, after a calculation had been done. When the stored variable was needed for another iteration it could be retrieved from the memory by the accumulator. Someone also realized (history has forgotten who) that the instructions for the computation, what we now call the program, could also be stored in the same memory as numbers that coded for the operations. These numbers are called operation codes, or opcodes for short. An electronic unit would interpret the opcodes, and cause the computer to perform the calculations desired. The electronic unit that combines the accumulator, the arithmetic-logic unit for calculations, and the instruction decoding circuitry is called a processor. The computer system is composed of the processor, the memory, and input/output devices – paper tape and teletypes in early computers.

The idea for a stored-program computer was circulated in a draft manuscript written by the famous mathematician John von Neumann. For this reason, the stored-program computer is often said to have von Neumann architecture. In truth, it was not von Neumann who came up with the idea, but the draft manuscript he authored was the main vehicle of the spread of the idea.

The first true electronic stored-program “von Neumann” computer was built at the University of Manchester, UK. Named the Small Scale Experimental Machine (informally the “Baby”), it ran the first true computer program on June 21, 1948. This program was a highest-factor routine, the basic routine that can determine if a number is prime or not. In honor of this achievement, the appendix of this manual has a highest-factor program that you can load and run on your finished Z80 computer. A highest-factor program is an example of an iterative computation that cannot be done by a simple calculator. Only a computer – human or machine – can tell you if a given number is prime.

The CPUville Z80 Computer System

The computer system in this kit is very simple. It is simple on purpose, so that it will be easy to understand. It is probably the simplest system that can be called a general-purpose computer. It is really a microcomputer, which is a computer that has a processor that is a single component, in this case the Z80 CPU (Central Processing Unit). In early computers, the processor was built out of separate parts, by hand. Early computer processors used relays (automatic switches in which the electrons flow though a wire), vacuum tubes (automatic switches in which the electrons flow through a vacuum), or transistors (automatic switches in which the electrons flow through a solid semiconductor material). These automatic switches were assembled into modules, each using just a few switches, that acted as logic gates. Logic gates are the basic building blocks of digital computers. You can see on the memory and input-output port schematics of the CPUville Z80 computer the use of a few simple logic gates (AND, OR, and NOR) to make decisions electronically. More complicated networks of logic gates can perform addition, and act as memory circuits.

When the integrated circuit was invented one of its first uses was to create logic gate chips. These can be built into a computer processor, as you can see on my main web site, cpuville.com. It wasn't long (late 1970's) before multiple logic gates were assembled onto single integrated circuit chips to create single-component computer processors and computer memories, and the microcomputer was born.

The Zilog Z80[14] is one of the early 8-bit microprocessors. The 8-bit designation means that data flows into and out of the microprocessor 8-bits at a time. Computer scientists made fun of such a small data word width, when “real” computers of that day (late 1970's) had data word widths of 32-bits or more. However, it is simple to assemble larger words out of multiple 8-bit “bytes”, so the only handicap the 8-bit computer has is slower speed. If you need to, you can calculate pi to 100 decimal digits using an 8-bit microcomputer, or draw a Mandelbrot set, or anything else a 32- or 64-bit computer can do. It just takes a much longer time. 8-bit microcomputers became popular with consumers because 8-bits is plenty of word-width to do word processing, create video games, and do number-crunching with the types of numbers that a consumer would most likely have to deal with. Popular early microcomputers using the Z80 were the Tandy Radio Shack TRS-80, and the Timex-Sinclair ZX80. The Texas Instruments graphing programmable calculator TI-81 (and successors) used the Z80. The first generation of Nintendo Game Boy video game systems used a Z80 processor.

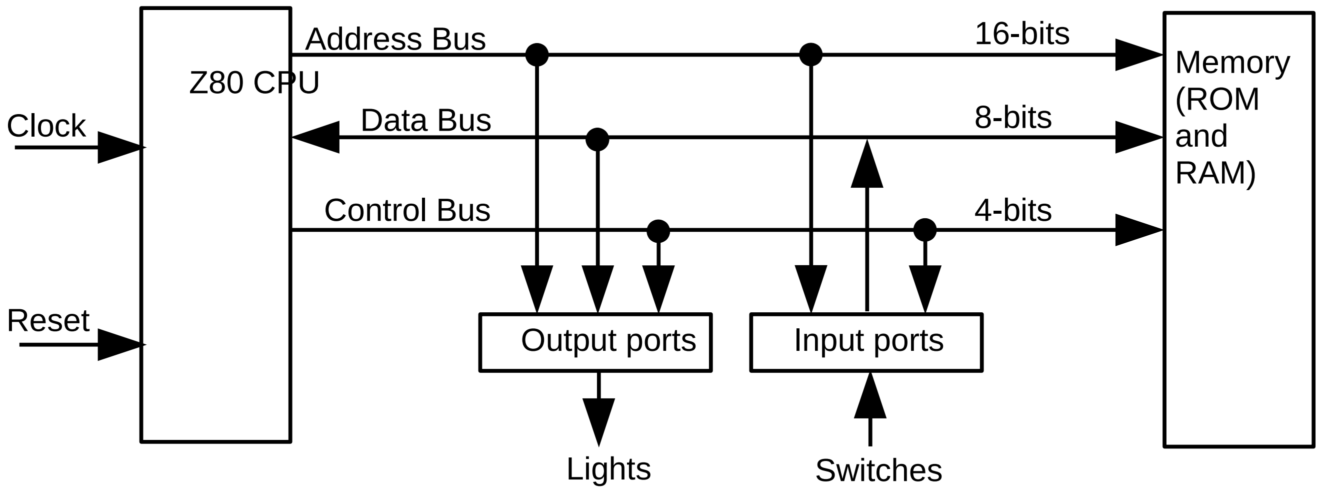
The CPUville computer kit uses a Z80 processor because it is easy to understand how it fits into a system, and it is easy to design and build with. There are lots of books and web sites about using the Z80 that will help you understand it (see appendix). Modern microprocessors have a lot in common with the simpler Z80, so understanding the 8-bit microcomputer system will help you understand the more complicated newer 32- and 64-bit microcomputers. And, despite being an “antique”, the Z80 is still being made, and is pretty cheap.[15]

14 Z80 is a registered trademark of the Zilog corporation.

15 When shopping for Z80s for this kit I noticed that prices have been rising in the past few years. This may be because of increasing hobbyist demand for a limited supply. But, it may also be due to the phenomenon of IC collectors. I saw on eBay early edition Z80s demanding prices in double digits.

Computer block diagram

This diagram shows the overall architecture of the CPUville Z80 system. Any general-purpose stored-program computer will have a similar architecture:



This block diagram shows the computer system as functional units. The Reset input controls whether the CPU is running or not. The Clock input drives the CPU when it is running. The CPU is connected with the memory that holds the program opcodes and data by a series of buses. Each bus is a set of parallel wires, one wire for each bit of information that is carried back and forth between these two units. The input and output ports are connected to the same buses.

The design of the processor dictates the features of the system buses. Since the Z80 is an 8-bit processor, the data bus will be 8-bits wide. One bit is carried on one wire, so there are 8 parallel wires in the data bus. The data bus is bidirectional. That is, data can flow from the CPU to the memory or output ports, or from the memory or input ports to the CPU. More about bi-directionality later.

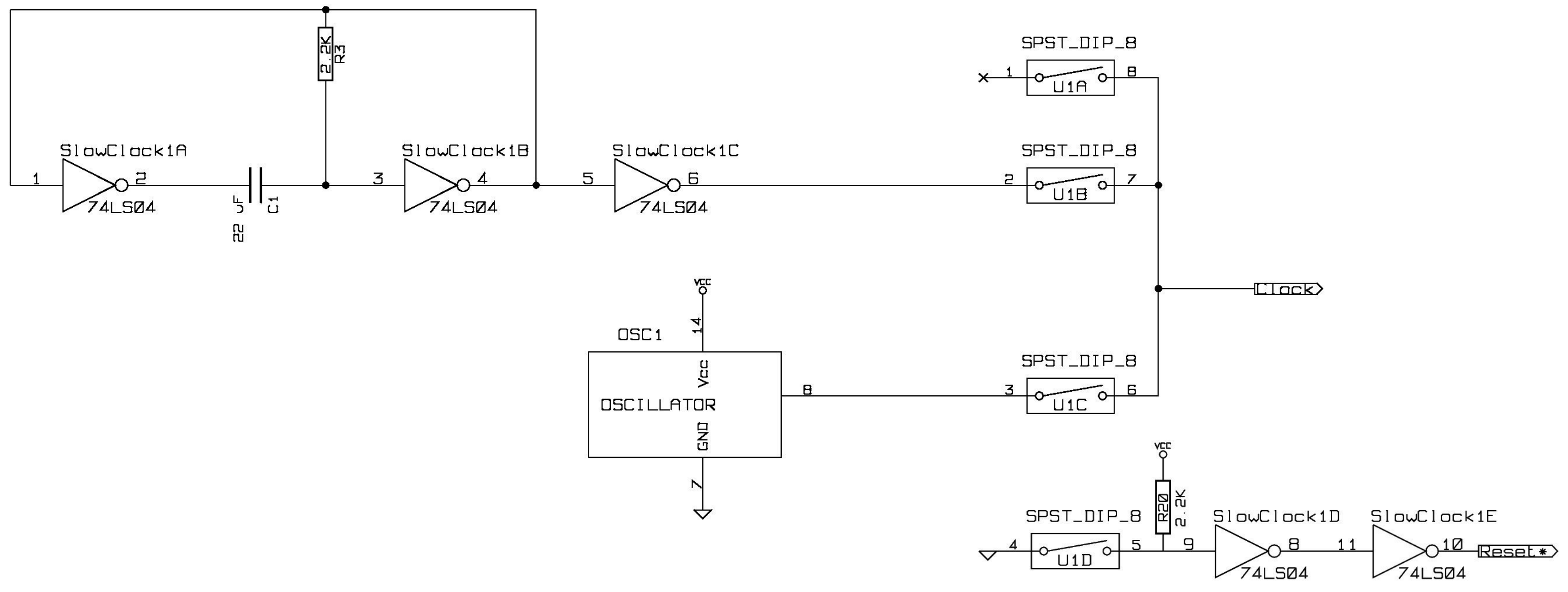
The address bus is one-way, with information flowing from the CPU to the memory and ports. It is the way the CPU tells the memory or ports which location is to be written to or read from. The Z80 has 16 wires in its address bus, meaning that 2^16 or 65,535 locations can be accessed. This is the “memory address space” of the processor. This number is often called “64K” informally.

The control bus is a set of 4 CPU outputs, named I/O_Req (input-output request), MemReq (memory request), Read, and Write. When the CPU wants to read a memory location, it will activate the MemReq and Read outputs. The computer system is designed so that the memory responds properly to this request by placing data onto the data bus so the CPU can read it. Similarly, if I/O_Req and Write are active, the CPU is telling the system that it wants to put a data byte onto the output port lights. The port circuitry is designed to respond appropriately.

This functional description must be translated into a detailed schematic design, that creates a working computer out of real electronic components. The following section explains the schematic design of the computer in detail.

Computer Schematics and Explanations

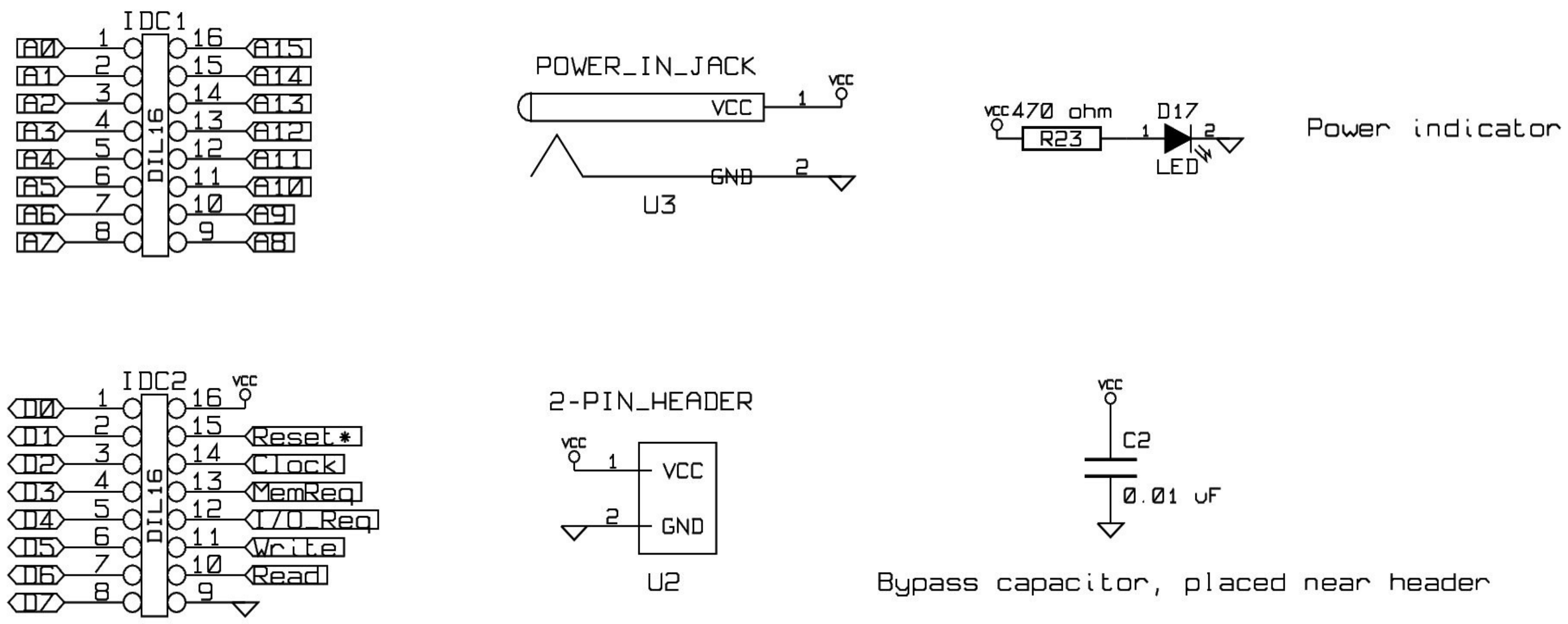
Clocks and Reset



The clock circuitry creates a regular train of square-wave pulses that the CPU needs in order to work. The slow clock is an R-C (resistor-capacitor) oscillator. The inverter gates act as amplifiers to keep the oscillator going, and give a square-wave output. The final inverter gate acts as the output. With the capacitor and resistor values here, the frequency of the slow clock is about 3 cycles per second (3 Hz). The fast clock oscillator is a quartz crystal, with associated circuitry, that puts out a 2MHz square wave.

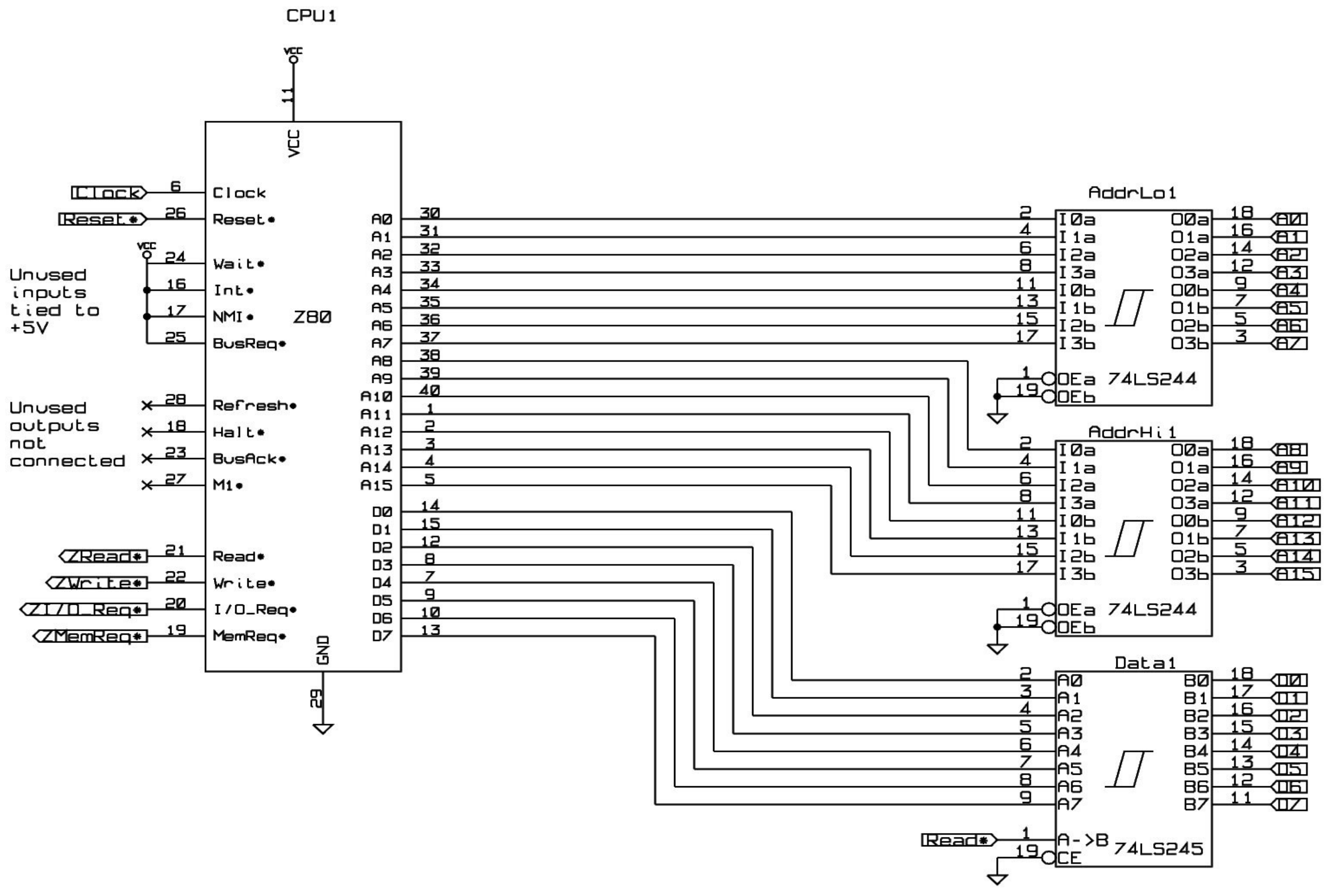
The reset circuit is just a buffered switch. When the switch is open (in the down position on the computer board), the resistor connected to Vcc (+5V in this design) causes the Reset* output to be +5V, and the CPU will run. When the reset switch is closed, the output is GND (0V) and the CPU will stop. Note that an asterisk (*) on a label in this schematic indicates that the signal is active-low. Therefore, when the Reset* signal is 0V, the computer is in the reset state (stopped).

Connectors



This schematic shows the connections on the two DIP (dual in-line package) sockets for connecting the computer to the display or to an expansion board. These signals, shown by the labels, are the address and data buses, power, and the control signals. The system clock and Reset* signals are also carried by the ICD2 connector. Note that the control signals (MemReq, I/O_Req, Write, and Read) on this connector are active-high (no asterisk on the labels). The power-in jack and the 2-pin header connect directly to the power traces on the board. You can connect either one to a +5V DC regulated power supply to power the board, but the 2-pin header connection is intended for the logic probe connection. The bypass capacitor prevents stray noise from getting from the logic probe connection wires onto the computer board, which could cause irregular behavior. The power indicator is just an LED connected to the power traces through a current-limiting resistor. The LED would burn out if connected directly, without the resistor.

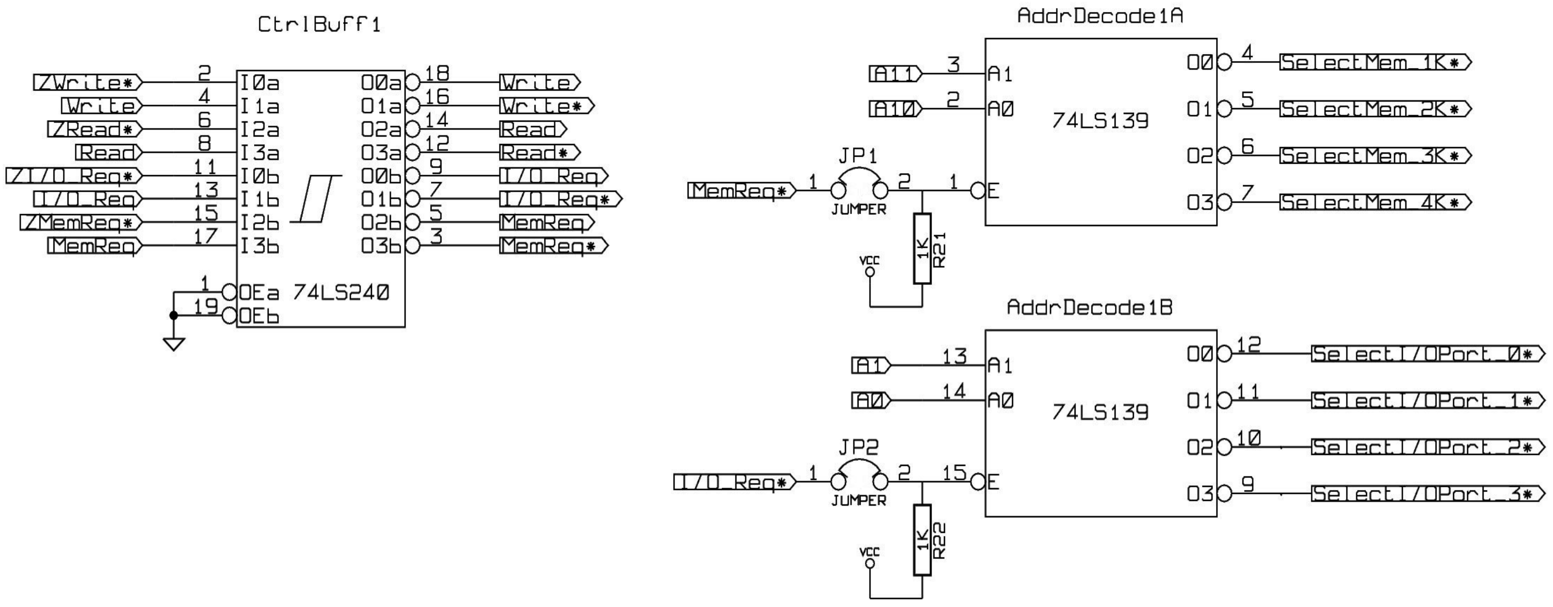
Z80 CPU and Buffers



Here is the heart of the computer system. The Z80 address and data outputs (A0 to A15, and D0 to D7) pass to buffers, which amplify the signals so that they can drive more inputs. The outputs of these buffers are the system address and data buses to which the memory and ports are connected. The address bus is output only, that is, the address is sent out from the CPU, through the buffers, to the rest of the system. The data bus is bi-directional, that is, data can be sent from the CPU out to the system, or read by the CPU from the system. Its buffer is connected to the Read* control signal. This buffer is “turned around” (inputs become outputs, outputs become inputs) when the CPU sets the Read* signal low, thereby informing the system it wants to read data from the data bus. The labels on the buffers on the are global labels. So, when looking at the rest of the schematic, if you see an A0 label on an input, you know this pin is connected to the A0 output on the AddrLo1 buffer. The power connections on the buffer ICs are not plotted, but are understood to be there. Most ICs have the GND input at the right lower corner of the IC (pin 10 is GND on these 20-pin buffer ICs), and Vcc, which equal +5V in this system, at the upper left corner (pin 20). If an IC has power connections that violate this convention, as the Z80 does, the pins are shown on the schematic.

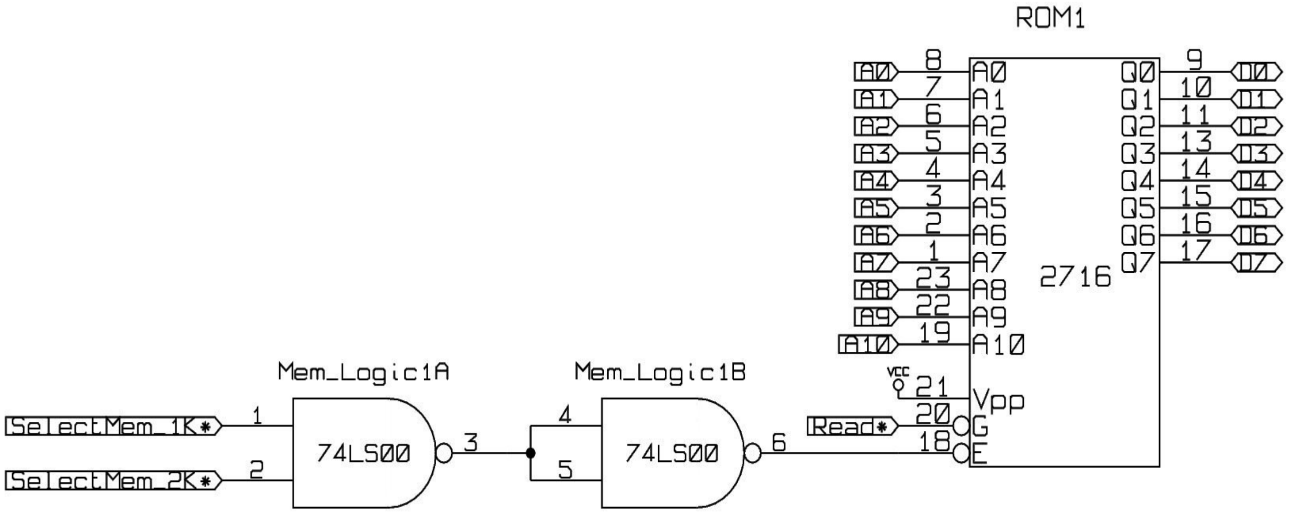
The Z80 has inputs and outputs for features that are not used in the CPUville computer. These are the interrupt and direct memory access systems. The CPU inputs that trigger the use of these systems are tied to Vcc, and therefore inactivated. The outputs that the CPU uses to operate these systems are left unconnected. The unused Refresh* output is meant for dynamic memory refreshing. The CPUville computer uses only static memory, so this output is not used.

Control Bus Buffer and Decoders



The control bus buffer simply amplifies the Z80 control signal outputs so they can be connected to many inputs in the system. Both the uninverted and inverted outputs (designated by *) are given labels for use in the system. The decoders are two halves of the same IC. The decoders are used to select the proper memory or input/output port IC when data is being written to, or read by, the the CPU. It “decodes” a two-bit address on the A0 and A1 inputs to select one of four outputs for activation. It is important that only one IC on the data bus is active at any one time. If there were two active ICs, both feeding data to the CPU, the data would be scrambled. The top decoder is used to select the active memory IC. Since there are two 2K memory chips (the 2716 EPROM, and the 2016 RAM), I made the memory decoder look at the address bus lines A10 and A11. The addresses in the first 2K of memory (in the EPROM) will have A11 set at 0. These addresses will active the EPROM through the SelectMem_1K* and SelectMem_2K* outputs (see the ROM schematic for further details on how this is done). When A11 is 1, the 2K RAM IC is selected using SelectMem_3K* and SelectMem_4K* outputs. Higher addresses will “wrap around”. In other words, addresses above 4K that leave both A10 and A11 off (such as binary 0001 0000 0000 0000) will activate the 2K ROM. The other decoder activates the ICs for input/output ports 0 and 1 using address lines A0 and A1. The decoder also creates port select signals for ports 2 and 3, but these are not implemented in the current computer system. The jumpers inactivate the decoders. This is necessary if you build an extension board with its own RAM and I/O ports. You would have to put more decoders on your expansion board, to selectively activate the memory and input/output port ICs that are there.

2K ROM

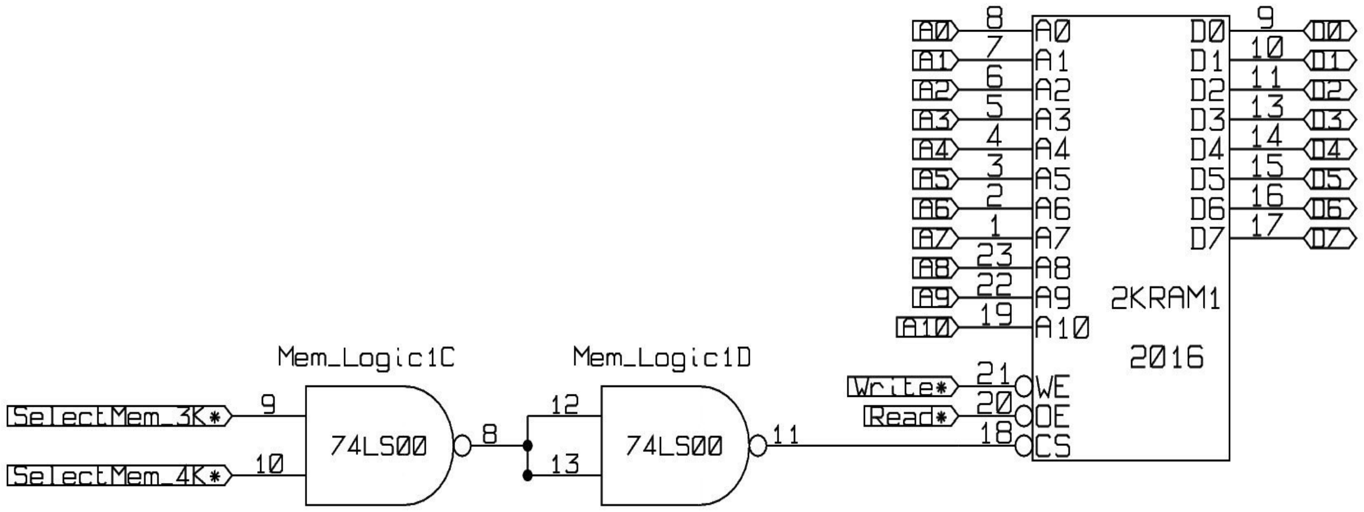


The 2K ROM chip is activated through the SelectMem_1K* and SelectMem_2K* decoder outputs. These signals are combined by the AND operation created by the two logic gates. The SelectMem signals, and the chip enable input are “active-low” signals. That is, when the chip enable input (pin 18) is 0, the chip is selected. The decoder will make only one SelectMem output 0 at a time, so only the last three table rows are possible. Here is the truth table of the AND operation on the SelectMem inputs:

SelectMem_1K*	SelectMem_2K*	AND result
0	0	0 (not possible)
0	1	0 (chip selected)
1	0	0 (chip selected)
1	1	1 (chip not selected)

The ROM will put data on the data bus when it is selected, and when the Read* signal is given. The ROM is a “read-only memory”, and cannot be written to when it is part of the computer system. To write the ROM, one needs to remove it an put it into an E/EPROM programmer. But, it will hold its data when the power is off, so the CPU will have code to execute as soon as the system is powered on. A PC has a ROM also, called the BIOS (for basic input-output system) that will be used by the processor to start the system.

2K RAM

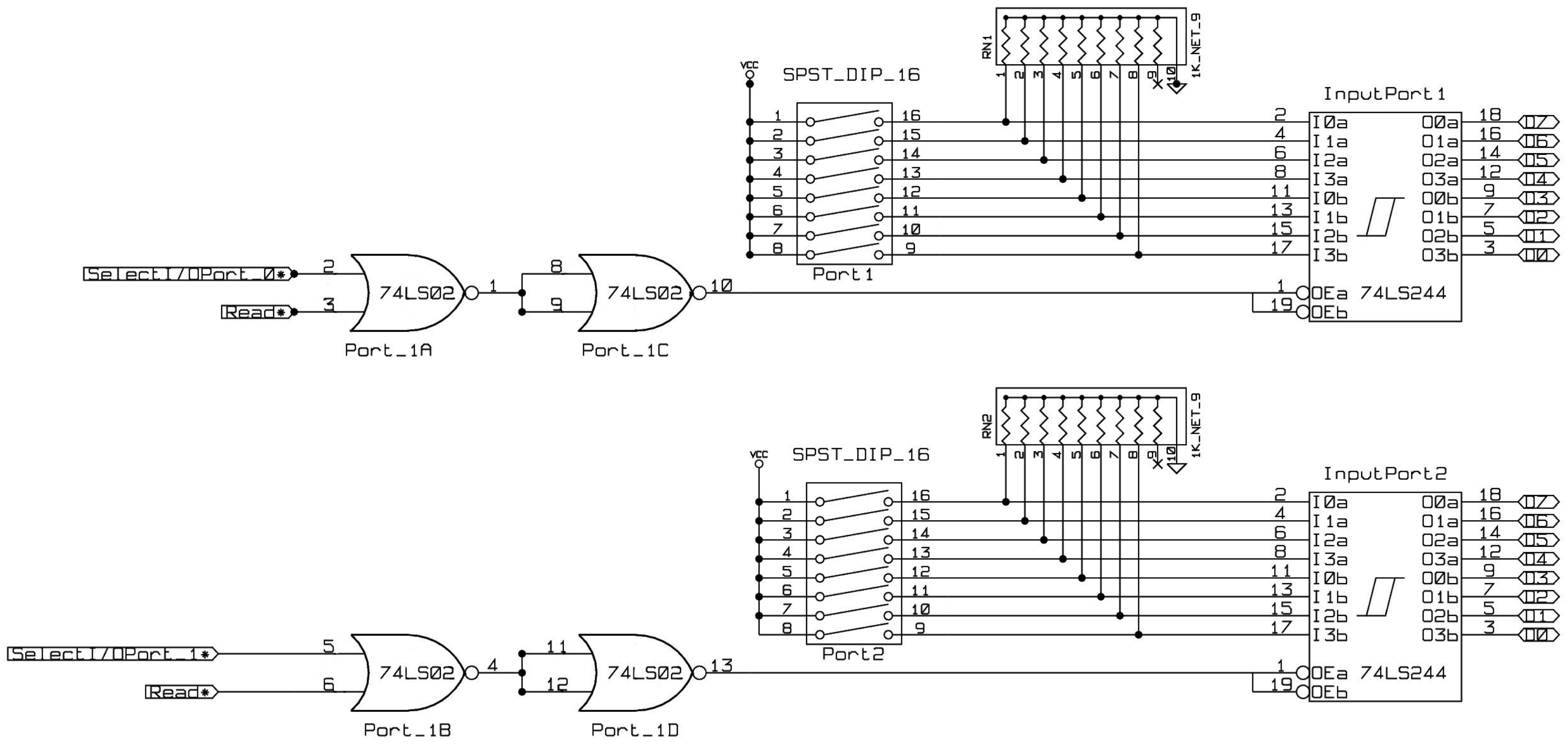


The 2K RAM is set up almost exactly the same as the 2K ROM. Unlike the ROM, the RAM can be written to and read from while is is in the computer system. That is why you see both Read* and Write* signals connected to the chip. The state of these inputs determines whether the chip takes input (Write*) or gives output (Read*). The truth table for the SelectMem* inputs is similar to that for the ROM, except that the SelectMem_3K and 4K decoder outputs are used:

SelectMem_3K*	SelectMem_4K*	AND result
0	0	0 (not possible)
0	1	0 (chip selected)
1	0	0 (chip selected)
1	1	1 (chip not selected)

Note the schematic shows a 2016 RAM IC, but kits may also be supplied with an equivalent 6116 RAM IC.

Input Ports

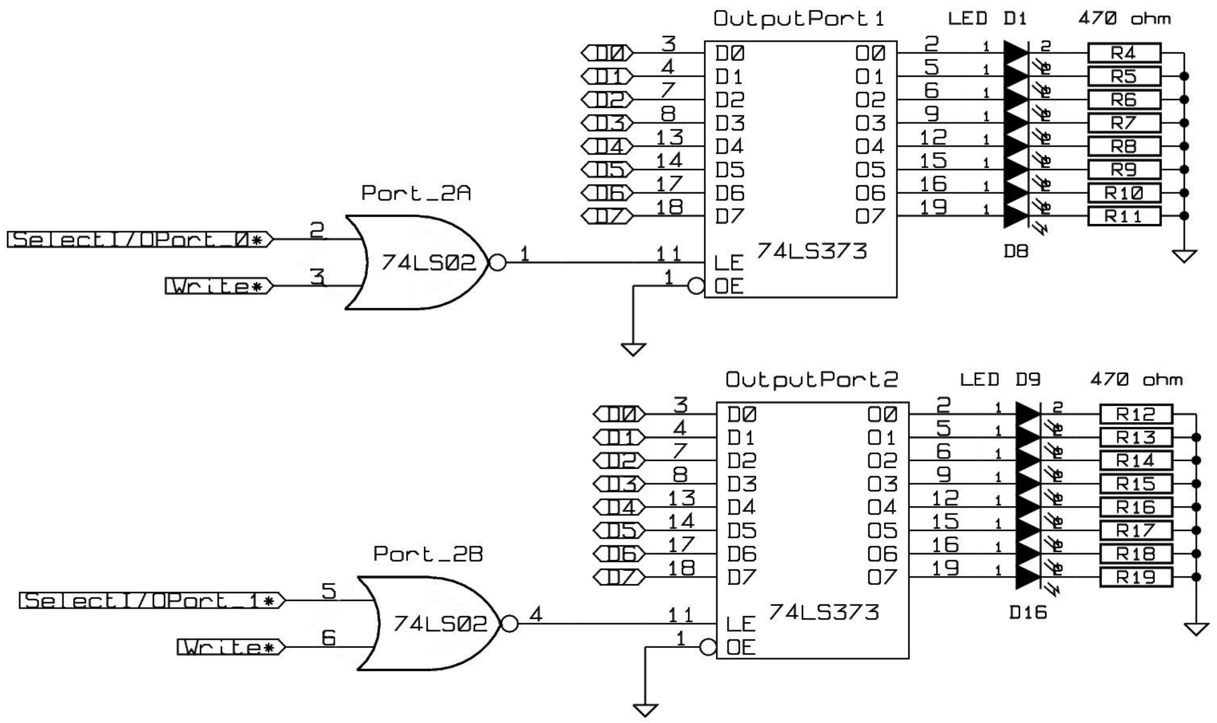


The switches control the signals to the inputs of the buffers for each port. The resistor networks will drain the "back current" produced by the buffer inputs so that when the switches are open the inputs will be at ground. Please note that the reference names of the two buffer ICs (InputPort1 and InputPort2) do not reflect the system addresses of these ports, which are 0 and 1, respectively. The reference names end with 1 and 2 because the computer schematic software I used would not allow a reference name to end in 0. A buffer is activated (puts output on the data bus) when the enable inputs on pins 1 and 19 are low (logic 0), that is, 0V or ground. The logic gates make this calculation by looking at the SelectI/OPort* and Read* signals, which come from the port decoder and the control bus buffer. The two NOR logic gates are configured to produce the logical OR operation (the second NOR gate is configured as an inverter). Here is the truth table for the port 0 signals:

SelectI/OPort_0*	Read*	OR result
0	0	0
0	1	1
1	0	1
1	1	1

You can see that the only time the port buffer is active (result is 0) is when both the SelectI/OPort_0* and Read* signals are active (both 0).

Output Ports

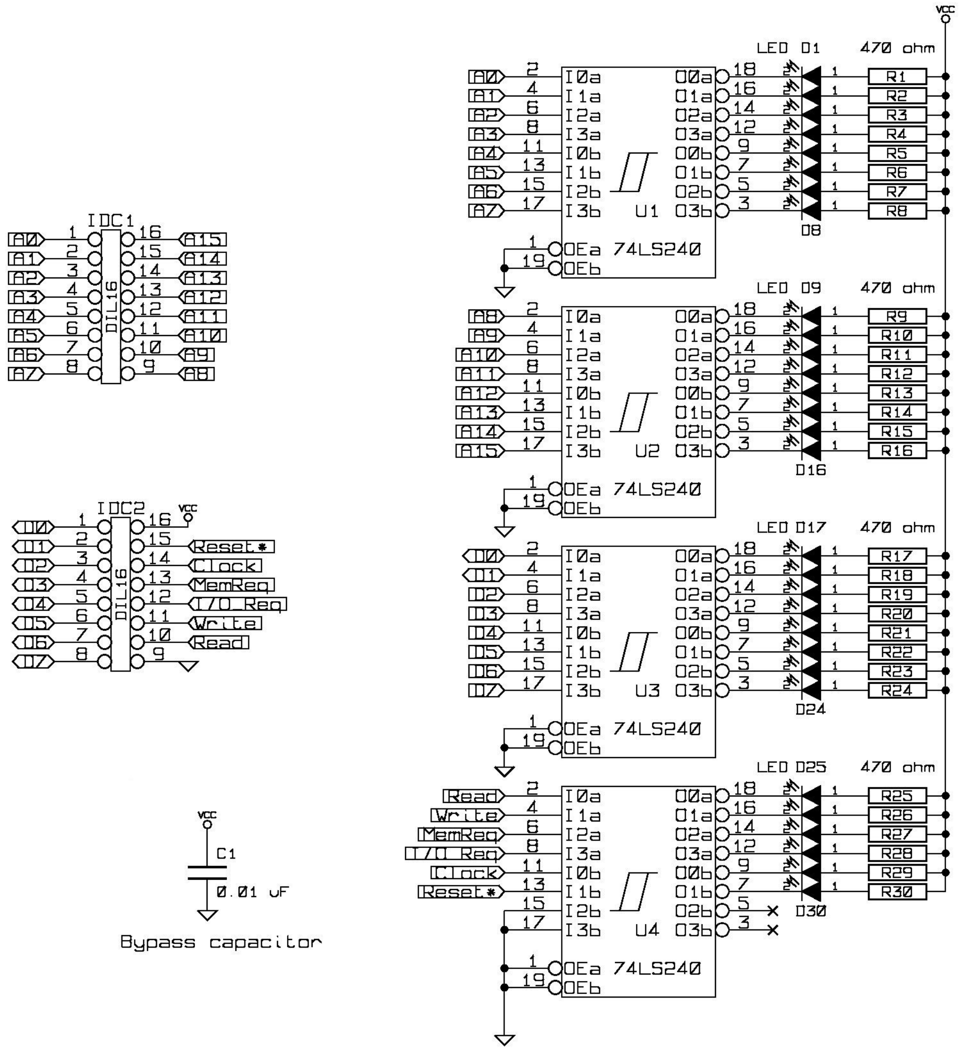


The output port ICs are latches, that hold on to data once it is loaded in. Unlike the input port buffers, which open briefly to allow the data onto the data bus, these output ports must hold onto the data they receive from the data bus until they are written again. The latch enable latch-enable (LE) inputs on pin 11 are active-**high**, as opposed to most of the enable inputs we have seen in the system so far, which are active-low. So, the signal decoding to enable the latches is a little different. It requires a logical NOR operation. Here is the truth table for the port 0 logic:

SelectI/OPort_0*	Write*	NOR result
0	0	1
0	1	0
1	0	0
1	1	0

You can see that the latch is enabled (result is 1) when SelectI/O_0* and Write* are both active (that is, are both 0).

Display Schematic and Explanation



The display is a simple schematic. The two connectors IDC1 and IDC2 are exactly the same as the connectors on the computer. The signals from the connectors are inputs to the four buffer ICs, which drive the LED outputs. There is a bypass capacitor connected across the power lines to prevent noise from the display unit from getting onto the computer power lines.

Logic Probe Schematic and Explanation

The logic probe schematic and explanation is in the new logic probe instruction manual, which can be obtained from this link:

http://www.cpuville.com/Kits/logic-probe-instructions.pdf

Appendix

Logic Probe parts organizer and list

The parts organizer and list for the logic probe is in the new logic probe instructions, which can be obtained from this link:

http://www.cpuville.com/Kits/logic-probe-instructions.pdf

Display parts organizer and list

Capacitor, 0.01 uF disk 1	Red LED 30	Resistor, 470 ohm Yellow-Violet-Brown 30	DIL16 connector 2
74LS240 4			

Ref	Value	Ref	Value
C1	0.01 uF	R1	470 ohm
D1	LED	R2	470 ohm
D2	LED	R3	470 ohm
D3	LED	R4	470 ohm
D4	LED	R5	470 ohm
D5	LED	R6	470 ohm
D6	LED	R7	470 ohm
D7	LED	R8	470 ohm
D8	LED	R9	470 ohm
D9	LED	R10	470 ohm
D10	LED	R11	470 ohm
D11	LED	R12	470 ohm
D12	LED	R13	470 ohm
D13	LED	R14	470 ohm
D14	LED	R15	470 ohm
D15	LED	R16	470 ohm
D16	LED	R17	470 ohm
D17	LED	R18	470 ohm
D18	LED	R19	470 ohm
D19	LED	R20	470 ohm
D20	LED	R21	470 ohm
D21	LED	R22	470 ohm
D22	LED	R23	470 ohm
D23	LED	R24	470 ohm
D24	LED	R25	470 ohm
D25	LED	R26	470 ohm
D26	LED	R27	470 ohm
D27	LED	R28	470 ohm
D28	LED	R29	470 ohm
D29	LED	R30	470 ohm
D30	LED	U1	74LS240
IDC1	DIL16	U2	74LS240
IDC2	DIL16	U3	74LS240
		U4	74LS240

Computer parts organizer and list

Capacitor, 22 uF, 16V 1	Resistor, 2.2K, ¼ watt Red-Red-Red 2	74LS00 quad NAND 1	74LS02 quad NOR 2
74LS04 hex inverter 1	SPST_DIP_16 switches 2	SPST_DIP_8 switches 1	DIL_16 socket 2
DIL_24 socket 2	74LS139 decoder 1	2016 or 6116 2K RAM 1	2716 2K ROM 1
74LS373 latch 2	74LS240 octal buffer, inverting 1 1	74LS244 octal buffer 4	74LS245 bus transceiver 1
Z80 CPU 1	Resistor, 470 ohm, ¼ watt Yellow-Violet-Brown 17	Resistor, 1K, ¼ watt Brown-Black-Red 2	Resistor network, 1K x 9 2
LED 17	Oscillator, 2 MHz 1	2-pin header 2	Shorting block 2
Capacitor, 0.01 uF disk 1	Header with clip, 2 pin 1	Power-in jack 1	DIL 40-pin socket 1

Ref	**Value**
2KRAM1	2016
AddrDecode1	74LS139
AddrHi1	74LS244
AddrLo1	74LS244
C1	22 uF
C2	0.01 uF
CPU1	Z80
CtrlBuff1	74LS240
D1	LED
D2	LED
D3	LED
D4	LED
D5	LED
D6	LED
D7	LED
D8	LED
D9	LED
D10	LED
D11	LED
D12	LED
D13	LED
D14	LED
D15	LED
D16	LED
D17	LED
Data1	74LS245
IDC1	DIL16
IDC2	DIL16
InputPort1	74LS244
InputPort2	74LS244
JP1	JUMPER (2-pin header with shorting block)
JP2	JUMPER (2-pin header with shorting block)
Mem_Logic1	74LS00
OSC1	OSCILLATOR
OutputPort1	74LS373
OutputPort2	74LS373
Port1	SPST_DIP_16
Port2	SPST_DIP_16
Port_1	74LS02
Port_2	74LS02
R3	2.2K
R4	470 ohm
R5	470 ohm
R6	470 ohm
R7	470 ohm
R8	470 ohm
R9	470 ohm
R10	470 ohm
R11	470 ohm
R12	470 ohm
R13	470 ohm
R14	470 ohm
R15	470 ohm
R16	470 ohm
R17	470 ohm
R18	470 ohm
R19	470 ohm
R20	2.2K
R21	1K
R22	1K
R23	470 ohm
RN1	1K_NET_9
RN2	1K_NET_9
ROM1	2716
SlowClock1	74LS04
U1	SPST_DIP_8
U2	2-PIN_HEADER
U3	POWER_IN_JACK

Program Listings

The following are the programs that I have written and assembled for this computer kit project. The listings are outputs of the assembler program I am using, z80asm, an open-source program written by Bas Wijnen that can be found at http://packages.qa.debian.org/z/z80asm.html. Each program is written with a text editor, and that file is used as the input for the assembler. The output of the assembler program is two files. One is the binary file of machine code that is intended to be loaded into memory and executed by the processor. The other is this listing file, which is a text file that can be read by humans.

There are five columns in this listing. In the first column are the 16-bit hexadecimal memory addresses where the machine code is to be stored. The second column has the hexadecimal machine code bytes stored in the memory locations shown by the addresses. The third column has the labels, if the address has one. The fourth has the assembly language instructions (opcode and operand(s)). The fifth has comments. The labels, assembly language instructions, and comments are from the program file that I wrote. The memory addresses and machine code bytes are generated by the assembler program.

When the Z80 processor is taken out of reset (set to Run), it automatically get its first instruction from memory location 0000h. The 2K ROM listing here shows the contents of the ROM, starting at address 0000h. The ROM starts with a jump instruction in location 0000h, which will always be the first instruction the Z80 executes when it is taken out of reset. This instruction causes program execution to skip over the text message "CPUville Z80 ROM v.7" and start the Get_address program at location 0018h. This program will read a 16-bit address from the input ports (that was put there while the computer was in Reset) and jump to whatever program is at that address. You can jump anywhere in memory when you start the computer, even to RAM addresses, where you have placed your own programs.

```
# File 2K_ROM_7.asm
0000                                    org   00000h
0000              Start_of_RAM:         equ   0x0800
0000 c3 18 00                           jp    Get_address       ;Skip over message
0003 .. 00                              defm  "CPUville Z80 ROM v.7",0
0018 db 00        Get_address:          in    a,(0)             ;Get address from input ports
001a 6f                                 ld    l,a
001b db 01                              in    a,(1)
001d 67                                 ld    h,a
001e e9                                 jp    (hl)              ;Jump to the address
001f db 00        Port_Reflector:       in    a,(0)             ;Simple program to test ports
0021 d3 00                              out   (0),a
0023 db 01                              in    a,(1)
0025 d3 01                              out   (1),a
0027 c3 1f 00                           jp    Port_Reflector
002a 3e 00        Simple_Counter:       ld    a,000h            ;One-byte counter for slow clock
002c d3 00        Loop_1:               out   (0),a
002e 3c                                 inc   a
002f c3 2c 00                           jp    Loop_1
0032 2e 00        Count_to_a_million:   ld    l,000h            ;Two-byte (16-bit) counter
0034 26 00                              ld    h,000h            ;Clear registers
0036 3e 10        Loop_2:               ld    a,010h            ;Count 16 times, then
0038 3d           Loop_3:               dec   a
0039 c2 38 00                           jp    nz,Loop_3
003c 23                                 inc   hl                ;increment the 16-bit number
003d 7d                                 ld    a,l
003e d3 00                              out   (0),a             ;Output the 16-bit number
0040 7c                                 ld    a,h
0041 d3 01                              out   (1),a
0043 c3 36 00                           jp    Loop_2            ;Do it again
0046 21 00 08     Program_loader:       ld    hl,Start_of_RAM   ;Load a program in RAM
0049 db 01        Loop_4:               in    a,(1)
004b e6 81                              and   081h              ;Check input port 1
004d ca 49 00                           jp    z,Loop_4          ;If switches 0 and 7 open, loop
0050 cd f5 00                           call  debounce
0053 db 01                              in    a,(1)             ;Get input port byte again
0055 e6 80                              and   080h              ;Is the left switch (bit 7) closed?
```

```
0057 c2 00 08                        jp    nz,Start_of_RAM   ;Yes, run loaded program
005a db 00                           in    a,(0)             ;No, then right switch (bit 0) closed.
005c d3 00                           out   (0),a             ;Get byte from port 0, display on output
005e 77                              ld    (hl),a            ;Store it in RAM
005f 3e ff                           ld    a,0ffh            ;Turn port 1 lights on (signal that
0061 d3 01                           out   (1),a             ;a byte was stored)
0063 db 01        Loop_6:            in    a,(1)             ;Wait for switch to open
0065 e6 01                           and   001h
0067 c2 63 00                        jp    nz,Loop_6
006a cd f5 00                        call  debounce
006d 7d                              ld    a,l               ;Put low byte of address on port 1
006e d3 01                           out   (1),a
0070 23                              inc   hl                ;Point to next location in RAM
0071 c3 49 00                        jp    Loop_4            ;Do it again
0074 21 00 08     Memory_test:       ld    hl,Start_of_RAM   ;check RAM by writing and reading each location
0077 db 01        Loop_8:            in    a,(1)             ;read port 1 to get a bit pattern
0079 47                              ld    b,a               ;copy it to register b
007a 77                              ld    (hl),a            ;store it in memory
007b 7e                              ld    a,(hl)            ;read back the same location
007c b8                              cp    b                 ;same as reg b?
007d c2 84 00                        jp    nz,Exit_1         ;no, test failed, exit
0080 23                              inc   hl                ;yes, RAM location OK
0081 c3 77 00                        jp    Loop_8            ;keep going
0084 7c           Exit_1:            ld    a,h               ;display the address
0085 d3 01                           out   (1),a             ;where the test failed
0087 7d                              ld    a,l               ;should be 4K (cycled around to ROM)
0088 d3 00                           out   (0),a             ;any other value means bad RAM
008a c3 74 00                        jp    Memory_test       ;do it again (use a different bit pattern)
008d db 00        Peek:              in    a,(0)             ;Get low byte
008f 6f                              ld    l,a               ;Put in reg L
0090 db 01                           in    a,(1)             ;Get hi byte
0092 67                              ld    h,a               ;Put in reg H
0093 7e                              ld    a,(hl)            ;Get byte from memory
0094 d3 00                           out   (0),a             ;Display on port 0 LEDs
0096 c3 8d 00                        jp    Peek              ;Do it again
0099 3e 00        Poke:              ld    a,000h            ;Clear output port LEDs
009b d3 00                           out   (0),a
```

```
009d d3 01                              out   (1),a
009f db 01      Loop_9:                 in    a,(1)           ;Look for switch closure
00a1 e6 01                              and   001h
00a3 ca 9f 00                           jp    z,Loop_9
00a6 cd f5 00                           call  debounce
00a9 3e ff                              ld    a,0ffh          ;Light port 1 LEDs
00ab d3 01                              out   (1),a
00ad db 00                              in    a,(0)           ;Get hi byte
00af 67                                 ld    h,a             ;Put in reg H
00b0 db 01      Loop_11:                in    a,(1)           ;Look for switch open
00b2 e6 01                              and   001h
00b4 c2 b0 00                           jp    nz,Loop_11
00b7 cd f5 00                           call  debounce
00ba 7c                                 ld    a,h             ;Show hi byte on port 1
00bb d3 01                              out   (1),a
00bd db 01      Loop_13:                in    a,(1)           ;Look for switch closure
00bf e6 01                              and   001h
00c1 ca bd 00                           jp    z,Loop_13
00c4 cd f5 00                           call  debounce
00c7 3e ff                              ld    a,0ffh          ;Light port 0 LEDs
00c9 d3 00                              out   (0),a
00cb db 00                              in    a,(0)           ;Get lo byte
00cd 6f                                 ld    l,a             ;Put in reg L
00ce db 01      Loop_15:                in    a,(1)           ;Look for switch open
00d0 e6 01                              and   001h
00d2 c2 ce 00                           jp    nz,Loop_15
00d5 cd f5 00                           call  debounce
00d8 7d                                 ld    a,l             ;Show lo byte on port 0
00d9 d3 00                              out   (0),a
00db db 01      Loop_17:                in    a,(1)           ;Look for switch closure
00dd e6 01                              and   001h
00df ca db 00                           jp    z,Loop_17
00e2 cd f5 00                           call  debounce
00e5 db 00                              in    a,(0)           ;Get byte to load
00e7 77                                 ld    (hl),a          ;Store in memory
00e8 db 01      Loop_19:                in    a,(1)           ;Look for switch open
00ea e6 01                              and   001h
```

```
00ec c2 e8 00                          jp    nz,Loop_19
00ef cd f5 00                          call  debounce
00f2 c3 99 00                          jp    Poke              ;Start over
00f5                 ;
00f5                 ;Subroutine for a switch debounce delay
00f5 3e 10           debounce:         ld    a,010h            ;Outer loop
00f7 06 ff           debounce_loop:    ld    b,0ffh            ;Inner loop
00f9 10 fe                             djnz  $+0               ;Loop here until B reg is zero
00fb 3d                                dec   a
00fc c2 f7 00                          jp    nz,debounce_loop
00ff c9                                ret
0100                 ;
0100                 ;The following code is for a system with a serial port.
0100                 ;Assumes the UART data port address is 02h and control/status address is 03h
0100                 ;
0100                 ;The subroutines for the serial port use these variables in high RAM:
0100                 current_location:      equ   0x0f80            ;word variable in RAM
0100                 line_count:            equ   0x0f82            ;byte variable in RAM
0100                 byte_count:            equ   0x0f83            ;byte variable in RAM
0100                 value_pointer:         equ   0x0f84            ;word variable in RAM
0100                 current_value:         equ   0x0f86            ;word variable in RAM
0100                 buffer:                equ   0x0f88            ;buffer in RAM -- up to stack area
0100                 ;
0100                 ;Subroutine to initialize serial port UART
0100                 ;Needs to be called only once after computer comes out of reset.
0100                 ;If called while port is active will cause port to fail.
0100                 ;16x = 9600 baud
0100 3e 4e           initialize_port:       ld    a,04eh            ;1 stop bit, no parity, 8-bit char, 16x baud
0102 d3 03                                  out   (3),a             ;write to control port
0104 3e 37                                  ld    a,037h            ;enable receive and transmit
0106 d3 03                                  out   (3),a             ;write to control port
0108 c9                                     ret
0109                 ;
0109                 ;Puts a single char (byte value) on serial output
0109                 ;Call with char to send in A register. Uses B register
0109 47              write_char:            ld    b,a               ;store char
010a db 03           write_char_loop:       in    a,(3)             ;check if OK to send
```

```
010c e6 01                        and   001h              ;check TxRDY bit
010e ca 0a 01                     jp    z,write_char_loop ;loop if not set
0111 78                           ld    a,b               ;get char back
0112 d3 02                        out   (2),a             ;send to output
0114 c9                           ret                     ;returns with char in a
0115             ;
0115             ;Subroutine to write a zero-terminated string to serial output
0115             ;Pass address of string in HL register
0115             ;No error checking
0115 db 03       write_string:    in    a,(3)             ;read status
0117 e6 01                        and   001h              ;check TxRDY bit
0119 ca 15 01                     jp    z,write_string    ;loop if not set
011c 7e                           ld    a,(hl)            ;get char from string
011d a7                           and   a                 ;check if 0
011e c8                           ret   z                 ;yes, finished
011f d3 02                        out   (2),a             ;no, write char to output
0121 23                           inc   hl                ;next char in string
0122 c3 15 01                     jp    write_string            ;start over
0125             ;
0125             ;Binary loader. Receive a binary file, place in memory.
0125             ;Address of load passed in HL, length of load (= file length) in BC
0125 db 03       bload:           in    a,(3)             ;get status
0127 e6 02                        and   002h              ;check RxRDY bit
0129 ca 25 01                     jp    z,bload           ;not ready, loop
012c db 02                        in    a,(2)
012e 77                           ld    (hl),a
012f 23                           inc   hl
0130 0b                           dec   bc                ;byte counter
0131 78                           ld    a,b               ;need to test BC this way because
0132 b1                           or    c                 ;dec rp instruction does not change flags
0133 c2 25 01                     jp    nz,bload
0136 c9                           ret
0137             ;
0137             ;Binary dump to port. Send a stream of binary data from memory to serial output
0137             ;Address of dump passed in HL, length of dump in BC
0137 db 03       bdump:           in    a,(3)             ;get status
0139 e6 01                        and   001h              ;check TxRDY bit
```

```
013b ca 37 01                          jp    z,bdump           ;not ready, loop
013e 7e                                ld    a,(hl)
013f d3 02                             out   (2),a
0141 23                                inc   hl
0142 0b                                dec   bc
0143 78                                ld    a,b               ;need to test this way because
0144 b1                                or    c                 ;dec rp instruction does not change flags
0145 c2 37 01                          jp    nz,bdump
0148 c9                                ret
0149                 ;
0149                 ;Subroutine to get a string from serial input, place in buffer.
0149                 ;Buffer address passed in HL reg.
0149                 ;Uses A,BC,DE,HL registers (including calls to other subroutines).
0149                 ;Line entry ends by hitting return key. Return char not included in string (replaced by zero).
0149                 ;Backspace editing OK. No error checking.
0149                 ;
0149 0e 00           get_line:         ld    c,000h            ;line position
014b 7c                                ld    a,h               ;put original buffer address in de
014c 57                                ld    d,a               ;after this don't need to preserve hl
014d 7d                                ld    a,l               ;subroutines called don't use de
014e 5f                                ld    e,a
014f db 03           get_line_next_char:     in    a,(3)                   ;get status
0151 e6 02                             and   002h                    ;check RxRDY bit
0153 ca 4f 01                          jp    z,get_line_next_char    ;not ready, loop
0156 db 02                             in    a,(2)                   ;get char
0158 fe 0d                             cp    00dh                    ;check if return
015a c8                                ret   z                       ;yes, normal exit
015b fe 7f                             cp    07fh                    ;check if backspace (VT102 keys)
015d ca 71 01                          jp    z,get_line_backspace    ;yes, jump to backspace routine
0160 fe 08                             cp    008h                    ;check if backspace (ANSI keys)
0162 ca 71 01                          jp    z,get_line_backspace    ;yes, jump to backspace
0165 cd 09 01                          call  write_char              ;put char on screen
0168 12                                ld    (de),a                  ;store char in buffer
0169 13                                inc   de                      ;point to next space in buffer
016a 0c                                inc   c                       ;inc counter
016b 3e 00                             ld    a,000h
016d 12                                ld    (de),a                  ;leaves a zero-terminated string in buffer
```

```
016e c3 4f 01                        jp    get_line_next_char
0171 79          get_line_backspace:     ld    a,c                     ;check current position in line
0172 fe 00                           cp    000h                    ;at beginning of line?
0174 ca 4f 01                        jp    z,get_line_next_char    ;yes, ignore backspace, get next char
0177 1b                              dec   de                      ;no, erase char from buffer
0178 0d                              dec   c                       ;back up one
0179 3e 00                           ld    a,000h                  ;replace last char with zero
017b 12                              ld    (de),a
017c 21 e1 03                        ld    hl,erase_char_string    ;ANSI seq. to delete one char
017f cd 15 01                        call  write_string            ;backspace and erase char
0182 c3 4f 01                        jp    get_line_next_char
0185             ;
0185             ;Creates a two-char hex string from the byte value passed in register A
0185             ;Location to place string passed in HL
0185             ;String is zero-terminated, stored in 3 locations starting at HL
0185             ;Also uses registers b,d, and e
0185 47          byte_to_hex_string:     ld    b,a               ;store original byte
0186 cb 3f                           srl   a                 ;shift right 4 times, putting
0188 cb 3f                           srl   a                 ;high nybble in low-nybble spot
018a cb 3f                           srl   a                 ;and zeros in high-nybble spot
018c cb 3f                           srl   a
018e 16 00                           ld    d,000h            ;prepare for 16-bit addition
0190 5f                              ld    e,a               ;de contains offset
0191 e5                              push  hl                ;temporarily store string target address
0192 21 eb 01                        ld    hl,hex_char_table ;use char table to get high-nybble character
0195 19                              add   hl,de             ;add offset to start of table
0196 7e                              ld    a,(hl)            ;get char
0197 e1                              pop   hl                ;get string target address
0198 77                              ld    (hl),a            ;store first char of string
0199 23                              inc   hl                ;point to next string target address
019a 78                              ld    a,b               ;get original byte back from reg b
019b e6 0f                           and   00fh              ;mask off high-nybble
019d 5f                              ld    e,a               ;d still has 000h, now de has offset
019e e5                              push  hl                ;temp store string target address
019f 21 eb 01                        ld    hl,hex_char_table ;start of table
01a2 19                              add   hl,de             ;add offset
01a3 7e                              ld    a,(hl)            ;get char
```

```
01a4 e1                             pop   hl                ;get string target address
01a5 77                             ld    (hl),a            ;store second char of string
01a6 23                             inc   hl                ;point to third location
01a7 3e 00                          ld    a,000h            ;zero to terminate string
01a9 77                             ld    (hl),a            ;store the zero
01aa c9                             ret                     ;done
01ab            ;
01ab            ;Converts a single ASCII hex char to a nybble value
01ab            ;Pass char in reg A. Letter numerals must be upper case.
01ab            ;Return nybble value in low-order reg A with zeros in high-order nybble if no error.
01ab            ;Return 0ffh in reg A if error (char not a valid hex numeral).
01ab            ;Also uses b, c, and hl registers.
01ab 21 eb 01   hex_char_to_nybble:     ld    hl,hex_char_table
01ae 06 0f                          ld    b,00fh                  ;no. of valid characters in table - 1.
01b0 0e 00                          ld    c,000h                  ;will be nybble value
01b2 be         hex_to_nybble_loop:     cp    (hl)                    ;character match here?
01b3 ca bf 01                       jp    z,hex_to_nybble_ok      ;match found, exit
01b6 05                             dec   b                       ;no match, check if at end of table
01b7 fa c1 01                       jp    m,hex_to_nybble_err     ;table limit exceded, exit with error
01ba 0c                             inc   c                       ;still inside table, continue search
01bb 23                             inc   hl
01bc c3 b2 01                       jp    hex_to_nybble_loop
01bf 79         hex_to_nybble_ok:       ld    a,c                     ;put nybble value in a
01c0 c9                             ret
01c1 3e ff      hex_to_nybble_err:      ld    a,0ffh                  ;error value
01c3 c9                             ret
01c4            ;
01c4            ;Converts a hex character pair to a byte value
01c4            ;Called with location of high-order char in HL
01c4            ;If no error carry flag clear, returns with byte value in register A, and
01c4            ;HL pointing to next mem location after char pair.
01c4            ;If error (non-hex char) carry flag set, HL pointing to invalid char
01c4 7e         hex_to_byte:            ld    a,(hl)            ;location of character pair
01c5 e5                             push  hl                ;store hl (hex_char_to_nybble uses it)
01c6 cd ab 01                       call  hex_char_to_nybble
01c9 e1                             pop   hl                ;returns with nybble in a reg, or 0ffh if error
01ca fe ff                          cp    0ffh              ;non-hex character?
```

```
01cc ca e9 01                           jp    z,hex_to_byte_err ;yes, exit with error
01cf cb 27                              sla   a                 ;no, move low order nybble to high side
01d1 cb 27                              sla   a
01d3 cb 27                              sla   a
01d5 cb 27                              sla   a
01d7 57                                 ld    d,a               ;store high-nybble
01d8 23                                 inc   hl                ;get next character of the pair
01d9 7e                                 ld    a,(hl)
01da e5                                 push  hl                ;store hl
01db cd ab 01                           call  hex_char_to_nybble
01de e1                                 pop   hl
01df fe ff                              cp    0ffh              ;non-hex character?
01e1 ca e9 01                           jp    z,hex_to_byte_err ;yes, exit with error
01e4 b2                                 or    d                 ;no, combine with high-nybble
01e5 23                                 inc   hl                ;point to next memory location after char pair
01e6 37                                 scf
01e7 3f                                 ccf                     ;no-error exit (carry = 0)
01e8 c9                                 ret
01e9 37             hex_to_byte_err:    scf                     ;error, carry flag set
01ea c9                                 ret
01eb ..             hex_char_table:     defm  "0123456789ABCDEF"      ;ASCII hex table
01fb                ;
01fb                ;Subroutine to get a two-byte address from serial input.
01fb                ;Returns with address value in HL
01fb                ;Uses locations in RAM for buffer and variables
01fb 21 88 0f       address_entry:      ld    hl,buffer               ;location for entered string
01fe cd 49 01                           call  get_line                ;returns with address string in buffer
0201 21 88 0f                           ld    hl,buffer               ;location of stored address entry string
0204 cd c4 01                           call  hex_to_byte             ;will get high-order byte first
0207 da 1d 02                           jp    c, address_entry_error  ;if error, jump
020a 32 81 0f                           ld    (current_location+1),a  ;store high-order byte, little-endian
020d 21 8a 0f                           ld    hl,buffer+2             ;point to low-order hex char pair
0210 cd c4 01                           call  hex_to_byte             ;get low-order byte
0213 da 1d 02                           jp    c, address_entry_error  ;jump if error
0216 32 80 0f                           ld    (current_location),a    ;store low-order byte in lower memory
0219 2a 80 0f                           ld    hl,(current_location)   ;put memory address in hl
021c c9                                 ret
```

```
021d 21 1f 04     address_entry_error:    ld    hl,address_error_msg
0220 cd 15 01                             call  write_string
0223 c3 fb 01                             jp    address_entry
0226              ;
0226              ;Subroutine to get a decimal string, return a word value
0226              ;Calls decimal_string_to_word subroutine
0226 21 88 0f     decimal_entry:          ld    hl,buffer
0229 cd 49 01                             call  get_line          ;returns with DE pointing to terminating zero
022c 21 88 0f                             ld    hl,buffer
022f cd 3c 02                             call  decimal_string_to_word
0232 d0                                   ret   nc                ;no error, return with word in hl
0233 21 93 04                             ld    hl,decimal_error_msg    ;error, try again
0236 cd 15 01                             call  write_string
0239 c3 26 02                             jp    decimal_entry
023c              ;
023c              ;Subroutine to convert a decimal string to a word value
023c              ;Call with address of string in HL, pointer to end of string in DE
023c              ;Carry flag set if error (non-decimal char)
023c              ;Carry flag clear, word value in HL if no error.
023c 42           decimal_string_to_word: ld    b,d
023d 4b                                   ld    c,e                     ;use BC as string pointer
023e 22 80 0f                             ld    (current_location),hl   ;store addr. of start of buffer in RAM
0241 21 00 00                             ld    hl,000h                 ;starting value zero
0244 22 86 0f                             ld    (current_value),hl
0247 21 8c 02                             ld    hl,decimal_place_value  ;pointer to values
024a 22 84 0f                             ld    (value_pointer),hl
024d 0b           decimal_next_char:      dec   bc                      ;next char in string (moving R to L)
024e 2a 80 0f                             ld    hl,(current_location)   ;check if at end of decimal string
0251 37                                   scf
0252 3f                                   ccf                           ;set carry to zero (clear)
0253 ed 42                                sbc   hl,bc                   ;cont. if bc > or = hl (buffer address)
0255 da 61 02                             jp    c,decimal_continue      ;borrow means bc > hl
0258 ca 61 02                             jp    z,decimal_continue      ;z means bc = hl
025b 2a 86 0f                             ld    hl,(current_value)      ;return if de < buffer address (no borrow)
025e 37                                   scf                           ;get value back from RAM variable
025f 3f                                   ccf
0260 c9                                   ret                           ;return with carry clear, value in hl
```

```
0261 0a           decimal_continue:       ld    a,(bc)                 ;next char in string (right to left)
0262 d6 30                                sub   030h                   ;ASCII value of zero char
0264 fa 87 02                             jp    m,decimal_error        ;error if char value less than 030h
0267 fe 0a                                cp    00ah                   ;error if byte value > or = 10 decimal
0269 f2 87 02                             jp    p,decimal_error        ;a reg now has value of decimal numeral
026c 2a 84 0f                             ld    hl,(value_pointer)     ;get value to add an put in de
026f 5e                                   ld    e,(hl)                 ;little-endian (low byte in low memory)
0270 23                                   inc   hl
0271 56                                   ld    d,(hl)
0272 23                                   inc   hl                     ;hl now points to next value
0273 22 84 0f                             ld    (value_pointer),hl
0276 2a 86 0f                             ld    hl,(current_value)     ;get back current value
0279 3d           decimal_add:            dec   a                      ;add loop to increase total value
027a fa 81 02                             jp    m,decimal_add_done     ;end of multiplication
027d 19                                   add   hl,de
027e c3 79 02                             jp    decimal_add
0281 22 86 0f     decimal_add_done:       ld    (current_value),hl
0284 c3 4d 02                             jp    decimal_next_char
0287 37           decimal_error:          scf
0288 c9                                   ret
0289 c3 79 02                             jp    decimal_add
028c 01 00 0a 00 64 00 e8 03 10 27  decimal_place_value:    defw  1,10,100,1000,10000
0296              ;
0296              ;Memory dump
0296              ;Displays a 256-byte block of memory in 16-byte rows.
0296              ;Called with address of start of block in HL
0296 22 80 0f     memory_dump:            ld    (current_location),hl  ;store address of block to be displayed
0299 3e 00                                ld    a,000h
029b 32 83 0f                             ld    (byte_count),a         ;initialize byte count
029e 32 82 0f                             ld    (line_count),a         ;initialize line count
02a1 c3 d6 02                             jp    dump_new_line
02a4 2a 80 0f     dump_next_byte:         ld    hl,(current_location)  ;get byte address from storage,
02a7 7e                                   ld    a,(hl)                 ;get byte to be converted to string
02a8 23                                   inc   hl                     ;increment address and
02a9 22 80 0f                             ld    (current_location),hl  ;store back
02ac 21 88 0f                             ld    hl,buffer              ;location to store string
02af cd 85 01                             call  byte_to_hex_string     ;convert
```

```
02b2 21 88 0f                       ld    hl,buffer               ;display string
02b5 cd 15 01                       call  write_string
02b8 3a 83 0f                       ld    a,(byte_count)          ;next byte
02bb 3c                             inc   a
02bc ca 06 03                       jp    z,dump_done             ;stop when 256 bytes displayed
02bf 32 83 0f                       ld    (byte_count),a          ;not finished yet, store
02c2 3a 82 0f                       ld    a,(line_count)          ;end of line (16 characters)?
02c5 fe 0f                          cp    00fh                    ;yes, start new line
02c7 ca d6 02                       jp    z,dump_new_line
02ca 3c                             inc   a                       ;no, increment line count
02cb 32 82 0f                       ld    (line_count),a
02ce 3e 20                          ld    a,020h                  ;print space
02d0 cd 09 01                       call  write_char
02d3 c3 a4 02                       jp    dump_next_byte          ;continue
02d6 3e 00        dump_new_line:    ld    a,000h                  ;reset line count to zero
02d8 32 82 0f                       ld    (line_count),a
02db cd 86 03                       call  write_newline
02de 2a 80 0f                       ld    hl,(current_location)   ;location of start of line
02e1 7c                             ld    a,h                     ;high byte of address
02e2 21 88 0f                       ld    hl, buffer
02e5 cd 85 01                       call  byte_to_hex_string      ;convert
02e8 21 88 0f                       ld    hl,buffer
02eb cd 15 01                       call  write_string            ;write high byte
02ee 2a 80 0f                       ld    hl,(current_location)
02f1 7d                             ld    a,l                     ;low byte of address
02f2 21 88 0f                       ld    hl, buffer
02f5 cd 85 01                       call  byte_to_hex_string      ;convert
02f8 21 88 0f                       ld    hl,buffer
02fb cd 15 01                       call  write_string            ;write low byte
02fe 3e 20                          ld    a,020h                  ;space
0300 cd 09 01                       call  write_char
0303 c3 a4 02                       jp    dump_next_byte          ;now write 16 bytes
0306 3e 00        dump_done:        ld    a,000h
0308 21 88 0f                       ld    hl,buffer
030b 77                             ld    (hl),a                  ;clear buffer of last string
030c cd 86 03                       call  write_newline
030f c9                             ret
```

```
0310                ;
0310                ;Memory load
0310                ;Loads RAM memory with bytes entered as hex characters
0310                ;Called with address to start loading in HL
0310                ;Displays entered data in 16-byte rows.
0310 22 80 0f       memory_load:            ld    (current_location),hl
0313 21 4b 04                               ld    hl,data_entry_msg
0316 cd 15 01                               call  write_string
0319 c3 63 03                               jp    load_new_line
031c cd 7c 03       load_next_char:         call  get_char
031f fe 0d                                  cp    00dh                    ;return char entered?
0321 ca 78 03                               jp    z,load_done             ;yes, quit
0324 32 88 0f                               ld    (buffer),a
0327 cd 7c 03                               call  get_char
032a fe 0d                                  cp    00dh                    ;return?
032c ca 78 03                               jp    z,load_done             ;yes, quit
032f 32 89 0f                               ld    (buffer+1),a
0332 21 88 0f                               ld    hl,buffer
0335 cd c4 01                               call  hex_to_byte
0338 da 6e 03                               jp    c,load_data_entry_error ;non-hex character
033b 2a 80 0f                               ld    hl,(current_location)   ;get byte address from storage,
033e 77                                     ld    (hl),a                  ;store byte
033f 23                                     inc   hl                      ;increment address and
0340 22 80 0f                               ld    (current_location),hl   ;store back
0343 3a 88 0f                               ld    a,(buffer)
0346 cd 09 01                               call  write_char
0349 3a 89 0f                               ld    a,(buffer+1)
034c cd 09 01                               call  write_char
034f 3a 82 0f                               ld    a,(line_count)          ;end of line (16 characters)?
0352 fe 0f                                  cp    00fh                    ;yes, start new line
0354 ca 63 03                               jp    z,load_new_line
0357 3c                                     inc   a                       ;no, increment line count
0358 32 82 0f                               ld    (line_count),a
035b 3e 20                                  ld    a,020h                  ;print space
035d cd 09 01                               call  write_char
0360 c3 1c 03                               jp    load_next_char          ;continue
0363 3e 00          load_new_line:          ld    a,000h                  ;reset line count to zero
```

```
0365 32 82 0f                     ld    (line_count),a
0368 cd 86 03                     call  write_newline
036b c3 1c 03                     jp    load_next_char          ;continue
036e cd 86 03     load_data_entry_error:  call  write_newline
0371 21 78 04                     ld    hl,data_error_msg
0374 cd 15 01                     call  write_string
0377 c9                           ret
0378 cd 86 03     load_done:             call  write_newline
037b c9                           ret
037c              ;
037c              ;Get one ASCII character from the serial port.
037c              ;Returns with char in A reg. No error checking.
037c db 03        get_char:              in    a,(3)             ;get status
037e e6 02                        and   002h              ;check RxRDY bit
0380 ca 7c 03                     jp    z,get_char        ;not ready, loop
0383 db 02                        in    a,(2)             ;get char
0385 c9                           ret
0386              ;
0386              ;Subroutine to start a new line
0386 3e 0d        write_newline:         ld    a,00dh                  ;ASCII carriage return character
0388 cd 09 01                     call  write_char
038b 3e 0a                        ld    a,00ah                  ;new line (line feed) character
038d cd 09 01                     call  write_char
0390 c9                           ret
0391              ;
0391              ;Strings used in subroutines
0391 .. 00        length_entry_string:   defm  "Enter length of file to load (decimal): ",0
03ba .. 00        dump_entry_string:     defm  "Enter no. of bytes to dump (decimal): ",0
03e1 08 1b .. 00  erase_char_string:     defm  008h,01bh,"[K",000h     ;ANSI seq. for BS, erase to end of line.
03e6 .. 00        address_entry_msg:     defm  "Enter 4-digit hex address (use upper-case A through F): ",0
041f .. 00        address_error_msg:     defm  "\r\nError: invalid hex character, try again: ",0
044b .. 00        data_entry_msg:        defm  "Enter hex bytes, hit return when finished.\r\n",0
0478 .. 00        data_error_msg:        defm  "Error: invalid hex byte.\r\n",0
0493 .. 00        decimal_error_msg:     defm  "\r\nError: invalid decimal number, try again: ",0
04c0              ;
04c0              ;Simple monitor program for CPUville Z80 computer with serial interface.
04c0 cd 00 01     monitor_cold_start:    call  initialize_port
```

```
04c3 21 da 05                      ld    hl,monitor_message
04c6 cd 15 01                      call  write_string
04c9 cd 86 03     monitor_warm_start:     call  write_newline     ;return here to avoid re-initialization of port
04cc 3e 3e                         ld    a,03eh              ;prompt (cursor symbol)
04ce cd 09 01                      call  write_char
04d1 21 88 0f                      ld    hl,buffer
04d4 cd 49 01                      call  get_line            ;get monitor input string (command)
04d7 cd 86 03                      call  write_newline
04da cd de 04                      call  parse               ;interpret command, ret. With jump addr. in HL
04dd e9                            jp    (hl)
04de              ;
04de              ;Parses an input line stored in buffer for available commands as described in parse table.
04de              ;Returns with address of jump to action for the command in HL
04de 01 9f 07     parse:                  ld    bc,parse_table    ;bc is pointer to parse_table
04e1 0a           parse_start:            ld    a,(bc)            ;get pointer to match string from parse table
04e2 5f                            ld    e,a
04e3 03                            inc   bc
04e4 0a                            ld    a,(bc)
04e5 57                            ld    d,a                 ;de will is pointer to strings for matching
04e6 1a                            ld    a,(de)              ;get first char from match string
04e7 f6 00                         or    000h                ;zero?
04e9 ca 04 05                      jp    z,parser_exit       ;yes, exit no_match
04ec 21 88 0f                      ld    hl,buffer           ;no, parse input string
04ef be           match_loop:             cp    (hl)              ;compare buffer char with match string char
04f0 c2 fe 04                      jp    nz,no_match         ;no match, go to next match string
04f3 f6 00                         or    000h                ;end of strings (zero)?
04f5 ca 04 05                      jp    z,parser_exit       ;yes, matching string found
04f8 13                            inc   de                  ;match so far, point to next char
04f9 1a                            ld    a,(de)              ;get next character from match string
04fa 23                            inc   hl                  ;and point to next char in input string
04fb c3 ef 04                      jp    match_loop          ;check for match
04fe 03           no_match:               inc   bc                ;skip over jump target to
04ff 03                            inc   bc
0500 03                            inc   bc                  ;get address of next matching string
0501 c3 e1 04                      jp    parse_start
0504 03           parser_exit:            inc   bc                ;skip to address of jump for match
0505 0a                            ld    a,(bc)
```

```
0506 6f                             ld    l,a
0507 03                             inc   bc
0508 0a                             ld    a,(bc)
0509 67                             ld    h,a               ;returns with jump address in hl
050a c9                             ret
050b                 ;
050b                 ;Actions to be taken on match
050b                 ;
050b                 ;Memory dump program
050b                 ;Input 4-digit hexadecimal address
050b                 ;Calls memory_dump subroutine
050b 21 4e 06        dump_jump:             ld    hl,dump_message         ;Display greeting
050e cd 15 01                               call  write_string
0511 21 e6 03                               ld    hl,address_entry_msg    ;get ready to get address
0514 cd 15 01                               call  write_string
0517 cd fb 01                               call  address_entry           ;returns with address in HL
051a cd 86 03                               call  write_newline
051d cd 96 02                               call  memory_dump
0520 c3 c9 04                               jp    monitor_warm_start
0523                 ;
0523                 ;Hex loader, displays formatted input
0523 21 75 06        load_jump:             ld    hl,load_message         ;Display greeting
0526 cd 15 01                               call  write_string            ;get address to load
0529 21 e6 03                               ld    hl,address_entry_msg    ;get ready to get address
052c cd 15 01                               call  write_string
052f cd fb 01                               call  address_entry
0532 cd 86 03                               call  write_newline
0535 cd 10 03                               call  memory_load
0538 c3 c9 04                               jp    monitor_warm_start
053b                 ;
053b                 ;Jump and run do the same thing: get an address and jump to it.
053b 21 a4 06        run_jump:              ld    hl,run_message          ;Display greeting
053e cd 15 01                               call  write_string
0541 21 e6 03                               ld    hl,address_entry_msg    ;get ready to get address
0544 cd 15 01                               call  write_string
0547 cd fb 01                               call  address_entry
054a e9                                     jp    (hl)
```

```
054b                ;
054b                ;Help and ? do the same thing, display the available commands
054b 21 24 06       help_jump:              ld    hl,help_message
054e cd 15 01                               call  write_string
0551 01 9f 07                               ld    bc,parse_table    ;table with pointers to command strings
0554 0a             help_loop:              ld    a,(bc)            ;displays the strings for matching commands,
0555 6f                                     ld    l,a               ;getting the string addresses from the
0556 03                                     inc   bc                ;parse table
0557 0a                                     ld    a,(bc)            ;pass address of string to hl through a reg
0558 67                                     ld    h,a
0559 7e                                     ld    a,(hl)            ;hl now points to start of match string
055a f6 00                                  or    000h              ;exit if no_match string
055c ca 6f 05                               jp    z,help_done
055f c5                                     push  bc                ;write_char uses b register
0560 3e 20                                  ld    a,020h            ;space char
0562 cd 09 01                               call  write_char
0565 c1                                     pop   bc
0566 cd 15 01                               call  write_string      ;writes match string
0569 03                                     inc   bc                ;pass over jump address in table
056a 03                                     inc   bc
056b 03                                     inc   bc
056c c3 54 05                               jp    help_loop
056f c3 c9 04       help_done:              jp    monitor_warm_start
0572                ;
0572                ;Binary file load. Need both address to load and length of file
0572 21 d9 06       bload_jump:             ld    hl,bload_message
0575 cd 15 01                               call  write_string
0578 21 e6 03                               ld    hl,address_entry_msg
057b cd 15 01                               call  write_string
057e cd fb 01                               call  address_entry
0581 cd 86 03                               call  write_newline
0584 e5                                     push  hl
0585 21 91 03                               ld    hl,length_entry_string
0588 cd 15 01                               call  write_string
058b cd 26 02                               call  decimal_entry
058e 44                                     ld    b,h
058f 4d                                     ld    c,l
```

```
0590 21 fc 06                           ld    hl,bload_ready_message
0593 cd 15 01                           call  write_string
0596 e1                                 pop   hl
0597 cd 25 01                           call  bload
059a c3 c9 04                           jp    monitor_warm_start
059d                ;
059d                ;Binary memory dump. Need address of start of dump and no. bytes
059d 21 20 07       bdump_jump:         ld    hl,bdump_message
05a0 cd 15 01                           call  write_string
05a3 21 e6 03                           ld    hl,address_entry_msg
05a6 cd 15 01                           call  write_string
05a9 cd fb 01                           call  address_entry
05ac cd 86 03                           call  write_newline
05af e5                                 push  hl
05b0 21 ba 03                           ld    hl,dump_entry_string
05b3 cd 15 01                           call  write_string
05b6 cd 26 02                           call  decimal_entry
05b9 44                                 ld    b,h
05ba 4d                                 ld    c,l
05bb 21 50 07                           ld    hl,bdump_ready_message
05be cd 15 01                           call  write_string
05c1 cd 7c 03                           call  get_char
05c4 e1                                 pop   hl
05c5 cd 37 01                           call  bdump
05c8 c3 c9 04                           jp    monitor_warm_start
05cb                ;Prints message for no match to entered command
05cb 21 03 06       no_match_jump:      ld    hl,no_match_message
05ce cd 15 01                           call  write_string
05d1 21 88 0f                           ld    hl, buffer
05d4 cd 15 01                           call  write_string
05d7 c3 c9 04                           jp    monitor_warm_start
05da                ;
05da                ;Monitor data structures:
05da                ;
05da .. 00          monitor_message:  defm  "\r\nCPUville Z80 computer, ROM version 7\r\n",0
0603 .. 00          no_match_message: defm  "No match found for input string ",0
0624 .. 00          help_message:           defm  "The following commands are implemented:\r\n",0
```

```
064e .. 00        dump_message:           defm  "Displays a 256-byte block of memory.\r\n",0
0675 .. 00        load_message:           defm  "Enter hex bytes starting at memory location.\r\n",0
06a4 .. 00        run_message:            defm  "Will jump to (execute) program at address entered.\r\n",0
06d9 .. 00        bload_message:          defm  "Loads a binary file into memory.\r\n",0
06fc .. 00        bload_ready_message:    defm  "\n\rReady to receive, start transfer.",0
0720 .. 00        bdump_message:          defm  "Dumps binary data from memory to serial port.\r\n",0
0750 .. 00        bdump_ready_message:    defm  "\n\rReady to send, hit any key to start.",0
0777              ;Strings for matching:
0777 .. 00        dump_string:            defm  "dump",0
077c .. 00        load_string:            defm  "load",0
0781 .. 00        jump_string:            defm  "jump",0
0786 .. 00        run_string:             defm  "run",0
078a .. 00        question_string:        defm  "?",0
078c .. 00        help_string:            defm  "help",0
0791 .. 00        bload_string:           defm  "bload",0
0797 .. 00        bdump_string:           defm  "bdump",0
079d 00 00        no_match_string:        defm  0,0
079f              ;Table for matching strings to jumps
079f 77 07 0b 05 7c 07 23 05  parse_table:      defw  dump_string,dump_jump,load_string,load_jump
07a7 81 07 3b 05 86 07 3b 05                    defw  jump_string,run_jump,run_string,run_jump
07af 8a 07 4b 05 8c 07 4b 05                    defw  question_string,help_jump,help_string,help_jump
07b7 91 07 72 05 97 07 9d 05                    defw  bload_string,bload_jump,bdump_string,bdump_jump
07bf 9d 07 cb 05                                defw  no_match_string,no_match_jump
07c3
# End of file 2K_ROM_7.asm
07c3
```

```
# File RAM_test_1.asm
0000                                                          ;Program to test Program Loader
0000                                                          ;Simple output and halt
0000                                    org   0800h           ;Address of start of RAM
0800 3e 05                              ld    a,005h          ;Bit pattern for port 0
0802 d3 00                              out   (000h),a        ;Output pattern to port
0804 3e 0a                              ld    a,000ah         ;Bit pattern for port 1
0806 d3 01                              out   (001h),a        ;Output pattern to port
0808 76                                 halt
# End of file RAM_test_1.asm

# File Highest_factor_2.asm
0000                                                          ;Highest Factor program
0000                                                          ;Calculates highest factor of a one-byte number
0000                                                          ;read from input port 0, and displays it on
0000                                                          ;output port 0. Displays the number itself
0000                                                          ;if it is a prime number.
0000                                    org   00800h          ;Start of RAM
0800 3e 00        Program_start:        ld    a,000h          ;Clear output ports
0802 d3 00                              out   (000h),a
0804 d3 01                              out   (001h),a
0806 db 00        Get_number:           in    a,(000h)        ;Get one byte number to factor
0808 32 40 08                           ld    (Original_number),a     ;Store original number
080b 32 41 08                           ld    (Test_factor),a
080e 3a 41 08     Factor_test:          ld    a,(Test_factor)
0811 3d                                 dec   a
0812 ca 06 08                           jp    z,Get_number    ;Don't try to divide by 0
0815 fe 01                              cp    001h
0817 ca 2b 08                           jp    z,Prime         ;No more factors to test
081a 32 41 08                           ld    (Test_factor),a ;Store factor for next test
081d 47                                 ld    b,a
081e 3a 40 08                           ld    a,(Original_number)
0821 90           Factor_loop:          sub   a,b       ;Serial subtraction for division
0822 fa 0e 08                           jp    m,Factor_test   ;Too far, try next factor
0825 ca 35 08                           jp    z,Factor        ;Exact divisor = factor
0828 c3 21 08                           jp    Factor_loop ;Register a still positive, keep subtracting
082b 3a 40 08     Prime:                ld    a,(Original_number)
```

```
082e d3 00                      out   (000h),a
0830 d3 01                      out   (001h),a
0832 c3 06 08                   jp    Get_number
0835 3a 40 08     Factor:       ld    a,(Original_number)
0838 d3 00                      out   (000h),a
083a 78                         ld    a,b
083b d3 01                      out   (001h),a
083d c3 06 08                   jp    Get_number
0840                                                  ;Variables
0840 00           Original_number:  defb  000h
0841 00           Test_factor:  defb  000h
0842
# End of file Highest_factor_2.asm

# File adder_1.asm
0000                            org   0800h             ;Start of RAM
0800 3e 00        Add_Program:  ld    a,00h             ;Clear outputs to start
0802 d3 00                      out   (0),a
0804 d3 01                      out   (1),a
0806 db 00        Get_addends:  in    a,(0)             ;Get 8-bit addends
0808 47                         ld    b,a               ;Store one in B register
0809 db 01                      in    a,(1)             ;Get the other
080b 80                         add   a,b               ;Add them
080c d3 00                      out   (0),a             ;Output the result
080e 3e 00                      ld    a,00h             ;Clear port 1 LEDs
0810 d3 01                      out   (1),a
0812 d2 06 08                   jp    nc,Get_addends    ;All done if no carry
0815 3e 01                      ld    a,01h             ;If carry, put 1 on port 1
0817 d3 01                      out   (1),a
0819 c3 06 08                   jp    Get_addends       ;Start again
# End of file adder_1.asm
```

Table for hand assembling a program

Memory address (hexadecimal)	Machine code (hexadecimal)	Label	Assembly language	Comment
0800				
0801				
0802				
0803				
0804				
0805				
0806				
0807				
0808				
0809				
080A				
080B				
080C				
080D				
080E				
080F				
0810				
0811				
0812				
0813				
0814				
0815				
0816				
0817				
0818				
0819				

081A				
081B				
081C				
081D				
081E				
081F				
0820				
0821				
0822				
0823				
0824				
0825				
0826				
0827				
0828				
0829				
082A				
082B				
082C				
082D				
082E				
082F				
0830				

Resources

Web Sites

“Home of the Z80 CPU”. Lots of resources including links to assemblers. http://www.z80.info/

"8080/Z80 Instruction Set”. Web page with complete operation codes and mnemonics for Z80 assembly language. http://nemesis.lonestar.org/computers/tandy/software/apps/m4/qd/opcodes.html

"CPU World Z80 Page”. Information about Z80 from different manufacturers around the world, with photos. http://www.cpu-world.com/CPUs/Z80/index.html

"Z80 Family CPU User Manual”. Full pdf of the compete Z80 reference from Zilog, 1.8 Mb. http://www.zilog.com/docs/z80/um0080.pdf

"SDCC – Small Device C Compiler” Want to program the Z80 using C instead of assembly language? Get this compiler. http://sdcc.sourceforge.net/

"The Z80 Microprocessor”. An old Sourceforge page with information about the Z80, including links to instuction set table. http://penguicon.sourceforge.net/comphist/links/cpm/z80.html

“Z80 Instruction Set (Complete)” Good web page of the Z80 instruction set. http://www.ftp83plus.net/Tutorials/z80inset_fullA.html

Books

Z80 Assembly Language Programming by Lance Leventhal, 1979, Osborne/McGraw-Hill, Berkeley, California. The book I used to learn Z80, very complete.

Z80 Microprocessor Family User's Manual, 1995, Zilog Inc, Campbell, California. Complete reference. You can get this from the Zilog web side (see above).

Build Your Own Z80 Computer by Steve Ciarcia, 1981, BYTE Books/McGraw-Hill, Peterborough, New Hampshire. The classic book from which I got most of the information used to design my computer. You can view the book for free on Google Books.

Z-80 Microcomputer Design Projects by William Barden, Jr, 1980, Howard W. Sams & Co., Inc., Indianapolis, Indiana. Lots of additional information about small Z80 systems.

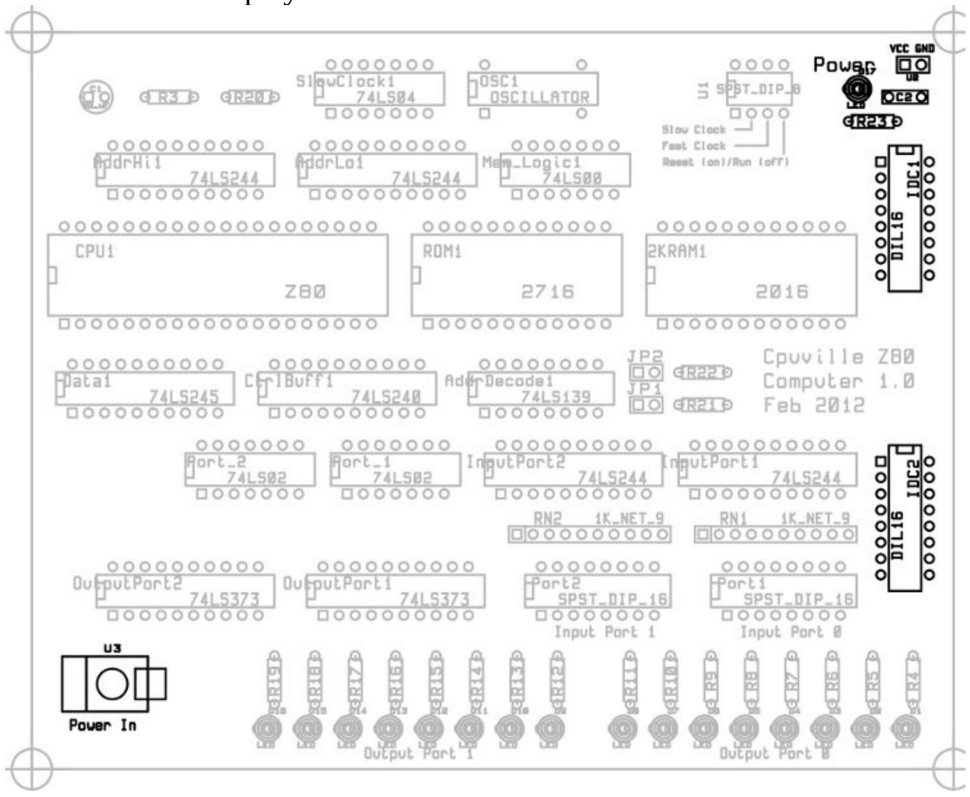
Engineer's Notebook II: A Handbook of Integrated Circuit Applications by Forrest M. Mims, III, 1982, Radio Shack. My reference for designing the digital support circuits for the computer, display, and logic probe kits. This is probably out of print, but look for other books by this author, they will always be well-written.

Computer Organization & Design: The Hardware/Software Interface by David A. Patterson and John L. Hennessy, 1998, Morgan Kaufmann Publishers, Inc., San Francisco, California. Complete, college-level textbook by the designer of the MIPS family of microprocessors. This book taught me how to build a processor (see my web site main page, cpuville.com).

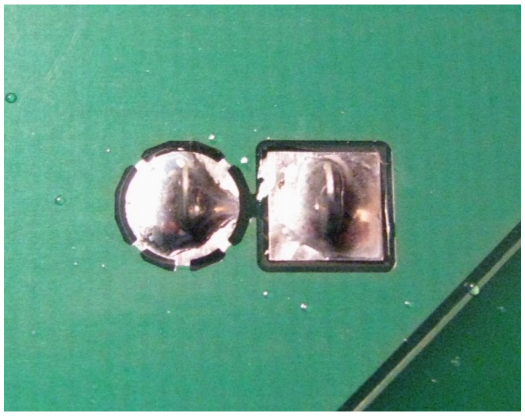
Supplementary Materials: Building by Sections

If you have a logic probe, you can make the kit in sections, and test the function of each section before you go on to the next one. Building this way is a little more educational, but it is a little more difficult, because you will put in some tall parts at the start. That makes it harder to solder in the shorter parts later. But, you can use a little folded paper or styrofoam to hold the parts against the board when it is upside down, or use a little solder drop to hold a part in place while you solder the other pins (see Soldering Tips). Here are the sections:

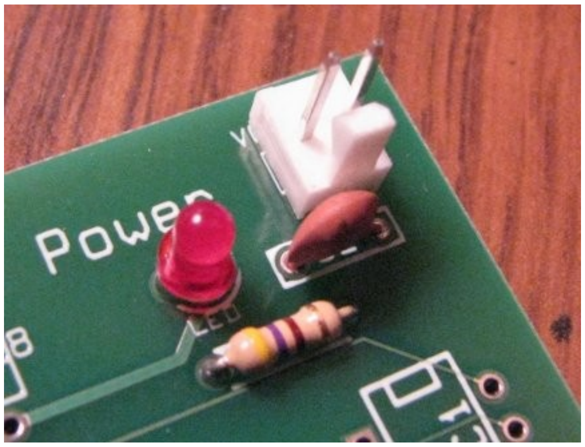
1. Power section and display connectors.

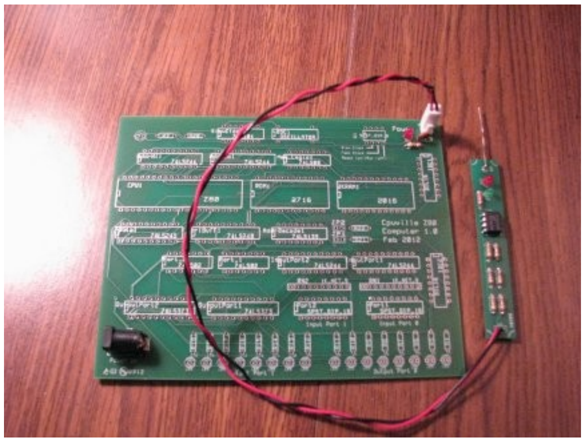


You might have to apply some force on the power jack pins to get them to go through the holes. When you solder it, just fill up the holes with solder. (I made the board with round holes instead of slots because holes are about $3 cheaper).



If you bought a CPUville logic probe, you can solder in the connector and capacitor at the right upper corner and plug it in.

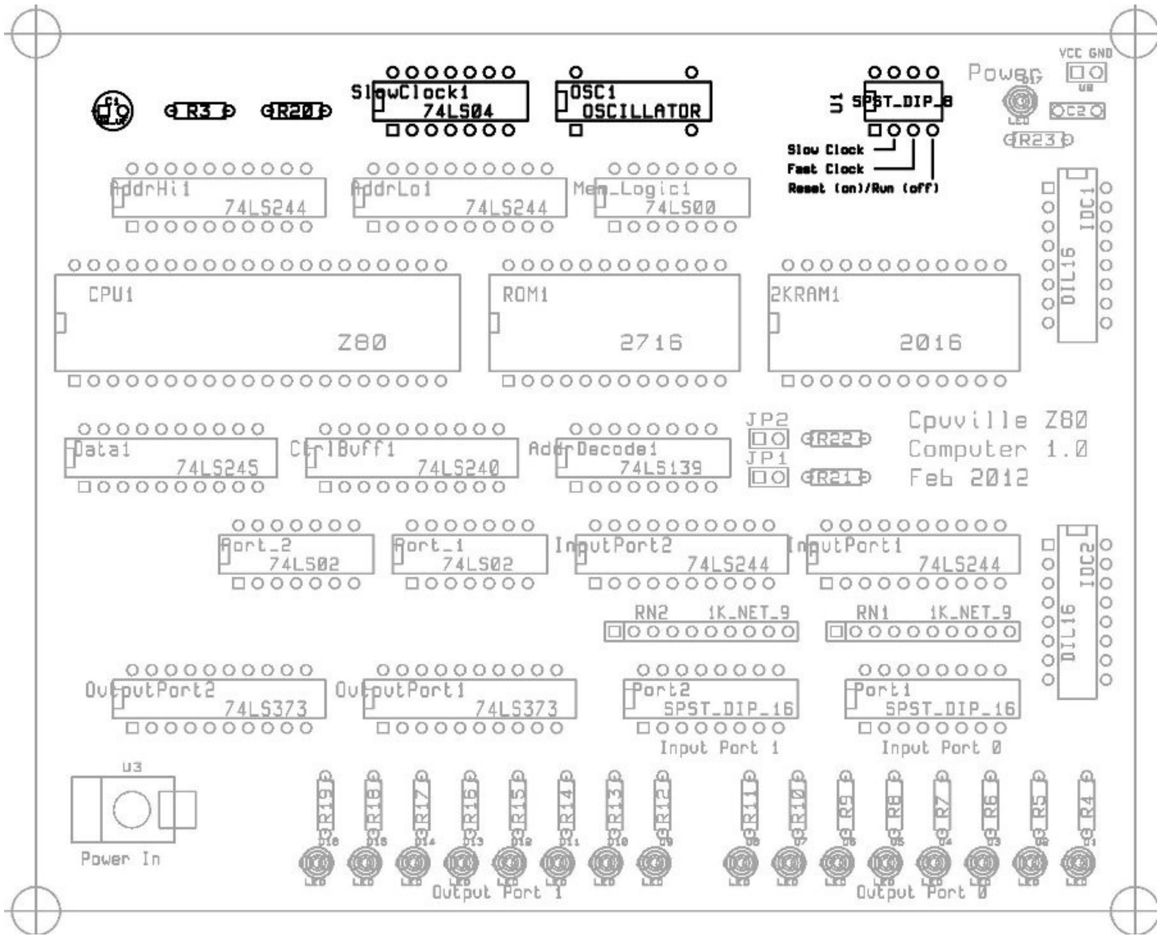




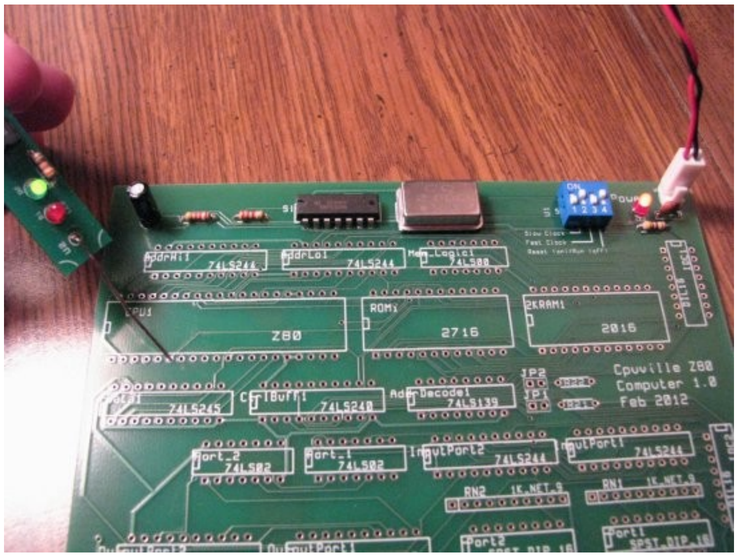
Apply +5V Regulated DC[16] to the board. With a logic probe, you can check that many pads on the board now have either high (+5V) or low (ground) levels on them.

16 This project requires a +5V **regulated** DC power supply capable of at least 2000 mA (i.e., a 10 watt power supply). An unregulated power supply will not work properly and may damage the system.

2. Clock and reset section.

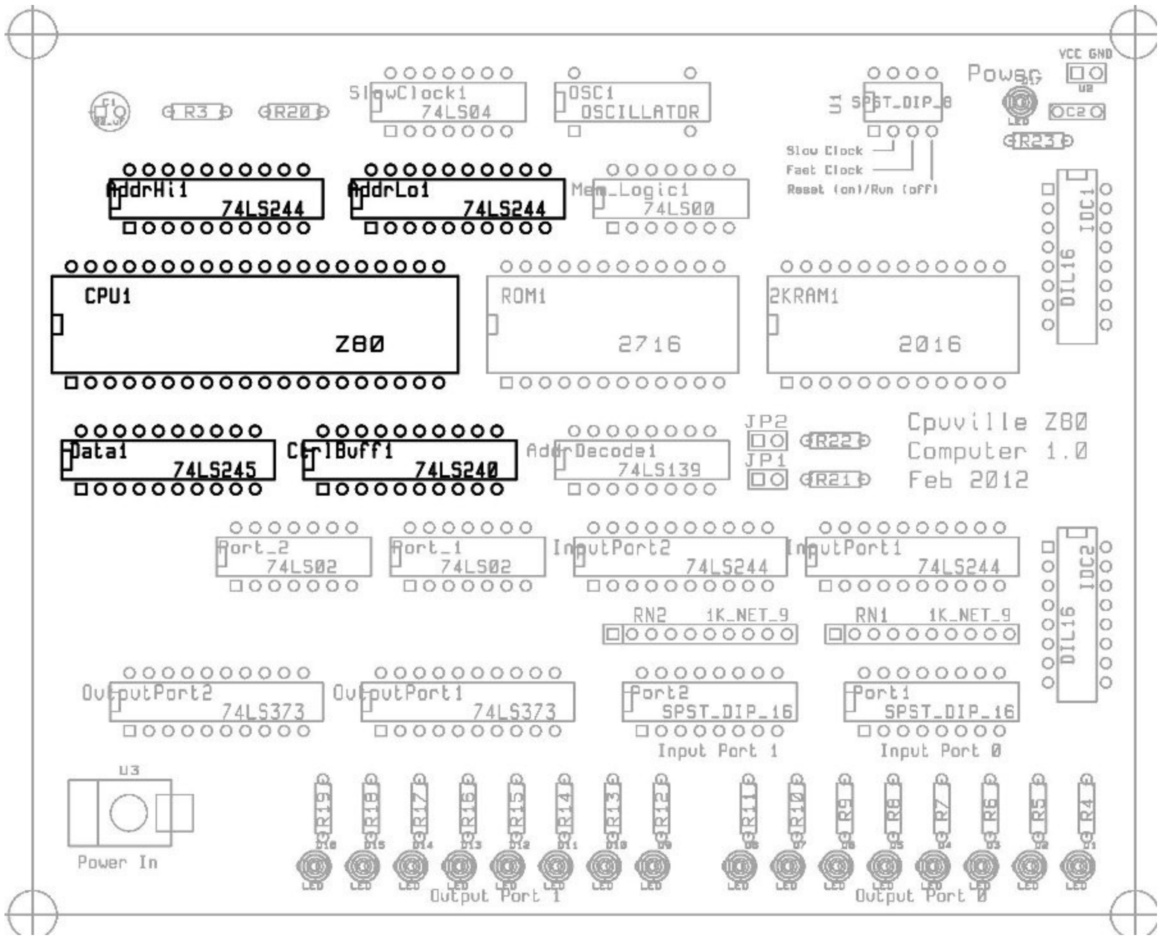


After finishing this, you can use the switches to select either the fast or slow clock for the Z80. The Z80 clock input is pin 6. Test pin 6 with the logic probe with the slow clock selected, and you will see it cycling. (The logic probe will not detect the fast clock with the current board configuration).

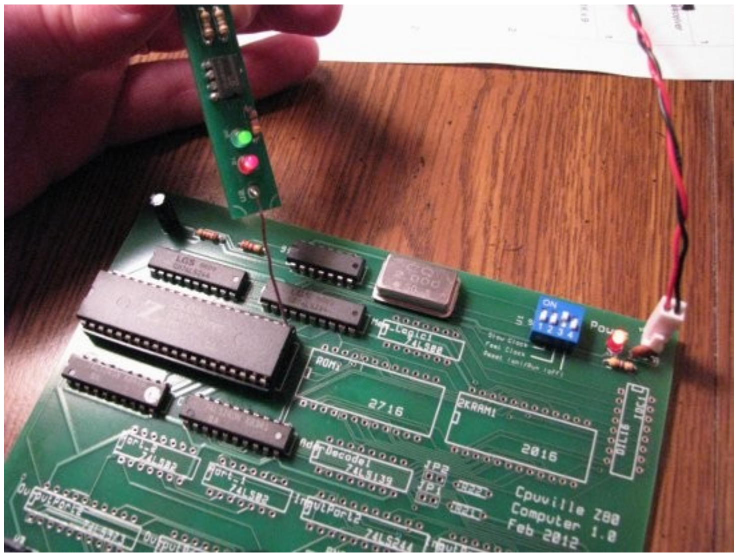


When the Reset switch is on, the Reset input on the Z80 (pin 26) should be low (ground), and when the Reset switch is off, the Reset input should be high (+5V). If you have built the display board, you can solder in the sockets (IDC1 and IDC2), connect it, and see activity on the Clock and Reset LEDs.

3. Z80 and buffers.



After finishing this section, you can actually run the Z80. Select the slow clock, and set the Reset switch off. Since the Z80 is not connected to the memory yet it won't be doing anything interesting, but it won't damage it to run it. The Address and Data pins should be cycling when it is running.

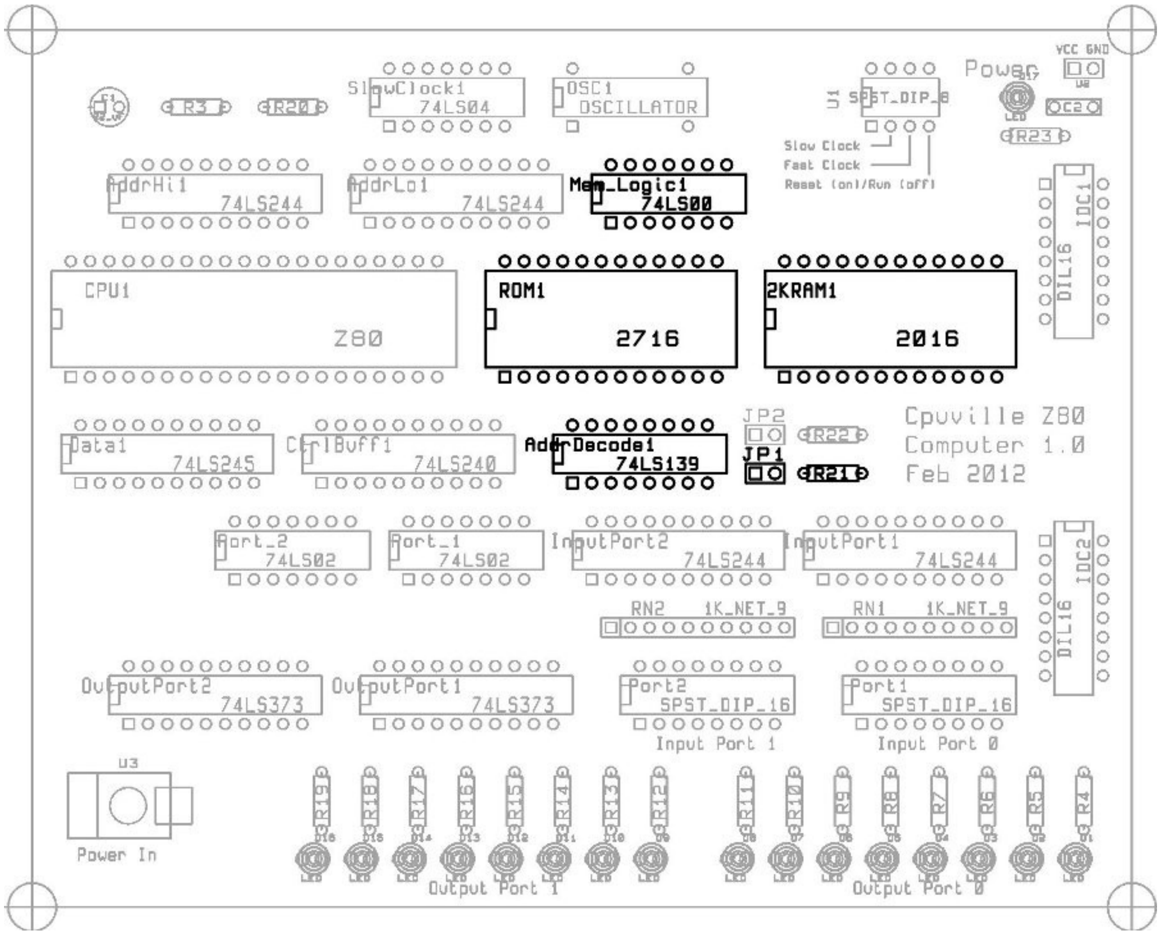


You can look at the other pins, and you should see this:

Unused inputs pins 16, 17, 24 and 25: High
Unused outputs pins 27 and 28: Cycling
Unused outputs pins 18 and 23: High
Control pins 19 and 21: Cycling
Control pins 20 and 22: High most of the time, might cycle occasionally
Clock pin 6: Cycling
Reset pin 26: High
Vcc (power in) pin 11: High
Gnd (power in) pin 29: Low

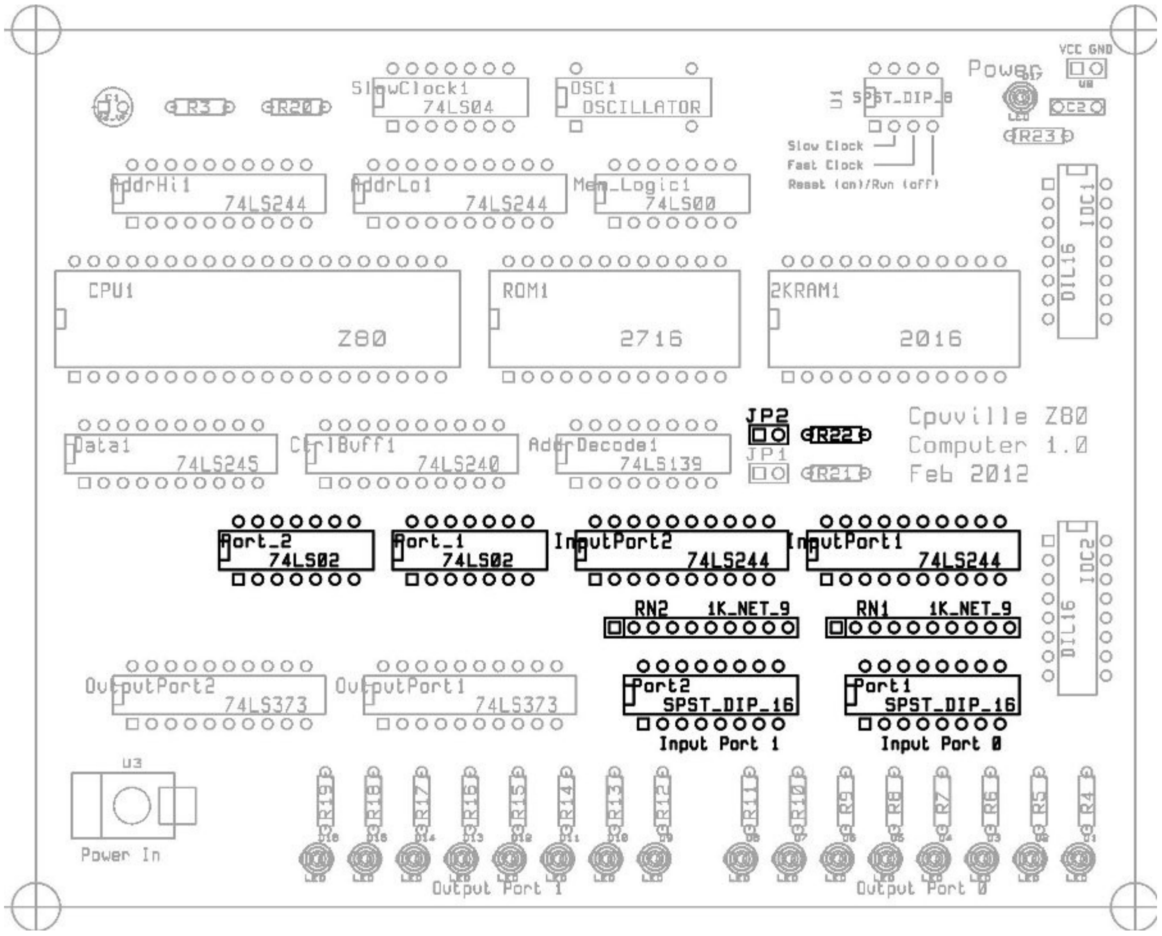
The address, data, and control bus buffers should show the same behavior on their output pins. If you put the processor into reset the cycling should stop (except for the clock signal).

4. Memory section.

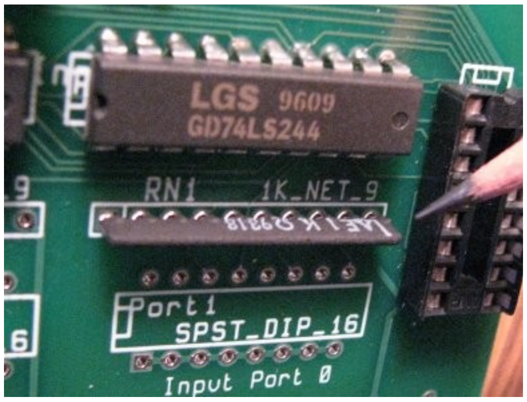


This section has the 2K EPROM with the window in it, and the 2K static RAM, as well as the decoding logic (explained in the section on the schematics). The JP1 jumper is for disabling the on-board memory in case you want to make an add-on board with its own memory. If you run the computer now, you won't notice much difference from running the CPU only, except you might have more activity on the I/O Req and Write pins (20 and 22).

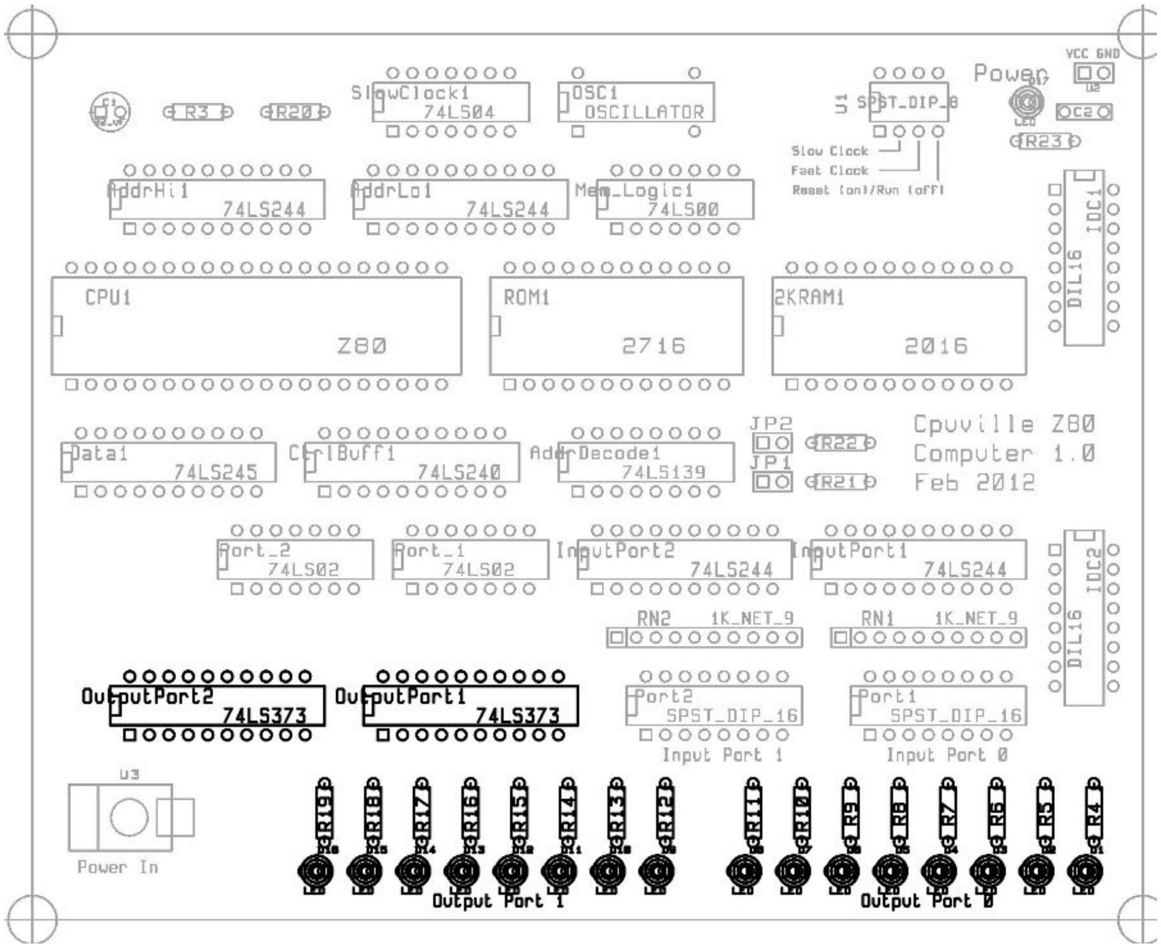
5. Input ports section.



This section has the input port switches, resistor networks, buffers that act as gateways to the data bus, the port logic, and a jumper. The JP2 jumper disables the on-board input and output ports in case you want to make an add-on board with its own ports. Be careful to solder the resistor networks in with the marked pin to the RIGHT:



6. Output ports section.



This section includes the output port LEDs, current-limiting resistors, and the latches that grab and hold the data for display. Make sure you put the LEDs in with the short lead and flat side of the flange to the RIGHT.