CPUville Z80 Computer Kit Instruction Manual

By Donn Stewart

© 2015 by Donn Stewart
Table of Contents

Introduction........................................................................................................................................3
Building Tips.......................................................................................................................................5
Building the Logic Probe....................................................................................................................9
Building the Display........................................................................................................................12
Building the Computer.....................................................................................................................15
Binary, briefly.......................................................................................................................................22
Testing the Computer........................................................................................................................26
Z80 Programming................................................................................................................................29
Computers in General.........................................................................................................................42
The CPUville Z80 Computer System.................................................................................................44
Computer block diagram..................................................................................................................45
Computer Schematics and Explanations............................................................................................46
  Clocks and Reset.................................................................................................................................46
  Connectors.......................................................................................................................................48
  Z80 CPU and Buffers.........................................................................................................................49
  Control Bus Buffer and Decoders......................................................................................................51
  2K ROM...........................................................................................................................................53
  2K RAM...........................................................................................................................................55
  Input Ports.........................................................................................................................................56
  Output Ports......................................................................................................................................58
Display Schematic and Explanation.....................................................................................................60
Logic Probe Schematic and Explanation............................................................................................62
Appendix.............................................................................................................................................64
  Logic Probe parts organizer and list.................................................................................................64
  Display parts organizer and list.........................................................................................................65
  Computer parts organizer and list.....................................................................................................66
  Program Listings.................................................................................................................................68
  Table for hand assembling a program...............................................................................................90
Resources...........................................................................................................................................92
  Web Sites..........................................................................................................................................92
  Books...............................................................................................................................................92
Supplementary Materials: Building by Sections..................................................................................94
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 online 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/Z80.htm](http://cpuville.com/Z80.htm). 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.
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 interface, 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 1/8\textsuperscript{th} 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:

![Image of pads with connections to zones]

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.

6. 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

If you bought all three kits (logic probe, display, and computer), start by building the logic probe since it is the easiest.

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

2. Some 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. IC: The LEFT-hand side of the IC has a little cut-out:

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

3. Connector wires: The red wire goes in the BOTTOM hole (the one closest to you):

3. Resistors do not have to be oriented.
4. The logic probe circuit board is small and light, you will probably need to hold it down with the handle of a tool or with a small clamp in order to solder without pushing it all over the table.
5. Be careful to put the correct resistors in the correct places. Check the parts list, and the resistor color code to be sure they match.
6. You can test the logic probe when you build the computer power section. If you want to test it sooner, you can insert solid-core wires into the connector housing, and connect +5V regulated
DC\(^5\) to the red wire side, and connect the black wire side to ground. Then, if you touch the probe tip to a +5V connection in the same circuit, the red LED should come on. If you touch it to ground the green LED should come on. If the probe is touching a dead connection, or is in the air, neither LED should come on.

7. Handle the logic probe by the edges. It is very sensitive. If you touch any of the leads with a finger you can get a false signal. If this makes it too hard to work with you can wrap electrical tape around it to insulate it.

---

\(^5\) 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.
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\(^6\) 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

---

\(^6\) 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.
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:
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\(^7\). 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:

---

\(^7\) Some early computers, such as ENIAC, used decimal numbers, and had circuits with 10 stable states.
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:
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:  
1111 1100
+0010 0101
0010 0001

Decimal equivalent:
252
+37
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:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0010</td>
<td>+2</td>
</tr>
<tr>
<td>0000 0001</td>
<td>+1</td>
</tr>
<tr>
<td>0000 0000</td>
<td>+0</td>
</tr>
<tr>
<td>1111 1111</td>
<td>-1</td>
</tr>
<tr>
<td>1111 1110</td>
<td>-2</td>
</tr>
<tr>
<td>1111 1101</td>
<td>-3</td>
</tr>
<tr>
<td>1111 1100</td>
<td>-127</td>
</tr>
<tr>
<td>1111 1100</td>
<td>-128</td>
</tr>
<tr>
<td>0111 1111</td>
<td>+127</td>
</tr>
<tr>
<td>0111 1110</td>
<td>+126</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1100</td>
<td>+12</td>
</tr>
<tr>
<td>+1111 1110</td>
<td>+(-2)</td>
</tr>
<tr>
<td>0000 1010</td>
<td>+10</td>
</tr>
</tbody>
</table>

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 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:

**Port reflector, address 0x001F (binary 0000 0000 0001 1111)**
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.

**Simple counter, address 0x002A (binary 0000 0000 0010 1010)**
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.

**Count to a million, address 0x0032 (binary 0000 0000 0011 0010)**
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.

**Program loader, address 0x0046 (binary 0000 0000 0100 0110)**
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)

---

8 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.
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.
   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.

Memory test, address 0x0074 (binary 0000 0000 0111 0100)
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.

---

9 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. 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. 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,

---

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

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

12 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”.

---

12 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”.

---

31
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Operand1</th>
<th>Operand2</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DJNZ disp</td>
<td>10</td>
<td>disp</td>
<td></td>
<td>Decrement B, jump disp&lt;sup&gt;13&lt;/sup&gt; if not zero (operation used as a counter)</td>
</tr>
<tr>
<td>HALT</td>
<td>76</td>
<td></td>
<td></td>
<td>Stop CPU execution. Only interrupt can restart</td>
</tr>
<tr>
<td>JP (HL)</td>
<td>E9</td>
<td></td>
<td></td>
<td>Jump to the address indicated by HL</td>
</tr>
<tr>
<td>JP addr</td>
<td>C3</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Jump to address (unconditional jump)</td>
</tr>
<tr>
<td>JP C,addr</td>
<td>DA</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Jump to address if carry flag set (C = 1)</td>
</tr>
<tr>
<td>JP NC,addr</td>
<td>D2</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Jump to address if carry flag cleared (C = 0)</td>
</tr>
<tr>
<td>JP Z, addr</td>
<td>CA</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Jump to address if zero flag set (Z = 1)</td>
</tr>
<tr>
<td>JP NZ, addr</td>
<td>C2</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Jump to address if zero flag cleared (Z = 0)</td>
</tr>
<tr>
<td>JP M,addr</td>
<td>FA</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Jump to address if minus flag set (M = 1)</td>
</tr>
<tr>
<td>JP P,addr</td>
<td>F2</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Jump to address if minus flag cleared (M = 0)</td>
</tr>
</tbody>
</table>

<sup>13</sup> 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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Operand1</th>
<th>Operand2</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD (addr),A</td>
<td>32</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Load memory location with contents of A</td>
</tr>
<tr>
<td>LD (HL),A</td>
<td>77</td>
<td></td>
<td></td>
<td>Load memory location indicated by HL with A</td>
</tr>
<tr>
<td>LD A,(addr)</td>
<td>3A</td>
<td>addr lo</td>
<td>addr hi</td>
<td>Load A with contents of memory location</td>
</tr>
<tr>
<td>LD A,(HL)</td>
<td>7E</td>
<td></td>
<td></td>
<td>Load A with memory location indicated by HL</td>
</tr>
<tr>
<td>LD A,B</td>
<td>78</td>
<td></td>
<td></td>
<td>Load A with contents of B</td>
</tr>
<tr>
<td>LD A,H</td>
<td>7C</td>
<td></td>
<td></td>
<td>Load A with contents of H</td>
</tr>
<tr>
<td>LD A,L</td>
<td>7D</td>
<td></td>
<td></td>
<td>Load A with contents of L</td>
</tr>
<tr>
<td>LD A,number</td>
<td>3E</td>
<td>number</td>
<td></td>
<td>Load A with an 8-bit number</td>
</tr>
<tr>
<td>LD B,A</td>
<td>47</td>
<td></td>
<td></td>
<td>Load B with contents of A</td>
</tr>
<tr>
<td>LD B,number</td>
<td>06</td>
<td>number</td>
<td></td>
<td>Load B with an 8-bit number</td>
</tr>
<tr>
<td>LD H,A</td>
<td>67</td>
<td></td>
<td></td>
<td>Load H with the contents of A</td>
</tr>
<tr>
<td>LD H,number</td>
<td>26</td>
<td>number</td>
<td></td>
<td>Load H with an 8-bit number</td>
</tr>
<tr>
<td>LD HL,number</td>
<td>21</td>
<td>num lo</td>
<td>num hi</td>
<td>Load HL with a 16-bit number</td>
</tr>
<tr>
<td>LD L,A</td>
<td>6F</td>
<td></td>
<td></td>
<td>Load L with the contents of A</td>
</tr>
<tr>
<td>LD L,number</td>
<td>2E</td>
<td>number</td>
<td></td>
<td>Load L with an 8-bit number</td>
</tr>
<tr>
<td>OUT (port),A</td>
<td>D3</td>
<td>port</td>
<td></td>
<td>Place the 8-bit contents of A onto an output port</td>
</tr>
<tr>
<td>IN A,(port)</td>
<td>DB</td>
<td>port</td>
<td></td>
<td>Place the 8-bit value from port into the A register</td>
</tr>
</tbody>
</table>
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 lower-memory address location, and the high-order byte will be in the higher memory address location. This is called little-endian byte order\textsuperscript{14}. 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
```

\textsuperscript{14} 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:
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                         |                           |       |                   |         
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                         |                           |       |                   |         
0802                         |                           |       |                   |         
```

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:

<table>
<thead>
<tr>
<th>Memory address (hexadecimal)</th>
<th>Machine code (hexadecimal)</th>
<th>Label</th>
<th>Assembly language</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>3E</td>
<td>Add_Program:</td>
<td>ld a,00h</td>
<td>;Clear outputs to start</td>
</tr>
<tr>
<td>0801</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0802</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Memory address (hexadecimal)</th>
<th>Machine code (hexadecimal)</th>
<th>Label</th>
<th>Assembly language</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>3E</td>
<td>Add_Program:</td>
<td>ld a,00h</td>
<td>;Clear outputs to start</td>
</tr>
<tr>
<td>0801</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0802</td>
<td>D3</td>
<td></td>
<td>out (0),a</td>
<td></td>
</tr>
<tr>
<td>0803</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Memory address (hexadecimal)</th>
<th>Machine code (hexadecimal)</th>
<th>Label</th>
<th>Assembly language</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>3E</td>
<td>Add_Program:</td>
<td>ld a,00h</td>
<td>;Clear outputs to start</td>
</tr>
<tr>
<td>0801</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0802</td>
<td>D3</td>
<td></td>
<td>out (0),a</td>
<td></td>
</tr>
<tr>
<td>0803</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory address (hexadecimal)</td>
<td>Machine code (hexadecimal)</td>
<td>Label</td>
<td>Assembly language</td>
<td>Comment</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td>----------------</td>
<td>-------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>0800</td>
<td>3E</td>
<td>Add_Program:</td>
<td>ld a,00h</td>
<td>;Clear outputs to start</td>
</tr>
<tr>
<td>0801</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:
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:

<table>
<thead>
<tr>
<th>Memory address (hexadecimal)</th>
<th>Machine code (hexadecimal)</th>
<th>Label</th>
<th>Assembly language</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>3E</td>
<td>Add_Program:</td>
<td>ld a, 00h</td>
<td>;Clear outputs to start</td>
</tr>
<tr>
<td>0801</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some assembly examples:
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0802</td>
<td>D3</td>
<td><code>out (0),a</code></td>
</tr>
<tr>
<td>0803</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0804</td>
<td>D3</td>
<td><code>out (1),a</code></td>
</tr>
<tr>
<td>0805</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>0806</td>
<td>DB</td>
<td><code>Get_addends: in a,(0)</code> ;Get 8-bit addends</td>
</tr>
<tr>
<td>0807</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0808</td>
<td>47</td>
<td><code>ld b,a</code> ;Store one in B register</td>
</tr>
<tr>
<td>0809</td>
<td>DB</td>
<td><code>in a,(1)</code> ;Get the other</td>
</tr>
<tr>
<td>080A</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>080B</td>
<td>80</td>
<td><code>add a,b</code> ;Add them</td>
</tr>
<tr>
<td>080C</td>
<td>D3</td>
<td><code>out (0),a</code> ;Output the result</td>
</tr>
<tr>
<td>080D</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>080E</td>
<td>3E</td>
<td><code>ld a,00h</code> ;Clear port 1 LEDs</td>
</tr>
<tr>
<td>080F</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0810</td>
<td>D3</td>
<td><code>out (1),a</code></td>
</tr>
<tr>
<td>0811</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>0812</td>
<td>D2</td>
<td><code>jp nc,Get_addends</code> ;All done if no carry</td>
</tr>
<tr>
<td>0813</td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>0814</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>0815</td>
<td>3E</td>
<td><code>ld a,01h</code> ;If carry, put 1 on port 1</td>
</tr>
<tr>
<td>0816</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>0817</td>
<td>D3</td>
<td><code>out (1),a</code></td>
</tr>
<tr>
<td>0818</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>0819</td>
<td>C3</td>
<td><code>jp Get_addends</code> ;Start again</td>
</tr>
<tr>
<td>081A</td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>081B</td>
<td>08</td>
<td></td>
</tr>
</tbody>
</table>
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\textsuperscript{15} 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 π 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.\textsuperscript{16}

\textsuperscript{15} Z80 is a registered trademark of the Zilog corporation.
\textsuperscript{16} 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.
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.
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 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 activate 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.
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:

<table>
<thead>
<tr>
<th>SelectMem_1K*</th>
<th>SelectMem_2K*</th>
<th>AND result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 (not possible)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 (chip selected)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 (chip selected)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (chip not selected)</td>
</tr>
</tbody>
</table>

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 and 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.
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:

<table>
<thead>
<tr>
<th>SelectMem_3K*</th>
<th>SelectMem_4K*</th>
<th>AND result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 (not possible)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 (chip selected)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 (chip selected)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (chip not selected)</td>
</tr>
</tbody>
</table>
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:
<table>
<thead>
<tr>
<th>SelectI/OPort_0*</th>
<th>Read*</th>
<th>OR result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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:
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).

<table>
<thead>
<tr>
<th>SelectI/O_0*</th>
<th>Write*</th>
<th>NOR result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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 is not a digital device. It is an analog device. The resistor chains that are connected between Vcc and ground act as voltage dividers. When Vcc is +5V, as it is in this computer system, the voltage at the op amp pins 3 and 6 is half the total voltage drop of the resistor chain, or 2.5 V. This voltage serves as a reference voltage for the two op-amps. When the probe is not touching anything, the voltage at the probe connection, which is halfway through the other resistor chain, will also be 2.5V. The voltage drop from Vcc to pin 2 will be 1/1.1 x 2.5 = 2.27, so the voltage will be 5 – 2.27 = 2.73 V. Similarly, the voltage drop from the probe connection to pin 5 will be 0.1/1.1 x 2.5 = 0.23, so the voltage will be 2.5 – 0.23 = 2.27 V. So when the probe tip is not touching anything, the voltage difference between the pins on the top op amp is 2.5 – 2.73 = -0.23, and the op amp output is 0. The voltage difference between the pins on the bottom op amp will be 2.27 – 2.5 = -0.23, and this op amp output will also be 0. So, no current will flow between the two op amp outputs, and neither LED will light up. When the probe touches a +5V signal, the voltage drop across the two resistors on top of the chain is now 0, and the difference between the inputs on the top op amp will be 2.5 – 5 = -2.5, and the output will be 0. However, for the bottom op amp, the voltage drop from the probe connection to ground will now be +5V. The voltage drop from the probe connection to pin 3 will be 0.1/1.1 x 5 = 0.45, and the voltage will be
5 – 0.45 = 4.55V. Now the difference between the inputs on the bottom op amp will be 4.55 – 2.5 = 2.05, and the output will be +5V. Now, current will flow from the bottom op amp into the top op amp, lighting the red LED. When the probe tip touches 0V, the situation is reversed.
Appendix

Logic Probe parts organizer and list

<table>
<thead>
<tr>
<th>Reference</th>
<th>Value</th>
<th>Quantity</th>
<th>Value</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Red LED</td>
<td>1</td>
<td>1 Meg</td>
<td>4</td>
</tr>
<tr>
<td>D2</td>
<td>Green LED</td>
<td>1</td>
<td>100K</td>
<td>2</td>
</tr>
<tr>
<td>R1</td>
<td>1 Meg</td>
<td>1</td>
<td>1 Meg</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>100K</td>
<td>1</td>
<td>1 Meg</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>100K</td>
<td>1</td>
<td>1 Meg</td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td>1 Meg</td>
<td>1</td>
<td>1 Meg</td>
<td>1</td>
</tr>
<tr>
<td>R5</td>
<td>1 Meg</td>
<td>1</td>
<td>1 Meg</td>
<td>1</td>
</tr>
<tr>
<td>R6</td>
<td>1 Meg</td>
<td>1</td>
<td>1 Meg</td>
<td>1</td>
</tr>
<tr>
<td>R7</td>
<td>330 ohm</td>
<td>1</td>
<td>TL082</td>
<td>1</td>
</tr>
<tr>
<td>U1</td>
<td>TL082</td>
<td>1</td>
<td>Probe tip</td>
<td>1</td>
</tr>
<tr>
<td>U2</td>
<td>PROBE</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U3</td>
<td>Power connector</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Logic probe parts list
### Display parts organizer and list

<table>
<thead>
<tr>
<th>Capacitor, 0.01 uF disk</th>
<th>Red LED</th>
<th>Resistor, 470 ohm</th>
<th>DIL16 connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>74LS240</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Part Description</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacitor, 22 uF, 16V</td>
<td>1</td>
</tr>
<tr>
<td>Resistor, 2.2K, ¼ watt Red-Red-Red</td>
<td>2</td>
</tr>
<tr>
<td>74LS00 quad NAND</td>
<td>1</td>
</tr>
<tr>
<td>74LS02 quad NOR</td>
<td>2</td>
</tr>
<tr>
<td>74LS04 hex inverter</td>
<td>1</td>
</tr>
<tr>
<td>SPST_DIP_16 switches</td>
<td>2</td>
</tr>
<tr>
<td>SPST_DIP_8 switches</td>
<td>1</td>
</tr>
<tr>
<td>DIL_16 socket</td>
<td>2</td>
</tr>
<tr>
<td>DIL_24 socket</td>
<td>1</td>
</tr>
<tr>
<td>74LS139 decoder</td>
<td>1</td>
</tr>
<tr>
<td>2016 2K RAM</td>
<td>1</td>
</tr>
<tr>
<td>2716 2K ROM</td>
<td>1</td>
</tr>
<tr>
<td>74LS373 latch</td>
<td>2</td>
</tr>
<tr>
<td>74LS240 octal buffer, inverting</td>
<td>1</td>
</tr>
<tr>
<td>74LS244 octal buffer</td>
<td>4</td>
</tr>
<tr>
<td>74LS245 bus transceiver</td>
<td>1</td>
</tr>
<tr>
<td>Z80 CPU</td>
<td>1</td>
</tr>
<tr>
<td>Resistor, 470 ohm, ¼ watt Yellow-Violet-Brown</td>
<td>17</td>
</tr>
<tr>
<td>Resistor, 1K, ¼ watt Brown-Black-Red</td>
<td>2</td>
</tr>
<tr>
<td>Resistor network, 1K x 9</td>
<td>2</td>
</tr>
<tr>
<td>LED</td>
<td>17</td>
</tr>
<tr>
<td>Oscillator</td>
<td>1</td>
</tr>
<tr>
<td>2-pin header</td>
<td>2</td>
</tr>
<tr>
<td>Shorting block</td>
<td>2</td>
</tr>
<tr>
<td>Capacitor, 0.01 uF disk</td>
<td>1</td>
</tr>
<tr>
<td>Header with clip, 2 pin</td>
<td>1</td>
</tr>
<tr>
<td>Power-in jack</td>
<td>1</td>
</tr>
<tr>
<td>DIL 40-pin socket</td>
<td>1</td>
</tr>
<tr>
<td>Ref</td>
<td>Value</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>2KRAM1</td>
<td>2016</td>
</tr>
<tr>
<td>AddrDecode1</td>
<td>74LS139</td>
</tr>
<tr>
<td>AddrHi1</td>
<td>74LS244</td>
</tr>
<tr>
<td>AddrLo1</td>
<td>74LS244</td>
</tr>
<tr>
<td>C1</td>
<td>22 uF</td>
</tr>
<tr>
<td>C2</td>
<td>0.01 uF</td>
</tr>
<tr>
<td>CPU1</td>
<td>Z80</td>
</tr>
<tr>
<td>CtrlBuff1</td>
<td>74LS240</td>
</tr>
<tr>
<td>D1</td>
<td>LED</td>
</tr>
<tr>
<td>D2</td>
<td>LED</td>
</tr>
<tr>
<td>D3</td>
<td>LED</td>
</tr>
<tr>
<td>D4</td>
<td>LED</td>
</tr>
<tr>
<td>D5</td>
<td>LED</td>
</tr>
<tr>
<td>D6</td>
<td>LED</td>
</tr>
<tr>
<td>D7</td>
<td>LED</td>
</tr>
<tr>
<td>D8</td>
<td>LED</td>
</tr>
<tr>
<td>D9</td>
<td>LED</td>
</tr>
<tr>
<td>D10</td>
<td>LED</td>
</tr>
<tr>
<td>D11</td>
<td>LED</td>
</tr>
<tr>
<td>D12</td>
<td>LED</td>
</tr>
<tr>
<td>D13</td>
<td>LED</td>
</tr>
<tr>
<td>D14</td>
<td>LED</td>
</tr>
<tr>
<td>D15</td>
<td>LED</td>
</tr>
<tr>
<td>D16</td>
<td>LED</td>
</tr>
<tr>
<td>D17</td>
<td>LED</td>
</tr>
<tr>
<td>Data1</td>
<td>74LS245</td>
</tr>
<tr>
<td>IDC1</td>
<td>DIL16</td>
</tr>
<tr>
<td>IDC2</td>
<td>DIL16</td>
</tr>
<tr>
<td>InputPort1</td>
<td>74LS244</td>
</tr>
<tr>
<td>InputPort2</td>
<td>74LS244</td>
</tr>
<tr>
<td>JP1</td>
<td>JUMPER (2-pin header with shorting block)</td>
</tr>
<tr>
<td>JP2</td>
<td>JUMPER (2-pin header with shorting block)</td>
</tr>
<tr>
<td>Mem_Logic1</td>
<td>74LS00</td>
</tr>
<tr>
<td>OSC1</td>
<td>OSCILLATOR</td>
</tr>
<tr>
<td>OutputPort1</td>
<td>74LS373</td>
</tr>
<tr>
<td>OutputPort2</td>
<td>74LS373</td>
</tr>
<tr>
<td>Port1</td>
<td>SPST_DIP_16</td>
</tr>
<tr>
<td>Port2</td>
<td>SPST_DIP_16</td>
</tr>
<tr>
<td>Port_1</td>
<td>74LS02</td>
</tr>
<tr>
<td>Port_2</td>
<td>74LS02</td>
</tr>
</tbody>
</table>
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

```assembly
0000  org  00000h
0000   Start_of_RAM:  equ  0x0000
0003   Get_address:  jp  Get_address  ;Skip over message
0008   defm  "CPUville Z80 ROM v.7",0
0018   db  00  Get_address:  in  a,(0)  ;Get address from input ports
001a   ld  l,a
001b   in  a,(1)
001d   ld  h,a
001e   jp  (hl)  ;Jump to the address
0021   in  a,(0)  ;Simple program to test ports
0022   ld  l,a
0023   in  a,(1)
0025   out  (1),a
0027   jp  Port_Reflector
002a   in  a,(0)  ;One-byte counter for slow clock
002c   ld  a,000h
002e   inc  a
0032   Count_to_a_million:  ld  l,000h  ;Two-byte (16-bit) counter
0034   ld  h,000h  ;Clear registers
0036   Loop_2:  ld  a,010h  ;Count 16 times, then
0038   dec  a
003a   Loop_3:  dec  a  ;Do it again
003c   inc  hl  ;increment the 16-bit number
003d   ld  a,l
003e   out  (0),a  ;Output the 16-bit number
0040   Loop_4:  out  (1),a
0043   Loop_2:  out  (0),a
0046   Program_loader:  ld  hl,Start_of_RAM  ;Load a program in RAM
0049   in  a,(1)
004b   and  081h  ;Check input port 1
004d   z,Loop_4  ;If switches 0 and 7 open, loop
0050   call  debounce
0053   in  a,(1)  ;Get input port byte again
0055   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
                     ;a byte was stored)
0061 d3 01   Loop_6: in a,(1)   ;Wait for switch to open
0063 db 01               and 001h  
0065 e6 01            out (0),a    ;No, then right switch (bit 0) closed.
0066 c2 63 00         ld a,l     ;Put low byte of address on port 1
006a cd f5 00       call debounce
006d 7d               jp nz,Loop_6
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
007c 23               inc hl      ;Point to next location in RAM
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  
0087 7d             ld a,l    ;should be 4K (cycled around to ROM)
0088 d3 00               out (0),a  
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  
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  

Loop_9:  in a,(1) ;Look for switch closure
        and 001h
        jp z,Loop_9
        call debounce
        ld a,0ffh ;Light port 1 LEDs
        out (1),a
        in a,(0) ;Get hi byte
        ld h,a ;Put in reg H
        Loop_11: in a,(1) ;Look for switch open
        and 001h
        jp nz,Loop_11
        call debounce
        ld a,h ;Show hi byte on port 1
        out (1),a
        Loop_13: in a,(1) ;Look for switch closure
        and 001h
        jp z,Loop_13
        call debounce
        ld a,0ffh ;Light port 0 LEDs
        out (0),a
        in a,(0) ;Get lo byte
        ld l,a ;Put in reg L
        Loop_15: in a,(1) ;Look for switch open
        and 001h
        jp nz,Loop_15
        call debounce
        ld a,l ;Show lo byte on port 0
        out (0),a
        Loop_17: in a,(1) ;Look for switch closure
        and 001h
        jp z,Loop_17
        call debounce
        in a,(0) ;Get byte to load
        ld (hl),a ;Store in memory
        Loop_19: in a,(1) ;Look for switch open
        and 001h
jp  nz,Loop_19
call  debounce
jp  Poke ;Start over

;Subroutine for a switch debounce delay
debounce:  ld  a,010h ;Outer loop
debounce_loop:  ld  b,0ffh ;Inner loop
djnz  $+0 ;Loop here until B reg is zero
dec  a
jp  nz,debounce_loop
ret

;The following code is for a system with a serial port.
;Assumes the UART data port address is 02h and control/status address is 03h

;The subroutines for the serial port use these variables in high RAM:
current_location:  equ  0x0f80 ;word variable in RAM
line_count:  equ  0x0f82 ;byte variable in RAM
byte_count:  equ  0x0f83 ;byte variable in RAM
value_pointer:  equ  0x0f84 ;word variable in RAM
current_value:  equ  0x0f86 ;word variable in RAM
buffer:  equ  0x0f88 ;buffer in RAM -- up to stack area

;Subroutine to initialize serial port UART
;Needs to be called only once after computer comes out of reset.
;if called while port is active will cause port to fail.
;16x = 9600 baud
initialize_port:  ld  a,04eh ;1 stop bit, no parity, 8-bit char, 16x baud
out  (3),a ;write to control port
ld  a,037h ;enable receive and transmit
out  (3),a ;write to control port
ret

;Puts a single char (byte value) on serial output
;Call with char to send in A register. Uses B register
write_char:  ld  b,a ;store char
write_char_loop:  in  a,(3) ;check if OK to send
Subroutine to get a string from serial input, place in buffer. Buffer address passed in HL reg.

; Uses A,BC,DE,HL registers (including calls to other subroutines).
; Line entry ends by hitting return key. Return char not included in string (replaced by zero).
; Backspace editing OK. No error checking.

get_line:
ld c,000h ; line position
ld a,h ; put original buffer address in de
ld d,a ; after this don't need to preserve hl
ld a,l ; subroutines called don't use de
ld e,a

get_line_next_char:
in a,(3) ; get status
and 002h ; check RxRDY bit
jp z,get_line_next_char ; not ready, loop
in a,(2) ; get char
cp 00dh ; check if return
ret z ; yes, normal exit
cp 07fh ; check if backspace (VT102 keys)
jp z,get_line_backspace ; yes, jump to backspace routine
cp 008h ; check if backspace (ANSI keys)
jp z,get_line_backspace ; yes, jump to backspace

get_line_backspace:
call write_char ; put char on screen
ld (de),a ; store char in buffer

write_char:
ld a,000h ; point to next space in buffer
inc de ; inc counter
inc c
ld (de),a ; leaves a zero-terminated string in buffer
get_line_backspace:    ld a,c               ;check current position in line
    cp 000h               ;at beginning of line?
    jp z,get_line_next_char ;yes, ignore backspace, get next char
    dec de               ;no, erase char from buffer
    dec c                ;back up one
    ld a,000h            ;replace last char with zero
    ld (de),a
    ld hl,erase_char_string ;ANSI seq. to delete one char
    call write_string    ;backspace and erase char
    jp get_line_next_char

byte_to_hex_string:    ld b,a               ;store original byte
    srl a               ;shift right 4 times, putting
    srl a
    srl a
    srl a
    ld d,000h           ;prepare for 16-bit addition
    ld e,a              ;de contains offset
    push hl             ;temporarily store string target address
    ld hl,hex_char_table ;use char table to get high-nybble character
    add hl,de            ;add offset to start of table
    ld a,(hl)            ;get char
    pop hl              ;get string target address
    ld (hl),a            ;store first char of string
    inc hl              ;point to next string target address
    ld a,b               ;get original byte back from reg b
    and 00fh             ;mask off high-nybble
    ld e,a               ;d still has 000h, now de has offset
    push hl              ;temp store string target address
    ld hl,hex_char_table ;start of table
    add hl,de            ;add offset
    ld a,(hl)            ;get char
pop hl ;get string target address
ld (hl),a ;store second char of string
inc hl ;point to third location
ld a,000h ;zero to terminate string
ld (hl),a ;store the zero
ret ;done

;Converts a single ASCII hex char to a nybble value
;Pass char in reg A. Letter numerals must be upper case.
;Return nybble value in low-order reg A with zeros in high-order nybble if no error.
;Return 0ffh in reg A if error (char not a valid hex numeral).
;Also uses b, c, and hl registers.

hex_char_to_nybble:
ld hl,hex_char_table
ld b,00fh ;no. of valid characters in table - 1.
ld c,000h ;will be nybble value
hex_to_nybble_loop:
cp (hl) ;character match here?
jp z,hex_to_nybble_ok ;match found, exit
dec b ;no match, check if at end of table
jp m,hex_to_nybble_err ;table limit exceeded, exit with error
inc c ;still inside table, continue search
inc hl
jp hex_to_nybble_loop
ld a,c ;put nybble value in a
ret

hex_to_byte:
ld a,(hl) ;location of character pair
push hl ;store hl (hex_char_to_nybble uses it)
call hex_char_to_nybble
pop hl ;returns with nybble in a reg, or 0ffh if error
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  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+2  ;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
address_entry_error:    ld   hl,address_error_msg
                   call  write_string
                   jp    address_entry

decimal_entry:    ld   hl,buffer
                   call  get_line       ;returns with DE pointing to terminating zero
                   ld   hl,buffer
                   call  decimal_string_to_word
                   ret nc                ;no error, return with word in hl
                   ld   hl,decimal_error_msg ;error, try again
                   call  write_string
                   jp    decimal_entry

decimal_string_to_word:    ld   b,d
                   ld   c,e
                   ld   (current_location),hl ;store addr. of start of buffer in RAM
                   ld   hl,000h      ;starting value zero
                   ld   hl,decimal_place_value ;pointer to values
                   ld   (value_pointer),hl
                   dec   bc           ;next char in string (moving R to L)
                   ld   hl,(current_location) ;check if at end of decimal string
                   scf
                   ccf
                   jp    decimal_continue ;borrow means bc > hl
                   scf
                   ccf
                   ld   hl,(current_value) ;return if de < buffer address (no borrow)
                   scf
                   ccf
                   ret                ;return with carry clear, value in hl
```assembly
decimal_continue:  ld  a,(bc)  ;next char in string (right to left)
sub 030h  ;ASCII value of zero char
j p m,decimal_error  ;error if char value less than 030h
cp 00ah  ;error if byte value > or = 10 decimal
j p p,decimal_error  ;a reg now has value of decimal numeral
ld hl,(value_pointer)  ;get value to add an put in de
ld e,(hl)  ;little-endian (low byte in low memory)
inc hl
ld d,(hl)
inc hl  ;hl now points to next value
ld (value_pointer),hl
ld hl,(current_value)  ;get back current value
decimal_add:  dec a  ;add loop to increase total value
jp m,decimal_add_done  ;end of multiplication
add hl,de
jp decimal_add
memory_dump:
;Displays a 256-byte block of memory in 16-byte rows.
;Called with address of start of block in HL
memory_dump:
ld (current_location),hl  ;store address of block to be displayed
ld a,000h
ld (byte_count),a  ;initialize byte count
ld (line_count),a  ;initialize line count
jp dump_new_line
dump_next_byte:  ld hl,(current_location)  ;get byte address from storage,
ld a,(hl)  ;get byte to be converted to string
inc hl  ;increment address and
ld (current_location),hl  ;store back
ld hl,buffer  ;location to store string
call byte_to_hex_string  ;convert
```
ld hl,buffer ;display string

call write_string

ld a,(byte_count) ;next byte

inc a

jp z,dump_done ;stop when 256 bytes displayed

ld (byte_count),a ;not finished yet, store

ld a,(line_count) ;end of line (16 characters)?

cp 00fh ;yes, start new line

jp z,dump_new_line

inc a ;no, increment line count

ld (line_count),a

ld a,020h ;print space

call write_char

jp dump_next_byte ;continue

ld (line_count),a

call write_newline

ld hl,(current_location) ;location of start of line

ld a,h ;high byte of address

ld hl,buffer

call write_string ;write high byte

ld hl,(current_location)

ld a,l ;low byte of address

ld hl,buffer

call byte_to_hex_string ;convert

ld hl,buffer

call write_string ;write low byte

ld a,020h ;space

call write_char

jp dump_next_byte ;now write 16 bytes

ld a,000h

ld hl,buffer

call byte_to_hex_string ;convert

ld hl,buffer

call write_string ;write low byte

ld a,020h ;space

call write_char

jp dump_next_byte ;now write 16 bytes

ld a,000h

ld hl,buffer

call write_string

ld a,020h ;clear buffer of last string

call write_newline

call write_string

ld a,000h

ld hl,buffer

call byte_to_hex_string ;convert

ld hl,buffer

call write_string ;write low byte

ld a,020h ;space

call write_char

ld a,000h

ld hl,buffer

call write_string

ld a,020h ;clear buffer of last string

call write_newline

ret
; Memory load
; Loads RAM memory with bytes entered as hex characters
; Called with address to start loading in HL
; Displays entered data in 16-byte rows.

memory_load:  ld (current_location),hl
               ld hl,data_entry_msg
               call write_string
               jp load_new_line

load_next_char:  call get_char
               cp 00dh ; return char entered?
               jp z,load_done ; yes, quit
               ld (buffer),a
               call get_char
               cp 00dh ; return?
               jp z,load_done ; yes, quit
               ld (buffer+1),a
               call hex_to_byte
               jp c,load_data_entry_error ; non-hex character
               ld hl,(current_location) ; get byte address from storage,
               ld (hl),a ; store byte
               inc hl ; increment address and
               ld (current_location),hl ; store back
               ld a,(buffer)
               call write_char
               ld a,(buffer+1)
               call write_char
               ld a,(line_count) ; end of line (16 characters)?
               cp 00fh ; yes, start new line
               jp z,load_new_line
               inc a ; no, increment line count
               ld (line_count),a
               ld a,020h ; print space
               call write_char
               jp load_next_char ; continue

load_new_line:  ld a,000h ; reset line count to zero
ld (line_count), a

;continue

ld hl, data_entry_error
call write_newline

ld hl, data_error_msg
call write_string

ret

ld hl, load_data_entry_error:
call write_newline

ld hl, load_done:
call write_newline

;Get one ASCII character from the serial port.
;Returns with char in A reg. No error checking.

in a, (3) ;get status
and 002h ;check RxRDY bit
jp z, get_char ;not ready, loop
in a, (2) ;get char
ret

;Subroutine to start a new line

ld a, 00dh ;ASCII carriage return character
call write_char
ld a, 00ah ;new line (line feed) character
call write_char
ret

;Strings used in subroutines

length_entry_string: defm "Enter length of file to load (decimal): ", 0

dump_entry_string: defm "Enter no. of bytes to dump (decimal): ", 0

erase_char_string: defm 008h, 01bh, "[K", 000h ;ANSI seq. for BS, erase to end of line.

address_entry_msg: defm "Enter 4-digit hex address (use upper-case A through F): ", 0

address_error_msg: defm "\n\nError: invalid hex character, try again: ", 0

data_entry_msg: defm "Enter hex bytes, hit return when finished.\r\n", 0

data_error_msg: defm "\n\nError: invalid hex byte.\r\n", 0

decimal_error_msg: defm "\n\nError: invalid decimal number, try again: ", 0

;Simple monitor program for CPUville Z80 computer with serial interface.

monitor_cold_start: call initialize_port
ld hl,monitor_message

ld a,03eh ;prompt (cursor symbol)

ld hl,buffer

call get_line ;get monitor input string (command)

ld hl,buffer

call write_newline

;return here to avoid re-initialization of port

;Parses an input line stored in buffer for available commands as described in parse table.

;Returns with address of jump to action for the command in HL

ld bc,parse_table ;bc is pointer to parse_table

ld a,(bc) ;get pointer to match string from parse table

ld e,a

inc bc

ld a,(de) ;de will is pointer to strings for matching

ld a,(de) ;get first char from match string

or 000h ;zero?

jp z,parser_exit ;yes, exit no match

ld hl,buffer ;no, parse input string

or 000h ;end of strings (zero)?

jp z,parser_exit ;yes, matching string found

inc de ;match so far, point to next char

ld a,(de) ;get next character from match string

inc hl ;and point to next char in input string

jp match_loop ;check for match

inc bc ;skip over jump target to

inc bc

inc bc ;get address of next matching string

jp parse_start

inc bc ;skip to address of jump for match

ld a,(bc)
;Actions to be taken on match
;Memory dump program
;Input 4-digit hexadecimal address
;Calls memory_dump subroutine

dump_jump:    ld  hl,dump_message ;Display greeting
              call  write_string
              ld  hl,address_entry_msg ;get ready to get address
              call  write_string
              call  address_entry ;returns with address in HL
              call  write_newline
              call  memory_dump
              jp    monitor_warm_start

;Hex loader, displays formatted input
load_jump:    ld  hl,load_message ;Display greeting
              call  write_string ;get address to load
              ld  hl,address_entry_msg ;get ready to get address
              call  write_string
              call  address_entry
              call  write_newline
              call  memory_load
              jp    monitor_warm_start

;Jump and run do the same thing: get an address and jump to it.
run_jump:     ld  hl,run_message ;Display greeting
              call  write_string
              ld  hl,address_entry_msg ;get ready to get address
              call  write_string
              call  address_entry
              jp    (hl)
;Help and ? do the same thing, display the available commands
help_jump:    ld  hl,help_message
              call write_string
help_loop:   ld  a,(bc)        ;displays the strings for matching commands,
              inc  bc        ;parse table
              ld  a,(bc)     ;pass address of string to hl through a reg
              ld  h,a
              ld  a,(hl)     ;hl now points to start of match string
              or  000h       ;exit if no_match string
              jp  z,help_done
              push bc       ;write_char uses b register
              ld  a,020h     ;space char
              call write_char
              pop bc
              call write_string        ;writes match string
              inc  bc        ;pass over jump address in table
              inc  bc
              inc  bc
              jp  help_loop
help_done:   jp  help_loop
             ;Binary file load. Need both address to load and length of file
bload_jump:  ld  hl,bload_message
              call write_string
              ld  hl,address_entry_msg
              call write_string
              call address_entry
              call write_newline
              push hl
              ld  hl,length_entry_string
              call write_string
              call decimal_entry
              ld  b,h
              ld  c,l
Binary memory dump. Need address of start of dump and no. bytes

bdump_jump:
  ld hl, bdump_message
  call write_string
  ld hl, address_entry_msg
  call write_string
  call address_entry
  call write_newline
  push hl
  ld hl, dump_entry_string
  call write_string
  ld b, h
  ld c, l
  ld hl, bdump_ready_message
  call write_string
  call get_char
  pop hl
  call bdump
  j p monitor_warm_start

 Prints message for no match to entered command

no_match_jump:
  ld hl, no_match_message
  call write_string
  ld hl, buffer
  call write_string
  j p monitor_warm_start

; Prints message for no match to entered command

; Monitor data structures:

; Monitor message: defm \nCPUville Z80 computer, ROM version 7 \n
; No match message: defm "No match found for input string ", 0

; The following commands are implemented:

}
dump_message: defm "Displays a 256-byte block of memory.\r\n",0
load_message: defm "Enter hex bytes starting at memory location.\r\n",0
run_message: defm "Will jump to (execute) program at address entered.\r\n",0
bload_message: defm "Loads a binary file into memory.\r\n",0
bload_ready_message: defm "\n\rReady to receive, start transfer.\",0
bdump_message: defm "Dumps binary data from memory to serial port.\r\n",0
bdump_ready_message: defm "\n\rReady to send, hit any key to start.\",0
;
Strings for matching:
dump_string: defm "dump",0
load_string: defm "load",0
jump_string: defm "jump",0
run_string: defm "run",0
question_string: defm "?",0
help_string: defm "help",0
bload_string: defm "bload",0
bdump_string: defm "bdump",0
no_match_string: defm 0,0
;
Table for matching strings to jumps
dump_string,dump_jump,load_string,load_jump
jump_string,run_jump,run_string,run_jump
question_string,help_jump,help_string,help_jump
bload_string,bload_jump,bdump_string,bdump_jump
no_match_string,no_match_jump

# End of file 2K_ROM_7.asm
# 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)
Highest_factor_2.asm

; Variables
Original_number: defb 000h
Test_factor: defb 000h

adder_1.asm

; Start of RAM
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
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

# End of file adder_1.asm
Table for hand assembling a program

<table>
<thead>
<tr>
<th>Memory address (hexadecimal)</th>
<th>Machine code (hexadecimal)</th>
<th>Label</th>
<th>Assembly language</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0801</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0802</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0803</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0804</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0805</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0806</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0807</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0808</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0809</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>080A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>080B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>080C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>080D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>080E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>080F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0810</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0811</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0812</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0813</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0814</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0815</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0816</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0817</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0818</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0819</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Resources

Web Sites

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


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


"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 instruction set table. http://penguicon.sourceforge.net/comphist/links/cpm/z80.html


Books

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


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.

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\textsuperscript{17} 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.

\textsuperscript{17} 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.