CPUville 8-bit Processor Kit Instruction Manual

By Donn Stewart

Table of Contents

Introduction

The CPUville 8-bit processor is a general purpose, accumulator-memory computer processor in kit form. The processor is implemented on three 6.5 by 4.5 inch two-layer circuit boards using 74LS series TTL integrated circuits. The processor bus architecture allows it to replace the Z80 processor in the CPUville Original and Single-board Z80 computers. An accessory register display, together with the slow and single-step system clocks on the processor control board, allow the hobbyist or student to fully examine the inner workings of the processor. This processor kit is intended for educational and recreational purposes. It should not be used to control processes or machinery where system failure would result in damage or injury.

The three boards that make up the processor are the arithmetic-logic unit (ALU), the main board, and the control board. The ALU is a logic circuit that performs addition, subtraction, and logical operations on 8-bit operands. The main board is the processor data path which has the processor registers and multiplexers that direct the data to the places appropriate to the instruction being executed. The control board has the logic circuits that interpret the program instructions and provide the multiplexer control signals and register write signals for the data path. Together these three boards make up the processor. The processor together with a system board, which has memory and input/output ports, make up a complete computer system.

The processor instruction set consists of 30 instructions, explained in detail in this manual in the Instruction Set and Programming sections. The processor has two addressing modes, direct (operand in the instruction) and memory (operand in a memory location referenced by the instruction). It can access directly a 64K memory space. The processor can run at a clock speed of up to 2 MHz, and can perform 400,000 additions per second.

The processor was designed with simplicity as a main goal. The entire design of the processor is open for study, and the schematics are complete. It is my hope that anyone studying this processor will be able to understand how it works, and by extension, how more complex processors work. Admittedly the processor lacks many of the features of modern processors, but those features add complexity, and the functionality of most of those features can be implemented in software. For example, this processor does not have registers for address indexing, but indexing can be done by placing an instruction in RAM and indexing the instruction’s address operand there (see the Special Programming Techniques section in the Appendix).

I hope you enjoy making and using this processor kit. I have enjoyed designing it.

--Donn Stewart, January 2019

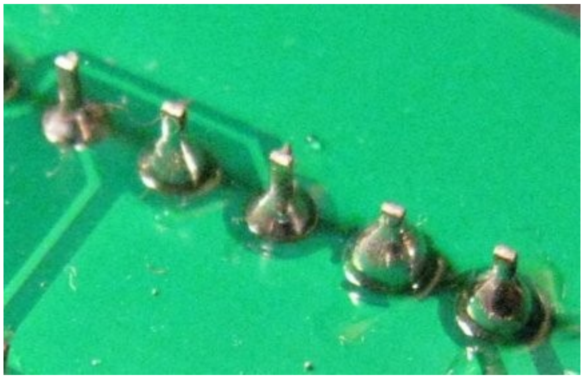
Building Tips[1]

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

1. Soldering iron. I strongly recommend a pencil-tip type of iron, from 15 to 30 watts.
2. Solder. Use rosin core solder. Lead-free or lead-containing solders 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 try to 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:

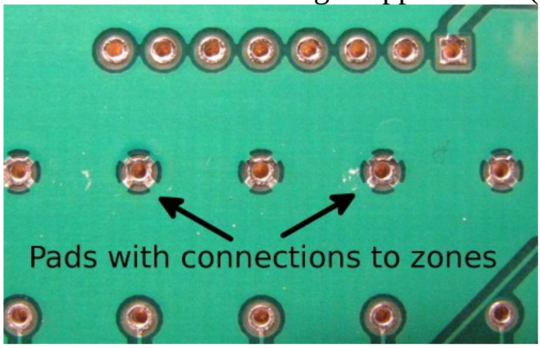
1. Before you plug in the iron, clean the tip with something mildly abrasive, like steel wool or a 3M Scotchbrite pad (plain ones, not the ones with soap in them).
2. Let the iron get hot, then tin the tip with lots of solder (let it drip off some). With a fresh coat of shiny solder the heat transfer 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:



[1] These are generic building tips that apply to all the CPUville kits. The photos here may or may not be of the kit(s) you have purchased.

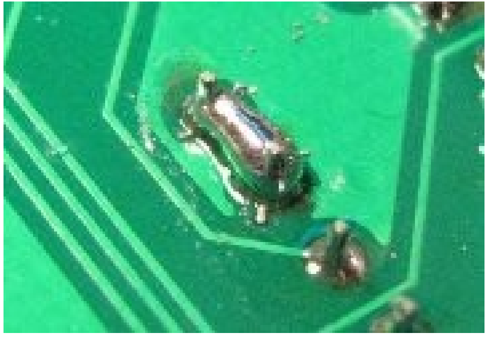
This is only about 1/8th inch of the 0.032 inch diameter solder that I use. If you keep applying the solder, it will drip down the lead to the other side of the board, and you can get shorts. Plus, it looks bad.

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



These require extra heat to make good connections, because the zones wick away the soldering iron heat. You will usually need to let a 15-watt iron rest on the pin and pad for more time before applying the solder (count to 10). You also can use a more powerful (30 watt) soldering iron.

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



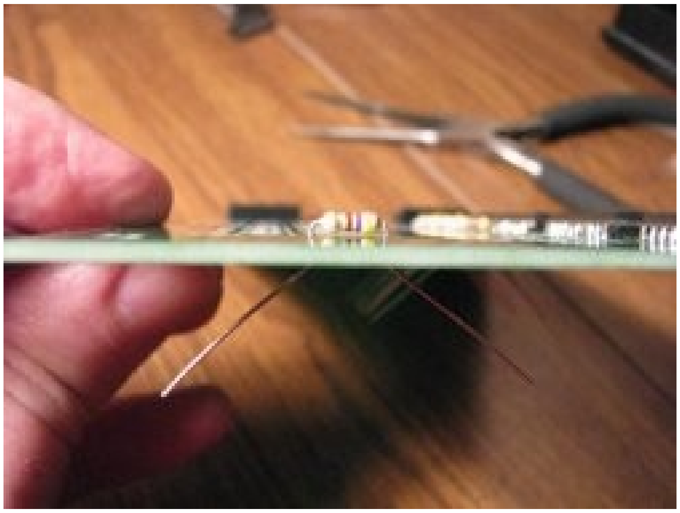
If this happens, you have to remove the solder with a desoldering tool, and re-do the joints.

Other tips:

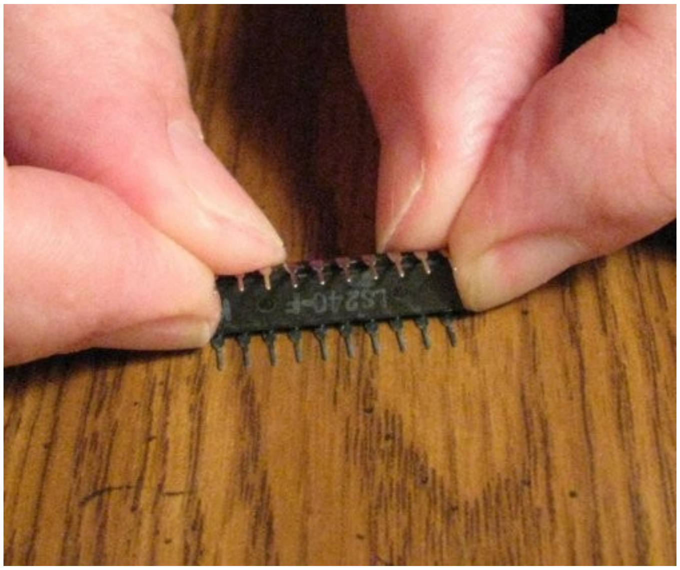
1. Be careful not to damage the traces on the board. They are very thin copper films, just under a thin plastic layer of solder mask (the green stuff). If you plop the board down on a hard

surface that has hard debris on it (like ICs, screws etc.) it is easy to cut a trace. Such damage can be fixed, if you can find it, but try to avoid it in the first place.

2. When soldering multi-pin components, like the ICs or IC sockets, 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 work 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 resistors, 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, sockets, 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 or the handle of a tool. Another technique is to put a tiny drop of solder on the tip of the iron, press the part 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:

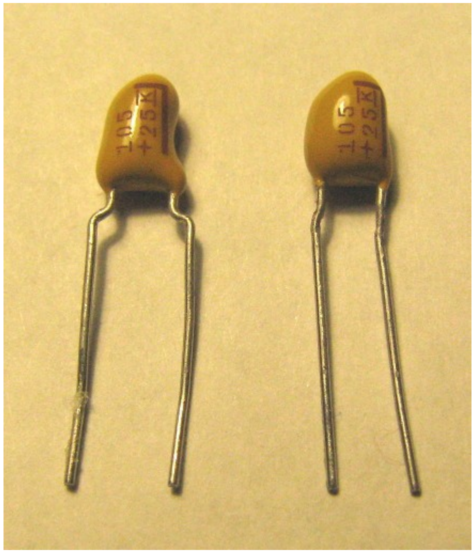


4. You might have to bend the leads on components, ICs or IC sockets to get them to fit into the holes on the boards. For an IC, place the part on the table and bend the leads all at once, like this:



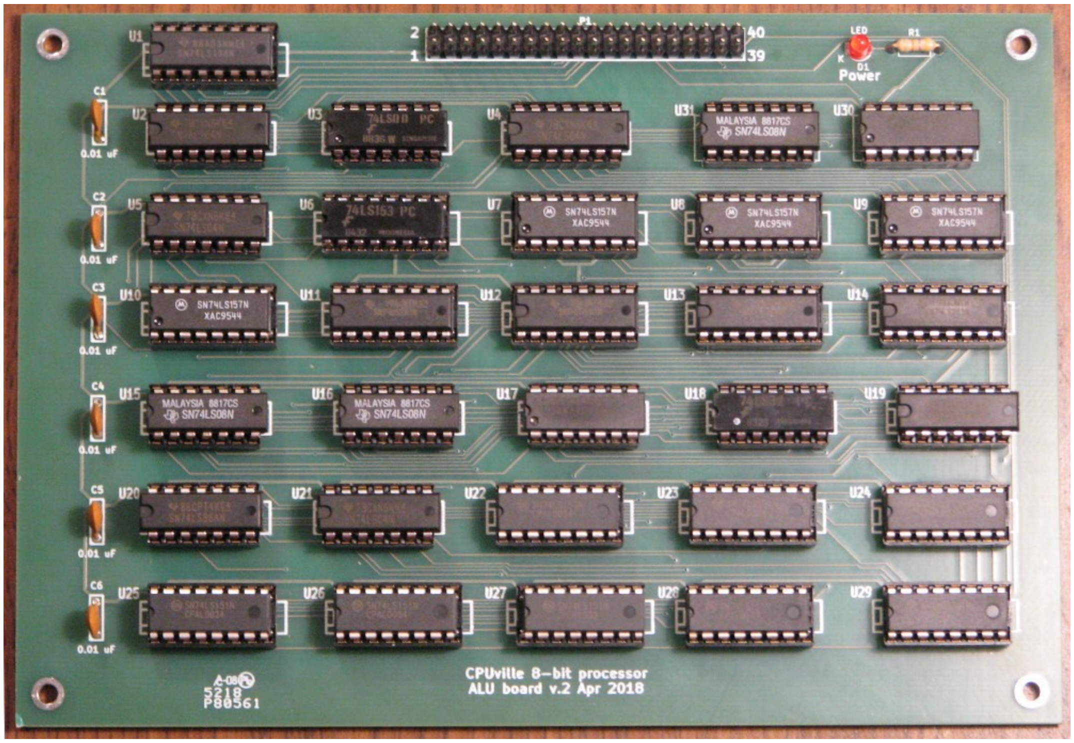
Bending the leads one-by-one or all together with the needle nose pliers doesn't work as well for some reason.

Also, some components have leads bent outward to fit in a certain printed circuit board footprint, but will fit a smaller footprint if you bend the leads in with a needle-nosed pliers. Here is a tantalum capacitor, one with wide leads, the other with narrow leads, from bending the wide leads in:

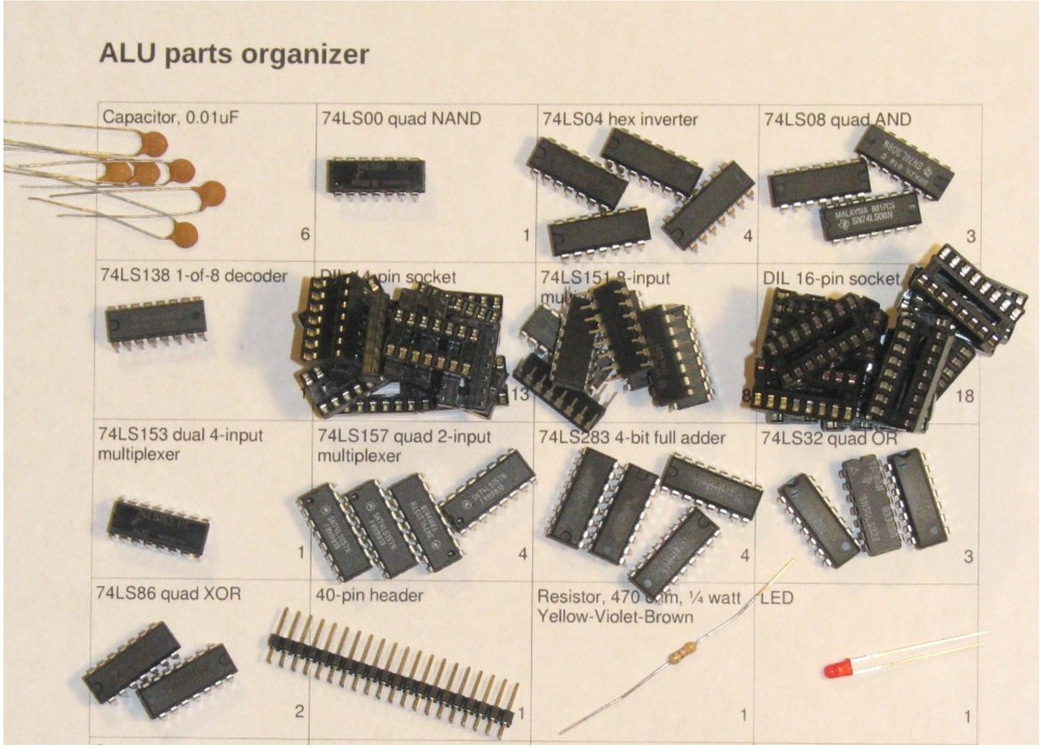


5. After you have soldered a row or two check the joints with a magnifying glass. These kits have small leads and pads, and it can be hard to see if you got the solder on correctly by naked eye. You can miss tiny hair-like solder bridges unless you inspect carefully. It is good to brush off the bottom of the board from time to time with something like a dry paintbrush or toothbrush, to get off any small solder drops that are sitting there. After you are finished, wiping with an alcohol-soaked rag will get off rosin splatter.
6. The connectors, like the 40-pin IDE drive connector and the system connector some kits have pins that are a little more massive than the IC socket or component pins. This means that more time, or perhaps more wattage, will be required to heat these pins with the soldering iron, to ensure good electrical connections.

Building the ALU



Print out the ALU Parts Organizer (in the Appendix) and put the parts on the organizer to make sure you have them all, and to get familiar with them:



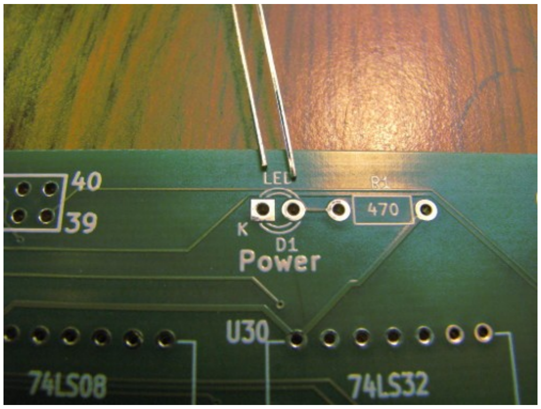
ALU parts organizer

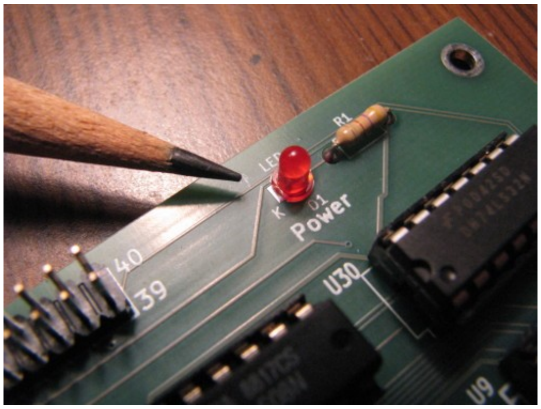
Capacitor, 0.01uF 6	74LS00 quad NAND 1	74LS04 hex inverter 4	74LS08 quad AND 3
74LS138 1-of-8 decoder 1	DIL 14-pin socket 13	74LS151 8-input multi 8	DIL 16-pin socket 18
74LS153 dual 4-input multiplexer 1	74LS157 quad 2-input multiplexer 4	74LS283 4-bit full adder 4	74LS32 quad OR 3
74LS86 quad XOR 2	40-pin header 1	Resistor, 470 ohm, ¼ watt Yellow-Violet-Brown 1	LED 1

Once you have checked the parts you can start to solder them onto the circuit board.

The easiest way to solder the components is to start with the shortest (parts that lie closest to the board) and proceed to the tallest. The order is resistors, sockets, LEDs, capacitors, and the 40-pin connectors. Some components need to be oriented properly, as described below.

1. The resistors can be soldered first. They do not have to be oriented.
2. The IC sockets are next. They do not need to be oriented.
3. The LED is next. The cathode, which is side with the shorter lead, and the flat side of the plastic base, is oriented toward the right. There is a small “K” on the circuit board symbol by the cathode hole:



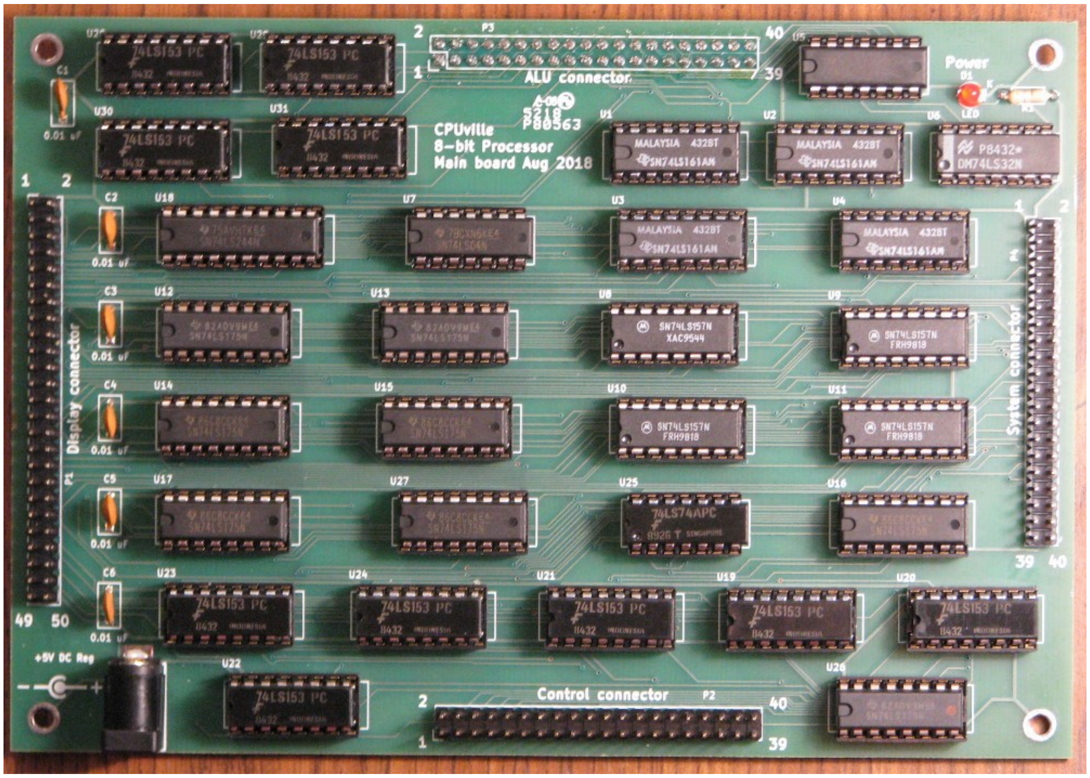


4. The bypass capacitors are next. They do not need to be oriented.
5. The 40-pin ALU connector is next. No orientation is necessary, but it has fairly large leads and may require more time or soldering iron wattage to solder.
6. Once you have finished soldering all the parts on the board, inspect the board to make sure there are no solder bridges or unsoldered pins. Lightly brush the back of the board with an old toothbrush or paintbrush to clear off loose debris or tiny solder hairs. You can wipe the back of the board with a cloth soaked in alcohol to remove small drops of rosin flux that have spattered about.

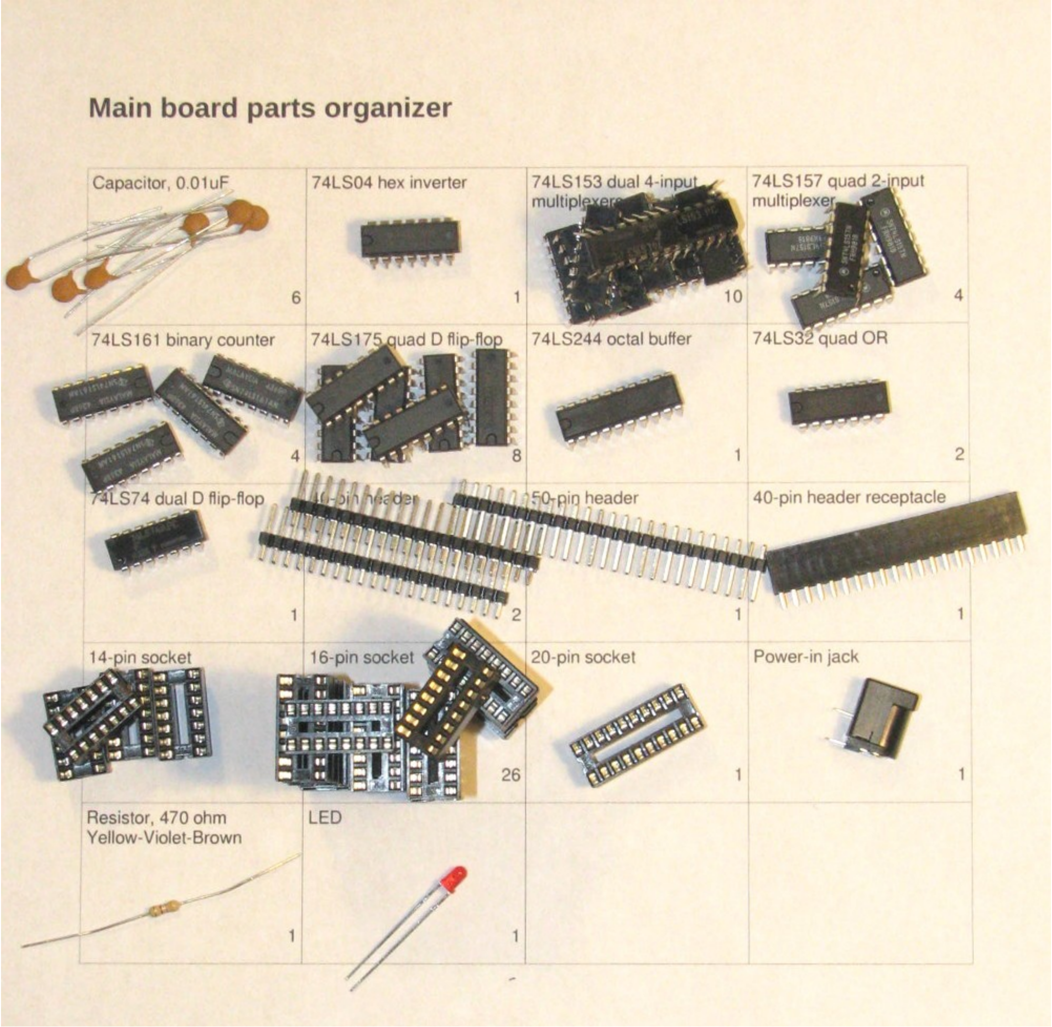
   Hold the finished board against a bright light. If you can see light coming through a pin hole, go back and solder it again, to make sure you have a good electrical connection. This does not apply to the vias, the plated holes where a trace goes from one side of the board to the other. These can be left open.

See the section “Assembling the 8-bit Processor” for instructions on inserting the ICs.

Building the Main Board



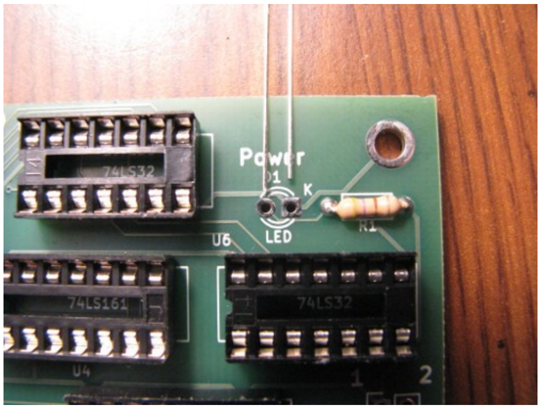
Print out the Main Board Parts Organizer (in the Appendix) and put the parts on the organizer to make sure you have them all, and to get familiar with them:

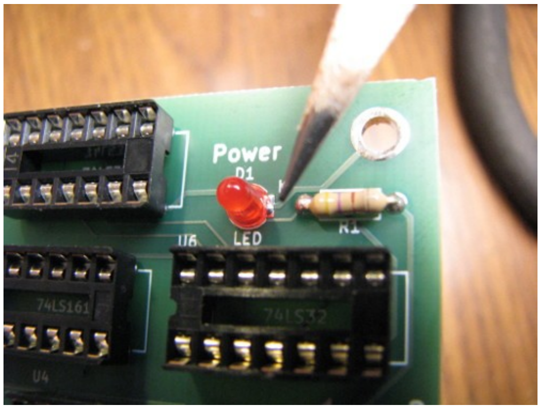


Once you have checked the parts you can start to solder them onto the circuit board.

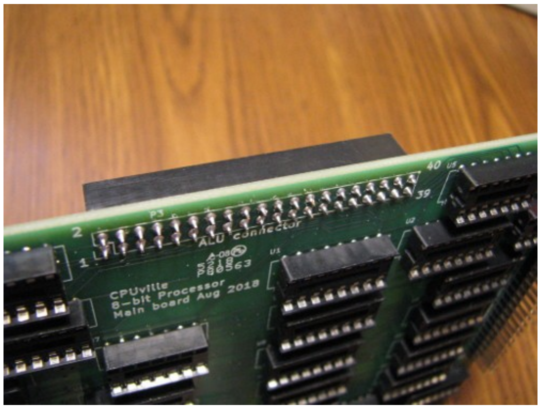
The easiest way to solder the components is to start with the shortest (parts that lie closest to the board) and proceed to the tallest. The order is resistors, sockets, LED, capacitors, and the 40-pin and 50-pin connectors. One 40-pin header receptacle is soldered on the back of the board. Some components need to be oriented properly, as described below.

1. The resistors can be soldered first. They do not have to be oriented.
2. The IC sockets are next. They do not need to be oriented.
3. The LED is next. The cathode, which is side with the shorter lead, and the flat side of the plastic base, is oriented toward the right. There is a small “K” on the circuit board symbol by the cathode hole:





4. The bypass capacitors are next. They do not need to be oriented.
5. The 40-pin control and system connectors, and the 50-pin register display connectors are next. No orientation is necessary, but these connectors have fairly large leads and may require more time and/or soldering iron wattage to solder.
6. The 40-pin ALU header receptacle is soldered to the back of the board. Apply solder to the pads and pins on the top of the board:



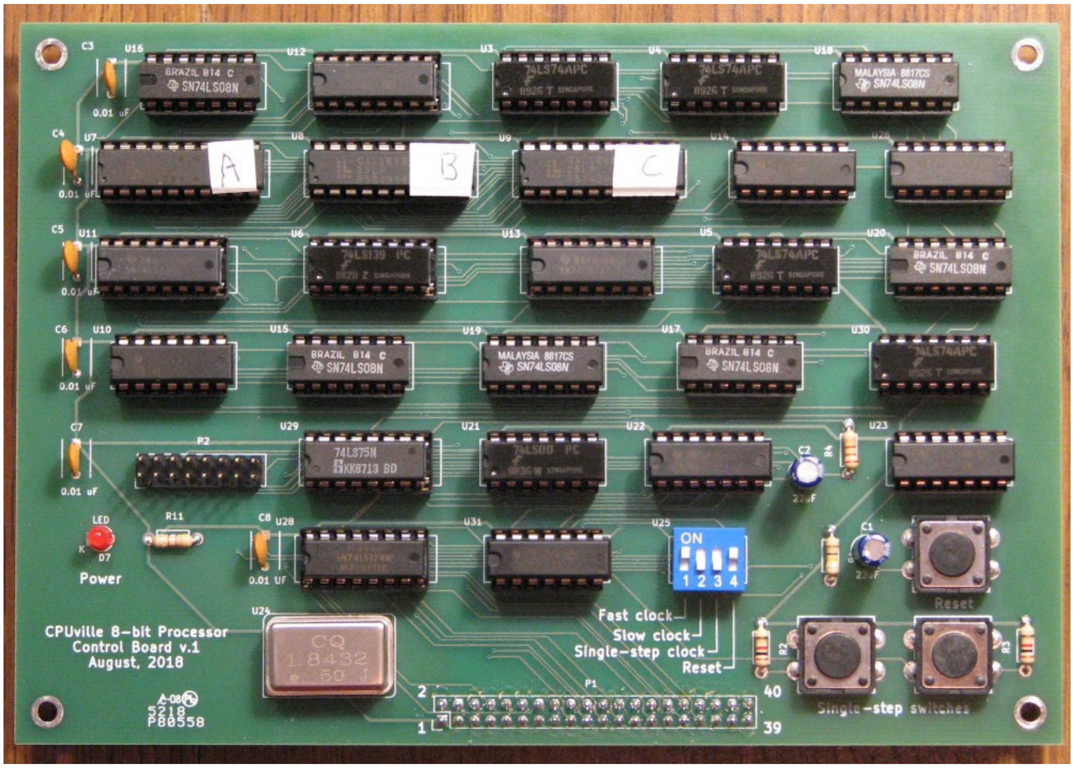
7. Once you have finished soldering all the parts on the board, inspect the board to make sure there are no solder bridges or unsoldered pins. Lightly brush the back of the board with an old

toothbrush or paintbrush to clear off loose debris or tiny solder hairs. You can wipe the back of the board with a cloth soaked in alcohol to remove small drops of rosin flux that have spattered about.

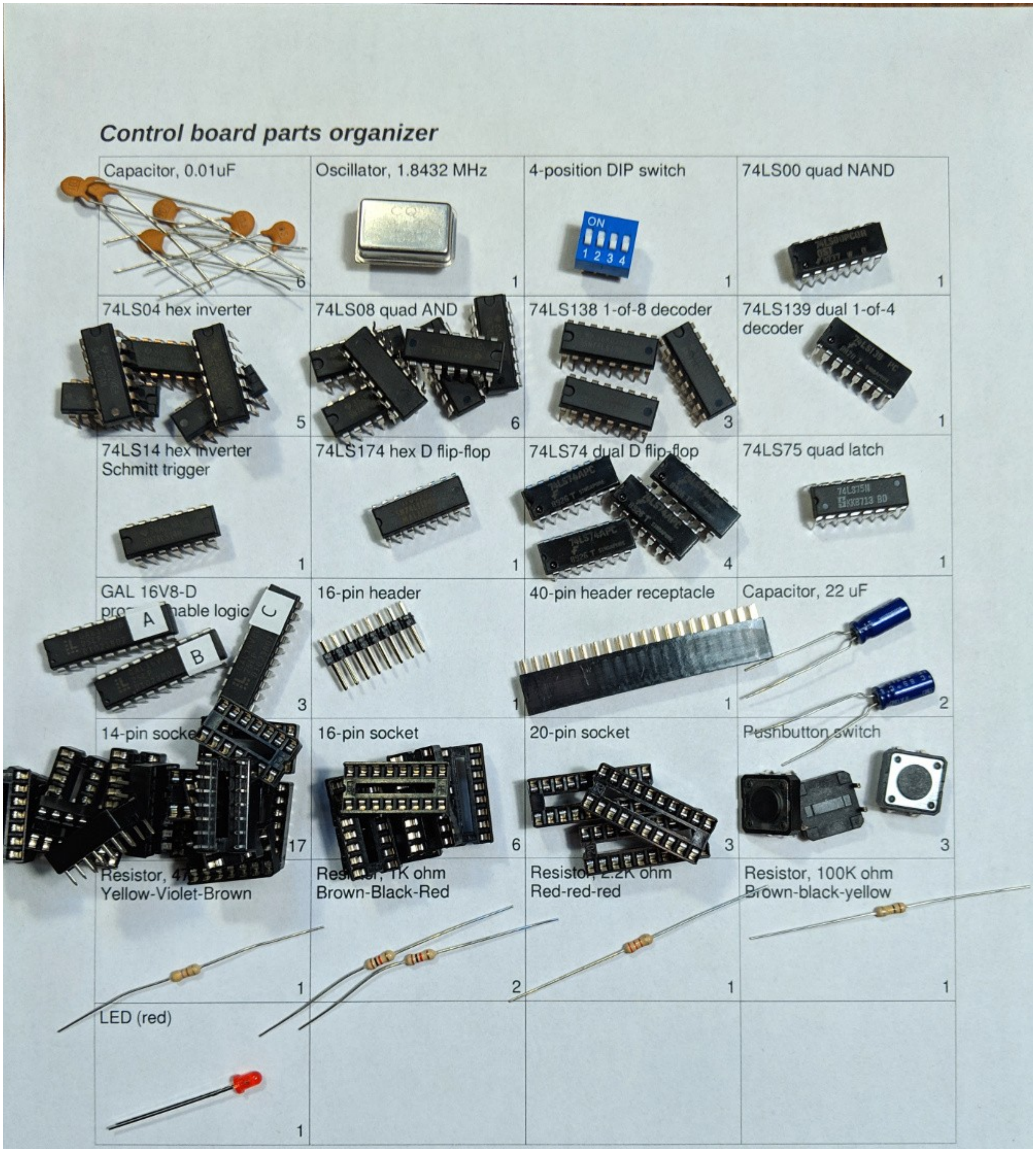
Hold the finished board against a bright light. If you can see light coming through a pin hole, go back and solder it again, to make sure you have a good electrical connection. This does not apply to the vias, the plated holes where a trace goes from one side of the board to the other. These can be left open.

See the section “Assembling the 8-bit Processor” for instructions on inserting the ICs.

Building the Control Board



Print out the Control Board Parts Organizer (in the Appendix) and put the parts on the organizer to make sure you have them all, and to get familiar with them:



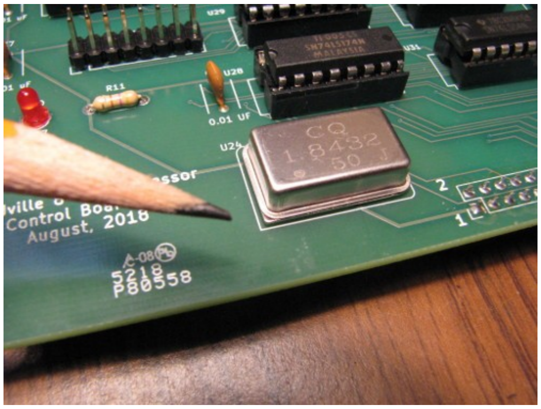
Control board parts organizer

Capacitor, 0.01uF 6	Oscillator, 1.8432 MHz 1	4-position DIP switch 1	74LS00 quad NAND 1
74LS04 hex inverter 5	74LS08 quad AND 6	74LS138 1-of-8 decoder 3	74LS139 dual 1-of-4 decoder 1
74LS14 hex inverter Schmitt trigger 1	74LS174 hex D flip-flop 1	74LS74 dual D flip-flop 4	74LS75 quad latch 1
GAL 16V8-D programmable logic 3	16-pin header 1	40-pin header receptacle 1	Capacitor, 22 uF 2
14-pin socket 17	16-pin socket 6	20-pin socket 3	Pushbutton switch 3
Resistor, 47[illegible] Yellow-Violet-Brown 1	Resistor, 1K ohm Brown-Black-Red 2	Resistor, 2.2K ohm Red-red-red 1	Resistor, 100K ohm Brown-black-yellow 1
LED (red) 1			

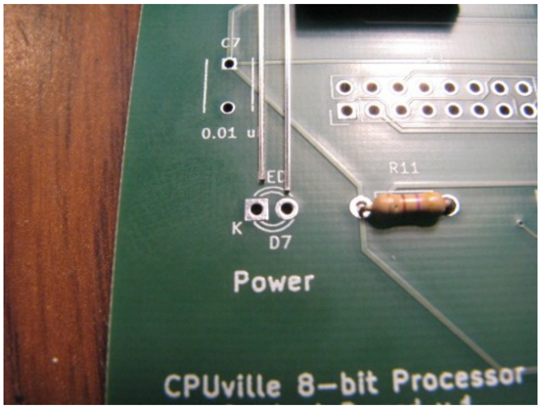
Once you have checked the parts you can start to solder them onto the circuit board.

The easiest way to solder the components is to start with the shortest (parts that lie closest to the board) and proceed to the tallest. The order is resistors, pushbutton switches, oscillator, sockets, LED, ceramic bypass capacitors, DIP switches, electrolytic capacitors. The 40-pin header receptacle is soldered on the back of the board. Some components need to be oriented properly, as described below.

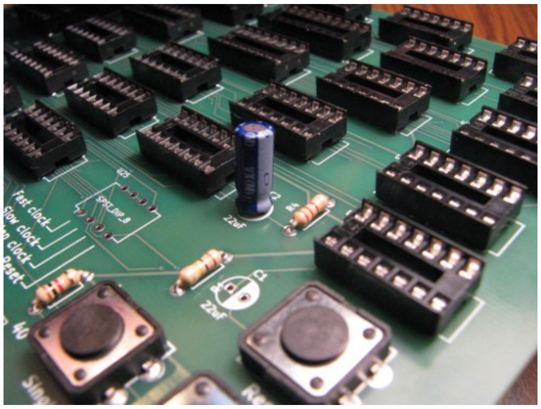
1. The resistors can be soldered first. They do not have to be oriented.
2. The pushbuttons are next. Make sure the leads are pushed all the way in. They kind of snap in their holes.
3. The oscillator is oriented with the sharp corner at the front left:



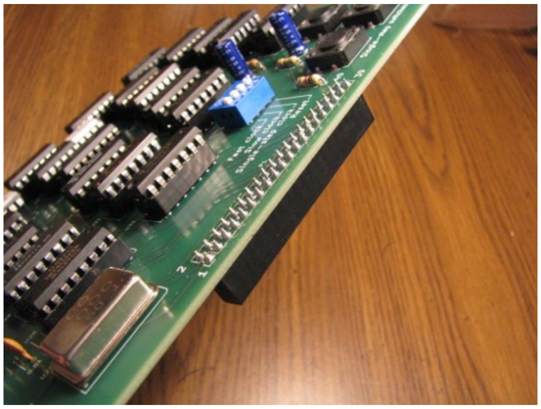
4. The IC sockets are next. They do not need to be oriented.
5. The LED is next. The cathode, which is side with the shorter lead, and the flat side of the plastic base, is oriented toward the right. There is a small “K” on the circuit board symbol by the cathode hole:



6. The bypass capacitors are next. They do not need to be oriented.
7. The DIP switches are placed so that ON is up.
8. The electrolytic capacitors are placed with the negative stripe toward the right:



9. The 40-pin control connector header receptacle is soldered to the back of the board. Apply solder to the pads and pins on the top of the board



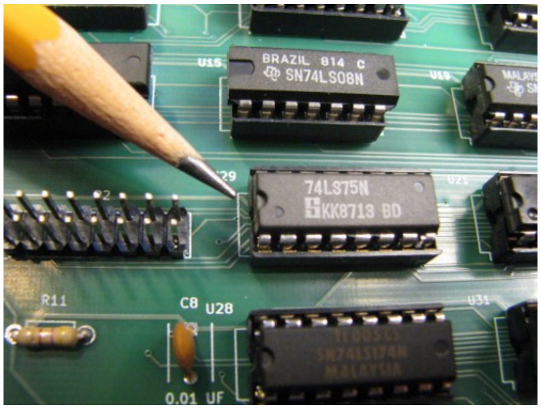
10. Once you have finished soldering all the parts on the board, inspect the board to make sure there are no solder bridges or unsoldered pins. Lightly brush the back of the board with an old toothbrush or paintbrush to clear off loose debris or tiny solder hairs. You can wipe the back of the board with a cloth soaked in alcohol to remove small drops of rosin flux that have spattered about.

    Hold the finished board against a bright light. If you can see light coming through a pin hole, go back and solder it again, to make sure you have a good electrical connection. This does not apply to the vias, the plated holes where a trace goes from one side of the board to the other. These can be left open.

    See the section “Assembling the 8-bit Processor” for instructions on inserting the ICs.

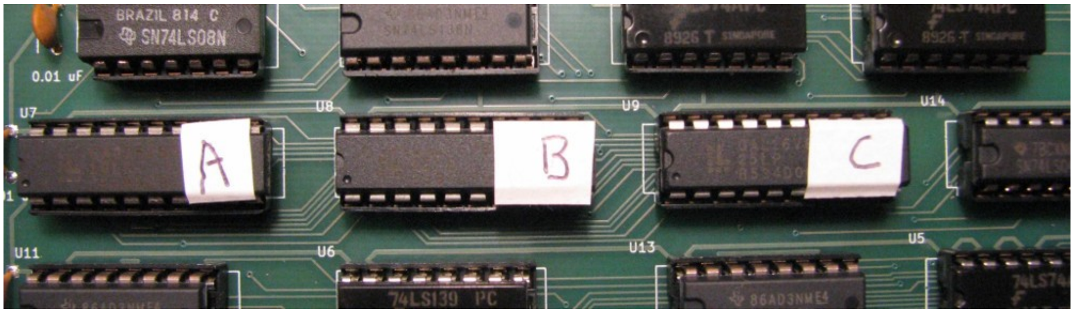
Assembling the 8-bit Processor

Carefully insert the ICs in their sockets on each board. Double check the IC labels to be sure you are putting the correct ones in the correct locations. They are oriented with the small cut-out toward the left:

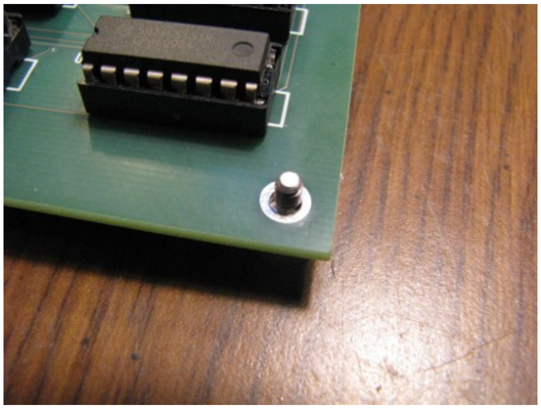


You may have to bend the pins a little to make them go straight down, to better align with the pin holes in the sockets. Make sure you do not fold any pins under when inserting the ICs. This is easy to do if you are not careful, and can create a frustrating hardware bug that can be difficult to find. A folded-under pin can look exactly like a normally inserted pin from the top.

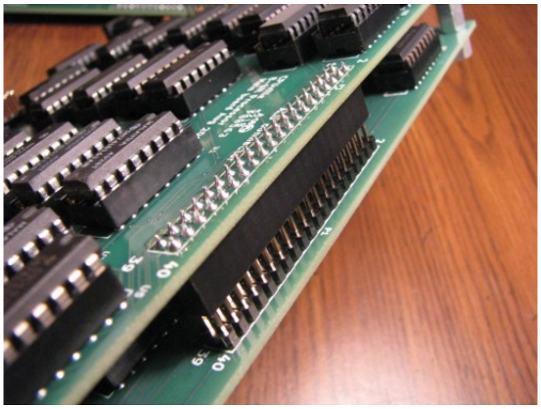
Insert the three labeled GAL16V8 ICs in the control board A, B, C left to right:



After inserting the ICs the three-board stack can be assembled. Place the quarter-inch long M/F standoffs into the mounting holes under the ALU board with the threads up:



Then place the half-inch standoffs on those. Place the main board on the stack next. Be careful to line up the pins on the ALU header with the receptacle on the bottom of the main board:

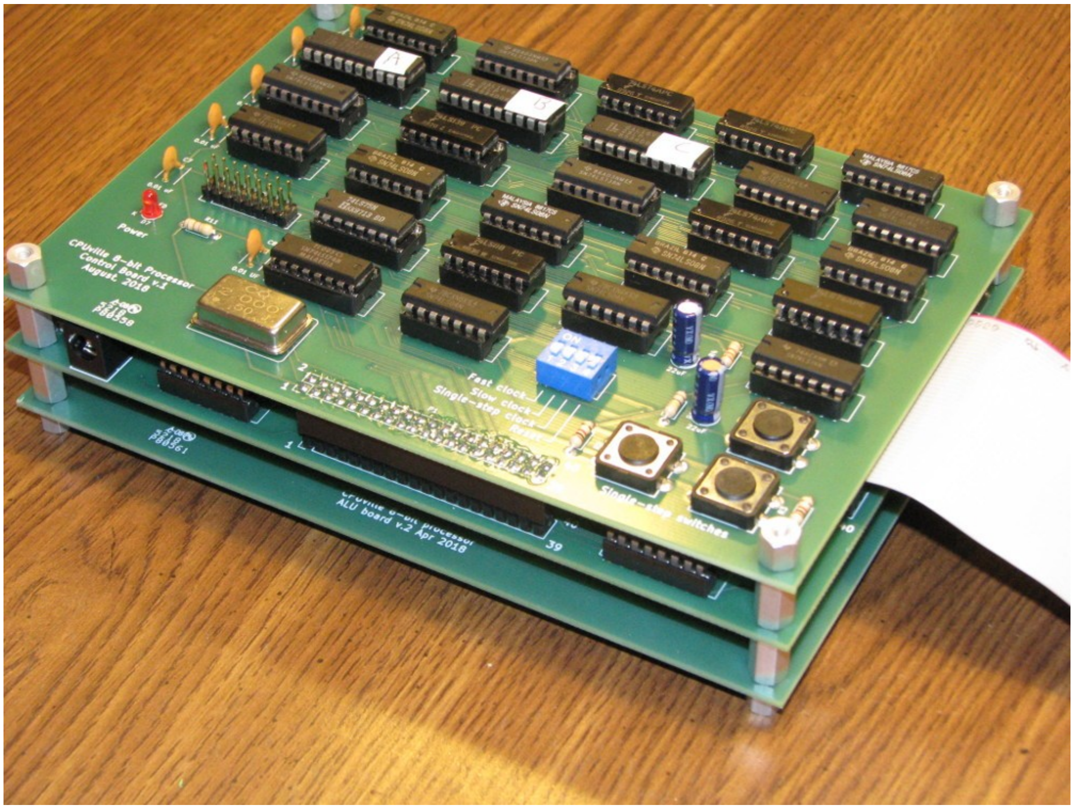


If lined up properly the main board the mounting holes on the main board will fit exactly over the threads of the stand-offs.

Plug the cable or adapter for your system board onto the main board system connector (the 40-pin header on the right).

Do not connect your system board yet.

Place half-inch stand-offs onto the threads of the standoffs beneath the main board. Then place the control board on the stack, with its 40-pin receptacle lined up with the control connector on the main board. Finish the stack by putting the quarter-inch nuts (female 0.25 inch standoffs) on the threads of the top standoffs:



The header receptacles may not sit all the way down on the headers. This is OK, there will still be a good electrical connection even with 1/8th inch of the header pins showing.

Once the stack is complete, connect power to the input jack on the main board. The processor uses +5V DC, regulated, with a minimum of 2 amps (a 10 watt supply). This is also enough to run the processor with a system board attached. Check to see if the power LEDs on each board are lit. If not, you may not have good connections between the boards. Recheck to be sure the headers and plugs are lined up correctly. If the power LEDs all light up, remove power, and connect the processor to your system board. You can connect power to either the jack on the front of the processor main board, or the jack on your system board, whichever is more convenient.

Control Board Switches

For convenience, the processor control board has several oscillators, a reset circuit, and switches to allow easy operation of the computer system when the boards are stacked with a system board on the bottom.

The control board has three clock oscillators that can be selected. The fast clock is a quartz crystal oscillator that runs at 1.8432 MHz. The slow clock is an R-C oscillator that runs at a few cycles per second. The single-step clock will produce an upgoing clock edge when the right-hand single-step pushbutton is pressed, and a downgoing edge when the left button is pressed. Select the desired oscillator by turning on its corresponding switch. Only one of the three clock selection switches should be on at any time. If you have more than one on, no harm will be done, but the clock signal will not be reliable.

The reset switch on the control board connects the reset circuit on the control board to the system reset line. This circuit has an R-C delay that holds the system in reset for about a second after power-up to allow the system to start properly. After about one second the reset is automatically released and the system begins to run. The reset button, when held down, will hold the system in reset. When the reset button is released, after about a second, the system will begin to run again.

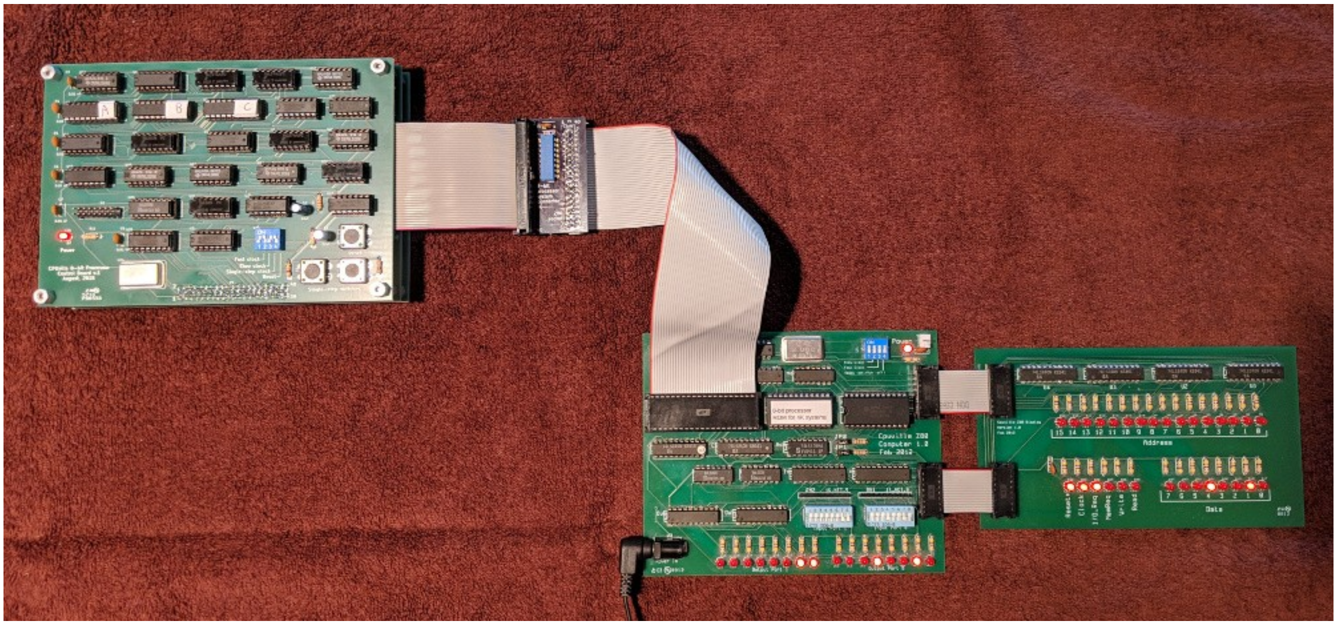
If your system board has an oscillator or oscillators on it, and you want to use it or them, leave the control board clock selection switches off, and select a clock on your system board. If you want to use the oscillators on the control board, leave the oscillator switches or clock jumper on your system board off.

Similarly, if you want to use the reset switch or button on your system board, leave the reset switch on the control board off. If you want to use the control board reset circuit, turn the reset switch on your system board off, or remove the reset jumper, and turn the reset switch on the control board on.

Using the 8-bit Processor with a 4K System

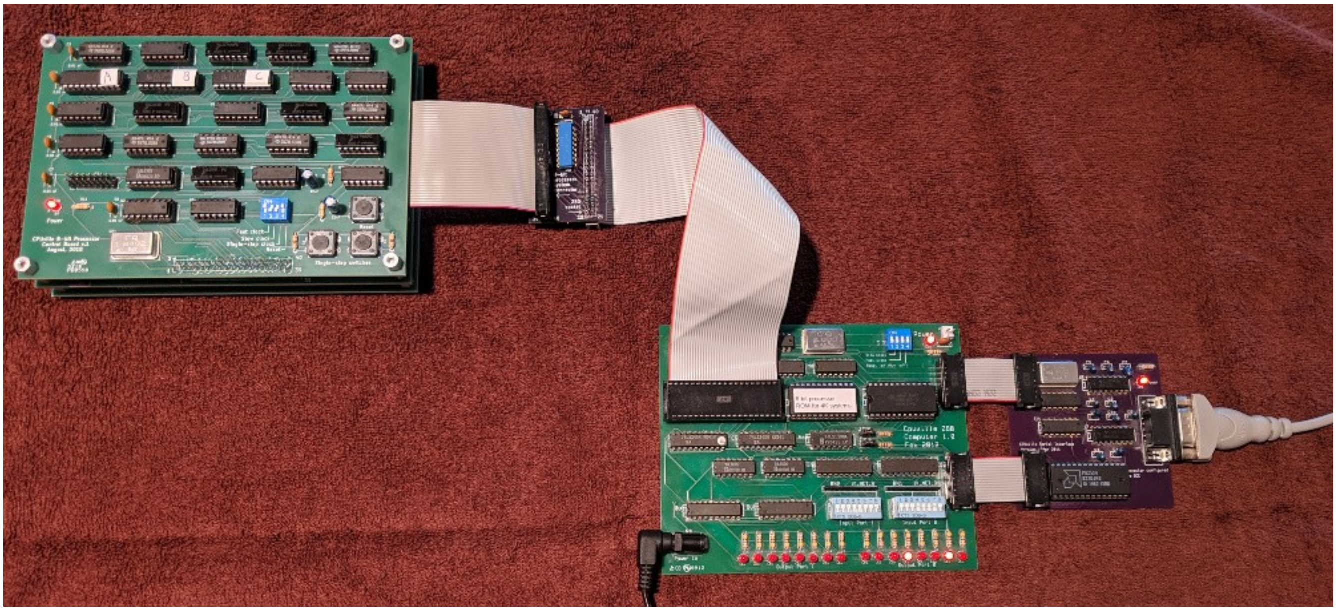
If using an Original Z80 kit with 2K ROM and 2K RAM (4K total memory space) as the system board for the processor, remove the Z80 and the v.7 ROM from their sockets. Switch off both clock switches and the reset switch on the Z80 computer kit board to use the clock oscillators and reset circuit on the processor control board, or turn off the clock switches and reset switch on the control board to use the oscillators and reset switch on the Z80 computer kit board. Place the 8-bit processor ROM for 4K systems in the ROM socket.

The processor is connected to the Z80 computer kit board with a special adapter circuit board and cables. A short 40-conductor cable connects the adapter circuit board to the system connector on the processor main board, and a special cable connects the adapter circuit board to the Z80 socket. The 40-pin DIP plug on the end of the special cable plugs into the Z80 socket. Here is a photo of a 4K system with the bus display attached:



The edge of the ribbon cable for Z80 pins 1 and 40 should be on the left of the Z80 socket, as shown above.

The 4K system can also be used with the serial interface board:



The ROM for 4K systems has code for a few simple test programs that make use of the input DIP switches and output LEDs, and for testing and operating the serial interface[2]. There is also a program loader. The program loader will take hexadecimal character input from the keyboard, convert the characters to binary data (bytes), store the bytes in RAM starting at location 0x0810, and jump to the start of the program when you hit the Enter key. A listing of the contents of this ROM, and a short adder program to test the program loader, can be found in the Program Listings section of this manual. Here are the programs in the ROM and the addresses:

0x0012	Port reflector
0x001D	Simple counter
0x0025	Two-byte counter
0x0043	One-byte highest factor routine
0x0069	Serial interface test (echos characters)
0x008E	Program loader

To use these programs, determine from the ROM listing, or from the above list, the address of the program you want to run, and place the address on the DIP switches of the computer board. Then power up the system. When the system comes out of reset it will then jump to the address on the input switches and run from there. To reset the computer while it is powered up, you can press the reset button on the control board, or turn on the reset switch on the system board if using this.

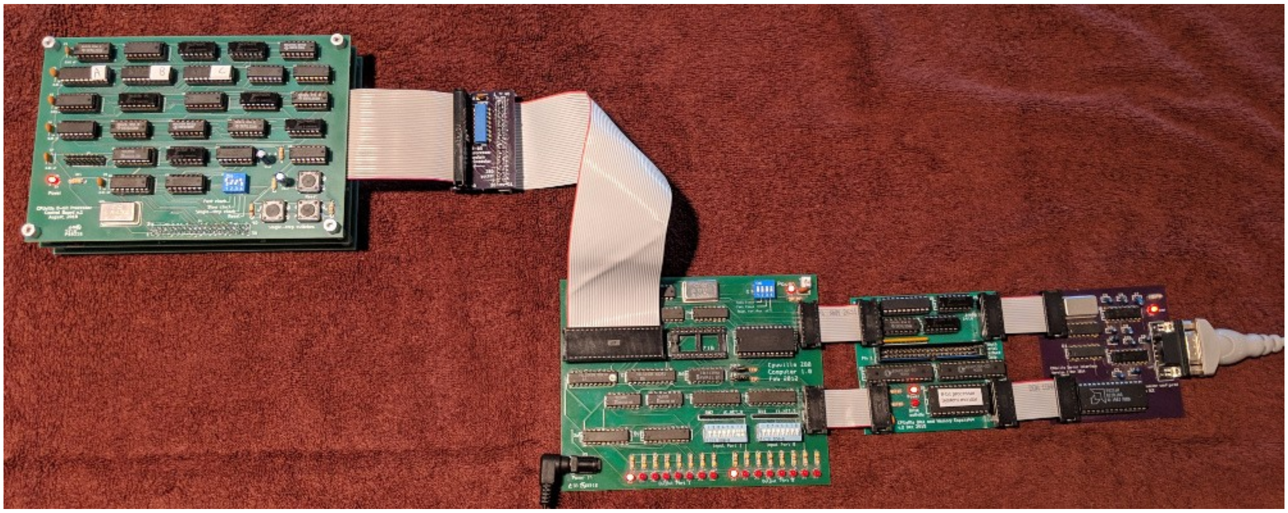
The 4K system can also use the 8-bit processor System Monitor, which is described in the next section. However, you will only have access to RAM addresses 0x0900 to 0x0FFF. The System Monitor reserves RAM addresses 0x0800 to 0x08FF for its variables and buffers.

Using the 8-bit Processor with a 64K System

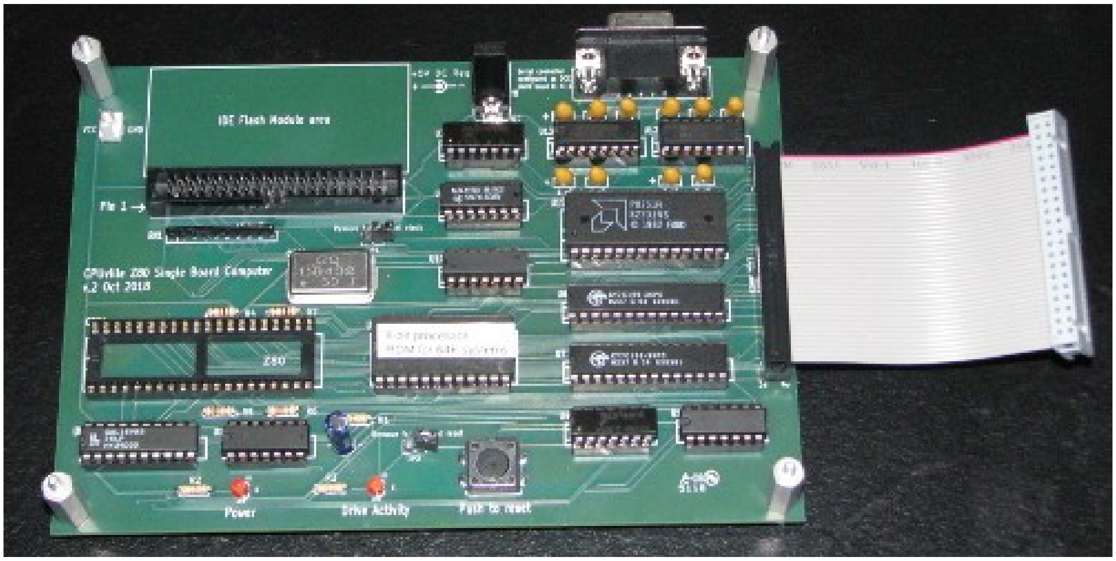
There are two types of 64K systems you can use with the 8-bit processor. These are the Original Z80 kit with the disk and memory expansion, and the Single-board Z80 kit.

To use the Original Z80 kit with the disk and memory expansion as a 64K system for the processor, connect the processor to the computer board with the disk and memory expansion and serial interface attached, through the adapter and Z80 socket as shown for the 4K system above. The ROM and ports on the computer board need to be disabled by removing the jumpers (you do not need to remove the v.7 ROM, but you can if you want). Remove the v.8 ROM from the socket on the disk and memory expansion board, and replace it with the 8-bit processor System Monitor ROM. Switch off both clock switches and the reset switch on the computer board to use the clock and reset circuits on the processor control board, or switch off the clock and reset switches on the processor control board to use the corresponding circuits on the system board, as described for the 4K system above. Here is a photo the Original Z80 computer kit configured as a 64K system for the 8-bit processor:

[2] For details on configuring and using the serial interface with a terminal emulation program see the Serial Interface Kit instruction manual.

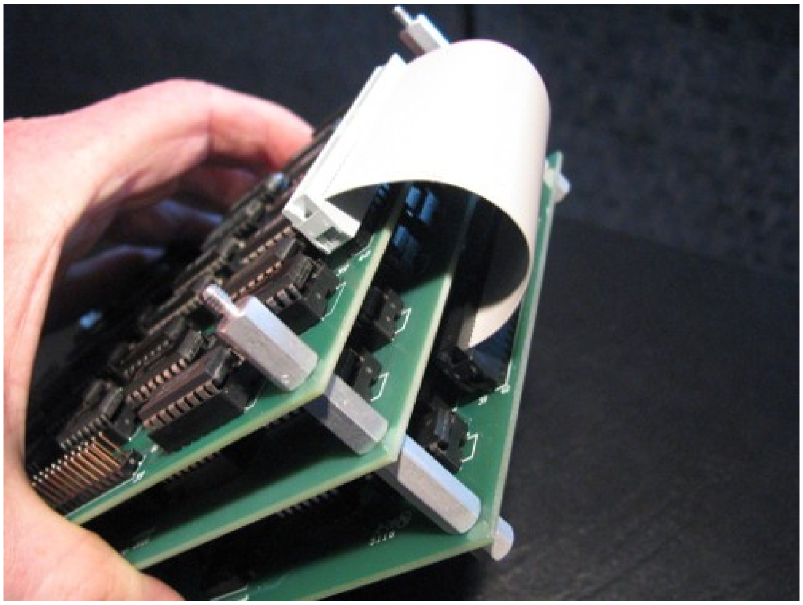


If using a Single-board Z80 kit computer as your system board, remove the clock and reset jumpers on the computer board (the clock and reset will be provided by the processor control board). Remove the Z80 processor and v.8 ROM from their sockets. Place the 8-bit processor System Monitor ROM in the ROM socket. Place the ¼ inch male/female standoffs beneath the board, with the threads coming through the mounting holes, and place the ¾ inch standoffs on top of the board:



The longer standoffs are provided with the Single-board Z80 kit in case you want to attach a disk module to the IDE socket on the computer board.

The Single-board computer can be then be placed on the bottom of the processor stack and connected to the main board of the processor by a 40-conductor ribbon cable:



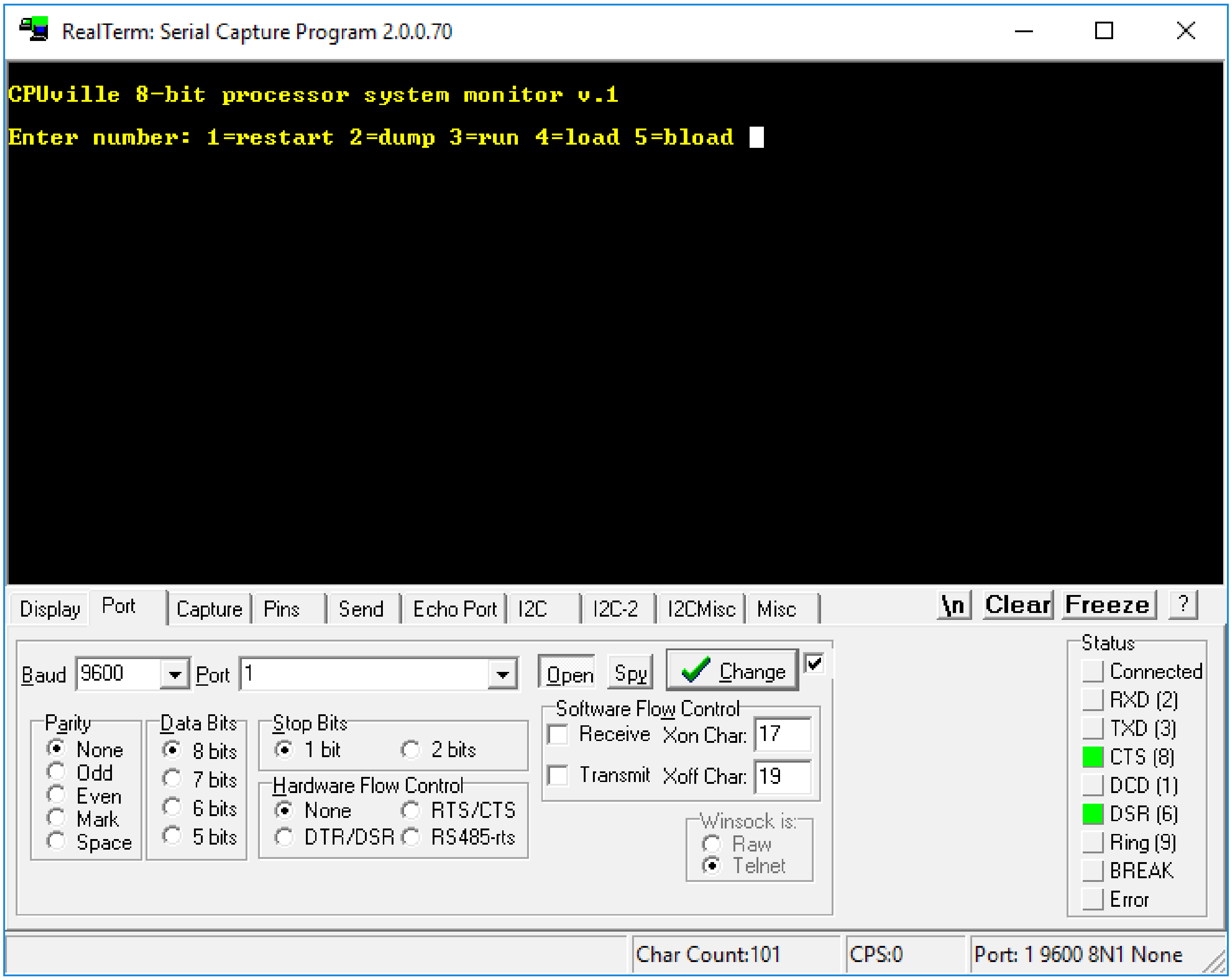
Carefully place the control board on the top of the stack, lining up the receptacle on the bottom of the control board with the main board control header, to complete assembly of the 64K 8-bit processor computer system.

The system monitor program in the ROM is intended for use with the computer connected through the serial interface to a dumb terminal, or to a PC running a terminal emulation program[3]. This allows text output to a display, and text input using a keyboard. Connect the assembled 64K system to the PC's serial port using a straight-through serial cable, or to a USB port using an RS-232-to-USB adapter. Start a terminal emulation program, configure the port for 8-N-1, 9600 baud communication. Connect +5V DC regulated, 2 amp minimum power to the computer through either the jack on the front of the main board, or the jack on the back of the system board.

The following examples use the RealTerm terminal emulation program running under Windows.

At power-up the system will display the system monitor greeting message and a list of commands:

[3] For details on configuring and using the serial interface of the single-board computer see the Single-board Z80 Computer Kit instruction manual.



RealTerm: Serial Capture Program 2.0.0.70

```
CPUville 8-bit processor system monitor v.1
Enter number: 1=restart 2=dump 3=run 4=load 5=bload
```

Baud 9600 Port 1

Parity: ● None ○ Odd ○ Even ○ Mark ○ Space

Data Bits: ● 8 bits ○ 7 bits ○ 6 bits ○ 5 bits

Stop Bits: ● 1 bit ○ 2 bits

Hardware Flow Control: ● None ○ RTS/CTS ○ DTR/DSR ○ RS485-rts

Software Flow Control: □ Receive Xon Char: 17 □ Transmit Xoff Char: 19

Winsock is: ○ Raw ● Telnet

Status: □ Connected □ RXD (2) □ TXD (3) 🟩 CTS (8) □ DCD (1) 🟩 DSR (6) □ Ring (9) □ BREAK □ Error

Char Count:101 CPS:0 Port: 1 9600 8N1 None

The commands are entered by pressing the number keys. After entering the number of the monitor command, further input is taken from the keyboard. Here is a list of the commands.

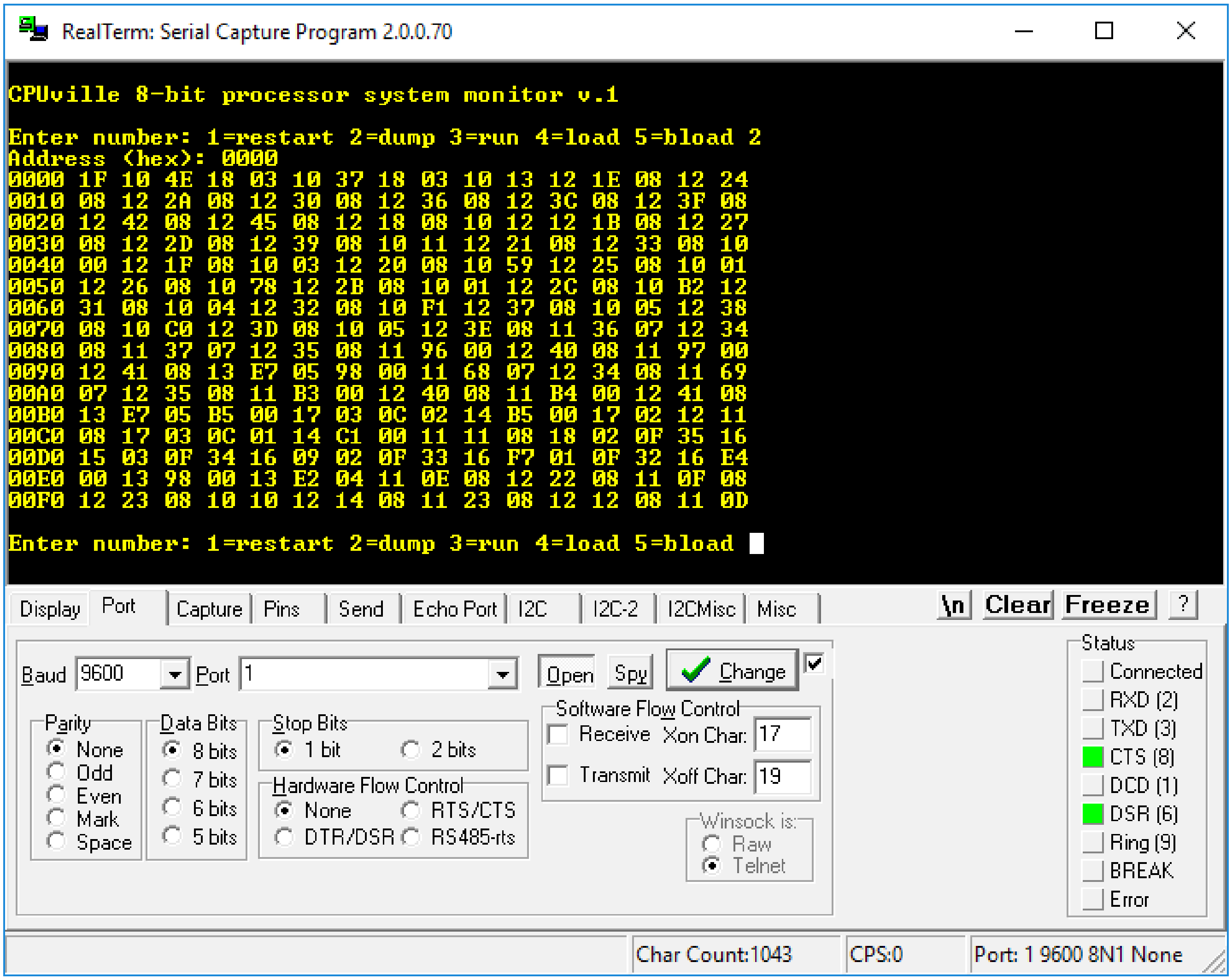
Monitor commands

1=restart

This simply restarts the monitor program. You should get the command list again, without the greeting message. This just verifies that the computer is alive and well.

2=dump

Displays a 256-byte block of the computer's memory. The command takes a 4-character hexadecimal address as input, with characters A through F as upper case. The output display shows the 4-character hexadecimal address of the first byte of each row, then 16 bytes of data as hexadecimal characters. Here is a dump display of the first 256 bytes of the ROM:



RealTerm: Serial Capture Program 2.0.0.70

```
CPUville 8-bit processor system monitor v.1

Enter number: 1=restart 2=dump 3=run 4=load 5=bload 2
Address (hex): 0000
0000 1F 10 4E 18 03 10 37 18 03 10 13 12 1E 08 12 24
0010 08 12 2A 08 12 30 08 12 36 08 12 3C 08 12 3F 08
0020 12 42 08 12 45 08 12 18 08 10 12 12 1B 08 12 27
0030 08 12 2D 08 12 39 08 10 11 12 21 08 12 33 08 10
0040 00 12 1F 08 10 03 12 20 08 10 59 12 25 08 10 01
0050 12 26 08 10 78 12 2B 08 10 01 12 2C 08 10 B2 12
0060 31 08 10 04 12 32 08 10 F1 12 37 08 10 05 12 38
0070 08 10 C0 12 3D 08 10 05 12 3E 08 11 36 07 12 34
0080 08 11 37 07 12 35 08 11 96 00 12 40 08 11 97 00
0090 12 41 08 13 E7 05 98 00 11 68 07 12 34 08 11 69
00A0 07 12 35 08 11 B3 00 12 40 08 11 B4 00 12 41 08
00B0 13 E7 05 B5 00 17 03 0C 02 14 B5 00 17 02 12 11
00C0 08 17 03 0C 01 14 C1 00 11 11 08 18 02 0F 35 16
00D0 15 03 0F 34 16 09 02 0F 33 16 F7 01 0F 32 16 E4
00E0 00 13 98 00 13 E2 04 11 0E 08 12 22 08 11 0F 08
00F0 12 23 08 10 10 12 14 08 11 23 08 12 12 08 11 0D

Enter number: 1=restart 2=dump 3=run 4=load 5=bload
```

Baud 9600 Port 1

Parity: ● None ○ Odd ○ Even ○ Mark ○ Space

Data Bits: ● 8 bits ○ 7 bits ○ 6 bits ○ 5 bits

Stop Bits: ● 1 bit ○ 2 bits

Hardware Flow Control: ● None ○ RTS/CTS ○ DTR/DSR ○ RS485-rts

Software Flow Control: □ Receive Xon Char: 17 □ Transmit Xoff Char: 19

Winsock is: ○ Raw ● Telnet

Status: □ Connected □ RXD (2) □ TXD (3) 🟩 CTS (8) □ DCD (1) 🟩 DSR (6) □ Ring (9) □ BREAK □ Error

Char Count:1043 CPS:0 Port: 1 9600 8N1 None

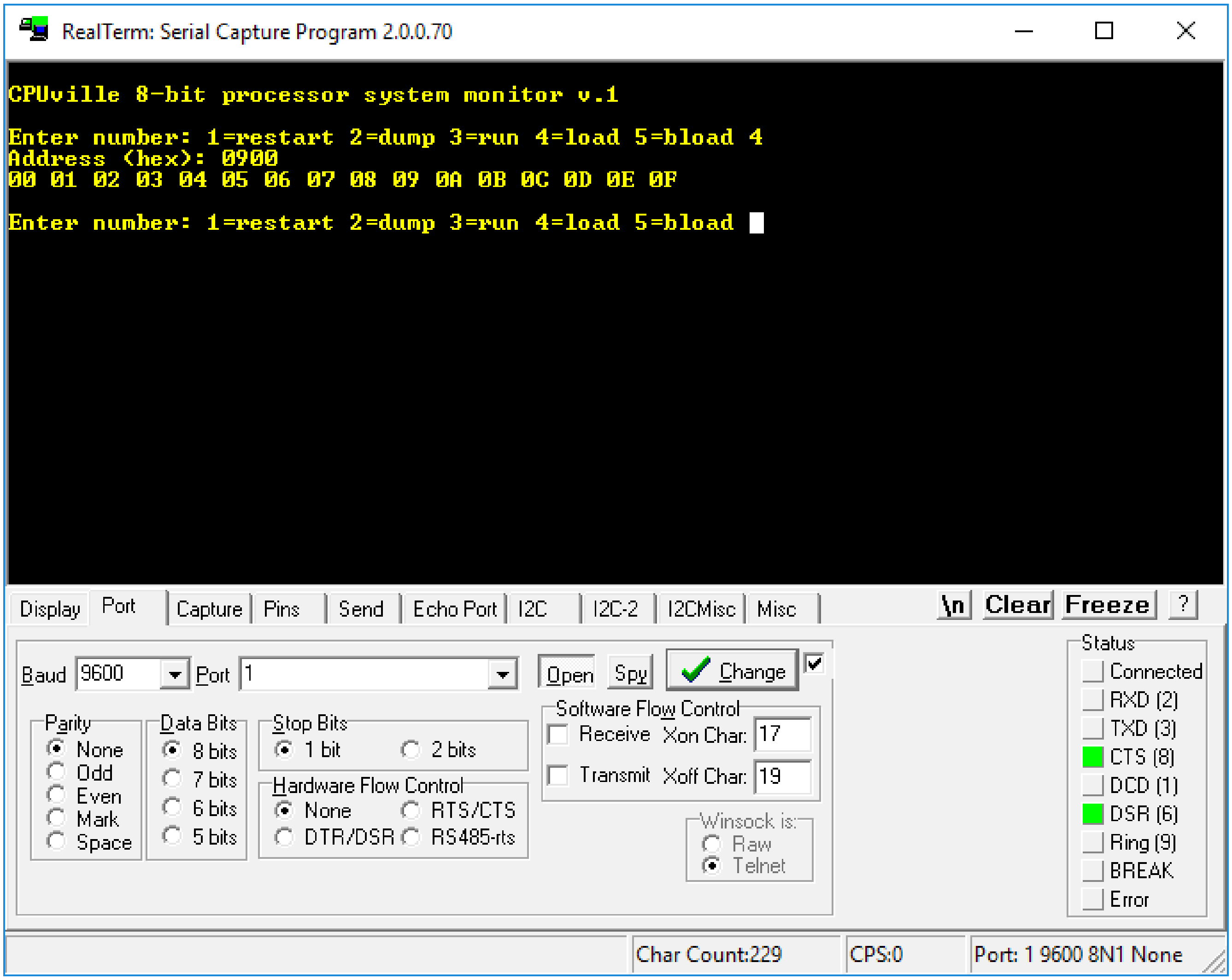
This command is useful for debugging programs in RAM, as you can see the machine code, and the values of your variables.

3=run

This causes the processor to jump to the address you enter and run code from there. See the example for `bload` below.

4=load

This command takes input from the keyboard, as hexadecimal characters, and loads the input into memory as binary byte values. Hit the Enter key to stop the input. During the load, the display shows 16-byte rows of input data in a manner similar to the `dump` command, without the addresses. Here is an example, entering the first 16 hexadecimal numbers into RAM starting at location 0x0900:



```
CPUville 8-bit processor system monitor v.1

Enter number: 1=restart 2=dump 3=run 4=load 5=bload 4
Address (hex): 0900
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

Enter number: 1=restart 2=dump 3=run 4=load 5=bload
```

Baud 9600 Port 1

Parity: ● None ○ Odd ○ Even ○ Mark ○ Space

Data Bits: ● 8 bits ○ 7 bits ○ 6 bits ○ 5 bits

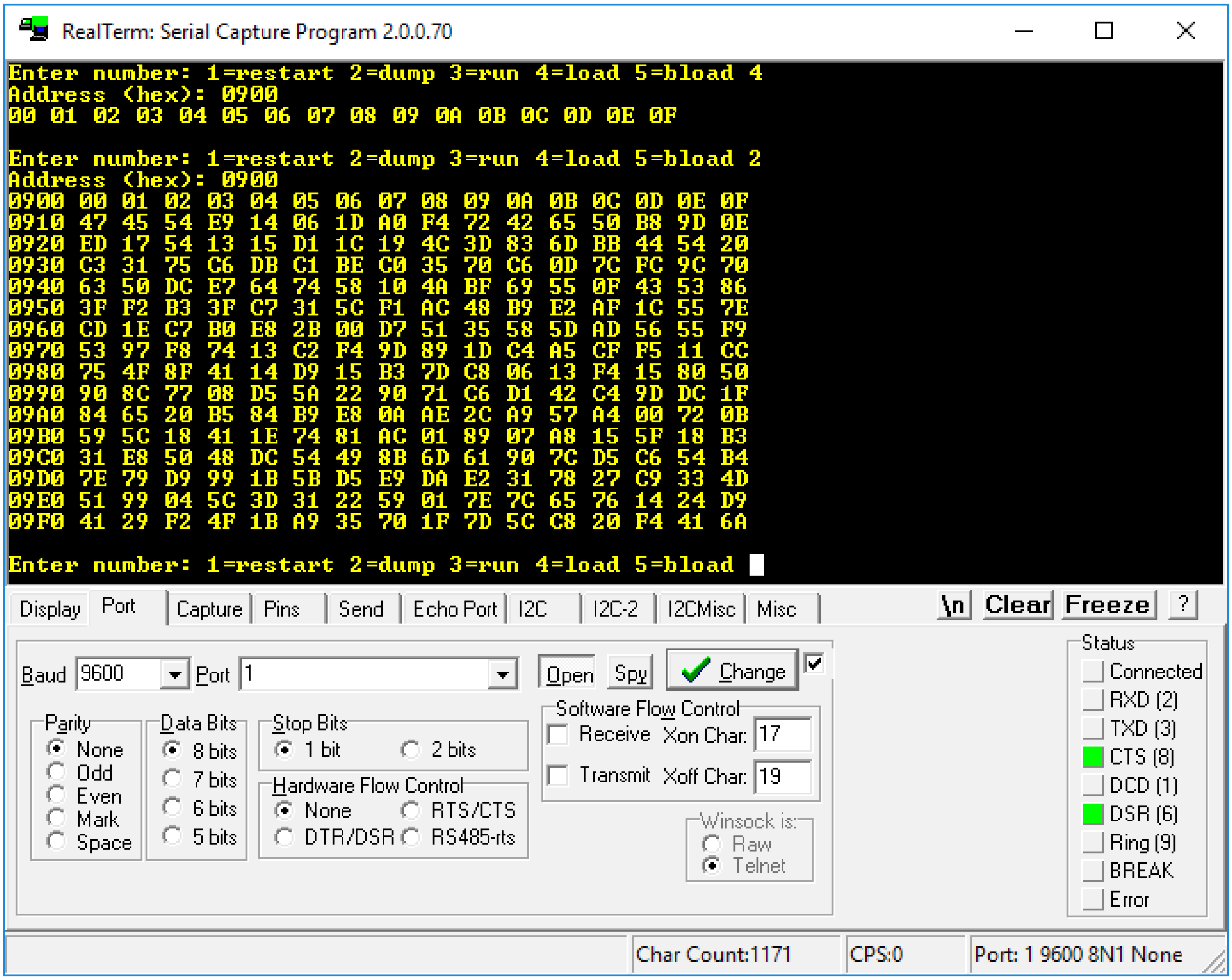
Stop Bits: ● 1 bit ○ 2 bits

Hardware Flow Control: ● None ○ RTS/CTS ○ DTR/DSR ○ RS485-rts

Software Flow Control: □ Receive Xon Char: 17 □ Transmit Xoff Char: 19

Char Count:229 CPS:0 Port: 1 9600 8N1 None

Here is a dump display of RAM starting at location 0x0900. You can see the 16 bytes I entered:



RealTerm: Serial Capture Program 2.0.0.70

```
Enter number: 1=restart 2=dump 3=run 4=load 5=bload 4
Address (hex): 0900
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

Enter number: 1=restart 2=dump 3=run 4=load 5=bload 2
Address (hex): 0900
0900 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0910 47 45 54 E9 14 06 1D A0 F4 72 42 65 50 B8 9D 0E
0920 ED 17 54 13 15 D1 1C 19 4C 3D 83 6D BB 44 54 20
0930 C3 31 75 C6 DB C1 BE C0 35 70 C6 0D 7C FC 9C 70
0940 63 50 DC E7 64 74 58 10 4A BF 69 55 0F 43 53 86
0950 3F F2 B3 3F C7 31 5C F1 AC 48 B9 E2 AF 1C 55 7E
0960 CD 1E C7 B0 E8 2B 00 D7 51 35 58 5D AD 56 55 F9
0970 53 97 F8 74 13 C2 F4 9D 89 1D C4 A5 CF F5 11 CC
0980 75 4F 8F 41 14 D9 15 B3 7D C8 06 13 F4 15 80 50
0990 90 8C 77 08 D5 5A 22 90 71 C6 D1 42 C4 9D DC 1F
09A0 84 65 20 B5 84 B9 E8 0A AE 2C A9 57 A4 00 72 0B
09B0 59 5C 18 41 1E 74 81 AC 01 89 07 A8 15 5F 18 B3
09C0 31 E8 50 48 DC 54 49 8B 6D 61 90 7C D5 C6 54 B4
09D0 7E 79 D9 99 1B 5B D5 E9 DA E2 31 78 27 C9 33 4D
09E0 51 99 04 5C 3D 31 22 59 01 7E 7C 65 76 14 24 D9
09F0 41 29 F2 4F 1B A9 35 70 1F 7D 5C C8 20 F4 41 6A

Enter number: 1=restart 2=dump 3=run 4=load 5=bload
```

Baud 9600 Port 1

Parity: ● None ○ Odd ○ Even ○ Mark ○ Space

Data Bits: ● 8 bits ○ 7 bits ○ 6 bits ○ 5 bits

Stop Bits: ● 1 bit ○ 2 bits

Hardware Flow Control: ● None ○ RTS/CTS ○ DTR/DSR ○ RS485-rts

Software Flow Control: □ Receive Xon Char: 17 □ Transmit Xoff Char: 19

Winsock is: ○ Raw ● Telnet

Status: □ Connected □ RXD (2) □ TXD (3) 🟩 CTS (8) □ DCD (1) 🟩 DSR (6) □ Ring (9) □ BREAK □ Error

Char Count:1171 CPS:0 Port: 1 9600 8N1 None

The rest of the RAM has digital garbage in it.

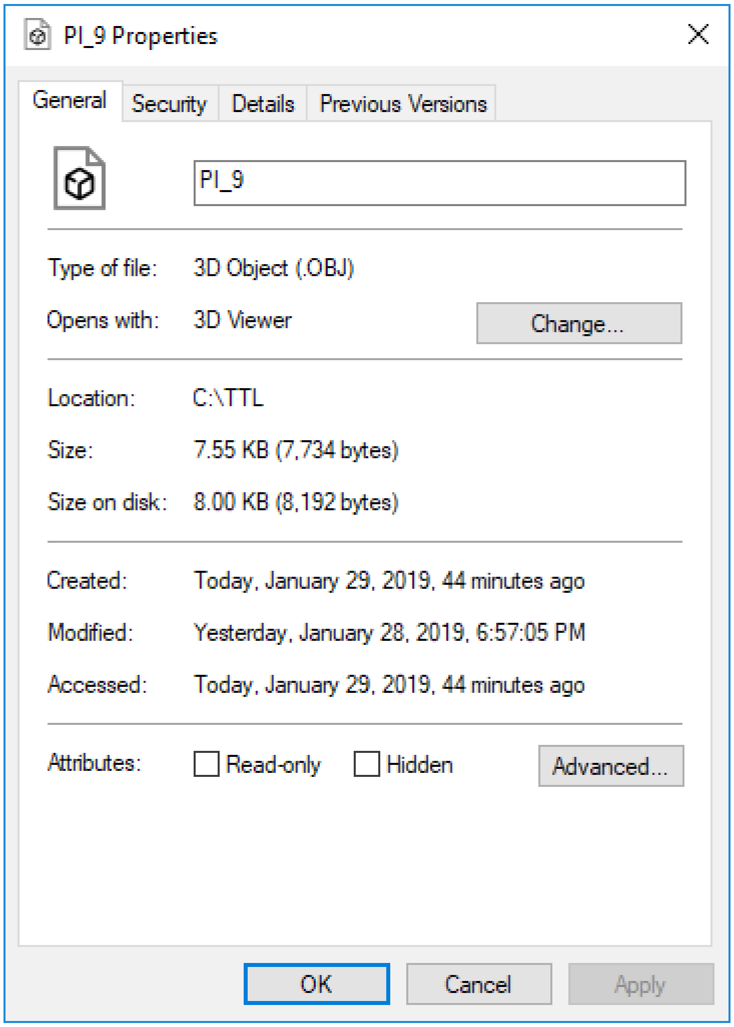
You can use the `load` command to quickly change a byte of program code or a variable, to clear memory by putting in zeros (just hold down the zero key, the repeats from the keyboard are entered), and to load small programs by hand.

5=bload

This command is for loading binary files (**b**inary **load**) over the serial interface into the computer memory. The command takes a four-character hexadecimal address input, and a decimal file length input. Then, it waits for the file to be sent from the PC to the kit computer. It works best if you enter the exact length of the binary file. The `bload` command will hang if the file is shorter than the length you enter.

The following is an example of loading a binary file using the `bload` command. We will load and execute the program PI_9.OBJ which calculates a value for pi using a numerical integration. The assembly language, list file, and binary object file for this program can be found on the CPUville website.

The program has been assembled to load and run from the beginning of the user RAM at 0x0900[4]. First we need the exact file size, which we can obtain by hovering over the file name, or right-click-Properties:



PI_9

Type of file: 3D Object (.OBJ)

Opens with: 3D Viewer

Location: C:\TTL

Size: 7.55 KB (7,734 bytes)

Size on disk: 8.00 KB (8,192 bytes)

Created: Today, January 29, 2019, 44 minutes ago

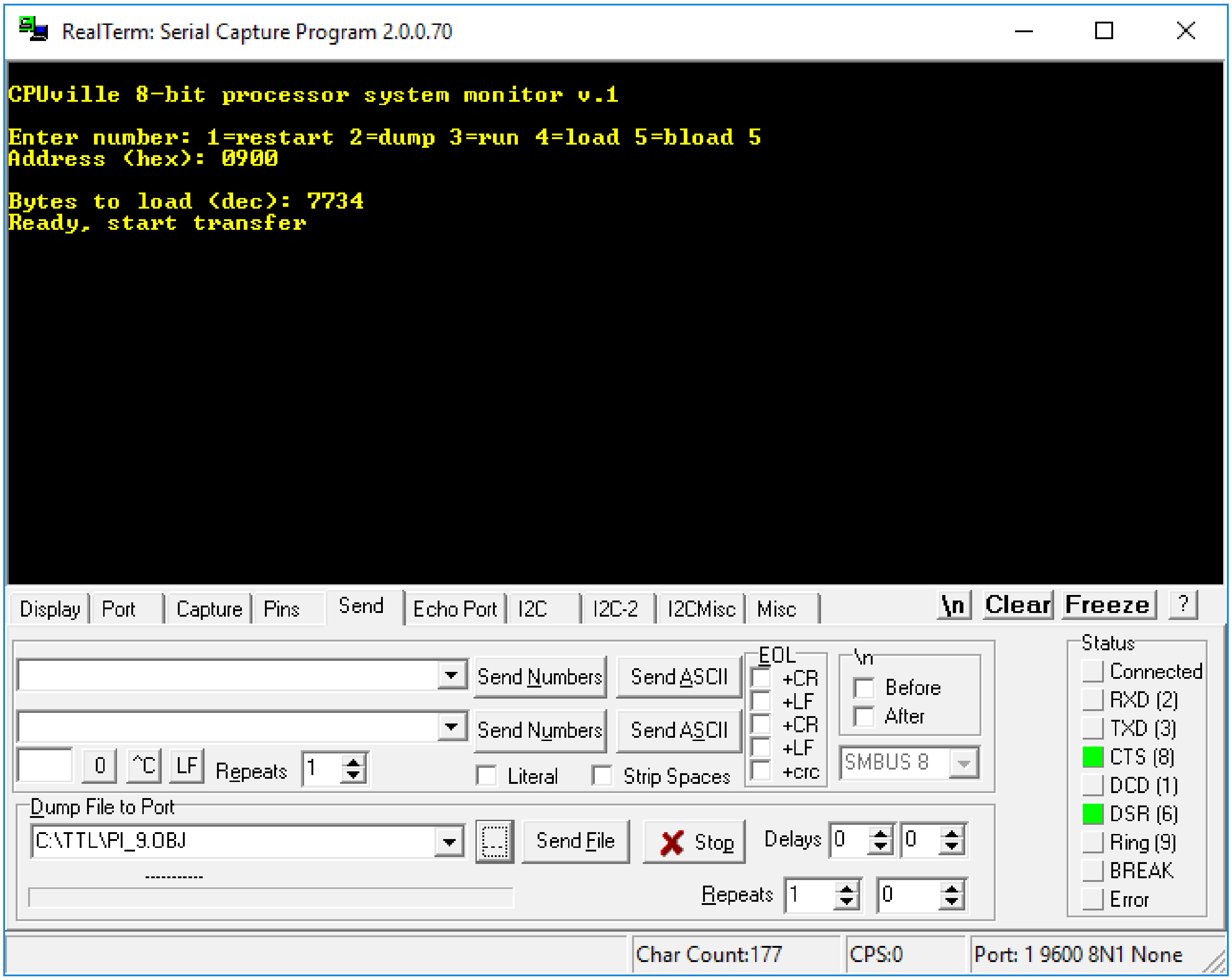
Modified: Yesterday, January 28, 2019, 6:57:05 PM

Accessed: Today, January 29, 2019, 44 minutes ago

Attributes: □ Read-only □ Hidden

We need the exact size, which is 7734 bytes. Now we run the `bload` command, and enter the target address as 0900. Hit the enter key, then enter the file length as decimal 7734. Hit enter after entering the length. The monitor program displays “Ready, start transfer”.In the RealTerm Send tab, we navigate to the PI_9.OBJ file using the … button. Click Open, and RealTerm is set up to send the file from the PC to the serial port:

[4] The first page of RAM, from 0x0800 to 0x08FF is reserved for workspace for the system monitor.

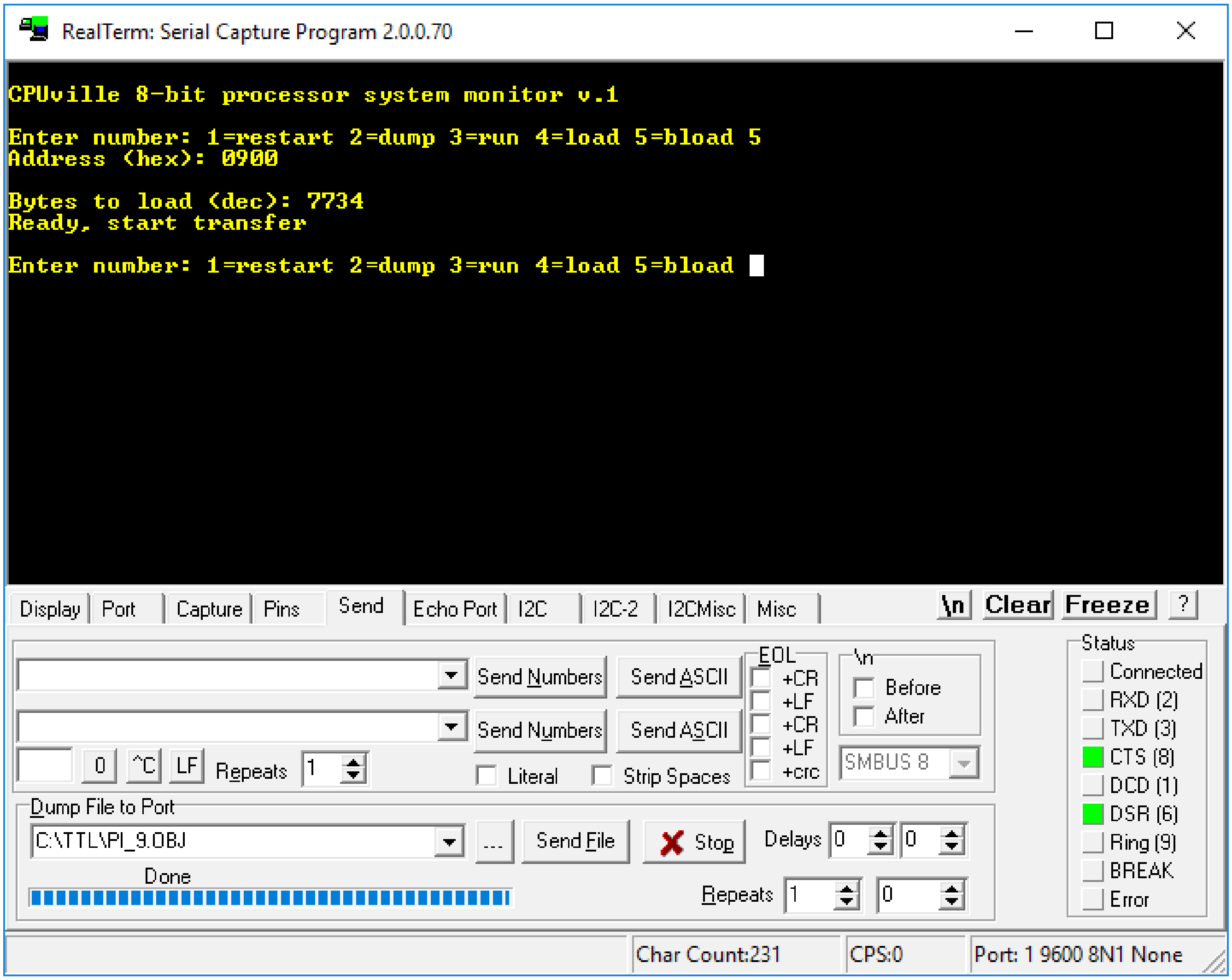


```
CPUville 8-bit processor system monitor v.1

Enter number: 1=restart 2=dump 3=run 4=load 5=bload 5
Address (hex): 0900

Bytes to load (dec): 7734
Ready, start transfer
```

Dump File to Port: C:\TTL\PI_9.OBJ

Char Count:177 | CPS:0 | Port: 1 9600 8N1 None

Now we click the Send File button. The Dump File to Port area background turns red, and a blue progress bar is shown. After RealTerm has finished sending the file the background turns white again, and "Done" appears above the blue file progress bar. The monitor commands will reappear in the RealTerm port window, letting you know the command was successfully executed:

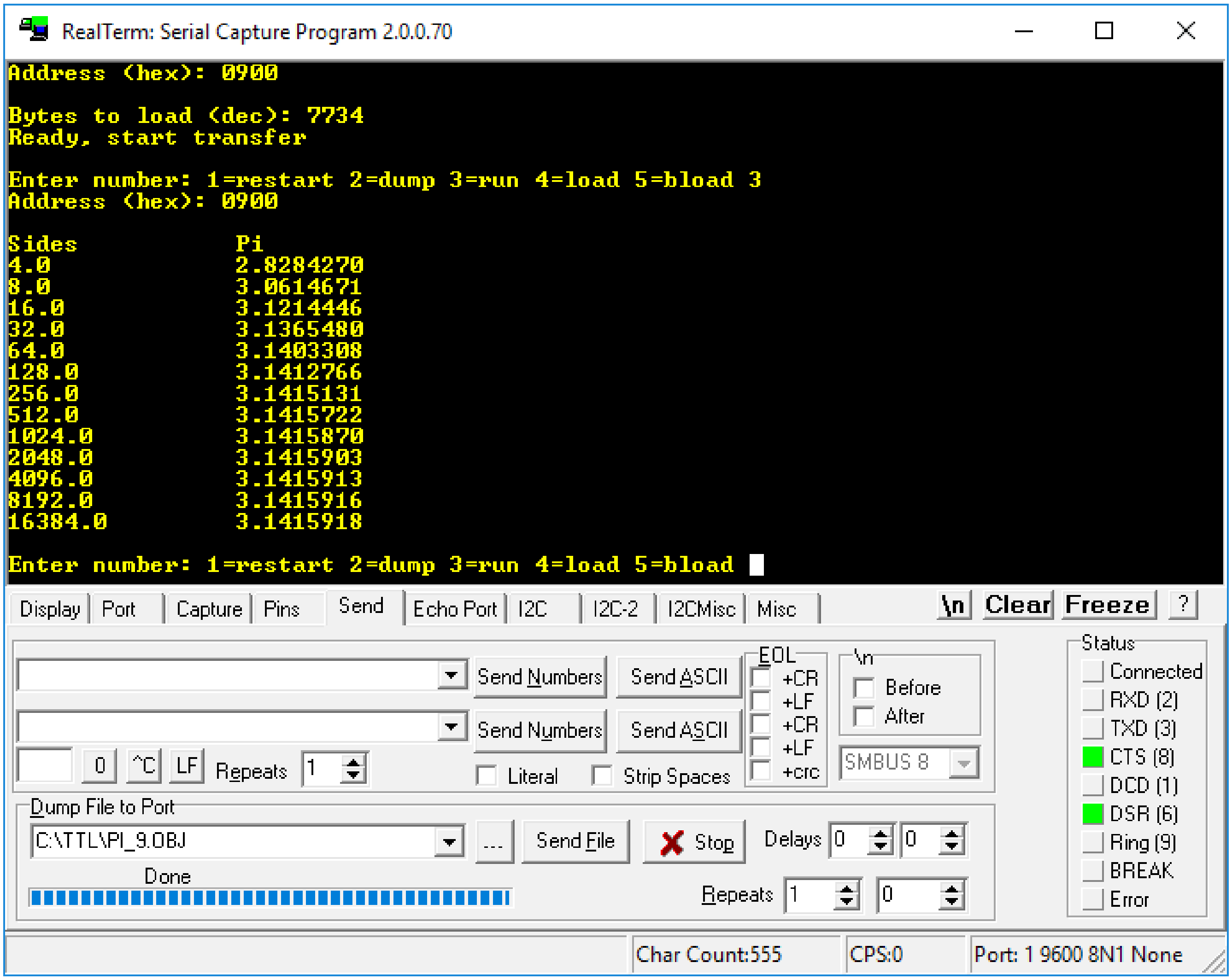


```
CPUville 8-bit processor system monitor v.1

Enter number: 1=restart 2=dump 3=run 4=load 5=bload 5
Address (hex): 0900

Bytes to load (dec): 7734
Ready, start transfer

Enter number: 1=restart 2=dump 3=run 4=load 5=bload 
```

Dump File to Port: C:\TTL\PI_9.OBJ

Char Count:231 CPS:0 Port: 1 9600 8N1 None

Now we will run the program using the `run` command. We enter the starting address of the program, hit Enter, and the program runs, printing out a list of values of pi calculated using polygons of increasing numbers of sides:



```
Address (hex): 0900

Bytes to load (dec): 7734
Ready, start transfer

Enter number: 1=restart 2=dump 3=run 4=load 5=bload 3
Address (hex): 0900

Sides           Pi
4.0             2.8284270
8.0             3.0614671
16.0            3.1214446
32.0            3.1365480
64.0            3.1403308
128.0           3.1412766
256.0           3.1415131
512.0           3.1415722
1024.0          3.1415870
2048.0          3.1415903
4096.0          3.1415913
8192.0          3.1415916
16384.0         3.1415918

Enter number: 1=restart 2=dump 3=run 4=load 5=bload
```

Repeats 1

Dump File to Port: C:\TTL\PI_9.OBJ

Done

Delays 0 0

Repeats 1 0

☑ CTS (8) ☑ DSR (6)

Char Count:555 CPS:0 Port: 1 9600 8N1 None

Connecting a disk drive

The 64K systems described above have IDE disk drive connectors. A variety of drives have been tested with these systems using a Z80 as the processor (see the Table of Tested Disk Drives in the instruction manual for the Disk and Memory Expansion kit or the Single-board Z80 kits). However, as of writing this manual I have not written code to test a disk drive with the 8-bit processor. There is no reason a disk drive should not work, but the code is not yet available. If you have written code to use a disk drive, please let me know, and I will post it on the CPUville website. For details on connecting a disk drive, see the manuals for the Disk and Memory Expansion kit or the Single-board Z80 kit. A 40-pin right-angle adapter can be used to attach a disk module to the system board when it is in a stack with the processor boards.

Introduction to Programming for the 8-bit Processor

The CPUville 8-bit processor was designed to be a very simple, yet complete, general-purpose processor. It is simple, in that it has a small instruction set and simple design architecture, but complete, in that it has all the instructions needed to perform the tasks of any computer. The trade-off is that with a simple design and instruction set, programming is made more difficult than it would be for a processor with a complex design and instruction set. For example, a more complex processor will have registers and instructions that allow address indexing, but the CPUville 8-bit processor lacks these. Therefore, to perform address indexing, one has to place the instruction with the address to be indexed in RAM, and index the address portion of that instruction (see the "Special Programming Techniques" section in the Appendix). But I believe the trade-off makes for a less expensive and more easily understandable processor. Since the goal of this processor kit is understanding, and not processing power or programming convenience, I think the trade-off is worth it.

The Instruction Set

Listed in this section are the instructions implemented by the CPUville 8-bit processor, in alphabetical order of the assembly language mnemonics. These mnemonics are those I have chosen to use with the TASM assembler, but these can be changed to suit the user by changing the TASM assembly language table. See the "Using TASM" section of this manual. The instruction set is also summarized in tables found in the Appendix.

The instruction format is variable, that is, an instruction can be one, two or three bytes long. The first byte is always the opcode. The processor has 30 instructions, and the lower 5 bits of the opcode byte are used. The upper three bits are ignored. Optional one- and two-byte operands follow the opcode. One byte (8-bit) operands are either data or port addresses, and two-byte (16-bit) operands are memory locations. The CPUville 8-bit processor uses the little-endian model for storing two-byte (16-bit) instruction address operands. That is, the low-order byte of the address operand is stored in the lower memory location, and the high-order byte is stored in the higher memory location. For example, the instruction JMP 0CF34H, with the address operand CF34, is stored in memory as hex bytes 13, 34, CF (13 is the hex opcode for JMP).

The programming model is a simple one: all arithmetic-logic instructions that use two operands take one operand from the accumulator, and the other from either memory or from the instruction itself (immediate addressing). The result is always placed in the accumulator. The mnemonics for the arithmetic-logic instructions that take the second operand from memory are plain, thus ADD, OR etc., and the mnemonics for the instructions that take the second operand from the instruction itself append IM (for immediate), thus ADDIM, ORIM etc. There are also special arithmetic instructions that increment the accumulator by one (INC) or decrement the accumulator by one (DEC), and a CMP (compare) instruction that performs a subtract-immediate operation that only affects the carry flag, and does not change the value in the accumulator.

The processor has three flags, zero, minus and carry. The processor flags minus and zero are not registered, that is, they reflect what is currently in the accumulator. This can be useful when scanning input or output for a zero or negative byte, since no arithmetic operation is needed. The carry flag is registered, and contains the carry-out bit from the most recent arithmetic operation, or as set by the most recent CCF (clear carry flag) or SCF (set carry flag) instructions. One quirk of this processor is that for subtract operations, a borrow request sets the carry flag to 0. That is, in A – B, if B > A the carry flag will be 0, otherwise it will be 1.

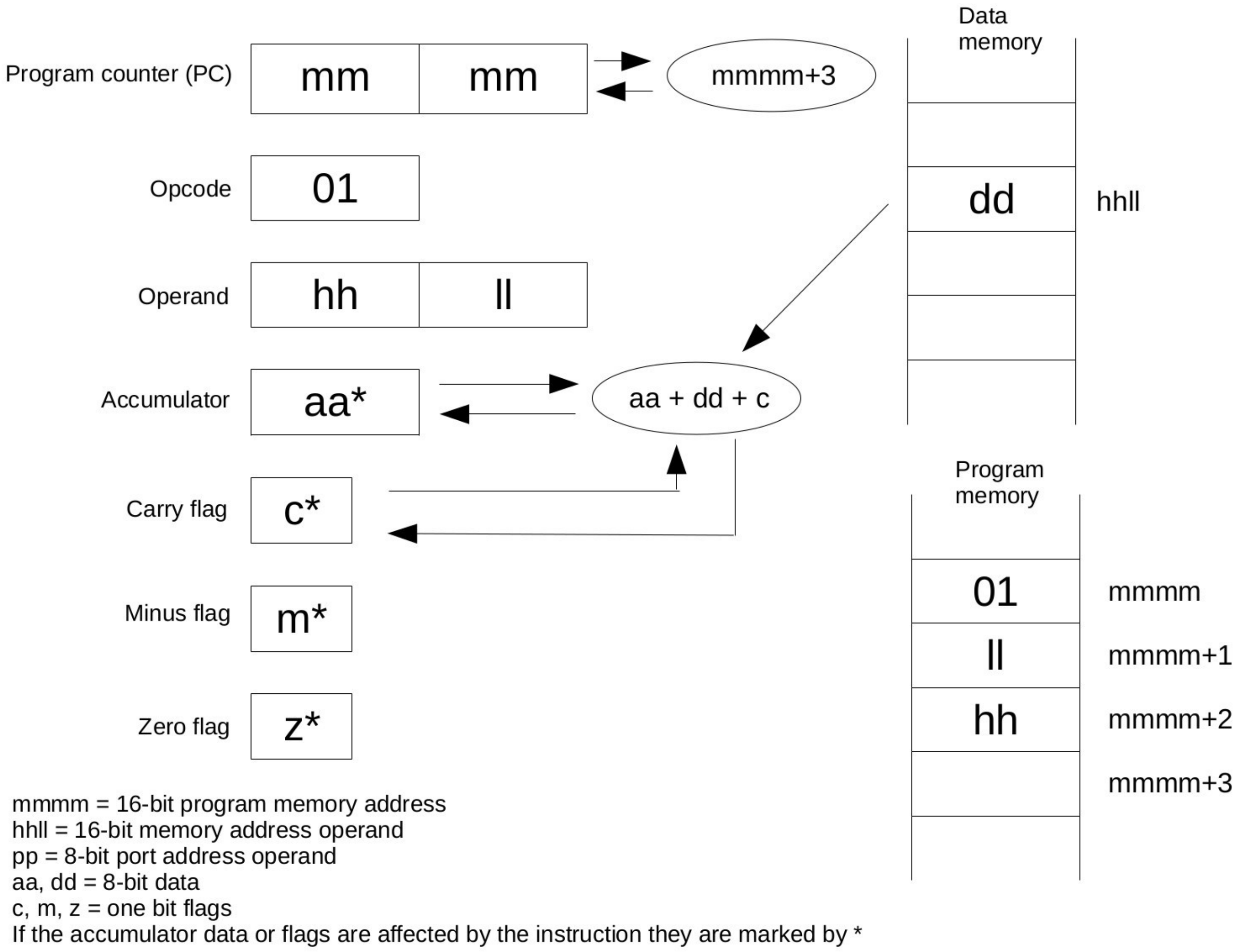
All transfers of data between the processor and memory or ports go through the accumulator. That is, the accumulator always serves as either a source or destination of a transfer. The data transfer instructions are STM (store accumulator to memory), LDM (load accumulator from memory), LDI (load accumulator immediate), IN (load accumulator with a byte from an input port), and OUT (send a byte from the accumulator to the output port).

There is a simple set of flow-of-control instructions. These are the unconditional jump JMP, and the conditional jumps JPM[5] (jump if minus), JPC (jump if carry), and JPZ (jump if zero).

There is a no-operation instruction (NOP) for placeholding and other purposes.

The processor treats operands of arithmetic operations as unsigned integers, that is, there are no hardware facilities for sign extension. There is no overflow flag. When working with signed integers, the programmer must allow for this, and watch for overflow and do sign extension in software.

ADC – add with carry



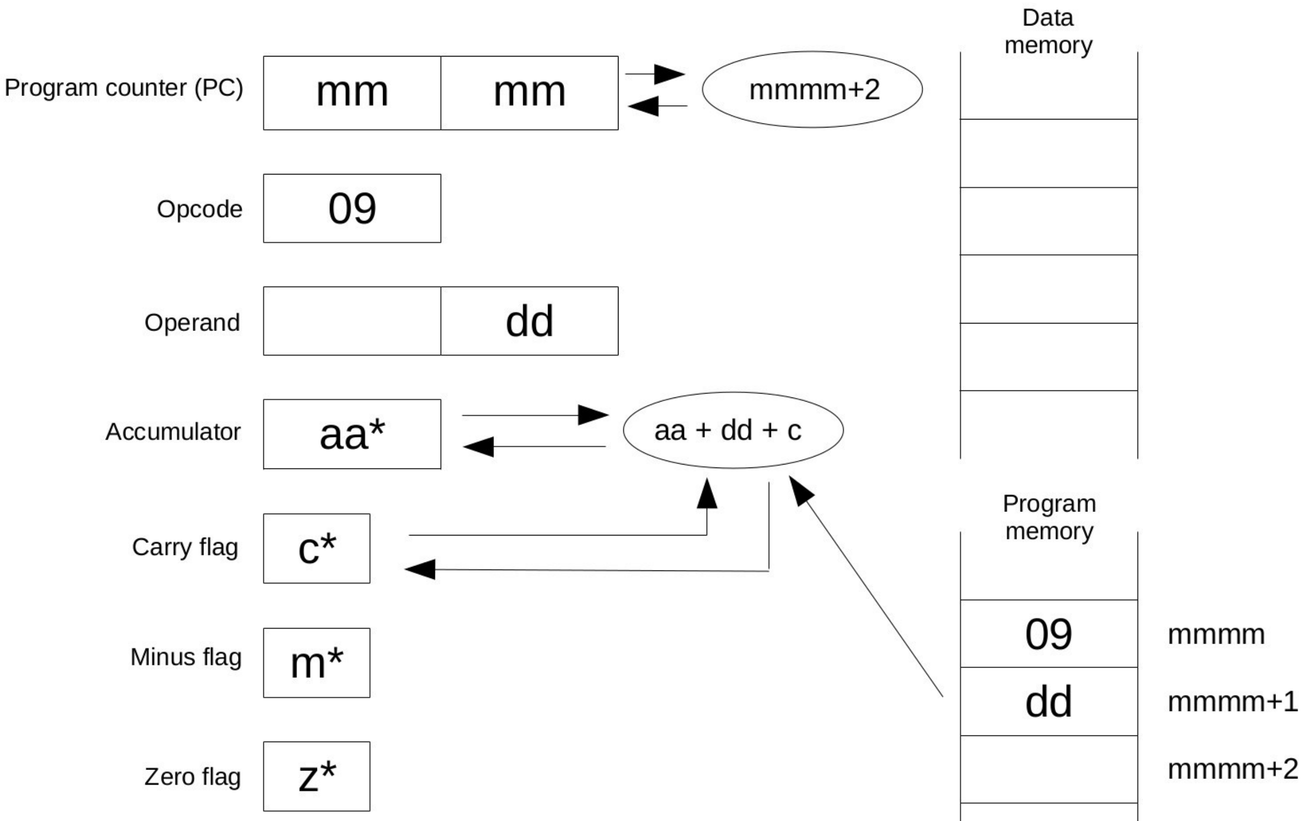
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction adds the value of the accumulator and the carry flag to the value in the memory location referenced by the operand to the instruction. It places the result in the accumulator, and replaces the carry flag with the carry-out from this operation. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and

[5] The most frequent programming error I have made is confusing the JMP and JPM instructions. If you want, you can change the mnemonics by changing the assembly language table.

the most significant byte in the higher memory address location.

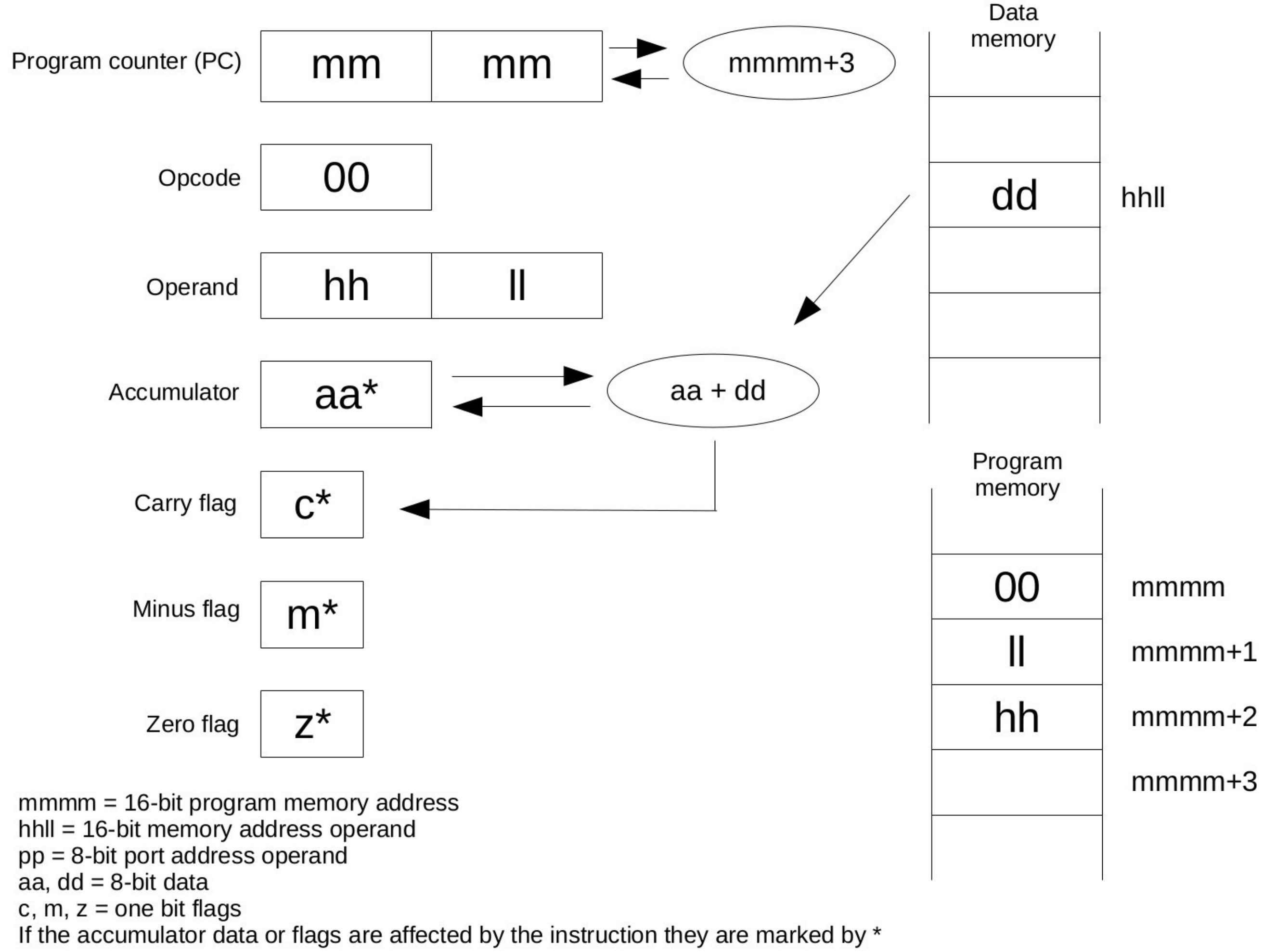
ADCIM – add with carry, immediate



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction adds the instruction operand and the carry flag to the accumulator.

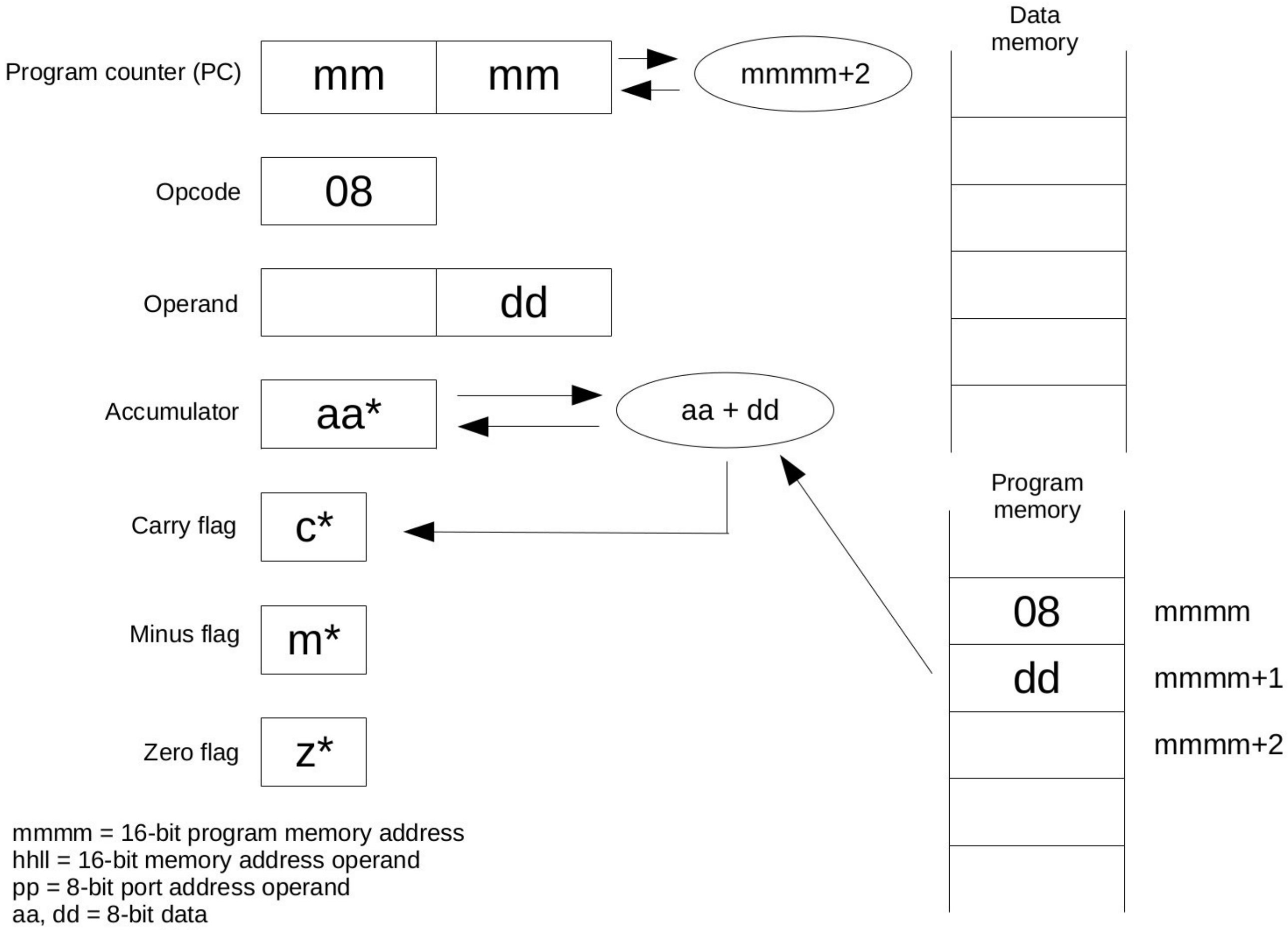
ADD – add memory data to the accumulator



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction adds the byte value contained in the memory location referenced by the two-byte instruction operand to the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

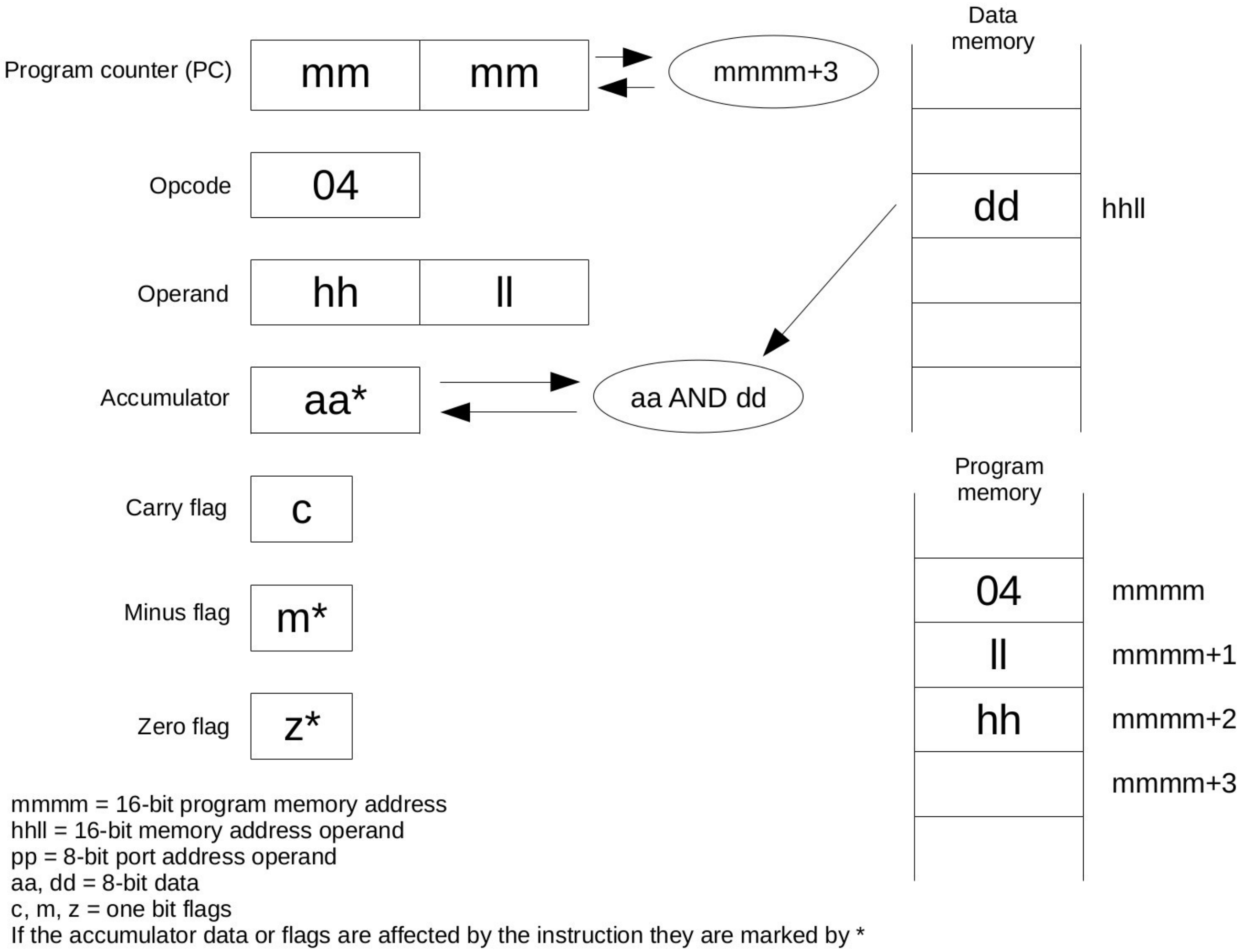
ADDIM – add immediate data to the accumulator



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction adds the byte value of the instruction operand to the accumulator.

AND – bitwise logical AND of memory data with the accumulator

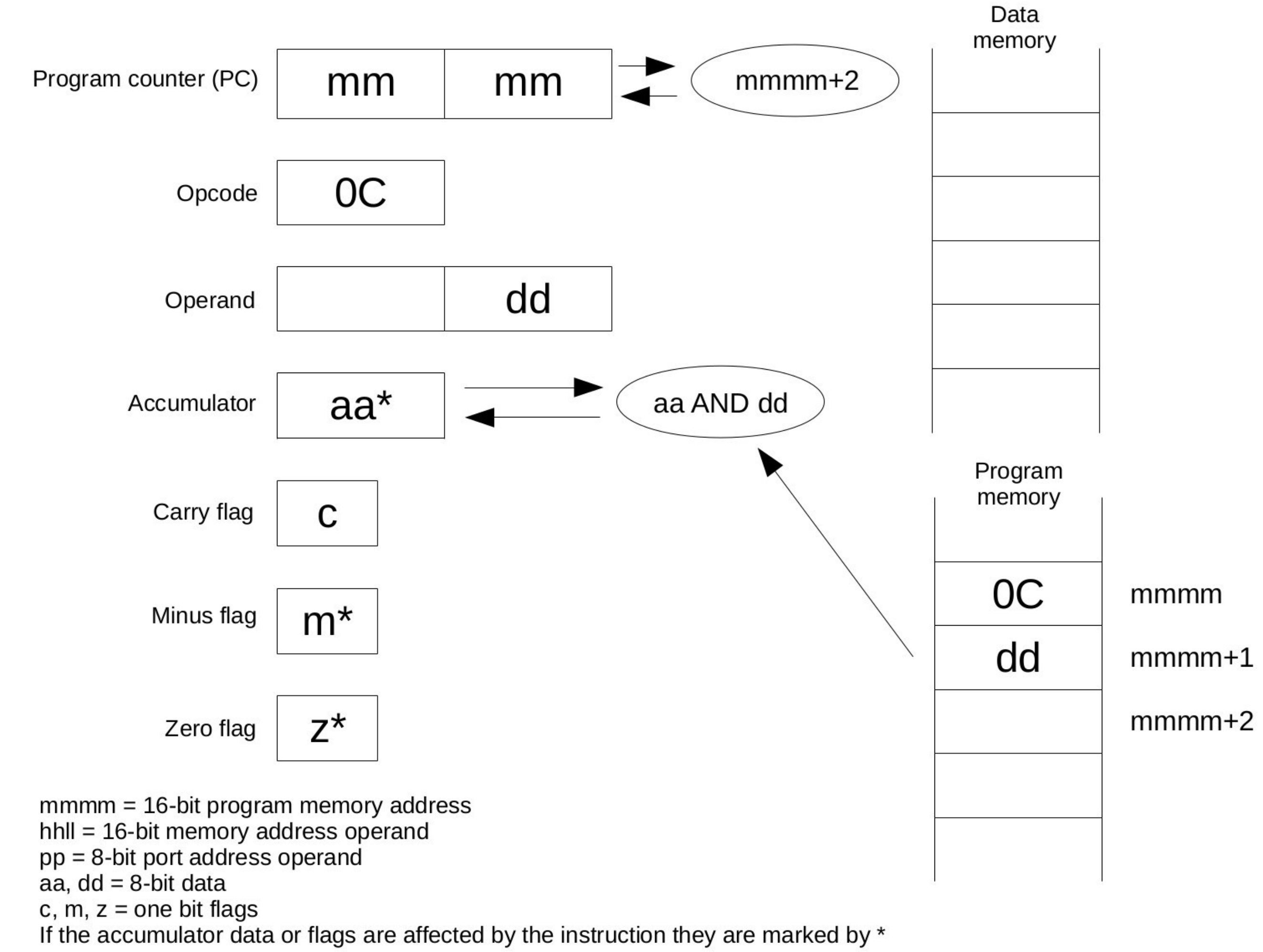


This instruction performs a bitwise AND operation on the accumulator and byte from memory, placing the result in the accumulator. For example, if dd = 1011 0101b and aa = 1100 0011b:

```
    1011 0101
AND 1100 0011
    1000 0001
```

The value 1000 0001 will be placed in the accumulator. The carry flag is not affected, but the minus and zero flags are affected, as they are in all operations that affect the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

ANDIM – bitwise logical AND of immediate data with the accumulator



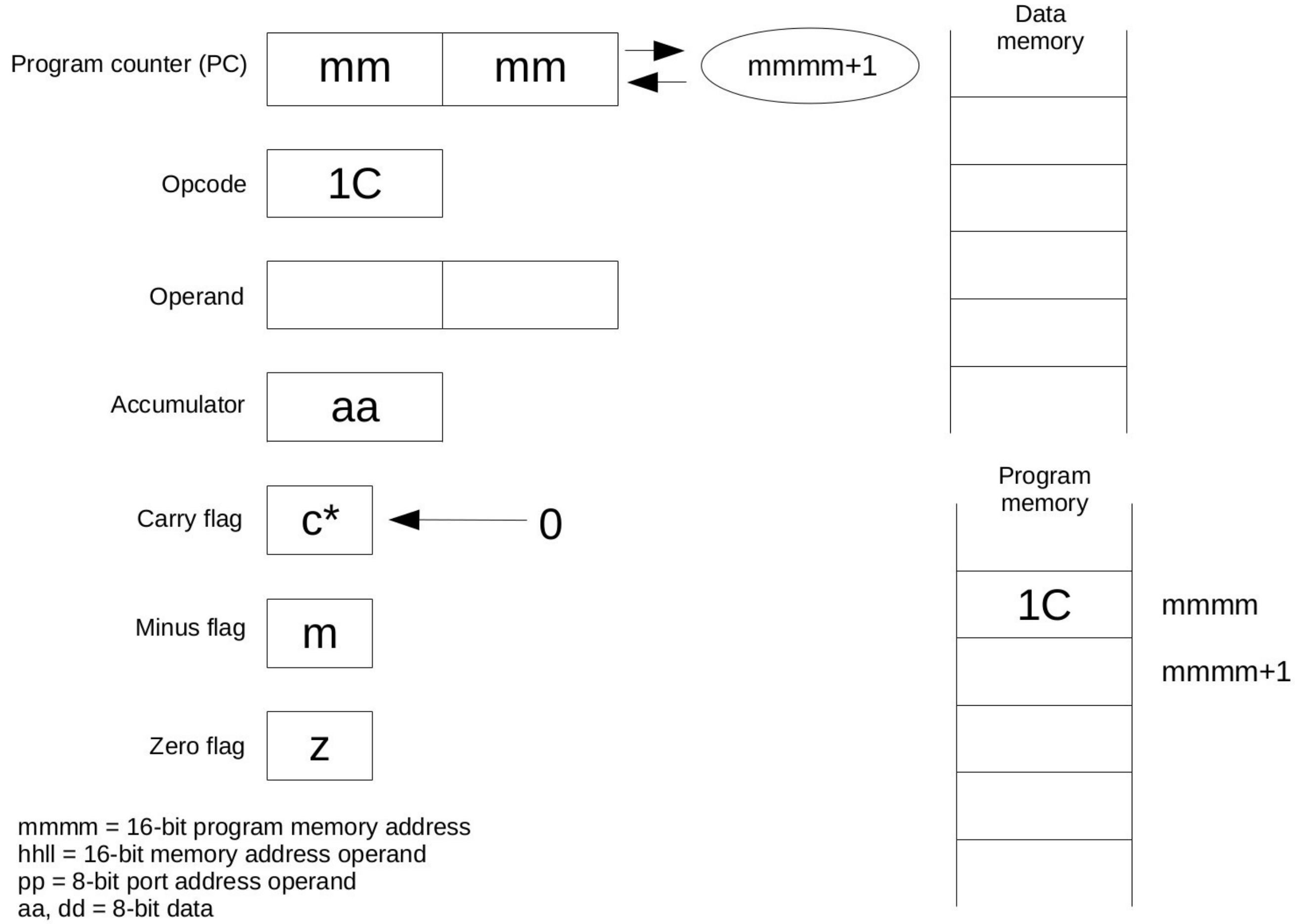
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction performs a bitwise AND operation on the accumulator and a one byte instruction operand, placing the result in the accumulator. For example, if dd = 0011 0101b and aa = 1010 0111b:

```
    0011 0101
AND 1010 0111
    0010 0101
```

The value 0010 0101b will be placed in the accumulator. The carry flag is not affected, but the minus and zero flags are affected, as they are in all operations that affect the accumulator.

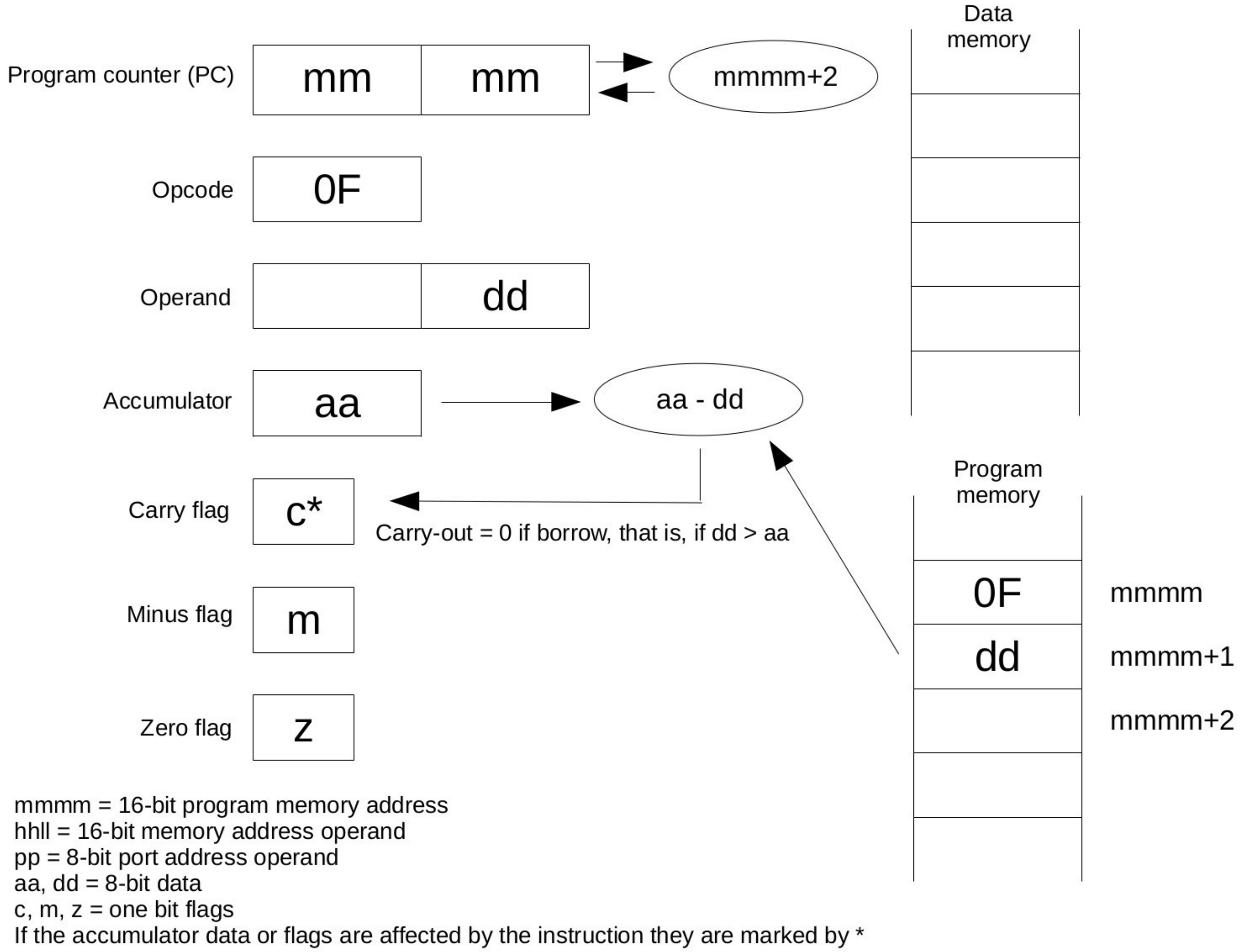
CCF – clear carry flag



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction places the bit value 0 in the carry flag flip-flop. The accumulator and other flags are not affected. It is the complement of the SCF (set carry flag) instruction.

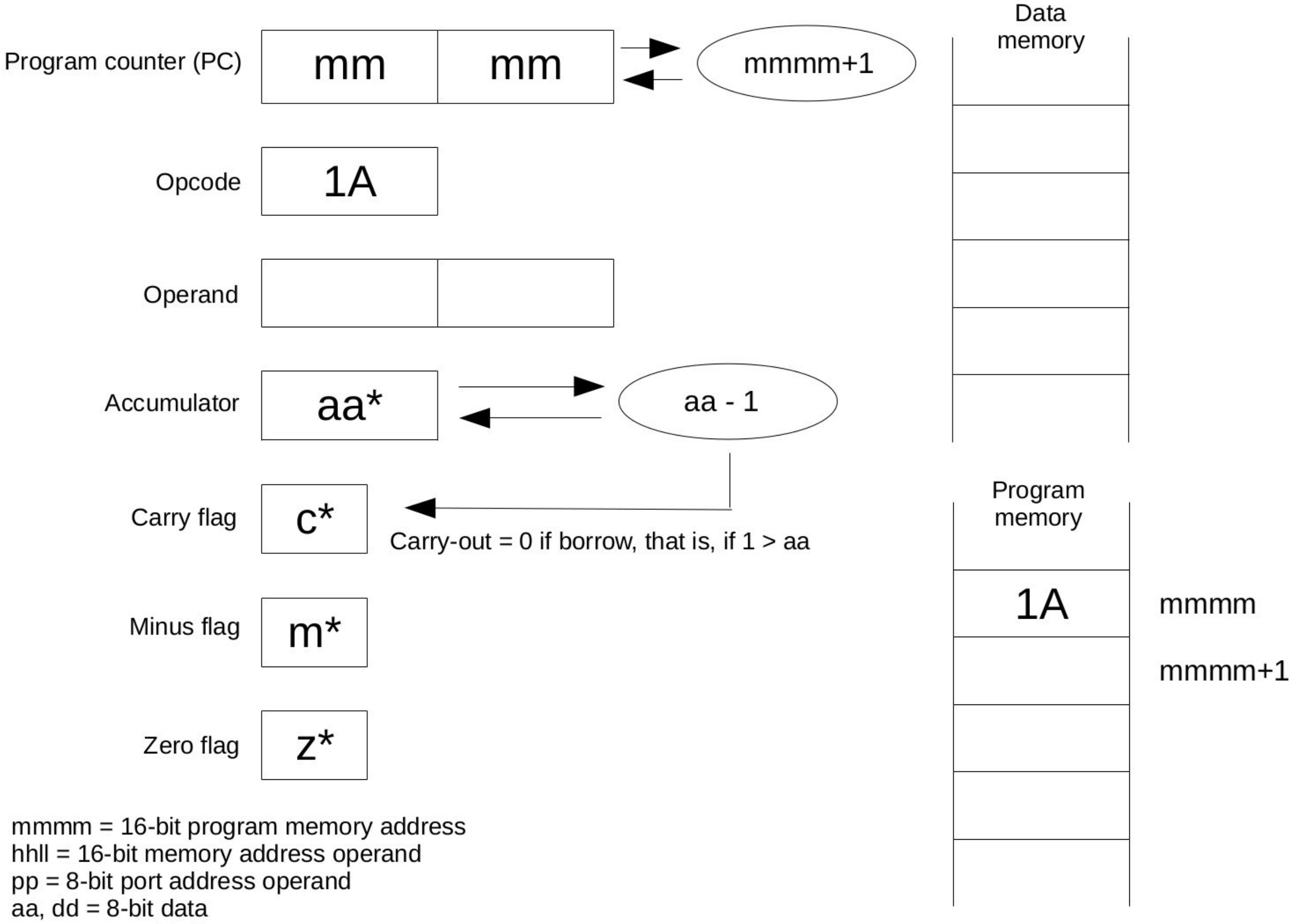
CMP – compare



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction subtracts the immediate data in the operand of the instruction from the accumulator, and changes the carry flag to reflect the result of the operation. If dd > aa, the carry flag will be 0, otherwise it will be set to 1, as in all subtraction operations. The accumulator and the other flags are not affected.

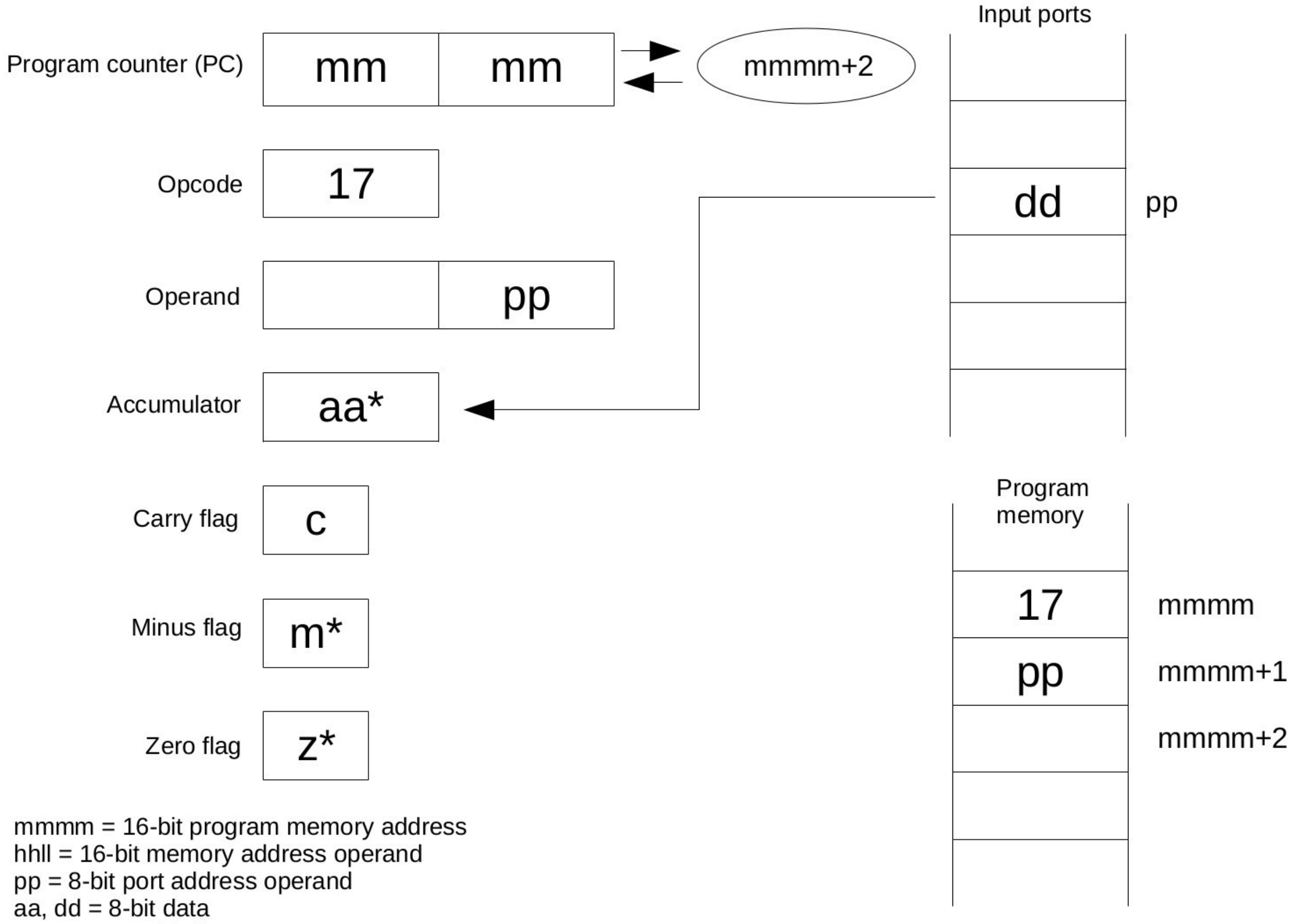
DEC – decrement accumulator



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction subtracts 1 from the accumulator, and the result is placed back in the accumulator. If aa < 1, (that is, if aa = 0) the carry out will be 0 (borrow request), otherwise it will be 1, as in all subtraction operations. The minus and zero flags will be affected, as they are in all instructions that affect the accumulator.

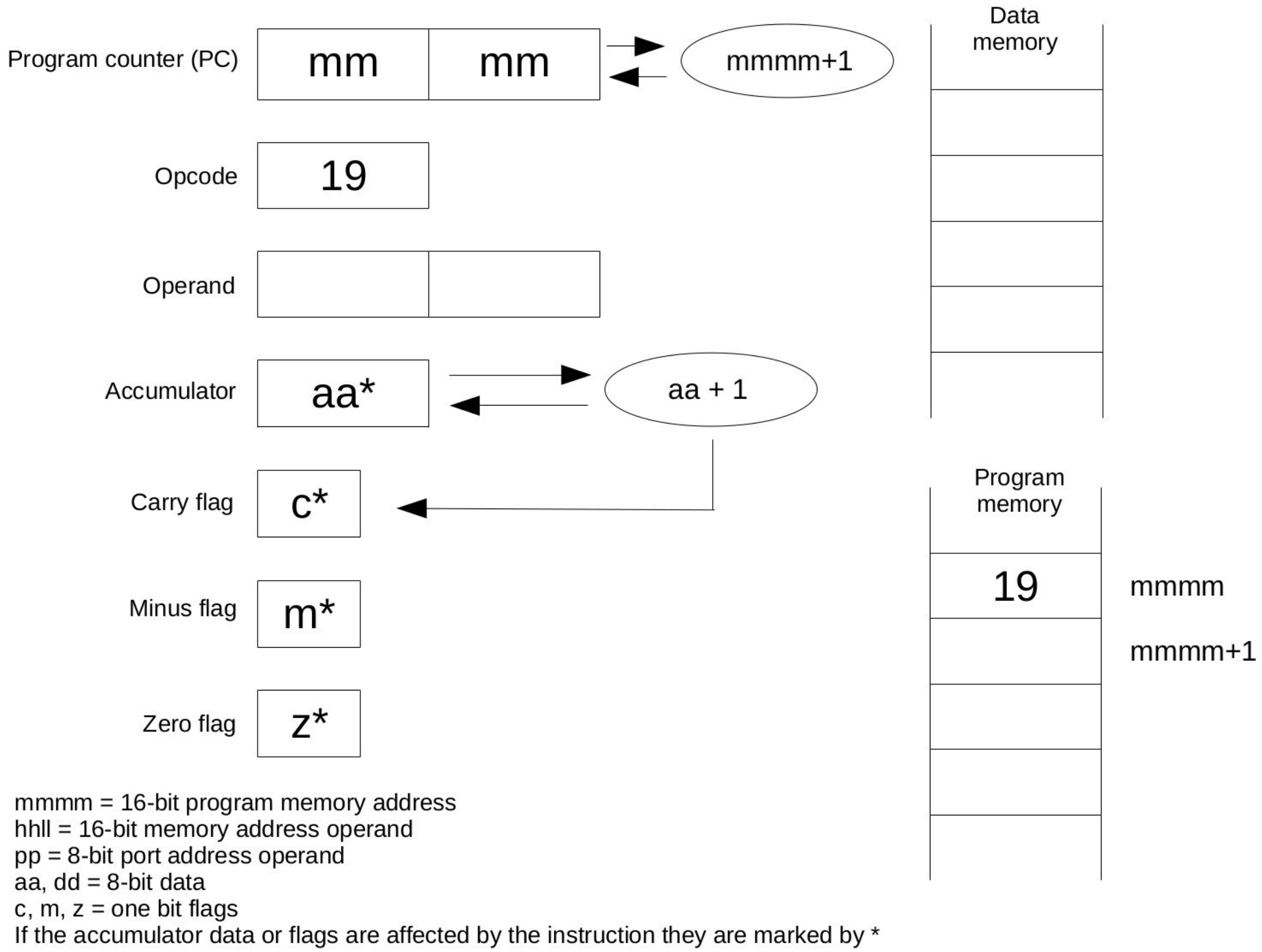
IN – load accumulator with data from an input port



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction places a byte value from an input port into the accumulator. The port address is the one-byte operand of the instruction. The minus and zero flags are affected, as they are with any instruction that affects the accumulator.

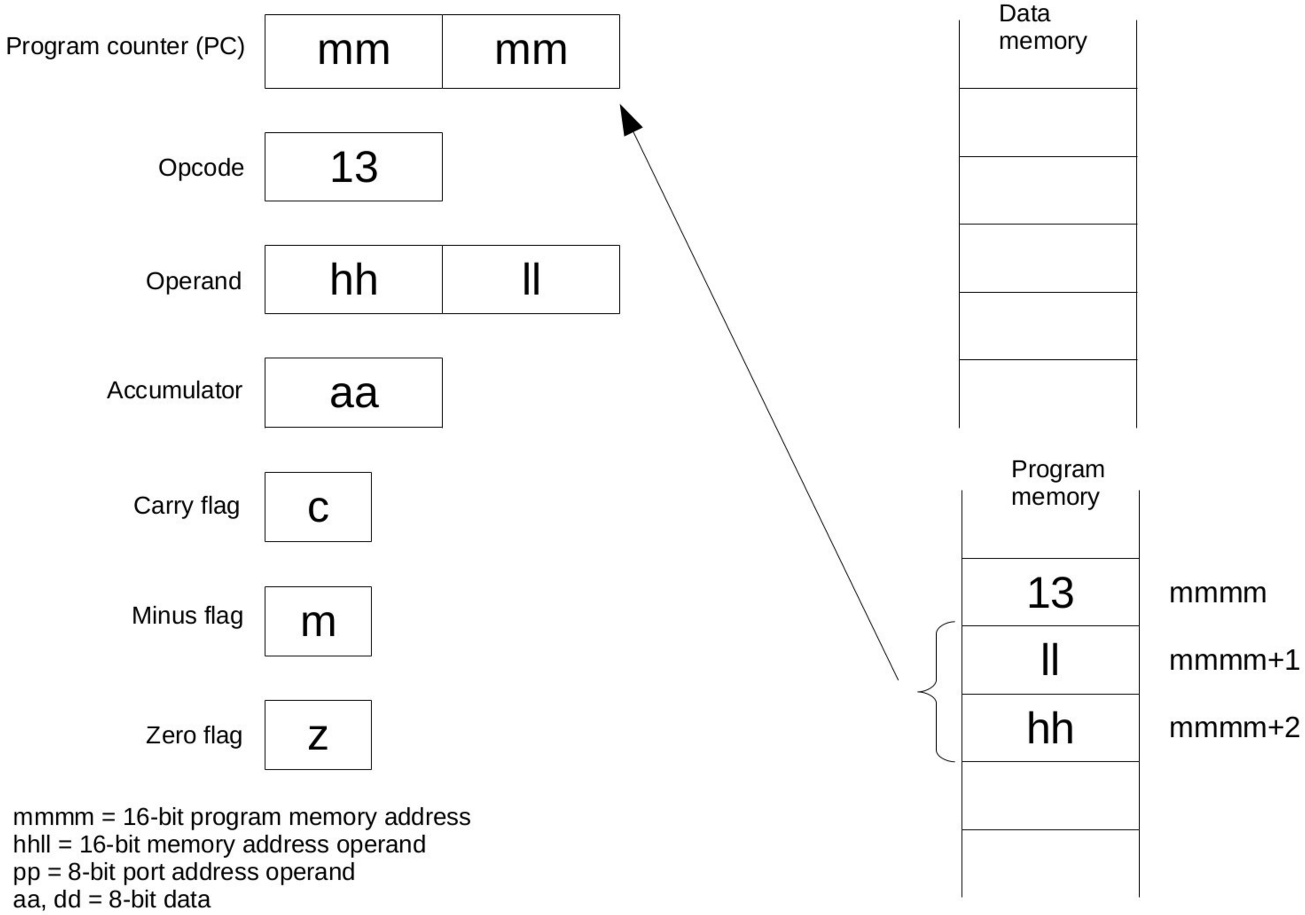
INC – increment accumulator



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

Adds one (increments) the accumulator. The flags are all affected.

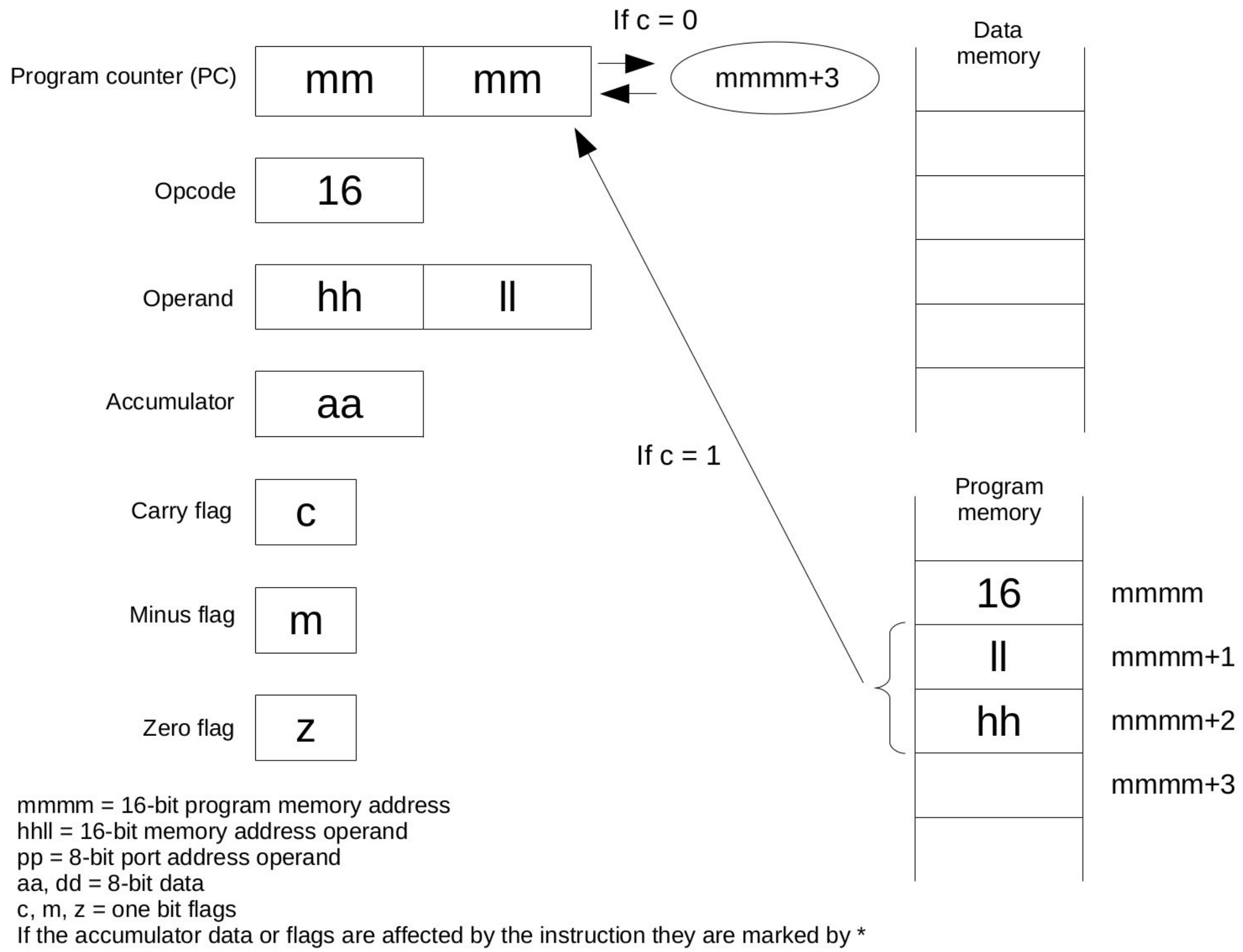
JMP – jump unconditional



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction places the 16-bit memory address operand of the instruction into the program counter, causing program flow to jump to the instruction at that address. The accumulator and flags are not affected. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

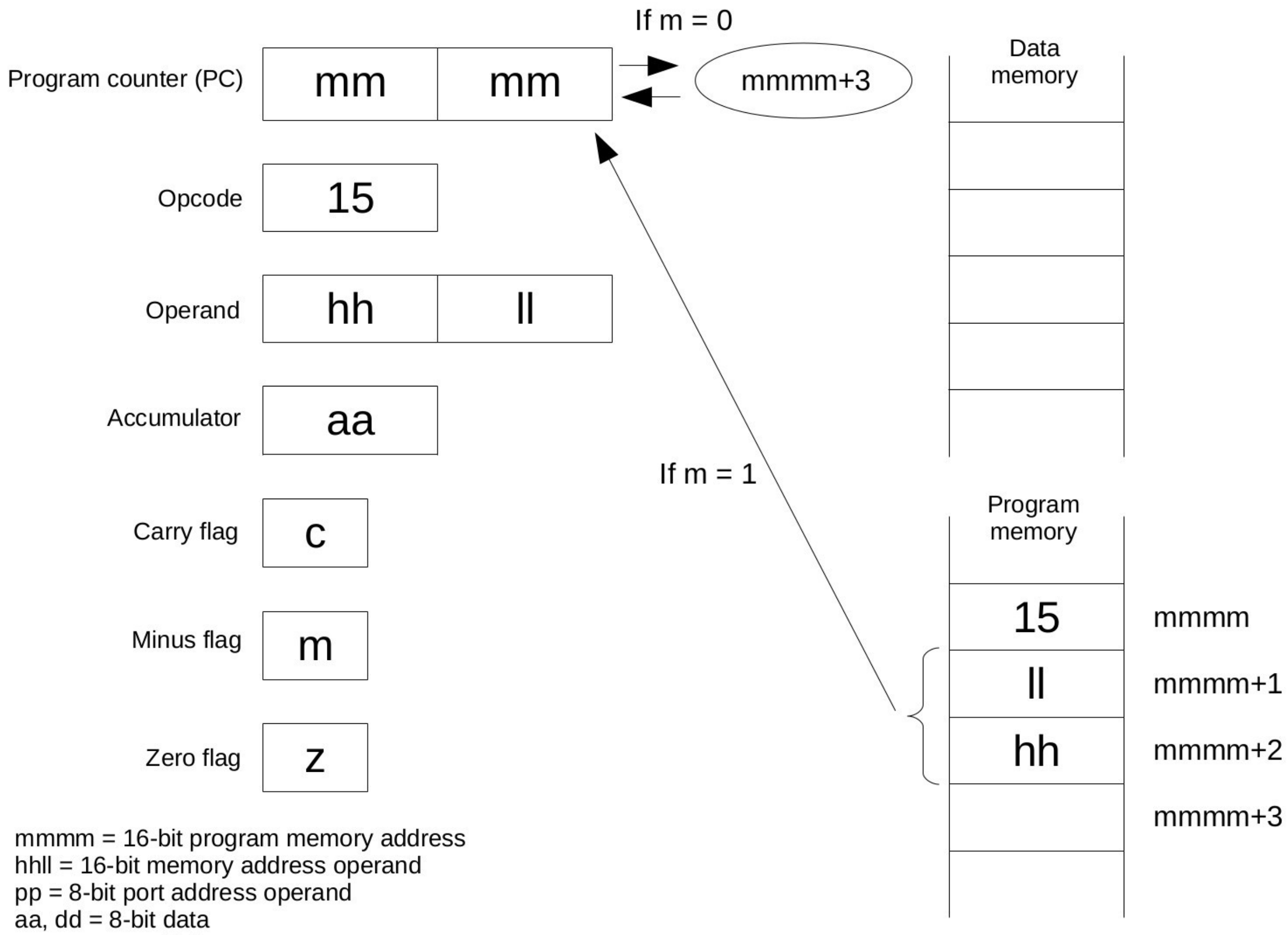
JPC – jump if carry



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction places the 16-bit memory address operand of the instruction into the program counter, causing program flow to jump to the instruction at that address, if the carry flag is 1, otherwise the program counter continues with normal flow at the instruction address + 3. The accumulator and flags are not affected. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

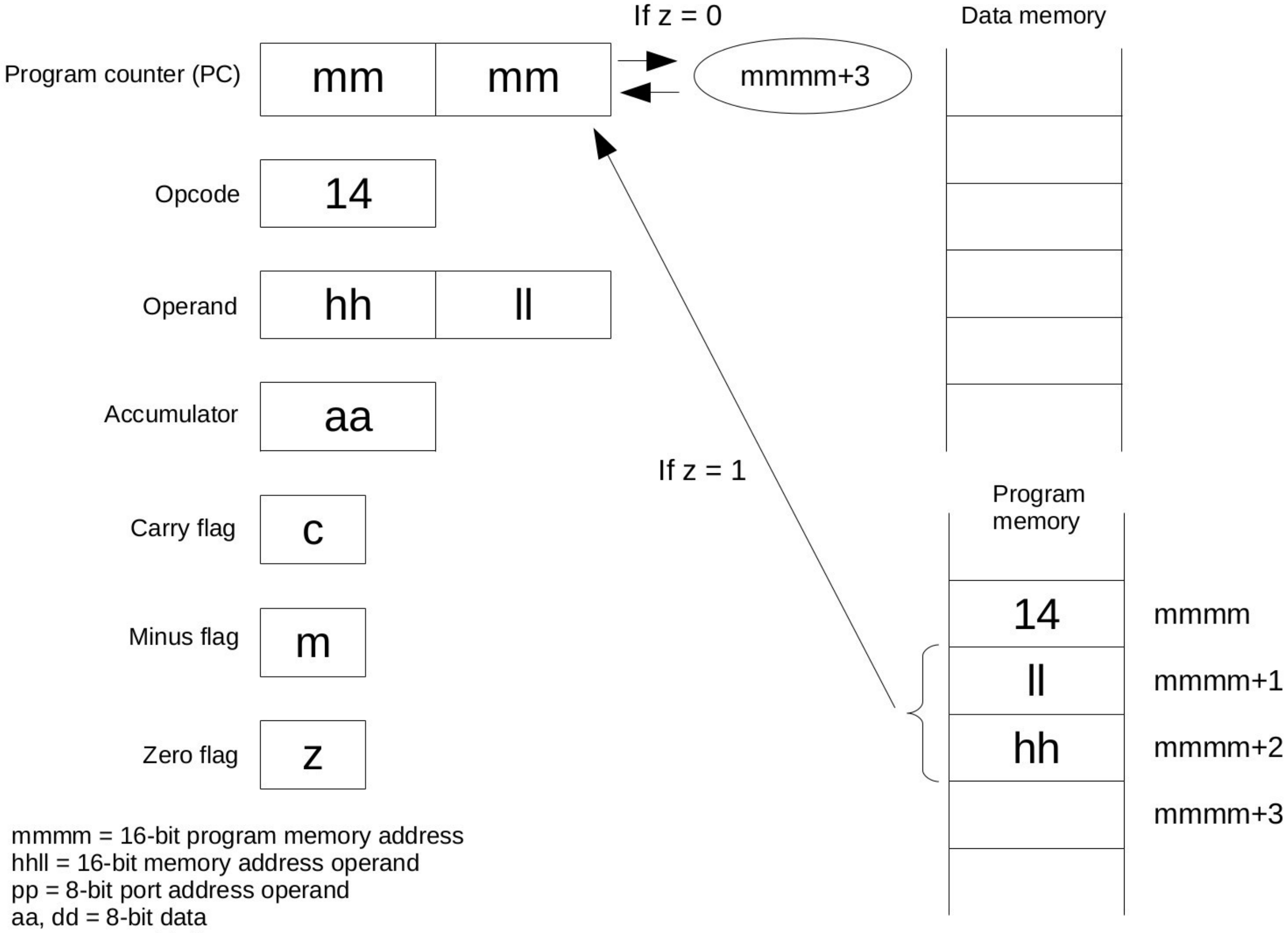
JPM – jump if minus



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction places the 16-bit memory address operand of the instruction into the program counter, causing program flow to jump to the instruction at that address, if the minus flag is 1, otherwise the program counter continues with normal flow at the instruction address + 3. The accumulator and flags are not affected. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

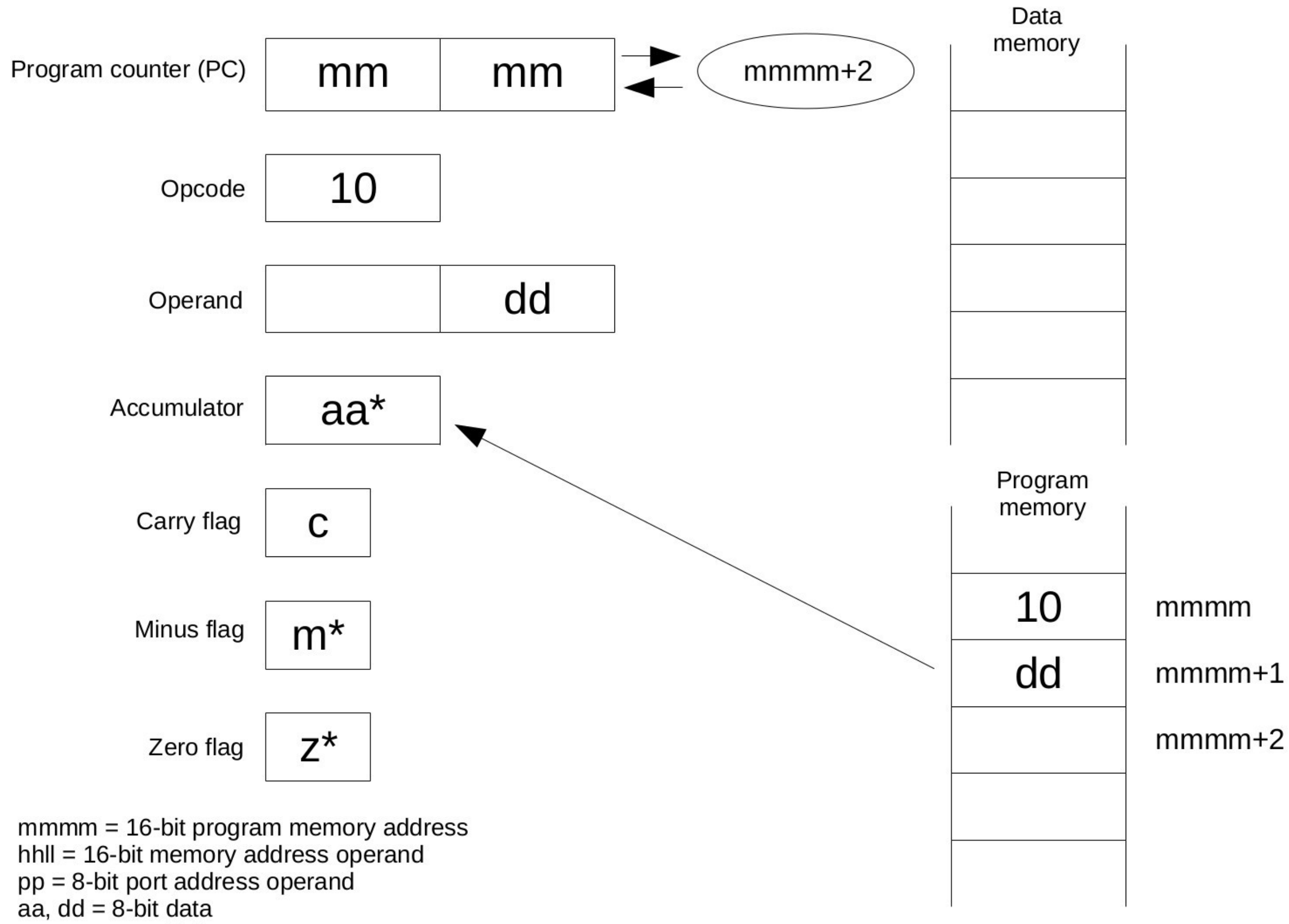
JPZ – jump if zero



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction places the 16-bit memory address operand of the instruction into the program counter, causing program flow to jump to the instruction at that address, if the zero flag is 1, otherwise the program counter continues with normal flow at the instruction address + 3. The accumulator and flags are not affected. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

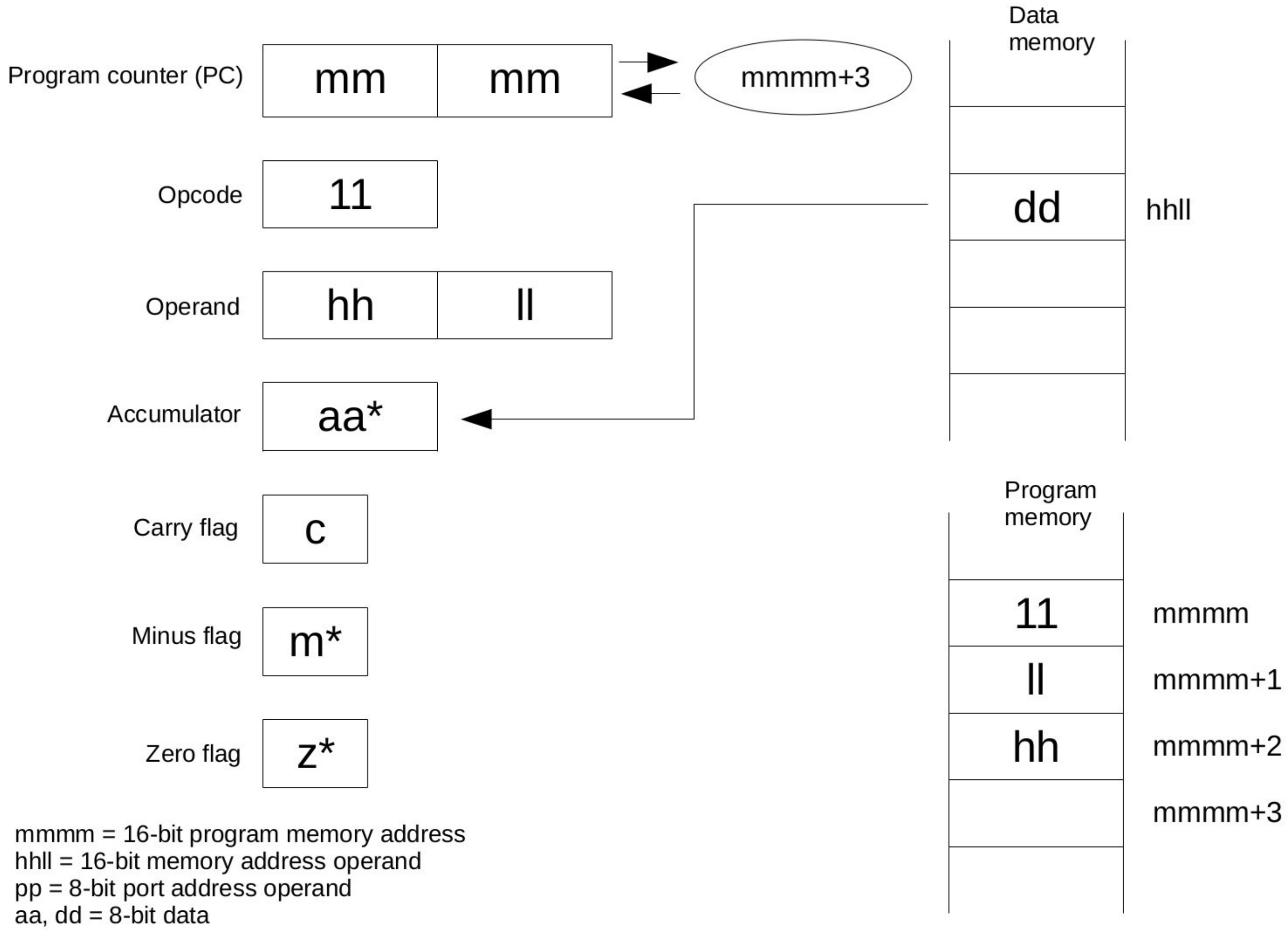
LDI – load accumulator immediate



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction places the one-byte operand of the instruction into the accumulator. The carry flag is not affected, but the minus and zero flags are affected, as they are in all instructions that affect the accumulator.

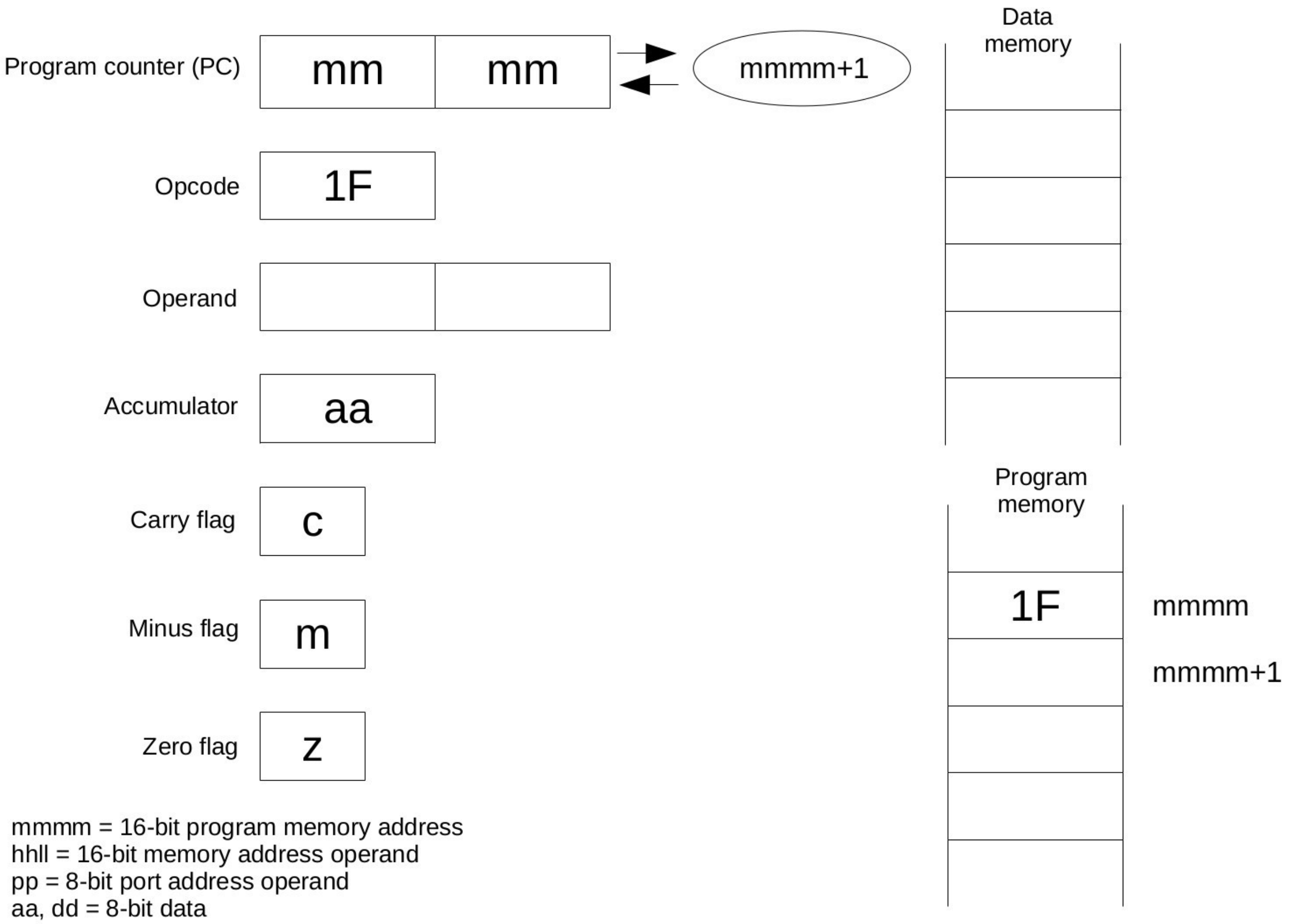
LDM – load accumulator from memory



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction loads the accumulator with the one-byte value contained in the memory address location referenced by the 16-bit address instruction operand. The carry flag is not affected, but the zero and minus flags are affected, as they are in all operations that affect the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

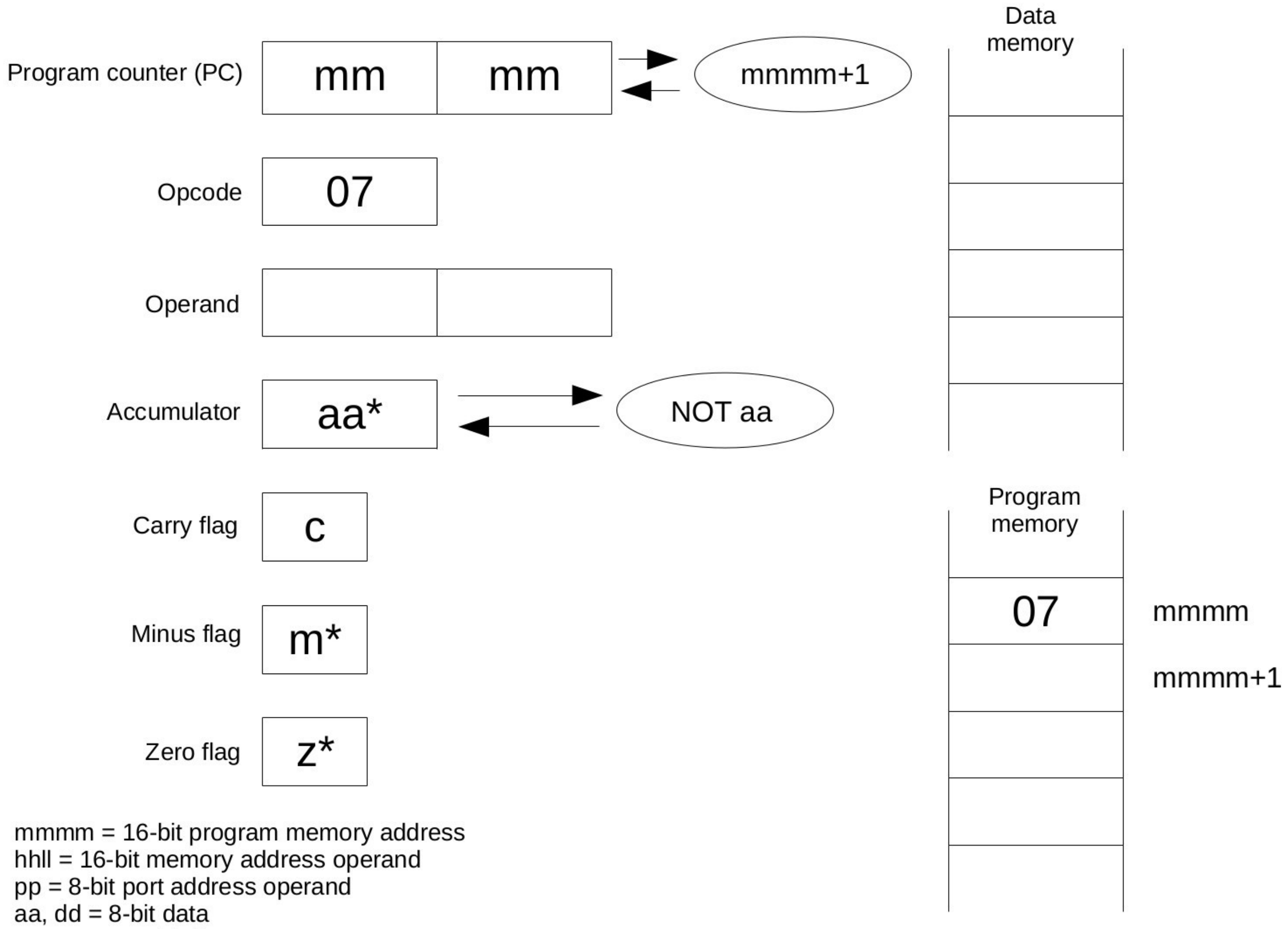
NOP – no operation



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

No operation performed. The program counter is incremented. The accumulator and flags are not affected.

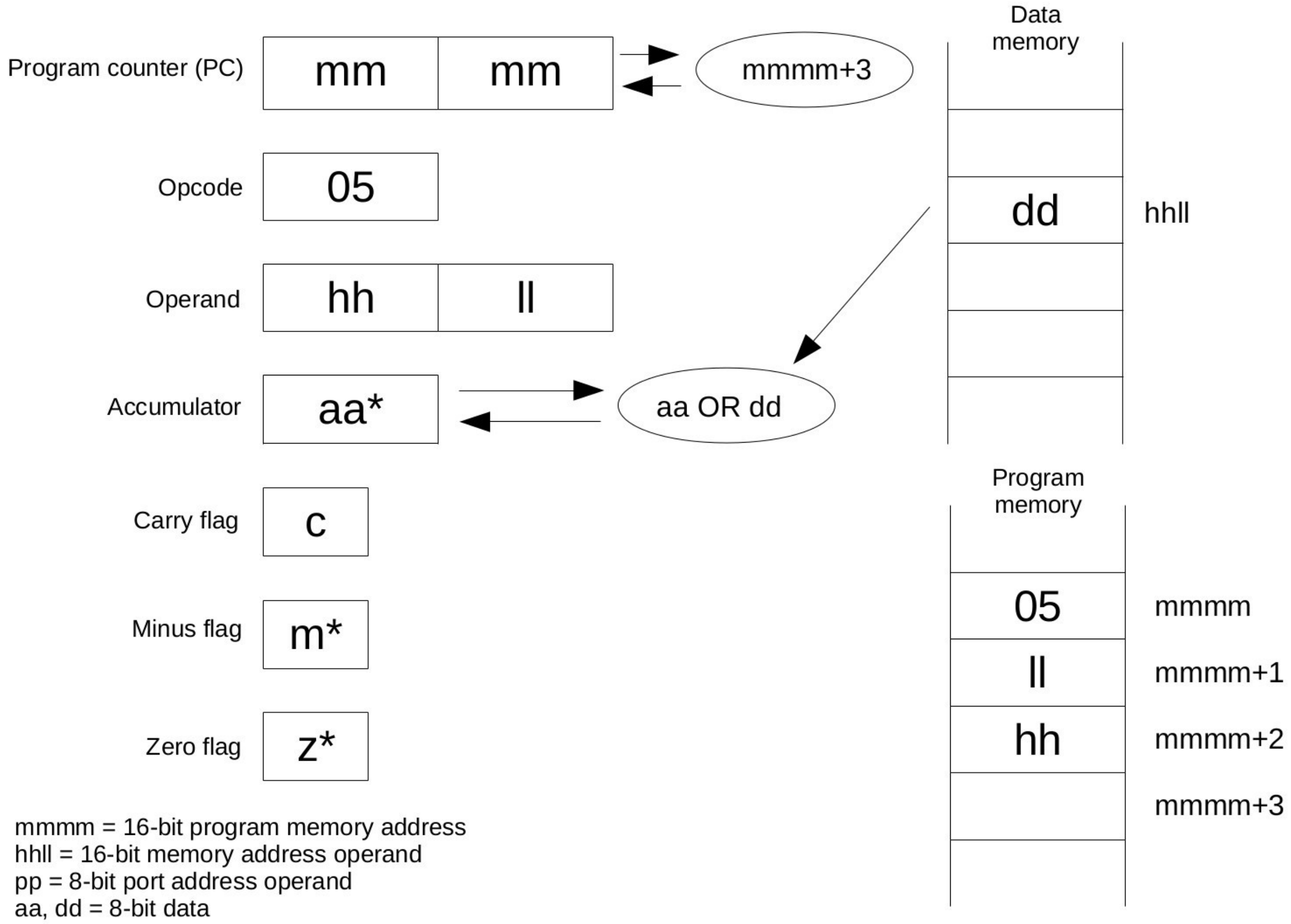
NOT – bitwise invert (ones-complement) accumulator



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction performs a ones-complement (inversion) of the accumulator. For example, if the accumulator contains 0011 1010b, after the operation the accumulator will contain 1100 0101b. The carry flag is not affected, but the zero and minus flags are affected, as they are in all instructions that affect the accumulator.

OR – bitwise logical OR of memory data with accumulator



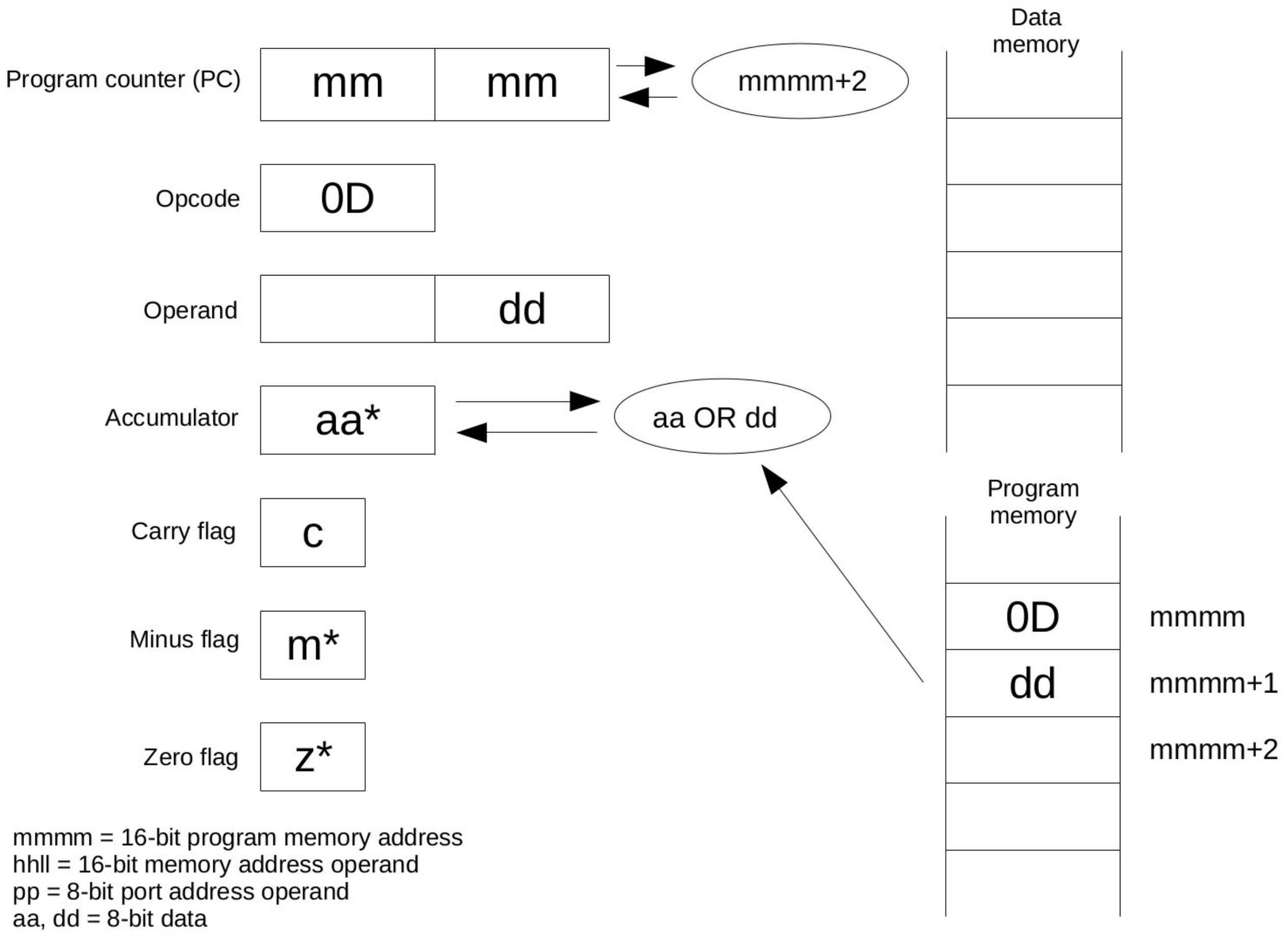
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction performs a bitwise logical OR operation on the accumulator and a byte from memory, placing the result in the accumulator. For example, if dd = 1011 0101b and aa = 1100 0011b:

```
   1011 0101
OR 1100 0011
   1111 0111
```

The value 1111 0111b will be placed in the accumulator. The carry flag is not affected, but the minus and zero flags are affected, as they are in all operations that affect the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

ORIM – bitwise logical OR of immediate data with accumulator



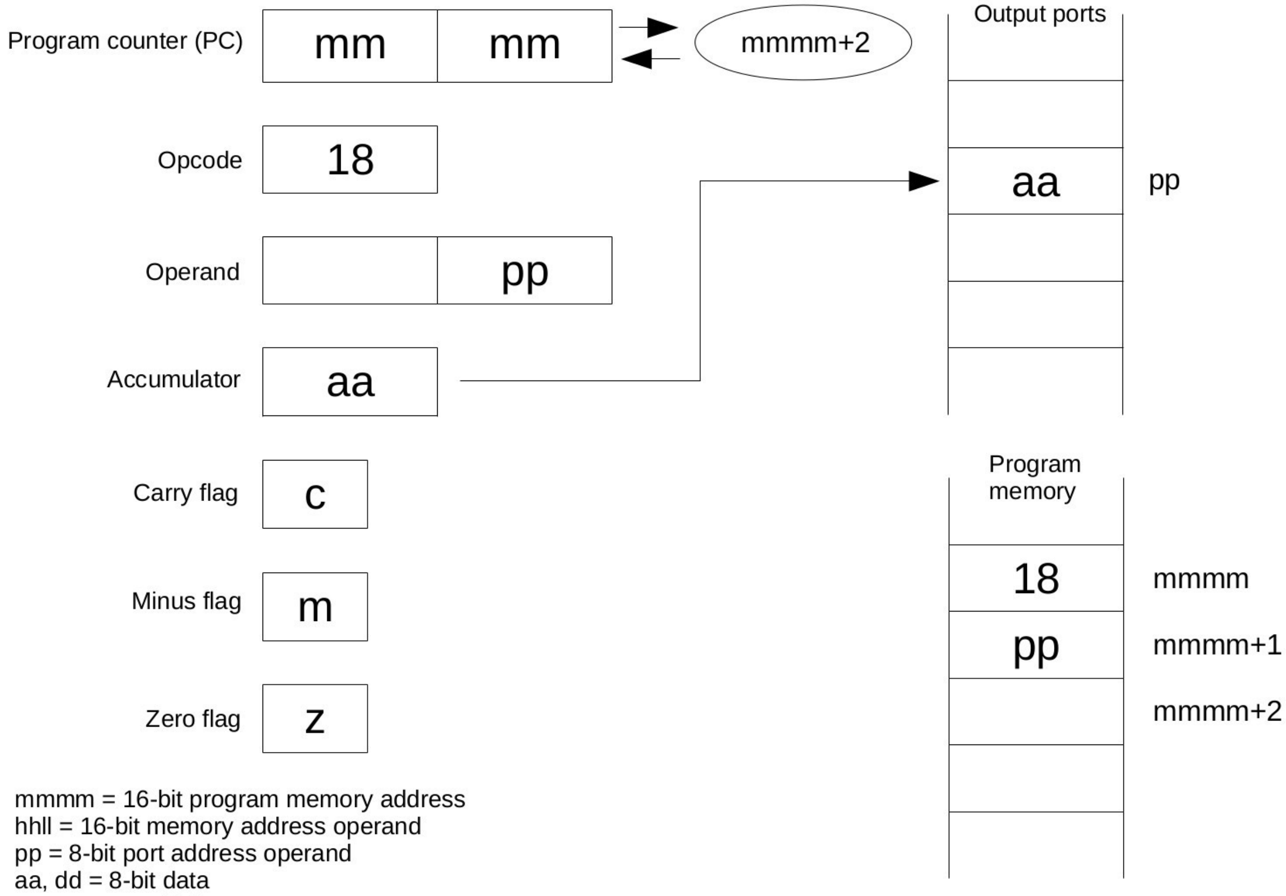
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction performs a bitwise logical OR operation on the accumulator and a one byte instruction operand, placing the result in the accumulator. For example, if dd = 0011 0101b and aa = 1010 0111b:

```
   0011 0101
OR 1010 0111
   1011 0111
```

The value 1011 0111b will be placed in the accumulator. The carry flag is not affected, but the minus and zero flags are affected, as they are in all operations that affect the accumulator.

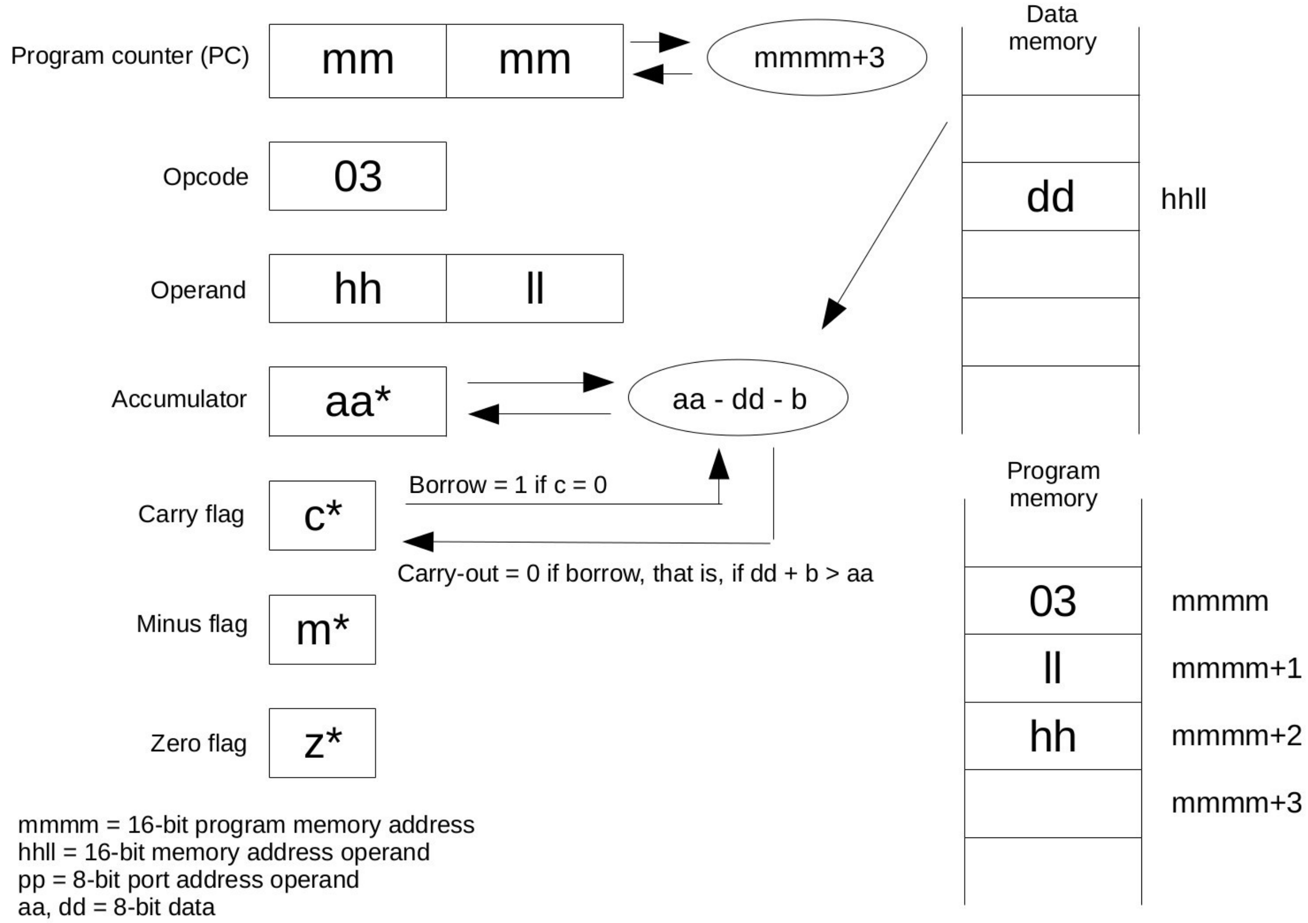
OUT – load output port with accumulator



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

Places the contents of the accumulator into the output port referenced by the 8-bit address in the instruction. The accumulator and flags are not affected.

SBB – subtract memory and borrow from accumulator



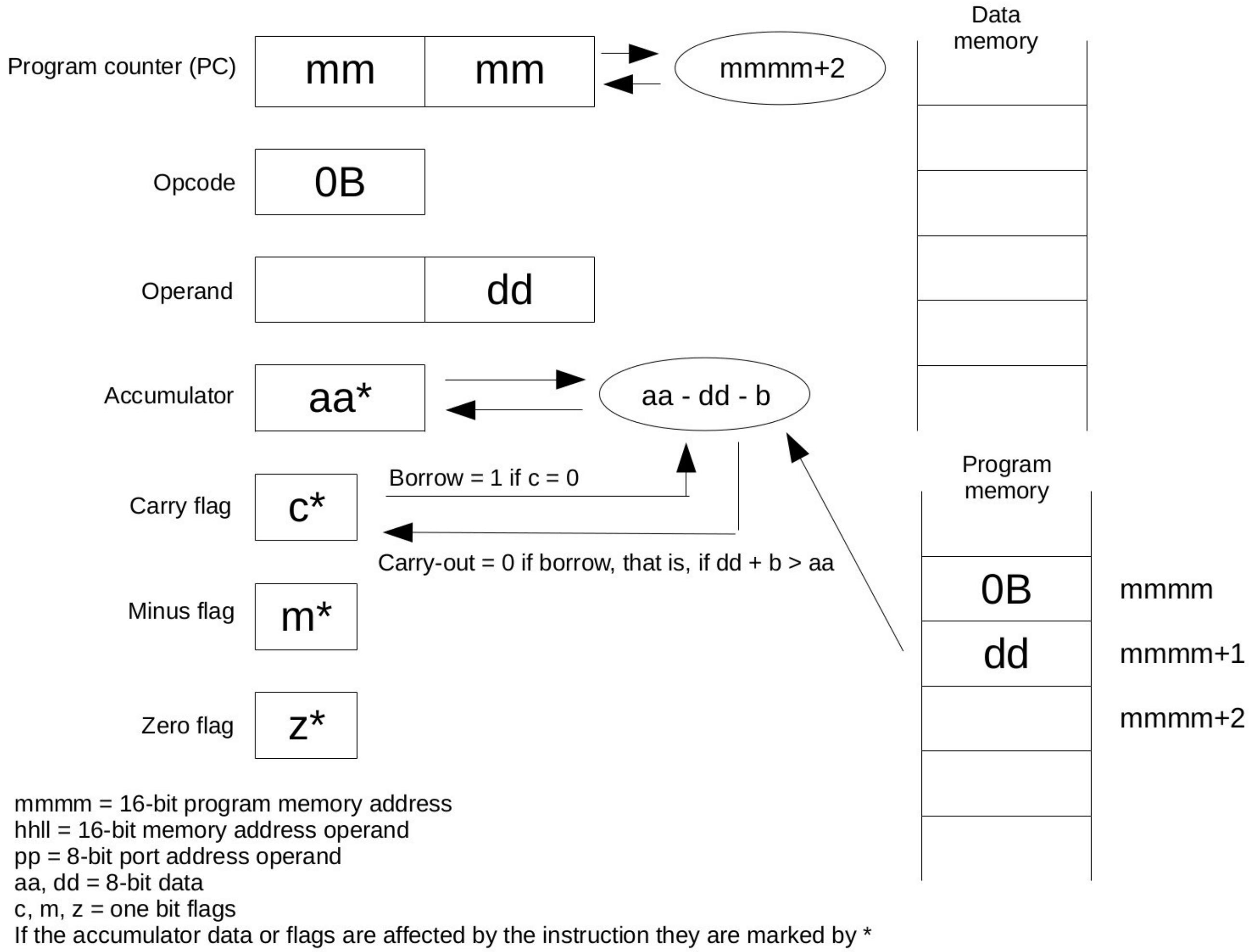
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction subtracts the value of the byte in the memory location referenced by the 16-bit instruction operand and the borrow from the accumulator, and places the result in the accumulator. If the carry flag is zero, the borrow is 1, otherwise the borrow is zero. For example, if aa is 1100 0111b and dd is 1010 1000b, and c = 1, the following operation is performed:

```
dd: 1010 1000
dd + borrow: 1010 1000
one’s complement of dd + borrow: 0101 0111
two’s complement of dd + borrow: 0101 1000
aa + two’s complement of dd + borrow:
  1100 0111
+ 0101 1000
  0001 1111, carry-out = 1 (no borrow)
```

The value 0001 1111b will be placed in the accumulator, and the carry flag will be set to 1. The zero and minus flags will be affected as they are in all instructions that affect the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

SBBIM – subtract immediate data and borrow from accumulator



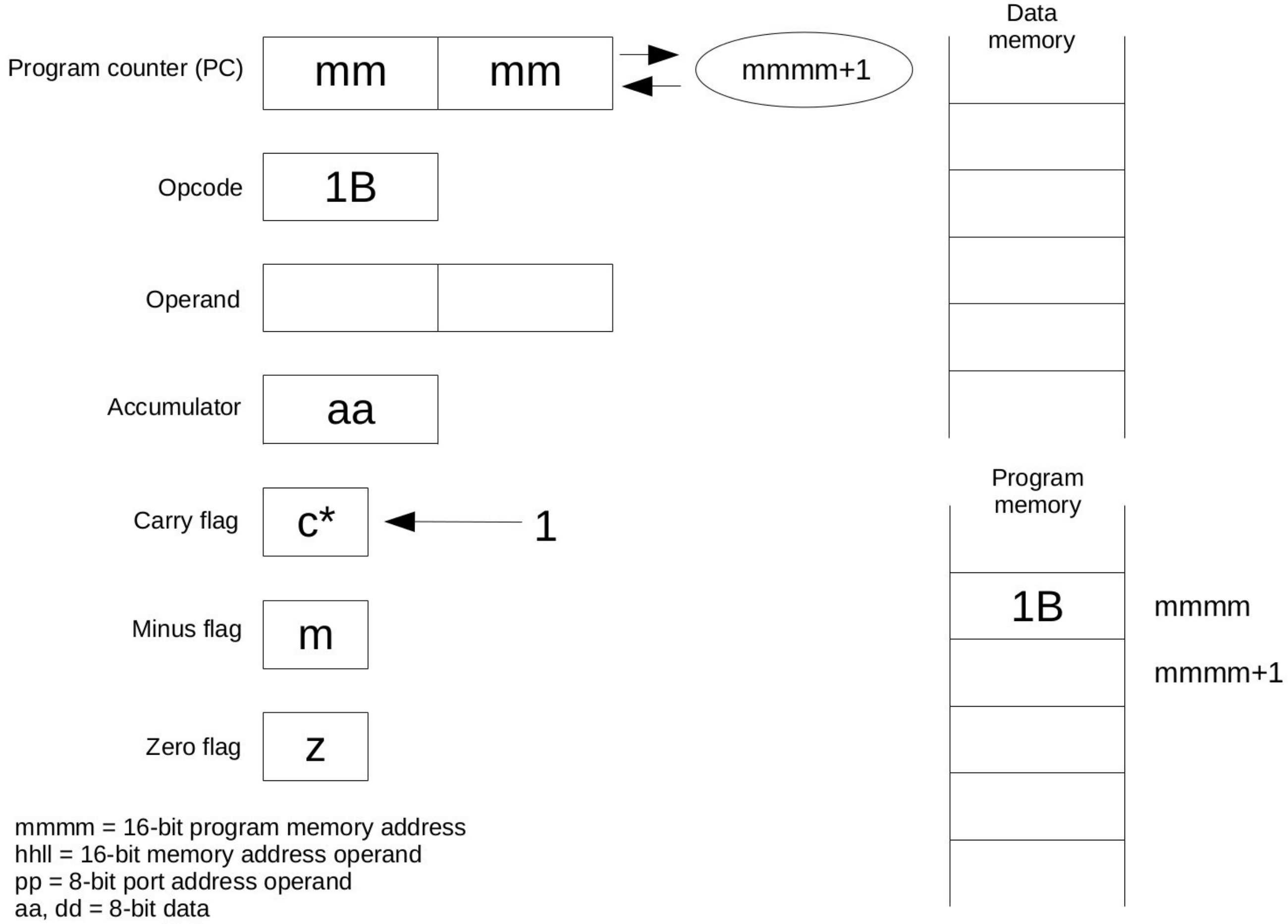
Program counter (PC)
mm
mm
mmmm+2
Data memory
Opcode
0B
Operand
dd
Accumulator
aa*
aa - dd - b
Carry flag
c*
Borrow = 1 if c = 0
Carry-out = 0 if borrow, that is, if dd + b > aa
Program memory
Minus flag
m*
0B
mmmm
dd
mmmm+1
mmmm+2
Zero flag
z*
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction subtracts the value of the byte in the operand of the instruction and the borrow from the accumulator, and places the result in the accumulator. If the carry flag is zero, the borrow is 1, otherwise the borrow is zero. For example, if aa is 0101 0111b and dd is 1010 1000b, and c = 0, the following operation is performed:

```
dd: 1010 1000

dd + borrow: 1010 1001

one’s complement of dd + borrow: 0101 0110

two’s complement of dd + borrow: 0101 0111

aa + two’s complement of dd + borrow:
  0101 0111
+ 0101 0111
  1010 1110, carry-out = 0 (borrow)
```

The value 1010 1110b will be placed in the accumulator, and the carry flag will be set to 0. The minus and zero flags will be affected as they are with all instructions that affect the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

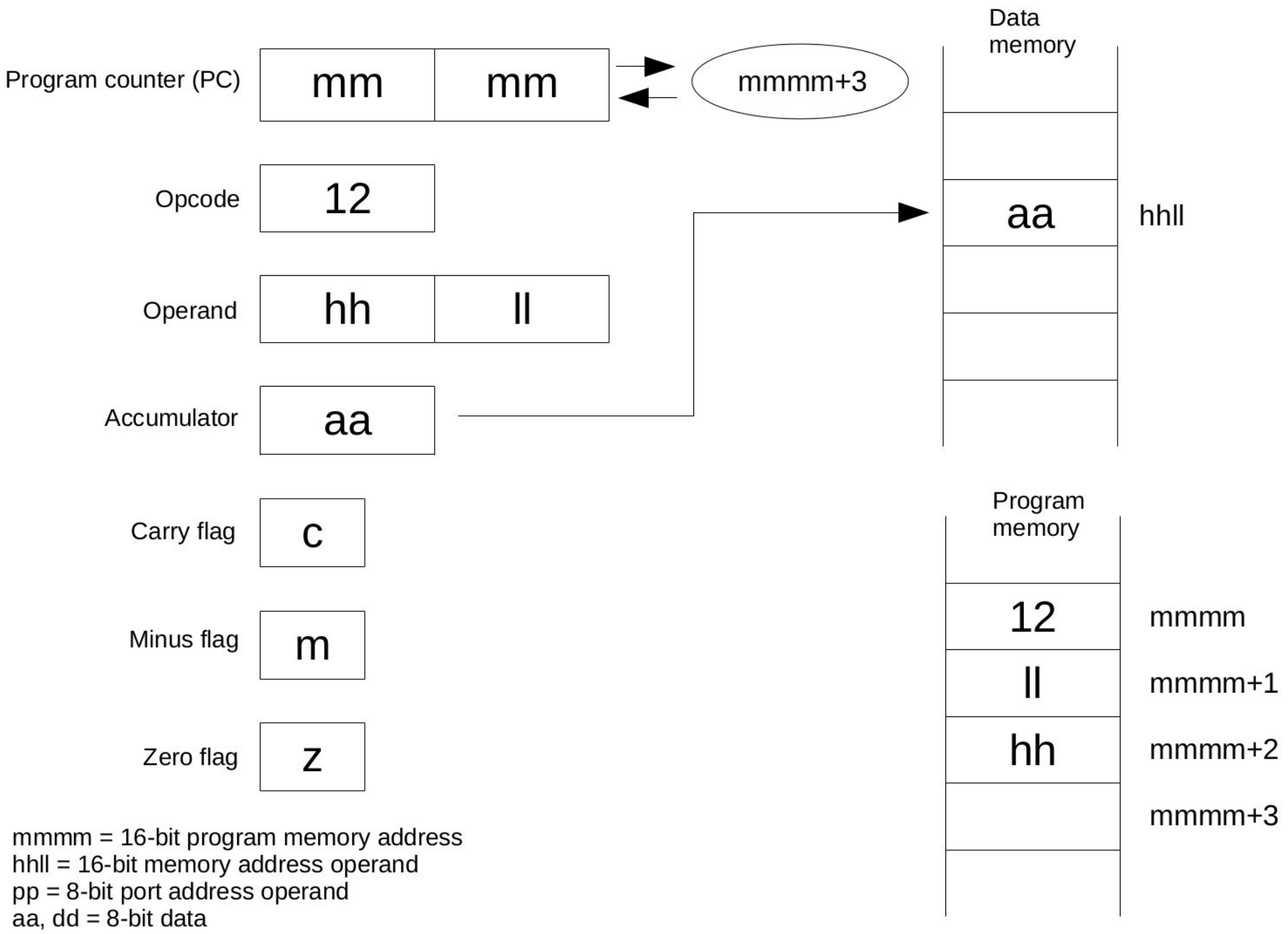
SCF – set carry flag



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction sets the carry flag to 1. The accumulator and the other flags are not affected. It is the complement of the clear carry flag (CCF) instruction.

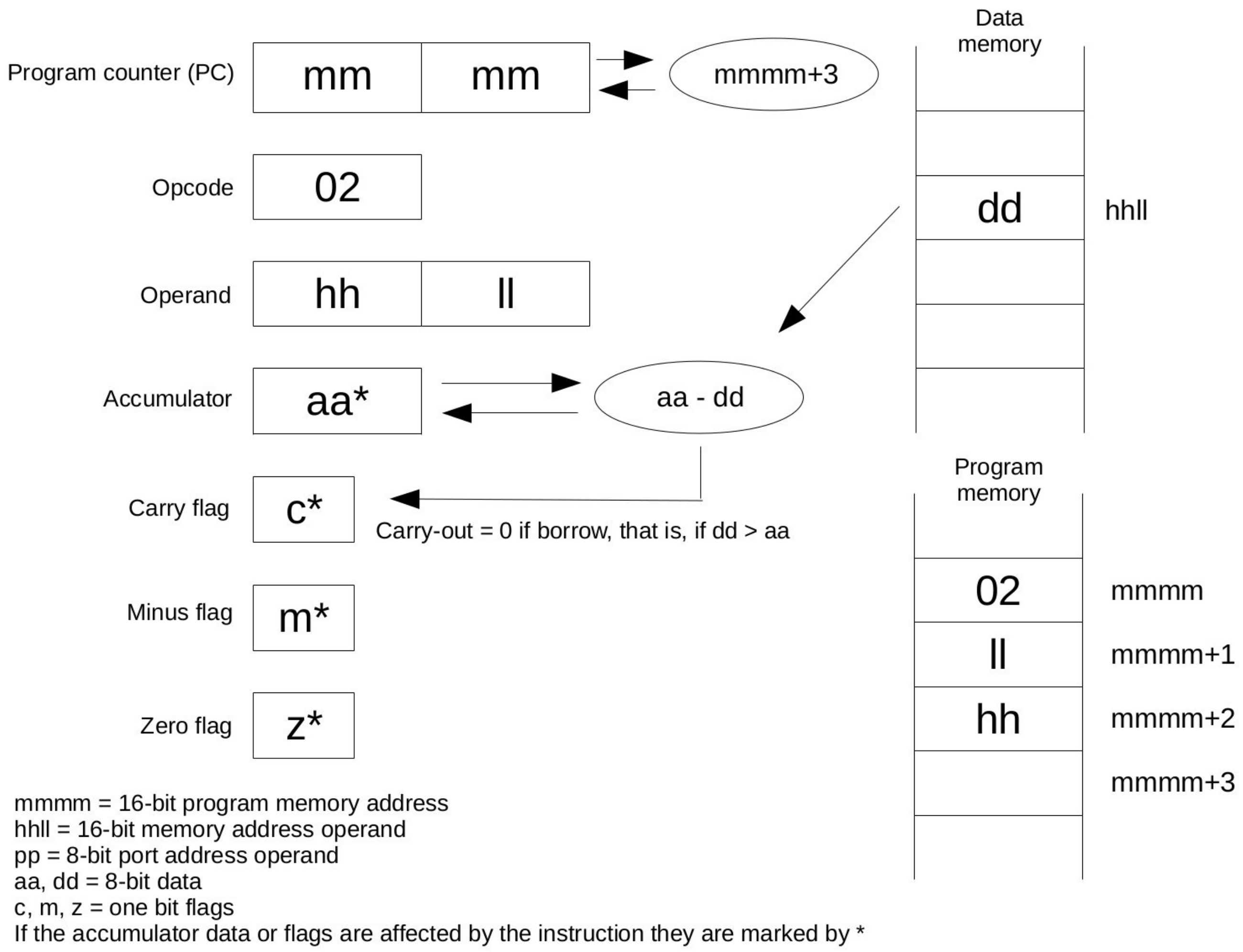
STM – store accumulator to memory



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction stores the contents of the accumulator into the memory location referenced by the 16-bit address operand in the instruction. The accumulator and flags are not affected.

SUB – subtract memory from accumulator



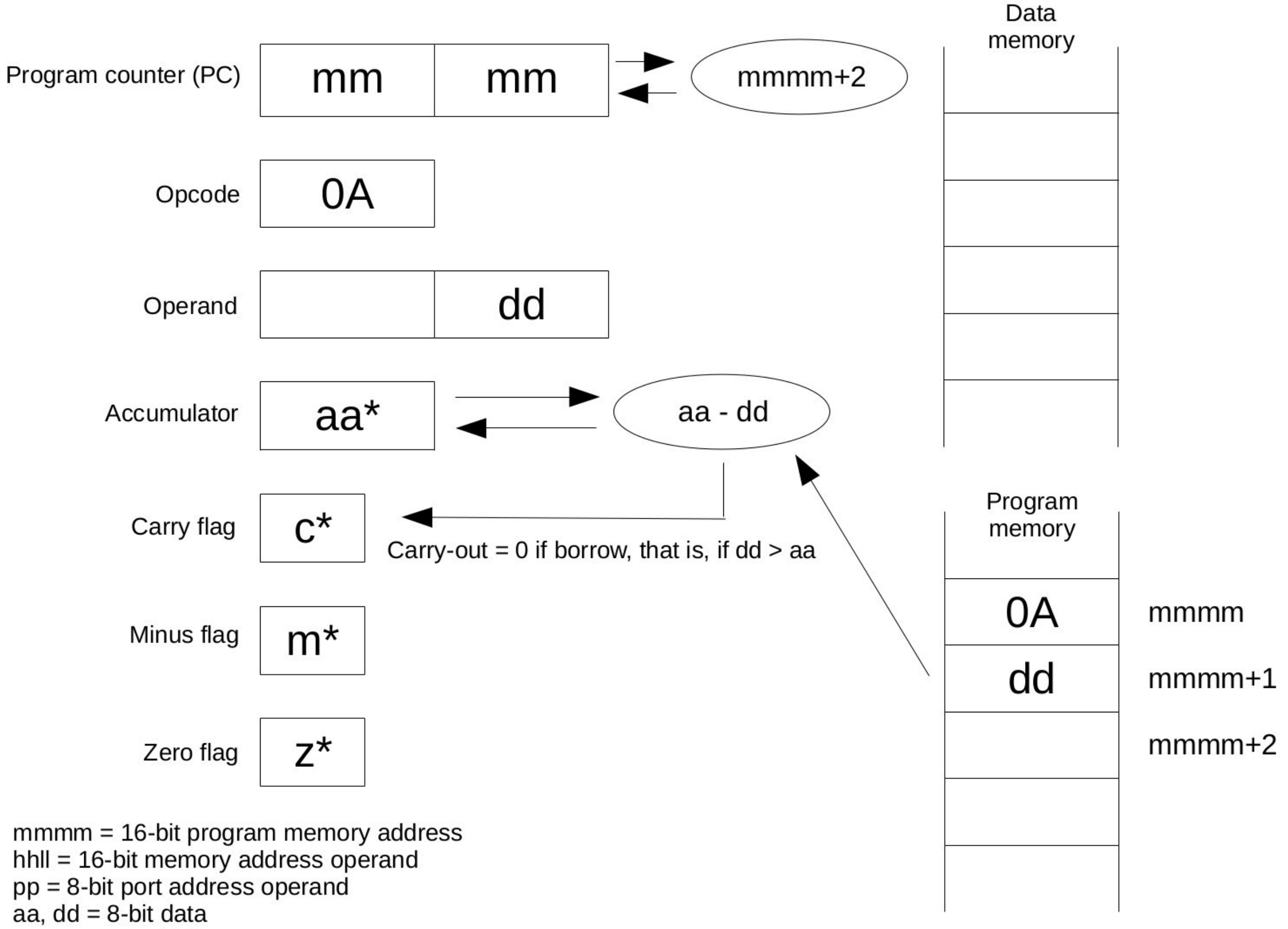
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction subtracts the byte value contained in the memory location referenced by the two-byte instruction operand from the accumulator. For example, if aa is 0101 0111b and dd is 1010 1000b, the following operation is performed:

```
dd: 1010 1000
one’s complement of dd: 0101 0111
two’s complement of dd: 0101 1000
aa + two’s complement of dd:
  0101 0111
+ 0101 1000
  1010 1111, carry-out = 0 (borrow)
```

The value 1010 1111b will be placed in the accumulator, and the carry flag will be set to 0. The minus and zero flags will be affected as they are with all instructions that affect the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

SUBIM – subtract immediate data from accumulator



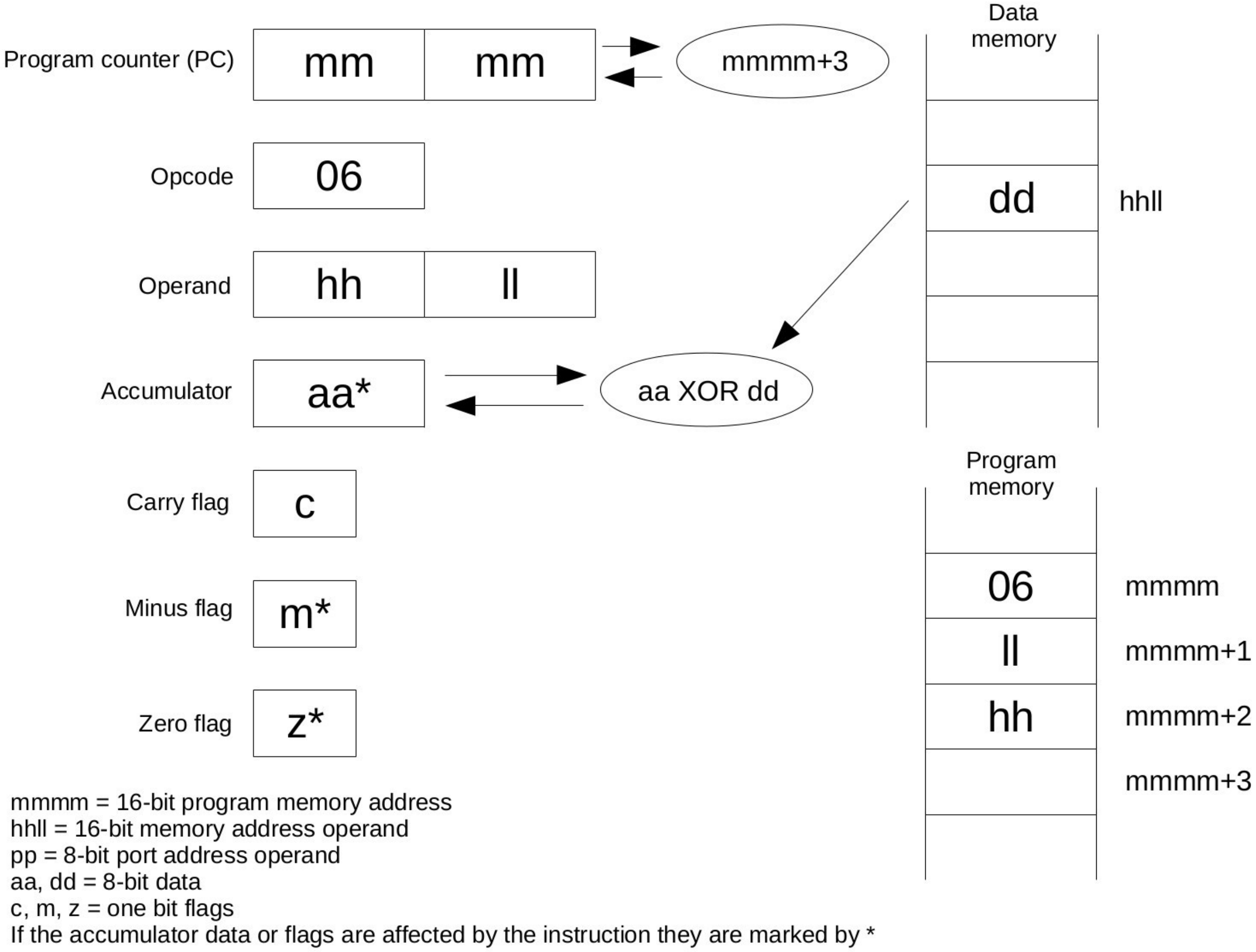
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction subtracts the value of the byte in the operand of the instruction from the accumulator, and places the result in the accumulator. For example, if aa is 0011 0111b and dd is 1011 1001b, the following operation is performed:

```
dd: 1011 1001
one’s complement of dd: 0100 0110
two’s complement of dd: 0100 0111
aa + two’s complement of dd:
  0011 0111
+ 0100 0111
  0111 1110, carry-out = 0 (borrow)
```

The value 0111 1110b will be placed in the accumulator, and the carry flag will be set to 0. Note there is overflow with this operation, since there is a borrow-out but the high-bit of the result is zero. The programmer needs to check for overflow with subtraction operations and deal with it in software, since there is no overflow flag in this processor. The minus and zero flags will be affected by this instruction as they are with all instructions that affect the accumulator.

XOR – bitwise exclusive OR of memory with accumulator



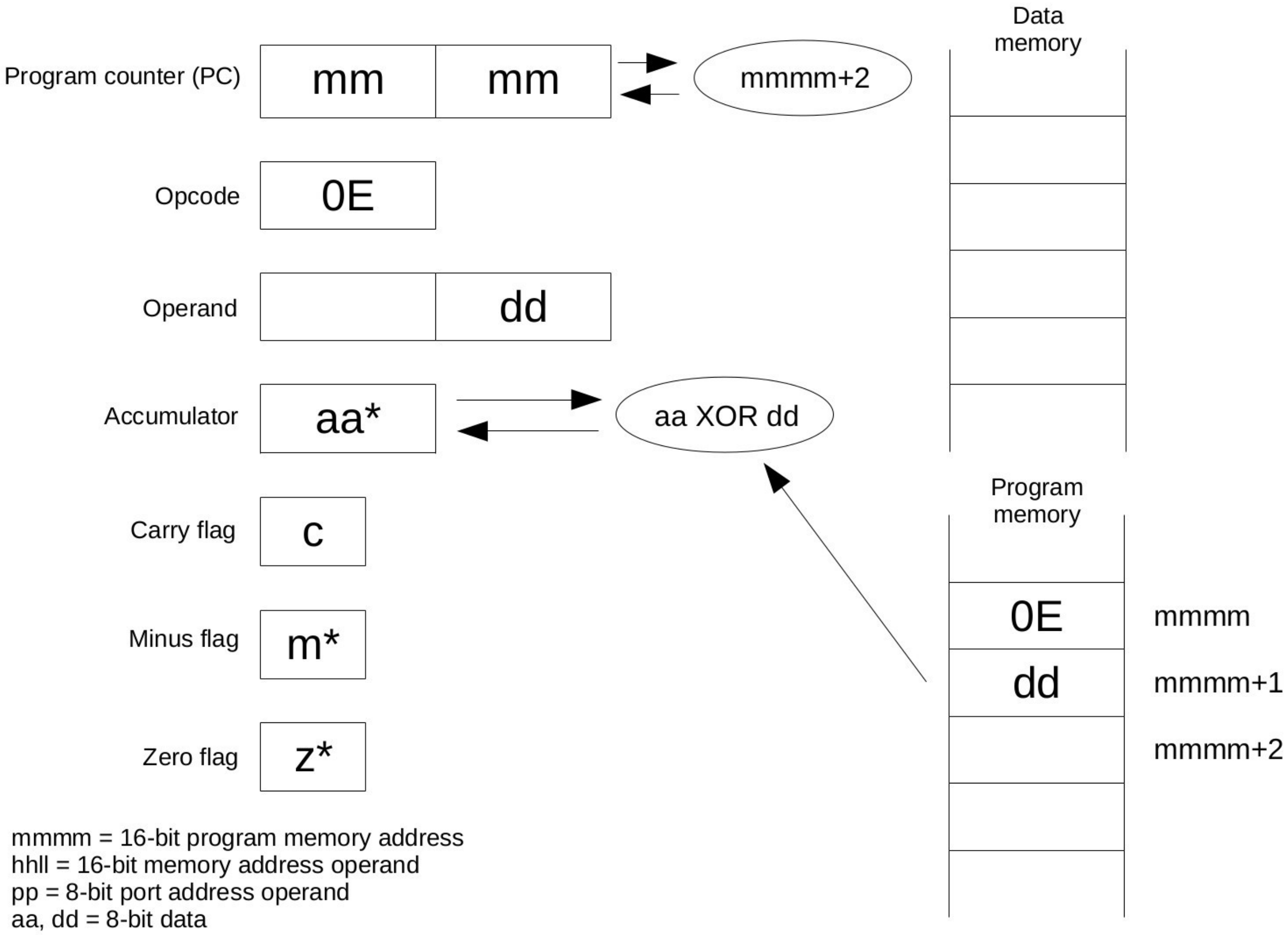
mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction performs a bitwise exclusive-OR operation on the accumulator and byte from the memory, referenced by the 16-bit operand in the instruction. The result is placed into the accumulator. For example, if dd = 1011 0101b and aa = 1100 0011b:

```
    1011 0101
XOR 1100 0011
    0111 0110
```

The value 0111 0110b will be placed in the accumulator. The carry flag is not affected, but the minus and zero flags are affected, as they are in all operations that affect the accumulator. The address operand of the instruction is stored in little-endian fashion, with the least significant byte in the lower memory address location, and the most significant byte in the higher memory address location.

XORIM – bitwise exclusive-OR of immediate data with accumulator



mmmm = 16-bit program memory address
hhll = 16-bit memory address operand
pp = 8-bit port address operand
aa, dd = 8-bit data
c, m, z = one bit flags
If the accumulator data or flags are affected by the instruction they are marked by *

This instruction performs a logical bitwise exclusive-OR operation on the accumulator and a one byte operand from the instruction, placing the result in the accumulator. For example, if dd = 0011 0101b and aa = 1010 0111b:

```
   0011 0101
OR 1010 0111
   1001 0010
```

The value 1001 0010b will be placed in the accumulator. The carry flag is not affected, but the minus and zero flags are affected, as they are in all operations that affect the accumulator.

Using TASM

The TASM assembler program is a flexible macro assembler that runs under 16-bit MS-DOS. It is shareware. The executable file can be obtained from a variety of sites. Here is one:

https://github.com/feilipu/NASCOM_BASIC_4.7/tree/master/TASM31

The TASM manual is on the above site, as well as on this web page:

http://www.cpcalive.com/docs/TASMMAN.HTM

Please note that this is not the Borland Turbo-assembler, also called TASM, but the Telemark Assembler, v. 3.1.

TASM is able to assemble code for the CPUville 8-bit processor. It requires a table file that matches the 8-bit processor assembly language to the opcodes. The table file, TASM08.TAB, is available on the CPUville website.

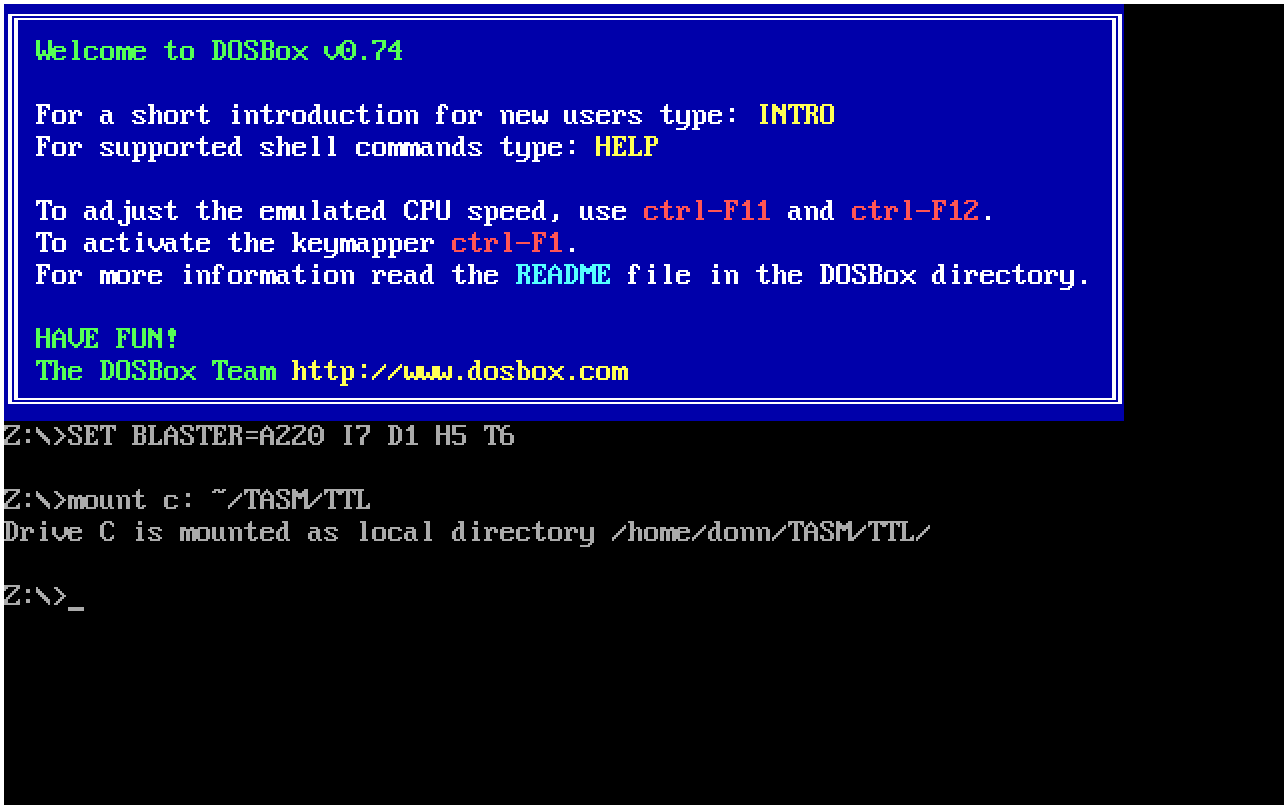
To run TASM, it is simplest to use an MS-DOS emulator. The one I use is DOSBox, available as a free download here:

https://www.dosbox.com/

DOSBox is available for both Windows and Linux.

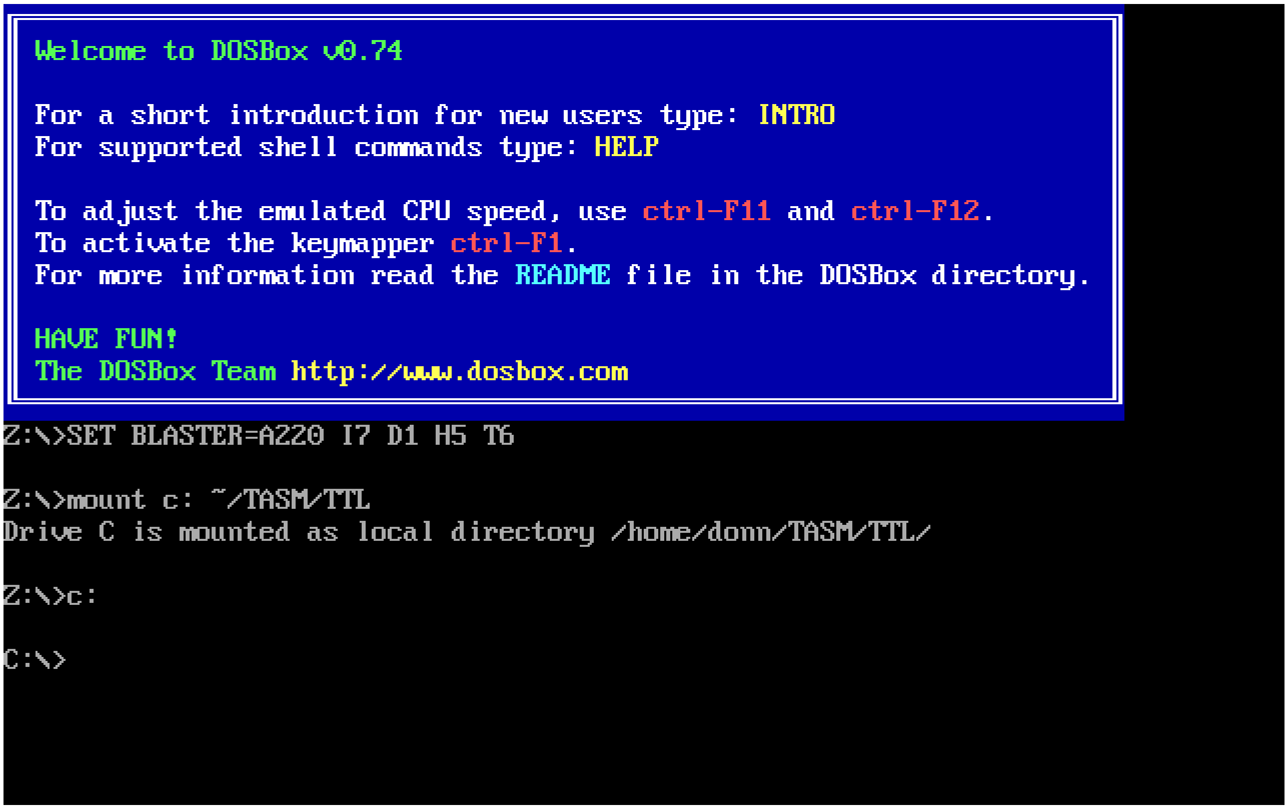
Before you begin, create a folder for TASM assembly projects. Put the TASM.EXE and TASM08.TAB files in the folder. Create your assembly language program as a text file, using whatever text editor you like for your operating system. The example I am using here demonstrates assembly of the PI_9 program. I saved the assembly language file as pi_9.asm in the folder ~/TASM/TTL on my Linux system.

Start DOSBox. The DOSBox window opens. Now, mount your working TASM folder into the DOSBox emulator as the C drive:



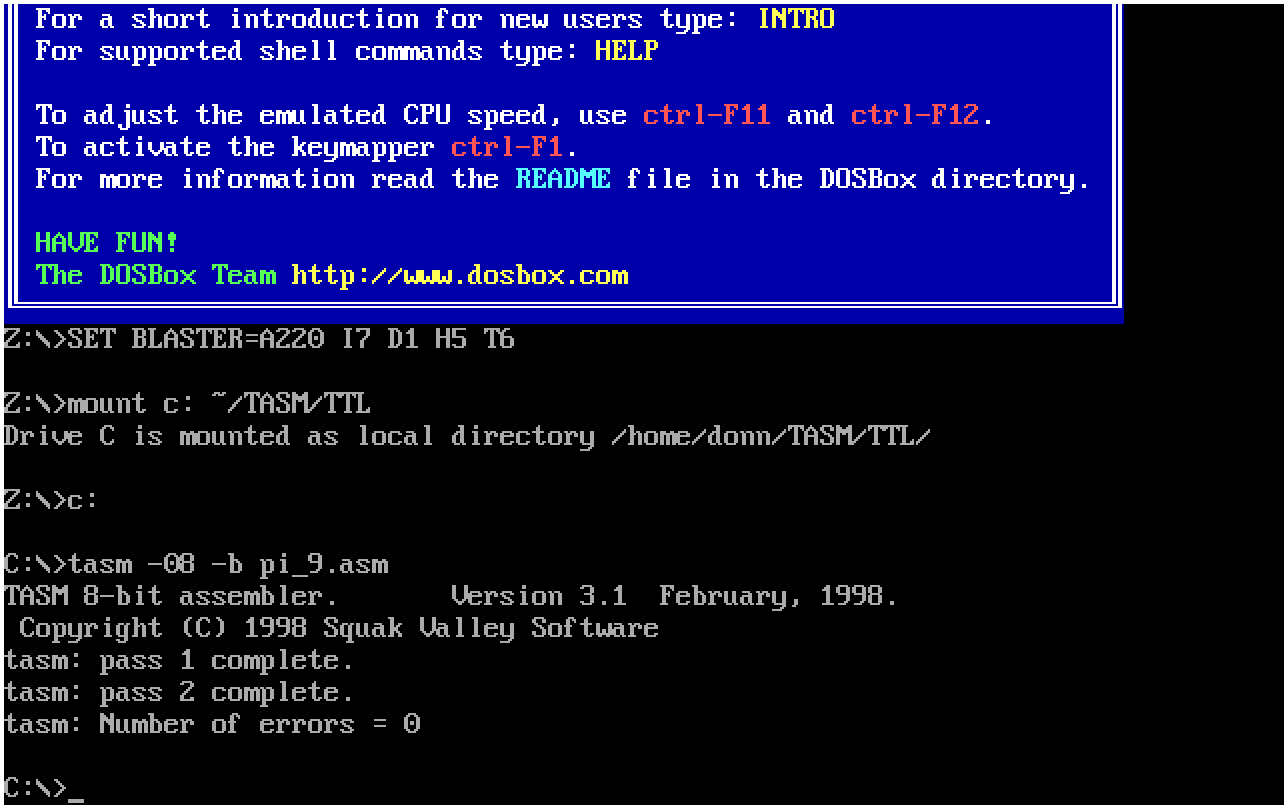
```
Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c: ~/TASM/TTL
Drive C is mounted as local directory /home/donn/TASM/TTL/

Z:\>_
```

Change to the C: drive:



```
Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c: ~/TASM/TTL
Drive C is mounted as local directory /home/donn/TASM/TTL/

Z:\>c:

C:\>
```

Invoke TASM and assemble the program file using the command `tasm -08 -b pi_9.asm.` The command line option `-08` indicates which table file to use, the “08” coming from the table file name. The `-b` option causes the assembler to create a binary object code file (its default is Intel Hex).



```
 For a short introduction for new users type: INTRO
 For supported shell commands type: HELP

 To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
 To activate the keymapper ctrl-F1.
 For more information read the README file in the DOSBox directory.

 HAVE FUN!
 The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c: ~/TASM/TTL
Drive C is mounted as local directory /home/donn/TASM/TTL/

Z:\>c:

C:\>tasm -08 -b pi_9.asm
TASM 8-bit assembler.       Version 3.1  February, 1998.
 Copyright (C) 1998 Squak Valley Software
tasm: pass 1 complete.
tasm: pass 2 complete.
tasm: Number of errors = 0

C:\>_
```

If there are syntax errors, the assembler will list them. Keep file names short, and without spaces in the original MS-DOS style to prevent headaches.

If the assembly completes without errors, the assembler will create two files, PI_9.LST and PI_9.OBJ. The object file will be in binary format because of the `-b` option in the TASM command.

When you are finished, type `exit` on the DOSBox command line.

For examples of assembly language files that will assemble with TASM, see the “Selected Program Listings” section of this manual, and the program listings on the CPUville website. Note that TASM can take assembly language mnemonics in upper or lower case, but labels are case-sensitive. Labels are separated from the assembly language statements by white space, with an optional colon character after the label. Assembler directives (pseudo-operations) are preceded by the period character. Other characteristics of TASM are discussed in the TASM manual.

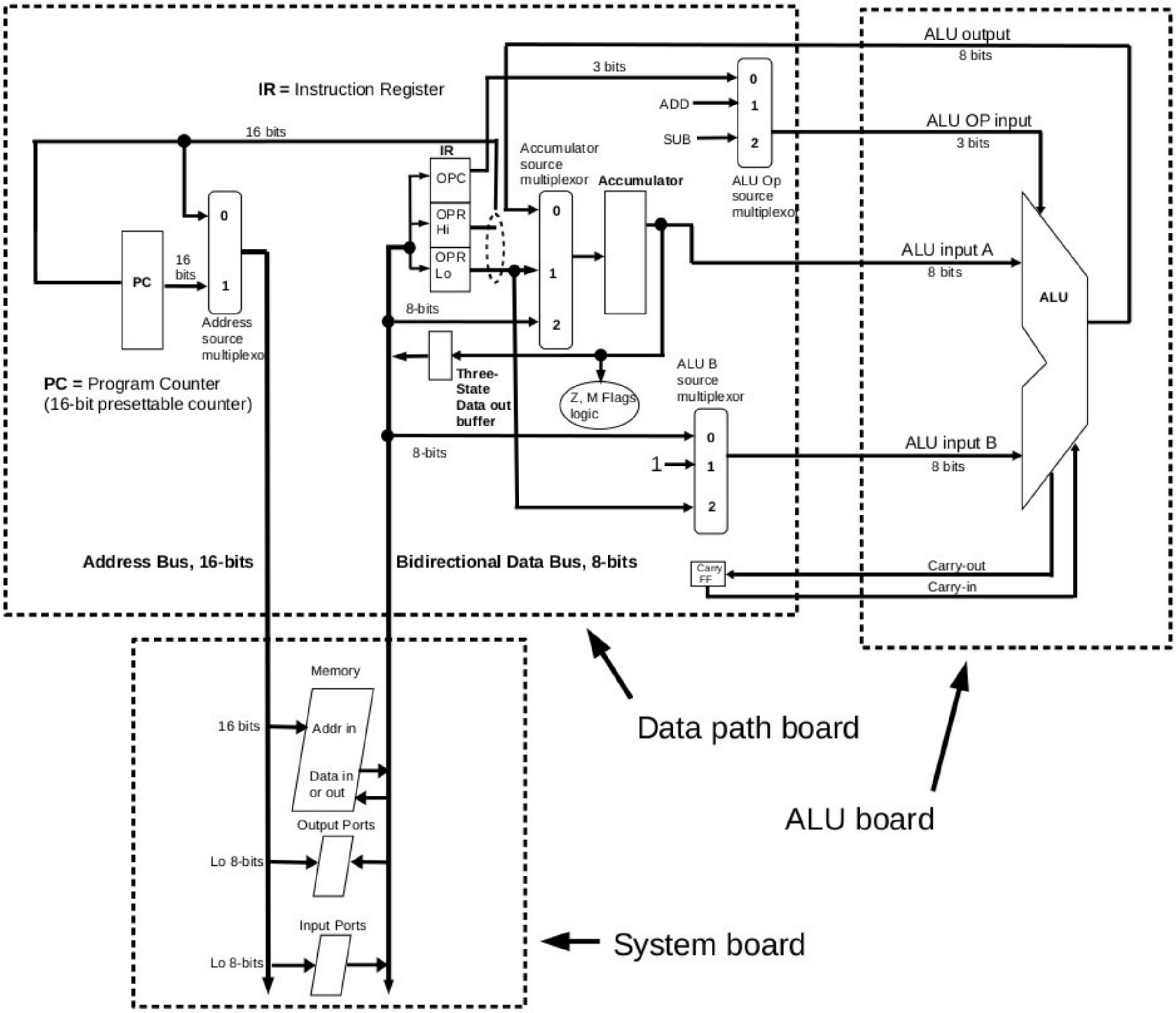
TASM is a macro-assembler, and the use of macros is a great aid in programming for the 8-bit processor. Using assembler macros and the processor’s simple instruction set the programmer can create code that overcomes many of the processor’s limitations. See the section “Special Programming Techniques” in the Appendix for examples.

Schematics and Explanations

This section is an attempt to explain how the processor is made. It has block and schematic diagrams of the parts of the processor, and explains the functional sections. The whole schematic of each board is too large to be shown accurately in this document, so only selected portions are shown and discussed. The map images in this section are only to show where on the larger schematic the circuit sections are taken from; the underlying schematic in these map images is of low resolution. The high-resolution schematic images can be downloaded from the CPUville website.

I show here the inter-board connectors and the active circuit elements of each schematic. The connections between the circuit elements are not shown, but of course they are of vital importance. Sometimes the connections can be inferred from the pin labels on the integrated circuit symbols. If you want to study how the circuit elements connect to each other in detail I recommend downloading the high-resolution schematics from the CPUville website.

Overall design of the processor data path



This diagram shows the elements of the processor, and the computer system, that deal with the flow of data.

The data path board is the main board of the processor. It has on it the registers that hold important data, and multiplexers (data selectors) that guide the flow of data though the processor.

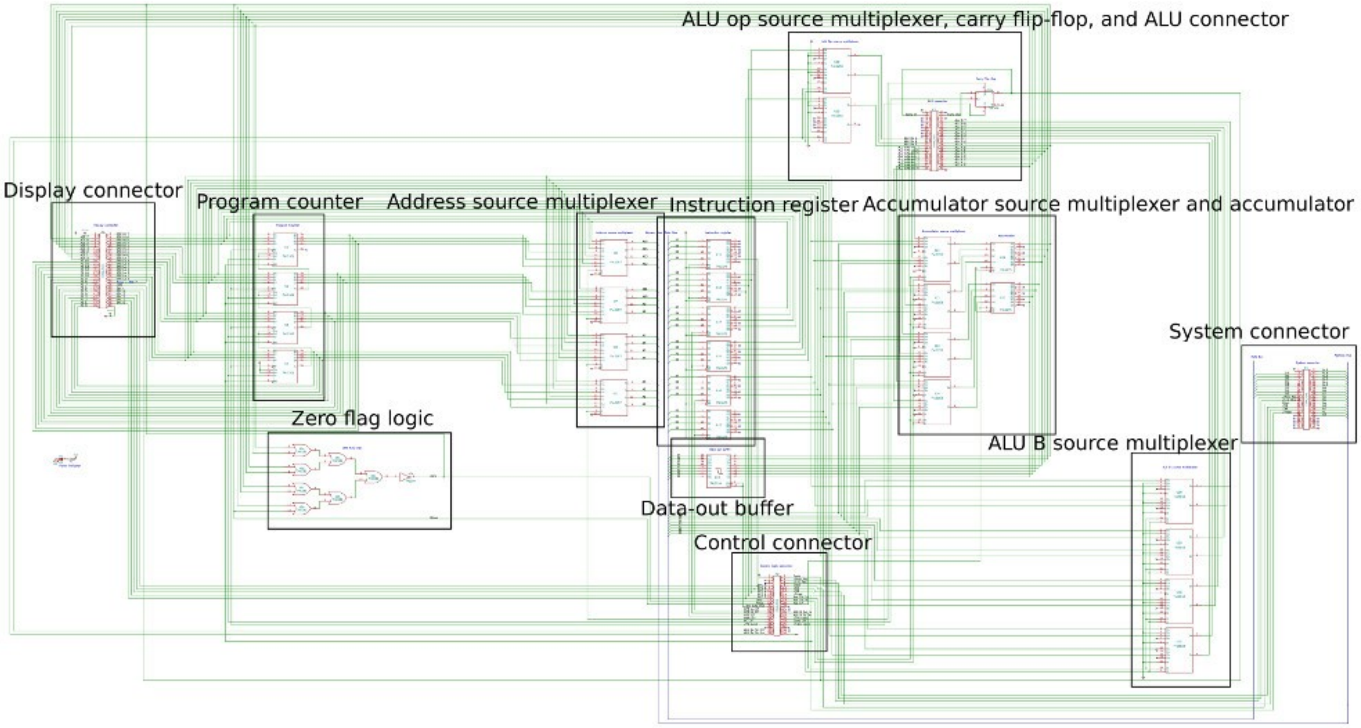
The ALU board is shown here as plain ALU symbol, with the inputs and outputs indicated. The ALU is a piece of combinational logic, that is, it does not store any data or have any state. When the inputs change, the outputs change after a brief delay. A given input always produces the same output. For detail on how the ALU works, see the ALU schematics section below.

The system board has the computer memory (ROM and RAM) and the input output ports. This may be the Original Z80 computer or Single-board Z80 computer configured to run with the 8-bit processor, or your own system board.

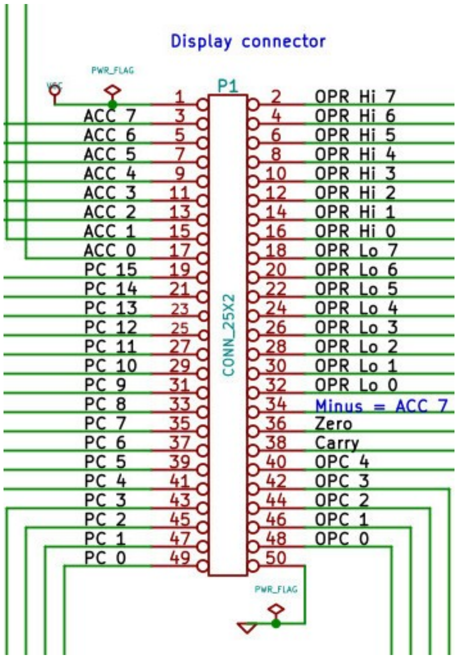
Not shown in the diagram above is the control logic board. The control board takes as input the current instruction and the flags, and produces outputs that determine the settings of the multiplexers on the main board and which registers will be written, in order to perform the instruction. The control logic also produces the read and write signals for the system memory and ports. Unlike the ALU, the control logic “has state”. That is, for a given set of inputs, the outputs are determined by the current state, stored in a register on the control board. The next state to be performed is determined by the next-state logic on the control board, which takes as input the current state, the instruction, and the flags. There is a detailed explanation of how the next-state logic works in the Appendix. The next-state becomes the current state on each upgoing clock edge. The control logic will run through a series of states unique to each instruction, which causes the data path and system to perform the instruction. The control board of this processor also has clock oscillators and a reset circuit for convenience; these circuits are not usually thought of as elements of processor control, but rather of the overall system.

Main board (data path) schematics

Below is a map image that shows where the following schematics are taken from. The schematic in the map image is low resolution; a high-resolution schematic can be obtained from the CPUville website.

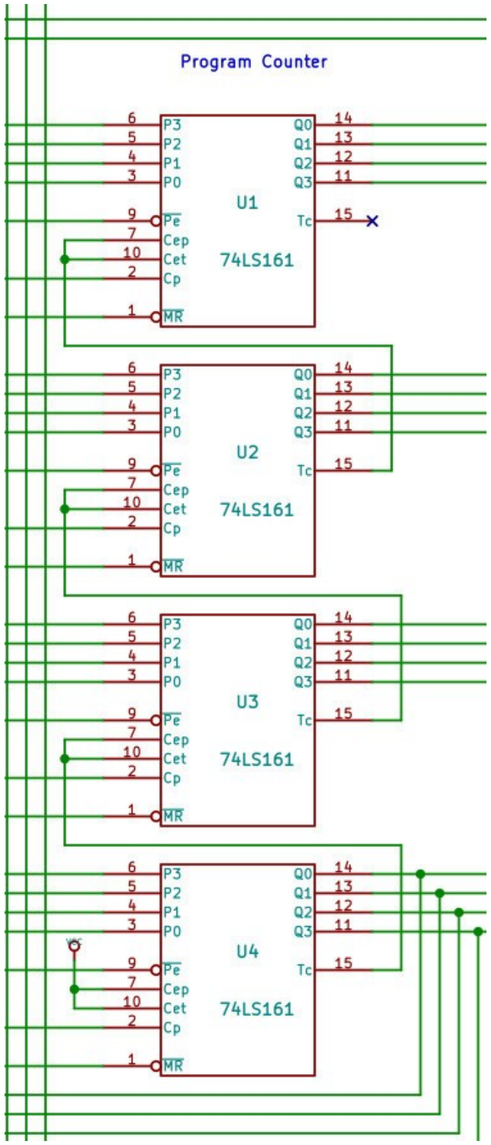


Display connector



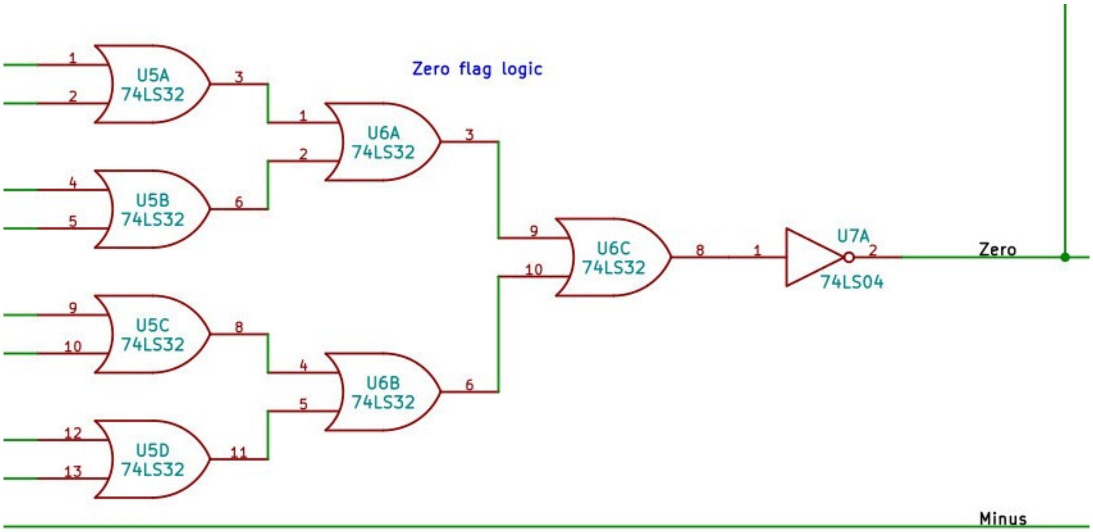
The display connector has outputs of the main board registers and flags for passing to a register display board, which is an accessory.

Program counter



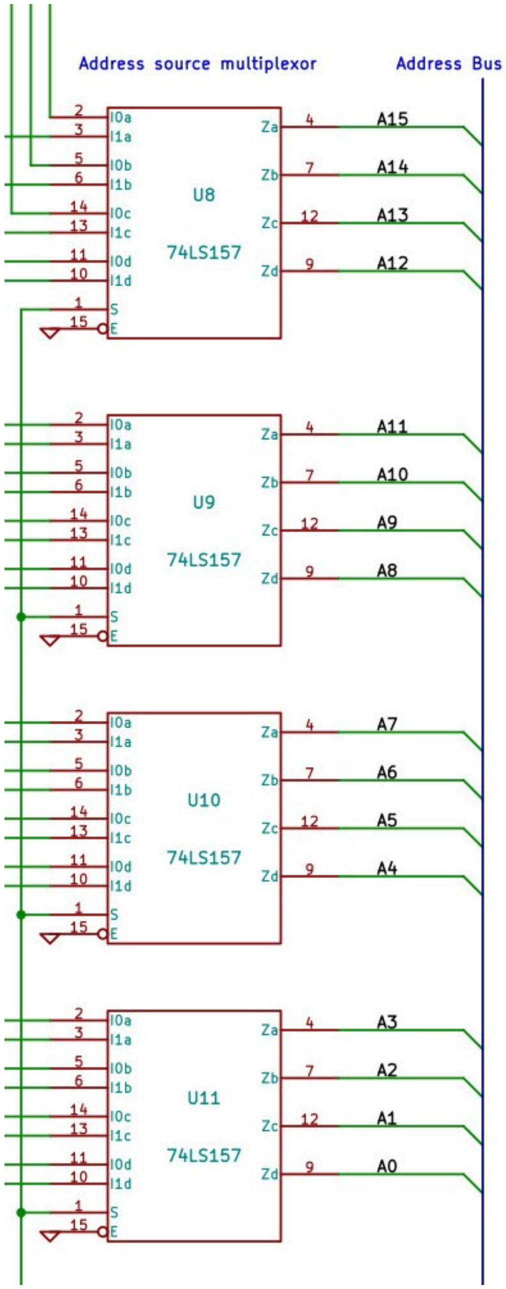
The program counter is a 16-bit, presettable, clearable binary counter. It contains the address of the instruction to be fetched. It is made from four 4-bit synchronous counter ICs. Synchronous counters will follow a counting sequence based on internal logic, rather than propagation from one bit to the next, or one counter to the next, so there is no propagation delay. In fact, you can see that the clock pulse is fed to each 4-bit counter. When the control logic determines that the address of the PC needs to incremented, it creates a clock pulse. When the processor is reset, the master reset signal is asserted, and the PC is forced to zero (cleared). This is why the processor starts execution at address location 0x0000 after coming out of reset. A jump instruction will use the preset control input /Pe to load an address into the PC over the counter inputs P0 to P3.

Zero flag logic



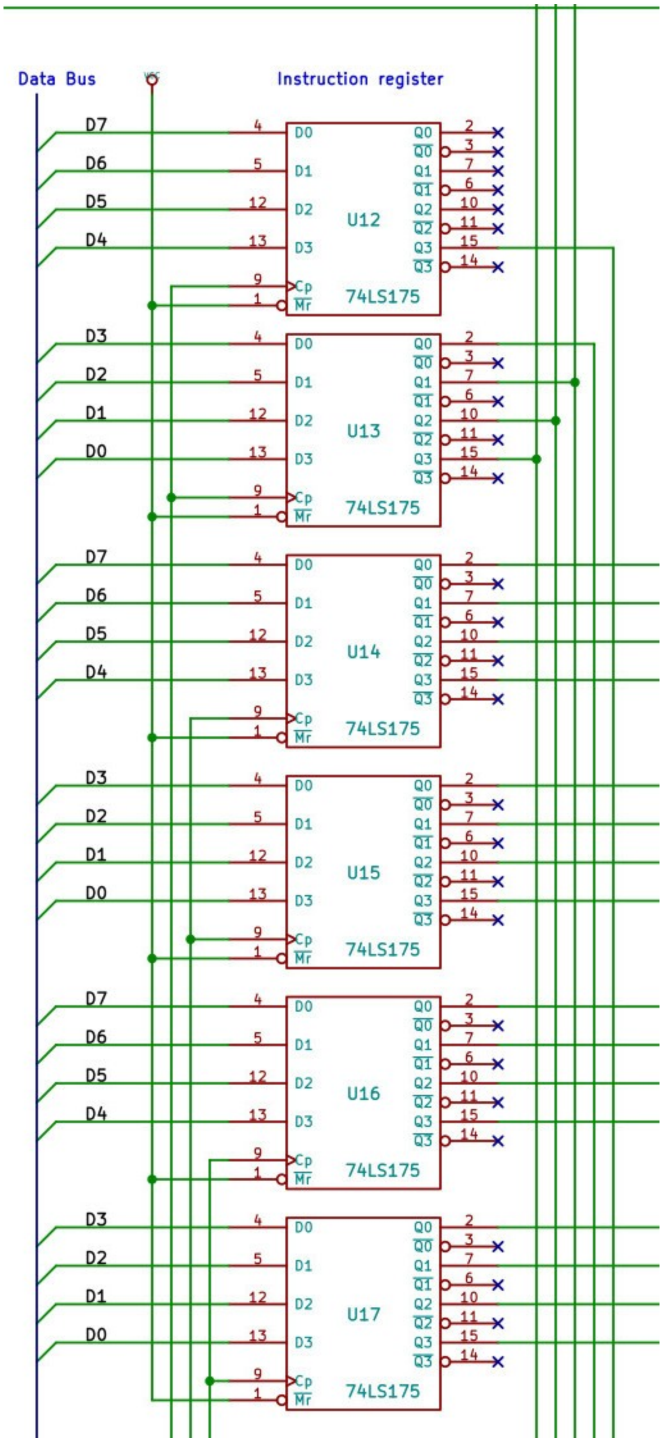
The zero flag logic is an 8-input NOR operation. Each bit of the accumulator value is an input. When all the bits are zero, the final output is one.

Address source multiplexer



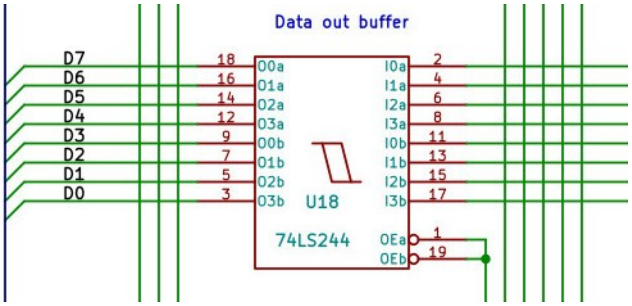
This multiplexer determines which processor address to place on the system address bus. There are only two address sources in the processor, the program counter (PC) and the instruction operand register (OPR Hi and Lo, combined as a 16-bit address value). The one-bit select input (S) determines which of the two addresses will appear on the multiplexer outputs. The control line for this select bit is Addr Src (address source).

Instruction register



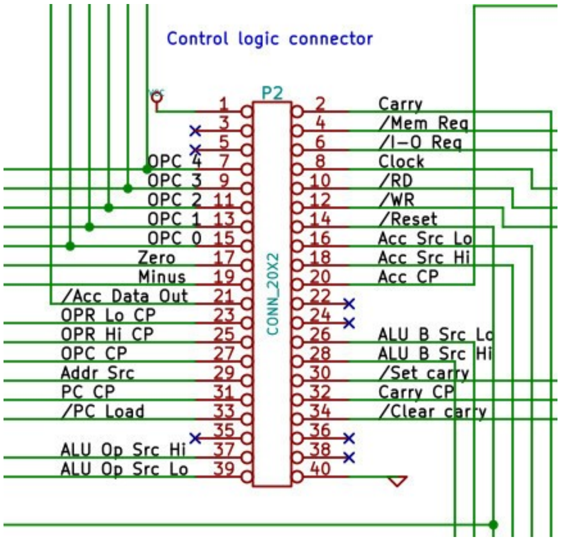
The instruction format of this processor is a 5-bit operation code (opcode, hexadecimal 00 to 1F), and an optional one- or two-byte operand. This register holds both the opcode and the operand of the current instruction, once they are fetched from memory. Note only the lower 5 bits of the opcode byte, stored in U12 and U13, are used.

Data-out buffer



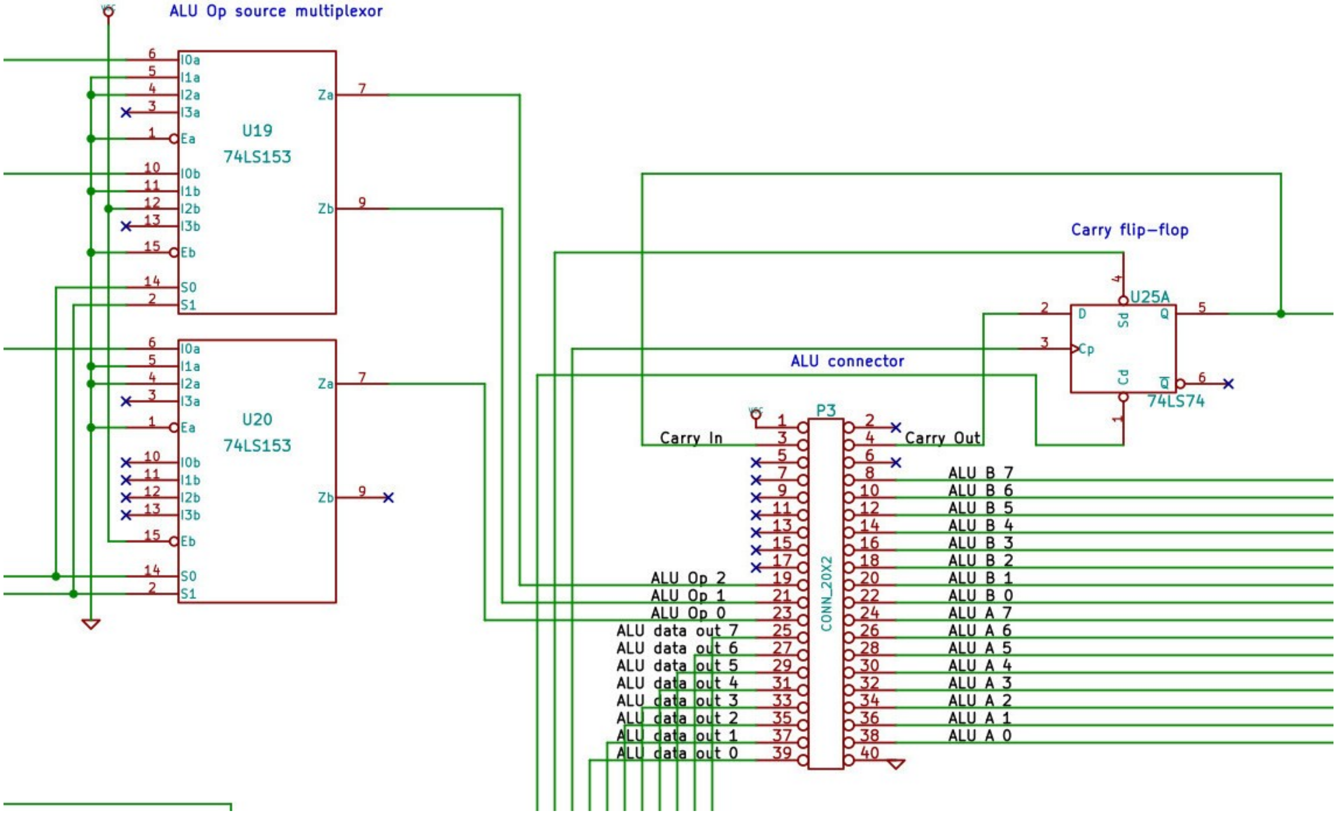
This processor uses a bi-directional data bus to connect to system memory and input/output ports. If a read or input instruction is being performed, the processor is receiving data from the system, and this buffer is “closed” (the outputs are in third-state, a high-impedance state like a cut wire). If a write or output instruction is being performed, the processor is sending data out onto the address bus. Then the buffer “opens”, controlled by the control logic, and its outputs become active.

Control connector



Here you see the various inputs to and outputs from the control logic board. The inputs are the current operation code (opcode) and the flags (zero, minus and carry).The outputs are of several types. There are the multiplexer address select lines, like Addr_Src (address source, for the address source multiplexer). There are the register write clock pulses, like Acc_CP (accumulator clock pulse), and a pulse to increment the program counter (PC_CP). There are control signals to load the PC, and to set and clear the carry flag. The Carry_CP causes the carry flip-flop to store the current carry-out bit from the ALU. Finally there are the system control signals, memory request (/Mem_Req), input-output request (/I-O_Req), read (/RD) and write (/WR). The forward slash indicates an active-low signal, that is, that the signal is asserted when it is zero.

ALU op source multiplexer, carry flip-flop, and ALU connector

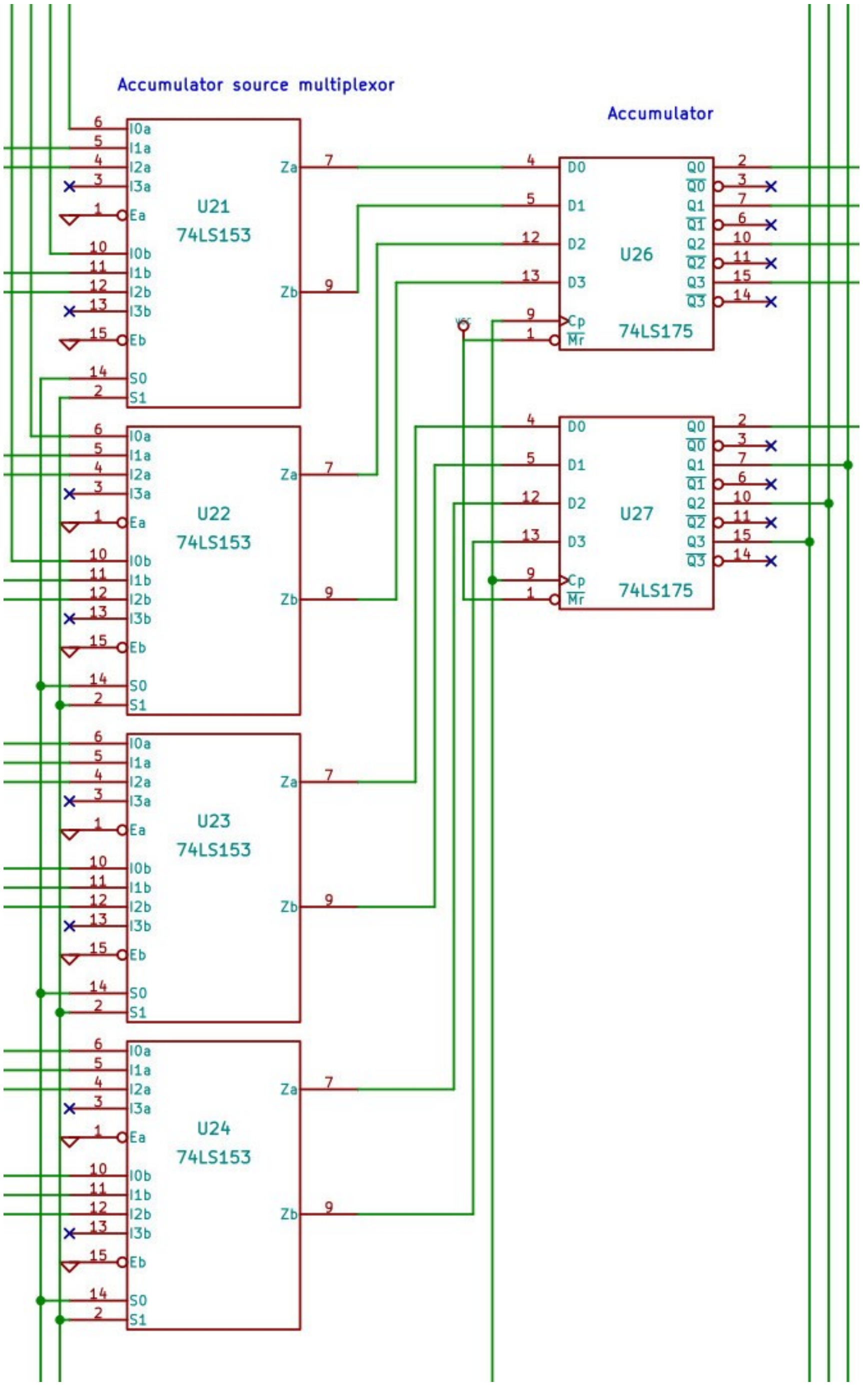


The ALU connector shows the inputs to and outputs from the ALU. The data inputs to the ALU are two 8-bit operands, ALU_A and ALU_B, and the carry-in from the carry flip-flop. The ALU operand is a three-bit input that signals the ALU which operation to perform. For simplicity, the ALU opcode is the low-order three bits of the arithmetic-logic processor opcodes. The outputs from the ALU are the 8-bit ALU data out (for example, the sum from an addition operation) and the carry-out.

The carry flip-flop stores the carry-out from the most recent arithmetic operation. It can also be set or cleared by the set carry flag (SCF) and clear carry flag (CCF) instructions through the Sd (set data) and Cd (clear data) inputs.

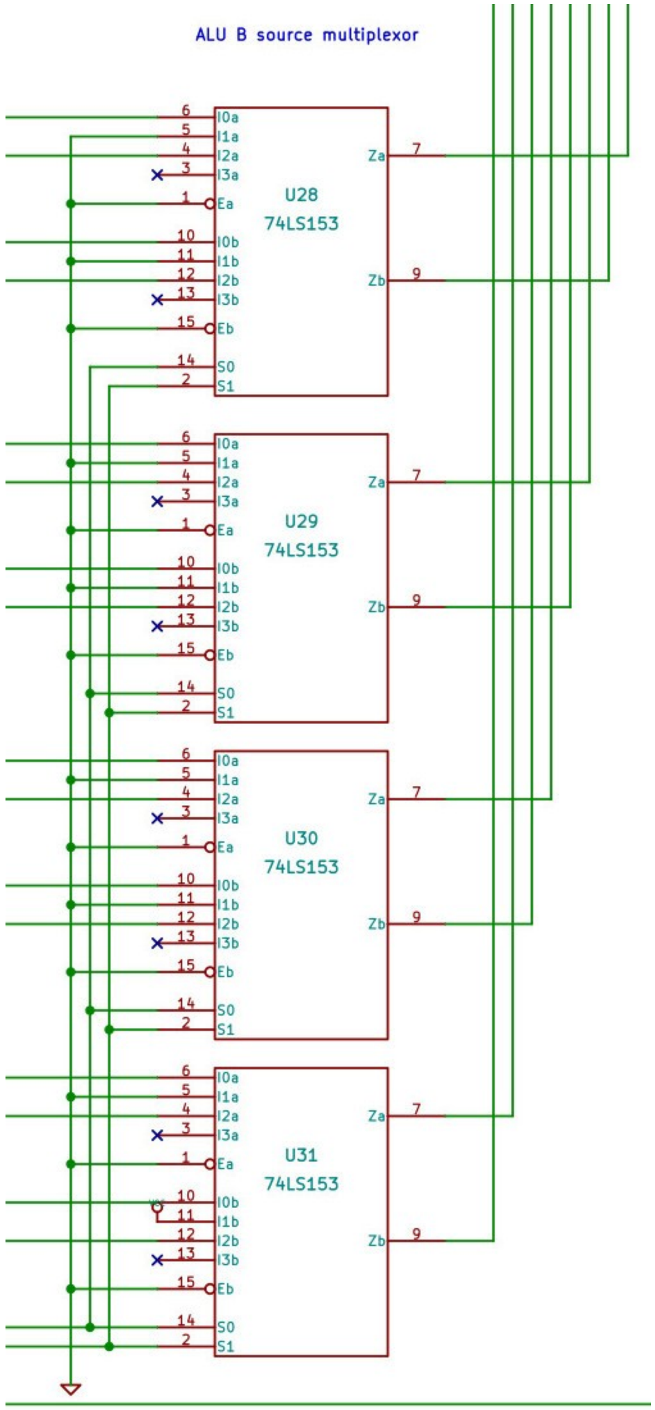
There are special instructions to increment and decrement the accumulator, and a compare instruction that subtracts but only sets the carry flag. For those instructions, the ALU op source multiplexer is configured to send an addition ALU opcode (000 binary) or a subtraction ALU opcode (010 binary) through the multiplexer inputs 1 and 2, respectively. All other arithmetic operations pass the ALU opcode from the lower 3 bits of the processor instruction opcode to the ALU through the 0 inputs of the multiplexer.

Accumulator source multiplexer and accumulator



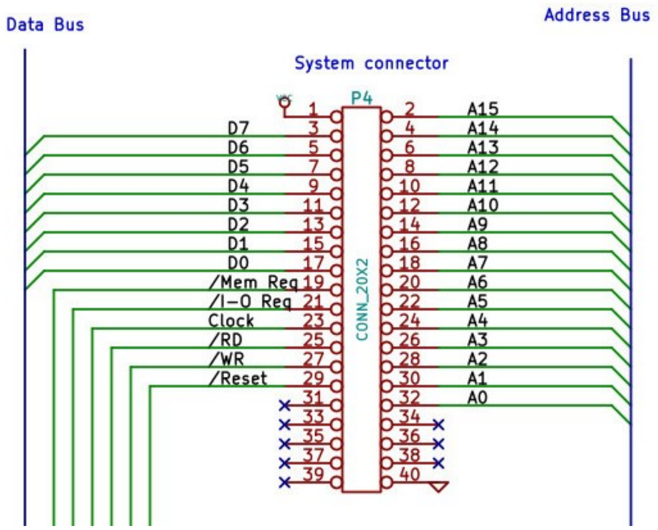
The accumulator is an 8-bit register that is the only data register in the processor available to the programmer. The accumulator source multiplexer selects the data input for the accumulator from three possible sources: The ALU output, the lower 8-bits of the instruction operand, or the data bus (from memory or ports). The multiplexer address and accumulator register write pulse come from the control logic.

ALU B source multiplexer



The ALU has two 8-bit data inputs, A and B. The A input always comes from the accumulator. The B input varies depending on the instruction. The ALU B source multiplexer controls where the ALU B input comes from. There are three possible sources: The data bus (from memory), the lower 8-bits of the instruction operand (immediate operations), and a hard-wired 0000 0001b for the increment and decrement operations (input 1 of the multiplexer).

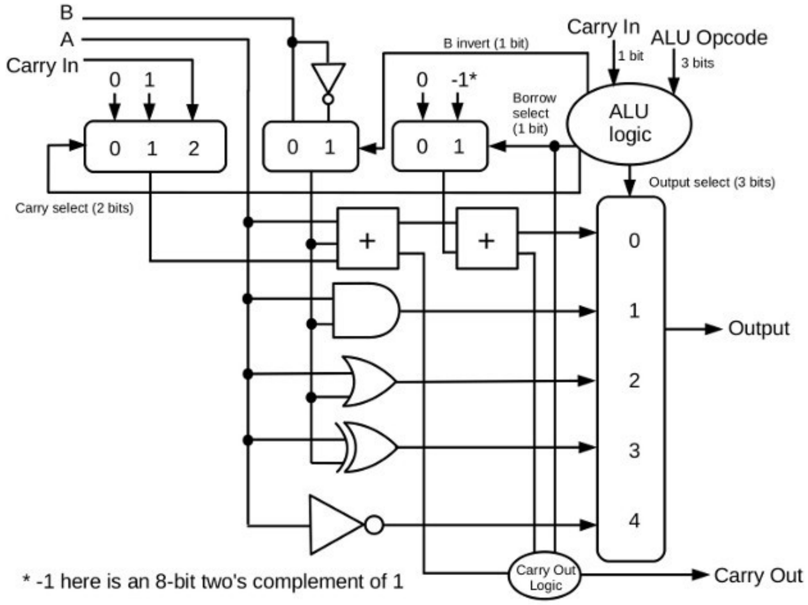
System connector



This connector has the address and data bus connections, and the control outputs for reading and writing the ports and memory that are on the system board. Also passed through this connector are the system clock and the reset signal, allowing for using a clock and reset switch on the system board or the processor control board, depending on switch and jumper settings. +5V and GND are also passed through. You may note the similarity between the signals here and signals from a Z80 processor. This is intentional, to allow Z80-based systems such as the CPUville Original Z80 and Single-board Z80 computers to serve as system boards for the CPUville 8-bit processor.

ALU schematics

The ALU is a piece of combinational logic that performs the arithmetic (addition and subtraction) and logical operations of the processor. It does this by performing all the operations on the inputs at the same time, and selecting only the one indicated by the instruction to appear on the outputs. This applies to the carry-out also, which is computed by the adder even when requesting a logical operation. These unwanted carry-outs are not stored in the carry flip-flop, so they have no affect on the system. Here is a block diagram of the ALU:

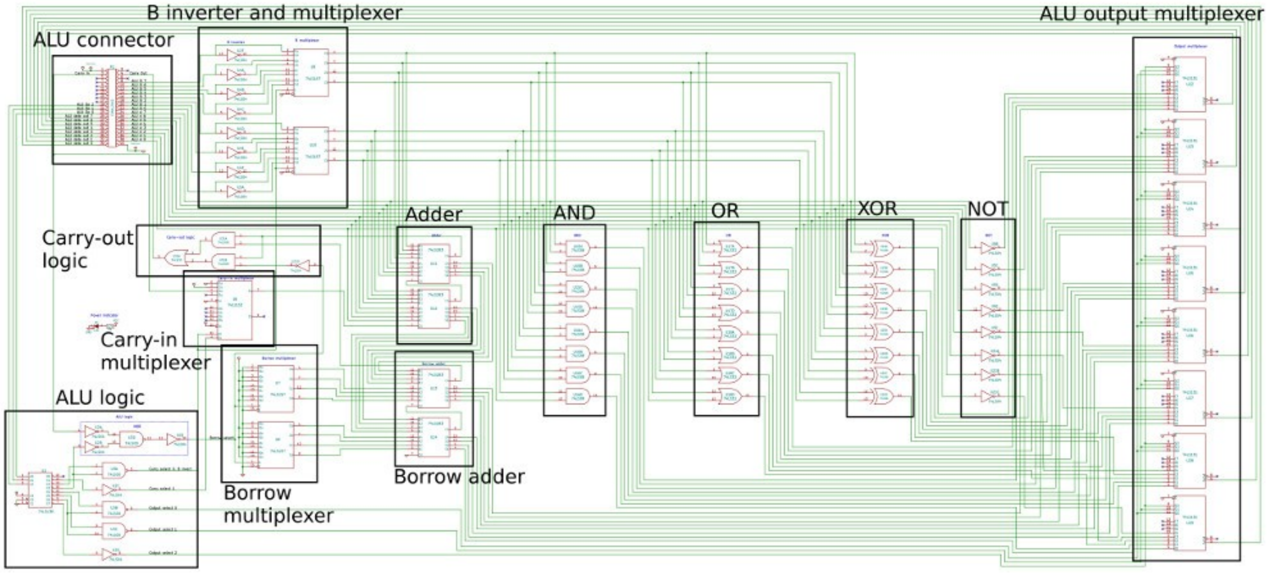


The A and B data inputs, and the carry-in from the carry flip-flop on the main board, are shown on the left. The 3-bit ALU opcode and the carry-in are inputs to a logic circuit that sets multiplexer addresses for the carry-in, B inversion, borrow select, and output select multiplexers.

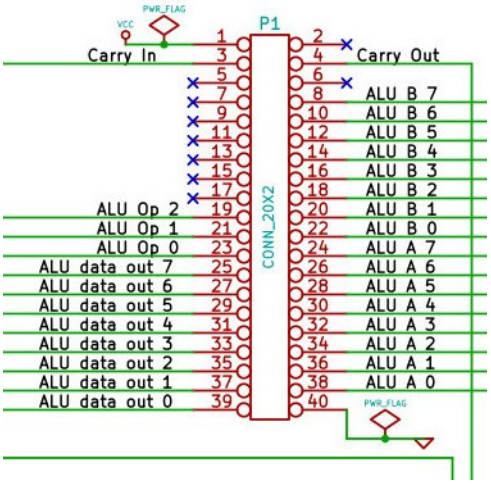
The A input is fed to all the arithmetic-logic elements of the ALU at the same time. The B input may be inverted for subtraction by twos-complement addition. The carry-in may be from the carry flip-flop (add with carry operations) or zero (for plain add operations). In subtraction, a carry-in of one is selected to complete the two's-complement addition. A second "borrow adder" is in series with the first adder, to subtract one (again, by two's-complement addition) from the result of a subtraction if there is a borrow-in.

The outputs are the carry-out, and the output of the ALU output multiplexer. Also shown is a bit of logic that determines if the carry-out should come from the first adder, or from the borrow adder in the case of a subtract with borrow operation.

Below is a map image showing where the following schematics come from. The schematic underlying this map image is low-resolution. To see a high-resolution image of the ALU schematic, download it from the CPUville website.

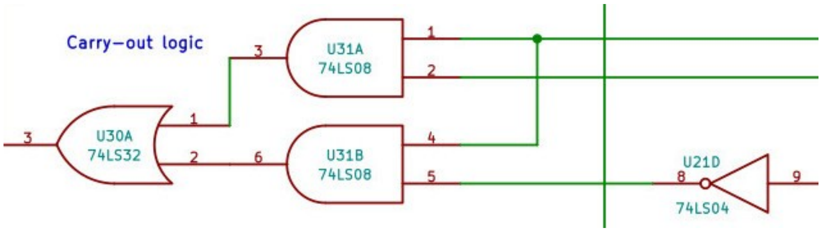


ALU connector



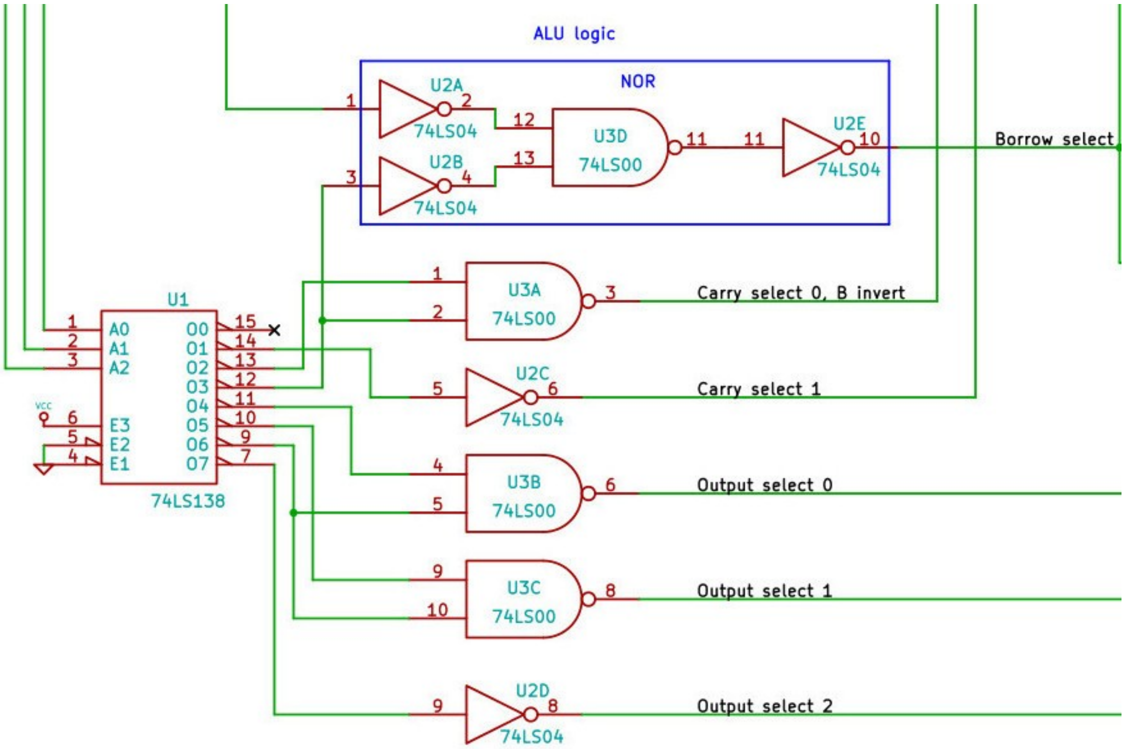
This is the same connector as seen on the main board. The outputs there are inputs here to the ALU. The ALU inputs are the 8-bit operands A and B, the carry-in, and the ALU opcode. The outputs are the ALU data out, and the carry-out. Power (+5V and ground) is also passed to the ALU through this connector.

Carry-out logic



This bit of logic determines if the carry-out from subtract-with-borrow operations will be the carry-out from the main adder, or the carry-out from the borrow adder. There is a detailed explanation of the logic for this in the Appendix.

ALU logic

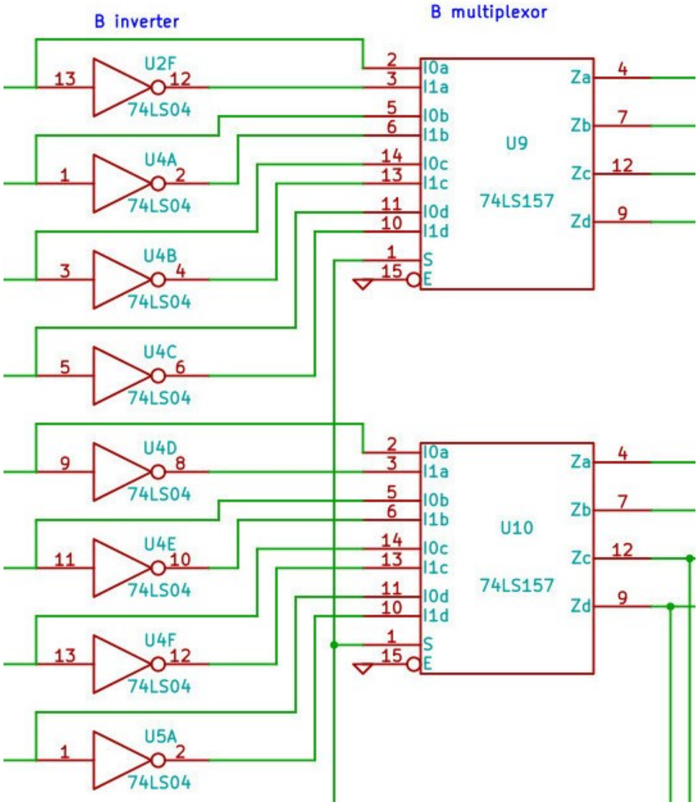


This logic circuit takes as input the three-bit ALU opcode, and the carry-in. The outputs are the various ALU multiplexer select lines. The 74LS138 is a 1-of-8 decoder. Only 7 outputs are used.

Note the boxed in NOR operation, made of three inverters and a NAND gate. I did it this way because there were extra inverters and NAND gates available. To use a straight NOR gate I would have had to add another IC to the board.

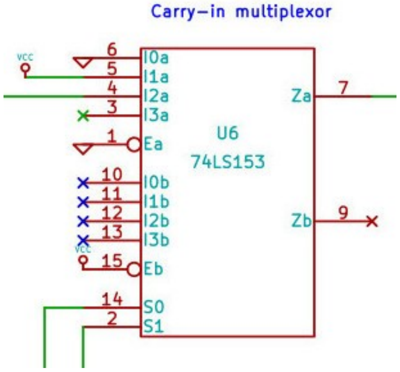
For a detailed explanation of this logic circuit, see the Appendix.

B inverter and multiplexer



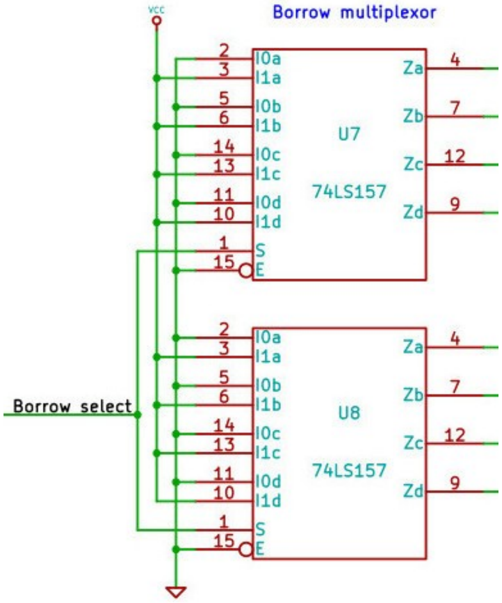
The B inverter inverts the B input for use in subtraction operations. The multiplexer selects whether an uninverted or inverted B value is used as an input to the adder.

Carry-in multiplexer



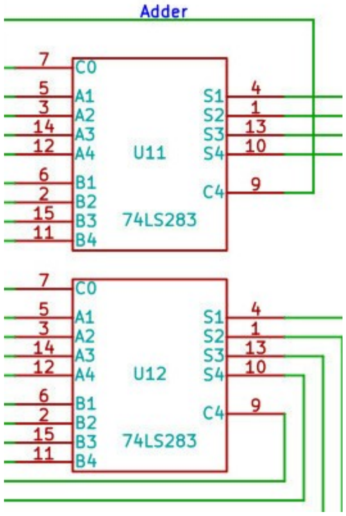
This multiplexer selects which carry-in to feed to the adder. There are three choices: the carry flag from the carry flip-flop, one (for subtract operations to complete a twos-complement addition) and zero, for plain additions, like the ADD operation. The select lines come from the control logic. Note there are two multiplexers on this IC, only one is used (the A multiplexer), and only 3 of the 4 inputs of that one are used.

Borrow multiplexer



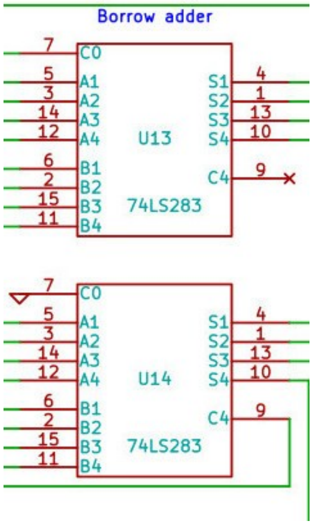
For subtract-with-borrow operations, the borrow-in will be a twos-complement addition (subtraction) of 1 from the result of the main subtraction. This multiplexer selects either a negative 1 or zero input for the borrow adder, depending on whether there is a borrow-in or not.

Adder



This is the main adder, used for addition, addition with carry, and subtraction operations. For subtract-with-borrow, there is another adder in series with this one, to subtract 1 by two’s complement addition from the main result if there is a borrow-in.

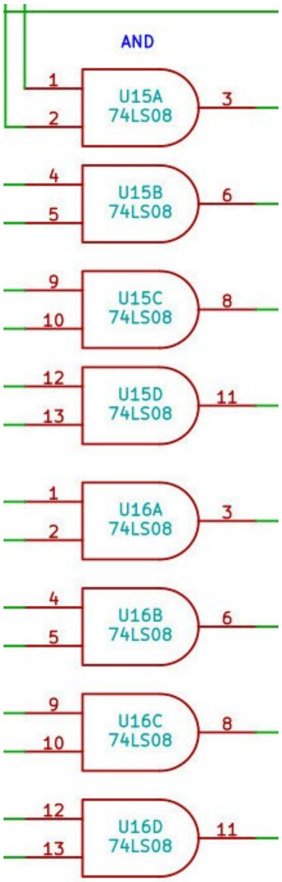
Borrow adder



This is the borrow adder that is in series with the main adder. It is used to subtract 1 from the result of a subtraction operation performed by the main adder, if there is a borrow-in.

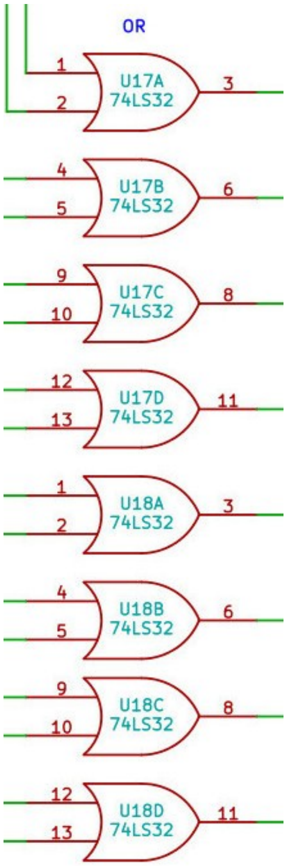
I am not totally happy with the subtraction part of the ALU. I think perhaps designing a dedicated subtracter, with basic logic gates, might have been a better approach. The implementation I used here works, but it feels a little kludge-y.

AND



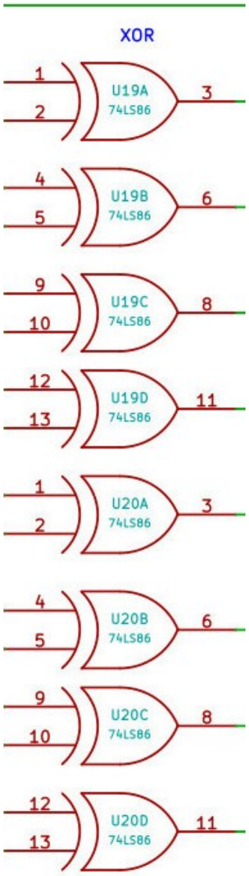
This is a simple 8-bit, two input AND operation, performed on the ALU A and B inputs. The output is fed to the ALU output multiplexer inputs.

OR



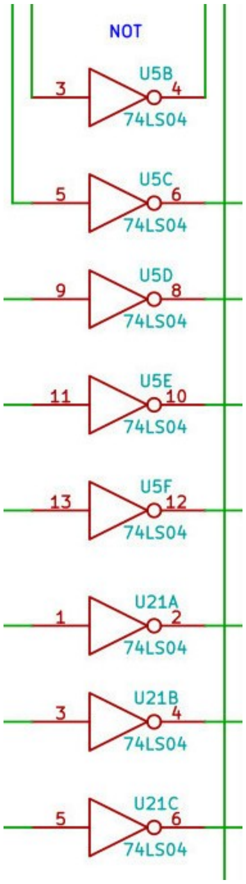
This is a simple 8-bit, two input OR operation, performed on the ALU A and B inputs. The output is fed to the ALU output multiplexer inputs.

XOR



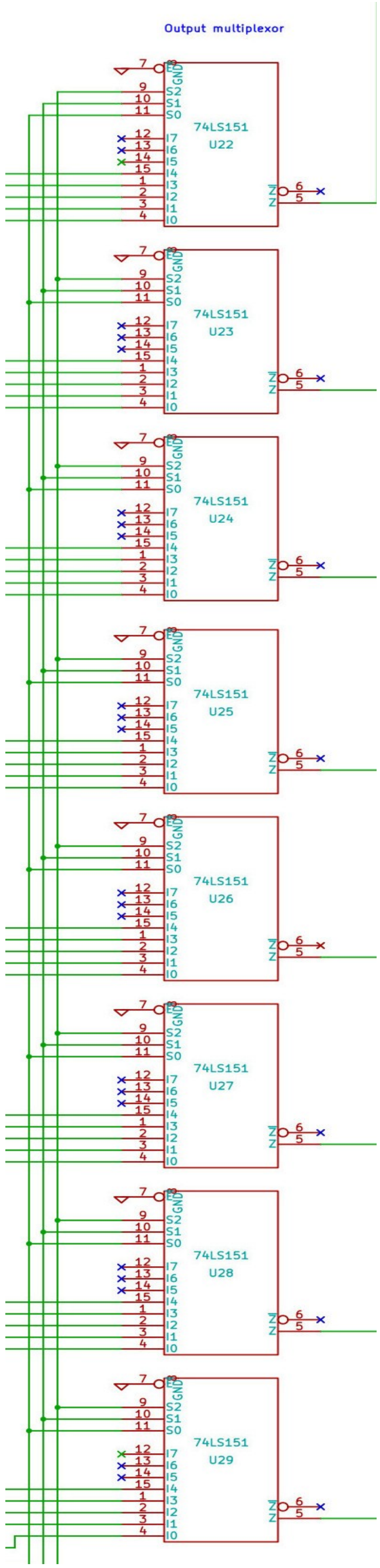
This is a simple 8-bit, two input exclusive-OR (XOR) operation, performed on the ALU A and B inputs. The output is fed to the ALU output multiplexer inputs.

NOT



This is a simple inversion operation, performed on the ALU A input. The output is sent to the ALU output multiplexer inputs.

ALU output multiplexer



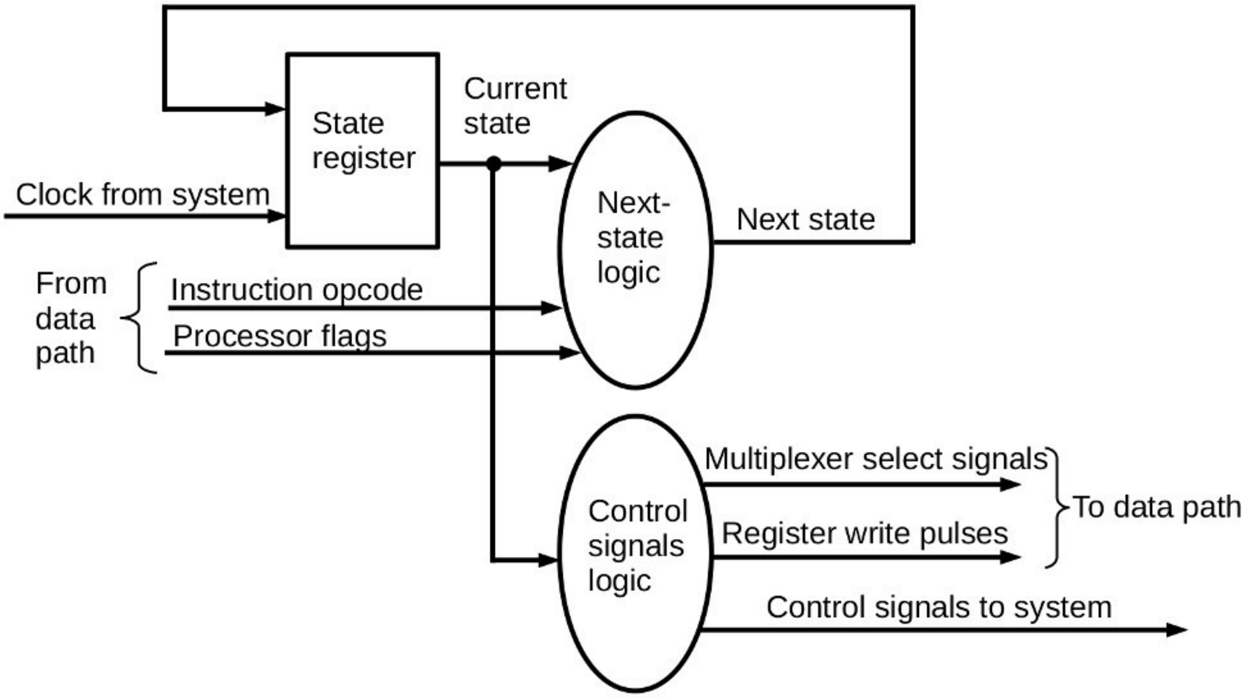
This is an 8-bit, eight-input multiplexer. Only 5 of the possible 8 inputs are used. This multiplexer determines which ALU operation output is sent out to the main board through the ALU connector. Although all operations are performed simultaneously by the ALU, only the selected operation output is sent.

Control board schematic

The control board is the beating heart of the processor. The main board and the ALU are directed, or controlled, by the outputs of the control board.

The central core of the control board is the finite state machine, made of the state register and the next-state logic. This electronic machine is prodded by the upgoing system clock edges to run through a series of states, which are bit patterns (numbers) generated by the next-state logic and stored in the state register. The next-state logic takes as input the current state, current instruction opcode and the processor flags, and produces the next state.

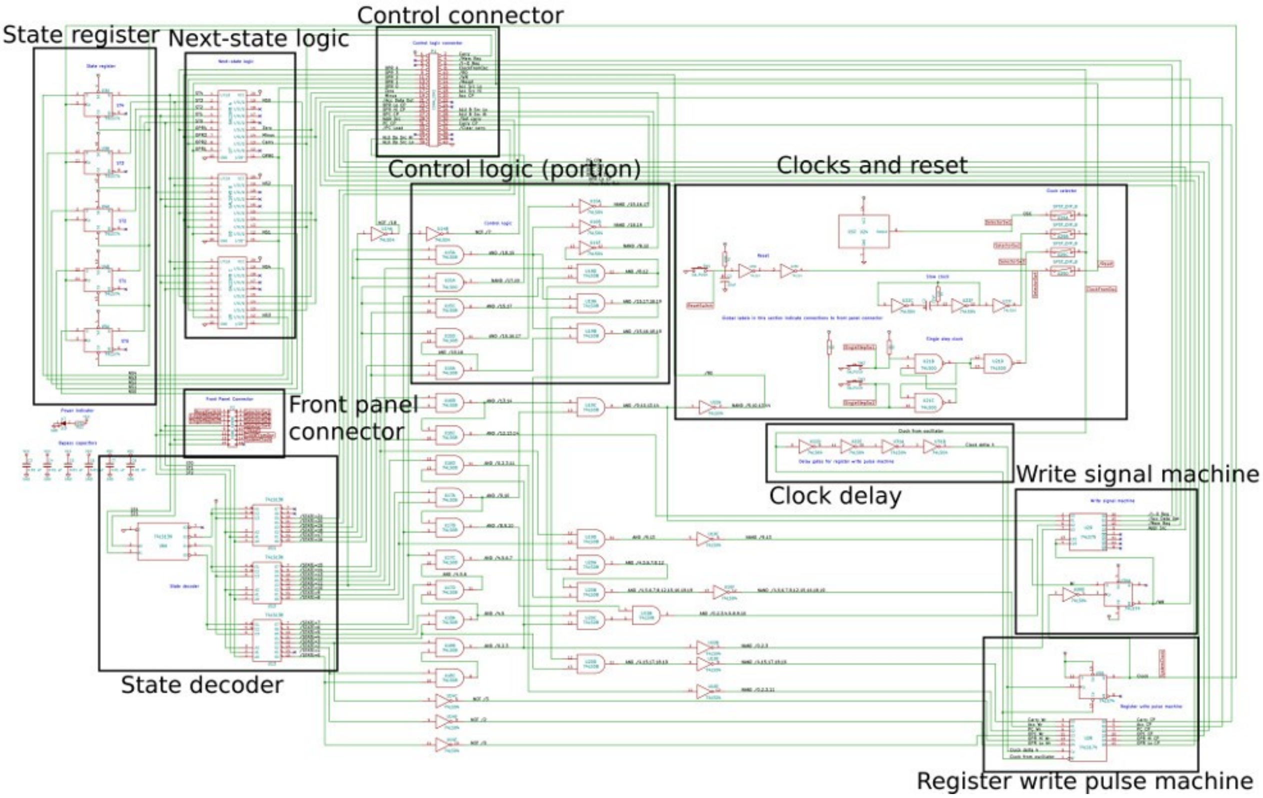
Each current state is also the only input to the control signals logic. The control logic outputs are the multiplexer address lines, the register write pulses, and the system control signals, such as memory request and read and write, that together make the computer run. Here is a block diagram of the control board:



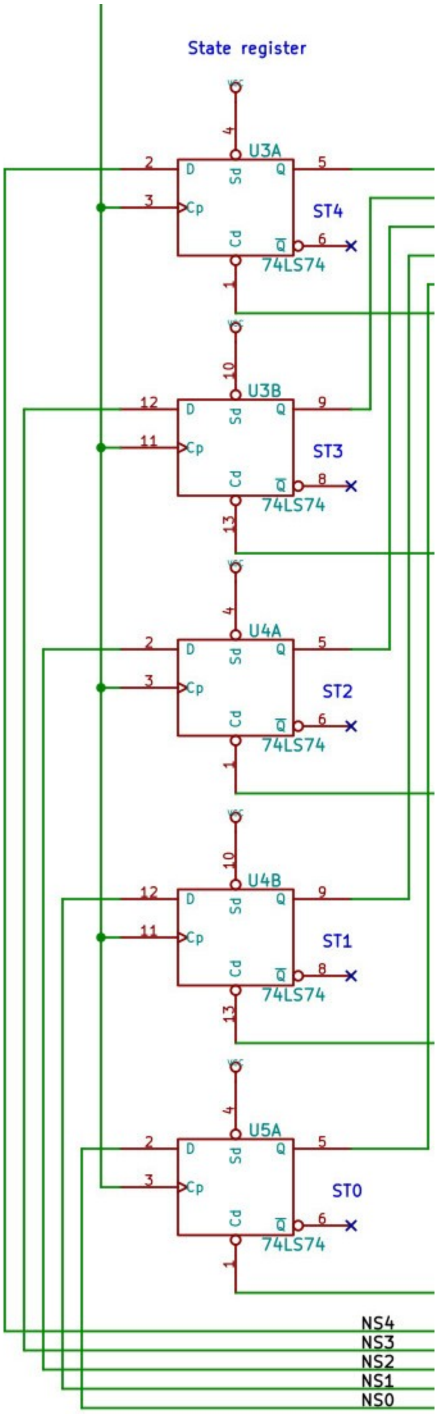
There are detailed explanations of the next-state and control signals logic in the Appendix.

For convenience the control board also has three clock signal generators and a reset circuit. These are here so the processor can be used with a variety of simple system boards that might be on the bottom of the stack, allowing the clock selection and reset to be performed on the control board, which will usually be on the top of the stack.

Here is a map image showing where the individual schematics of the control board are taken from. The schematic underlying this map image is low resolution. For a high-resolution schematic, download the schematic image from the CPUville website.

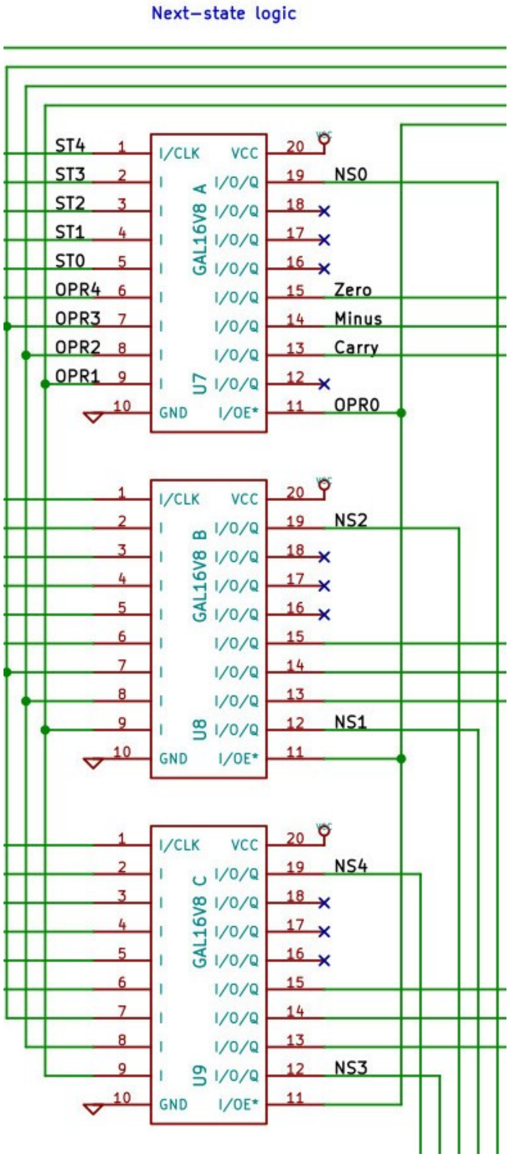


State register



This register holds the current state value. The processor has a total of 22 states (from 0 to 21), so a five-bit state register is adequate. The state register is written on every upgoing edge of the system clock, taking as input the next-state value from the next-state logic (NS bits 0 to 4). The state register is cleared to zero during a system reset through the Cd inputs. Together with the cleared program counter, this assures that when coming out of reset, state 0 (opcode fetch) is executed on address 0x0000.

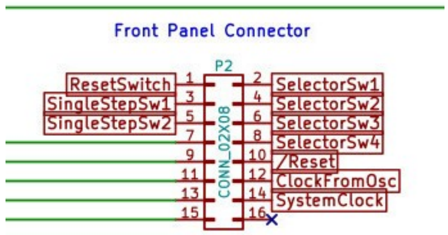
Next-state logic



The next-state logic is a piece of combinational logic that takes as input the current state, the instruction opcode, and the system flags, and produces as output the next state. It is implemented on GAL16V8 programmable logic ICs. I used them because each GAL16V8 takes the place of about 10 discrete logic ICs (AND and OR gates), and saves the processor from needed another circuit board. It does hide some complexity though. There is a full explanation of this logic circuit in the Appendix.

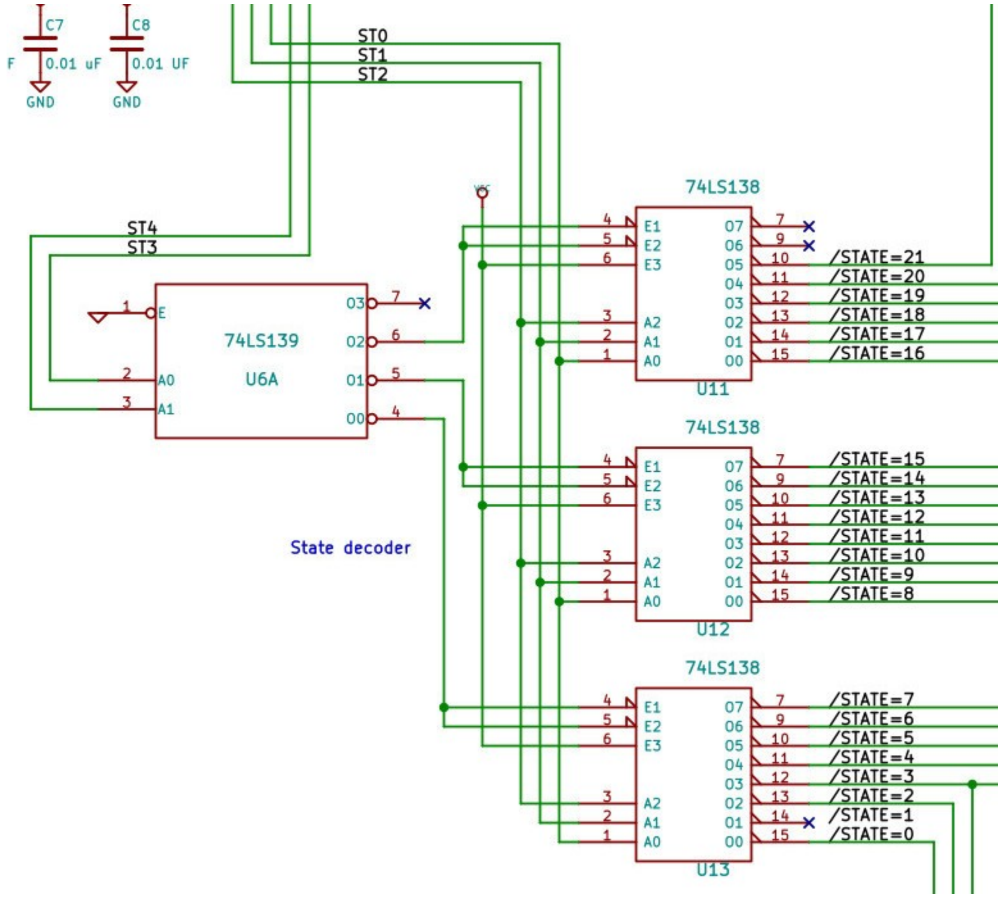
If you are a TTL purist, and want to have this logic circuit implemented with discrete logic ICs, contact me, and I will work up a schematic for you. You can connect your discrete-IC next-state logic board to the control board through 20-conductor ribbon cables terminated with 20-pin DIP plugs that plug into these three GAL sockets.

Front panel connector



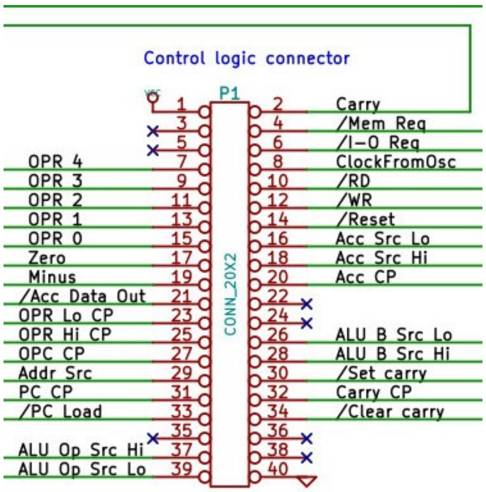
This connector allows for clock selection and reset switches on a front panel display to control the computer. The global labels (red square labels) on the schematic are connections to the points with similar labels on the Clocks and Reset schematic. It also carries the state bits from the state register to the front panel display.

State decoder



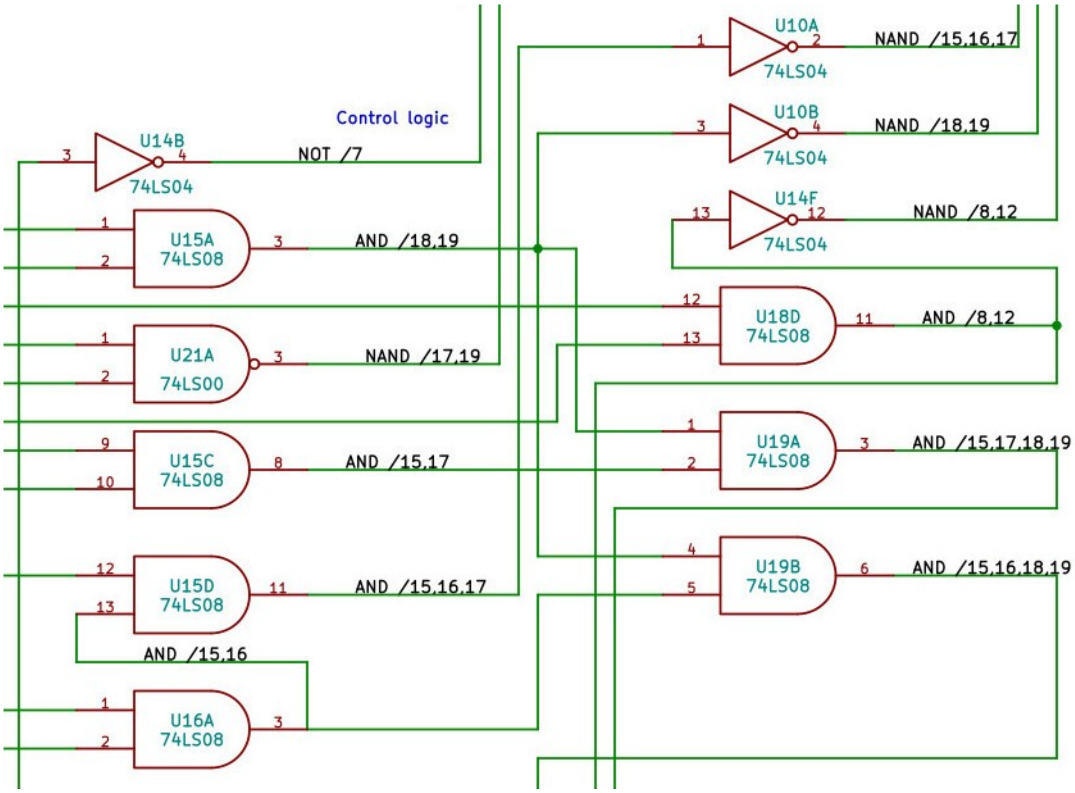
The state decoder is a 1-of-24 decoder, only 21 of the outputs are used. The outputs are active-low state signals. These outputs become the input to the control signals logic. Note there the state=1 output is not used. None of the control logic equations use this bit. See the “Control Signals Logic Explanation” in the Appendix for details.

Control connector



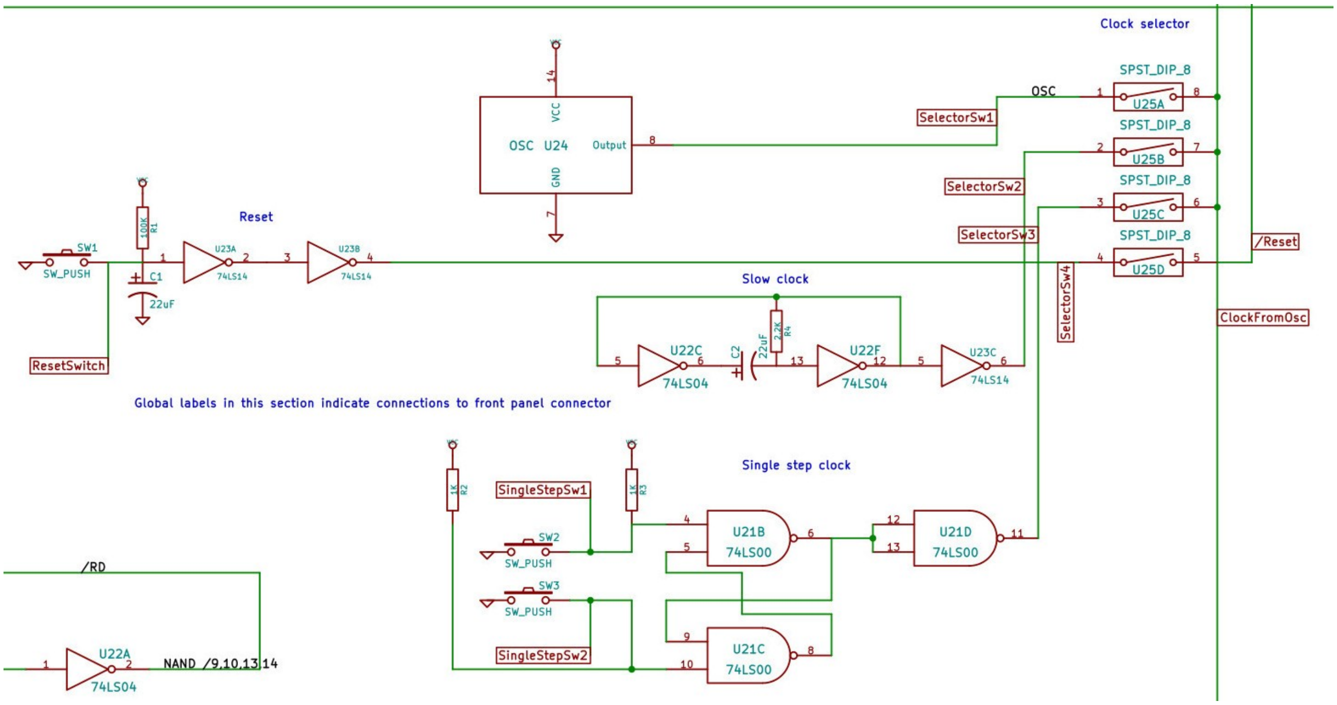
This connector is identical in its schematic form to the control connector on the main board.

Control logic (portion)



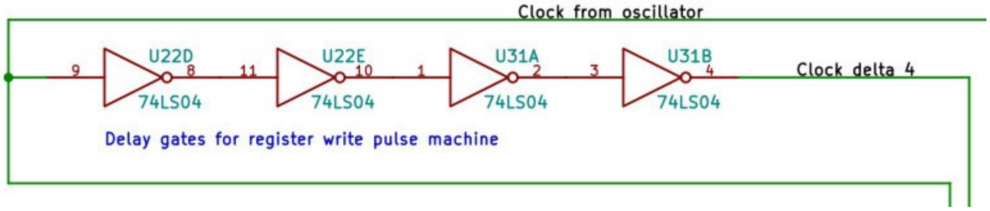
This shows part of the logic network that generates the multiplexer select addresses, register write pulses, and system control signals. The control signals logic is explained in detail in the Appendix.

Clocks and reset



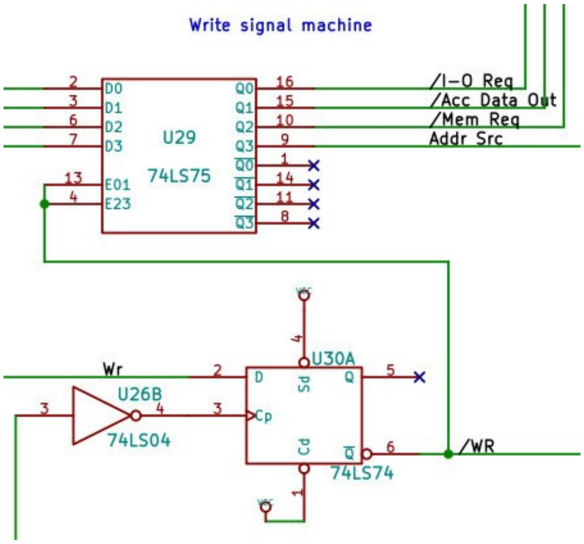
Usually part of a system board, these circuits are on the control board to allow convenient operation of the computer if the system board is on the bottom of the stack. It also allows a builder to make a simple system board that does not have these circuits on it, since these can be used instead. The global labels here show connections to the front panel connector, allowing switches on the front panel to control these circuits.

Clock delay



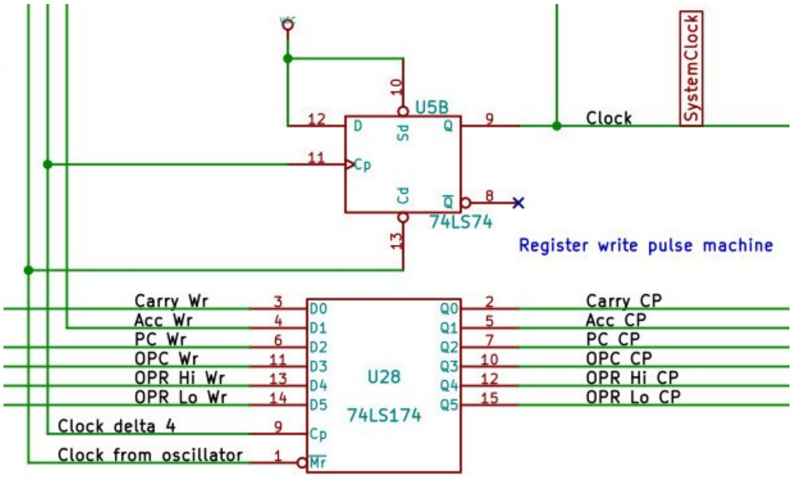
The register write pulse machine, described below, uses an offset from the system clock to perform its function. Running a signal through a series of gates adds about a 10 nanosecond per gate delay.

Write signal machine



A flip-flop (the 74LS74) holds the system write signal (/WR, active-low) constant over a change in states, which is important for the timing needed to write data to RAM and to output ports. A latch (the 74LS75) holds steady the address source (Addr_Src) multiplexer select signal, and the other signals needed to write data to the system. Details of write signal timing can be found in the Appendix.

Register write pulse machine



The control logic creates register write signals as levels, not clock edges. This machine converts these levels into edges. The Clock_delta_4 input writes the signals into the 74LS174 register, creating the clock pulses (edges) on the outputs needed to write the registers and increment the program counter (PC). After these edges are generated a master reset is performed by a low level on the Clock_from_oscillator, which is offset from Clock_delta_4 because of the built in delay shown above. This reset is important, because the register write pulses need to return to low after an upgoing edge is sent. Note the system clock is generated from another flip-flop so that its edges, which write the state register, will be synchronous with the other register write pulse edges. Details of register write machine timing can be found in the Appendix.

Appendix

ALU parts organizer

Capacitor, 0.01uF 6	74LS00 quad NAND 1	74LS04 hex inverter 4	74LS08 quad AND 3
74LS138 1-of-8 decoder 1	DIL 14-pin socket 13	74LS151 8-input multiplexer 8	DIL 16-pin socket 18
74LS153 dual 4-input multiplexer 1	74LS157 quad 2-input multiplexer 4	74LS283 4-bit full adder 4	74LS32 quad OR 3
74LS86 quad XOR 2	40-pin header 1	Resistor, 470 ohm, ¼ watt Yellow-Violet-Brown 1	LED 1

ALU parts list

I buy almost all my parts from Jameco. If you buy from a different supplier, you can check the datasheets for these parts on the Jameco website by referring to the part number.

Part	PCB reference	Number per unit	Jameco Part No.
Cap 0.01 uF	C1 – C6	6	15229
74LS00	U3	1	46252
74LS04	U2, U4, U5, U21	4	46316
74LS08	U15, U16, U31	3	46375
74LS138	U1	1	46607
74LS151	U22 – U29	8	46703
74LS153	U6	1	46720
74LS157	U7 – U10	4	46771
74LS283	U11 – U14	4	47423
74LS32	U17, U18, U30	3	47466
74LS86	U19, U20	2	48098
40-pin header	P1	1	53532
14-pin socket		13	112214
16-pin socket		18	112222
470 ohm	R1	1	690785
LED (red)	D1	1	2081932

Main board parts organizer

Capacitor, 0.01uF 6	74LS04 hex inverter 1	74LS153 dual 4-input multiplexers 10	74LS157 quad 2-input multiplexer 4
74LS161 binary counter 4	74LS175 quad D flip-flop 8	74LS244 octal buffer 1	74LS32 quad OR 2
74LS74 dual D flip-flop 1	40-pin header 2	50-pin header 1	40-pin header receptacle 1
14-pin socket 4	16-pin socket 26	20-pin socket 1	Power-in jack 1
Resistor, 470 ohm Yellow-Violet-Brown 1	LED 1		

Main board parts list

Part	PCB reference	Number per unit	Jameco Part No.
Cap 0.01 uF	C1 – C6	6	15229
74LS04	U7	1	46316
74LS153	U19 – U24, U28 – U31	10	46720
74LS157	U8 – U11	4	46771
74LS161	U1 – U4	4	46818
74LS175	U12 – U17, U26, U27	8	46957
74LS244	U18	1	47183
74LS32	U5, U6	2	47466
74LS74	U25	1	48004
40-pin header	P2, P4	2	53532
50-pin header	P1	1	53560
40-pin header receptacle	P3	1	111705
14-pin socket		4	112214
16-pin socket		26	112222
20-pin socket		1	112248
Power-in jack		1	137673
470 ohm	R1	1	690785
LED (red)	D1	1	2081932

Control board parts organizer

Capacitor, 0.01uF 6	Oscillator, 1.8432 MHz 1	4-position DIP switch 1	74LS00 quad NAND 1
74LS04 hex inverter 5	74LS08 quad AND 6	74LS138 1-of-8 decoder 3	74LS139 dual 1-of-4 decoder 1
74LS14 hex inverter Schmitt trigger 1	74LS174 hex D flip-flop 1	74LS74 dual D flip-flop 4	74LS75 quad latch 1
GAL 16V8-D programmable logic 3	16-pin header 1	40-pin header receptacle 1	Capacitor, 22 uF 2
14-pin socket 17	16-pin socket 6	20-pin socket 3	Pushbutton switch 3
Resistor, 470 ohm Yellow-Violet-Brown 1	Resistor, 1K ohm Brown-Black-Red 2	Resistor, 2.2K ohm Red-red-red 1	Resistor, 100K ohm Brown-black-yellow 1
LED (red) 1			

Control board parts list

Part	PCB Reference	Number per unit	Jameco Part no.
Cap 0.01 uF	C3 – C8	6	15229
Osc 1.8432 MHz	U24	1	27879
4-position DIP switch	U25	1	38820
74LS00	U21	1	46252
74LS04	U10, U14, U22, U26, U31	5	46316
74LS08	U15 – U20	6	46375
74LS138	U11, U12, U13	3	46607
74LS139	U6	1	46623
74LS14	U23	1	46640
74LS174	U28	1	46931
74LS74	U3, U4, U5, U30	4	48004
74LS75	U29	1	48021
16-pin header	P2	1	109568
40-pin header receptacle	P1	1	111705
14-pin socket		17	112214
16-pin socket		6	112222
20-pin socket		3	112248
Push button switch	SW1, SW2, SW3	3	122973
Resistor 470 ohm	R11	1	690785
Resistor 1K	R2, R3	2	690865
Resistor 2.2K	R4	1	690945
Resistor 100K	R1	1	691340
GAL16V8-D	U7, U8, U9	3	876539
Cap, 22 uF	C1, C2	2	1946295
LED (red)	D7	1	2081932

ALU carry-out logic explanation

Problem: With the subtract-with-borrow instruction, in the case where the difference from the first half-subtractor (the adder configured to do subtraction) is zero, the final carry-out (borrow-out) cannot be the carry-out from the first half-subtractor. In this case, the first carry-out will be 1 (no borrow), and the carry-out from the second half-subtractor (the “borrow adder”) will be zero (borrow). What is needed is a logic circuit that in this case will output the correct borrow, the one from the borrow adder. Here is the truth table:

Borrow Select	Adder C.O.	Borrow C.O.	Final C.O.
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	**1**	**0**	**0**
1	1	1	1

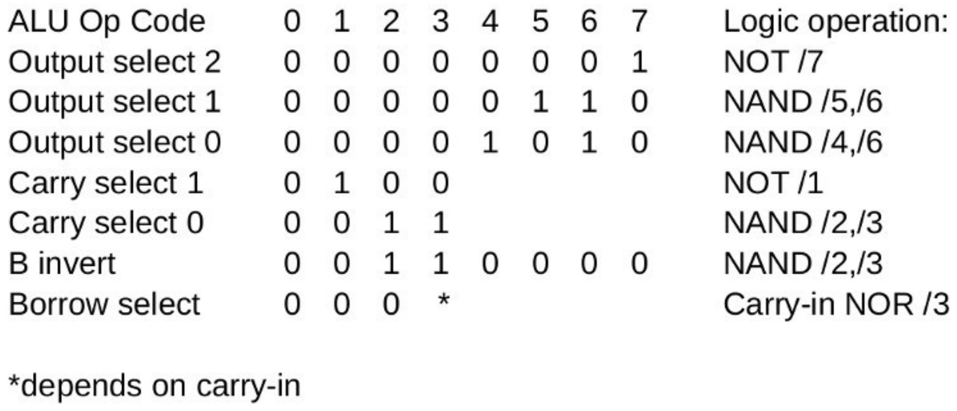
The second-to-bottom row is where the final c.o. cannot follow the main adder c.o. If the borrow select is 1, and if the borrow adder c.o. is zero, one can use that, otherwise use the adder c.o. In other words, if the borrow select is 1, the final c.o. can be adder c.o. AND borrow c.o. Here is the logic equation:

FinalCO = (NOT(BorrowSelect) AND AdderCO) OR (AdderCO AND BorrowCO)

This logic equation is implemented on the ALU board by an inverter, two AND gates, and one OR gate.

ALU logic explanation

The ALU logic sets the various ALU multiplexer select inputs, depending on the ALU operation code and the carry-in. The ALU opcode (numbers in the top row) serves as the input to a one-of-eight decoder (only 7 outputs are used). The outputs of this decoder are active-low, that is, if the ALU opcode is decimal 7, the corresponding decoder output 7 will be 0, the other decoder outputs will be 1. These outputs are used in the logic operations listed in the table below, which determine the multiplexer select control outputs.



ALU Op Code	0	1	2	3	4	5	6	7	Logic operation:
Output select 2	0	0	0	0	0	0	0	1	NOT /7
Output select 1	0	0	0	0	0	1	1	0	NAND /5,/6
Output select 0	0	0	0	0	1	0	1	0	NAND /4,/6
Carry select 1	0	1	0	0					NOT /1
Carry select 0	0	0	1	1					NAND /2,/3
B invert	0	0	1	1	0	0	0	0	NAND /2,/3
Borrow select	0	0	0	*					Carry-in NOR /3

*depends on carry-in

The borrow select control is determined by a NOR operation between decoder output 3 and the carry-in, as shown in the ALU logic schematic.

Next-state logic explanation

Each state causes the processor to perform some part of an instruction. The part it performs must finish within one clock cycle (about 500 nanoseconds if running at 2 MHz). Each instruction is made up of a sequence of states. Here are the states and what they do:

State	Function	Controls used	Registers written
0	Instruction fetch, increment PC	Addr src, Data out, Mem Req, Rd, PC inc	OPC, PC (incremented)
1	Instruction interpretation		
2	Operand Lo fetch, increment PC	Addr src, Data out, Mem Req, Rd, PC inc	OPR Lo, PC (incremented)
3	Operand Hi fetch, increment PC	Addr src, Data out, Mem Req, Rd, PC inc	OPR Hi, PC (incremented)
4	Arithmetic memory	Addr src, Data out, ALU B src, ALU Op src, Acc src	Accumulator, Carry FF
5	Logical memory	Addr src, Data out, ALU B src, ALU Op src, Acc src	Accumulator
6	NOT	ALU Op src, Acc src	Accumulator
7	Load 8-bit data from instruction to Acc	Acc src	Accumulator
8	Load 8-bit data from memory to Acc	Addr src, Data out, Acc src, Mem Req, Rd	Accumulator
9	Store 8-bit data from Acc to memory A	Addr src, Data out, Mem Req, Wr	Set Addr Src FF (control logic)
10	Store 8-bit data from Acc to memory B	Addr src, Data out, Mem Req, Wr	Reset Addr Src FF (control logic)
11	Jump		PC (load preset)
12	Input 8-bit data from port to Acc	Addr src, Data out, Acc src, I/O Req, Rd	Accumulator
13	Output 8-bit data from Acc to port A	Addr src, Data out, I/O Req, Wr	
14	Output 8-bit data from Acc to port B	Addr src, Data out, I/O Req, Wr	
15	Arithmetic immediate	ALU B src, ALU Op src, Acc src	Accumulator, Carry FF
16	Logical immediate	ALU B src, ALU Op src, Acc src	Accumulator
17	Compare immediate	ALU B src, ALU Op src	Carry FF
18	Increment Acc	ALU B src, ALU Op src	Accumulator, Carry FF
19	Decrement Acc	ALU B src, ALU Op src	Accumulator, Carry FF
20	Set carry flag	Set carry flag	Carry FF
21	Clear carry flag	Clear carry flag	Carry FF

For example, to perform the ADD instruction (add memory to accumulator) the processor must perform this state sequence:

State 0: Fetch the instruction opcode. This state also increments the program counter.

State 1: Interpret the instruction – allows the next-state logic to calculate the next state.

State 2: Fetch the low-order byte of the memory address (operand-low). Increment the PC.

State 3: Fetch the high-order byte of the memory address (operand-high). Increment the PC.

State 4: Perform the addition. The address in the instruction operand causes the memory to output the byte to be added. This byte is sent to the ALU along with the contents of the accumulator. The ALU performs the addition. The sum is written into the accumulator and the the carry-out is stored in the carry flip-flop at the end of the clock cycle.

All instructions begin with state 0 and 1. The state(s) after 1 perform the instruction. The one exception is the NOP instruction (no operation) which only performs states 0 and 1.

After the last state of each instruction is performed the next-state logic produces the 0 state, which is the default output of the next-state logic. That is, if no 1 bits are specified by the next-state logic, all the next-state output bits will be zero. This means the next-state will be 0, and an instruction fetch will be done.

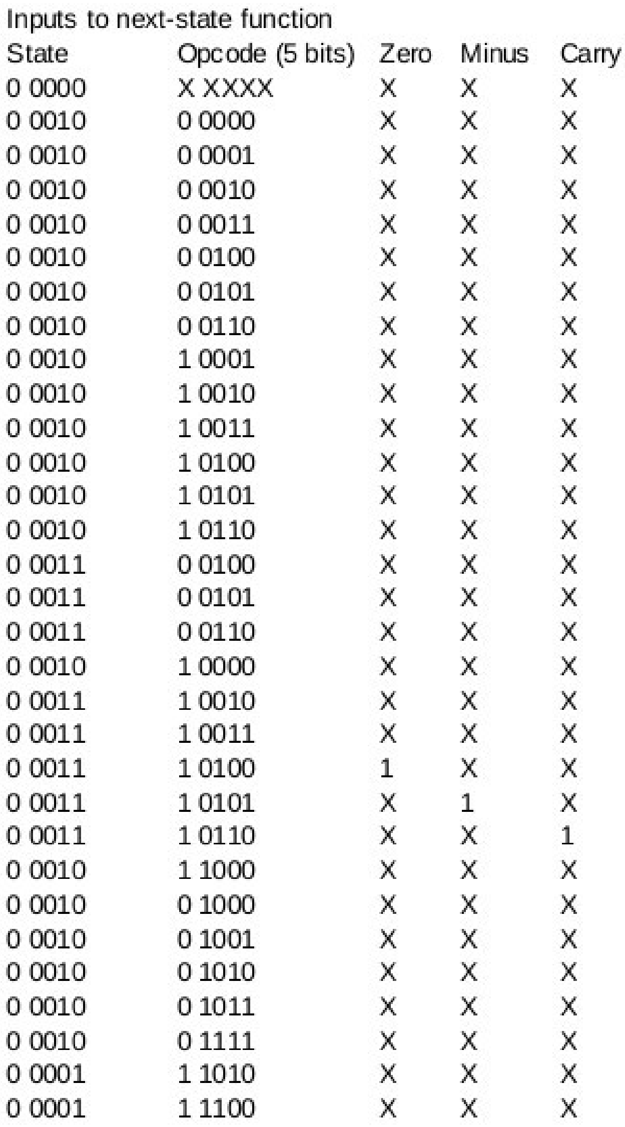
Here is a table of the instructions with their state sequences:

Opcode	Mnemonic	Function	State sequence
00	ADD	Add memory to accumulator	0, 1, 2, 3, 4
01	ADC	Add memory and carry to accumulator	0, 1, 2, 3, 4
02	SUB	Subtract memory from accumulator	0, 1, 2, 3, 4
03	SBB	Subtract memory and borrow from accumulator	0, 1, 2, 3, 4
04	AND	Binary AND memory with accumulator	0, 1, 2, 3, 5
05	OR	Binary OR memory with accumulator	0, 1, 2, 3, 5
06	XOR	Binary XOR memory with accumulator	0, 1, 2, 3, 5
07	NOT	Complement accumulator	0, 1, 6
08	ADDIM	Add 8-bit value in instruction to accumulator	0, 1, 2, 15
09	ADCIM	Add 8-bit value in instruction and carry to accumulator	0, 1, 2, 15
0A	SUBIM	Subtract 8-bit value in instruction from accumulator	0, 1, 2, 15
0B	SBBIM	Subtract 8-bit value in instruction and borrow from accumulator	0, 1, 2, 15
0C	ANDIM	Binary AND 8-bit value in instruction with accumulator	0, 1, 2, 16
0D	ORIM	Binary OR 8-bit value in instruction with accumulator	0, 1, 2, 16
0E	XORIM	Binary XOR 8-bit value in instruction with accumulator	0, 1, 2, 16
0F	CMP	Subtract 8-bit value in instruction from accumulator, set carry only	0, 1, 2, 17
10	LDI	Load 8-bit value in instruction into accumulator	0, 1, 2, 7
11	LDM	Load accumulator from memory	0, 1, 2, 3, 8
12	STM	Store accumulator into memory	0, 1, 2, 3, 9,10
13	JMP	Jump to memory location	0, 1, 2, 3, 11
14	JPZ	Jump to memory location if zero	0, 1, 2, 3, [11 if met]
15	JPM	Jump to memory location if minus	0, 1, 2, 3, [11 if met]
16	JPC	Jump to memory location if carry	0, 1, 2, 3, [11 if met]
17	IN	Load accumulator with 8-bit value from port	0, 1, 2, 12
18	OUT	Send 8-bit value from accumulator to port	0, 1, 2, 13, 14
19	INC	Add 1 to accumulator	0, 1, 18
1A	DEC	Subtract 1 from accumulator	0, 1, 19
1B	SCF	Set carry flag	0, 1, 20
1C	CCF	Clear carry flag	0, 1, 21
1D		not implemented	
1E		not implemented	
1F	NOP	No operation	0, 1

The task of the next-state logic is to create the state sequences for each instruction. For example, if the current state is state 0, and the instruction opcode is 00h, the next-state needs to be 1. If the current state is 1, and the instruction opcode is 00h, the next-state needs to be 2.

So we need a logic circuit that takes as input the current state (5-bits), the instruction opcode (5 bits), and the zero, minus and carry flags (3-bits, for the conditional jumps), and generates the 5-bit next-state output.

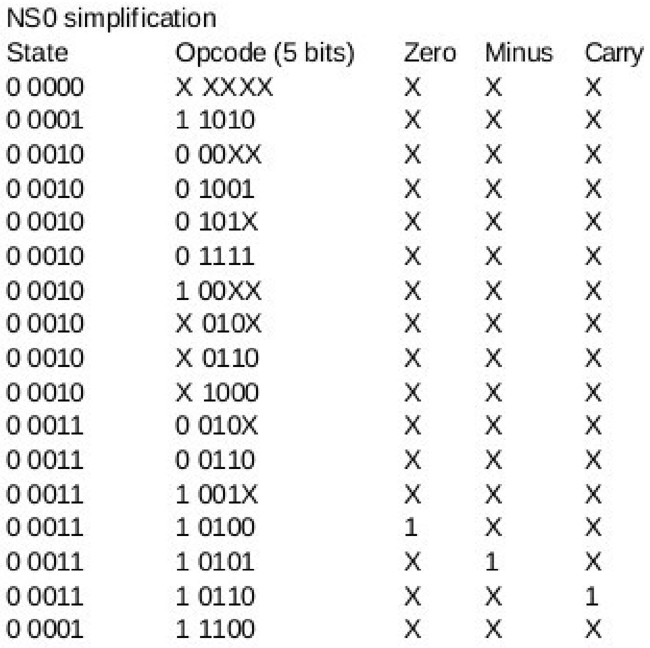
To do this, we break the problem into 5 different logic circuits, one for each next-state bit. For example, the next-state 0 bit will be 1 for next-state outputs 1, 3, 5, 7, 9, 11, 13, 19 and 21 (odd-numbered next-states). So, next-state bit 0 will be 1 if the current state is zero (for all instructions), or if the current state is 2 for opcodes 00h to 06h, 08h to 0Bh, 0Fh to 16h, 18h, 1Ah, and 1Ch, or if the current state is 3 for opcodes 04h to 06h, 12h, 13h, and 14h to 16h if the condition is met. We can write these requirements as a table, showing the binary values of the opcodes, current states and flags:



Inputs to next-state function

State	Opcode (5 bits)	Zero	Minus	Carry
0 0000	X XXXX	X	X	X
0 0010	0 0000	X	X	X
0 0010	0 0001	X	X	X
0 0010	0 0010	X	X	X
0 0010	0 0011	X	X	X
0 0010	0 0100	X	X	X
0 0010	0 0101	X	X	X
0 0010	0 0110	X	X	X
0 0010	1 0001	X	X	X
0 0010	1 0010	X	X	X
0 0010	1 0011	X	X	X
0 0010	1 0100	X	X	X
0 0010	1 0101	X	X	X
0 0010	1 0110	X	X	X
0 0011	0 0100	X	X	X
0 0011	0 0101	X	X	X
0 0011	0 0110	X	X	X
0 0010	1 0000	X	X	X
0 0011	1 0010	X	X	X
0 0011	1 0011	X	X	X
0 0011	1 0100	1	X	X
0 0011	1 0101	X	1	X
0 0011	1 0110	X	X	1
0 0010	1 1000	X	X	X
0 0010	0 1000	X	X	X
0 0010	0 1001	X	X	X
0 0010	0 1010	X	X	X
0 0010	0 1011	X	X	X
0 0010	0 1111	X	X	X
0 0001	1 1010	X	X	X
0 0001	1 1100	X	X	X

The table is a list of all the combinations of current state, opcode, and flags that need to cause the next-state zero bit (NS0) to be 1. An “X” means we don’t care what the bit value is. If the current state is 0, we don’t care what the instruction is, the next-state 0 bit will always be a 1. The table can be simplified, because some bits in the input can be either 1 or 0. For example, if the current state is 2, we don’t care what the value of opcode bit 0 is, it can be either a 1 or a 0. Going over this table and simplifying it, we get this:



NS0 simplification

State	Opcode (5 bits)	Zero	Minus	Carry
0 0000	X XXXX	X	X	X
0 0001	1 1010	X	X	X
0 0010	0 00XX	X	X	X
0 0010	0 1001	X	X	X
0 0010	0 101X	X	X	X
0 0010	0 1111	X	X	X
0 0010	1 00XX	X	X	X
0 0010	X 010X	X	X	X
0 0010	X 0110	X	X	X
0 0010	X 1000	X	X	X
0 0011	0 010X	X	X	X
0 0011	0 0110	X	X	X
0 0011	1 001X	X	X	X
0 0011	1 0100	1	X	X
0 0011	1 0101	X	1	X
0 0011	1 0110	X	X	1
0 0001	1 1100	X	X	X

Now we can write a logic equation for the next-state 0 bit, using AND-OR array logic. The equation will be long. Here is the first part of it:

NS0 = NOT(S4) AND NOT(S3) AND NOT(S2) AND NOT(S1) AND NOT(S0) OR
NOT(S4) AND NOT(S3) AND NOT(S2) AND NOT(S1) AND S0 AND OP4 AND
OP3 AND NOT(OP2) AND OP1 AND NOT(OP0) OR ...

We could implement this using a decoder for the current states. Then the equation would look like this:

NS0 = STATE0 OR STATE1 AND OP4 AND OP3 AND NOT(OP2) AND OP1 AND NOT(OP0) OR
STATE2 AND NOT(OP4) AND NOT(OP3) AND NOT(OP2) OR …

The rest of the next-state bits can be computed in a similar way.

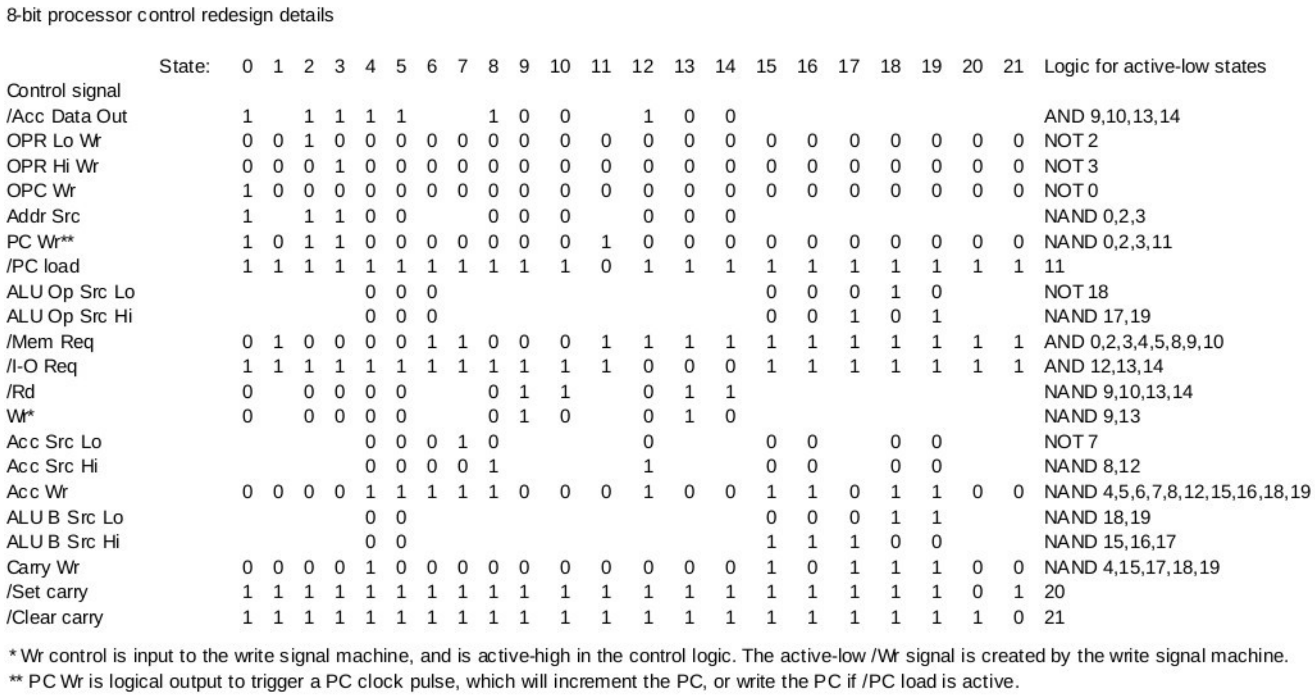
In this processor, the next-state logic is implemented on 3 programmable AND-OR array ICs, which are GAL16V8s. Next state logic can also be implemented with discrete logic ICs, or using a programmable ROM. If using a ROM, the state, opcode and flags become the input address, and the next-state outputs are the contents of the memory cells. This next-state logic could be implemented by a single 16K by 8-bit EPROM.

Control signals logic explanation

The control signals logic in this processor takes as its only input the current state. The output of the control logic is a set of 0 or 1 levels on the processor control lines. These lines are the multiplexer data select addresses, the register write and program counter increment lines, and signals to create the control signals for the system memory and input/output ports.

All outputs are created as logic levels, that is, 0s or 1s. The multiplexer address inputs can use these levels directly. However, the processor registers require upgoing edges to be written. The control logic board has a register write pulse machine that creates the edges from the control logic level outputs. Also, the write signals for system memory and output ports require a timing sequence that is carried over two states, and there is a write signal machine that carries this out.

Here is a figure that summarizes the control logic:



8-bit processor control redesign details

State:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	Logic for active-low states
Control signal																							
/Acc Data Out	1		1	1	1	1			1	0	0		1	0	0								AND 9,10,13,14
OPR Lo Wr	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOT 2
OPR Hi Wr	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOT 3
OPC Wr	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOT 0
Addr Src	1		1	1	0	0			0	0	0		0	0	0								NAND 0,2,3
PC Wr**	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	NAND 0,2,3,11
/PC load	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	11
ALU Op Src Lo					0	0	0									0	0	0	1	0			NOT 18
ALU Op Src Hi					0	0	0									0	0	1	0	1			NAND 17,19
/Mem Req	0	1	0	0	0	0	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	AND 0,2,3,4,5,8,9,10
/I-O Req	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	AND 12,13,14
/Rd	0		0	0	0	0			0	1	1		0	1	1								NAND 9,10,13,14
Wr*	0		0	0	0	0			0	1	0		0	1	0								NAND 9,13
Acc Src Lo					0	0	0	1	0				0			0	0		0	0			NOT 7
Acc Src Hi					0	0	0	0	1				1			0	0		0	0			NAND 8,12
Acc Wr	0	0	0	0	1	1	1	1	1	0	0	0	1	0	0	1	1	0	1	1	0	0	NAND 4,5,6,7,8,12,15,16,18,19
ALU B Src Lo					0	0										0	0	0	1	1			NAND 18,19
ALU B Src Hi					0	0										1	1	1	0	0			NAND 15,16,17
Carry Wr	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	NAND 4,15,17,18,19
/Set carry	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	20
/Clear carry	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	21

* Wr control is input to the write signal machine, and is active-high in the control logic. The active-low /Wr signal is created by the write signal machine.

** PC Wr is logical output to trigger a PC clock pulse, which will increment the PC, or write the PC if /PC load is active.

The one-bit control signal lines are in the column on the left. The processor states are in the row across the top. To create the control signal logic, I started with the above table, with all the cells empty, and went across state-by-state, filling in the 1s and 0s as appropriate for each state. Once filled in, for each control signal line, there is now a series of 1s and 0s in its table row that show what that line should be for each state. If it does not matter whether the control line is 1 or 0 for a particular state, that cell is left blank (that is, a “don’t care”).

For example, look at state 0, the instruction fetch state. There are 15 control lines that need to be set to 0 or 1 for this state to perform its function.

First, there are a number of register write signals that must be de-asserted (made inactive – that is, “don’t write this register in this state”). They are write signals for the instruction operands (OPR Lo Wr and OPR Hi Wr), system memory (Wr), the accumulator (Acc Wr), and the carry flip-flop (Carry Wr). Other de-asserted signals are /Acc Data Out (places accumulator data on the data bus – don’t what this when reading data from memory), /PC load (used in jump instructions), /I-O Req, /Set carry and /Clear carry.

Second, there are the asserted control signals OPC Wr (opcode write – this is what we are fetching from the memory), PC Wr (increments the program counter at the end of the cycle), /Mem Req, and /Rd, to tell the system memory to put data on the data bus – the opcode byte that is being fetched.

Third, there are the multiplexer address inputs that move data to the places needed to carry out the fetch. In state 0, the only one we care about is Addr Src (address source). This control line selects the address to put on the system address bus. If Addr Src is 0, the address comes from the current instruction. If it is 1, the address comes from the program counter. So, in an instruction fetch, we want the address to come from the program counter, so Addr Src needs to be 1.

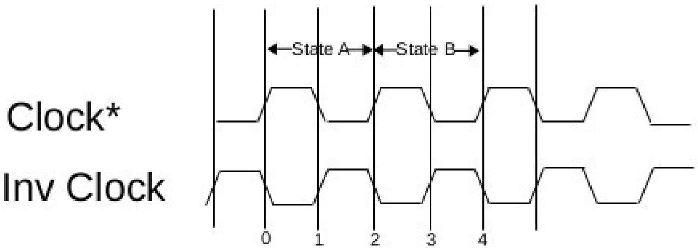
The logic operations that determine the value of each control line are in the far right column. The current state inputs to the control logic are active-low outputs from the state decoder (see the control board schematic). That is, if the current state is 5, the decoder output for state 5 will be 0, and all the other output lines will be 1.

Each control line is the one-bit result of the logic operation in the far right column. For example, the active-low /Acc Data Out signal is the result of a four-input AND operation of states 9, 10, 13 and 14. That is, /Acc Data Out will be one (inactive) only when all four states 9, 10, 13 and 14 are 1. If any of these 4 states is the current state (is zero), the output of the AND operation will be 0, and the /Acc Data Out signal will be zero (active). The next row, for the OPR Lo Wr signal, the logic operation is just the inverse of the state 2 input. That is, when state 2 is the current state, the state 2 output from the state decoder will be 0, NOT state 2 will be 1, and OPR Lo Wr will be 1. In all other cases (all other states) the result of NOT 2 will be 0.

Each equation for each control signal is implemented with simple logic gates and inverters. For some signals, such as /PC load the state itself is sufficient.

System write signal timing

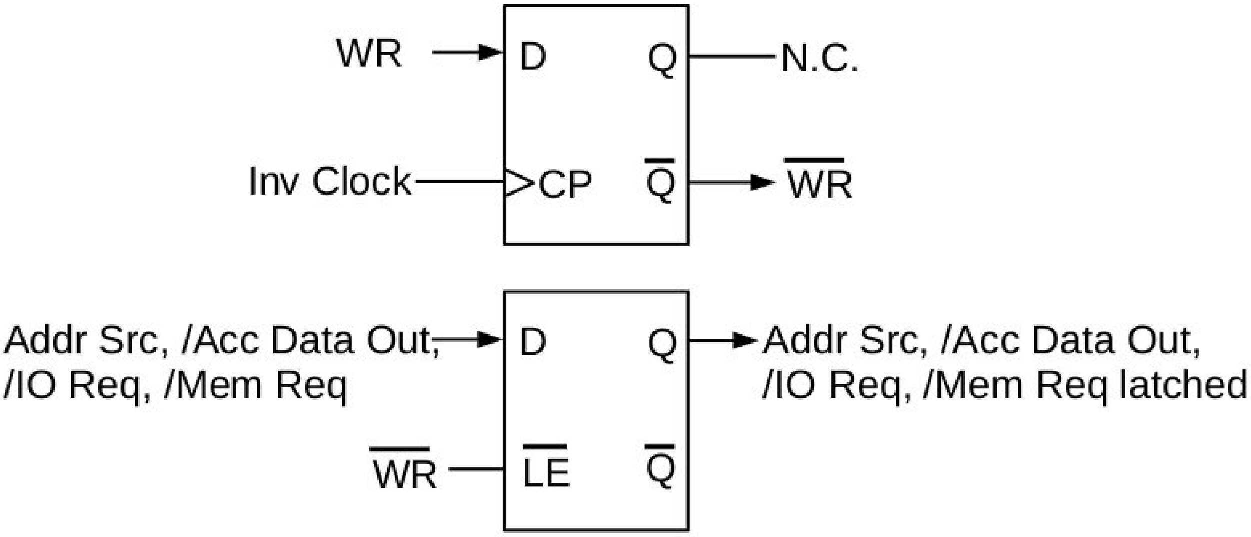
This figure summarizes the timing of the system write signal machine:



Edge 0: WR from control logic asserted to FF data input
IO Req or Mem Req asserted
Edge 1: WR written and output from FF as $\overline{\text{WR}}$
Edge 2: WR from control logic de-asserted
Edge 3: $\overline{\text{WR}}$ from FF de-asserted
Edge 4: IO Req or Mem Req de-asserted

Addr src, /Acc Data Out, /IO Req or /Mem Req must remain stably asserted throughout states A and B. Can latch these using $\overline{\text{WR}}$ as the latch enable.

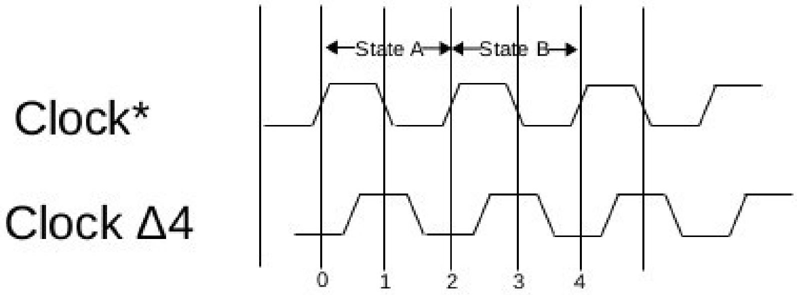
* Clock is synchronous with register write clock pulses



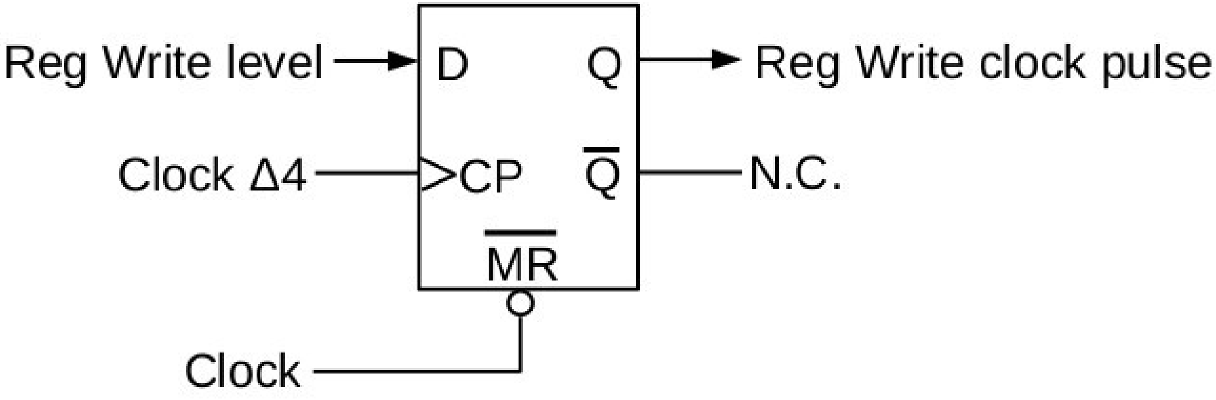
Shown here is the flip-flop that creates the /WR signal (active-low) from the control logic WR level output (active-high) and the inverted clock signal. Also shown is the latch that uses the /WR signal as the latch enable for the control signals that need to be held steady over the two states that are used to write to memory or output ports. In the timing diagram, the various edges listed are the upgoing edges of the clock signals. That is, edge 0 is the upgoing edge in the Clock signal, edge 1 is the upgoing edge in the Inv Clock signal, etc. The overall reason for this machine is to provide a clean write signal to memory or output ports. This is done by having the memory request or input-output request signal asserted and stable before the write signal is asserted, and the write signal de-asserted well before the memory request or input-output request signal is de-asserted. The write signal needs to be asserted for some minimum time depending on the type of memory or output port IC used. This scheme will have the write signal asserted from edge 1 to edge 3, a full cycle time (500 nanoseconds at 2 MHz). This is adequate for most ICs used with simple 8-bit systems.

Register write pulse timing

This diagram shows how the register write pulses are created from the register write machine flip-flops:



* Clock for this machine is primary oscillator



Clock Δ4 is created by putting Clock through a series of 4 inverters. The delay is enough to assure that the Clock level has returned to Hi (that is, the reset has finished) before Clock Δ4 upgoing edge arrives to write the register. When this register is written, the output goes to write the target register in the data path.

Special Programming Techniques

The CPUville 8-bit processor has a very simple architecture. This makes the processor easy to build and understand, but makes programming more difficult. Especially, the processor lacks registers and instructions to perform address indexing and stack operations.

Indexing

To perform address indexing, one needs to place the load or store instruction with the address to be indexed in RAM and increment or decrement the address portion of the instruction as required. If the instruction is placed away from the rest of the code, as it would need to be if the main program code is in ROM, it needs to be followed by a jump instruction to return to the main code.

Here is an example taken from the system monitor program. This program is in ROM, so the instructions to be indexed need to be in RAM. Locations 0x0800 to 0x08FF in RAM are reserved for system monitor variables, including indexed store and load instructions.

First, a place for the RAM instruction must be created as one would for any other variable. The variables are in the assembly language file for the ROM system monitor, at the end after an `.org 0800h` pseudo-operation that places them in the lowest page of RAM:

```
;The following section contains labels for RAM variables and other structures
                  .org  0800h       ;Start of RAM
;RAM Variables
dp_value          .dw   0000h
dp_10000s         .db   00h
.
.
.
;Indexed store for load routine, must be in RAM
ld_indexed_stm    stm   0000h
                  jmp   ld_stm_back
```

Here we see a location established for a store instruction with the label `ld_indexed_stm,` followed by a place for a jump instruction with the operand `ld_stm_back.` The `ld_indexed_stm` label is here in order for the assembler to make the code that uses it, but these locations in RAM will just contain garbage when the computer is powered up. The ROM code will have to initialize these locations with the `stm` and `jmp` opcodes and the address operands before these instructions can be used.

Here is the section in the ROM that initializes the RAM instruction opcodes:

```
;Opcode initialization for RAM instructions
                  ldi   13h             ;jmp opcode
                  stm   ld_indexed_stm+3 ;return jumps for indexed instructions
.
.
.
                  ldi   12h             ;stm opcode
                  stm   ld_indexed_stm   ;indexed store instructions
```

Here is the section in ROM that initializes the address operand of the RAM store instruction. The address is obtained from the `address` variable:

```
                    ldm   address
                    stm   ld_indexed_stm+1
                    ldm   address+1
                    stm   ld_indexed_stm+2
```

Here is the section in ROM that initializes the address operand of the return `jmp` instruction. This section of code needs to be written after a prior assembly, in order to know the target address, which will be found by examining the the assembly list file:

```
                    ldi   00h              ;address of ld_stm_back
                    stm   ld_indexed_stm+4
                    ldi   03h
                    stm   ld_indexed_stm+5
```

Here is the section in the ROM that uses and indexes the store instruction:

```
                    JMP   ld_indexed_stm   ;Store the byte in RAM
ld_stm_back:        LDM   ld_indexed_stm+1 ;Increment byte pointer, lo byte first
                    INC
                    STM   ld_indexed_stm+1
                    LDM   ld_indexed_stm+2 ;Increment hi byte if a carry occurred
when lo byte incremented
                    ADCIM 00H
                    STM   ld_indexed_stm+2
```

The `JMP ld_indexed_stm` instruction performs the store operation by jumping to the store instruction in RAM. The RAM `jmp ld_stm_back` instruction after the indexed `stm` instruction returns program flow to the code in ROM. Then, the address operand of the RAM `stm` instruction is incremented by a 16-bit addition operation.

Subroutines

Calling subroutines in most processors is done with a stack operation. The instruction that calls the subroutine pushes the program counter onto the stack, and after the subroutine is finished, the return instruction pops the address off the stack and places it in the program counter. These instructions, and others like push and pop, use a stack pointer stored in a processor register.

The CPUville 8-bit processor lacks a stack pointer register, and does not have the associated instructions in its instruction set. Instead, the programmer must develop a means to call subroutines using software.

While a full stack system could be implemented, in simple programs with minimal nesting of subroutine calls, a simple system of macros for calls and returns will suffice. This is the system I used when I wrote the 8-bit processor system monitor, and the pi calculation program.

TASM has the pseudo-operation `.set` that allows the programmer to create labels that can be assigned values by assembly language instructions. This allows the call and return macros to be created easily.

The system monitor program has at most 3 levels of subroutine nesting. Without creating a true stack pointer system and stack, I simply created call and return macros for three levels. Here is the code:

```
;Macro definitions for three levels of nested call and ret
#define call0(address) \return: .set $+15\ ldm return\ stm return_jump0+1\ ldm
return+1\ stm return_jump0+2
#defcont \ jmp address\ .dw $+2
#define ret0 \ jmp return_jump0
#define call1(address) \return: .set $+15\ ldm return\ stm return_jump1+1\ ldm
return+1\ stm return_jump1+2
#defcont \ jmp address\ .dw $+2
#define ret1 \ jmp return_jump1
#define call2(address) \return: .set $+15\ ldm return\ stm return_jump2+1\ ldm
return+1\ stm return_jump2+2
#defcont \ jmp address\ .dw $+2
#define ret2 \ jmp return_jump2
```

TASM macros are defined by using the `#define` directive, followed by the macro definition, followed by the macro code statements, separated by \ characters. Additional lines of macro code can be added using the `#defcont` directive.

The `call` macro does two things. It sets up the return address of the subroutine call, and jumps to the subroutine. The `ret` macro causes a jump to the return address. Each call and return macro has a level associated with it. The call macro for a certain level will create the return address for the return macro to use for that level. So, `call0` creates the return address for `ret0`, `call1` creates the return address for `ret1`, etc.

The first macro definition is `call0(address)`. The macro operand `address` is the address label of the subroutine that is being called.

The call macro has 7 statements:

`return: .set $+15` sets the label `return:` to the 16-bit value of the return address for `call0`, which the assembler calculates after the code has been assembled. `$+15` points to the address data placed in the code by the `.dw $+2` pseudo-operation at the end of the macro definition. The label `return:` must be defined before the first use of a call macro. In the ROM monitor code `return:` is defined at the start of the code, where it refers to a no-operation placeholder instruction:

```
                  .org  0000h
return:           .db   1fh   ;placeholder for first definition of variable label
-- NOP
```

`ldm return` loads the low-byte of the return address into the accumulator.

`stm return_jump0+1` stores the low byte of the return address in the RAM as the low byte of the return jump instruction for call level 0.

`ldm return+1` places the high byte of the return address into the accumulator.

`stm return_jump0+2` stores the high byte of the return address in the RAM as the high byte of the return jump instruction for call level 0.

`jmp address` is the instruction that jumps to the subroutine code.

`.dw $+2` is a pseudo-operation that places the 16-bit value of the current program location plus 2 into the code here. The current location plus 2 is the target of the return jump. The .dw $+2 value is in the location referenced by the .set $+15 pseudo-operation at the beginning of the macro.

The definition of the ret0 macro is simply jmp return_jump0. This jumps to a jump instruction in RAM with the return address operand that has been placed in it by the call0 macro.

The return: label must be reset to its original value after the macros have been placed in the code, otherwise the assembler returns a syntax error. I reset the return: label at the end of the code, like this:

```
return:     .set  0000h ;assembler needs variable label set back to original value
            .end
```

To use the call and return macros one simply uses them like one would use the call and return instructions in a more complex processor. Here is an example from the system monitor code:

```
;Print greeting
sm_lukewarm       ldm   sm_greeting
                  stm   ws_inst+1
                  ldm   sm_greeting+1
                  stm   ws_inst+2
                  call0(write_string)
```

This section sets up the address of a string to be displayed by the write_string subroutine, then calls it. The write_string subroutine uses a ret0 macro to return, so it needs to be called as a level 0 subroutine.

This scheme allows for easy addition of more call levels, by adding a new call and ret macros, with the corresponding jmp return_jump variable instruction in RAM.

The main drawback of this scheme is that the programmer must keep track of which call levels are currently active, to avoid calling a level that is already in use. This would over-write the return address for the original call with a new address. The levels are not ordered, so a call2 subroutine can perform a call0, and a call0 can perform a call2. But, a call2 cannot perform a call2. So, true recursion is not allowed by this scheme. But it is enough for writing code with many subroutines.

If you have developed a scheme for true recursion, please let me know, and I will post it on the CPUville website.

Instruction set table, sorted by opcode

Opcode (hex)	Mnemonic	Function	Inst. length, bytes	Clock cycles	Flags affected
00	ADD	Add memory to accumulator	3	5	C, Z, M
01	ADC	Add memory and carry to accumulator	3	5	C, Z, M
02	SUB	Subtract memory from accumulator	3	5	C, Z, M
03	SBB	Subtract memory and borrow from accumulator	3	5	C, Z, M
04	AND	Binary AND memory with accumulator	3	5	Z, M
05	OR	Binary OR memory with accumulator	3	5	Z, M
06	XOR	Binary XOR memory with accumulator	3	5	Z, M
07	NOT	Complement accumulator	1	3	Z, M
08	ADDIM	Add 8-bit value in instruction to accumulator	2	4	C, Z, M
09	ADCIM	Add 8-bit value in instruction and carry to accumulator	2	4	C, Z, M
0A	SUBIM	Subtract 8-bit value in instruction from accumulator	2	4	C, Z, M
0B	SBBIM	Subtract 8-bit value in instruction and borrow from accumulator	2	4	C, Z, M
0C	ANDIM	Binary AND 8-bit value in instruction with accumulator	2	4	Z, M
0D	ORIM	Binary OR 8-bit value in instruction with accumulator	2	4	Z, M
0E	XORIM	Binary XOR 8-bit value in instruction with accumulator	2	4	Z, M
0F	CMP	Subtract 8-bit value in instruction from accumulator, set carry only	2	4	C
10	LDI	Load 8-bit value in instruction into accumulator	2	4	Z, M
11	LDM	Load accumulator from memory	3	5	Z, M
12	STM	Store accumulator into memory	3	6	
13	JMP	Jump to memory location	3	5	
14	JPZ	Jump to memory location if zero	3	4 or 5	
15	JPM	Jump to memory location if minus	3	4 or 5	
16	JPC	Jump to memory location if carry	3	4 or 5	
17	IN	Load accumulator with 8-bit value from port	2	4	Z, M
18	OUT	Send 8-bit value from accumulator to port	2	5	
19	INC	Add 1 to accumulator	1	3	C, Z, M
1A	DEC	Subtract 1 from accumulator	1	3	C, Z, M
1B	SCF	Set carry flag	1	3	C
1C	CCF	Clear carry flag	1	3	C
1D		not implemented			
1E		not implemented			
1F	NOP	No operation	1	2	

Instruction set table, sorted by mnemonic

Mnemonic	Opcode (hex)	Function	Inst. length, bytes	Clock cycles	Flags affected
ADC	01	Add memory and carry to accumulator	3	5	C, Z, M
ADCIM	09	Add 8-bit value in instruction and carry to accumulator	2	4	C, Z, M
ADD	00	Add memory to accumulator	3	5	C, Z, M
ADDIM	08	Add 8-bit value in instruction to accumulator	2	4	C, Z, M
AND	04	Binary AND memory with accumulator	3	5	Z, M
ANDIM	0C	Binary AND 8-bit value in instruction with accumulator	2	4	Z, M
CCF	1C	Clear carry flag	1	3	C
CMP	0F	Subtract 8-bit value in instruction from accumulator, set carry only	2	4	C
DEC	1A	Subtract 1 from accumulator	1	3	C, Z, M
IN	17	Load accumulator with 8-bit value from port	2	4	Z, M
INC	19	Add 1 to accumulator	1	3	C, Z, M
JMP	13	Jump to memory location	3	5	
JPC	16	Jump to memory location if carry	3	4 or 5	
JPM	15	Jump to memory location if minus	3	4 or 5	
JPZ	14	Jump to memory location if zero	3	4 or 5	
LDI	10	Load 8-bit value in instruction into accumulator	2	4	Z, M
LDM	11	Load accumulator from memory	3	5	Z, M
NOP	1F	No operation	1	2	
NOT	07	Complement accumulator	1	3	Z, M
OR	05	Binary OR memory with accumulator	3	5	Z, M
ORIM	0D	Binary OR 8-bit value in instruction with accumulator	2	4	Z, M
OUT	18	Send 8-bit value from accumulator to port	2	5	
SBB	03	Subtract memory and borrow from accumulator	3	5	C, Z, M
SBBIM	0B	Subtract 8-bit value in instruction and borrow from accumulator	2	4	C, Z, M
SCF	1B	Set carry flag	1	3	C
STM	12	Store accumulator into memory	3	6	
SUB	02	Subtract memory from accumulator	3	5	C, Z, M
SUBIM	0A	Subtract 8-bit value in instruction from accumulator	2	4	C, Z, M
XOR	06	Binary XOR memory with accumulator	3	5	Z, M
XORIM	0E	Binary XOR 8-bit value in instruction with accumulator	2	4	Z, M
	1D	not implemented			
	1E	not implemented			

Selected Program Listings

ROM for 4K systems

```
0000             ;Test programs for 8-bit TTL computer.
0000                              .ORG  0000H ;Get address from switches and jump to it
0000 10 13                        LDI   13H   ;JMP instruction
0002 12 00 08                     STM   0800H ;Start of RAM
0005 17 00                        IN    00H   ;Low byte of jump target
0007 12 01 08                     STM   0801H
000A 17 01                        IN    01H   ;High byte of jump target
000C 12 02 08                     STM   0802H ;Full jump instruction in place now
000F 13 00 08                     JMP   0800H ;Jump to the jump instruction
0012             ;Simple port reflector
0012 17 00       LOOP:            IN    00H
0014 18 00                        OUT   00H
0016 17 01                        IN    01H
0018 18 01                        OUT   01H
001A 13 12 00                     JMP   LOOP
001D             ;Simple counter -- run with slow clock
001D 10 00                        LDI   00H
001F 18 00       LOOPA:           OUT   00H
0021 19                           INC
0022 13 1F 00                     JMP   LOOPA
0025             ;Two-byte up counter -- run with fast clock
0025 10 00                        LDI   00H
0027 12 00 08                     STM   0800H ;Hi byte
002A 18 01                        OUT   01H   ;Clear port 1 LEDs
002C 18 00       LOOPB:           OUT   00H   ;Output low byte
002E 19                           INC
002F 14 35 00                     JPZ   NEXTB ;If zero, jump to increment high byte
0032 13 2C 00                     JMP   LOOPB ;Low-byte increment loop
0035 11 00 08    NEXTB:           LDM   0800H ;Get high byte from memory
0038 19                           INC
0039 18 01                        OUT   01H   ;Output high byte to port 1 LEDs
003B 12 00 08                     STM   0800H ;Store high byte
003E 10 00                        LDI   00H
0040 13 2C 00                     JMP   LOOPB ;Go back to low-byte increment loop
0043             ;8-bit highest factor routine
```

```
0043             ;Factor test
0043 17 00       FACSTRT:IN       00H   ;Number to factor
0045 12 0A 08                     STM   ORIG
0048 12 0B 08                     STM   TESTF
004B 11 0B 08    LOOP15:          LDM   TESTF
004E 1A                           DEC
004F 12 0B 08                     STM   TESTF
0052 11 0A 08                     LDM   ORIG
0055 02 0B 08    LOOP16:          SUB   TESTF
0058 14 61 00                     JPZ   DONE  ;Factor found
005B 16 55 00                     JPC   LOOP16      ;For A - B, carry set if A >= B
005E 13 4B 00                     JMP   LOOP15      ;No carry, means A < B, and not a factor
0061 11 0B 08    DONE:            LDM   TESTF
0064 18 00                        OUT   00H
0066 13 43 00                     JMP   FACSTRT
0069             ;Test for serial interface with 8-bit code
0069             ;Set port to 9600 baud, 8-bit, no parity, 1 stop bit
0069 10 4E                        LDI   4EH         ;1 stop bit, no parity, 8-bit char, 16x baud
006B 18 03                        OUT   03H         ;write to UART control port
006D 10 37                        LDI   37H         ;enable receive and transmit
006F 18 03                        OUT   03H         ;write to control port
0071 17 03       LOOP1:           IN    03H         ;get status
0073 0C 02                        ANDIM 02H         ;check RxRDY bit
0075 14 71 00                     JPZ   LOOP1       ;not ready, loop
0078 17 02                        IN    02H         ;get char from data port
007A 18 00                        OUT   00H         ;put on LEDs
007C 12 06 08                     STM   TEMP        ;store the character
007F 17 03       LOOP2:           IN    03H         ;get status
0081 0C 01                        ANDIM 01H         ;check TxRDY bit
0083 14 7F 00                     JPZ   LOOP2       ;loop if not ready
0086 11 06 08                     LDM   TEMP        ;get char back
0089 18 02                        OUT   02H         ;send to UART for output
008B 13 71 00                     JMP   LOOP1       ;start over
008E             ;Program loader
008E             ;Takes input from serial port, creates byte values from hex character pairs
008E             ;Loads byte values sequentially into RAM starting at 0x0810
008E             ;Jumps to location 0x0810 to start execution upon receiving return character
008E             ;Quits without execution if invalid hex character input received
008E             ;Setup routine for serial port
008E 10 4E                        LDI   4EH         ;1 stop bit, no parity, 8-bit char, 16x baud
```

```
0090 18 03                        OUT   03H         ;write to UART control port
0092 10 37                        LDI   37H         ;enable receive and transmit
0094 18 03                        OUT   03H         ;write to control port
0096             ;Need to put instruction to store bytes in RAM so can increment the target address
0096 10 12                        LDI   12H         ;STM instruction
0098 12 00 08                     STM   STORE_BYTE
009B 10 10                        LDI   10H         ;Low byte of storage buffer start address
009D 12 01 08                     STM   STORE_BYTE+1
00A0 10 08                        LDI   08H         ;Hi byte of storage buffer start address
00A2 12 02 08                     STM   STORE_BYTE+2
00A5             ;Need to set return jump after STORE_BYTE
00A5 10 13                        LDI   13H         ;JMP instruction for return
00A7 12 03 08                     STM   STORE_BYTE+3
00AA 11 9E 01                     LDM   RETURN      ;Lo byte of return address
00AD 12 04 08                     STM   STORE_BYTE+4
00B0 11 9F 01                     LDM   RETURN+1    ;Hi byte of return address
00B3 12 05 08                     STM   STORE_BYTE+5
00B6 10 10                        LDI   16
00B8 12 09 08                     STM   BYTE_COUNTER      ;initialize line length variable
00BB 17 03       GET_HI:          IN    03H         ;Get hi-order nybble of pair
00BD 0C 02                        ANDIM 02H         ;check RxRDY bit
00BF 14 BB 00                     JPZ   GET_HI      ;not ready, loop
00C2 17 02                        IN    02H         ;get char from data port
00C4 12 06 08                     STM   TEMP        ;Store character
00C7 0A 0D                        SUBIM 0DH         ;Carriage return?
00C9 14 B2 01                     JPZ   RUN         ;Yes, run code
00CC 17 03       LOOP3:           IN    03H         ;No, output character and validate
00CE 0C 01                        ANDIM 01H         ;check TxRDY bit
00D0 14 CC 00                     JPZ   LOOP3       ;loop if not ready
00D3 11 06 08                     LDM   TEMP        ;get char back
00D6 18 02                        OUT   02H         ;send to UART for output
00D8             ;Code to validate hex character
00D8 0F 30                        CMP   30H         ;Lower limit of hex characters
00DA 16 E0 00                     JPC   NEXT1       ;Char >= 30H, possibly valid
00DD 13 FE 00                     JMP   INVALID     ;Char < 30H, invalid hex char
00E0 0F 47       NEXT1:           CMP   47H         ;ASCII for "G"
00E2 16 FE 00                     JPC   INVALID     ;Char is G or greater, invalid
00E5 0F 41                        CMP   41H         ;ASCII for "A"
00E7 16 F2 00                     JPC   VALIDAF_HI  ;Char is valid A-F
00EA 0F 3A                        CMP   3AH         ;ASCII for ":"
```

```
00EC 16 FE 00                   JPC   INVALID     ;Char is ":" or greater, but < "A", invalid
00EF 13 F9 00                   JMP   VALID09_HI  ;Char is valid 0-9
00F2 0C 0F      VALIDAF_HI:     ANDIM 0FH         ;Mask off high bits
00F4 08 09                      ADDIM 9           ;Adjust ASCII to binary value
00F6 13 01 01                   JMP   SHIFT_HI
00F9 0C 0F      VALID09_HI:     ANDIM 0FH         ;Mask off high bits
00FB 13 01 01                   JMP   SHIFT_HI
00FE 13 B5 01   INVALID:        JMP   ERROR       ;Invalid hex char, quit
0101 12 07 08   SHIFT_HI:       STM   BYTE        ;Will eventually contain the byte to load
0104 12 06 08                   STM   TEMP        ;Value to add
0107 10 10                      LDI   10H         ;Multiply x 16 to shift into high-order nybble
0109 12 08 08                   STM   COUNTER
010C 11 08 08   MULTLOOP:       LDM   COUNTER
010F 1A                         DEC
0110 14 22 01                   JPZ   GET_LO      ;Have added 16 times, done
0113 12 08 08                   STM   COUNTER
0116 11 06 08                   LDM   TEMP        ;Original nybble
0119 00 07 08                   ADD   BYTE        ;Add to BYTE and store
011C 12 07 08                   STM   BYTE
011F 13 0C 01                   JMP   MULTLOOP    ;Keep adding
0122 17 03      GET_LO:         IN    03H         ;Get lo-order nybble of pair
0124 0C 02                      ANDIM 02H         ;check RxRDY bit
0126 14 22 01                   JPZ   GET_LO      ;not ready, loop
0129 17 02                      IN    02H         ;get char from data port
012B 12 06 08                   STM   TEMP        ;Store character
012E 17 03      LOOP4:          IN    03H         ;Output character
0130 0C 01                      ANDIM 01H         ;check TxRDY bit
0132 14 2E 01                   JPZ   LOOP4
0135 11 06 08                   LDM   TEMP        ;When ready, retrieve character and output
0138 18 02                      OUT   02H
013A 17 03      LOOP5:          IN    03H
013C 0C 01                      ANDIM 01H
013E 14 3A 01                   JPZ   LOOP5
0141 10 20                      LDI   20H         ;Space character
0143 18 02                      OUT   02H         ;send to UART for output
0145            ;Check if 16 bytes have been displayed. If so, write newline
0145 11 09 08                   LDM   BYTE_COUNTER      ;Check if 16 bytes have been displayed
0148 1A                         DEC
0149 12 09 08                   STM   BYTE_COUNTER
014C 14 52 01                   JPZ   NEXT4       ;Yes, reset counter and write newline
```

```
0156   014F 13 6D 01                     JMP   NEXT5       ;No, keep going
0157   0152 10 10       NEXT4:           LDI   16
0158   0154 12 09 08                     STM   BYTE_COUNTER
0159   0157 17 03       LOOP6:           IN    03H
0160   0159 0C 01                        ANDIM 01H
0161   015B 14 57 01                     JPZ   LOOP6
0162   015E 10 0D                        LDI   0DH         ;Return character
0163   0160 18 02                        OUT   02H         ;send to UART for output
0164   0162 17 03       LOOP7:           IN    03H
0165   0164 0C 01                        ANDIM 01H
0166   0166 14 62 01                     JPZ   LOOP7
0167   0169 10 0A                        LDI   0AH         ;Linefeed character
0168   016B 18 02                        OUT   02H         ;send to UART for output
0169   016D             ;Code to validate hex character
0170   016D 11 06 08    NEXT5:           LDM   TEMP        ;Retrieve character and validate
0171   0170 0F 30                        CMP   30H         ;Lower limit of hex characters
0172   0172 16 78 01                     JPC   NEXT2       ;Char >= 30H, possibly valid
0173   0175 13 FE 00                     JMP   INVALID     ;Char < 30H, invalid hex char
0174   0178 0F 47       NEXT2:           CMP   47H         ;ASCII for "G"
0175   017A 16 FE 00                     JPC   INVALID     ;Char is G or greater, invalid
0176   017D 0F 41                        CMP   41H         ;ASCII for "A"
0177   017F 16 8A 01                     JPC   VALIDAF_LO  ;Char is valid A-F
0178   0182 0F 3A                        CMP   3AH         ;ASCII for ":"
0179   0184 16 FE 00                     JPC   INVALID     ;Char is ":" or greater, but < "A", invalid
0180   0187 13 94 01                     JMP   VALID09_LO  ;Char is valid 0-9
0181   018A 0C 0F       VALIDAF_LO:      ANDIM 0FH         ;Mask off high bits
0182   018C 08 09                        ADDIM 9           ;Now lo nybble correct
0183   018E 00 07 08                     ADD   BYTE        ;Combine with hi nybble stored in BYTE
0184   0191 13 99 01                     JMP   STORE       ;Store the byte in RAM
0185   0194 0C 0F       VALID09_LO:      ANDIM 0FH         ;Mask off high bits
0186   0196 00 07 08                     ADD   BYTE        ;Now full byte assembled
0187   0199 18 00       STORE:           OUT   00H         ;Display on LEDs
0188   019B 13 00 08                     JMP   STORE_BYTE  ;Store the byte in RAM
0189   019E A0 01       RETURN:          .DW   $+2         ;Address to return from storage instruction
0190   01A0 11 01 08                     LDM   STORE_BYTE+1      ;Increment byte pointer, lo byte first
0191   01A3 19                           INC
0192   01A4 12 01 08                     STM   STORE_BYTE+1
0193   01A7 11 02 08                     LDM   STORE_BYTE+2      ;Increment hi byte if a carry occurred when lo
byte incremented
0194   01AA 09 00                        ADCIM 00H
```

```
0195   01AC 12 02 08                    STM   STORE_BYTE+2
0196   01AF 13 BB 00                    JMP   GET_HI
0197   01B2 13 10 08    RUN:            JMP   0810H        ;Run program
0198   01B5 18 00       ERROR:          OUT   00H          ;Display erroneous character on LEDs
0199   01B7 13 B7 01    HALT:           JMP   HALT         ;Halt
0200   0800                             .ORG  0800H
0201   0800 000000000000STORE_BYTE:     .DB 0,0,0,0,0,0   ;Six spaces for storage instruction and return
0202   0806 00          TEMP:           .DB   00H          ;Temp storage for character, data
0203   0807 00          BYTE:           .DB   00H          ;For multiplication (shifting)
0204   0808 00          COUNTER:        .DB   00H          ;For multiplication (shifting)
0205   0809 00          BYTE_COUNTER    .DB   00H          ;For length of display line
0206   080A 00          ORIG:           .DB   00H
0207   080B 00          TESTF:          .DB   00H
0208   080C                             .END
tasm: Number of errors = 0
```

adder

```
0001   0000             ;Simple byte adder program
0002   0000             ;Adds bytes on input port switches
0003   0000             ;Displays output on LEDs
0004   0810                  .ORG  0810H ;Location in RAM where program 4K loader places code
0005   0810 17 00       LOOP: IN    00H   ;Get first byte from right-hand switches
0006   0812 12 25 08         STM   TEMP  ;Store byte in RAM
0007   0815 17 01            IN    01H   ;Get second byte from left-hand switches
0008   0817 00 25 08         ADD   TEMP  ;Add the bytes
0009   081A 18 00            OUT   00H   ;lower 8-bits of sum to right-hand LEDs
0010   081C 10 00            LDI   00H   ;load accumulator with zero
0011   081E 09 00            ADCIM 00H   ;Add carry to zero
0012   0820 18 01            OUT   01H   ;upper 8-bits of sum to left-hand LEDs
0013   0822 13 10 08         JMP   LOOP  ;do it again
0014   0825 00          TEMP: .DB   00H   ;Location of TEMP variable
0015   0826                  .END        ;End of code
0016   0826
tasm: Number of errors = 0
```

ROM System Monitor

```
0001   0000             ;ROM system monitor
0002   0000             ;Macro definitions for three levels of nested call and ret
0003   0000             #define call0(address) \return: .set $+15\ ldm return\ stm return_jump0+1\ ldm return+1\
stm return_jump0+2
0004   0000             #defcont \ jmp address\ .dw $+2
0005   0000             #define ret0 \ jmp return_jump0
0006   0000             #define call1(address) \return: .set $+15\ ldm return\ stm return_jump1+1\ ldm return+1\
stm return_jump1+2
0007   0000             #defcont \ jmp address\ .dw $+2
0008   0000             #define ret1 \ jmp return_jump1
0009   0000             #define call2(address) \return: .set $+15\ ldm return\ stm return_jump2+1\ ldm return+1\
stm return_jump2+2
0010   0000             #defcont \ jmp address\ .dw $+2
0011   0000             #define ret2 \ jmp return_jump2
0012   0000
0013   0000             ;Buffer location defined by these constant values
0014   0000             ;Needs to be in RAM above variables and variable instructions
0015   0000             buff_low:   .equ  80h   ;low byte of buffer address
0016   0000             buff_high:  .equ  08h   ;high byte of buffer address
0017   0000             buffer:     .equ  0880h ;two-byte address constant
0018   0000                         .org  0000h
0019   0000
0020   0000 1F          return:     .db   1fh   ;placeholder for first definition of variable label -- NOP
0021   0001
0022   0001             ;Initialize port
0023   0001 10 4E                   LDI   4EH   ;1 stop bit, no parity, 8-bit char, 16x baud
0024   0003 18 03                   OUT   03H   ;write to UART control port
0025   0005 10 37                   LDI   37H   ;enable receive and transmit
0026   0007 18 03                   OUT   03H   ;write to control port
0027   0009
0028   0009             ;Opcode initialization for RAM instructions
0029   0009 10 13                   ldi   13h               ;jmp opcode
0030   000B 12 1E 08                stm   ld_indexed_stm+3  ;return jumps for indexed instructions
0031   000E 12 24 08                stm   d_indexed_ldm+3
0032   0011 12 2A 08                stm   d_indexed_stm+3
0033   0014 12 30 08                stm   bl_indexed_stm+3
0034   0017 12 36 08                stm   ws_inst+3
0035   001A 12 3C 08                stm   gl_indexed_stm+3
```

```
001D 12 3F 08               stm   return_jump0            ;other variable jumps
0020 12 42 08               stm   return_jump1
0023 12 45 08               stm   return_jump2
0026 12 18 08               stm   run_jump
0029 10 12                  ldi   12h               ;stm opcode
002B 12 1B 08               stm   ld_indexed_stm          ;indexed store instructions
002E 12 27 08               stm   d_indexed_stm
0031 12 2D 08               stm   bl_indexed_stm
0034 12 39 08               stm   gl_indexed_stm
0037 10 11                  ldi   11h               ;ldm opcode
0039 12 21 08               stm   d_indexed_ldm
003C 12 33 08               stm   ws_inst
003F 10 00                  ldi   00h               ;address of ld_stm_back
0041 12 1F 08               stm   ld_indexed_stm+4
0044 10 03                  ldi   03h
0046 12 20 08               stm   ld_indexed_stm+5
0049 10 59                  ldi   59h               ;address of d_ldm_back
004B 12 25 08               stm   d_indexed_ldm+4
004E 10 01                  ldi   01h
0050 12 26 08               stm   d_indexed_ldm+5
0053 10 78                  ldi   78h               ;address of d_stm_back
0055 12 2B 08               stm   d_indexed_stm+4
0058 10 01                  ldi   01h
005A 12 2C 08               stm   d_indexed_stm+5
005D 10 B2                  ldi   0b2h              ;address of bl_back
005F 12 31 08               stm   bl_indexed_stm+4
0062 10 04                  ldi   04h
0064 12 32 08               stm   bl_indexed_stm+5
0067 10 F1                  ldi   0f1h              ;address of ws_back
0069 12 37 08               stm   ws_inst+4
006C 10 05                  ldi   05h
006E 12 38 08               stm   ws_inst+5
0071 10 C0                  ldi   0C0h              ;address of gl_back
0073 12 3D 08               stm   gl_indexed_stm+4
0076 10 05                  ldi   05h
0078 12 3E 08               stm   gl_indexed_stm+5
007B
007B             ;Print greeting
007B 11 36 07    sm_lukewarmldm   sm_greeting
007E 12 34 08               stm   ws_inst+1
```

```
0076   0081 11 37 07               ldm   sm_greeting+1
0077   0084 12 35 08               stm   ws_inst+2
0078   0087                        call0(write_string)
0078   0087
0078   0087 11 96 00
0078   008A 12 40 08
0078   008D 11 97 00
0078   0090 12 41 08
0078   0093 13 E7 05
0078   0096 98 00
0079   0098
0080   0098             ;Warm start for system monitor, re-entry point after commands have finished
0081   0098             ;Prompt for routine number input
0082   0098 11 68 07    sm_warm    ldm   sm_prompt
0083   009B 12 34 08               stm   ws_inst+1
0084   009E 11 69 07               ldm   sm_prompt+1
0085   00A1 12 35 08               stm   ws_inst+2
0086   00A4                        call0(write_string)
0086   00A4
0086   00A4 11 B3 00
0086   00A7 12 40 08
0086   00AA 11 B4 00
0086   00AD 12 41 08
0086   00B0 13 E7 05
0086   00B3 B5 00
0087   00B5
0088   00B5             ;Get character and jump to monitor routine
0089   00B5 17 03       sm_chk_loop:     in    03h         ;get status
0090   00B7 0C 02                        andim 02h         ;check RxRDY
0091   00B9 14 B5 00                     jpz   sm_chk_loop
0092   00BC 17 02                        in    02h         ;get char from port and echo
0093   00BE 12 11 08                     stm   choice
0094   00C1 17 03       sm_echo_loop     in    03h
0095   00C3 0C 01                        andim 01h         ;check TxRDY
0096   00C5 14 C1 00                     jpz   sm_echo_loop
0097   00C8 11 11 08                     ldm   choice
0098   00CB 18 02                        out   02h
0099   00CD 0F 35                        cmp   '5'
0100   00CF 16 15 03                     jpc   sm_bload
0101   00D2 0F 34                        cmp   '4'
```

```
0102   00D4 16 09 02               jpc   sm_load
0103   00D7 0F 33                  cmp   '3'
0104   00D9 16 F7 01               jpc   sm_run
0105   00DC 0F 32                  cmp   '2'
0106   00DE 16 E4 00               jpc   sm_dump
0107   00E1 13 98 00               jmp   sm_warm           ;any number other than 2 to 5 results in warm restart
0108   00E4
0109   00E4             ;Memory dump routine
0110   00E4             ;Get address from input string
0111   00E4 13 E2 04    sm_dump          jmp   get_address
0112   00E7 11 0E 08    d_addr_back ldm   address
0113   00EA 12 22 08               stm   d_indexed_ldm+1
0114   00ED 11 0F 08               ldm   address+1
0115   00F0 12 23 08               stm   d_indexed_ldm+2
0116   00F3
0117   00F3             ;Dump 16 lines of 16 characters each
0118   00F3             ;Set up line counter
0119   00F3 10 10                  ldi   16
0120   00F5 12 14 08               stm   line_counter
0121   00F8             ;Loop for putting a memory dump line in the buffer
0122   00F8             ;Start with 4 characters of the starting address of the line, followed by space
0123   00F8 11 23 08    d_line_loop ldm   d_indexed_ldm+2   ;high byte of memory address
0124   00FB 12 12 08               stm   byte
0125   00FE                        call0(byte_to_hex_pair)
0125   00FE
0125   00FE 11 0D 01
0125   0101 12 40 08
0125   0104 11 0E 01
0125   0107 12 41 08
0125   010A 13 B4 06
0125   010D 0F 01
0126   010F 11 0A 08               ldm   char_pair
0127   0112 12 80 08               stm   buffer            ;start of line
0128   0115 11 0B 08               ldm   char_pair+1
0129   0118 12 81 08               stm   buffer+1
0130   011B 11 22 08               ldm   d_indexed_ldm+1   ;low byte of memory address
0131   011E 12 12 08               stm   byte
0132   0121                        call0(byte_to_hex_pair)
0132   0121
0132   0121 11 30 01
```

```
0132   0124 12 40 08
0132   0127 11 31 01
0132   012A 12 41 08
0132   012D 13 B4 06
0132   0130 32 01
0133   0132 11 0A 08                ldm   char_pair
0134   0135 12 82 08                stm   buffer+2
0135   0138 11 0B 08                ldm   char_pair+1
0136   013B 12 83 08                stm   buffer+3
0137   013E 10 20                   ldi   20h         ;space character
0138   0140 12 84 08                stm   buffer+4
0139   0143             ;Set up for getting 16 memory bytes, converting to characters, and putting in string
buffer
0140   0143 10 80                   ldi   buff_low
0141   0145 08 05                   addim 5
0142   0147 12 28 08                stm   d_indexed_stm+1   ;low byte of location of first character in output
string
0143   014A 10 08                   ldi   buff_high
0144   014C 09 00                   adcim 0           ;16-bit addition
0145   014E 12 29 08                stm   d_indexed_stm+2   ;high byte of location of first character in output
string
0146   0151 10 10                   ldi   16
0147   0153 12 16 08                stm   byte_counter      ;number of bytes to get, convert, and display in one
line
0148   0156 13 21 08    d_byte_loop:      jmp   d_indexed_ldm     ;get byte from memory
0149   0159 12 12 08    d_ldm_back: stm   byte
0150   015C                         call0(byte_to_hex_pair) ;convert to hex pair
0150   015C
0150   015C 11 6B 01
0150   015F 12 40 08
0150   0162 11 6C 01
0150   0165 12 41 08
0150   0168 13 B4 06
0150   016B 6D 01
0151   016D 10 03                   ldi   3
0152   016F 12 17 08                stm   nybble_counter
0153   0172 11 0A 08                ldm   char_pair
0154   0175 13 27 08    d_nybble_loop:    jmp   d_indexed_stm     ;store char of byte in string buffer
0155   0178 11 28 08    d_stm_back: ldm   d_indexed_stm+1   ;increment pointer by 16-bit incrementation
0156   017B 19                      inc
```

```
0157   017C 12 28 08                stm   d_indexed_stm+1
0158   017F 11 29 08                ldm   d_indexed_stm+2
0159   0182 09 00                   adcim 0
0160   0184 12 29 08                stm   d_indexed_stm+2   ;pointing to next spot in buffer
0161   0187 11 17 08                ldm   nybble_counter
0162   018A 1A                      dec               ;all three characters stored (hex chars plus space)?
0163   018B 14 A1 01                jpz   d_nybble_done     ;yes, next byte
0164   018E 12 17 08                stm   nybble_counter    ;no, place next character or space
0165   0191 0A 01                   subim 1           ;if nybble count = 1, put a space next
0166   0193 14 9C 01                jpz   d_put_space
0167   0196 11 0B 08                ldm   char_pair+1 ;otherwise, get next char and store
0168   0199 13 75 01                jmp   d_nybble_loop
0169   019C 10 20       d_put_space:      ldi   20h         ;space character
0170   019E 13 75 01                jmp   d_nybble_loop
0171   01A1 11 22 08    d_nybble_done:    ldm   d_indexed_ldm+1   ;increment memory pointer
0172   01A4 19                      inc
0173   01A5 12 22 08                stm   d_indexed_ldm+1
0174   01A8 11 23 08                ldm   d_indexed_ldm+2
0175   01AB 09 00                   adcim 0
0176   01AD 12 23 08                stm   d_indexed_ldm+2
0177   01B0 11 16 08                ldm   byte_counter
0178   01B3 1A                      dec
0179   01B4 14 BD 01                jpz   d_line_done
0180   01B7 12 16 08                stm   byte_counter
0181   01BA 13 56 01                jmp   d_byte_loop
0182   01BD 10 0D       d_line_done:      ldi   0dh         ;newline characters
0183   01BF 12 B4 08                stm   buffer+52
0184   01C2 10 0A                   ldi   0ah
0185   01C4 12 B5 08                stm   buffer+53
0186   01C7 10 00                   ldi   0
0187   01C9 12 B6 08                stm   buffer+54   ;where the end of the line will be
0188   01CC             ;Write string to screen
0189   01CC 10 80                   ldi   buff_low
0190   01CE 12 34 08                stm   ws_inst+1
0191   01D1 10 08                   ldi   buff_high
0192   01D3 12 35 08                stm   ws_inst+2
0193   01D6                         call0(write_string)
0193   01D6
0193   01D6 11 E5 01
0193   01D9 12 40 08
```

```
01DC 11 E6 01
01DF 12 41 08
01E2 13 E7 05
01E5 E7 01
01E7
01E7             ;Check if 16 lines done
01E7 11 14 08                ldm   line_counter
01EA 1A                      dec
01EB 14 F4 01                jpz   d_done
01EE 12 14 08                stm   line_counter
01F1 13 F8 00                jmp   d_line_loop
01F4             d_done:
01F4 13 98 00    error:      jmp   sm_warm
01F7
01F7             ;Monitor routine to jump and execute code
01F7             ;Gets target address from terminal
01F7
01F7 13 E2 04    sm_run      jmp   get_address
01FA 11 0E 08    run_addr_back     ldm   address
01FD 12 19 08                stm   run_jump+1
0200 11 0F 08                ldm   address+1
0203 12 1A 08                stm   run_jump+2
0206 13 18 08                jmp   run_jump
0209
0209             ;Routine to get hex char pairs from input and load bytes in RAM
0209             ;Get address first
0209 13 E2 04    sm_load     jmp   get_address
020C 11 0E 08    ld_addr_back      ldm   address
020F 12 1C 08                stm   ld_indexed_stm+1
0212 11 0F 08                ldm   address+1
0215 12 1D 08                stm   ld_indexed_stm+2
0218             ;Initialize display bytes counter
0218 10 10                   ldi   10h         ;16 bytes per line
021A 12 16 08                stm   byte_counter
021D             ;Get characters
021D             ;First character of pair
021D 17 03       ld_get_hi:  IN    03H         ;Get hi-order nybble of pair
021F 0C 02                   ANDIM 02H         ;check RxRDY bit
0221 14 1D 02                JPZ   ld_get_hi         ;not ready, loop
0224 17 02                   IN    02H         ;get char from data port
```

```
0226 12 10 08               STM   temp        ;Store character
0229 0A 0D                  SUBIM 0DH         ;Carriage return?
022B 14 12 03               JPZ   ld_done           ;Yes, return to monitor
022E 17 03       ld_loop_1: IN    03H         ;No, output character and validate
0230 0C 01                  ANDIM 01H         ;check TxRDY bit
0232 14 2E 02               JPZ   ld_loop_1   ;loop if not ready
0235 11 10 08               LDM   temp        ;get char back
0238 18 02                  OUT   02H         ;send to UART for output
023A             ;Code to validate hex character
023A 0F 30                  CMP   30H         ;Lower limit of hex characters
023C 16 42 02               JPC   ld_next_1   ;Char >= 30H, possibly valid
023F 13 60 02               JMP   ld_invalid  ;Char < 30H, invalid hex char
0242 0F 47       ld_next_1: CMP   47H         ;ASCII for "G"
0244 16 60 02               JPC   ld_invalid  ;Char is G or greater, invalid
0247 0F 41                  CMP   41H         ;ASCII for "A"
0249 16 54 02               JPC   ld_validAF_hi     ;Char is valid A-F
024C 0F 3A                  CMP   3AH         ;ASCII for ":"
024E 16 60 02               JPC   ld_invalid  ;Char is ":" or greater, but < "A", invalid
0251 13 5B 02               JMP   ld_valid09_hi     ;Char is valid 0-9
0254 0C 0F       ld_validAF_hi:    ANDIM 0FH         ;Mask off high bits
0256 08 09                  ADDIM 9           ;Adjust ASCII to binary value
0258 13 63 02               JMP   ld_shift_hi
025B 0C 0F       ld_valid09_hi:    ANDIM 0FH         ;Mask off high bits
025D 13 63 02               JMP   ld_shift_hi
0260 13 12 03    ld_invalid:JMP   ld_error    ;Invalid hex char, quit
0263 12 12 08    ld_shift_hi:      STM   byte        ;Will eventually contain the byte to load
0266 12 10 08               STM   temp        ;Value to add
0269 10 10                  LDI   10H         ;Multiply x 16 to shift into high-order nybble
026B 12 13 08               STM   counter
026E 11 13 08    ld_multloop:      LDM   counter
0271 1A                     DEC
0272 14 84 02               JPZ   ld_get_lo   ;Have added 16 times, done
0275 12 13 08               STM   counter
0278 11 10 08               LDM   temp        ;Original nybble
027B 00 12 08               ADD   byte        ;Add to BYTE and store
027E 12 12 08               STM   byte
0281 13 6E 02               JMP   ld_multloop ;Keep adding
0284 17 03       ld_get_lo: IN    03H         ;Get lo-order nybble of pair
0286 0C 02                  ANDIM 02H         ;check RxRDY bit
0288 14 84 02               JPZ   ld_get_lo   ;not ready, loop
```

```
028B 17 02                  IN    02H         ;get char from data port
028D 12 10 08               STM   temp        ;Store character
0290 17 03       ld_loop2:  IN    03H         ;Output character
0292 0C 01                  ANDIM 01H         ;check TxRDY bit
0294 14 90 02               JPZ   ld_loop2
0297 11 10 08               LDM   temp        ;When ready, retrieve character and output
029A 18 02                  OUT   02H
029C 17 03       ld_loop3:  IN    03H
029E 0C 01                  ANDIM 01H
02A0 14 9C 02               JPZ   ld_loop3
02A3 10 20                  LDI   20H         ;Space character
02A5 18 02                  OUT   02H         ;send to UART for output
02A7 11 16 08               ldm   byte_counter      ;Check if 16 bytes have been displayed
02AA 1A                     dec
02AB 12 16 08               stm   byte_counter
02AE 14 B4 02               jpz   ld_next4    ;Yes, reset counter and write newline
02B1 13 CF 02               jmp   ld_next5    ;No, keep going
02B4 10 10       ld_next4:  ldi   10h
02B6 12 16 08               stm   byte_counter
02B9             ;Write newline
02B9 17 03       ld_loop4:  IN    03H
02BB 0C 01                  ANDIM 01H
02BD 14 B9 02               JPZ   ld_loop4
02C0 10 0D                  LDI   0DH         ;Return character
02C2 18 02                  OUT   02H         ;send to UART for output
02C4 17 03       ld_loop5:  IN    03H
02C6 0C 01                  ANDIM 01H
02C8 14 C4 02               JPZ   ld_loop5
02CB 10 0A                  LDI   0AH         ;Linefeed character
02CD 18 02                  OUT   02H         ;send to UART for output
02CF             ;Code to validate hex character
02CF 11 10 08    ld_next5:  LDM   temp        ;Retrieve character and validate
02D2 0F 30                  CMP   30H         ;Lower limit of hex characters
02D4 16 DA 02               JPC   ld_next2    ;Char >= 30H, possibly valid
02D7 13 60 02               JMP   ld_invalid  ;Char < 30H, invalid hex char
02DA 0F 47       ld_next2:  CMP   47H         ;ASCII for "G"
02DC 16 60 02               JPC   ld_invalid  ;Char is G or greater, invalid
02DF 0F 41                  CMP   41H         ;ASCII for "A"
02E1 16 EC 02               JPC   ld_validAF_lo     ;Char is valid A-F
02E4 0F 3A                  CMP   3AH         ;ASCII for ":"
```

```
0310   02E6 16 60 02            JPC   ld_invalid  ;Char is ":" or greater, but < "A", invalid
0311   02E9 13 F6 02            JMP   ld_valid09_lo     ;Char is valid 0-9
0312   02EC 0C 0F      ld_validAF_lo:   ANDIM 0FH         ;Mask off high bits
0313   02EE 08 09               ADDIM 9           ;Now lo nybble correct
0314   02F0 00 12 08            ADD   byte        ;Combine with hi nybble stored in BYTE
0315   02F3 13 1B 08            JMP   ld_indexed_stm    ;Store the byte in RAM
0316   02F6 0C 0F      ld_valid09_lo:   ANDIM 0FH         ;Mask off high bits
0317   02F8 00 12 08            ADD   byte        ;Now full byte assembled
0318   02FB 18 00               OUT   00H         ;Display on LEDs
0319   02FD 13 1B 08            JMP   ld_indexed_stm    ;Store the byte in RAM
0320   0300 11 1C 08   ld_stm_back:     LDM   ld_indexed_stm+1  ;Increment byte pointer, lo byte first
0321   0303 19                  INC
0322   0304 12 1C 08            STM   ld_indexed_stm+1
0323   0307 11 1D 08            LDM   ld_indexed_stm+2  ;Increment hi byte if a carry occurred when lo byte
incremented
0324   030A 09 00               ADCIM 00H
0325   030C 12 1D 08            STM   ld_indexed_stm+2
0326   030F 13 1D 02            JMP   ld_get_hi
0327   0312            ld_done:
0328   0312 13 98 00   ld_error:   JMP   sm_warm           ;Return to monitor
0329   0315
0330   0315            ;Monitor routine for binary load
0331   0315            ;Gets load target address and number of bytes from terminal
0332   0315            ;Loads bytes in RAM and returns to monitor
0333   0315 13 E2 04   sm_bload    jmp   get_address
0334   0318 11 0E 08   bl_addr_back     ldm   address
0335   031B 12 2E 08            stm   bl_indexed_stm+1
0336   031E 11 0F 08            ldm   address+1
0337   0321 12 2F 08            stm   bl_indexed_stm+2
0338   0324
0339   0324            ;Gets number of bytes in decimal from input using get_line, called as level 0 subroutine
0340   0324            ;Write newline
0341   0324 11 B5 07   bl_get_bytes     ldm   bytes_str
0342   0327 12 34 08            stm   ws_inst+1
0343   032A 11 B6 07            ldm   bytes_str+1
0344   032D 12 35 08            stm   ws_inst+2
0345   0330                     call0(write_string)
0345   0330
0345   0330 11 3F 03
0345   0333 12 40 08
```

```
0345   0336 11 40 03
0345   0339 12 41 08
0345   033C 13 E7 05
0345   033F 41 03
0346   0341
0347   0341             ;Get input decimal number string
0348   0341                         call0(get_line)
0348   0341
0348   0341 11 50 03
0348   0344 12 40 08
0348   0347 11 51 03
0348   034A 12 41 08
0348   034D 13 8D 05
0348   0350 52 03
0349   0352
0350   0352             ;Get word value from input string
0351   0352             ;No error checking for final value -- must be between 0 and 65535 (0000 and FFFF hex)
0352   0352             ;No error checking for numerals -- must be 0 to 9
0353   0352
0354   0352 10 00                   ldi   0           ;starting value
0355   0354 12 00 08                stm   dp_value    ;zero final value variable
0356   0357 12 01 08                stm   dp_value+1
0357   035A 11 07 08    dp_input    ldm   gl_str_len  ;input string length from get_line
0358   035D 0F 05                   cmp   5
0359   035F 16 79 03                jpc   dp_setup_5
0360   0362 0F 04                   cmp   4
0361   0364 16 9A 03                jpc   dp_setup_4
0362   0367 0F 03                   cmp   3
0363   0369 16 B5 03                jpc   dp_setup_3
0364   036C 0F 02                   cmp   2
0365   036E 16 CA 03                jpc   dp_setup_2
0366   0371 0F 01                   cmp   1
0367   0373 16 D9 03                jpc   dp_setup_1
0368   0376 13 98 00                jmp   sm_warm
0369   0379
0370   0379 11 84 08    dp_setup_5  ldm   buffer+4
0371   037C 12 06 08                stm   dp_1s
0372   037F 11 83 08                ldm   buffer+3
0373   0382 12 05 08                stm   dp_10s
0374   0385 11 82 08                ldm   buffer+2
```

```
0388 12 04 08               stm   dp_100s
038B 11 81 08               ldm   buffer+1
038E 12 03 08               stm   dp_1000s
0391 11 80 08               ldm   buffer
0394 12 02 08               stm   dp_10000s
0397 13 E2 03               jmp   dp_10000_mult
039A 11 83 08    dp_setup_4 ldm   buffer+3
039D 12 06 08               stm   dp_1s
03A0 11 82 08               ldm   buffer+2
03A3 12 05 08               stm   dp_10s
03A6 11 81 08               ldm   buffer+1
03A9 12 04 08               stm   dp_100s
03AC 11 80 08               ldm   buffer
03AF 12 03 08               stm   dp_1000s
03B2 13 04 04               jmp   dp_1000_mult
03B5 11 82 08    dp_setup_3 ldm   buffer+2
03B8 12 06 08               stm   dp_1s
03BB 11 81 08               ldm   buffer+1
03BE 12 05 08               stm   dp_10s
03C1 11 80 08               ldm   buffer
03C4 12 04 08               stm   dp_100s
03C7 13 26 04               jmp   dp_100_mult
03CA 11 81 08    dp_setup_2 ldm   buffer+1
03CD 12 06 08               stm   dp_1s
03D0 11 80 08               ldm   buffer
03D3 12 05 08               stm   dp_10s
03D6 13 48 04               jmp   dp_10_mult
03D9 11 80 08    dp_setup_1 ldm   buffer
03DC 12 06 08               stm   dp_1s
03DF 13 6A 04               jmp   dp_1_mult
03E2
03E2
03E2
03E2             ;decimal parser multiplication
03E2 11 02 08    dp_10000_mult    ldm   dp_10000s
03E5 0A 30                  subim 30h         ;ASCII for '0'
03E7 14 04 04               jpz   dp_10000_done
03EA 10 10                  ldi   10h         ;hex low byte of 10,000 decimal
03EC 00 00 08               addm  dp_value
03EF 12 00 08               stm   dp_value
```

```
03F2 10 27                   ldi   27h         ;hex high byte of 10,000 decimal
03F4 01 01 08                adcm  dp_value+1
03F7 12 01 08                stm   dp_value+1
03FA 11 02 08                ldm   dp_10000s
03FD 1A                      dec
03FE 12 02 08                stm   dp_10000s
0401 13 E2 03                jmp   dp_10000_mult
0404             dp_10000_done
0404 11 03 08    dp_1000_mult      ldm   dp_1000s
0407 0A 30                   subim 30h         ;ASCII for '0'
0409 14 26 04                jpz   dp_1000_done
040C 10 E8                   ldi   0e8h        ;hex low byte of 1000 decimal
040E 00 00 08                addm  dp_value
0411 12 00 08                stm   dp_value
0414 10 03                   ldi   03h         ;hex high byte of 1000 decimal
0416 01 01 08                adcm  dp_value+1
0419 12 01 08                stm   dp_value+1
041C 11 03 08                ldm   dp_1000s
041F 1A                      dec
0420 12 03 08                stm   dp_1000s
0423 13 04 04                jmp   dp_1000_mult
0426             dp_1000_done
0426 11 04 08    dp_100_mult ldm   dp_100s
0429 0A 30                   subim 30h         ;ASCII for '0'
042B 14 48 04                jpz   dp_100_done
042E 10 64                   ldi   64h         ;hex low byte of 100 decimal
0430 00 00 08                addm  dp_value
0433 12 00 08                stm   dp_value
0436 10 00                   ldi   00h         ;hex high byte of 100 decimal
0438 01 01 08                adcm  dp_value+1
043B 12 01 08                stm   dp_value+1
043E 11 04 08                ldm   dp_100s
0441 1A                      dec
0442 12 04 08                stm   dp_100s
0445 13 26 04                jmp   dp_100_mult
0448             dp_100_done
0448 11 05 08    dp_10_mult  ldm   dp_10s
044B 0A 30                   subim 30h         ;ASCII for '0'
044D 14 6A 04                jpz   dp_10_done
0450 10 0A                   ldi   0ah         ;hex low byte of 10 decimal
```

```
0455   0452 00 00 08               addm  dp_value
0456   0455 12 00 08               stm   dp_value
0457   0458 10 00                  ldi   00h         ;hex high byte of 10 decimal
0458   045A 01 01 08               adcm  dp_value+1
0459   045D 12 01 08               stm   dp_value+1
0460   0460 11 05 08               ldm   dp_10s
0461   0463 1A                     dec
0462   0464 12 05 08               stm   dp_10s
0463   0467 13 48 04               jmp   dp_10_mult
0464   046A             dp_10_done
0465   046A 11 06 08    dp_1_mult  ldm   dp_1s
0466   046D 0A 30                  subim 30h         ;ASCII for '0'
0467   046F 00 00 08               addm  dp_value
0468   0472 12 00 08               stm   dp_value
0469   0475 10 00                  ldi   00h         ;hex high byte of 100 decimal
0470   0477 01 01 08               adcm  dp_value+1
0471   047A 12 01 08               stm   dp_value+1
0472   047D
0473   047D             ;Set up byte counter and write ready string
0474   047D 11 00 08               ldm   dp_value
0475   0480 12 08 08               stm   bl_byte_counter
0476   0483 11 01 08               ldm   dp_value+1
0477   0486 12 09 08               stm   bl_byte_counter+1
0478   0489 11 CF 07               ldm   bl_ready_str
0479   048C 12 34 08               stm   ws_inst+1
0480   048F 11 D0 07               ldm   bl_ready_str+1
0481   0492 12 35 08               stm   ws_inst+2
0482   0495                        call0(write_string)
0482   0495
0482   0495 11 A4 04
0482   0498 12 40 08
0482   049B 11 A5 04
0482   049E 12 41 08
0482   04A1 13 E7 05
0482   04A4 A6 04
0483   04A6
0484   04A6             ;Loop to get binary data and store
0485   04A6 17 03       bl_chk_loop:     in    03h         ;get status
0486   04A8 0C 02                  andim 02h         ;check RxRDY
0487   04AA 14 A6 04               jpz   bl_chk_loop
```

```
04AD 17 02                  in    02h         ;get binary from port
04AF 13 2D 08               jmp   bl_indexed_stm    ;store in RAM
04B2 11 2E 08    bl_back    ldm   bl_indexed_stm+1;increment pointer
04B5 19                     inc
04B6 12 2E 08               stm   bl_indexed_stm+1
04B9 11 2F 08               ldm   bl_indexed_stm+2
04BC 09 00                  adcim 0
04BE 12 2F 08               stm   bl_indexed_stm+2
04C1 11 08 08               ldm   bl_byte_counter   ;decrement byte counter
04C4 1A                     dec
04C5 12 08 08               stm   bl_byte_counter
04C8 11 09 08               ldm   bl_byte_counter+1
04CB 0B 00                  sbbim 0
04CD 12 09 08               stm   bl_byte_counter+1
04D0 14 D6 04               jpz   bl_low_zero ;check if byte counter = zero
04D3 13 A6 04               jmp   bl_chk_loop
04D6 11 08 08    bl_low_zero ldm  bl_byte_counter
04D9 14 DF 04               jpz   bl_done           ;yes, done -- return to monitor
04DC 13 A6 04               jmp   bl_chk_loop ;no, get next byte
04DF 13 98 00    bl_done    jmp   sm_warm
04E2
04E2
04E2
04E2             ;Routine to get address
04E2             ;Not called as a subroutine, return jump by switch structure
04E2 11 A1 07    get_address ldm  addr_str
04E5 12 34 08               stm   ws_inst+1
04E8 11 A2 07               ldm   addr_str+1
04EB 12 35 08               stm   ws_inst+2
04EE                        call0(write_string)
04EE
04EE 11 FD 04
04F1 12 40 08
04F4 11 FE 04
04F7 12 41 08
04FA 13 E7 05
04FD FF 04
04FF
04FF             ;Get hex input string for address
04FF                        call0(get_line)
```

```
0520   04FF
0520   04FF 11 0E 05
0520   0502 12 40 08
0520   0505 11 0F 05
0520   0508 12 41 08
0520   050B 13 8D 05
0520   050E 10 05
0521   0510
0522   0510             ;Write newline
0523   0510 11 31 07               ldm   new_line
0524   0513 12 34 08               stm   ws_inst+1
0525   0516 11 32 07               ldm   new_line+1
0526   0519 12 35 08               stm   ws_inst+2
0527   051C                        call0(write_string)
0527   051C
0527   051C 11 2B 05
0527   051F 12 40 08
0527   0522 11 2C 05
0527   0525 12 41 08
0527   0528 13 E7 05
0527   052B 2D 05
0528   052D
0529   052D             ;No error checking for length of string -- must be exactly 4 hex characters
0530   052D             ;Memory address stored in address variable
0531   052D 11 80 08               ldm   buffer           ;first character
0532   0530 12 0A 08               stm   char_pair
0533   0533 11 81 08               ldm   buffer+1    ;second character
0534   0536 12 0B 08               stm   char_pair+1
0535   0539                        call1(hex_pair_to_byte)
0535   0539
0535   0539 11 48 05
0535   053C 12 43 08
0535   053F 11 49 05
0535   0542 12 44 08
0535   0545 13 54 06
0535   0548 4A 05
0536   054A 16 98 00               jpc   sm_warm          ;non-hex character detected, quit
0537   054D 12 0F 08               stm   address+1   ;high byte of address
0538   0550 11 82 08               ldm   buffer+2    ;third character
0539   0553 12 0A 08               stm   char_pair
```

```
0556 11 83 08               ldm   buffer+3    ;fourth character
0559 12 0B 08               stm   char_pair+1
055C                        call1(hex_pair_to_byte)
055C
055C 11 6B 05
055F 12 43 08
0562 11 6C 05
0565 12 44 08
0568 13 54 06
056B 6D 05
056D 16 98 00               jpc   sm_warm           ;non-hex character detected, quit
0570 12 0E 08               stm   address           ;low byte of address
0573             ;Switch for return jumps
0573 11 11 08               ldm   choice
0576 0F 35                  cmp   '5'
0578 16 18 03               jpc   bl_addr_back
057B 0F 34                  cmp   '4'
057D 16 0C 02               jpc   ld_addr_back
0580 0F 33                  cmp   '3'
0582 16 FA 01               jpc   run_addr_back
0585 0F 32                  cmp   '2'
0587 16 E7 00               jpc   d_addr_back
058A 13 98 00               jmp   sm_warm
058D
058D             ;Get_line subroutine, call as level 0
058D             ;Gets a line from input, puts zero-terminated string in buffer
058D             ;Echos characters to screen, except terminating carriage return
058D             ;Address of buffer in buff_low and buff_high constants
058D             ;Uses RAM variable address instruction gl_indexed_stm
058D             ;length of input string returned in gl_str_len
058D             ;Returns when carriage return entered
058D
058D 10 00       get_line:  ldi   0
058F 12 07 08               stm   gl_str_len  ;string length
0592 10 80                  ldi   buff_low    ;low byte of buffer address
0594 12 3A 08               stm   gl_indexed_stm+1
0597 10 08                  ldi   buff_high   ;high byte of buffer address
0599 12 3B 08               stm   gl_indexed_stm+2
059C 17 03       gl_chk_loop:     in    03h         ;get status
059E 0C 02                  andim 02h         ;check RxRDY
```

```
05A0 14 9C 05               jpz   gl_chk_loop
05A3 17 02                  in    02h         ;get char from port
05A5 12 10 08               stm   temp        ;save character
05A8 0A 0D                  subim 0dh         ;is it a return character?
05AA 14 BA 05               jpz   gl_end_of_line    ;yes, replace with a zero
05AD 11 07 08               ldm   gl_str_len  ;no, increment string length
05B0 19                     inc
05B1 12 07 08               stm   gl_str_len
05B4 11 10 08               ldm   temp        ;get back char
05B7 13 BD 05               jmp   gl_store_it ;place character in buffer
05BA 12 10 08    gl_end_of_line:  stm   temp        ;place zero in temp
05BD 13 39 08    gl_store_it:     jmp   gl_indexed_stm    ;store character in buffer
05C0 11 10 08    gl_back:   ldm   temp        ;check if end-of-line (temp = 0)
05C3 14 E4 05               jpz   gl_done           ;yes, quit
05C6 17 03       gl_out_loop:     in    03h         ;no, send char to screen
05C8 0C 01                  andim 01h         ;check TxRDY
05CA 14 C6 05               jpz   gl_out_loop ;loop if not ready
05CD 11 10 08               ldm   temp
05D0 18 02                  out   02h         ;output character to port
05D2 11 3A 08               ldm   gl_indexed_stm+1  ;increment indexing pointer
05D5 19                     inc
05D6 12 3A 08               stm   gl_indexed_stm+1
05D9 11 3B 08               ldm   gl_indexed_stm+2
05DC 09 00                  adcim 00h         ;16-bit addition
05DE 12 3B 08               stm   gl_indexed_stm+2
05E1 13 9C 05               jmp   gl_chk_loop
05E4             gl_done:   ret0
05E4 13 3F 08
05E7
05E7             ;Write_string subroutine, call as level 0
05E7             ;Writes a zero-terminated string to screen at current cursor location
05E7             ;Must set up address of string to be written in ws_inst+1 and ws_inst+2
05E7             write_string:
05E7 17 03       ws_chk_loop:     in    03h   ;get status
05E9 0C 01                  andim 01h   ;check TxRDY
05EB 14 E7 05               jpz   ws_chk_loop
05EE 13 33 08               jmp   ws_inst     ;get a character when port ready
05F1 14 08 06    ws_back:   jpz   ws_done     ;quit if end-of-string
05F4 18 02                  out   02h   ;output character to port
05F6 11 34 08               ldm   ws_inst+1   ;indexing pointer
```

```
0612   05F9 19                        inc
0613   05FA 12 34 08                  stm   ws_inst+1
0614   05FD 11 35 08                  ldm   ws_inst+2
0615   0600 09 00                     adcim 00h   ;16-bit addition
0616   0602 12 35 08                  stm   ws_inst+2
0617   0605 13 E7 05                  jmp   ws_chk_loop
0618   0608             ws_done:   ret0
0618   0608 13 3F 08
0619   060B
0620   060B             ;Subroutine hex_to word -- call as level 0
0621   060B             ;Calls hex_pair_to_byte as level 1
0622   060B             ;Get 16-bit word value from input string in buffer
0623   060B             ;No error checking for length of string -- must be exactly 4 hex characters
0624   060B             ;16-bit value placed in h2w_value
0625   060B 11 80 08    hex_to_word:     ldm   buffer            ;first character
0626   060E 12 0A 08              stm   char_pair
0627   0611 11 81 08              ldm   buffer+1    ;second character
0628   0614 12 0B 08              stm   char_pair+1
0629   0617                       call1(hex_pair_to_byte) ;high-order byte
0629   0617
0629   0617 11 26 06
0629   061A 12 43 08
0629   061D 11 27 06
0629   0620 12 44 08
0629   0623 13 54 06
0629   0626 28 06
0630   0628 16 51 06              jpc   h2w_done    ;non-hex character detected, exit with carry set
0631   062B 12 0D 08              stm   h2w_value+1 ;high byte of address
0632   062E 11 82 08              ldm   buffer+2    ;third character
0633   0631 12 0A 08              stm   char_pair
0634   0634 11 83 08              ldm   buffer+3    ;fourth character
0635   0637 12 0B 08              stm   char_pair+1
0636   063A                       call1(hex_pair_to_byte) ;low-order byte
0636   063A
0636   063A 11 49 06
0636   063D 12 43 08
0636   0640 11 4A 06
0636   0643 12 44 08
0636   0646 13 54 06
0636   0649 4B 06
```

```
064B 16 51 06               jpc   h2w_done    ;exit with carry set if error
064E 12 0C 08               stm   h2w_value   ;low byte of address
0651            h2w_done    ret0
0651 13 3F 08
0654
0654            ;Subroutine to convert hex character pair to byte, call as level 1
0654            ;Character pair in memory location char_pair, stored hi-low
0654            ;Returns with byte in accumulator and carry flag clear if no error
0654            ;Returns with character in accumulator and carry flag set if error
0654            ;Calls char_to_nybble as level 2 subroutine
0654 11 0A 08    hex_pair_to_byte: ldm  char_pair   ;high order character of pair
0657 12 10 08               stm   temp        ;char_to_nybble needs char in TEMP
065A                        call2(char_to_nybble)
065A
065A 11 69 06
065D 12 46 08
0660 11 6A 06
0663 12 47 08
0666 13 FF 06
0669 6B 06
066B 16 B0 06               jpc   c2n_error
066E 12 12 08               STM   byte        ;Will eventually contain the byte to load
0671 12 10 08               STM   temp        ;Value to add
0674 10 10                  LDI   10H         ;Multiply x 16 to shift into high-order nybble
0676 12 13 08               STM   counter
0679 11 13 08    MULTLOOP:  LDM   counter
067C 1A                     DEC
067D 14 8F 06               JPZ   GET_LO            ;Have added 16 times, done
0680 12 13 08               STM   counter
0683 11 10 08               LDM   temp        ;Original nybble
0686 00 12 08               ADD   byte        ;Add to BYTE and store
0689 12 12 08               STM   byte
068C 13 79 06               JMP   MULTLOOP    ;Keep adding
068F 11 0B 08    GET_LO:    ldm   char_pair+1
0692 12 10 08               stm   temp
0695                        call2(char_to_nybble)
0695
0695 11 A4 06
0698 12 46 08
069B 11 A5 06
```

```
069E 12 47 08
06A1 13 FF 06
06A4 A6 06
06A6 16 B0 06                jpc   c2n_error
06A9 00 12 08                ADD   byte        ;Combine with hi nybble stored in BYTE
06AC 1C                      ccf               ;in case addition changed it
06AD 13 B1 06                JMP   c2b_done    ;Done, no error
06B0 1B          c2n_error:  scf
06B1             c2b_done:   ret1
06B1 13 42 08
06B4
06B4             ;Subroutine to convert byte to hex character pair, call as level 0
06B4             ;Gets byte from byte variable
06B4             ;Returns with character pair in memory location char_pair, stored hi-low
06B4             ;
06B4             byte_to_hex_pair:
06B4 11 12 08                ldm   byte
06B7 0C F0                   andim 0f0h        ;dealing with high nybble
06B9 12 10 08                stm   temp
06BC 10 00                   ldi   00h         ;prepare to shift down (divide by 16)
06BE 12 13 08                stm   counter
06C1 11 10 08    b2h_divide: ldm   temp
06C4 0A 10                   subim 16
06C6 16 D6 06                jpc   b2h_cont    ;continue if nybble >=0
06C9 11 13 08                ldm   counter           ;hi-nybble now in low position
06CC 0F 0A                   cmp   10          ;is value <10?
06CE 16 E3 06                jpc   b2h_hi_A2F  ;yes, jump and convert to char
06D1 08 30                   addim 30h         ;no, convert to char
06D3 13 E5 06                jmp   b2h_store_hi
06D6 12 10 08    b2h_cont:   stm   temp
06D9 11 13 08                ldm   counter
06DC 19                      inc
06DD 12 13 08                stm   counter
06E0 13 C1 06                jmp   b2h_divide
06E3 08 37       b2h_hi_A2F: addim 37h
06E5 12 0A 08    b2h_store_hi:     stm   char_pair   ;hi-order hex char
06E8 11 12 08                ldm   byte
06EB 0C 0F                   andim 0fh         ;dealing with low-order nybble
06ED 0F 0A                   cmp   10          ;is value <10?
06EF 16 F7 06                jpc   b2h_lo_A2F  ;yes, jump and convert to char
```

```
0701   06F2 08 30                  addim 30h         ;no, convert to char
0702   06F4 13 F9 06               jmp   b2h_store_lo
0703   06F7 08 37       b2h_lo_A2F:addim 37h
0704   06F9 12 0B 08    b2h_store_lo:    stm   char_pair+1 ;now char pair is in variable
0705   06FC                        ret0
0705   06FC 13 3F 08
0706   06FF
0707   06FF
0708   06FF             ;Subroutine to convert hex char to nybble, call as level 2
0709   06FF             ;Checks for validity of char, 0-9 and A-F (upper case only)
0710   06FF             ;Carry flag set on exit if error
0711   06FF             ;Carry flag clear if character valid
0712   06FF             ;Call with char in temp
0713   06FF             ;Exits with nybble in lower half of accumulator if no error
0714   06FF             ;Original character in accumulator if error
0715   06FF
0716   06FF 11 10 08    char_to_nybble:  ldm   temp        ;Get character
0717   0702 0F 30                  cmp   30H         ;Lower limit of hex characters
0718   0704 16 0A 07               jpc   c2n_next    ;Char >= 30H, possibly valid
0719   0707 13 2A 07               jmp   invalid           ;Char < 30H, invalid hex char
0720   070A 0F 47       c2n_next:  cmp   47h         ;ASCII for "G"
0721   070C 16 2A 07               jpc   invalid           ;Char is G or greater, invalid
0722   070F 0F 41                  cmp   41h         ;ASCII for "A"
0723   0711 16 1C 07               jpc   validAF           ;Char is valid A-F
0724   0714 0F 3A                  cmp   3Ah         ;ASCII for ":"
0725   0716 16 2A 07               jpc   invalid           ;Char is ":" or greater, but < "A", invalid
0726   0719 13 24 07               jmp   valid09           ;Char is valid 0-9
0727   071C 0C 0F       validAF:   andim 0fh         ;Mask off high bits
0728   071E 08 09                  addim 9           ;Adjust ASCII to binary value
0729   0720 1C                     ccf               ;exit no error
0730   0721                        ret2
0730   0721 13 45 08
0731   0724 0C 0F       valid09:   andim 0fh         ;Mask off high bits
0732   0726 1C                     ccf               ;exit no error
0733   0727                        ret2
0733   0727 13 45 08
0734   072A 11 10 08    invalid:   ldm   temp        ;put char in accumulator
0735   072D 1B                     scf               ;Set carry flag
0736   072E                        ret2
0736   072E 13 45 08
```

```
0737   0731             ;String constants
0738   0731 33 07       new_line    .dw   $+2
0739   0733 0D 0A 00                .db   0dh,0ah,0
0740   0736 38 07       sm_greeting:      .dw   $+2
0741   0738 0D 0A                   .db   0dh,0ah
0742   073A 43505576696C            .text "CPUville 8-bit processor system monitor v.1"
0742   0740 6C6520382D6269742070726F636573736F722073797374656D206D6F6E69746F7220762E31
0743   0765 0D 0A 00                .db   0dh,0ah,0
0744   0768 6A 07       sm_prompt   .dw   $+2
0745   076A 0D 0A                   .db   0dh,0ah
0746   076C 456E74657220            .text "Enter number: 1=restart 2=dump 3=run 4=load 5=bload "
0746   0772 6E756D6265723A20313D7265737461727420323D64756D7020333D72756E20343D6C6F616420353D626C6F616420
0747   07A0 00                      .db   0
0748   07A1 A3 07       addr_str    .dw   $+2
0749   07A3 0D 0A                   .db   0dh,0ah
0750   07A5 416464726573            .text Address (hex):
0750   07AB 732028686578293A20
0751   07B4 00                      .db   0
0752   07B5 B7 07       bytes_str   .dw   $+2
0753   07B7 0D 0A                   .db   0dh,0ah
0754   07B9 427974657320            .text Bytes to load (dec):
0754   07BF 746F206C6F6164202864656563293A20
0755   07CE 00                      .db   0
0756   07CF D1 07       bl_ready_str      .dw   $+2
0757   07D1 0D 0A                   .db   0dh,0ah
0758   07D3 52656164792C            .text Ready, start transfer
0758   07D9 2073746172742074726616E73666572
0759   07E8 0D 0A 00                .db   0dh,0ah,0
0760   07EB
0761   07EB             ;The following section contains labels for RAM variables and other structures
0762   0800                         .org  0800h      ;Start of RAM
0763   0800             ;RAM Variables
0764   0800 00 00       dp_value    .dw   0000h
0765   0802 00          dp_10000s   .db   00h
0766   0803 00          dp_1000s    .db   00h
0767   0804 00          dp_100s           .db   00h
0768   0805 00          dp_10s            .db   00h
0769   0806 00          dp_1s       .db   00h
0770   0807 00          gl_str_len  .db   00h
0771   0808 00 00       bl_byte_counter   .dw   0000h
```

```
080A 00 00       char_pair  .dw   0000h
080C 00 00       h2w_value  .dw   0000h
080E 00 00       address    .dw   0000h
0810 00          temp       .db   00h
0811 00          choice     .db   00h
0812 00          byte       .db   00h
0813 00          counter    .db   00h
0814 00          line_counter     .db   00h
0815 00          char_count .db   00h
0816 00          byte_counter     .db   00h
0817 00          nybble_counter   .db   00h
0818
0818             ;RAM instructions with variable address (must initialize opcode when monitor is in ROM)
0818             ;Jump instruction for run routine, must be in RAM
0818 13 00 00    run_jump   jmp   0000h
081B             ;Indexed load for load routine, must be in RAM
081B 12 00 00    ld_indexed_stm   stm   0000h
081E 13 00 03               jmp   ld_stm_back
0821             ;Indexed load and store instructions for dump, must be in RAM
0821 11 00 00    d_indexed_ldm    ldm   0000h
0824 13 59 01               jmp   d_ldm_back
0827 12 00 00    d_indexed_stm    stm   0000h
082A 13 78 01               jmp   d_stm_back
082D             ;Indexed store instruction for binary loader, must be in RAM
082D 12 00 00    bl_indexed_stm:  stm   0000h
0830 13 B2 04               jmp   bl_back
0833             ;Indexed load instruction for write_string, must be in RAM
0833 11 00 00    ws_inst:   ldm   0000h
0836 13 F1 05               jmp   ws_back
0839             ;Indexed store instruction for get_line, must be in RAM
0839 12 00 00    gl_indexed_stm:  stm   0000h
083C 13 C0 05               jmp   gl_back
083F             ;Return instruction for level 0 call macros, must be in RAM
083F 13 00 00    return_jump0:    jmp   0000h
0842             ;Return instruction for level 1 call macros, must be in RAM
0842 13 00 00    return_jump1:    jmp   0000h
0845             ;Return instruction for level 2 call macros, must be in RAM
0845 13 00 00    return_jump2:    jmp   0000h
0848
0848             return:          .set  0000h ;assembler needs variable label set back to original value
```

```
0812   0848                        .end
0813   0848
0814   0848
0815   0848
0816   0848
0817   0848
0818   0848
0819   0848
tasm: Number of errors = 0
```