Table of Contents

Introduction .......................................................................................................................... 5
Building Tips .......................................................................................................................... 6
Building the ALU ................................................................................................................... 10
Building the Main Board ...................................................................................................... 12
Building the Control Board ................................................................................................. 15
Assembling the 8-bit Processor .............................................................................................. 18
Control Board Switches ........................................................................................................ 21
Using the 8-bit Processor with a 4K System ......................................................................... 21
Using the 8-bit Processor with a 64K System ....................................................................... 23

Monitor commands .............................................................................................................. 26
1 = restart ............................................................................................................................ 26
2 = dump ............................................................................................................................. 26
3 = run ................................................................................................................................ 27
4 = load ................................................................................................................................ 27
5 = bload ............................................................................................................................. 29

Connecting a disk drive ........................................................................................................ 33
Introduction to Programming for the 8-bit Processor .............................................................. 33
The Instruction Set ................................................................................................................ 34
ADC – add with carry ............................................................................................................ 35
ADClM – add with carry, immediate ..................................................................................... 37
ADD – add memory data to the accumulator ....................................................................... 38
ADDIM – add immediate data to the accumulator ............................................................... 39
AND – bitwise logical AND of memory data with the accumulator .................................... 40
ANDIM – bitwise logical AND of immediate data with the accumulator ......................... 41
CCF – clear carry flag ........................................................................................................ 42
CMP – compare .................................................................................................................. 43
DEC – decrement accumulator ............................................................................................ 44
IN – load accumulator with data from an input port .......................................................... 45
INC – increment accumulator ............................................................................................ 46
JMP – jump unconditional .................................................................................................. 47
JPC – jump if carry .............................................................................................................. 48
JPM – jump if minus ............................................................................................................ 49
JPZ – jump if zero ................................................................................................................ 50
LDI – load accumulator immediate ..................................................................................... 51
LDM – load accumulator from memory ............................................................................... 52
NOP – no operation ............................................................................................................. 53
NOT – bitwise invert (ones-complement) accumulator ....................................................... 54
OR – bitwise logical OR of memory data with accumulator ............................................... 55
ORIM – bitwise logical OR of immediate data with accumulator ...................................... 56
OUT – load output port with accumulator .......................................................................... 57
SBB – subtract memory and borrow from accumulator ....................................................... 58
SBBIM – subtract immediate data and borrow from accumulator ...................................... 60
SCF – set carry flag ........................................................................................................... 62
STM – store accumulator to memory .................................................................................. 63
SUB – subtract memory from accumulator ................................................................. 64
SUBIM – subtract immediate data from accumulator .................................................. 66
XOR – bitwise exclusive OR of memory with accumulator ........................................... 68
XORIM – bitwise exclusive-OR of immediate data with accumulator ......................... 69
Using TASM .................................................................................................................. 70
Schematics and Explanations ....................................................................................... 73
Overall design of the processor data path ................................................................... 74
Main board (data path) schematics ............................................................................ 75
  Display connector ........................................................................................................ 76
  Program counter ........................................................................................................... 77
  Zero flag logic ............................................................................................................. 78
  Address source multiplexer ......................................................................................... 79
  Instruction register ....................................................................................................... 80
  Data-out buffer ............................................................................................................ 81
  Control connector ......................................................................................................... 81
  ALU op source multiplexer, carry flip-flop, and ALU connector ............................... 82
  Accumulator source multiplexer and accumulator ..................................................... 83
  ALU B source multiplexer ......................................................................................... 84
  System connector ........................................................................................................ 85
ALU schematics ............................................................................................................ 85
  ALU connector ............................................................................................................ 87
  Carry-out logic ............................................................................................................. 88
  ALU logic ..................................................................................................................... 88
  B inverter and multiplexer .......................................................................................... 89
  Carry-in multiplexer .................................................................................................... 89
  Borrow multiplexer ..................................................................................................... 90
  Adder .............................................................................................................................. 90
  Borrow adder ............................................................................................................... 91
AND ............................................................................................................................... 92
OR .................................................................................................................................. 93
XOR ............................................................................................................................... 94
NOT ............................................................................................................................... 95
ALU output multiplexer ............................................................................................... 96
Control board schematic ............................................................................................. 97
  State register ............................................................................................................... 99
  Next-state logic ........................................................................................................... 100
  Front panel connector ................................................................................................. 101
  State decoder .............................................................................................................. 101
  Control connector ....................................................................................................... 102
  Control logic (portion) ............................................................................................... 102
  Clocks and reset ......................................................................................................... 102
  Clock delay .................................................................................................................. 103
  Write signal machine ................................................................................................. 104
  Register write pulse machine .................................................................................... 104
Appendix ....................................................................................................................... 105
  ALU parts organizer ................................................................................................... 105
  ALU parts list .............................................................................................................. 106
Main board parts organizer
Main board parts list
Control board parts organizer
Control board parts list
ALU carry-out logic explanation
ALU logic explanation
Next-state logic explanation
Control signals logic explanation
System write signal timing
Register write pulse timing
Special Programming Techniques
  Indexing
  Subroutines
Instruction set table, sorted by opcode
Instruction set table, sorted by mnemonic
Selected Program Listings
  ROM for 4K systems
  adder
  ROM System Monitor
**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

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:

   ![Image of soldered leads]

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:

![Pads with connections to zones](image)

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 it flows over to another pad. This is bad, because it causes a short circuit, and can damage parts.

![Solder bridge](image)

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:
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:
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:

![Image of ICs in control board]

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

![Image of M/F stand-offs]

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\textsuperscript{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:

\textsuperscript{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.
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:
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:
Here is a dump display of RAM starting at location 0x0900. You can see the 16 bytes I entered:
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 (binary 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. First we need the exact file size, which we can obtain by hovering over the file name, or right-click-Properties:

![PL9 Properties](image)

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 PL_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.
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:
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:
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 (jump if minus), JPC (jump if carry), and JPZ (jump if zero).

There is a no-operation instruction (NOP) for placeholder 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**

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*

This instruction adds the instruction operand and the carry flag to the accumulator.
ADD – add memory data to the accumulator

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.
**ADIM – add immediate data to the accumulator**

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

- **Program counter (PC)**: mm mm → mmmm+2
- **Opcodes**: 08
- **Operand**: dd
- **Accumulator**: aa* ↔ aa + dd
- **Carry flag**: c*
- **Minus flag**: m*
- **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 *
**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 \( dd = 1011\ 0101_2 \) and \( aa = 1100\ 0011_2 \):

\[
\begin{array}{c}
1011\ 0101 \\
\text{AND} \quad 1100\ 0011 \\
1000\ 0001 \\
\end{array}
\]

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.

\[\begin{array}{|c|c|c|}
\hline
\text{Program counter (PC)} & \text{mm} & \text{mm} \\
\hline
\text{Opcode} & 04 & \\
\hline
\text{Operand} & hh & ll \\
\hline
\text{Accumulator} & aa^* & \\
\hline
\text{Carry flag} & c & \\
\hline
\text{Minus flag} & m^* & \\
\hline
\text{Zero flag} & z^* & \\
\hline
\text{Data memory} & mmmm+3 & \\
\hline
\text{dd} & \\
\hline
\text{Program memory} & 04 & \\
& mmmm & \\
& ll & mmmm+1 \\
& hh & mmmm+2 \\
& & mmmm+3 \\
\hline
\end{array}\]
ANDIM – bitwise logical AND of immediate data with the accumulator

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

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.

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>mm</th>
<th>mm</th>
<th>mmmm+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>1C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>aa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry flag</td>
<td>c*</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Minus flag</td>
<td>m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero flag</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data memory

<table>
<thead>
<tr>
<th>Program memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmmm</td>
</tr>
<tr>
<td>1C</td>
</tr>
<tr>
<td>mmmm+1</td>
</tr>
</tbody>
</table>

`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 `*`
**CMP – compare**

Program counter (PC) \[ \begin{array}{c} \text{mm} \\ \text{mm} \end{array} \quad \text{mmmm+2} \]

Opcode \[ \text{0F} \]

Operand \[ \text{dd} \]

Accumulator \[ \text{aa} \rightarrow \text{aa - dd} \]

Carry flag \[ \text{c*} \]
- Carry-out = 0 if borrow, that is, if \( \text{dd} > \text{aa} \)

Minus flag \[ \text{m} \]

Zero flag \[ \text{z} \]

- \text{mmmm} = 16-bit program memory address
- \text{hhll} = 16-bit memory address operand
- \text{pp} = 8-bit port address operand
- \text{aa, dd} = 8-bit data
- \text{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 \( \text{dd} > \text{aa} \), 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**

This instruction subtracts 1 from the accumulator, and the result is placed back in the accumulator. If \( \text{aa} < 1 \) (that is, if \( \text{aa} = 0 \)) 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

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**

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

**JMP – jump unconditional**

Program counter (PC) | mm | mm
--- | --- | ---
Opcode | 13 |
Operand | hh | ll |
Accumulator | aa |
Carry flag | c |
Minus flag | m |
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 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**

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**

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>mm</th>
<th>mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Operand</td>
<td>hh</td>
<td>ll</td>
</tr>
<tr>
<td>Accumulator</td>
<td>aa</td>
<td></td>
</tr>
<tr>
<td>Carry flag</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Minus flag</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Zero flag</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

If \( z = 0 \)

If \( z = 1 \)

Data memory

<table>
<thead>
<tr>
<th>Program memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmmm</td>
</tr>
<tr>
<td>mmmm+1</td>
</tr>
<tr>
<td>mmmm+2</td>
</tr>
<tr>
<td>mmmm+3</td>
</tr>
</tbody>
</table>

\( mmmm = \) 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**

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>mm</th>
<th>mm</th>
<th>mm</th>
<th>Data memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand</td>
<td>dd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>aa*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry flag</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minus flag</td>
<td>m*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero flag</td>
<td>z*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mmmmm = 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**

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.

- Program counter (PC): `mm` `mm` `mmmm+3`
- Opcode: `11`
- Operand: `hh` `ll`
- Accumulator: `aa*`
- Carry flag: `c`
- Minus flag: `m*`
- 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 *
**NOP – no operation**

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>mm</th>
<th>mm</th>
<th>mmmm+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>1F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>aa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry flag</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minus flag</td>
<td>m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero flag</td>
<td>z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data memory

<table>
<thead>
<tr>
<th>Program memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmmm</td>
</tr>
<tr>
<td>1F</td>
</tr>
<tr>
<td>mmmm+1</td>
</tr>
</tbody>
</table>

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**

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

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 \( dd = 1011 \ 0101_b \) and \( aa = 1100 \ 0011_b \):

\[
\begin{array}{c}
1011 \ 0101 \\
OR \ 1100 \ 0011 \\
1111 \ 0111
\end{array}
\]

The value \( 1111 \ 0111_b \) 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**

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>mm</th>
<th>mm</th>
<th>mmmm+2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>0D</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Operand</strong></td>
<td></td>
<td>dd</td>
<td></td>
</tr>
<tr>
<td><strong>Accumulator</strong></td>
<td>aa*</td>
<td></td>
<td>aa OR dd</td>
</tr>
<tr>
<td><strong>Program memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry flag</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minus flag</td>
<td>m*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero flag</td>
<td>z*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmmm = 16-bit program memory address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hhll = 16-bit memory address operand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pp = 8-bit port address operand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aa, dd = 8-bit data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c, m, z = one bit flags</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If the accumulator data or flags are affected by the instruction they are marked by *</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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**

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

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:
```
\[ 1100 \ 0111 \]
\[ + \ 0101 \ 1000 \]
\[ 0001 \ 1111, \text{ carry-out } = 1 \text{ (no borrow)} \]

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.
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:
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**

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>mm</th>
<th>mm</th>
<th>mmmm+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>1B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>aa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry flag</td>
<td>c*</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Minus flag</td>
<td>m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero flag</td>
<td>z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Data memory**

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.

\[
\text{mmmm} = 16\text{-bit program memory address} \\
\text{hhll} = 16\text{-bit memory address operand} \\
\text{pp} = 8\text{-bit port address operand} \\
\text{aa, dd} = 8\text{-bit data} \\
\text{c, m, z} = \text{one bit flags} \\
\text{If the accumulator data or flags are affected by the instruction they are marked by *}
\]
STM – store accumulator to memory

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**

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**

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 0111\(^b\) and \(dd\) is 1011 1001\(^b\), the following operation is performed:

\[dd: \quad 1011 \ 1001\]

one’s complement of \(dd\): 0100 0110

two’s complement of \(dd\): 0100 0111

\(aa\) + two’s complement of \(dd\):
\[ \begin{array}{c}
0011 \ 0111 \\
+ \ 0100 \ 0111 \\
= \ 0111 \ 1110, \ \text{carry-out} = 0 \ (\text{borrow})
\end{array} \]

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**

<table>
<thead>
<tr>
<th>Program counter (PC)</th>
<th>Data memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>mm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand</th>
<th>Accumulator</th>
<th>Carry flag</th>
<th>Minus flag</th>
<th>Zero flag</th>
<th>Program memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>hh ll</td>
<td>aa*</td>
<td>c</td>
<td>m*</td>
<td>z*</td>
<td>06</td>
</tr>
</tbody>
</table>

|    |    |    |    |    |    |    |
| 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

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 \( aa = 1010\ 0111b \):

\[
\begin{array}{c}
0011\ 0101 \\
\text{OR} \ 1010\ 0111 \\
\hline
1001\ 0010
\end{array}
\]

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:
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).

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.
The display connector has outputs of the main board registers and flags for passing to a register display board, which is an accessory.
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.
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).
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.
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.
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.
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.
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 two's-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.
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.
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.
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.
This is a simple inversion operation, performed on the ALU A input. The output is sent to the ALU output multiplexer inputs.
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:

![Block Diagram of Control Board](image)

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.
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.
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 needing 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

<table>
<thead>
<tr>
<th>Capacitor, 0.01uF</th>
<th>74LS00 quad NAND</th>
<th>74LS04 hex inverter</th>
<th>74LS08 quad AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>74LS138 1-of-8 decoder</th>
<th>DIL 14-pin socket</th>
<th>74LS151 8-input multiplexer</th>
<th>DIL 16-pin socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>8</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>74LS153 dual 4-input multiplexer</th>
<th>74LS157 quad 2-input multiplexer</th>
<th>74LS283 4-bit full adder</th>
<th>74LS32 quad OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>74LS86 quad XOR</th>
<th>40-pin header</th>
<th>Resistor, 470 ohm, 1/4 watt Yellow-Violet-Brown</th>
<th>LED</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**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.

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacitor, 0.01uF</td>
<td>6</td>
</tr>
<tr>
<td>74LS04 hex inverter</td>
<td>1</td>
</tr>
<tr>
<td>74LS153 dual 4-input multiplexers</td>
<td>10</td>
</tr>
<tr>
<td>74LS157 quad 2-input multiplexer</td>
<td>4</td>
</tr>
<tr>
<td>74LS161 binary counter</td>
<td>4</td>
</tr>
<tr>
<td>74LS175 quad D flip-flop</td>
<td>8</td>
</tr>
<tr>
<td>74LS244 octal buffer</td>
<td>1</td>
</tr>
<tr>
<td>74LS32 quad OR</td>
<td>2</td>
</tr>
<tr>
<td>74LS74 dual D flip-flop</td>
<td>1</td>
</tr>
<tr>
<td>40-pin header</td>
<td>2</td>
</tr>
<tr>
<td>50-pin header</td>
<td>1</td>
</tr>
<tr>
<td>40-pin header receptacle</td>
<td>1</td>
</tr>
<tr>
<td>14-pin socket</td>
<td>4</td>
</tr>
<tr>
<td>16-pin socket</td>
<td>26</td>
</tr>
<tr>
<td>20-pin socket</td>
<td>1</td>
</tr>
<tr>
<td>Power-in jack</td>
<td>1</td>
</tr>
<tr>
<td>Resistor, 470 ohm Yellow-Violet-Brown</td>
<td>1</td>
</tr>
<tr>
<td>LED</td>
<td>1</td>
</tr>
</tbody>
</table>
## Main board parts list

<table>
<thead>
<tr>
<th>Part Description</th>
<th>PCB reference</th>
<th>Number per unit</th>
<th>Jameco Part No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cap 0.01 uF</td>
<td>C1 – C6</td>
<td>6</td>
<td>15229</td>
</tr>
<tr>
<td>74LS04</td>
<td>U7</td>
<td>1</td>
<td>46316</td>
</tr>
<tr>
<td>74LS153</td>
<td>U19 – U24, U28 – U31</td>
<td>10</td>
<td>46720</td>
</tr>
<tr>
<td>74LS157</td>
<td>U8 – U11</td>
<td>4</td>
<td>46771</td>
</tr>
<tr>
<td>74LS161</td>
<td>U1 – U4</td>
<td>4</td>
<td>46818</td>
</tr>
<tr>
<td>74LS175</td>
<td>U12 – U17, U26, U27</td>
<td>8</td>
<td>46957</td>
</tr>
<tr>
<td>74LS244</td>
<td>U18</td>
<td>1</td>
<td>47183</td>
</tr>
<tr>
<td>74LS32</td>
<td>U5, U6</td>
<td>2</td>
<td>47466</td>
</tr>
<tr>
<td>74LS74</td>
<td>U25</td>
<td>1</td>
<td>48004</td>
</tr>
<tr>
<td>40-pin header</td>
<td>P2, P4</td>
<td>2</td>
<td>53532</td>
</tr>
<tr>
<td>50-pin header</td>
<td>P1</td>
<td>1</td>
<td>53560</td>
</tr>
<tr>
<td>40-pin header receptacle</td>
<td>P3</td>
<td>1</td>
<td>111705</td>
</tr>
<tr>
<td>14-pin socket</td>
<td></td>
<td>4</td>
<td>112214</td>
</tr>
<tr>
<td>16-pin socket</td>
<td></td>
<td>26</td>
<td>112222</td>
</tr>
<tr>
<td>20-pin socket</td>
<td></td>
<td>1</td>
<td>112248</td>
</tr>
<tr>
<td>Power-in jack</td>
<td></td>
<td>1</td>
<td>137673</td>
</tr>
<tr>
<td>470 ohm</td>
<td>R1</td>
<td>1</td>
<td>690785</td>
</tr>
<tr>
<td>LED (red)</td>
<td>D1</td>
<td>1</td>
<td>2081932</td>
</tr>
<tr>
<td>Control board parts organizer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Capacitor, 0.01uF</strong></td>
<td><strong>Oscillator, 1.8432 MHz</strong></td>
<td><strong>4-position DIP switch</strong></td>
<td><strong>74LS00 quad NAND</strong></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>74LS04 hex inverter</strong></td>
<td><strong>74LS08 quad AND</strong></td>
<td><strong>74LS138 1-of-8 decoder</strong></td>
<td><strong>74LS139 dual 1-of-4 decoder</strong></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>74LS14 hex inverter</strong></td>
<td><strong>74LS174 hex D flip-flop</strong></td>
<td><strong>74LS74 dual D flip-flop</strong></td>
<td><strong>74LS75 quad latch</strong></td>
</tr>
<tr>
<td><strong>Schmitt trigger</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>GAL 16V8-D programmable logic</strong></td>
<td><strong>16-pin header</strong></td>
<td><strong>40-pin header receptacle</strong></td>
<td><strong>Capacitor, 22 uF</strong></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>14-pin socket</strong></td>
<td><strong>16-pin socket</strong></td>
<td><strong>20-pin socket</strong></td>
<td><strong>Pushbutton switch</strong></td>
</tr>
<tr>
<td>17</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>Resistor, 470 ohm</strong></td>
<td><strong>Resistor, 1K ohm</strong></td>
<td><strong>Resistor, 2.2K ohm</strong></td>
<td><strong>Resistor, 100K ohm</strong></td>
</tr>
<tr>
<td><strong>Yellow-Violet-Brown</strong></td>
<td><strong>Brown-Black-Red</strong></td>
<td><strong>Red-red-red</strong></td>
<td><strong>Brown-black-yellow</strong></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>LED (red)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Control board parts list

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

<table>
<thead>
<tr>
<th>Borrow Select</th>
<th>Adder C.O.</th>
<th>Borrow C.O.</th>
<th>Final C.O.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

$$\text{FinalCO} = (\text{NOT}(\text{BorrowSelect}) \text{ AND } \text{AdderCO}) \text{ OR } (\text{AdderCO AND } \text{BorrowCO})$$

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.

<table>
<thead>
<tr>
<th>ALU Op Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Logic operation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output select 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>NOT /7</td>
</tr>
<tr>
<td>Output select 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NAND /5,/6</td>
</tr>
<tr>
<td>Output select 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>NAND /4,/6</td>
</tr>
<tr>
<td>Carry select 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOT /1</td>
</tr>
<tr>
<td>Carry select 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NAND /2,/3</td>
</tr>
<tr>
<td>B invert</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NAND /2,/3</td>
</tr>
<tr>
<td>Borrow select</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Carry-in NOR /3</td>
</tr>
</tbody>
</table>

*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:

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

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:

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

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:

<table>
<thead>
<tr>
<th>State</th>
<th>Opcode (5 bits)</th>
<th>Zero</th>
<th>Minus</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000</td>
<td>X X XXXX</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 0000</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 0001</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 0010</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 0011</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 0100</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 0101</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>1 0010</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>1 0011</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>1 1000</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>1 1010</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>0 0000</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>0 0001</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>0 0010</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>0 0011</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>0 0100</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>0 0101</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>1 0010</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>1 0011</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>1 1000</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0011</td>
<td>1 1010</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 1000</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 1010</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 1011</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0010</td>
<td>0 1110</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0001</td>
<td>1 0100</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0 0001</td>
<td>1 1100</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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:
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:

\[
NS_0 = \text{NOT}(S_4) \text{ AND NOT}(S_3) \text{ AND NOT}(S_2) \text{ AND NOT}(S_1) \text{ AND NOT}(S_0) \text{ OR AND}
\]

\[
\text{NOT}(S_4) \text{ AND NOT}(S_3) \text{ AND NOT}(S_2) \text{ AND NOT}(S_1) \text{ AND } S_0 \text{ AND } OP_4 \text{ AND OP}_3 \text{ AND NOT(OP}_2) \text{ AND OP}_1 \text{ AND NOT(OP}_0) \text{ OR ...}
\]

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

\[
NS_0 = \text{STATE}_0 \text{ OR STATE}_1 \text{ AND OP}_4 \text{ AND OP}_3 \text{ AND NOT(OP}_2) \text{ AND OP}_1 \text{ AND NOT(OP}_0) \text{ OR}
\]

\[
\text{STATE}_2 \text{ AND NOT(OP}_4) \text{ AND NOT(OP}_3) \text{ AND NOT(OP}_2) \text{ OR} ...
\]

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 |

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:
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

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

* Clock is synchronous with register write clock pulses
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 \( \Delta 4 \) 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 \( \Delta 4 \) 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

```tasm
#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:` 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:
  ```tasm
  .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

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

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

**ROM for 4K systems**

0001  0000 ;Test programs for 8-bit TTL computer.
0002  0000 .ORG 0000H ;Get address from switches and jump to it
0003  0000 10 13  LDI 13H ;JMP instruction
0004  0002 12 00 08  STM 0800H ;Start of RAM
0005  0005 17 00  IN  00H ;Low byte of jump target
0006  0007 12 01 08  STM 0801H
0007  000A 17 01  IN  01H ;High byte of jump target
0008  000C 12 02 08  STM 0802H ;Full jump instruction in place now
0009  000F 13 00 08  JMP 0800H ;Jump to the jump instruction
0010  0012 ;Simple port reflector
0011  0012 17 00 LOOP:
0012  0014 18 00  OUT 00H
0013  0016 17 01  IN  01H
0014  0018 18 01  OUT 01H
0015  001A 13 12 00  JMP LOOP
0016  001D ;Simple counter -- run with slow clock
0017  001D 10 00  LDI 00H
0018  001F 18 00 LOOPA:
0019  0021 19  INC
0020  0022 13 1F 00  JMP LOOPA
0021  0025 ;Two-byte up counter -- run with fast clock
0022  0025 10 00  LDI 00H
0023  0027 12 00 08  STM 0800H ;Hi byte
0024  002A 18 01  OUT 01H ;Clear port 1 LEDs
0025  002C 18 00 LOOPB:
0026  002E 19  INC
0027  002F 14 35 00  JPZ NEXTB ;If zero, jump to increment high byte
0028  0032 13 2C 00  JMP LOOPB ;Low-byte increment loop
0029  0035 11 00 08 NEXTB:
002A  0038 19  INC
002B  0039 18 01  OUT 01H ;Output high byte to port 1 LEDs
002C  003B 12 00 08  STM 0800H ;Store high byte
002D  003E 10 00  LDI 00H
002E  0040 13 2C 00  JMP LOOPB ;Go back to low-byte increment loop
0035  0043 ;8-bit highest factor routine
; Factor test
FACSTRT:IN

; Number to factor

STM ORIG
STM TESTF
LDM TESTF
DEC
STM TESTF
LDM ORIG
SUB TESTF
JPZ DONE ; Factor found
JPC LOOP16 ; For A - B, carry set if A >= B
JMP LOOP15 ; No carry, means A < B, and not a factor
DONE:
LDM TESTF
OUT 00H

; Test for serial interface with 8-bit code
Set port to 9600 baud, 8-bit, no parity, 1 stop bit
LDI 4EH ; 1 stop bit, no parity, 8-bit char, 16x baud
OUT 03H ; write to UART control port
LDI 37H ; enable receive and transmit
OUT 03H ; write to control port
LOOP1:
IN 03H ; get status
ANDIM 01H ; check TxRDY bit
JPZ LOOP1 ; not ready, loop
IN 02H ; get char from data port
OUT 00H ; put on LEDs
STM TEMP ; store the character
LOOP2:
IN 03H ; get status
ANDIM 01H ; check TxRDY bit
JPZ LOOP2 ; loop if not ready
LDM TEMP ; get char back
OUT 02H ; send to UART for output
JMP LOOP1 ; start over

; Program loader
Takes input from serial port, creates byte values from hex character pairs
Loads byte values sequentially into RAM starting at 0x0810
Jumps to location 0x0810 to start execution upon receiving return character
Quits without execution if invalid hex character input received
Setup routine for serial port
LDI 4EH ; 1 stop bit, no parity, 8-bit char, 16x baud
0076 0090 18 03  OUT 03H ;write to UART control port
0077 0092 10 37  LDI 37H ;enable receive and transmit
0078 0094 18 03  OUT 03H ;write to control port
0079 0096 10 12  LDI 12H ;STM instruction
0080 0098 12 00 08  STM STORE_BYTE
0081 009A 12 01 08  STM STORE_BYTE+1
0082 009C 10 08  LDI 08H ;Hi byte of storage buffer start address
0083 009E 12 02 08  STM STORE_BYTE+2
0084 009F 12 03 08  STM STORE_BYTE+3
0085 00A1 11 9E 01  LDM RETURN
0086 00A3 12 04 08  STM STORE_BYTE+4
0087 00A5 12 05 08  STM STORE_BYTE+5
0088 00A7 10 13  LDI 13H ;JMP instruction for return
0089 00A9 12 03 08  STM STORE_BYTE+6
008A 00AA 11 9F 01  LDM RETURN+1 ;Hi byte of return address
008B 00AB 12 05 08  STM STORE_BYTE+7
008C 00AC 10 10  LDI 16
008D 00AD 12 09 08  STM BYTE_COUNTER ;initialize line length variable
008E 00AE 17 03  GET_HI:  IN 03H ;Get hi-order nybble of pair
008F 00B0 0C 02  ANDIM 02H ;check RxRDY bit
0090 00B2 14 BB 00  JPZ GET_HI ;not ready, loop
0091 00B4 17 03  IN 03H ;get char from data port
0092 00B6 12 06 08  STM TEMP ;Store character
0093 00B8 14 B2 01  JPZ RUN ;Yes, run code
0094 00BA 17 03  LOOP3:  IN 03H ;No, output character and validate
0095 00BD 0C 02  ANDIM 02H ;check TxRDY bit
0096 00BF 14 BB 00  JPZ LOOP3 ;loop if not ready
0097 00C0 11 06 08  LDM TEMP ;get char back
0098 00C2 18 02  OUT 02H ;send to UART for output
0099 00C4 14 B2 01  JPZ RUN ;Yes, run code
0100 00C6 17 03  LOOP:  IN 03H ;No, output character and validate
0101 00C8 0C 01  ANDIM 01H ;check TxRDY bit
0102 00CA 14 CC 00  JPZ LOOP ;loop if not ready
0103 00CC 11 06 08  LDM TEMP ;get char back
0104 00CD 18 02  OUT 02H ;send to UART for output
0105 00CE 0F 30  CMP 30H ;Lower limit of hex characters
0106 00CF 16 E0 00  JPC NEXT1 ;Char => 30H, possibly valid
0107 00D0 13 FE 00  JMP INVALID ;Char < 30H, invalid hex char
0108 00D2 0F 30  CMP 30H ;Lower limit of hex characters
0109 00D3 16 E0 00  JPC NEXT1 ;Char => 30H, possibly valid
0110 00D4 13 FE 00  JMP INVALID ;Char < 30H, invalid hex char
0111 00D6 0F 47  NEXT1:  CMP 47H ;ASCII for "G"
0112 00D7 16 FE 00  JMP INVALID ;Char is G or greater, invalid
0113 00D8 0F 41  CMP 41H ;ASCII for "A"
0114 00DA 16 F2 00  JMP INVALID ;Char is valid A-F
0115 00DB 0F 3A  CMP 3AH ;ASCII for ":"
0116 00EC 16 FE 00                JPC INVALID ;Char is ";" or greater, but < "A", invalid
0117 00EF 13 F9 00                JMP VALID09_HI ;Char is valid 0-9
0118 00F2 0C 0F                  VALIDAF_HI: ANDIM 0FH ;Mask off high bits
0119 00F4 08 09                  ADDIM 9 ;Adjust ASCII to binary value
0120 00F6 13 01 01                JMP SHIFT_HI
0121 00F9 0C 0F                  VALID09_HI: ANDIM 0FH ;Mask off high bits
0122 00FB 13 01 01                JMP SHIFT_HI
0123 00FE 13 B5 01                INVALID: JMP ERROR ;Invalid hex char, quit
0124 0101 12 07 08               SHIFT_HI: STM BYTE ;Will eventually contain the byte to load
0125 0104 12 06 08               STM TEMP ;Value to add
0126 0107 10 10                  LDI 10H ;Multiply x 16 to shift into high-order nybble
0127 0109 12 08 08                STM COUNTER
0128 010C 11 08 08               MULTLOOP: LDM COUNTER
0129 010F 1A                      DEC
0130 0110 14 22 01               JPZ GET_LO ;Have added 16 times, done
0131 0113 12 08 08                STM COUNTER
0132 0116 11 06 08                LDM TEMP ;Original nybble
0133 0119 00 07 08                ADD BYTE ;Add to BYTE and store
0134 011C 12 07 08                STM BYTE
0135 011F 13 0C 01                JMP MULTLOOP ;Keep adding
0136 0122 17 03                  GET_LO: IN 03H ;Get lo-order nybble of pair
0137 0124 0C 02                  ANDIM 02H ;check RxRDY bit
0138 0126 14 22 01               JPZ GET_LO ;not ready, loop
0139 0129 17 02                  IN 02H ;get char from data port
0140 012B 12 06 08                STM TEMP ;Store character
0141 012D 17 03                  LOOP4: IN 03H ;Output character
0142 0130 0C 01                  ANDIM 01H ;check TxRDY bit
0143 0132 14 2E 01                JPZ LOOP4
0144 0135 11 06 08                LDM TEMP ;When ready, retrieve character and output
0145 0138 18 02                  OUT 02H
0146 013A 17 03                  LOOP5: IN 03H
0147 013C 0C 01                  ANDIM 01H
0148 013E 14 3A 01                JPZ LOOP5
0149 0141 10 20                  LDI 20H ;Space character
0150 0143 18 02                  OUT 02H ;send to UART for output
0151 0145 11 09 08               ;Check if 16 bytes have been displayed. If so, write newline
0152 0145 11 09 08                LDM BYTE_COUNTER ;Check if 16 bytes have been displayed
0153 0148 1A                      DEC
0154 0149 12 09 08                STM BYTE_COUNTER
0155 014C 14 52 01                JPZ NEXT4 ;Yes, reset counter and write newline

129
JMP NEXT5 ;No, keep going
LDI 16
STM BYTE_COUNTER
IN 03H
ANDIM 01H
JPZ LOOP6
OUT 02H ;send to UART for output
IN 03H
ANDIM 01H
JPZ LOOP7
IN 03H
OUT 02H ;send to UART for output
OUT 02H ;send to UART for output
LDM TEMP ;Retrieve limit of hex characters
JMP NEXT2 ;Char >= 30H, possibly valid
JMP INVALID ;Char < 30H, invalid hex char
CMPI "G" ;ASCII for "G"
CMPI "A" ;ASCII for "A"
CMPI ":" ;ASCII for "A"
CMPI ":" ;ASCII for "A"
CMPI 09H ;Char is valid 0-9
ANDIM 0FH ;Mask off high bits
ADDIM 9 ;Now lo nybble correct
ADD BYTE ;Combine with hi nybble stored in BYTE
JMP STORE ;Store the byte in RAM
ANDIM 0FH ;Mask off high bits
ADD BYTE ;Now full byte assembled
OUT 00H ;Display on LEDs
JMP STORE_BYTE ;Store the byte in RAM
.DW $+2 ;Address to return from storage instruction
LDM STORE_BYTE+1 ;Increment byte pointer, lo byte first
INC
STM STORE_BYTE+1
LDM STORE_BYTE+2 ;Increment hi byte if a carry occurred when lo
byte incremented
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\ ldm 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\ ldm 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\ ldm return_jump2+2
0010 0000 #defcont \ jmp address\ .dw $+2
0011 0000 #define ret2 \ jmp return_jump2
0012 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 0800h ;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 ;Initialize port
0022 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 0009 10 13 ldi 13h ;jmp opcode
0030 000B 12 1E 08 stm ld Indexed ld+3 ;return jumps for indexed instructions
0031 000E 12 24 08 stm d Indexed ld+3
0032 0011 12 2A 08 stm d Indexed ld+3
0033 0014 12 30 08 stm bl Indexed ld+3
0034 0017 12 36 08 stm ws inst+3
0035 001A 12 3C 08 stm gl Indexed ld+3
stm return_jump0
stm return_jump1
stm return_jump2
stm run_jump

stm

ldi 12h ;stm opcode
stm ld_indexed_stm ;indexed store instructions
stm d_indexed_stm
stm bl_indexed_stm
stm gl_indexed_stm

ldi 11h ;ldm opcode
stm d_indexed_ldm
stm ws_inst

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

ldi 59h ;address of d_ldm_back
stm d_indexed_ldm+4
ldi 01h
stm d_indexed_ldm+5

ldi 01h ;address of d_stm_back
stm d_indexed_ldm+4
ldi 00h
stm d_indexed_ldm+5

ldi 01h
stm d_indexed_stm+4
ldi 04h
stm gl_indexed_stm+4

ldi 05h ;address of ws_back
stm ws_inst+4
ldi 05h
stm ws_inst+5

ldi 0C0h ;address of gl_back
stm gl_indexed_stm+4
ldi 05h
stm gl_indexed_stm+5

;Print greeting
ldi 0B2h ;address of bl_back
stm bl_indexed_stm+4
ldi 04h
stm bl_indexed_stm+5

ldi 01h
stm gl_indexed_stm+4
ldi 05h
stm gl_indexed_stm+5

;Print greeting
ldi 0C0h ;address of gl_back
stm ws_inst+4
ldi 05h
stm ws_inst+5

ldi 0C0h
stm gl_indexed_stm+4
ldi 05h
stm gl_indexed_stm+5

;Print greeting
ldi 0C0h
stm ws_inst+4
ldi 05h
stm ws_inst+5

ldi 0C0h
stm gl_indexed_stm+4
ldi 05h
stm gl_indexed_stm+5
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'
;Memory dump routine
;jmp get_address
;Get address from input string
;jmp get_address
;Dump 16 lines of 16 characters each
;set up line counter
;Start with 4 characters of the starting address of the line, followed by space
;d_line_loop;high byte of memory address
;low byte of memory address
;start of line
;Start with 4 characters of the starting address of the line, followed by space
;high byte of memory address
;low byte of memory address
;start of line
;Start with 4 characters of the starting address of the line, followed by space
;high byte of memory address
;low byte of memory address
;start of line
;Start with 4 characters of the starting address of the line, followed by space
;high byte of memory address
;low byte of memory address
ldm char_pair
stm buffer+2
ldm char_pair+1
stm buffer+3
ldi 20h ;space character
stm buffer+4

;Set up for getting 16 memory bytes, converting to characters, and putting in string buffer
ldi buff_low
addim 5
stm d_indexed_stm+1 ;low byte of location of first character in output string
ldi buff_high
adcim 0 ;16-bit addition
stm d_indexed_stm+2 ;high byte of location of first character in output string
ldi 16
stm byte_counter ;number of bytes to get, convert, and display in one line
d_byte_loop: jmp d_indexed_ldm ;get byte from memory
d_ldm_back: stm byte
call0(byte_to_hex_pair) ;convert to hex pair

stm byte_counter
ldi 3
stm nybble_counter
ldm char_pair
d_nybble_loop: jmp d_indexed_stm ;store char of byte in string buffer
d_stm_back: ldm d_indexed_stm+1 ;increment pointer by 16-bit incrementation
stm d_indexed_stm+1
ldm d_indexed_stm+2
adcim 0
stm d_indexed_stm+2 ;pointing to next spot in buffer
ldm nybble_counter
stm nybble_counter ;pointing to next spot in buffer
adcim 0
stm nybble_counter ;pointing to next spot in buffer
;all three characters stored (hex chars plus space)?
jpz d_nybble_done ;yes, next byte
stm nybble_counter ;no, place next character or space
subim 1 ;if nybble count = 1, put a space next
jpz d_put_space
ldm char_pair+1 ;otherwise, get next char and store
jmp d_nybble_loop
d_put_space:  ldi 20h ;space character
jmp d_nybble_loop
d_nybble_done:  ldm d_indexed_ldm+1 ;increment memory pointer
inc
stm d_indexed_ldm+1
ldm d_indexed_ldm+2
adcim 0
stm d_indexed_ldm+2
ldm byte_counter
dec
stm d_indexed_ldm+2
ldm byte_counter
jmp d_byte_loop
d_line_done:  ldi 0dh ;newline characters
stm buffer+52
ldi 0ah
stm buffer+53
ldi 0
stm buffer+54 ;where the end of the line will be
;Write string to screen
ldi buff_low
stm ws_inst+1
ldi buff_high
stm ws_inst+2
call0(write_string)
0193 01DC 11 E6 01
0193 01DF 12 41 08
0193 01E2 13 E7 05
0193 01E5 E7 01
0194 01E7
0195 01E7 ;Check if 16 lines done
0196 01E7 11 14 08  ldm  line_counter
0197 01EA 1A
dec
0198 01EB 14 F4 01  jpz  d_done
0199 01EE 12 14 08  stm  line_counter
0200 01F1 13 F8 00  jmp  d_line_loop
0201 01F4  d_done:
0202 01F4 13 98 00  jmp  sm_warm
0203 01F7
0204 01F7 ;Monitor routine to jump and execute code
0205 01F7 ;Gets target address from terminal
0206 01F7
0207 01F7 13 E2 04  sm_run   jmp  get_address
0208 01FA 11 0E 08  run_addr_back   ldm  address
0209 01FD 12 19 08  stm  run_jump+1
0210 0200 11 0F 08  ldm  address+1
0211 0203 12 1A 08  stm  run_jump+2
0212 0206 13 18 08  jmp  run_jump
0213 0209
0214 0209 ;Routine to get hex char pairs from input and load bytes in RAM
0215 0209 ;Get address first
0216 0209 13 E2 04  sm_load   jmp  get_address
0217 020C 11 0E 08  ld_addr_back   ldm  address
0218 020F 12 1C 08  stm  ld_indexed_stm+1
0219 0212 11 0F 08  ldm  address+1
0220 0215 12 1D 08  stm  ld_indexed_stm+2
0221 0218 ;Initialize display bytes counter
0222 0218 10 10  ldi  10h ;16 bytes per line
0223 021A 12 16 08  stm  byte_counter
0224 021D ;Get characters
0225 021D ;First character of pair
0226 021D 17 03  ld_get_hi:  IN  03H ;Get hi-order nybble of pair
0227 021F 0C 02  ANDIM  02H ;check RxRDY bit
0228 0221 14 1D 02  JPZ  ld_get_hi ;not ready, loop
0229 0224 17 02  IN  02H ;get char from data port
STM temp ;Store character
SUBIM 0DH ;Carriage return?
JPZ ld_done ;Yes, return to monitor
ld_loop_1: IN 03H ;No, output character and validate
ANDIC 01H ;Check TxRDY bit
JPC ld_loop_1 ;Loop if not ready
LDM temp ;Get char back
OUT 02H ;Send to UART for output
;Code to validate hex character
CMP 30H ;Lower limit of hex characters
JPC ld_next_1 ;Char >= 30H, possibly valid
JMPC ld_invalid ;Char < 30H, invalid hex char
ld_next_1: CMP 47H ;ASCII for "G"
JPC ld_invalid ;Char is G or greater, invalid
CMP 41H ;ASCII for "A"
JMP ld_validAF_hi ;Char is valid A-F
CMP 3AH ;ASCII for ":"
JPC ld_invalid ;Char is ":" or greater, but < "A", invalid
JMPC ld_valid09_hi ;Char is valid 0-9
ld_validAF_hi: ANDIC 0FH ;Mask off high bits
ADDIC 9 ;Adjust ASCII to binary value
JMPC ld_shift_hi
ld_valid09_hi: ANDIC 0FH ;Mask off high bits
JMPC ld_shift_hi
ld_invalid: JMP ld_error ;Invalid hex char, quit
JMPC ld_shift_hi ;Will eventually contain the byte to load
STM byte
LDI 10H ;Multiply x 16 to shift into high-order nybble
LDM counter
STM multloop: LDM counter
DEC
JMPC ld_get_lo ;Have added 16 times, done
STM counter
LDM temp ;Original nybble
ADD byte ;Add to BYTE and store
STM byte
JMPC ld_multloop ;Keep adding
JMPC ld_get_lo ;Not ready, loop
0270 028B 17 02   IN  02H   ;get char from data port
0271 028D 12 10 08 STM temp   ;Store character
0272 0290 17 03 ld_loop2: IN 03H   ;Output character
0273 0292 0C 01 ANDIM 01H   ;check TxRDY bit
0274 0294 14 90 02 JPZ ld_loop2   ;When ready, retrieve character and output
0275 0297 11 10 08 LDM temp
0276 029A 18 02 OUT 02H
0277 029C 17 03 ld_loop3: IN 03H
0278 029E 0C 01 ANDIM 01H
0279 02A0 14 9C 02 JPZ ld_loop3   ;Space character
0280 02A3 10 20 LDI 20H
0281 02A5 18 02 OUT 02H   ;send to UART for output
0282 02A7 11 16 08 ldm byte_counter   ;Check if 16 bytes have been displayed
0283 02AA 1A dec
0284 02AB 12 16 08 stm byte_counter
0285 02AE 14 B4 02 jnz ld_next4   ;Yes, reset counter and write newline
0286 02B1 13 CF 02 jmp ld_next5   ;No, keep going
0287 02B4 10 10 ld_next4: ldi 10h
0288 02B6 12 16 08 stm byte_counter
0289 02B9 ;Write newline
0290 02B9 17 03 ld_loop4: IN 03H
0291 02BB 0C 01 ANDIM 01H
0292 02BD 14 B9 02 JPZ ld_loop4
0293 02C0 10 0D LDI 00H   ;Return character
0294 02C2 18 02 OUT 02H   ;send to UART for output
0295 02C4 17 03 ld_loop5: IN 03H
0296 02C6 0C 01 ANDIM 01H
0297 02C8 14 C4 02 JPZ ld_loop5
0298 02CB 10 0A LDI 0AH   ;Linefeed character
0299 02CD 18 02 OUT 02H   ;send to UART for output
0300 02CF ;Code to validate hex character
0301 02CF 11 10 08 ld_next5: LDM temp   ;Retrieve character and validate
0302 02D2 0F 30 CMP 30H   ;Lower limit of hex characters
0303 02D4 16 DA 02 JPC ld_next2   ;Char >= 30H, possibly valid
0304 02D7 13 60 02 JMP ld_invalid   ;Char < 30H, invalid hex char
0305 02DA 0F 47 ld_next2: CMP 47H   ;ASCII for "G"
0306 02DC 16 60 02 JPC ld_invalid   ;Char is G or greater, invalid
0307 02DF 0F 41 CMP 41H   ;ASCII for "A"
0308 02E1 16 EC 02 JPC ld_validAF_lo   ;Char is valid A-F
0309 02E4 0F 3A CMP 3AH   ;ASCII for ":"
JPC  ld_invalid  ;Char is ":" or greater, but < "A", invalid
JMP  ld_valid09_lo  ;Char is valid 0-9
ANDIM 0FH  ;Mask off high bits
ADDIM 9  ;Now lo nybble correct
JMP  ld_indexed_stm  ;Store the byte in RAM
ANDIM 0FH  ;Mask off high bits
ADD  byte  ;Now full byte assembled
OUT 00H  ;Display on LEDs
JMP  ld_indexed_stm  ;Store the byte in RAM
LDM  ld_indexed_stm+1  ;Increment byte pointer, lo byte first
INC
STM  ld_indexed_stm+1
LDIM  ld_indexed_stm+2
JMP  ld_get_hi
JMP  sm_warm  ;Return to monitor
JMP  get_address
LDM  address
STM  bl_indexed_stm+1
LDM  address+1
STM  bl_indexed_stm+2
BL_addr_back  ldm  address
B_addr_back  ldm  address
BL_get_bytes  ldm  bytes_str
BL_get_bytes  ldm  bytes_str+1
BL_get_bytes  ldm  bytes_str+2
call0(write_string)
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 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
142
0375  0388 12 04 08          stm   dp_100s
0376  038B 11 81 08          ldm   buffer+1
0377  038E 12 03 08          stm   dp_1000s
0378  0391 11 80 08          ldm   buffer
0379  0394 12 02 08          stm   dp_10000s
0380  0397 13 E2 03          jmp   dp_10000_mult
0381  039A 11 83 08  dp_setup_4  ldm   buffer+3
0382  039D 12 06 08          stm   dp_1s
0383  03A0 11 82 08          ldm   buffer+2
0384  03A3 12 05 08          stm   dp_10s
0385  03A6 11 81 08          ldm   buffer+1
0386  03A9 12 04 08          stm   dp_100s
0387  03AC 11 80 08          ldm   buffer
0388  03AF 12 03 08          stm   dp_1000s
0389  03B2 13 04 04          jmp   dp_1000_mult
0390  03B5 11 82 08  dp_setup_3  ldm   buffer+2
0391  03B8 12 06 08          stm   dp_1s
0392  03BB 11 81 08          ldm   buffer+1
0393  03BE 12 05 08          stm   dp_10s
0394  03C1 11 80 08          ldm   buffer
0395  03C4 12 04 08          stm   dp_100s
0396  03C7 13 26 04          jmp   dp_100_mult
0397  03CA 11 81 08  dp_setup_2  ldm   buffer+1
0398  03CD 12 06 08          stm   dp_1s
0399  03D0 11 80 08          ldm   buffer
0400  03D3 12 05 08          stm   dp_10s
0401  03D6 13 48 04          jmp   dp_10_mult
0402  03D9 11 80 08  dp_setup_1  ldm   buffer
0403  03DC 12 06 08          stm   dp_1s
0404  03DF 13 6A 04          jmp   dp_1_mult
0405  03E2
0406  03E2
0407  03E2
0408  03E2            ;decimal parser multiplication
0409  03E2 11 02 08  dp_10000_mult  ldm   dp_10000s
0410  03E5 0A 30          subim  30h          ;ASCII for '0'
0411  03E7 14 04 04          jmp   dp_10000_done
0412  03EA 10 10          ldi   10h          ;hex low byte of 10,000 decimal
0413  03EC 00 00 08          addm   dp_value
0414  03EF 12 00 08          stm   dp_value
0415 03F2 10 27    ldi 27h ;hex high byte of 10,000 decimal
0416 03F4 01 01 08  adcm dp_value+1
0417 03F7 12 01 08  stm dp_value+1
0418 03FA 11 02 08  ldm dp_10000s
0419 03FD 1A       dec
0420 03FE 12 02 08  stm dp_10000s
0421 0401 13 E2 03 jmp dp_10000_mult
0422 0404           dp_10000_done  ldm dp_10000s
0423 0404 11 03 08  dp_10000_mult ldm dp_1000s
0424 0407 0A 30     subim 30h ;ASCII for '0'
0425 0409 14 26 04  jpz dp_10000_done
0426 040C 10 E8     ldi 0E8h ;hex low byte of 1000 decimal
0427 040E 00 00 08  addm dp_value
0428 0411 12 00 08  stm dp_value
0429 0414 10 03     ldi 03h ;hex high byte of 1000 decimal
0430 0416 01 01 08  adcm dp_value+1
0431 0419 12 01 08  stm dp_value+1
0432 041C 11 03 08  ldm dp_1000s
0433 041F 1A        dec
0434 0420 12 03 08  stm dp_1000s
0435 0423 13 04 04  jmp dp_10000_mult
0436 0426           dp_1000_done  ldm dp_1000s
0437 0426 11 04 08  dp_1000_mult ldm dp_10s
0438 0429 0A 30     subim 30h ;ASCII for '0'
0439 042B 14 48 04  jpz dp_1000_done
0440 042E 10 64     ldi 064h ;hex low byte of 100 decimal
0441 0430 00 00 08  addm dp_value
0442 0433 12 00 08  stm dp_value
0443 0436 10 00     ldi 00h ;hex high byte of 100 decimal
0444 0438 01 01 08  adcm dp_value+1
0445 043B 12 01 08  stm dp_value+1
0446 043E 11 04 08  ldm dp_100s
0447 0441 1A        dec
0448 0442 12 04 08  stm dp_100s
0449 0445 13 26 04  jmp dp_100_mult
0450 0448           dp_100_done  ldm dp_10s
0451 0448 11 05 08  dp_10_mult ldm dp_0Ah ;hex low byte of 10 decimal
addm \text{dp\_value}

stm \text{dp\_value}

ldi \text{00h}; \text{hex high byte of 10 decimal}

adcm \text{dp\_value}+1

stm \text{dp\_value}+1

ldi \text{00h}; \text{hex high byte of 100 decimal}

adcm \text{dp\_value}+1

stm \text{dp\_value}+1

; Set up byte counter and write ready string

ldm \text{dp\_value}

stm \text{bl\_byte\_counter}

ldm \text{dp\_value}+1

stm \text{bl\_byte\_counter}+1

ldm \text{bl\_ready\_str}

stm \text{ws\_inst}+1

ldm \text{bl\_ready\_str}+1

stm \text{ws\_inst}+2

call0(\text{write\_string})

; Loop to get binary data and store

\text{bl\_chk\_loop}:\text{in} 03h; \text{get status}

andim 02h; \text{check RxRDY}

jpz \text{bl\_chk\_loop}
in 02h; get binary from port
jmp bl_indexed_stm; store in RAM
bl_back ldm bl_indexed_stm+1; increment pointer
inc
stm bl_indexed_stm+1
ldm bl_indexed_stm+2
adcim 0
stm bl_indexed_stm+2
ldm bl_byte_counter; decrement byte counter
dec
stm bl_byte_counter
ldm bl_byte_counter+1
sbbim 0
stm bl_byte_counter+1
jpz bl_low_zero; check if byte counter = zero
jmp bl_chk_loop
ldm bl_byte_counter
jpz bl_done; yes, done -- return to monitor
jmp bl_chk_loop; no, get next byte
bl_done jmp sm_warm

goto address
Not called as a subroutine, return jump by switch structure
goto address
get_address ldm addr_str
stm ws_inst+1
ldm addr_str+1
stm ws_inst+2
call0(write_string)
call0(get_line)
;Write newline
**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
**0527** 051C             ;No error checking for length of string -- must be exactly 4 hex characters
**0527** 051C            ;Memory address stored in address variable
**0530** 052D
**0529** 052D             ;No error checking for length of string -- must be exactly 4 hex characters
**0529** 052D 11 2B 05
**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 11 80 08   ldm  buffer    ;first character
**0531** 052D 11 80 08
**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
**0535** 0539             ;Memory address stored in address variable
**0536** 0539 11 48 05
**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
ldm buffer+3 ;fourth character
stm char_pair+1
call1(hex_pair_to_byte)

ldm choice

get_line:
ldi 0
stm gl_str_len ;string length

ldi buff_low ;low byte of buffer address
stm gl_indexed_stm+1

ldi buff_high ;high byte of buffer address
stm gl_indexed_stm+2

gl_chk_loop: in 03h ;get status

andim 02h ;check RxRDY
Write_string subroutine, call as level 0

; Writes a zero-terminated string to screen at current cursor location
; Must set up address of string to be written in ws_inst+1 and ws_inst+2

write_string:

ws_chk_loop:
    in 03h ; get status

ws_back:
    jpz ws_done ; quit if end-of-string

out 02h ; output character to port

ldm ws_indexed_stm+1

jmp ws_store_it ; store character in buffer

gl_done:
    ret0

gl_indexed_stm:
    ldm ws_indexed_stm+2

inc

adcim 00h ; 16-bit addition

stm ws_indexed_stm+2

jmp gl_chk_loop

wp_indexed_stm:
    ldm ws_indexed_stm+1

inc

stm ws_indexed_stm+1

ws_out_loop:
    in 03h ; no, send char to screen

andim 01h ; check TxRDY

jmp ws_inst ; get a character when port ready

out 02h ; output character to port

ldm ws_indexed_stm+1

jmp ws_chk_loop

ws_done:
    ret0

ws_indexed_stm:
    ldm ws_indexed_stm+2

inc

stm ws_indexed_stm+2

jmp gl_chk_loop

gl_end_of_line:
    subim 0dh ; is it a return character?

jpz gl_end_of_line ; yes, replace with a zero

ldm gl_str_len ; no, increment string length

inc

stm gl_str_len

gl_store_it:
    jmp gl_indexed_stm ; store character in buffer

gl_back:
    ldm temp ; check if end-of-line (temp = 0)

jpz gl_done ; yes, quit

gl_out_loop:
    in 03h ; no, send char to screen

andim 01h ; check TxRDY

jmp ws_inst ; get a character when port ready

out 02h ; output character to port

ldm ws_indexed_stm+1

jmp ws_chk_loop

gl_done:
    ret0
inc
stm ws_inst+1
ldm ws_inst+2
adcm 00h ;16-bit addition
stm ws_inst+2
jmp ws_chk_loop

ws_done: ret0

;Subroutine hex_to_word -- call as level 0
;Calls hex_pair_to_byte as level 1
;Get 16-bit word value from input string in buffer
;No error checking for length of string -- must be exactly 4 hex characters
;16-bit value placed in h2w_value

hex_to_word:
ldm buffer ;first character
stm char_pair
ldm buffer+1 ;second character
stm char_pair+1
call1(hex_pair_to_byte) ;high-order byte

hex_to_word:
ldm buffer+2 ;third character
stm char_pair
ldm buffer+3 ;fourth character
stm char_pair+1
call1(hex_pair_to_byte) ;low-order byte
jpc h2w_done ;exit with carry set if error
stm h2w_value ;low byte of address
h2w_done ret0

;Subroutine to convert hex character pair to byte, call as level 1
;Character pair in memory location char_pair, stored hi-low
;Returns with byte in accumulator and carry flag clear if no error
;Returns with character in accumulator and carry flag set if error
;Calls char_to_nybble as level 2 subroutine

hex_pair_to_byte: ldm char_pair ;high order character of pair
stm temp ;char_to_nybble needs char in TEMP
call2(char_to_nybble)

STM byte ;Will eventually contain the byte to load
STM temp ;Value to add
LDI 10H ;Multiply x 16 to shift into high-order nybble
STM counter
LDM counter
MULTLOOP: LDM char_pair+1
DEC
JPZ GET_LO ;Have added 16 times, done
STM counter
LDM temp ;Original nybble
ADD byte ;Add to BYTE and store
STM byte
JMP MULTLOOP ;Keep adding

GET_LO: ldm char_pair+1
stm temp
call2(char_to_nybble)
jpc c2n_error ;Combine with hi nybble stored in BYTE
ccf ;in case addition changed it
JMP c2b_done ;Done, no error
c2n_error: scf
c2b_done: ret1

;Subroutine to convert byte to hex character pair, call as level 0
;Gets byte from byte variable
;Returns with character pair in memory location char_pair, stored hi-low

byte_to_hex_pair:

ldm byte ;dealing with high nybble

stm temp

ldi 00h ;prepare to shift down (divide by 16)

stm counter

b2h_divide: ldm temp

subim 16

jpc b2h_cont ;continue if nybble >=0

ldm counter ;hi-nybble now in low position

cmp 10 ;is value <10?

jpc b2h_hi_A2F ;yes, jump and convert to char

addim 30h ;no, convert to char

b2h_cont: stm temp

ldm counter

inc

stm counter

jmp b2h_divide

b2h_hi_A2F: addim 37h

b2h_store_hi: stm char_pair ;hi-order hex char

b2h_lo_A2F: jmp b2h_cont
The code includes a subroutine to convert hex characters to nybbles. Here is a detailed explanation:

```
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 08 08  b2h_store_lo:  stm  char_pair+1 ;now char pair is in variable
0705  06FC  ret0
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  jmp 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  jmp invalid  ;Char is G or greater, invalid
0722  070F 0F 41  cmp 41h ;ASCII for "A"
0723  0711 16 1C 07  jmp validAF  ;Char is valid A-F
0724  0714 0F 3A  cmp 3Ah ;ASCII for ":"
0725  0716 16 2A 07  jmp 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
0731  0724 0C 0F  valid09:  andim 0fh ;Mask off high bits
0732  0726 1C  ccf  ;exit no error
0733  0727  ret2
0734  0727 13 45 08  invalid:  ldm  temp  ;put char in accumulator
0735  072A 11 10 08  scf  ;Set carry flag
0736  072E  ret2
0736  072E 13 45 08
```
; String constants
new_line .dw $+2
.db 0dh,0ah,0
sm_greeting: .dw $+2
.db 0dh,0ah
.text "CPUville 8-bit processor system monitor v.1"
6c6520382d6269742070726f636573736f722073797374656d206d6f6e69746f7220762e31
.sm_greeting:
.db 0dh,0ah
.sm_prompt .dw $+2
.db 0dh,0ah
.text "Enter number: 1=restart 2=dump 3=run 4=load 5=bload"
.db 0
.addr_str .dw $+2
.db 0
.bytes_str .dw $+2
.db 0
.bl_ready_str .dw $+2
.db 0
.org 0800h ; Start of RAM
.text Address (hex):
732028686578293A20
.db 0
746F206C6F616420286465293A20
.db 0
52656164792C
 Ready, start transfer
..text Ready, start transfer
.db 0
.db 0

; The following section contains labels for RAM variables and other structures
; RAM Variables
.dp_value .dw 0000h
.db 0
.dp_10000s .dw 00h
.db 0h
.dp_1000s .dw 00h
.db 0h
.dp_100s .db 00h
.dp_10s .db 00h
.dp_1s .db 00h
.dp_10h .dw 0000h
.db 0
.db 0h
.db 0h
.db 0h
.db 0h
.db 0h
.db 0h
.db 0h
.db 0h
char_pair .dw 0000h
h2w_value .dw 0000h
address .dw 0000h
temp .db 00h
choice .db 00h
byte .db 00h
counter .db 00h
line_counter .db 00h
char_count .db 00h
byte_counter .db 00h
nybble_counter .db 00h

;RAM instructions with variable address (must initialize opcode when monitor is in ROM)
run_jump jmp 0000h
ld Indexed_load for load routine, must be in RAM
ld indexed_stm stm 0000h
jmp ld_stm_back

;Indexed load and store instructions for dump, must be in RAM
d Indexed_store instruction for binary loader, must be in RAM
bl Indexed load instruction for write_string, must be in RAM
ws inst: ldm 0000h
jmp ws back

;Indexed store instruction for get_line, must be in RAM
gl Indexed_stm: stm 0000h
jmp gl_back

;Return instruction for level 0 call macros, must be in RAM
return_jump0: jmp 0000h

;Return instruction for level 1 call macros, must be in RAM
return_jump1: jmp 0000h

;Return instruction for level 2 call macros, must be in RAM
return_jump2: jmp 0000h

return: .set 0000h ;assembler needs variable label set back to original value
tasm: Number of errors = 0