The successor to the Great Z80 Computer project . The new Z80 computer is much simplified in comparison to the original, but in many cases borrows ideas and circuits from it. When it is working I will be able to extend it as time permits.

Specifications:

- Processor: Z80B (4MHz 8-bit)
- Memory: 128 KBytes RAM + 32 KBytes Display RAM
- Graphics mode 1: 256 x 256 pixels, 16 colours (selectable from 256K colours)
- Graphics mode 2: 512 x 256 pizels, 4 colours (selectable from 256K colours)
- Text mode: 64 columns x 32 rows of ROM-generated characters
- Keyboard: 1980s Maplin experimenters keyboard
- Monitor: 12-inch Micro{gallery}newz80/1{/gallery}

Chipset

A word about the IC's used in this project. Most of the logic gates are from old circuits or salvaged

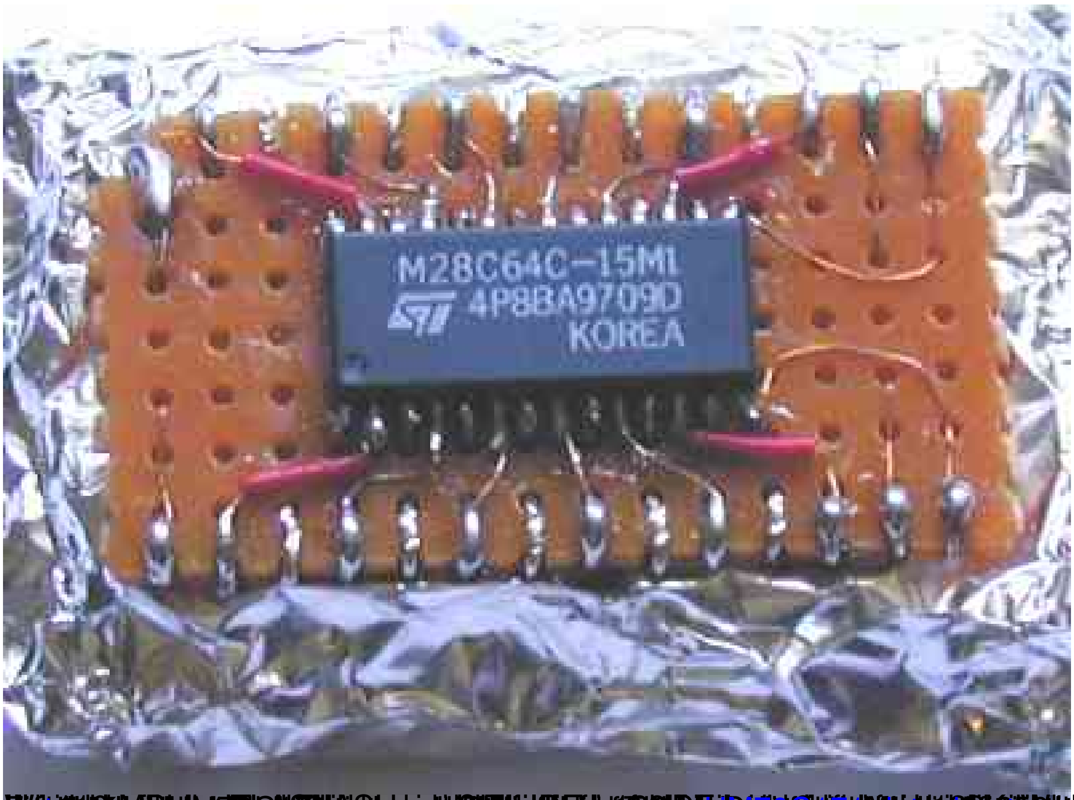
The display memory is a 32K static memory. You may recall that I had to use dynamic RAM for the display memory. At that tim

{gallery}newz80/2{/gallery}

As I want this project to be reasonably simple, I also use static RAM for the CPU, in this case a 128K device. For even greater simplicity I should have restricted it to 32K. the 128K chips were the same price as the 32K ones. So like an idiot I got the 128K chips. Later I remembered that since the Z80 can only address 64K of memory directly, I would have to mess around with memory banking. Either that, or only use part of the memory capacity but that would be untolerably wasteful.

For the character-generator ROM I used to use a 2764 device, which is a very comfortable to

To connect this chip, I cut a piece of plain (no copper strips, I never use those) 0.1-inch pitch matrix boa



[illegible]

Character Set

Because I love the Sinclair ZX Spectrum home computer, I decided to use the Spectrum character

I obtained the picture shown here by using a ZX Spectrum emulator, and writing a few lines of BASIC to {gallery}newz80/5{/gallery}

The ZX Spectrum character set uses the standard ASCII character codes, starting with the SPA {gallery}newz80/6{/gallery}

Switch Board

Of course when a computer is first made its memory is empty. There are no programs and no operating system. The computer just sits there and does nothing. It cannot write output to the monitor, nor can it accept input from the keyboard. All of these things require small programs to accomplish and somehow, these programs have to get into the computer's ROM. One way of doing this (the conventional way) is to build some kind of interface to the PC RS232 port and download programs from a development system running on your PC into the memory of the Z80 computer. Not the way I wanted to do things.

Accordingly I built the switch board pictured to the right. This has 13 switches connected to the add {gallery}newz80/7{/gallery}

8 more switches control every bit of the Z80 databus. The two remaining switches are the program switch which when ON allows the switches to program the memory, but when OFF starts the Z80 running at address 0000; and the STTS/WR switch for programming. Unlike the other switches which are just ordinary 2-way switches, the STTS/WR switch rests in its central position. Pushing it down to the WR position writes the byte specified by the data switches to the memory address specified by the address switches. Pushing it to the STTS position writes the data byte into the display status register on the board.

The switches are mounted on a rough piece of copper-clad PCB material. I never use this stuff for etching real PCB's, instead I occasionally use it to construct shielding or cases. Using these switches I painstakingly programmed the entire character set from the pictures shown above, bit by bit into the character ROM. Later I will program a keyboard driver. Then some kind of editor. That will definitely be the first task, to get away from having to program every bit of every byte by hand on the switches which makes the fingers sore.

More Photographs

Early Construction

Along with the 12-inch st{gallery}newz80/8{/gallery} the timing generator circuit is complete (page 1 of t

Half Done

Showing the rear connect{gallery}newz80/9{/gallery}struction and much of the video circuit is complete. E

Completed Board

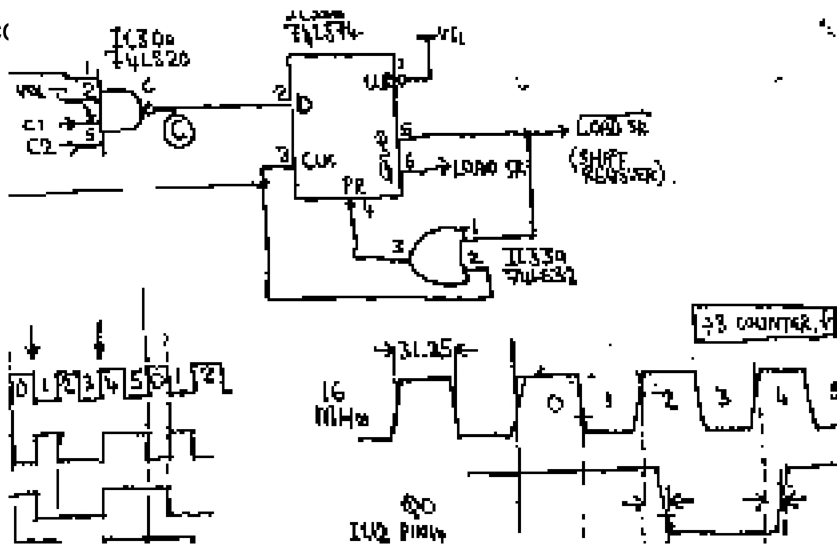
All socketed chips are no{gallery}newz80/10{/gallery} wires laid between the chips are tied at suitable inte

Underside of Completed Board

Showing the details of the{gallery}newz80/11{/gallery} builders I don't use copper-stipped matrix board. M

Circuit Diagrams

[CLICK HERE](#) to download a .pdf file co



Page 1 of 5: Clocks and Sync generator

The video timing generator circuit is almost identical to the one I designed for the Great Z80 Project. Therefore I have shamelessly copied much of the following description from the [original](#).

This part of the circuit, shown on page 1, is responsible for generating appropriate horizontal and vertical synchronisation pulses for the monitor, and synchronising the rest of the display circuit so that all the right pixels come out in the right places. Standard UK television resolution is 625 lines, with a refresh rate of 50 frames per second. In reality this is slightly confusing, since each frame actually only contains 312 lines. Any one frame only draws every other line, then the next frame comes and draws the other half of the lines, which is called interlacing.

Originally I used a Phillips 9-inch black & white monitor , but managed to procure a very nice microvitec colour monitor. It is capable of many different resolutions and refresh rates, but for simplicity I (and so I could use the same timing circuit as my original) I used the same timing as the Phillips monitor. i.e. a horizontal scan frequency of 15,625 Hz and vertical refresh rate of 50 Hz. Each line takes 64 uS (microseconds) to draw, of which a total of 16 uS is spent on the border to the left and right of the display area, and retracing to the start of the next line. Therefore the actual drawing area is 48 uS wide. I only draw 256 lines, the remaining 56 are the border at the top and bottom of the screen, and the vertical retrace. Into this 48 uS line, I squeeze 512 pixels, meaning each pixel lasts for 93.75 nS (nanoseconds).

At the top left is the 16MHz crystal oscillator circuit, from which all timing signals for the whole computer are derived. Dividing by four gives 4MHz which is the clock signal for the Z80. Dividing some more using dual 4-bit counter chip IC2 (74LS393) gives other frequencies for use in the timing of the monitor video signal. IC4 and the gates around it derive the Horizontal synchronisation pulse, which comes once per line in the 16 uS horizontal blanking period. IC7b and IC3 count 256 lines which represent the viewable display area. When the 257'th line arrives, it resets the counters (IC3) and IC5 starts counting. IC5 and IC6 count 56 rows, which are the blank lines at the top and bottom of the picture. The gates at the bottom left generate the vertical synchonisation pulse for the monitor, somewhere in the middle of the vertical blanking period.

There's a funny story about the row counter IC3. At first things didn't work as intended. However, since this was the second time I built this circuit, I had none of these difficulties. For a laugh, click here to read all about the debugging of the first project.

Page 2 of 5: Timing Generator

After the clock generation and synchronisation pulses generated on page 1 of the circuit, comes

some more timing logic.

Unfortunately the pixel rate requires a division of the primary 16 MHz clock by 3. Dividing by 3 is very easy but dividing by 3 and obtaining an equal mark/space ratio turned out to be a lot harder. Of course, my previous Z80 project also required 16 MHz / 3 for the pixel clock. I don't know if that worked properly or not. In this case when testing the video circuit with the monitor, it was readily apparent at first that alternate pixels on each horizontal line were brighter. The display was very ugly - when displaying text, instead of each whole character appearing a uniform brightness, vertical lines in the bright pixel column stood out clearly, etc. This is a direct result of alternate pixels being displayed for unequal times due to an unequal mark/space ratio in the divide-by-3.

To resolve this problem I created the elaborate divide-by-3 circuit shown, partly by trial and error. I used a 74LS163 presetable synchronous binary counter. Synchronous counters guarantee that all the binary outputs will change state at exactly the same time. The more usual asynchronous counters operate by chaining a number of D-type flip flops. The first is clocked, its output clocks the next and so on. In this way the count propagates down the chain such that the more-significant bits of the binary count change state a little later than the less significant. Because of this you often see the asynchonous type referred to as "ripple counters". I chose to use a synchonous counter as I was having enough trouble getting the timing exact anyway without having to worry about the inprecisions of a ripple counter.

The strategy is to load the counter with the number 13. 2 pulses later the count reaches 15. It cannot go any further than this because it's only a 4-bit counter. So it asserts a logic high on its TC output (TC = Terminal Count), which I use to drive the PE input (PE = Preset Enable). The following clock pulse loads the counter with 13 again. In this way the count will be 13, 14, 15, 13, 14, 15 etc. The time spent in each of the 3 states is exactly equal since the counter is synchronous. The OR-gate IC32a has 2 inputs. One is driven by the TC pin, causing a high pulse on the count of 15. The other is driven by NOR-gate IC9b. The output of this gate is a logic 1 only during the second half of state 13 (when the counter's Q1 output is 0 and the 16 MHz clock is 0). In this way the output of the OR-gate is 1 twice during each count of 3, i.e. during state 15 and the 2nd half of state 13. The leading edge of this pulse arrives exactly regularly. Applying this to the clock of D-type flip flop IC38b results in an output at pin 9 of IC38b which is has a frequency of exactly 16 MHz divided by 3 and an equal mark/space ratio. This is signal C0, i.e. pixel column 0. All of this is also illustrated in the timing diagram at the bottom right of the circuit diagram.

A small word about the CLR input of the flip flop IC38b. Why did I connect it to the Q0 output of

the 13, 14, 15 counter? This forces the phase of the CO signal to be in the first of two stable states. The situation I want is for the C0 signal to go low on the second half of the 16 MHz clock in state 13. This connection forces this to occur. Without it, I found that more-or-less at random the circuit could fall into the other stable state, i.e. C0 shifted 3 halves of the 16 MHz clock. Why do I care which configuration is used? Because I also have to create signals later to load the shift register and enable the video output at the start and end of each horizontal line. Without forcing the phase of C0 I got some peculiar effects - half the time the first pixel would dissappear off the left of the display.

A divide by 3 signal is also required to clock the pixel shift register at the character ROM output, for displaying the characters in the text mode. I thought I would be able to use the input to the clock of IC38b, since the leading edges are regular at the divide by 3 clock frequency. As it happened this was not the case. Using that signal directly resulted in the shift register being clocked at half the required rate, such that only the first half of each character appeared, stretched out over the entire character width. I don't know why this didn't work but rather than worry too much about it I inserted the quad XOR-gate IC40, arranged in the classic edge-detector configuration. This generates a pulse every time C0 changes state. The frequency of this signal is 16 MHz divided by 1.5.

Also shown on this page are the column address counters (IC13a and b), and some confusing circuit around the D-type flip flops IC41a and b. These generate the video enable output, which is '1' while the video beam is in the displayed part of the screen, and '0' at the edges and during horizontal and vertical flyback. This signal is generated from the LINE and FRAME signals from page 1, but it's not quite so simple - additional gating is required to delay the VIDEO EN signal by an appropriate amount such that the right edge of the display is not cut off. Originally I found that the leftmost character displayed what I thought should appear at the far right of the screen. Eventually I realised that the RAM lookup followed by character ROM lookup and latch actually delays the output of a character by one full character width, so I have to delay the VIDEO EN signal by a similar amount. The circuit also has to work properly when displaying graphics. Once I had it working with text, it then didn't work with the graphics mode. Only after much trial and error did I come up with the current circuit, which perfectly synchronises the VIDEO EN signal, divide-by-3 counter, column counter and ROM-shift-register-load.

Page 3 of 5: Video RAM, Character ROM, Timing Generator, multiplexing

So, the heart of the video generation circuit.

Here we see the 62256 video RAM (32K), and 28C64 8K EEPROM character ROM. Also note a multitude of 74LS157 quad 2-1 multiplexer chips, which I use to direct data and address busses according to the required mode. What mode? The video circuit has 4 modes of operation:

- o Boot: Only the top 1K of the ROM is used for the character patterns. At switch on, the lower 7K is used to boot the computer
- o CPU Read/Write: In this mode the CPU can read and write the RAM which holds the displayed character codes or graphics bitmap
- o Text Mode: RAM holds 16 pages of 64 x 32 characters, character patterns generated by the ROM
- o Graphics Mode: RAM holds 32K bitmap, 512 x 256 x 4 or 256 x 256 x 16 (hor, vert pixels, colours)

To illustrate the data flow in each of the four modes I drew some block diagrams, then I coloured them blue to show addresses and red to indicate the data path. The active IC's in each case are coloured yellow. The 74LS245 octal bus tranceiver operates as a bidirectional switch. Trapesoidal symbols represent the 2-1 multiplexers which are made up of several 74LS157 IC's (each 74LS157 contains 4 2-1 switches). By switching the multiplexers and opening the 74LS245's in the correct combination the circuit has the required configuration for each mode. More explanation follows!

Mode 1: Boot

The computer enters this {gallery}newz80/12{/gallery} mode after a system reset (e.g. when first switched on). So that I didn't ha

Mode 2: CPU Read/Write

To write anything at all to {gallery}newz80/13{/gallery} the screen the CPU needs to take control of the display RAM from the vi

Mode 3: Text Mode

Text mode is selected by {gallery}newz80/14{/gallery} setting bit 6 of register OUT0 to '0'. Only 2K of the display RAM is enoug

When the 7-bit ASCII character code is read from the display RAM, it is routed to the address inputs of the character ROM. The 3 least significant bits of the ROM address are formed from row 0, 1 and 2. Therefore the correct pixel row of the 8 rows which constitute the pattern for each character are displayed in the correct place. During text display, the upper 3 bits of the ROM address are forced to '1', selecting the top 1K of the 8K ROM, which is where the character patterns are held.

When one 8-pixel row defining the pattern of one eighth of a single character is ready at the data output of the character ROM, it is loaded into the 74LS165 parallel-load shift register (IC13). In the following 8 pixel clocks the state of each pixel is shifted out of this shift register. IC29 the quad 2-1 multiplexer (74LS157) selects between 4 text bits or 4 graphics bits. But the shift register only outputs 1 bit for the character pattern (Bit 0, e.g. corresponding to black or white). What about the remaining 3 bits? These 4 bits are fed to the colour pallette chip which gives me some flexibility of how I may want to use them. I decided to set them as follows.

Bit 1 comes straight from bit 7 of the display RAM. Recall that the ASCII code specifies only 128 characters, taking only 7 bits. I therefore have a spare bit which I can use with each character as an display attribute of that character. Depending on the pallette settings this could specify that the character is a different colour for example. However for even more flexibility I connected Bit 2 to the "FLASH" signal, which is the 50 Hz vertical synchronisation pulse divided by 16, i.e. something a little over 3 Hz (flashes per second). Then I connected Bit 3 to the output of a 4-input NAND gate such that it will be '0' during the last row of the 8-row character pattern, '1' otherwise. Depending on the colour settings in the pallette I can then use bit 7 to cause a character to flash, be underlined, or a different colour (including being displayed in "inverse" colours, e.g. black on a white background instead of white on black). Neat.

Mode 4: Graphics Mode

Graphics mode is selected by setting bit 0 of register OUT0 to '1'. Here the character ROM is not u

{gallery}newz80/15{/gallery}

The pallette chip can take 8 input bits (256 colours). To be a bit clever I decided to create a high resolution mode by simply connecting bit 5 of the pallette input to the column 0 (C0) counter output. This will be '0' in the first half of one of my ordinary 256 x 256 pixels and '1' in the second half. Merely by carefully programming the colours in the colour pallete I can then choose either a 256 x 256 x 16-colour graphics mode or a 512 x 256 x 4-colour mode! The remaining 3 bits of the colour pallette input I just drive from bits 0-3 of the OUT3 register. Therefore I can program 8 seperate pallettes and switch between them easily.

Note that I don't make any special arrangements for to ensure that the pixel mapping in the display RAM is convenient i.e. adjacent pixels occupy consecutive memory addresses. Due to the R0, R1 and R2 signals going direct to the character ROM during text mode, the addressing requirements would be somewhat different for graphics mode. To line up the pixels consecutively I'd have had to use yet another set of 74LS157 2-1 multiplexers to route the row and column count correctly. I decided against this, given that it will be simple to write an alogorithm in software to convert a normal X, Y pixel address to the corresponding display RAM address.

Page 4 of 5: CPU, Main memory, output ports

Here I finally show the actual Z80 CPU. It always surprises me that the circuit surrounding the CPU is actually the easiest and simplest part of the whole circuit. The real complication is in the video driver circuit.

IC42 the Z80B itself needs no explanation. The 431000 static RAM used as the main memory is in fact a 128K device. This creates a problem because the Z80 address bus is only 16 bits wide, implying a maximum address space of only 64K. I therefore divide the 128K memory into 4 pages, which I select using the two bits 4 and 5 of the OUT0 register (labelled STTS A15 and STTS A16).

The pallette register is programmed on output port OUT1. This page of the circuit diagram shows the OUT0-7 output decoder IC34, a 3-8 decoder. This decodes bits 4, 5 and 6 of the CPU's output port address space. The keyboard interface uses the same circuit I used in my old Z80 project. 4 address lines A0 to A3 are sent to the keyboard and decoded into 16 lines by a 74LS154 (4-16 decoder) inside the keyboard case. 8 data lines return from the keyboard matrix and are read back from the keyboard using the OUT2 signal. 4 8-bit 74LS256 bus tranceivers buffer signals from the computer to the outside world, via the expansion connector (IC's 35, 36, 37 and 50). The reset circuit also comes directly from the original Z80 project. It generates a reset pulse on switch on and when the CPU BUSRQ signal goes high. I use this to program the main memory via the switchboard, until such time as the computer can be programmed in a more convenient way by the keyboard!

The following is the IO map of this Z80 Computer. Addresses not in this list aren't currently used...

0-15: Status register (video), write only. Any IO in the range 0-15 writes the register.
Bit 0: SCR0)
Bit 1: SCR1 >-- Selection of display RAM page in text mode
Bit 2: SCR2)
Bit 3: SCR3)
Bit 4: A16 (main memory page 1)
Bit 5: A15 (main memory page 0)
Bit 6: '0' = TEXT mode, '1' = Graphics mode
Bit 7: '0' = CPU read/write mode, '1' = video circuit TEXT and GRAPHICS modes

16-19: Pallette registers 0-3, read/write

32-47: Keyboard scan lines 0-15, read only

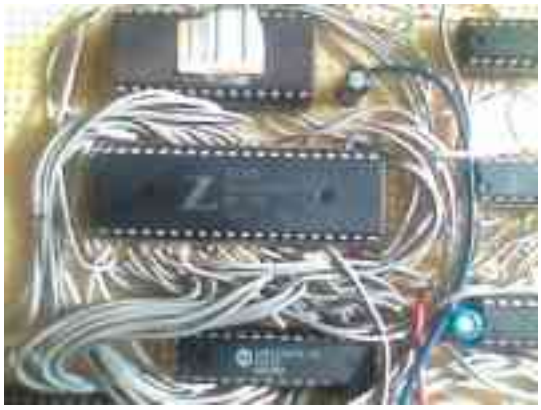
48-63: Control register, write only
Bit 0: PALLETTE 5)
Bit 1: PALLETTE 6 >-- Bits 5-7 of the pallette input. Specify 1 of 8 32-colour pallette pages
Bit 2: PALLETTE 7)
Bit 3: CAPS LOCK: Signal to drive the keyboard Caps Lock key's built in LED
Bit 4: SOUND: Signal fed to expansion connector, could drive a piezo sounder
Bit 5: OUT 0)

Bit 6: OUT 1 >-- Output signals fed to the expansion connector
Bit 7: OUT 2)

Page 5 of 5: Chip placement

The last page shows the chip placement on the board, and the pinout of the expansion connector.

Homebrew Z80 computer by Marton Kun-Szabo



Don't miss this fantastic Homebrew Z80 computer built by Marton Kun-Szabo (Hungary).