« CPCNG Notes »

The CPCNG is a new version of Amstrad CPC computer, build around a Zilog eZ80 and with full compatibility CPC mode.

Last update : 2006.

Table of contents

Prototype Board Contents

The prototype board will contain the following components:

1. eZ80 CPU from Zilog

2. 128K EEPROM

3. 128K Fast SRAM for video and CPC-mode

4. FPGA containing IDE hard drive interface, video, memory mapping and CPC compatibility unit.

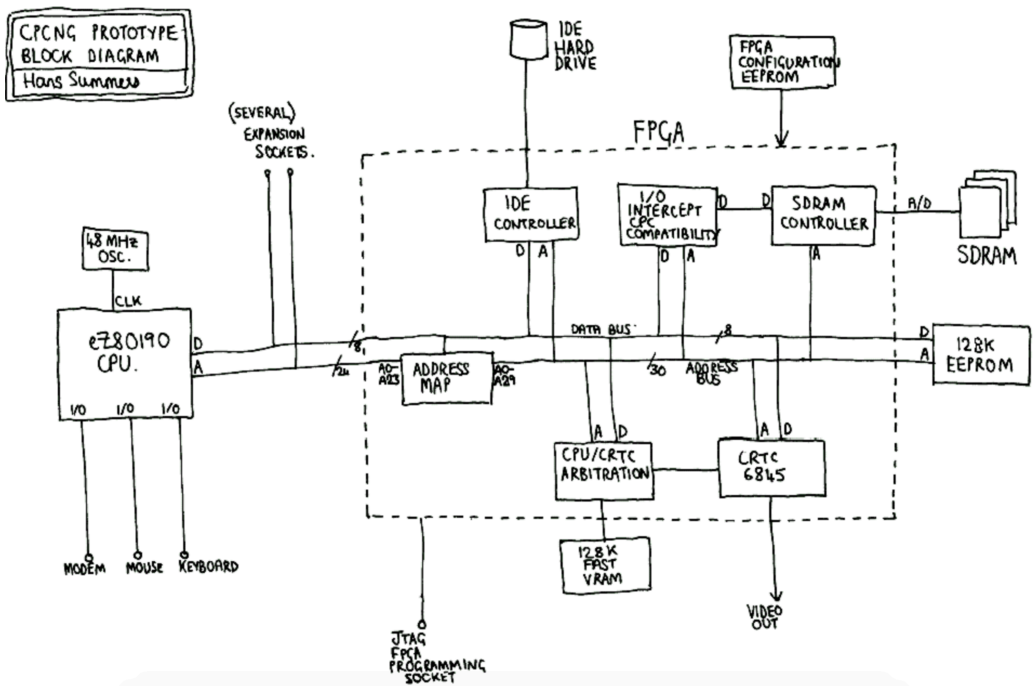
5. Sockets for large PC-style SDRAM cards

6. Sockets for modem, keyboard, mouse and power supply.

7. Expansion sockets for new cards to be designed later

Block Diagram

A simplified block diagram of the prototype is shown below. This shows the interconnection of the various units discussed below and it will be apparent that most of the units will be coded in the FPGA. This diagram does not show the control signals (WR/RD/IOREQ etc.) which would overcomplicate the diagram at this level.



- **CPU**

We will use an eZ80 CPU. A supplier has been located who are willing to provide them in small quantities at a price of £9.93 each (approx USD 14, EUR 16, FRF 105, DEM 31). The eZ80 actually comes as the eZ80190 Webserver, which is an eZ80 core plus various on-chip timers and interfaces. This chip is now available (Oct-01) but since it is new there is a lead time of some 2 months according to WBC.

- **Memory**

We will use PC-type SDRAM cards for the main memory. A 128K fast SRAM will also be used as video RAM and in the CPC compatibility mode. A 128K ROM (EPROM or EEPROM) will contain boot code and any other required code e.g. the old CPC ROM's for the compatibility mode. The eZ80 address space is 16M, and the large SDRAM will be mapped in up to 256 4M blocks giving a total capacity of up to 1G. The 128K video RAM and 128K ROM will be mapped at programmable locations.

- **CPC Compatibility**

The CPCNG will have a CPC mode where it will be able to run CPC software without modification. Software which relies on precise timing (e.g. demos) may not work as expected since the eZ80 is much faster than the old Z80, and certain software emulation of CPC hardware will be required.

- **Peripherals**

The CPCNG will initially use PC-compatible peripherals. These are readily available and cheap, or even free where they can be salvaged from surplus or salvaged obsolete PC's. The peripherals used will be:

1. PC power supply

2. PC keyboard (AT)

3. PC IDE hard disk

4. PC mouse

5. VGA compatible monitor

The keyboard interface will be initially (in the prototype) just a connection to I/O pins on the eZ80 which will manage the keyboard serial interface in software. Similarly the mouse socket will just be connected to eZ80 I/O pins. A modem socket will also be connected to eZ80 I/O pins, as certain eZ80 I/O ports are designed to be able to connect to a modem. Later keyboard and mouse interfaces can be designed in the FPGA, but in the prototype to keep the hardware as simple as possible, these interfaces will be managed in software by the eZ80 and its onboard I/O ports.

- **PCB**

The prototype PCB will be produced in a small quantity, perhaps by a supplier such as Express PCB. It will contain the eZ80, 128K ROM, 128K SRAM, FPGA and sockets for: modem, keyboard, IDE hard disk, mouse, VGA monitor, power supply, SDRAM (PC cards) and several expansion sockets for further developments.

- **FPGA**

A large Field Programmable Gate Array (FPGA) will be used to implement all of the logic in the CPCNG, including the video circuit, IDE hard drive interface, memory mapping and CPC compatibility unit. The availability of FPGA's is a major advantage to us as it allows the use of complex logic circuits while minimising the chipcount, thus making the computer cheap and easily constructed by the group members.

There has been no final choice of FPGA but Xilinx seems to be the popular choice. My suggestion is to use schematic entry where possible because this will be most easily understood by the majority of people. Various cheap Xilinx development boards can be found (e.g. XESS) but my suggestion is that we put the FPGA directly on the board, with a JTAG programming socket and do development directly on the CPCNG prototype board using PC FPGA development tools.

The following summarises the circuit blocks which will exist in the FPGA.

1. CRTC Video driver: For CPC compatibility we intend to use the 6845. This will be implemented in the FPGA, from a free VHDL model for the 6845 which is available from Opencores. The 6845 will use the 128K SRAM, and arbitration of this memory between the eZ80 and the 6845 will also be managed by logic in the FPGA.

2. IDE interface: A simple interface with latches for the upper 8-bits of the 16-bit IDE bus.

3. Memory mapping unit: maps SDRAM in 4M blocks. Also does mapping in the CPC mode in 16K blocks, as on an old CPC.

4. SDRAM controller: possibly using VHDL from Opencores.

5. CPC I/O intercept trick: To obtain maximum CPC compatibility, all I/O requests in CPC mode are intercepted by the FPGA and create a non-maskable interrupt. A routine at &0066 (NMI call address) translates the CPC I/O attempted, does the appropriate CPCNG thing, and puts the eZ80 registers in the state the CPC would have expected.

6. Video RAM arbitration unit: Switches the 128K fast SRAM between the eZ80 and the 6845 CRTC.

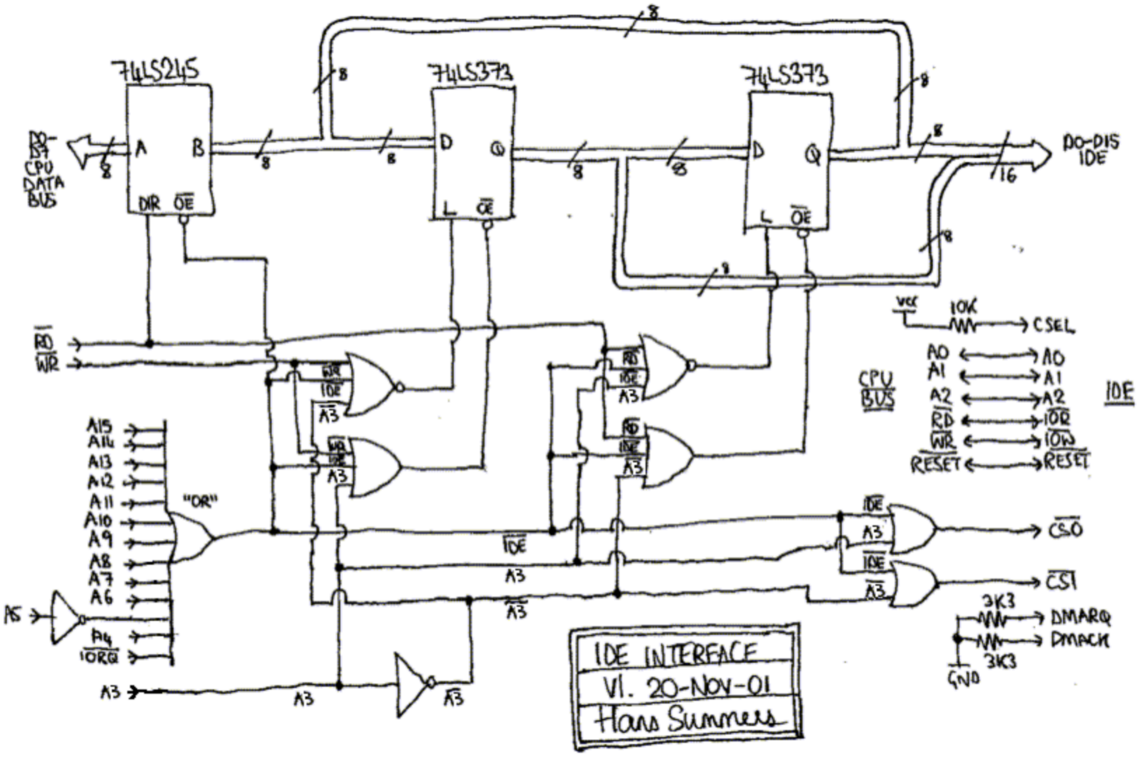
- **Future Versions**

Future versions of the CPCNG will include an advanced graphics processor and sound capabilities. The prototype described here deliberately omits these in an effort to minimise complexity. We need a prototype which is simple enough to be achievable, yet capable of expansion and operating as a test bed for the future verions. It is anticipated that once the CPCNG prototype board is working, a number will be built by members of the group, and hardware development will then expand. Different groups will be able to work on different sections of the hardware for the next version.

IDE Interface

This page describes the proposed IDE hard disk interface. The IDE specification is simpler than SCSI so has been chosen for the prototype. The IDE interface would be almost trivial but for the fact that the IDE databus is 16 bits wide, whereas the eZ80 data bus is only 8 bits wide. It is possible to connect only 8 bits of the IDE bus directly to the eZ80 data bus, but doing this sacrifices half the capacity of the disk drive.

The solution here is to buffer the high byte of the data word in an 8-bit latch, which is read/written to on an additional port number. There are a number of examples of this on the internet, with circuit diagrams and sample code (see list of links below). The information and circuit described here are derived from these project pages. Below is the draft circuit diagram.



The IDE registers are mapped to eZ80 I/O addresses in the range &0020 to &002F. The high byte is written or read on port &0028. When writing a byte, the high byte register must be written prior to writing the low byte (&0020) so that it is available for sending to the disk drive, which occurs on writing to &0020. Similarly, reading from port &0020 reads the low byte and latches the high byte from the disk drive, which can then be read from port &0028.

- **CPC NG I/O Addresses for IDE Registers**

The following table describes the port mapping of the IDE interface:

Port	Register	Function
&0020	IDELO	Data Port (low byte)
&0021	IDEERR	Read: Error Register; Write: Precomp
&0022	IDESECTC	Sector Count
&0023	IDESECTN	Sector Number
&0024	IDECYLLO	Cylinder Low
&0025	IDECYLHI	Cylinder High
&0026	IDEHEAD	Drive/Head
&0027	IDESTTS	Read: Status; Write: Command
&0028	IDEHI	Data Port (High Byte)
&0029		Not Used
&002A		Not Used
&002B		Not Used
&002C		Not Used
&002D		Not Used
&002E	IDECTRL	Read: Alternative Status; Write; Device Control
&002F	IDEADDR	Drive Address (Read Only)

- **IDE Register descriptions**

This section describes the IDE register useage.

&0021 (write): Write Precomp: A write to this port sets the "Write Precompensation Cylinder divided by 4". Not sure what this means.

&0021 (read): Error Register: The contents of the error register when in diagnostic mode are shown in this table:

Value	Error
1	No Error detected
2	Formatter device error
3	Sector buffer error
4	ECC circuitry error
5	Controlling microprocessor error

The contents of the error register (read &0021) when in operation mode are shown in this table:

Bit	Value	Meaning
0	0	DAM found
	1	DAM not found
1	0	Track 000 found
	1	Track 000 not found
2	0	Command completed
	1	Command aborted
3	0	Reserved
4	0	ID found
	1	ID not found
5	0	Reserved
6	0	No error
	1	Uncorrectable ECC error
7	0	Block OK
	1	Bad Block detected

&0026: Drive/Head: Bit 4 of the drive/head register selects drive 0 or drive 1, and bits 3-0 select the head.

&0027 (read): Status: Bits in the status register are set to 1 to indicate one of the following conditions:

Bit	Condition
0	Previous command ended in an error
1	Index: set to 1 each disk revolution
2	Disk data read successfully corrected
3	Sector buffer requires servicing
4	Seek complete
5	Write fault
6	Drive is ready
7	Controller is executing a command

&0027 (write): Command: Writes to the command register are as follows:

Command	Function

98 E5	check power mode (IDE)
90	execute drive diagnostics
50	format track
EC	identify drive (IDE)
97 E3	idle (IDE)
95 E1	idle immediate (IDE)
91	initialise drive parameters
1x	recalibrate
E4	read buffer (IDE)
C8	read DMA with retry (IDE)
C9	read DMA without retry (IDE)
C4	read multiples (IDE)
20	read sectors with retry
21	read sectors without retry
22	read long with retry
23	read long without retry
40	read verify sectors with retry
41	read verify sectors without retry
7x	seek
EF	set features (IDE)
C6	set multiple mode (IDE)
99 E6	set sleep mode (IDE)
96 E2	standby (IDE)
94 E0	standby immediate (IDE)
E8	write buffer (IDE)
CA	write DMA with retry (IDE)
CB	write DMA with retry (IDE)
C5	write multiple (IDE)
E9	write same (IDE)
30	write sectors with retry
31	write sectors without retry
32	write long with retry
33	write long without retry
3C	write verify (IDE)
9A	vendor unique (IDE)
C0-C3	vendor unique (IDE)
8x	vendor unique (IDE)
F0-F4	EATA standard (IDE)
F5-FF	vendor unique (IDE)

Memory Mapping

The eZ80 has 24 address lines, giving a maximum 16M address space. The CPCNG will use 4M blocks in the CPCNG mode. Each 4M region of the 16M address space is mapable to any of 256 4M blocks in the large SDRAM. This gives a maxmimum possible memory size of 1G, though CPCNG users will themselves decide how much memory they want to buy, so the OS must be capable of detecting memory size and operating with whatever is available. Addressing 1 GByte requires 30 address lines. The CPCNG is even able to function when NO main memory is present on the board, in this case it still runs in the CPC mode, with 128 RAM provided by the video RAM.

The mapping takes place as follows:

In CPCNG mode, there are four 8-bit registers CPCNGA0 - 3, one for each of the 4M blocks in the 16M eZ80 address space. These registers reside in the FPGA and are written using I/O ports &0018-&001B. The lower 22 address lines of the eZ80 address bus, a0 - a21 connect directly to the memory. During memory access the upper 2 lines a22 and a23 select one of the 4 8-bit banking registers. The output 8-bits of the selected register drive address lines a22 - a29 of the large memory.

CPCNG mode summary:

a0 - a21	from a0 - a21 of CPU
a22 - a29	from 1 of 4 banking registers CPCNGA0 - 3, selected by a22 and a23 of CPU

In the CPC mode, the eZ80 is in native Z80 mode and resets a16 - a23 of its address bus to '0'. In this mode, a0 - a13 still drives the memory directly. a22 and a23 are '0' and therefore in CPC mode only the lowest 4M block of the 16M address space is used. All 16K memory blocks that are used for the CPC mode must therefore exist in the same contiguous 4M block of memory, but that can be any 4M block chosen from the entire memory. One 4M block contains 256 16K CPC blocks.

In CPC mode, four 8-bit FPGA registers CPCRDA0 - 3 on I/O ports &0010-&0013 select four 16K blocks for reading from memory. a14 and a15 of the eZ80 address bus select one of these 4 registers which drives address lines a14 - a21 of the main memory. Therefore each of the 4 16K sections of the native Z80 mode 64K address space can be mapped for reading from any of 256 16K blocks of the selected 4M block from main memory.

Similarly, another four 8-bit registers CPCWRA0 - 3 on I/O ports &0014-&0017 select 4 16K blocks for writing to memory.

The reason for having separate registers for the reading and writing memory maps is that in a real CPC, if the ROM is mapped to a particular 16K block, reading takes place from ROM but writing to addresses in the block writes to the underlying RAM. In the CPCNG in CPC mode, I allow different parts of memory to be used for reading and writing for each of the 16K blocks. If a 16K block was to be used as ordinary RAM, the read and write registers would contain the same value. If a 16K block of the CPC was to be mapped as ROM, any 16K block of the main

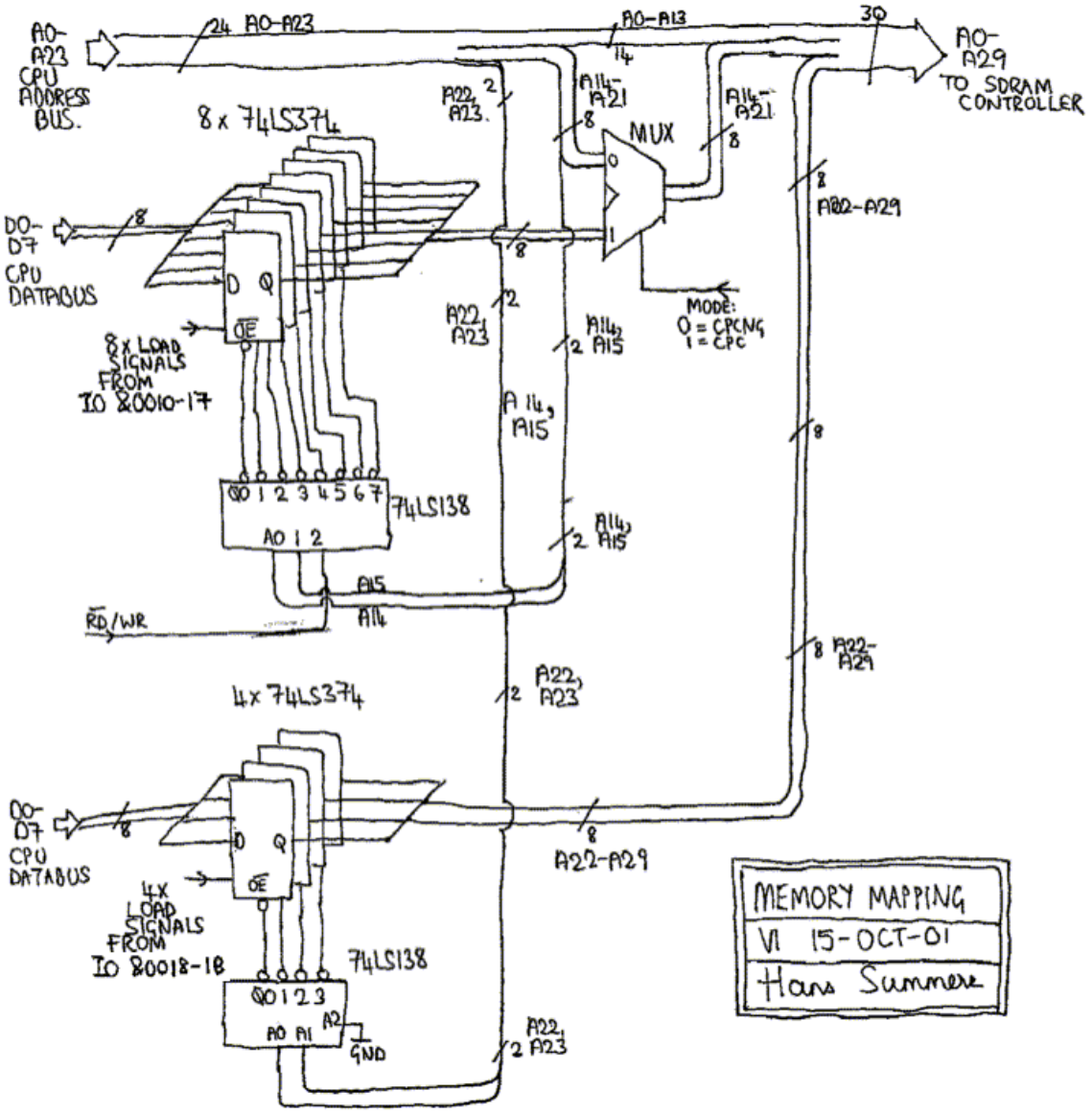
memory can be used for that ROM, and a different 16K block for the RAM at that location.

All of the CPC memory mappings are available using this method (100% compatibility), and it is possible to accomodate without too much difficulty in hardware, peacefully coexisting with the larger available memory and CPCNG mode mapping. Crucially, when an I/O occurs in CPC software, this will be handled by the I/O intercept trick: a /NMI (non maskable interrupt) is generated, calling routine at &0066. This will decode the CPC I/O request and in the case of memory mapping I/O, translate the old-style codes as documented at http://andercheran.aiind.upv.es/~amstrad/, to the necessary values for the 8 CPC-mode memory map registers CPCRDA0 - 3 and CPCWRA0 - 3.

CPC mode summary:

a0 - a13	from a0 - a13 of CPU
a13 - a21	READ: from 1 of 4 banking registers CPCRDA0-3, WRITE: from 1 of 4 banking registers CPCWRA0-3, selected by a22 and a23 of CPU
a22 - a29	from NG-mode banking register CPCNGA0

Below is my draft circuit diagram for the mapping unit using equivalent TTL 74LS-series part numbers. The multiplexer switches A14-A21 of the memory address bus from A14-A21 of the CPU address bus in CPCNG mode to the outputs of the CPC-mode mapping registers.



What of the video RAM? The 128K of video RAM can appear at any desired location in the 16MByte address space of the eZ80, on a 128K boundary. An 8-bit register VRAMA on port &001C selects this location. Bits d0 - d6 of VRAMA specify the 128K block (address lines a23 - a29) where the 128K VRAM will appear. Bit d7 will be '1' to disable the VRAM (but the CRTC still accesses it), bit d7 will be '0' to map the VRAM to the 128K block specified by the 7 bits d0 - d6. A comparator in the FPGA compares a23 - a29 with bits d0 - d6 of VRAMA, and selects the VRAM chip instead of main memory if there is an exact match. Effectively the 128K VRAM replaces main memory at the 128K block where it is mapped. When the CPU reads or writes to any address within the 128K block allocated VRAM, the request is routed to the VRAM rather than main memory. In CPC mode, the 128K VRAM should simply be mapped to a convenient 128K block in the first 4 MBytes of the eZ80 address space. The contents of CPCRDA0 - 3 and CPCWRA0 - 3, and the programming of the CRTC itself, take care of exactly replicating the CPC mode address map.

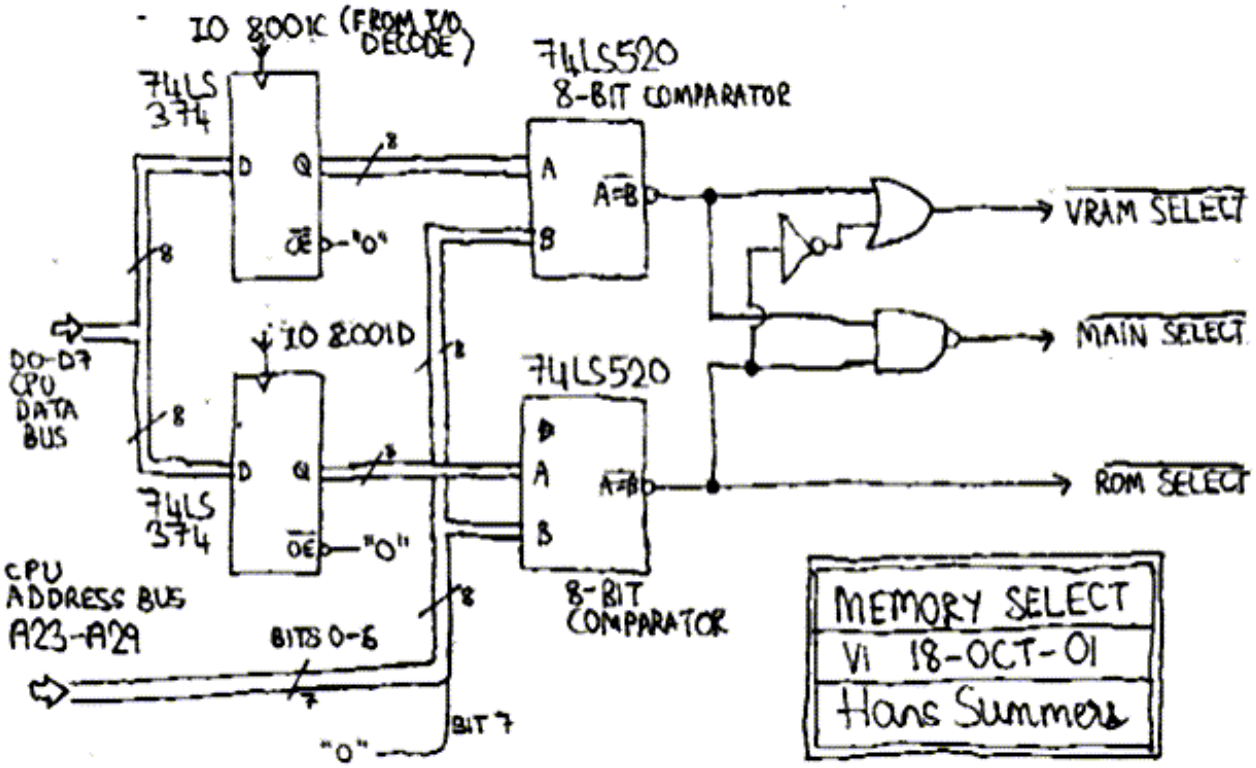
Arbitration of the video RAM between the CPU and CRTC is a separate matter that will be discussed elsewhere (/BUSREQ etc).

The CPCNG must also have a boot ROM. It will have a 128K EEPROM, this is more than enough to contain the boot code, and also if desired the 16K of CPC OS ROM, and 16K of CPC BASIC ROM. Similar to the way the VRAM is mapped, the 128K ROM will be mapped by bits d0 - d6 of another 8-bit I/O register called ROMA, appearing on I/O port &001D. Bit 7 = '1' disables the ROM. Any read in that 128K block will be routed from the ROM rather than main SDRAM memory. Likewise in CPC mode, the ROM can be mapped somewhere convenient in the lower 4M of the eZ80 address space, the contents of CPCRDA0 - 3 and CPCWRA0-3 taking care of exactly replicating the CPC memory map. At switch on, the ROM appears at &000000 so that boot can occur from it.

The CPC ROMs do not have to be in the CPCNG ROM. Instead it would be equally possible to have them loaded into RAM from the hard drive. The mapping registers allow the CPC ROM to be located in main memory if required.

This ROM must contain meaningful code at &0066 to handle the I/O intercept trick (/NMI). When in CPC mode an I/O attempt occurs, the hardware immediately switches the ROM bank to appear at &000000 so that the I/O intercept routing can operate. On returning from the /NMI call, the hardware returns the ROM bank to its programmed location as specified by ROMA.

Below is my draft circuit diagram for the memory select generator using equivalent TTL 74LS-series part numbers. The two registers hold the location to map the ROM and Video RAM. A match occurs when bits 0-6 of the stored value match A23-A29 of the CPU address bus, bit 7 must also be zero (compared against a hard-wired '0'). The output gating ensures that only one select is activated at a time. In case of conflict (both VRAM and ROM mapped to the same location) the preference is ROM then VRAM then main memory.



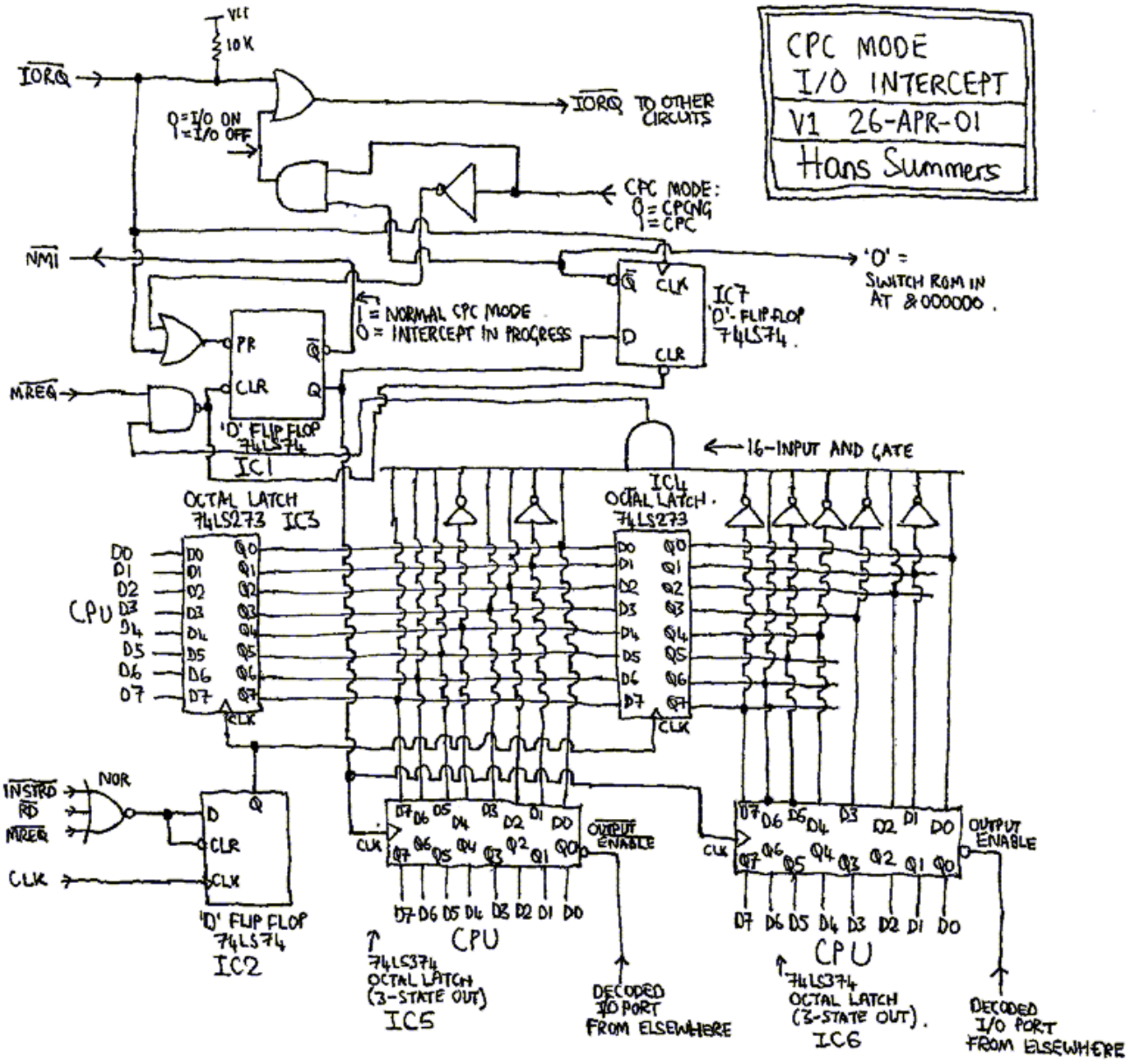
Summary of I/O registers for memory mapping:

Port	Register	Mode	Function
&0010	CPCRDA0	CPC	1 of 256 16K blocks for READ at &0000-&3FFF
&0011	CPCRDA1	CPC	1 of 256 16K blocks for READ at &4000-&7FFF
&0012	CPCRDA2	CPC	1 of 256 16K blocks for READ at &8000-&BFFF
&0013	CPCRDA3	CPC	1 of 256 16K blocks for READ at &C000-&FFFF
&0014	CPCWRA0	CPC	1 of 256 16K blocks for WRITE at &0000-&3FFF
&0015	CPCWRA1	CPC	1 of 256 16K blocks for WRITE at &4000-&7FFF
&0016	CPCWRA2	CPC	1 of 256 16K blocks for WRITE at &8000-&BFFF
&0017	CPCWRA3	CPC	1 of 256 16K blocks for WRITE at &C000-&FFFF
&0018	CPCNGA0	Both	1 of 256 4M blocks for &000000-&3FFFFF
&0019	CPCNGA1	CPCNG	1 of 256 4M blocks for &400000-&7FFFFF
&001A	CPCNGA2	CPCNG	1 of 256 4M blocks for &800000-&BFFFFF
&001B	CPCNGA3	CPCNG	1 of 256 4M blocks for &C00000-&FFFFFF
&001C	VRAMA	Both	Location of 128K Video RAM mapping
&001D	ROMA	Both	Location of 128K ROM mapping

CPC Compatibility Unit

This is a draft design for the CPC-compatibility circuit. This logic will reside in the FPGA, and will equip the CPCNG with e CPC-compatibility mode in which almost all old CPC software will run. Exceptions will be anything which makes use of precise timing, since the eZ80 runs at a totally different pace to the old Z80, the video memory arbitration is organised differently in the CPCNG, and there is an extra overhead associated with translation and emulation of CPC I/O.

Refer to the below diagram of the CPC-compatibility circuit. If we use schematic entry for the FPGA we can enter this circuit for the I/O intercept trick. I have marked the main chips in this design with equivalent TTL 74LS-series part numbers, just so that their functionality is well defined (in reality 74LS logic would be too slow to be able to build his circuit for real).



The signals at the left of the diagram connect directly to the eZ80 CPU. At the top, the /IORQ signal from the CPU goes only to this circuit, not to any I/O devices in the rest of the CPCNG.

There is a mode signal, which will come from 1 bit of a mode register elsewhere in the FPGA, which is logic 0 when in CPCNG mode, and 1 when in the CPC mode. When this bit is 0 (CPCNG mode) the /IORQ signal from the CPU is propagated directly to the output /IORQ signal which is sent to the rest of the circuit.

But when in CPC mode, a 1 is forced to the /IORQ output via the top OR gate, so that I/O requests are disabled. When an I/O request occurs in CPC mode (/IORQ goes to 0), this PResets the D-type flip flop IC1. The /Q output of this flip flop is therefore 1 when in normal CPC operating mode, but 0 when an I/O has occurred and the interrupt intercept software is running. This signal is sent back to the CPU as /NMI (non-maskable interrupt). So, when an I/O request occurs, /NMI is generated by setting this signal to 0.

When /IORQ goes back to 1, IC7 (also a D-type flip flop) clocks this signal so that for the next instruction, the I/O is re-enabled. This means that the I/O intercept routine can do the required translated CPCNG I/O. The output of this flip flop also goes to the memory mapping circuit, where it causes the ROM to be switched in at address &000000 so that we can be sure there is sensible I/O intercept code at the /NMI address &0066.

Now see the 8-bit latches at the bottom of my diagram. The NOR gate at the bottom left detects an op-code read. When an op-code read is in progress, the eZ80 /INSTRD pin is low (0). So is /RD and /MREQ, so the NOR output will be 1 only during op-code read. The next rising clock edge clocks this 1 into IC2 (D-type flip flop), whose Q output clocks both IC3 and IC4. These are 8-bit latches, IC3 latches what was on the CPU databus (the current opcode), while its output (the previously read opcode) is moved into IC4.

At the same time as the intercept mode is entered, when /IORQ goes low in CPC-mode and IC1 is PReset, the Q output of IC1 clocks the contents of IC3 and IC4 into latches IC5 and IC6. This means that when the intercept occurs and /NMI Is generated, IC5 and 6 will contain the two most recent opcodes which occured just prior to the /IORQ. That is, IC5 and IC6 contain the Opcode of the I/O instruction! All Z80 I/O instructions are 2 bytes. IC5 and IC6 have 3-state outputs, which will be enabled onto the CPU databus when a certain I/O port is read. This decoding of I/O ports will occur elsewhere in the FPGA.

Therefore, the first thing the I/O intercept software needs to do is read from these two ports to determine what I/O instruction was being performed at the time of the /NMI. This extra hardware isn't really necessary, because you could do it in software: you would need to read two bytes off the stack, representing the return address from the /NMI. Then you'd need to subtract 2 to get the address of the I/O instruction. Then you'd need to do some work with the memory banking to ensure that you could read from that address. Then you'd read it. That will all take quite a number of instructions so I thought it would be beneficial to provide this functionality in hardware since it's more simple there.

Then the I/O intercept software can work out what I/O instruction was being attempted, and perform necessary CPCNG I/O's to replicate the intended effect of the CPC I/O. The CPU registers should be put into the state which the CPC would have expected, i.e. an IN instruction should put the appropriate IN byte in the expected register etc.

Finally the intercept software executes a RETN instruction (return from /NMI). This has the opcode ED 45. The two most recent opcodes are stored in the intercept circuit in IC3 and IC4. A set of inverters and 16-input AND gate on the output of these two registers decodes

the opcodes ED 45. When it occurs, the output of the 16-input AND gate will be 1. When /MREQ goes back to 1, IC1 is cleared, releasing the /NMI, switching out the ROM, and putting the circuit back into normal CPC mode.

That completes the whole interrupt interception operation.