Re: 8301
Postby Nasta » Thu Aug 02, 2018 5:39 pm

The function of the 8301 is to be the main address decoder and the video interface.

While the decoding is fairly simple, it's the video interface that makes things quite 'rigid' in the hardware sense, and some of that rigidity is transferred to software.

Here is an example:

The 8301 uses a 15MHz internal clock to run it's memory controller logic, and runs the CPU at half that. The reason it is done this way is that the CPU internally runs 'double rate', things happen on both edges of it's clock, so one could say that the CPU internally works at 15MHz, so to 'track' what it is doing at any given time in order to know how to split cycles between video and CPU accessing the memory. If one was doing a replacement using programmable logic, it's not really difficult to run this clock at a rate that is inside rather wide limits - original QL logic is after all old, and in today's terms, slow.

However, the 8301 also divides the internal clock by 1.5 to get a 10MHz clock, which is used to generate video (it is used directly as the dot clock for mode 4). We all know that the QL produces a slightly wider than normal picture (the term is 'overscan'), and this is the reason - while deriving the required video timing (which is actually very rigidly specified) is quite simple from a 10MHz clock, 512 pixels in a line at the same rate will produce overscan and will not be compatible with all TVs and monitors, regarding the visibility of the full picture width.

If the internal clock was 16MHz, the dot clock would be 10.67MHz which is exactly enough to produce a fully visible picture in mode 4, although the logic needed to derive video timing would me slightly more complex.

But, this would also run the CPU at 8MHz (meaning various things like drivers for the network and microdrives would have to be 'adjusted' to work) and the RAM would also run proportionally faster, with some complication in deriving the basic timing logic. A deeper analysis shows that the timing and clock were derived based on 250ns access DRAM which was at the time prevalent and being slowly pushed out by faster 200 ans 180ns DRAM, which means prices were falling and likely to be cheap when the QL hits the market - and knowing SInclair, this was an important point, more so than overscan, considering that a full 512 pixels was intended for the 'professional' user that would use a dedicated monitor rather than a TV.

So, you could say that the design is 'too optimized' in some ways.

An interesting aside is that the 'as simple as possible logic' is one more clue amongst several that 8301 and 8302 probably started life as a single custom chip design, to the point that the 8302 bus is connected originally to the local DRAM bus of the 8301, which was separated once the HAL chip was introduced on newer board revisions. I can go into more detail with that at some other time. My guess (not entirely uneducated) is that Sinclair decided to use already proven smaller ULA chips rather than a newer and larger one, due to huge problems Acorn had at the time with bringing it's Electron computer to market due to problems with it's large single ULA, that took several tries (and huge amounts of cash) to get right. It should also be noted that the Electron used then brand new 64k x 4 DRAM chips to reduce price, which would significantly reduce the size and complexity of the QL motherboard, reducing the 16 DRAM chips to only 4 - but with 64k x 1 DRAM already being used in the Spectrum and the +128 possibly in the works, it was a different economy of scale.

We could also look at some other designs of the time. Atari ST had comparable resolutions (in terms of memory used, which also means in terms of bus bandwidth

needed to refresh the screen) and they managed to do it without a noticeable slow down of the CPU, by dividing the 4-clock access cycle of the CPU in a 2+2 scheme with 2 cycles dedicated to the actual CPU access and the other 2 for video - but the ST had a full 16-bit data bus so could pull 2x the data per clock, compared to the QL. While the QL does have 16 DRAM chips 1-bit wide each, in theory it could have used a 16-bit bus, but the ULA simply does not have enough pins for that (although that would have SIGNIFICANTLY simplified the internal ULA logic!), here again reducing to 8 bits eventually reduces cost - the ULA costs the same per given density, no matter how much of it's internal logic makes up useful circuits.

The reason I mention this is because an 8301 replacement would be heavily dependent on the existing RAM layout (unless it supplied it's own RAM, given that a replacement would have to be a board rather than a real chip), in which case one has to either do more or less the same the original did, or be very clever regarding the usage of existing RAM considering a 'lowest common denominator' timing mode as one can never be sure what kind of RAM will be found on a particular motherboard.

A little bit of perhaps not so commonly known history is in order here:
When Miracle systems announced the Masterpiece graphics card, it's basic spec was the same QL modes, but doubled resolution in both directions. Unfortunately, at the same time Miracle became a single person, the late Stuart Honeyball. At the time I was thinking about new graphics and Stuart, being pressed with non-QL projects, effectively dumped his work on Masterpiece in my lap, saying there is no reason to be competition. This was a VERY generous thing on his behalf, which I shall certainly never forget. In any case, later on, Stuart decided to resurrect the Masterpiece as a low-cost alternative to the Aurora for users that would decide to keep their QL's in the original box. This actually plugged into the 8301 socket (with flying lead to bus pin A19) and used the existing two banks of 64k to produce ONLY 640x480 resolution, VGA compatible. I am not sure but I think it would also do 512x256 and mode 8 in half the horizontal resolution as usual, centered inside a 640x480 field, for compatibility - and this required a lot of trickery to get the required bandwidth out of the old slow RAM. Unfortunately, it never happened, for the same reason GoldFire never happened - since Stuart had kindly given me one of his licences for AMD CPLD software, we were both based on those CPLD chips, and AMD decided to sell it's programmable logic business to Lattice, who promptly shut down most of AMDs advanced CPLDs, exactly the ones we would have used. Stuart got wind of this beforehand and found out Cypress was making improved clones, but alas when Lattice killed off the AMD versions, Cypress did the same for theirs - and a lot of work simply went down the drain.

Another interesting tidbit in this story is that the internal video data path of the 8301 is indeed 16-bit. One has only to look at the pixel data organization in memory. MODE 4 is the basic template, with one 16-bit word having one color component in the low byte and the other in the high byte of the word. In order to output a new group of 8 mode 4 or 4 mode 8 pixels to the screen, the ULA has to buffer 16 bits of data, i.e. 2 bytes. In actual fact it goes one step further and buffers 4 bytes at a time (more on this later). This complicates the logic, and... one might say, the way pixels are drawn as well. While MODE 4 drawing makes sense once one looks at the workings of the MOVEP assembler instruction, MODE 8 is decidedly more complex and thus slower. MODE 4 bit organization lends itself to simpler drawing of characters, which start as a 1-bit bitmap in ROM or RAM, but left/right PAN and mode 8 get to be complex and slow, perhaps one reason why there are not so many games on the QL! It is a pity the QL does not have a 'chunky' pixel organization (all bits of one pixel are next to each other in a byte/word/long word). While that does require about 1.5k of tables to help conversion of 1-bit images like characters, to 2 or 8/16 colors, and perhaps the reason it was not done is the ROM was already over-full, it would have made things easier for upgrades. The way the 8301 gets the required bandwidth from memory to display the screen, using an 8-bit data bus, however would have to stay the same even with that way of organizing bits to pixels. It would, however, simplify logic for a possible replacement with faster RAM :/

I will leave the exact way the 8301 generated the screen for the next post, and only say here that CPU access to RAM is heavily interlocked with the screen timing. This logic is simplified as far as it could have been given the underlying bus size and RAM speed, which also makes it not as optimal as it could have been.

What remains to be said here are a few things regarding the reliability of the chip itself.

There are two serious risks to it's health:

1) The video signals (RGB and synch) are directly connected to the monitor, which makes them susceptible to problems caused by the monitor connection - from static discharges to short circuits. The monochrome and composite outputs are much better in this respect as they are in a way additionally buffered and not directly connected to the ULA. Video signals are also available on the bus and susceptible to the same problems as others, like pins being bent and shorted. The signals themselves also have to travel the complete length of the motherboard which does not do good to signal quality - not a huge problem is used as TTL signals which means they will be re-shaped in the monitor, but not so good if used as analog signals. In particular using them as the latter without series resistors will severely overload the video signal pins of the ULA.

2) Heat - the main problem here being the requirement to drive the address lines of 16 RAM chips connected in parallel. Replacing that RAM with 64x4 parts reduces current consumption of the QL by half or more - also reduces the current required from the 8301 to drive the RAM signals by a factor of 4, lo and behold it remains very lukewarm under those conditions. Further, using a GC or SGC reduces the average number of accesses to the RAM drastically, further reducing power consumption and heat - in a GC/SGC system, the motherboard RAM is only accessed for writing, and only for the screen area (usually only screen 0), which means bank 1 os never accessed and also the top 32k of bank 0 unless Minerva is used in 2-screen mode.

Re: 8301
Postby Nasta » Fri Aug 03, 2018 2:34 am

Here is a bit more about how the 8301 works.
Beware, there will be some numbers to understand, as well as a lot about timing based on various clock cycles.

I will start with the more complicated bit, which is how the 8301 actually manages to read the screen RAM and use the data to create the image on the screen.

A bit about CRT screen basics is needed here - and the plus side is, much of the way these used to work is still the underlying logic for more modern flat panel displays.

QL users already know that the MODE 4 resolution is 512 x 256 pixels - so, 512 pixels per each of the 256 lines. However, a CRT monitor does not actually have 'pixels' in the usual sense, but rather each line has a part you can display an image within, and a part that is not seen, and falls 'outside of the screen'. This will already give us a clue why some of the QL's picture get's clipped on the sides - the part the QL uses to display pixels is actually slightly wider than standard and extends outside the screen, into the 'invisible area'.

The CRT displays a picture using a 'raster' of lines, basically it draws the display using a focused 'dot' of electrons hitting the phosphors on the screen. Depending on the current (which is basically the amount of electrons hitting the screen) the point will be more or less bright. The dot is made to move from the

top lefthand corner in horizontal lines, till it reaches the right end of the screen, then it returns quickly back to the lefthand side but a bit lower, so the next line it describes gets to be under the previous line. It repeats that until it reaches the bottom of the screen, then returns back to the top lefthand corner.

While it is doing that, the actual video signal modulates the electron dot and thus produces a picture.
In order for the monitor to know when to return from the right side of the screen to the left, there is a horizontal synch signal pulse that starts the 'retrace' back to the left side. Similar there is a vertical synch signal pulse that tells the monitor when the last line has been drawn and the dot should return to the top of the screen. In actuality, horizontal and vertical movement are independent so it's up to the device driving the monitor to properly generate the synch signals to get a stable picture.

One thing to know is that actual timing back then was based on a standard TV specification, and it is quite rigid. To top it off, it takes time for the dot to react to the synch signals and also to return to the left side, and during that time the dot travels backwards at a higher speed, so the video signal should be turned off (black level) or it would also write an image backwards and with less precision and resolution as the dot is now traveling faster and not necessarily as precisely as in the 'usable' L to R direction. As the definition of one display line is basically the time from one to the next synch pulse, this is why there is a portion of the line that can be used to display video within, while the other, non-usable part is called the retrace period.

The same exact logic applies to the vertical direction, but now the period between the synch pulses is expressed in lines. Similar to how each line is composed from a visible and invisible part, so is the entire frame of lines composed of visible and invisible lines, where the invisible lines now form the vertical retrace period.

For PAL TV, on which the QL video is based, each line takes 64us and there are 312 lines per frame, so the entire frame takes about 20ms to draw, resulting in a frame frequency of 50Hz.

Quick aside: Real PAL TV uses two consecutive frames with 312.5 lines each (.5 meaning the vertical synch pulse happens at about half the 313th line of the even frame and at the end of the 312th line of the odd frame) to get a total of 625 lines of vertical resolution by drawing first the even and then odd lines, at an effective 25Hz rate, the contents of the picture rarely being wildly different between the even and odd frame so the interlacing reduces flicker. However, the QL simply uses the non-interlaced version which reduces the vertical resolution to 312 lines but with twice the refresh rate, which is more suitable to a computer display where contents of lines can be completely independent, so flicker would be quite annoying at 25Hz even at twice the vertical resolution.

Now, remember that I said that not all of a line can be used, and neither can all of the lines be used to display pixels. Also, the signal that modulates the electron dot has a 'bandwidth', or, simply put, a maximum frequency so the number of pixels one can put into the allotted 64us of time is also limited. For a color system it's about 400 or so pixels in ideal circumstances, realistically around 320 if you used relatively high spec commercial video circuits and ICs. However, if you can drive the 'dot' directly, then it comes down only to the circuits in the actual monitor, and the sharpness of the dot - but the basic timing remained (back then) based on standard TV, if you wanted to avoid going bankrupt.

What this means is, one line is 64us, out of which 48us should be used for the actual pixels, and there are 312 lines out of which up to 288 in theory could be used for pixels, in other words, this defines the 'visible screen area'.

Since the QL was intended to have a 80 character text display, and (I will jump ahead a bit here) we know the pixels come out at a 10 MHz rate, this means that each pixel takes up 0.1us, so if 48us is visible, at 0.1us per pixel, that would give us 480 pixels per line. Using a 6 pixel wide character, we would get exactly 80 characters per line. And, if we wanted to use all of the 288 lines available for display, that would give us 138240 total pixels and a 480x288 resolution, and that comes out to 34560 bytes. And we do know that computers rather like things to be numbered in powers of 2, because it simplifies addressing of those bytes.

Let's explain this in a bit more detail.

The way various timing signals are generated by digital logic, is to use a master clock and then count cycles of it to get the various periods and frequencies. Again, knowing that 10MHz is used to drive the video system, it means all timings are derived from units of 0.1us. In this case it means that a whole diplay line (visible plus invisible part) takes 640 clocks at 10MHz, which is why the horizontal synch is 10MHz divided by 640, which gives us 15625Hz for the horizontal synch frequency. This signal is then used to count lines, again visible and invisible ones, 312 total.

If one uses standard binary counting, starting at 0, this means that one line has 640 cycles numbered 0 to 639. Lines are numbered 0 to 311. In binary, 629 requires 10 bits to encode, and the total pixels in a iline get numbered from 0000000000b to 1001111111b, the last number being 639=512+127. Once cycle 640 happens, the counter is reset to 0, so one could detect the combination 1010000000 to reset the line pixel counter - and in fact, this is done by only detecting the two 1's in the entire 10-bit counter, which makes the entire 'reset' circuit very simple.

For the line counter, 311 requires 9 bits to code, and the numbers go from 000000000b to 100110111. Once state 312 came up the counter would be reset, and 312 being 100111000, the hardware can detect the 4 1s in there using a 4-input and gate to reset the counter, so still not too bad for reset logic.

However, getting the address of the data to read from screen RAM to display at the right time, and this goes from address 0 to 34559, from the state of the pixel and line counters, is so complex that it's easier to actually have one 16 bit counter for the address (which is what is needed to code for numbers 0 to 34559) and reset it when the vertical counter is reset, and let it count only for certain combinations of states of the horizontal and vertical counter - namely, only while the horizontal counter counted from 0 to 479 and the vertical counter was counting from 0 to 287. So along with a 16-bit counter (which can be a problem because it takes time for the carry from lower bits to ripple into the higher bits for a simple counter implementation so it could get too slow, and a synchronous counter would have to be used instead which uses a LOT more logic for long counters), one would also need additional logic to figure out when the counter should advance in count and when not.

So what happens if the horizontal and vertical resolutions were some convenient power of 2? This is now getting awfully familiar, as using the closest values would be 512 horizontal and 256 vertical.

This makes the pixel count go from 0 to 511, which fits exactly 9 bits (the reason we used a power of 2 in the first place!) and line count from 0 to 255, which needs exactly 8 bits. So, the numbers go from 000000000 to 111111111 horizontally and 00000000 to 11111111 vertically. Since we are counting starting with visible pixels and lines, this means the initial state of the counters up to 511 for the horizontal and 255 vertical correspond exactly to the address of the pixel, so concatenating the lower 8 bits of the vertical counter with the lower 9 bits of the horizontal counter would directly give us a pixel address. Since there are 4 pixels per byte (given 2 bits per pixel), the bottom 2 bits of such address would give you the position of the 2 bit pixel within a byte and the remaining 15 bits would give you a byte address, from 0 to 32767, which is

exactly 32k. All of this is basically re-using horizontal and vertical counter bits and no additional logic - which is a considerable simplification of logic, which is what you want when designing custom logic that is supposed to be as cheap as possible.

The consequence is that now the horizontal resolution is increased to 512 pixels, which uses 51.2us of the complete display line, which breaks the 48us standard. So, there are 32 extra pixels - the timing is adjusted so that around 16 are added to each side of the 480 pixel visible area, and this is how TV mode was born - have 512 pixels horizontally, and limit the usable pixels in software. Simplifying the logic to 9 and 8 bit addressing for the visible pixels and lines also simplifies the logic that has to do with generating the vertical synch pulse and 'blanking' the display, i.e. to determine when pixels are to be fed to the monitor, or 'black' should be generated during the various invisible or unused areas of the screen - simply look at the top bit of the counter and if it is 1, generate blank (black) pixels.

Even better - it makes it also much simpler to write software for. Figuring out the byte address for a 480 pixel wide display, requires 'take x coordinate and add to y coordinate times 120', so 'real multiplication' while calculating with a 512 pixel width means simply shifting and splicing bytes.

There are also other ways this simplifies the actual logic that reads the data from RAM as well as generates the required timing for the RAM when the CPU reads or writes it, but more on this in the next post.

Re: 8301
Postby Nasta » Fri Aug 03, 2018 9:04 pm

Now before I go into the details of what 8301 actually does on a signal and nanosecond basis, an explanation is needed on the setup of the actual hardware.

As we know, the CPU is, contextually, the 'master' of the system, and does it's work by accessing various parts of memory and input/output devices, using three buses:

1) The address bus, which transmits the address, or in a way, WHERE to find or put data.

2) The data bus, which carries data back and forth, from CPU to other devices, or from devices to the CPU.

3) The control bus, which is a collective name for a set of signals that time the transactions and control when the addresses and data are valid, as well as what direction the data is to travel.

However, since the QL also has a video system to let it's user see a graphical representation of what is going on, the CPU is not the only thing that needs to access memory.

As I explained, the way the picture is generated on the screen is a repetitive process, because the screen only retains the image for a short time (milliseconds) before it fades again, so it needs to be continually refreshed. I also mentioned that the entire picture is a special 32k area of memory which contents are interpreted as pixels on the screen. Since the contents are dynamic under program control, this is obviously RAM.
So, this area needs to be read out over and over again, 50 times a second - and more importantly, this MUST be done at certain intervals and speed, no stopping, no waiting.

We also know that memory is in general 'single port' which means that only one request can be served at a time, so at any given time, it's either going to be accessed by the CPU or read for screen generation purposes - and, since the

latter must be exactly timed, should the CPU want access at the same time, it will have to wait.

In order to somewhat mitigate the problem, the QL design splits the bus between RAM and everything else. There is a 'switch' between the CPU bus and the RAM bus, which makes it possible for RAM data to travel on it's own bus to the 8301 when it reads it to refresh the screen, while the CPU can concurrently access everything else, like the ROM, or anything on the expansion bus. However, should the CPU want to access RAM, the 8301 has priority and the CPU has to wait.

* Small aside: depending on the version of the motherboard, the companion 8302 (and the IPC that communicates through it) can be on the RAM bus side or the CPU bus side. For instance, on ISS5 boards the 8302 is on the RAM side which means accessing the IO registers within it was subject to the same wait while the 8301 is accessing RAM. This changed on the latter boards with the HAL chip.

For people somewhat knowledgeable of the QL motherboard, the bus 'switch' is made out of 3 LSTTL chips, 74LS245 (for the data bus) and 2x 74LS257 (for the address bus). The address bus is also multiplexed by the 74LS257 on the way to the RAM, which is the normal way DRAM - which is the kind used in the QL - is addressed.

So, now we get to the nitty-gritty of how the 8301 does it's work. Somewhere at the beginning I also mentioned that the 8301 is the main system address decoder. So, amongst other things it enables or disables the 'bridge' depending on what address the CPU wants to access. Of course, it has to monitor where the CPU is within the course of the access as well as where the screen access is in its own timing, and synchronize the two. Aside from that, the 8301 lets the CPU access ROM and add-ons at full speed, and asynchronously - basically it lets the CPU 'time itself' when it does that, counting on knowing that the speed of the ROM is enough to execute even the fastest transfer the 68008 is capable of at a 7.5MHz clock.

However, if RAM is accessed, since the screen refresh gets priority, this is the part that actually determines how fast accessing the RAM happens.

* again - a small aside. DRAM comes in certain chip sizes and organizations, and while we also know the 8301 can support two screen areas, 64k total, it actually controls two 64k DRAM banks and actually treats themas a single block of RAM so screen access will slow down the CPU not only when it is accessing the screen area, but any address within the 128k of on-board DRAM. This is an obvious cost-cutting measure as it's not as easy to set up the 68008 to control DRAM, compard to, say, a Z80 in the Spectrum. Spectrum knowledgeable people will know that the ULA, which also controls screen refresh, had shared access only to the first 16k of RAM, while the added 32 (to make 48 total) was not slowed down.

Unfortunately, in order to save more TLL chips and logic, the 8301 controls the entire RAM, even though it can only use half of it to store two screen areas.

So, finally we get to the 'nasty' stuff.

I mentioned in the paragraphs above that the entire timing of the screen refresh is based on the specification of a standard TV raster line. We already know that the QL displays 512 visible pixels in each line out of 640 total, and that each line, having 512 pixels made out of 2 bits each (to get 4 colors), therefore is made out of 128 bytes. It takes 51.2us to display the pixels, so this means that, effectively, 128 bytes have to be read out of RAM to be translated into these 512 pixels, which means the memory bandwidth required is 128/0.0000512 bytes per second, or 2.5Mbytes per second. Just for a sanity check, let's see how fast the CPU can access memory - it takes a minimum of 4 clock cycles at 7.5MHz, so the maximum bandwidth is roughly 7.5/4 Mbytes per second, or 1.875Mbytes per second. In other words - the screen refresh process requires MORE bandwidth than the CPU! But then, let us see how fast the actual DRAM can work, and this roughly comes out to... drumroll: 2.5Mbytes per second.

So.... what now? It seems that there is not enough bandwidth to fit both screen refresh and CPU access!

Well... remember that the screen refresh 'only' happens for 51.2us out of 64us, so that means that in theory the RAM is used for screen refresh 4/5 of the time and the CPU can access it 1/5 of the time. While this would work, it would be downright crippling to the CPU and slow it down by a factor of 5 if it was executing code or fetching or storing data in RAM. We know that 8301 slows the RAM down a LOT but it's not this bad.

So, in one of the rare bouts of being clever, the 8301 uses the fact that screen refresh reads RAM from consecutive addresses, to speed up (well... somewhat) RAM access.

And here is how it does it:

Since we know that the pixels are shifted out from the 8301 at a rate of 10MHz, and it's clock is 15MHz, this means that it's using some convenient common denominator of the two clock to base it's timing on. And, indeed it does - it divides every display line into chunks that last 16 pixels in lenght, which takes 16 cycles at 10MHz and 24 cycles at 15MHz. The 10MHz clock is actually generated from the 15MHz clock by taking 3 half-cycles at 15MHz and using it as a full cycle at 10MHz, so as the 15MHz clock goes 010101 etc, the 10MHz clock goes 001001 etc, this pattern then keeps repeating.

Within the 24 cycles of the 15MHz clock, 16 cycles are used to read 4 consecutive bytes from screen RAM and 8 cycles are kept open for the CPU to access the RAM. Please note that 8 cycles at 15MHz is exactly 4 cycles at 7.5MHz and this is the 'natural' shortest access the CPU can generate if it's left to drive the bus at the maximum speed. That being said, using 8 cycles at 15MHz as the clock to drive the generation of the signals to the DRAM is the natural way to do it with a 68008 (and in general up to 68030) because the CPU itself generates it's control signals at twice the clock rate - it divides every 4 clock access into 8 half-cycles.

* More detail in here:
The 8301 generates a 'page mode' DRAM access to read 4 consecutive bytes much faster than if they were read as random bytes. Out of the 16 cycles at 15MHz it has to do this, it uses 3 to set up the access and then repeats the same 3-cycle sequence 4 times (12 cycles total) to read the 4 bytes, and then one more cycle to finish the access. So, it needs 16 cycles to get 4 bytes.

During the 8 cycles it can use for the CPU, it uses 3 cycles to set up the access, 3 to perform it and 2 to finish it - so it takes 8 cycles for 1 (yes - ONE) byte.

In other words, the bandwidth the RAM is capable of for consecutive data is twice as much as for random byte accesses.

Since the 4 bytes it reads consecutively will make up 16 pixels, they need to be buffered internally to the 8301, because they get read during 16 cycles of the 15MHz clock and get 'stretched' to 24 cycles at 15MHz (which is exactly 16 cycles at 10MHz). It is not exactly easy to figure out how the buffering is done but one could make an educated guess given that we need to save on logic. There are probably 2 2-byte (1 word) buffers, and the first 8 pixels ot of the total 16 read start at cycle 12 of each 24 cycle timing chunk.

* Aside: the 16-pixel timing chunk can sometimes be seen when the digital RGB signal from the 8301 is used to drive an analog input on a monitor or TV, when the screen displays white, one can see the screen seems to have 32 vertical bars that are slightly darker, at every 16 pixel interval, often they flicker slightly depending on what the computer is doing because it modulates the power supply, and this voltage is being directly output by the 8301 as logic 1 on the RGB lines.

Now, I laid out that the 8301 uses chunks of 24 cycles, out of which 8 are dedicated to the CPU. In case you missed it, let me rephrase it a bit: the CPU gets only 1/3 of the total time to access the RAM. And, in case you were wondering, yes it does slow it down to about 1/3 of it's maximum speed when it does. However, we know that when one measures the speed of the CPU having it execute code from RAM, it comes out as working at about half speed. So where's the difference?

Well, I did mention that only 512 out of the total 640 pixels in a line are visible, i.e. 51.2us are used out of 64. The entire line therefore consists of 40 total 16-pixel chunks, but 32 are used for the actual displayed pixels, during which the 8301 does it's 1/3rd bandwidth thing, while in the remaining 8 chunks it lets the CPU run full speed. If we look at it in CPU accesses, at 4 clock cycles at 7.5MHz, one display line can fit a maximum of 120 accesses, divided into 40 groups of 3. In the first 32 groups only one out of 3 is available to the CPU, in the last 8 groups all are available. This means a total of 56 out of the theoretical 120 accesses are usable by the CPU, which gives us an effective CPU speed of around 46.44% maximum, or a factor of 2.143 slowdown.

In reality this of course depends on the actual instructions executed s some have 'internal' clock cycles while the CPU does not use the bus, and can partially overlap with the 8301 reading the screen data.

But wait, you might say, did I not say that also not all display lines are used for actual visible pixels? Yes I did, and you would be right - 256 are used out of 312. So, along with 80% of any line being used for visible pixels, roughly 82% of all lines are used for visible pixels, so it would logically follow that along with 20% of each line time being accessible to the CPU full speed, the same full speed could be had for 18% of all display lines. Alas - the cleverness of the 8301, such as it is, does not stretch that far. Sadly, it still does the same even during the invisible lines, just does no actual reading of data.

* Aside: let's explore for a minute what the effective speed of the CPU would be when running from 8301 controlled RAM, if it actually only used the actually required lines to read screen data:

We already calculated that during the 256 lines of the visible display it lets the CPU have 56 out of the theoretical 120 access slots. During the remaining 56 lines it could theoretically give the CPU all of the 120 theoretical slots. On a full screen basis, we therefore have 120x312=37440 slots, and the math for the actual implementation of the 8301 makes 56x312=17472 slots available. If the unused lines were not used for fake 8301 accesses, this number would be 56x256+120x56=19880. This would mean the CPU would work at 53.1% maximum speed, rather than ~46.5%. It does not look like a significant improvement looking at it like that, but if one compares the two, the larger number is almost a 14% improvement compared to the actual situation. Given that in it's time people complained about this, a 14% improvement would not be unwelcome - compare that to the mere 6.66% improvement if the clock was upped to the maximum 8MHz the CPU would support.

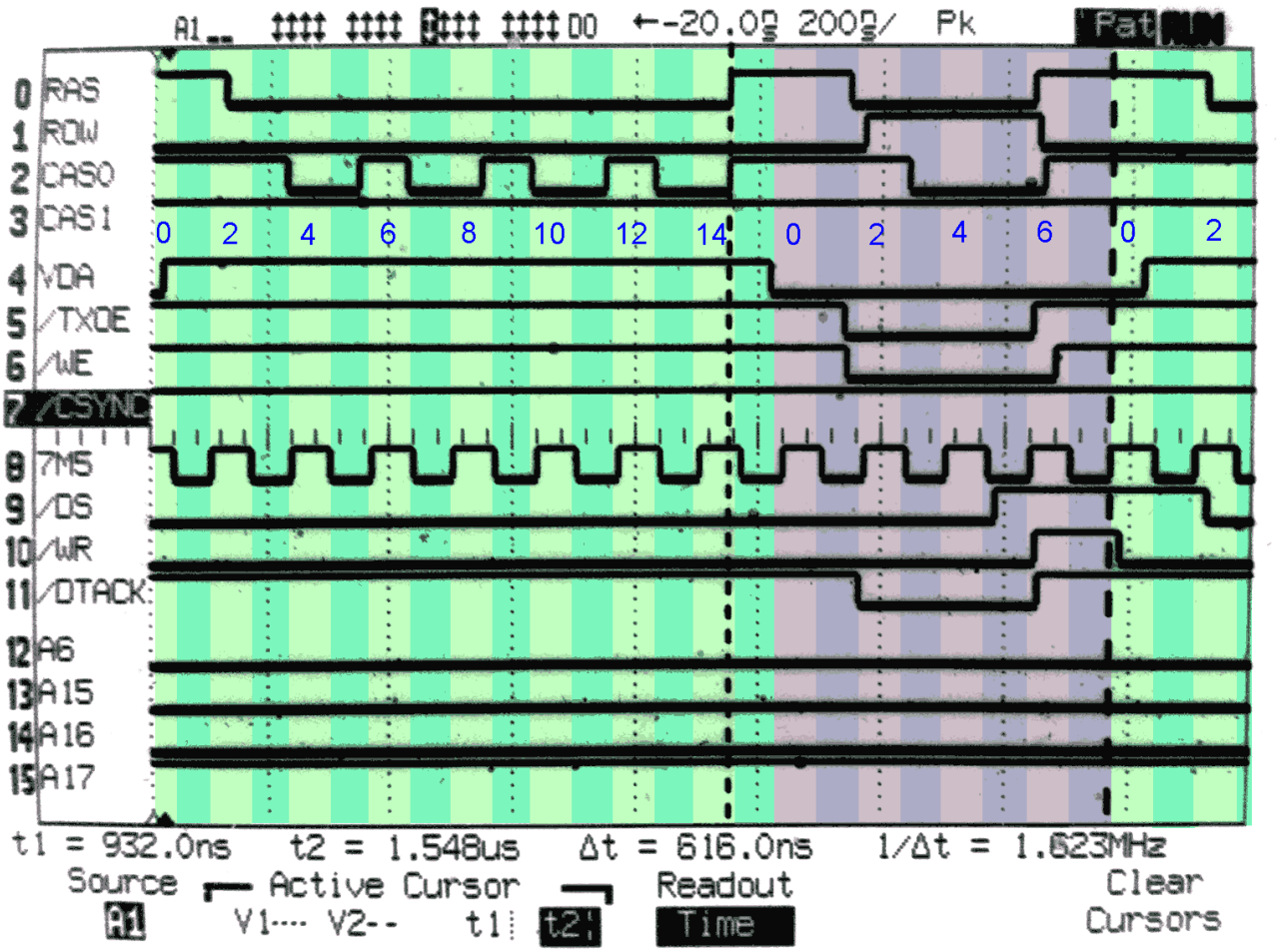
So, why was this not done? The partial reason comes down to the need of the DRAM to be refreshed periodically to guarantee the integrity of the data stored in it. But more on that, as well as other quirks of the 8301 in the final post.

Re: 8301
Postby Nasta » Sun Aug 05, 2018 3:14 am

Well, perhaps I will stretch the final part to several posts as it will be easier to comment the pictures.

So, for starters, here is a logic analyzer trace of the 8301 managing a write to RAM, the CPU having tried to access the RAM before/while the 8301 was reading

screen data:



The green part is the 8301 accessing screen RAM, and the grayish part is the actual CPU access to RAM.

I will explain some of the signals in detail, in order to follow how the actual logic I described makes the relevant signals behave in real life.

The top 8 traces are signals generated by the 8301 that control the RAM and the 'bridge' (buffer and multiplexer) circuits that separate the CPU bus from the RAM bus.

The bottom 8 traces are signals the CPU generates or looks at in order to signal the various devices on the bus what it's about to do, as well as signals the devices must generate in order to signal the CPU how they are reacting. Unlike the top 8 signals, 6 out of the 8 bottom signals are generated by the CPU and 2 are generated by the 8301 - these being the clock signal (7M5, standing for '7.5MHz'), and the /DTACK signal which tells the CPU when the device has finished with the current CPUs access so the CPU can proceed with he next access.

Here is a short explanation of the signals by name. First the CPU signals:
7M5 is the CPU 7.5MHz clock, the ULA 15MHz clock divided by 2. Since this is the only clock on the screen, both states/edges are marked by colored bars since every 15MHz clock cycle generates one edge on the 7.5M clock signal. Both the 8301 and the CPU effectively use both 7.5MHz edges.

A6, A15, A16, A17 are some of the CPU address signals which are relevant for decoding parts of the address map of the QL.

/DS is the CPU data strobe, when it goes low, it signals the 8301 that the CPU is starting an access. It also signals that the state of the address bus and data bus on write are stable.

/WR is the CPU read/write signal, shortened here to /WR as it goes low when the CPU wants to write data, i.e. output data on the data bus.

/DTACK is produced by a device the CPU is accessing when it has done doing what the CPU wanted from it :), at this point the device pulls this line low. The CPU detects this and finishes the current cycle and continues with the next one. In other words, this signal can be used to extend the access when needed. Since the 8301 is the decoder for all internal devices on the QL motherboard, it is responsible to generate this signal.

And here are some relevant 8301 signals.

/CSYNCH is the composite synch signal. It was added to the complement of signals so the logic analyzer can trigger a signal trace on it. The underlying reason is that /CSYNCH goes low when the currently displayed line of pixels ends, so triggering on it can tell the analyzer to start tracing signals at the beginning of a line.

VDA is generated by the 8301 and is high when the 8301 accesses display data. Actually, it is used to disable the address multiplexers on the motherboard (74LS257) to switch off the address coming from the CPU and replace it with an address it generates to access the required display data. It actually also multiplexes it's own internal counters that contain the address but it's done internally so the 8301 only has pins for a multiplexed address. When VDA is low, the CPU is free to access the RAM (*) if it wants to.

/TXOE is similar to VDA but works for the data bus. When high, it disables the data bus buffer (74LS245) which then disconnects the RAM data bus from the CPU data bus, in order that the 8301 can read it without it conflicting with data the CPU might be writing or reading to parts of the address map that are not RAM. There is a difference, in that the signal only goes low (and enables the data buffer) when there is actual data to be transferred, as not all cycles in a CPU access cycle are being used by the CPU to access data, some need to be used to set up control signals.

A subset of the ULA signals are specific to the way the dynamic RAM works, so let me explain them as a separete group. These are /RAS, /CAS (two of them), /WE and indirectly ROW

* A bit on DRAM operation: for various reasons, one of which is the reduction of pins on the chip, the address bus of the DRAM is multiplexed, and it gets communicated to the DRAM chip in two 'halves' usually half the number of bits of the total address. The QL uses 64k DRAM chips, and 64k requires 16 bits of an address. The actual DRAM uses 8 address lines and the 16 bits are input 8 bits at a time, in the context of a row and column address of a 256x256 matrix.

Internally, the DRAM actually accesses a whole row of data once the row address is input, and then the column address selects a part of data (1 bit in the case of the QL) that is to be output to the CPU or replaced by the CPU data. After that, the whole column actually gets written back, which is also how the data is refreshed. This is a crucially important property of dynamic RAM - data is actually stored in a matrix of capacitors, and left alone, they discharge slowly so data will be lost if it is not refreshed periodically. Also, in order to fit the maximum bits to the minimum area, the capacitors are so tiny that reading their state also destroys the data, which is why it then has to be written back from the row buffer.

So, on to the actual signals:

/WE is obvious, and when low it tells the RAM it's to write the data it is given

on the data bus. There are slight differences in the way particular DRAM chips treat it and the 8301 is being quite conservative in the way it generates it so many types could be used - obviously to make it easy to use almost any ol cr*p :)

/RAS goes low to latch the row address into the DRAM chip. It takes the states of the 8 address lines and 'remembers' them internally for the duration of the access.

/CAS goes low to tell the DRAM that the column address is present on the address bus

/ROW is used to drive the external multiplexer when the CPU accesses the DRAM, it selects the address lines from the CPU to present to the DRAM as row (when 0) or column (when 1) - this is the select input to the 74LS257 multiplexers. Note it is wrongly labeled as high active (without the / in front).

The way the DRAM works can be observed in the grayish area on the picture. The timing diagram goes left to right (shows how the signals change as time advances). The gray area is the part of the 'timing chunk' the 8301 uses to let the CPU access the RAM.

In state 0, the VDA signal goes low, for the CPU to access the RAM. The /DS signal is already low, telling us the CPU has already requested an access of RAM well before it got to actually be performed. In fact, if you look carefully, the CPU signals show that /DS is low and /WR is low right from the start, even before the part where the 8301 reads RAM happens, meaning the CPU has started the cycle well before the events we are observing in the diagram. So, it has already been made to wait at least 8 clock cycles. This means that all the address and data lines in the buses have long been stable.

* Aside: if the 8301 sees the CPU is trying to access the RAM from the state of the address lines, but /DS has not become low by 4 full clock cycles before the screen access starts, it will not let the CPU perform a RAM access even if it was 'it's turn' because it would not be able to finish it by the time the 8301 needs to read screen data. So it follows that the CPU can end up taking 16 clock cycles to perform a single byte access - 4x slower than maximum speed.

'Fortunately' the waits due to screen data access are so long that it has enough time to use the upcoming time slots when they come, as it usually starts a cycle sometime close to the start of the next 8301 access (see right-hand side of diagram, the green part after the gray one).

What we see next is that ROW stays low for a while after VDA goes low. The 74LS257 multiplexers have already selected the row address bits (a bit more about this later on) but they are put on the multiplexed address pins of the DRAM only whan VDA goes low. Before that, it was the address from the 8301 that was there. Next, /RAS goes low in cycle 2, and latches the row address into the RAM chips - all of them. A short time after that (and not clock related) /ROW goes high to replace the row address with the column address. The delay is part of the logic but is welcome as some DRAM chips require the row address to remain stable for a while after /RAS goes low (this is called a 'hold time'). Again, there is a delay before one of the /CAS signals goes low in state 3, because time is required for the actual signals to switch over and stabilize (this is called a 'setup time'), and then /CAS goes low.

As can be seen, there are two /CAS signals, /CAS0 and /CAS1. These are connected to the chips making up the lower and upper 64k of RAM respectively. In this case /CAS0 goes low, meaning the CPU accesses the low 64k - and this can actually be seen from the available address lines in the trace. /CAS needs to stay low for a while for the RAMs access time to pass and the data to get written into it.

* Note: there us a larger delay from the clock signal to /CAS going low than from the clock signal to /RAS going low. This is most likely because there is a

clock generated internal /CAS signal that then gets split into /CAS0 and /CAS1 using combinatorial logic, which results in a small additional delay.

We can also see that the /TXOE signal has gone low in order to let the data on the CPU data bus on the RAM data bus, and /WE has gone low to tell the RAM it's supposed to write it. /WE is a sort of gated and buffered CPU /WR signal. The 8301 has rather strong outputs for the address bits. /RAS, /CAS, /WE as they need to feed the signal to at least 8 or the full 16 RAM chips. Supplying the required current is one of the reasons for the 8301 generating heat.

Finally, if we go down to the CPU signals, we cee the 8301 has also now generated the /DTACK signal in order to tell the CPU that it has gained access to the RAM and the data has been written.

The somewhat curious thing to remember here is that /DTACK actually goes low ahead of the actual data write in the strict sense but also, the data has long been written when the CPU figures out it's time to get on with the next access. This is down to two things:

First, the RAM actually latches data when /CAS goes low if it finds /WE low at that time. So, for this to work, data has to be present and stable on the RAM data pins before /CAS goes low, as well as the /WE signal - and as one can see, the /TXOE signal has indeed enabled the data buffer so the CPU data can go to the RAM, and /WE is indeed already low when/CAS goes low. This means that the CPU could just as well continue on it's merry way just a bit after /CAS has gone low (remember, a short hold time is required), but:

Second, the 68008 bus protocol is such that the CPU starts looking for the /DTACK being low at a certain point (well passed in this case) but once it detects it, it will take one and a half more clocks to finish the current cycle.

So, while the 8301 could have been implemented to optimize writes, the logic used has been simplified so that it works the same for both reads and writes, where there are certain subtle differences. The take away point is, the 8301 is based on the 68008 bus protocol and expects a certain reaction of the CPU as it gives it various signals in response to an access attempt, SYNCHRONOUSLY with the 7.5MHz clock it generates. In other words, the 8301 logic expects the 68008 to work off the 7.5MHz clock as supplied by the 8301, so it can work in lock-step with the CPU.

The access as observed in the CPU portion of the trace (gray) is the simplest mode of access of DRAM. When data is read, the only difference is that /WE is not low, and the 74LS245 buffer has it's direction changed, expecting data to go from RAM to CPU. However, this time the data must be present and stable on the CPU bus, as provided by the RAM. This happens sometime after /CAS has gone low and is defined as the /CAS access time, worst case this will be about 1 clock cycle at 7.5MHz. However, we are back to the way the 68008 protocol works, and 8301 expecting to work in lock-step with the CPU, clock by clock - knowing that the CPU takes time to recognize the /DTACK signal, it is set low by the 8301 before the actual data is ready, because the protocol is such that when /DTACK is recognized as low, the actual data is taken by the CPU one clock cycle later. The 8301 logic anticipates this. This is why the CPU cannot be run from an asynchronous clock at a frequency that is much different than the actual 7.5MHz, if it is much higher, the 8301 anticipates it has one clock cycle at 7.5MHz of time to provide valid data to the CPU, but the CPU will expect it one clock cycle later but at a higher clock - i.e. after a shorter period than the 8301 is counting on. So, what happens is that the data is usually being written correctly (due to the RAM internally 'storing' data to be written when /CAS goes low, before the actual internal write happens - so it's like a buffer of sorts), but reading will be incorrect.

Now, let's look at the green portion of the diagram. The reason why I decided to explain that one after the CPU part is that the address multiplexing and switching from row to column is hidden inside the 8301 so it's not easy to

follow without knowing how it happens outside, which is seen when the CPU does it, as discrete logic chips are used to implement the multiplexing and data buffer.

However, what we can see immediately is that during the green part, VDA is high so the address from the CPU is disabled (and replaced by the address from the 8301) directly on the multiplexed signal level, and /TXOE is high, meaning the data buffer is disabled, preventing the data being read from the RAM by the 8301 from 'leaking' to the CPU bus.

Also, /WE is kept high, meaning that data is read - and indeed the 8301 only ever reads data to generate the display.

At the beginning things look familiar, first /RAS goes low in state 2, then a while later, in state 4 comparable with how it happens when the CPU is accessing RAM, /CAS goes low - and in this case it is also /CAS0. In fact, it will always be /CAS0 as the 8301 can fetch screen data only from the bottom 64k of RAM. In all probability in state 3 the row address is replaced by the column address using an internal multiplexer to get the relevant row and column counter bits to the multiplexed address bus pins.

But then something interesting happens, in state 6 /CAS goes high again and then the 3-half-clock period sequence repeats 3 more times, while /RAS stays low.

What is not seen in the diagram is that the two lowest address bits in the column address also change and count from the initial 00 binary in state 3, up through 01b, 10b and finally 11b. at the same time CAS goes back high.

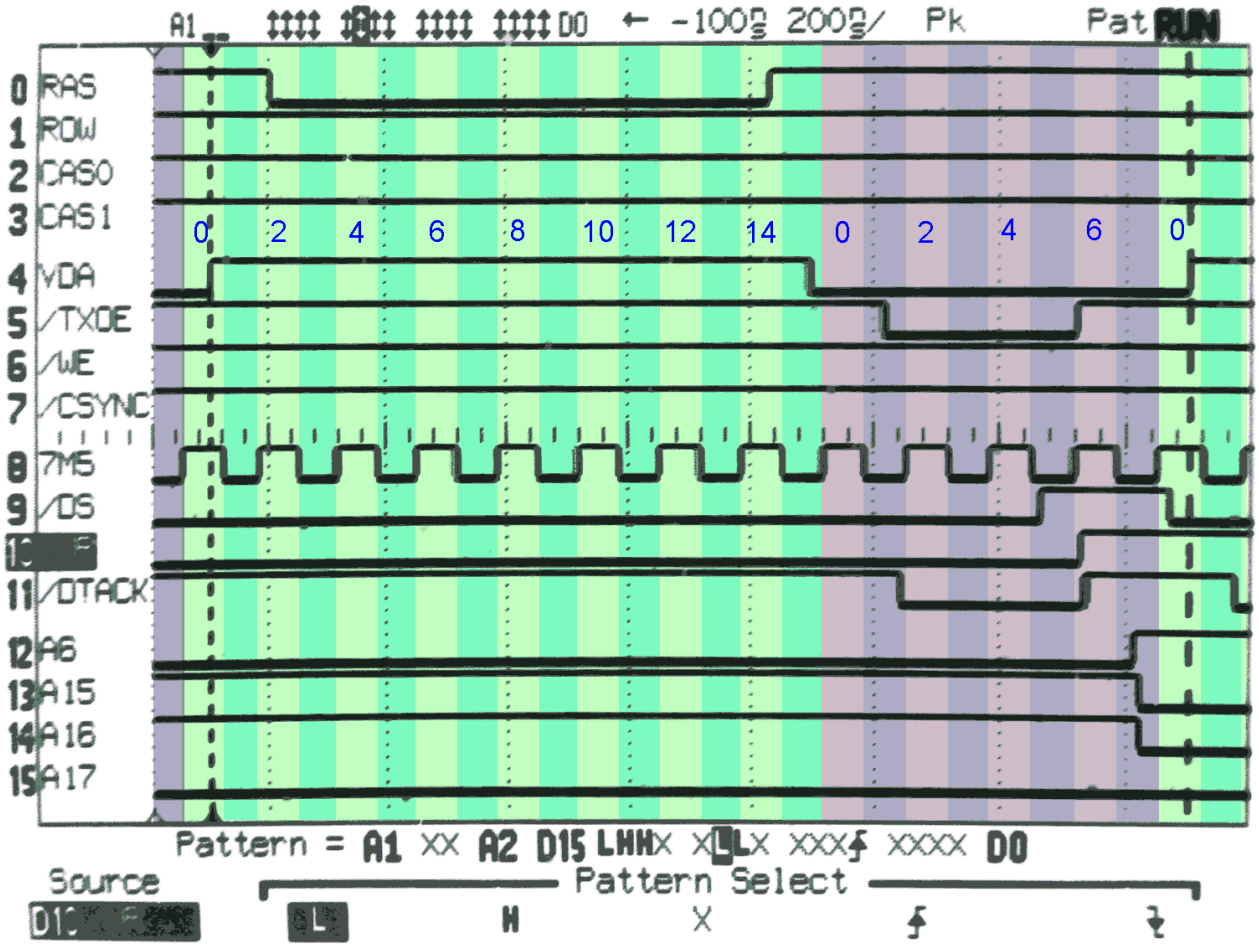
This is a faster mode of access for addresses that are all within the same row in the RAM - as I said, internally a whole row is read and then when the column select address is given, only one out of the 256 bits maiking up the whole row is chosen out of the 256 making the whole row. In this case, the 8301 signals to the RAM it needs to keep the same row address without reading it anew, and read 4 consecutive bytes from the already read row buffer 'in one go' taking about 2x the time needed to read a single byte when the CPU does it. This can be done much faster as the row does not need to be re-read every time. Since the 8301 does not access RAM almost randomly like a CPU does (because you never know what program is running and what order of access it requires at a given moment), but does it strictly in sequence, from the first to the last pixel, it's easy to implement this sort of 'shortcut' to get more speed. One could argue that things would be better if more consecutive bytes could be read and buffered this way, but the problem is exactly the buffer required - remember, 8 bytes are read in 8 cycles, but their contents are spread out as pixels coming out sequentially during 12 cycles of the 7.5M clock. So, this was the best that could be done with the least amount of buffer size.

* I have not been able to time the pixels versus accesses exactly but I am reasonably confident that the 16 pixel block to be generated by the data being read begins sometime in cycle 9, with the just read second byte of data being written directly to a shift register from where it will be shifted 2 bits at a time. The next 2 bytes are stored in an intermediate buffer in state 16 and get transferred to the shift register in state 5 of the CPU access slot.

Re: 8301
Postby Nasta » Mon Aug 06, 2018 2:00 pm

In the previous long post, a signal trace was shown of some CPU and 8301 signals, and this trace shows one of the 40 timing chunks the 8301 generates for each line of display it sends to the screen. In actual fact, it shows one of the 32 'active' chunks during which the 8301 is reading data from RAM that will be output as visible pixels.
The next picture shows a trace of one of the equivalent timing chunks that happens when the 8301 is generating completely blank lines, in our case lines 256 to 311 out of the (0 to 311) that make the whole screen. And, it's actually

a very interesting one as will be shown soon.



Just for a minute, let us loom at the first picture, namely at the state of the address lines shown on the trace.

There are only 4, because this is basically enough to figure out what part of the motherboard hardware is being addressed.

* Note, the traces were taken on a 'bare' QL.

So, the 8301 disregards A18 and A19 from the CPU which means it reduces the address map to 256k and this is then repeated 4 times within the 1M total. The decoding table is actually very simple:

A17:A16=00 and /WR=1 decodes the ROM. ROMOEH goes high and /DTACK is a copy of /DS which makes the CPU perform the fastest possible access in ROM.

A17:A16=00 and /WR=0 just generates /DTACK as a copy of /DS which makes the CPU perform a write cycle that effectively does nothing.

A17:A16=01 accesses the IO area. There are some differences between ULA versions it seems. There are only a few addresses used but the OS will use $18000 as the base address so, for this one also A15=1. It also uses A6 (and on some ULA versions A5) to select control registers within 8302 (A6=0), or 8301 (A6=1 or A6:A5=11).

When the 8302 is selected, the pin /PCENL on the 8301 also goes low, this is the chip select output that drives the equally named input pin on the 8301.

A17=1 accesses the RAM, and when A17=1 and A16=0, /CAS0 will be activated, when

A16=1, /CAS1 will be activated, i.e. A16 selects the RAM bank.

If we look at the previous trace, we can see that the CPU is addressing the first 32k of the RAM (A17:A15=100), i.e. screen 0.

In the picture above, we see something curious - it is addressing the IO area with A6=0, which are the registers inside the 8302, and the /WR signal being low tells us that it is writing data.

Also, we can see CSYNCH is high and an even more curious thing within the portion of the access chunk that is normally used to read screen data - there only /RAS goes low, but there is no /CAS signal and in fact if we look into the /RAM datasheet, this means no data is transferred at all (which is OK since there is nothing to display, these lines are all black) and the RAM is actually being refreshed. So, the CPU is writing a value into the 8302 on an idle QL during vertical retrace - what it is actually doing is serving the frame interrupt - which occurs every time a vertical retrace begins, and it is generated from the VSYNCH signal.

The trace however immediately poses two questions:

1) Why is the 8301 using 8 clock cycles just to refresh the RAM when it could use just 4, freeing twice the usual time for CPU access? And, further (though not shown in the diagram) why does it do it for every one of the 32 chunks that would normally be used for visible pixels in a line?

2) What does the 8302 have to do with this, as the 8301 is obviously letting data on the RAM data bus (/TXOE=0) even though the RAM is not using it?

Well, let us start with the easy one first - and that's question (2).
If I could have squeezed the signal /PCEN (or PCENL as it is written on the QL schematic) into the trace, it would be shown going low at the same time /TXOE goes low, and going back high a bit before /TXOE goes high. In other words, the 8302 is being selected.

Well, the explanation is that the 8302 was connected to the RAM data bus on all issues of the QL motherboard up to 5 (or in any case up to whichever one has the HAL chip on it). The 8301 represents one more small load on the bus along with 16 already existing loads on it. On the CPU side, there are two ROMs and the data buffer chip on the data bus so connectiong the 8302 there would really have made no problem at all. However, it appears that the logic inside the 8301 counts on the 8302 being connected to it's side of the data bus, and the only logical explanation is that these two started as a single design, with a single internal data bus - and the logic was just split without correcting it for the new situation.

And yes - this implies that accessing the 8302 in this case does incur the same slowdown as accessing RAM.

In later versions the 8302 data bus was connected directly to the CPU data bus, but the internal decoding logic of the 8301 had to be circumvented by a piece of decoding logic inside the HAL chip.

Now, question (1) has an easy and a more complex answer.

The easy part is that the logic was made the simplest possible, though... I think Albert Einstein is credited with a saying that goes like this: things should be as simple as possible but not simpler than that. And this may be an example of 'simple than that', but without knowing how complicated the logic inside of the 8301 is, it is difficult to say for sure. If it was a CPLD or FPGA it would have needed very little added logic to free 8 out of 12 total cycles during vertical retrace, so the access to RAM would have been twice as fast. In the grand scheme of things, the difference is not massive, but a small calculation will reveal it is still just a bit more than what one would get by

upping the CPU clock to the full 8MHz.

The more complex part has to do with refreshing of the RAM. As I mentioned before, QL uses DRAM and it requires refresh. The requirement is that all 256 rows should be refreshed (by one of several methods on offer) or at least guaranteed to be read, every 2ms. In other words, the complete set of row addresses must be cycled through tat least once every 2ms. Rather than perform explicit refresh, the 8301 relies on the sequential reading of the RAM when it's doing screen data reads, to go through all of the row addresses often enough. In the previous long post I have shown that there are a total of 56 unused lines, and since all 312 take roughly 20ms (19.96 to be exact), 56 take just about 3.5ms is longer than 2ms, so reading or explicit refresh must not stop during the invisible lines or the RAM will lose data.

Re: 8301
Postby Nasta » Mon Aug 06, 2018 6:59 pm

You can keep CAS quiet but RAS should probably be cycled at some minimal frequency because it is used for the internal charge pump that biases the substrate, I am not sure these old RAMs can work without that. The power consumption would surely be much lower but since it's NMOS there is a rather high quiescent current draw. Also, the address multiplexer would have to be enabled which means they would at least have to constantly drive the address bus.
To be honest, if something like a 8301 replacement was designed, it would probably be a VERY good idea to include it on a re-designed (and at least partially form factor) compatible motherboard.

Re: 8301
Postby Nasta » Mon Aug 06, 2018 10:35 pm

They are not needed but still on nearly all motherboards RAM is soldered on-board so not easy to remove. If they could be removed, sure, there would be no point in even putting any sort of signal on those pins. But since the RAM is there, as I said, I am not sure they like working powered up with no RAS cycling, yet signals present (again, no choice here as RAM is soldered on board and signals we would otherwise need are also connected to them). Of course I could be wrong, but the only thing it does say about this in the datasheet is that before the RAM will work as specified a few 'dummy' RAS cycles have to be performed.

To be honest, if something like a 8301 replacement was designed, it would probably be a VERY good idea to include it on a re-designed (and at least partially form factor) compatible motherboard.

From a technical standpoint that is certainly true, from a retro-computer view I‘m not sure this is the case. The point there is to alter the hardware as little as possible, I think.
I‘d like a minimal replacement that doesn‘t do more than the original ZX8301 except outputting the screen as VGA 1024x512 with pixel doubling and line trippling, for example. Technically inferior but a cool way to keep the old machines working.

You certainly have a point there. Also, for sure displaying the standard screen as a VESA compatible version would be foremost on the implementation list, and in fact that could be done even by adding line buffers to a clone of the existing logic - even with overscan correction. Lesser improvements: Optimized RAM access to get that few % of speed improvement, possibly two more screen areas (using /CAS1 - dead easy really) and perhaps a sort of packed pixel mode 16 (no extra video pins needed.

Re: 8301
Postby Nasta » Wed Aug 08, 2018 1:19 am

I remember tracing this (not sure... Samsung motherboard?) when I was designing Aurora. On it the 8301 is directly on the bus. Also, there is a GAL replacement of the HAL with equations, which also suggest the same thing. There is really no reason for it to be on the RAM side, nor for using the 8301 /PCEN decode, since the HAL does it all over again (and uses, or I guess CAN use the 8301 PCENL pin as well).

BTW there are other quirks to do with how PCENL is decoded - it is generated inside the 8301 using DSMCL (which is actually DSL from the CPU) and also uses DSL on the 8302. I remember being quite puzzled by this and tried just pulling DSL on the 8302 low, and guess what, it worked just fine (though I cannot guarantee it will work on every motherboard and every 8301 version, though). So in essence there is a redundant pin on the 8302... and, for that matter on the 8301 (though it's not obvious).

BTW 8302 bus logic is completely asynchronous and it can work with a much faster 68008 bus, no problem.

In any case, if the 8301 DSMCL signal is used to decode 8302, then it MUST be on the RAM side of the bus because 8301 enables the bus buffer via /TXOE when PCENL goes low. On write it would not make a difference, but if the 8302 was on the CPU side, on read the 8301 would enable the buffer expecting data to go from the RAM side bus (expecting 8302 to supply data there) to the CPU bus.

It's a little bit odd, given that they could have put almost any logic into the HAL given the few input and output signals and their simple relationship to clean up the design, but failed to do it on the first revision that had the HAL.

Re: 8301
Postby Nasta » Wed Aug 08, 2018 1:42 am

But then if you look at what people have been doing with Spectrum clones, it comes down to what path you want to take. For instance, if a FPGA replacement was made I would not be against, say, implementing faster RAM access, modern monitor compatibility and perhaps some minor improvements which could be argued to have been easily possible when the QL was originally made.

My gripe with the motherboard would be components that are problematic to replace or remove when they fail (in this case it would be RAM), and signal integrity. After all I just re-used the 8302 and IPC on the Aurora :P

Re: 8301
Postby Nasta » Wed Aug 08, 2018 11:01 am

The thing with the 8301 is that it is not that fragile itself, but other components and situations not really under it's control WILL kill it. For instance, RAM problems such as shorted address and control pins or dead chips that internally short pins will kill the RAM address/control pins. They are quite robust but also the highest current thing on that chip (RGB and the synch signals may be of the same size, not absolutely sure though) - abuse them long enough and the chip will die a heat death.

And then there is the classic static discharge to any of the monitor signals. I did consider a replacement motherboard, actually made a wire wrapped one a long time ago using a 68008FN.

The idea would be of course to fit it into the original case, with the usual connectors in their familiar places, with some minimal exceptions (like original 'mirrored BT' joystick and serial ports... really??? I hated those with passion), or external MDV connector.

BUT - absolutely it would have some improvements, like proper buffering on the monitor lines, and joystick ports. Old chips that can be safely replaced would be - use a 67008FN and good socket for the ROM(s) or EPROM, and even a flash chip on board. Optimize the 8301 using 128k SRAM (which in today's context costs ~nothing) for shadowing and one single RAM chip (yes there is a nice one that fits just right, and it can actually be used to implement the entire 128k). This would also make it possible to implement a 'turbo mode' with 68008 running completely asynchronously clocked from the 8301 and with potentially a higher clock rate.

Make it low power, i.e. HCMOS logic where needed, and include a switching regulator that can run the whole thing off of 9V only, with power for serial port(s) being generated on board.

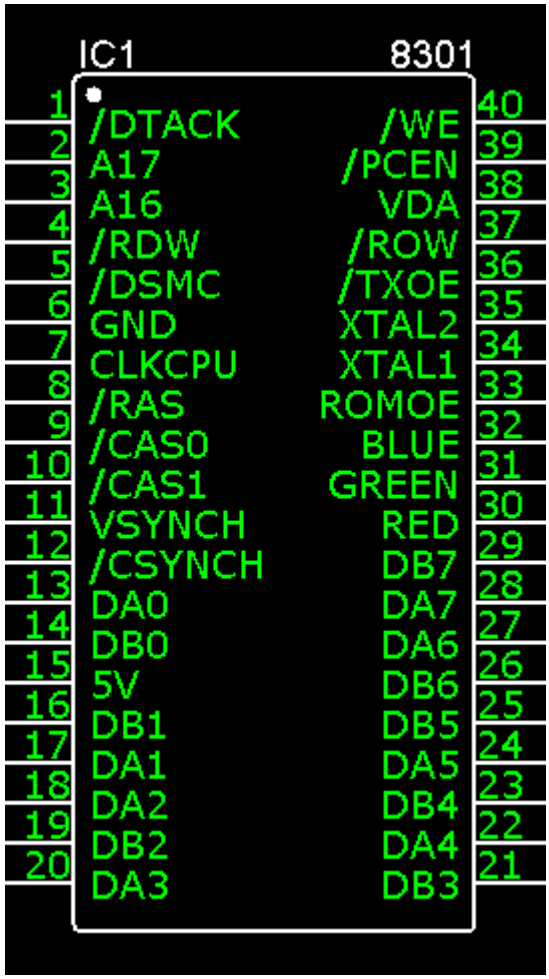
Place things so that the keyboard membrane fits as usual. If I could get Lau to release the Hermes code to the public domain, even using a PLCC 8749 chip for the IPC would be interesting.
The reason behind this would be that the board can be made VERY small, though somewhat odd shaped, but it makes it possible to retract the expansion connector quite a bit further inside so some expansions would fit inside entirely and keep it all neat.

The one thing I would make optional are microdrives, though.
Dave did design a MDV replacement board that makes it much more reliable WRT signal integrity but the only way to use this is to unsolder and then re-solder two critical components, the ULA and the R/W head, and neither is an operation guaranteed to work.

Re: 8301
Postby Nasta » Thu Aug 09, 2018 10:39 pm

OK, let me slowly wrap this up...

Here is a schematic symbol of the 8301 with correct pin-out:



So, here is formal description of the pin functions:

RAM signals:

DB0..DB7 - input - RAM (and in most cases 8302) data bus, connects to the CPU data bus using a 74LS245 bus buffer.

DA0..DA7 - input/output* - multiplexed RAM data bus, the CPU address bus connect to these via a pair of 74LS257 multiplexers, more below.

/RAS - output - RAM /RAS signal (row address select), latches the row address into the RAM roughly on it's falling edge.

/CAS0 - output - RAM /CAS signal for the low 64k of RAM (RAM bank 0), latches the column address into the RAM

/CAS1 - output - RAM /CAS signal for the high 64k of RAM (RAM bank 1), latches the column address into the RAM

/WE - output - RAM /WE, write enable signal (latches the write dada if low when either /CAS goes low). Remains high to read data.

CPU-RAM bridge control:

VDA - output - Video Display Address, when high it disables the multiplexed CPU address provided from the pair of 74LS257, and makes it possible for DA0..DA7 to drive the RAM address bus.

/ROW - output - Selects the row (low) or (column) part of the CPU address bus using the pair of 74LS257 multiplexers.

/TXOE - enables the bus buffer between the RAM and CPU data bus, so the CPU can pass data to and from the RAM and 8301.

Monitor signals:

R, G, B - output - pixel color components (digital), encodes one out of 8 colors.

/CSYNCH - output - composite synch (active low). This generates a low pulse once every display line (horizontal synch), but also encodes vertical synch by changing polarity when VSYNCH (below) is active. Horizontal synch pulses can be separated out of /CSYNCH by feeding /CSYNCH and VSYNCH into an EXOR gate.

VSYNCH - output - Vertical synch (active high) outputs a high pulse for a few display lines to re-trace the CRT to the top of the screen and start a new frame. It also generates an interrupt to the CPU.

CPU/System interface signals:

A16..A17 - input - CPU address lines A16, A17 which decode the basic 4 blocks of 64k containing the ROM, IO, RAM bank 0, RAM bank 1.

/DSMC - input - essentially the master chip select of the 8301 (and thus the QL motherboard), connected to the CPU /DS signal (which goes low when the CPU provides or expects data, and thus signals that an access is in progress). The QL does not use /AS, and assumes that the address bus is stable whenever /DS is low (which it is). More below.

/WR (RDWL) - input - Signals the direction of data transfer, low = write (CPU provides data), high = read (CPU expects data).

/DTACK - output - Data Transfer ACKnowledge, goes low when the 8301 signals that the operation requested by the CPU has been completed. It also uses it to extend the cycle while it's reading RAM to generate the display. As mentioned before, the 8301 expects the CPU to use the clock the 8301 supplies it so that it can work in lockstep with the CPU, to anticipate what the CPU will do, so it actually supplies the signal before the data is ready when RAM is being read. When the ROM is being read (or write cycles are generated to it's addresses), it goes low with /DSMC.
CLK (CPUCLK) - output - 7.5MHz clock generated by dividing the 15MHz 8301 clock by 2. It is expected to drive the CPU clock input.

/PCEN - output - goes low when the 8302 is to be selected, the basic condition is A17=0, A16=1, A6=0, /DSMCL=0. This may be an open drain signal since it is pulled up by a 2.2k resistor, however I have not checked this. The decoding mechanism in the 8301 expects the 8302 data bus to be connected to the RAM data bus so /PCEN is also only low when VDA is low.

ROMOE - output - goes high when the ROM is to be selected, the basic condition is A17=0, A16=0, /WR=1, /DSMCL=0.

Clock signals:

XTAL1 - input - crystal oscillator amplifier input (non-TTL), connects to XTAL2

through a 15MHz crystal and 1M resistor, also to ground via 22pF cap.
XTAL2 - output - crystal oscillator amplifier output (non-TTL), see above. It is possible to get a 15MHz clock by connecting a HCMOS buffer input to this pin and buffering the signal.

Still, there are a few things left unexplained, so let me explain :)

1) How are the RAM address lines multiplexed?
We can get this from the wiring of the 74LS257 multiplexers. So here's how it's done:

RAM DA0 = A2 (Row), A0 (Column)
RAM DA1 = A3 (Row), A1 (Column)
RAM DA2 = A4 (Row), A5 (Column)
RAM DA3 = A7 (Row), A6 (Column)
RAM DA4 = A8 (Row), A12 (Column)
RAM DA5 = A9 (Row), A13 (Column)
RAM DA6 = A10 (Row), A14 (Column)
RAM DA7 = A11 (Row), A15 (Column)

Just for reference, the RAM data bus (and the 8301 data bus) DB0..DB7 maps directly to the CPU data bus D0..D7.

It may be a bit odd that the address lines are somewhat 'mixed up', considering the RAM is organized in rows and then the bit is selected out of a row, one could expect to see the CPU high address lines in the row address and low address lines for the column address. However, this 'strangely' mixed up organization does make some sense.

First, one has to note that RAM address and data lines can be premutated. What this means is that you can connect any address line of the CPU to any address line of a RAM in any order, and the same for data lines - the labeling does not have to be strictly followed - A76543210 xan be passed as eg. A03257614 to the RAM. Any permutation will actually work because although this changes the actual address inside the RAM the data is stored, it is always a one-to-one mapping, and the data is always read from the place it is written as long as the address is the same. The same is true for data lines - the data bits may be scrambled on write, but they are de-scrambled right back on read. It works as long as only one thing writes and reads data, such as the CPU. If there is another device reading from the RAM and expecting a certain layout of the data, the permutation has to be equally applied, so the 'scrambling' of data bits and addresses is the same.

One might as why this would be useful, and one obvious answer is to simplify PCB routing - note that even just on the 8301 the data and address pins do not go strictly in sequence. However, it is sometimes useful when it is known that data may be regularly read is sequence, as it is in the case of refreshing the display.

I have mentioned the need of the DRAM (which is what the RAM type in the QL is) to be periodically refreshed. This is actually done by reading, either explicitly (data is used for something) or implicitly (only the /RAS signal goes low internally reading data but it is never output to the pins, instead only written back when /RAS goes high again). The latter is called a '/RAS only refresh', not very creatively but to the point :) .

* Aside: it is of course the write-back that does the actual refresh of the DRAM data, and it happens regardless of what kind of access is performed, including of course write, which does write new data to some part of a previously read row (depending on the column address) but for all other parts existing data is written back. When read, the data just read is written back. On the QL this is not relevant but in some applications data can also be sequentially written, and this can also be used as a refresh mechanism.

The refresh requirement is that all 256 rows must be accessed at least once within 2ms (Newer RAMs are much better and even 16ms figures are fairly common, but the capacity is higher so there are more rows to refresh ecen if the time limit has been improved). Since the 8301 reads data sequentially, it uses the fact that the address it reads increments sequentially and tries to map appropriate display RAM address bits to row address bits to satisfy the refresh timing.

It has to keep A0 and A1 as a column address as cycling these through the 4 possible combinations of 00..11 addresses 4 consecutive bytes (one long word) and this is used to change the column address inside the same RAM access cycle to exploit faster access times and get 4 sequential bytes out of the RAM in time it would normally take for 2 random bytes.

The 8301 also outputs the entire screen worth of data (32k) in 16.384ms, which means it goes through 8k long words within that period of time. Remember, we must look at long words as this is what the 8301 actually reads (as 4 consecutive bytes) so out of the 15 total bits (A0..A14, which is what we need to describe one out of 32k addresses), the bottom 2 must be used as a column address, therefore we are left with the top 13 bits, which effectively address long words within the screen RAM. If we look at how often the address bits change state when we count up from 0 to 32768 (or 32k), we can start with the obvious - the very top one, A14 goes through both states once in the whole 16.384ms cycle. The next one down, once in 8.192ms, the next one down in 4.096ms, and again the next one down, bit A11, once in 2.048ms - which is JUST shy of the 2ms requirement but is actually enough as the spec is given under absolute worse conditions of temperature and operating voltage. So, this gives us the top address bit we can use as a row address, and all the others have to be of lower significance, if we want to be sure that reading the screen data will absolutely go through all the 256 rows at least once in, in our case, 2.048ms.

All the higher bits, A12 up to A15(*) must be used as a column address (along with the already established A0 and A1), so bits A2..A11 can be freely distributed to the row or column address without breaking the refresh requirement.

* Small aside: 2.048ms rather than 2ms or slightly below might be another subtle clue that the design may have originally been intended to run the CPU at 8MHz, i.e. run the logic at 16MHz. Calculations would show that this precisely meets the requirement to fit the visible pixels of a display line into a 48ms period, which is a pretty major clue.

As we can see in the above mapping table, the row address bits map to CPU/8301 internal address bits A11, A10, A9, A8, A7, A4, A3, A2. That being said, it's quite obvious that we can chose 8 out of 10 bits, so whichever combination we chose, any 8 bits out of 10 will cycle 4 times within the chosen 2.048ms interval (as the extra 2 bits have to go through 4 states without that being reflected in the row address, but rather in the column address). So, in fact, every row is refreshed 4 times - hence the refresh requirement is not only satisfied, but the RAM is 'over-refreshed' by a factor of 4. This could have been used to some advantage, but more on that later.

2) How does the 8301 know to select the 8302 inside the IO area but only when A6=0, when there is no A6 pin?
Well, this is a 'tricky' one which might have cost the designers one extra pin.

The clue is in the state of the /ROW pin when A17=0 and A16=1, i.e. the IO area is accessed. If we look at the first signal trace, when the 8301 reads the screen and the CPU writes it, we can see /ROW is used to switch over the CPU address multiplexers from row to column address, the determining factor being address line A17, which is 1 if RAM is being accessed.

However in the second trace, A17 is zero and /ROW is high, meaning the column

address, or rather, as explained above, CPU addresses A15, A14, A13, A12, A6, A5, A1, A0 are present on the RAM address bus. Well, when A17 is 0, the DA0..7 lines of the 8301 become inputs and are used as inputs to a decoder that further decodes /PCEN to only be active when A6=0. There are some differences on what other lines are used depending on ULA version, so in some cases A15 and A14 are required to be 1 and 0 respectively for the 8302 and for that matter 8301 to be selected. On others, only A15 must be 1, on yet others A15 and A14 are don't care. There is more to be said on this subject, see below.

What is not obvious here is that this is a very strong clue the 8301 and 8302 started life as a single design. Astute enough readers might just have realized that the 8302 'incidentally' uses address bits A0, A1 and A5 to do it's internal decoding, and lo and behold exactly those bits (out of many possible combinations that could have been used) appear in the column address for the RAM bus, and are passed to the 8301. If you connect the 8302 entirely onto the RAM data and address bus knowing about this (not using the HAL chip logic), and tie /DS on it low, it works just fine.

* Aside: this choice of address lines comes with a price, which is not obvious - one extra pin, namely /ROW is needed. This is to differentiate it's function from /RAS, which is often used to switch over the DRAM address multiplexer, along with doing it's standard DRAM function - because the multiplexers are not infinitely fast, the row address is presented when /RAS is high and will reman a bit after /RAS goes low, which is long enough to satisfy the row address hold time for the DRAM (the time the row address has to stay on the address bus once /RAS went low - this is actually usually zero for most DRAMs so that's easy to satisfy). With this arrangement /ROW is needed to present the column address without /RAS going low. The irony is, this pin could have probably been avoided anyway as only /RAS going low will just refresh an address in RAM (no harm done except a bit of power usage) and there is more than enough time for either the 8301 or 8302 to be accessed during /RAS being low and abiding to DRAM timing. Alternatively, as was shown under (1) other address line mappings to row and column addresses could have been used. It seems, however, that the designers got stuck on using A0, A1 to select internal 8302 registers which are part of the column address. Ironically the only time they are used is to read the real time clock seconds counter as a long word, which is REALLY not something that happens often - it could have just as well be read as a sequence of bytes from other addresses. This would have made it possible to use the row address bits to transmit the required address bits for decoding so /RAS could remain high if it was also used to select the row/column address.

3) How does the 8301 decode and use it's one and only control register?
Well, the display control register (usually referred to as MCR, master chip control register) is a write only register that actually has only 3 bits. It is normally accessed at $18063 (hex address, to make it clear) but has many aliases inside the IO area. As was already mentioned, depending on 8301 version, these aliases may take either the entire 64k block starting at $10000, 32k, or 16k starting at $18000. The aliases happen because not all address bits are used for decoding the actual hardware register so all addresses where only the used bits are taken into the decoder will access the same hardware.

A lot of this has already been said above, so essentially the 8301 looks for /DS=0, /WR=0, (A17=0, A16=1), optionally (A15=1;A14=0), A6=1, optionally A5=1) and A0,A1=1. Using $18063 satisfies all of these conditions including the options. The register is write only and attempting to read should be considered to produce random data. From this it also follows that the RAM data bus pins on the 8301 are always inputs as it only ever gets data either from RAM (when generating the display) or from the CPU via the bus buffer, when MCR bits are being written.

Only bits 1, 3 and 7 are implemented:

Bit 7 selects the screen area, known as screen 0 (when 0) and screen 1 (when 1) - this actually appears as address bit A15 in the multiplexing scheme when the

8301 is supplying the multiplexed RAM address.

Bit 3 selects the resolution. Plenty has been said about this elsewhere. Suffice to say that if it is 1, it concatenates two successive mode 4 pixels into a 4-bit pixel with 3 bits used for the RGB components and one as a flash toggle. The hardware does this after RAM data has been serialized into mode 4 pixels.

Bit 1 blanks the display when 1, but, as has been shown, this does not stop the 8301 from 'reading' the screen RAM even if it is disregarding any actual data. I would consider this a missed opportunity, even though it would be reminiscent of the ZX81's 'fast' mode :) - more about this in the final installment.

So... next and finally: how could it have been improved without significant complications, and perhaps guidelines for a re-implementation.

Re: 8301
Postby Nasta » Fri Aug 10, 2018 3:27 pm

There are some discrepancies regarding the HAL chip as some signals are unused on the ISS6 board, but would have to be on the Samsung board for the 8302 to be on the CPU bus side (like gating off /TXOE when the 8301 detects the 8302 I/O addresses). I did not go tracing those as it is obvious from the logic what it's supposed to do, an it does actually work, so I guess it's also doing it :)

Re: 8301
Postby Nasta » Fri Aug 10, 2018 3:51 pm

Have also a look at the maximum period required for /RAS. It does have a maximum (as long as it is) so it cannot be completely static. But I am sure at that rate the current consumption would be FAR lower.

This suggests that you would implement the entire 128k inside the FPGA? Otherwise a part of the original RAM or some other RAM would have to be used. I would strongly vote for 'other RAM' as I'm guessing a FPGA with 128k block SRAM would be a huge overkill when it comes to the amount of logic it contains which would basically remain unused. Actually, you could probably implement a 68008 in there too.

Given the FPGA would come with lots of pins and the whole replacement would have to be a small PCB I vote for a smaller FPGA that can contain SCR0 and 1, and a 128k SRAM on the side to implement the actual 'CPU' RAM, and shadow the video RAM inside the FPGA. This way DB lines could still be kept input only and perhaps even simplify and streamline the logic as one could buffer the write to the screen RAM inside the FPGA and not worry about data being read, since that's provided from the SRAM chip.

Perhaps only through a discrete buffer and that should be disabled for VGA compatible timing. This way it wold still be possible to remain faithful to the original hardware for users who insist on the original monitor(s). Anything faster would I think need a separate output anyway. VSYNCH and /CSYNCH need to be provided at close to standard TV intervals for SGC compatibility (HSYNCH derived from VSYNCH and /CSYNCH is used to trigger DRAM refresh on the SGC).

Given that most output signals would either remain strapped to a constant level or can be 3V3, there are not that many that need fast level shifting - something like a quickswitch also provides 3V3 clamping and could turn out to be feasible, eg. for DA0..7 and DB0..7. Or even just series resistors and internal PCI clamps with the help of a capacitor or several to filter stuff where it's critical, especially for CPU signals that did not come through a TTL multiplexer and bus buffer. For the latter the added capacitances of the RAM chip pins might even be of some help.

Re: 8301
Postby Nasta » Sun Aug 12, 2018 4:20 am

So, to finish off, here is a short recap of some 'quirks' of the 8301, which could probably have been 'differently engineered'.

I should also put in a warning - all of this is (as 'educated' as I can make it) largely conjecture from the available data. Perhaps more could be known by de-capping the chip, though a better idea would be first figuring out the difference between the various versions (2 CLAxxxx and one ZX8301 that I know if).

Some ideas are based on knowledge of PLD hardware but in truth, gate arrays (which were the first custom logic available) have some fairly crucial differences. In particular, routing of signals is a point where huge differences can accrue - migrating a single layer metal to double layer metal process can mean the difference between fitting a circut even possible or not, onto a same give 'sea of gates' type chip. However, in most cases I think the points below have merit.

1) The issue of overscan and 7.5 vs 8MHz CPU clock.

As was amply explained, almost the entire operation of the QL RAM to CPU interface is based on how the 8301 accesses screen data.
So, we know that it uses a master 15MHz clock to derive all timings, and that 'time' in CPU sense is divided into 64us intervals, which are further divided into 40x 12-clock chunks. These are further divided into thirds (which are 4 clocks long each). During each 64us interval, 32 12-clock chunks have only one 4-clock period dedicated to CPU data access, while 8 have all 3 4-clock periods dedicated to CPU data access. When you crunch the numbers, you get a dot clock (I will be using MODE4 as reference) running at 15MHz/1.5, so 10MHz and the 32 chunks where screen data is accessed add up to exactly 51.2us of pixels, i.e. 512 pixels per line.

This is even continued during unused lines with the sole difference no actual data is being read to generate the pixels, though the actual RAM access does take place as a refresh cycle.

So, what would we need to do to get our 512 pixels into a 48us period, which is what is defined as 'standard', i.e. no overscan.

Obviously, the clock would have to run faster as more 'chunks' where pixel data is read need to be fitted into a shorter time. Once the math is done, you end up with exactly 8MHz for the CPU clock, which means a 16MHz main clock and still a /1.5 pixel clock which comes to 10.6666MHz.

However, the line interval, 64us, needs to remain unchanged and this is a bit of a problem. Given that the whole logic is based on a 12-clock timing chunk, and there are 512 clocks per one 64us line, it's rather obvious that 512 is not wholly divisible by 12, we get 42.66666 12-clock chunks.

So we can either extend this to 43, which slightly lengthens the 64us standard line period to 64.5us, which is still just in tolerance for a TV.
We can also shorten it to 42, which shortens the interval to 63us, which might even be more compatible because US NTSC uses a fater line rate and most TVs are more tolerant to faster rather than to slower rates.

Or, we can use different logic that makes it possible to have a 4-clock based timing for the period where no pixels are displayed during a scan line - and this would be the most complex change to the logic, as it has a 'special case'. The two former ones involve different logic to detect the number of the chunk where the line period should end, which is a LOT simpler. While it needs very slightly more complex logic for this, the most complex part, i.e. the various counters that generate the timing and the RAM timing state machine remain the

same.

Along with a more compatible picture, it comes with a speed increase, which is slightly over that of just increasing the clock to the CPU. The reason is that now we have more timing chunks per 64us line, but the number of them used to access screen data has remained the same, so more are available for CPU access to RAM.

Before there were 40 chunks of 3 4-clock periods, giving us 120 total 4-clock slots, 56 of which could be used by the CPU. This is a 46.66% ratio.
Now we would have 42 chunks (lets take the worst case implementation) of 3 4-clock periods, giving us 126 total 4-clock slots, 62 of which could be used by the CPU. This is a 49.21% ratio, but the CPU clock is also increased by 6.66% so the total increase over the current scheme is about 12.5%.

There is however also a drawback. The CPU clock is used to derive some timing, notably the serial port baud rate and, certainly at least possibly and therefore more problematic, microdrive data rate timing in the 8302, so as easy as this mod would have been for the 8301, it would imply a lot more problems for the 8302 logic.

2) What about the 56 unused display lines where the display data is still being 'fake read' and the CPU is being slowed down?

Well, this is a tougher one to crack. The question here is, do we want more CPU speed or perhaps, if we really need to sacrifice memory bandwidth, how about not making the reads 'fake' but real ones, and get a higher vertical resolution?

Let's deal with the second idea first:

Given that there still need to be counter bits to count the lines all the way from 0 up to 311 total, no change is needed to that part of the logic. What would be needed is a different logic that determines when the invisible lines start, which is now trivial, as we have visible lines going from 0 to 255, and need 9 bits (0..8) to count to 311 which is past 255 that 8 bits would give us, we simply use bit 8 to tell us if the lines are visible or not, as well as what type of screen data read should be done (real or refresh).

The PAL standard defines the maximum number of visible lines as 288, but this only gives us 24 lines of retrace which means it's likely to be a problem on some TVs.

However, old QL users will remember there was an 'extended 4' QL emulator which indeed could do 768x288 and in fact Aurora also supports this resolution, as well as 512x288. What needs to be done is more complex logic than just looking at bit 8 of the line counter to determine what the invisible lines are and what the position of the vertical synch pulse needs to be to center the extra 'tall' display inside the monitor or TV screen, and, finally bit 8 of the line counter must be passed as address bit A15 to the RAM address multiplexer in the 8301, instead of bit 7 of the MCR register. This would have meant that 368064 bytes would be used for display memory, reducing the amount of free RAM by exactly 4k. It would also require a different approach to switching between screen area 0 and 1, as now it's not neatly fitted into a power of 2 number of bytes - if this feature was deemed important enough to begin with as a trade-off vs having more vertical resolution. The obvious and possibly even more convenient solution would be to move screen 1 to the beginning of the top 64k of RAM, as that only changes the logic used to decode that /CAS1 should be used for screen data reads by the 8301, rather than /CAS0.

On a regular 128k QL (and possibly even on a 640k, don't have the data to calculate the size of the slave block table for 640k RAM) it would fall into the free RAM and could simply be reserved by job 0 using some simple tricks, and used for various things including games. The one game I am aware that used it had to completely use stand-alone code as using screen 1 meant not having any

system variables in the usual place, and before Minerva, that meant not having an OS once the game is loaded. Not a huge problem, but if one wanted to use two screen areas WITH the OS, this alternative solution would have probably been better.

Other than some slightly more complex logic and 4k less of free RAM this would have no other serious repercussions to the bare QL as we know it - even if the second screen was not implemented at all as a result, and it could be said that the response would have been positive - it would certainly put the QL at the top of usable graphics resolution at the time.

So, back to 'wasting' cycles during the invisible display lines.
Normally there are 56 of them, and some of what will follow has been written about in the previous posts.

If we go back and look at the way address bits are presented as row and column portions to the RAM, and given that the row address counting through all 256 rows is important to insure proper RAM refresh, we can see the following pattern of address bits:

Row address 7..0 = {A11, A10, A9, A8, A7, A4, A3, A2}.

Given that these are generated from the line and chunk counters that are used to generate the display timing, and we know they go sequentially through the 32k screen RAM, 4 bytes at a time. Given that there are 128 bytes per line, or 32 long words, address bits A6 down to A2 form a 5-bit counter that counts the long word to be accessed, from 0 to 31, starting with the lefthand side of the screen. A6 carries over into A7 and A7 to A14 then count the visible line number. We have already established that we need to go through a certain number of addresses within ~2ms which is what determines that A11 should be the top address bit of the row part of the RAM address.

However, we see that the row address bits do not continue down from A11 to A4 but rather A6 and A5 are 'skipped', or invisible. This means that when we look at the row address as a whole, A2..A4 count from 0 to 7 (000 to 111 binary) and then repeat this 4 times before A7..A11 change and count up by one, because A5 and A6 have to go through 00 to 11 (4 states total) to carry into A7, but we do not see this since A5 and A6 are not part of the row address.

What happens is that a sequence of row addresses XXXXX000, XXXXX001, etc, to XXXXX111 where XXXXX counts up for every new display line, appears 4 times within each line. This means that when the RAM is not actually read, but only refreshed during the inactive display lines, it is actually refreshed 4 times over, where once would have been enough.

This could have been used to free up bandwidth for the CPU to use.
Lets take a step back.

We said there were 40 chunks total of 12 clocks each per line, 8 of which were completely free to be used by the CPU. If you take the above in consideration, the refresh pattern 'just happens to repeat' every 8 chunks. And there are 5 total of these 8-chunk blocks in a scan line, and 1 is always free for the CPU to use, no screen data is read or refreshed. So, the simplest way to free more of them during the 56 invisible lines would have been to actually generate only ONE 8-chunk block with 8301 accesses. And the most logical way would be to simply reverse the logic for the visible lines - when a line is visible, use 4 blocks for the 8301+CPU, 1 block for CPU only. When a line is invisible, use the previously CPU only block for the 8301+CPU in refresh mode, and the previously 8301+CPU blocks for the CPU only - which actually SIMPLIFIES the logic as it re-uses the already existing logic to determine the visible vs invisible part of every line.

What would be the result?

Well, as we said before, each line has 120 4-clock slots, 24+32 used for the CPU, the rest for 8301. Multiply this by 312 lines, so we have 37440 total slots, 17472 used for the CPU (56 per each of 312 lines), the well known 46.66% utilization figure.

In the modified version, we start with the same total of 37440 total slots, 256x56 used by the CPU during visible lines, and 56x104 used by the CPU during invisible lines. This is a total of 20160 slots used by the CPU, 53.85% utilization figure, a 15.4% improvement.

We could push this a little further by shortening the time the 8301 uses to access RAM in refresh mode to one 4-cycle slot rather than 2, because it is using the same timing as if it was accessing 4 bytes of consecutive data, when in fact it is not accessing data at all. One 4-cycle slot would have been enough for refresh.

This would rise the number of slots used by the CPU from 20160 to 20608, giving us a 55% utilization figure and a roughly 18% speed improvement - and this would be about the maximum one can have without seriously changing the hardware. Things would not be that good if it had been decided to extend the vertical resolution to 288 lines.

Here are the figures: The simple version would get us 49.74% utilization, a mere 6.6% improvement over standard. The more complex version eeks out 1.1% more at 7.7% improvement over standard - but mind you, with 12.5% more pixels on the screen (32 more rows).

* Interesting bug (quirk?) I forgot to mention before:

Astute readers will note that with the row addressing set up as outlined, the 8301 reads or refreshes all rows every 32 display lines. What is not obvious is that there is actually a bug in the refresh scheme because the total number of lines is 312 and this is not wholly divisible by 32 - it comes out as 9.75. This means that all rows get refreshed within 2.048ms which is just to spec, only 9 out of 10 times. The tenth time only 3/4 of the rows get refreshed, so the other 1/4, namely the top 16k of both 64k RAM banks does not get refreshed. Doing the math tells us that this part of the RAM gets refreshed normally 8 times, followed by one refresh every 3.584ms rather than the specified ~2ms. While this may, now that you know about it, seem alarming, in real circumstances DRAM can retain data FAR longer than the refresh spec - as some have noted when the QL is quickly powered off and back on while holding the reset button - often most pixels of the display such as it was at power off remain preserved on power-up. That being said, there was a simple way this could have been avoided, and that is with a different mapping of screen address bits to row and column address. If the row address was made up as:

Row address 7..0 = {A9, A8, A7, A6, A5, A4, A3, A2} all the rows would be refreshed every 8 lines, and since 312/8=39, there would be no partial refresh. The above logic to get a better RAM bandwidth utilization figure would be different, though, the simple reversing of 8-chunk blocks could not be used, but it would still not be that complicated.

* Aside: The presence of the 'screen blanking' bit in the 8301 MCR register is curious to say the least as it has to my knowledge never been used by the OS. The obvious use would be a 'screen saver' though it would actually only display a black screen rather than switch off the video signals, which is perfectly fine given that the monitors, let alone TV sets had no notion of power saving and would not switch themselves into a low power (almost off) mode when there is no signal present for a while.

But there is a much more interesting and not obvious use, that has to do with a lot of what was discussed above, regarding memory bandwidth utilization. What if the blanking bit switched off the 8301's display data reading mechanism since no data is needed to display a blank screen, with the consequence of

freeing clock cycles for the CPU to access RAM during invisible (blank) display lines? Given that there still needs to be a refresh scheme in use, we can use the same idea but extend it to all lines when the blank bit is set. Under such circumstances we still have the aforementioned 37440 total potential CPU access slots, of which only 2496 would be used to refresh the RAM, giving us a 93.33% utilization figure for CPU-RAM bandwidth. Given a program and data in RAM, the QL would have been exactly 2x faster. While it's difficult to imagine a scenario where this could be exploited to execute some program faster, one interesting one does come to mind. Imagine you had some QL's networked to yours but with no screen attached or... simply, running some software remotely? It wouldn't be that bad to have them run rather faster than usual in that sort of configuration.

3) Why not 16 colors instead of flash?

The obvious answer would be, we need an extra pin for that. This would mean a different connector for the monitor as well. I'm not going to say anything about adding one more video related signal to the expansion connector J1, as IMHO having the original RGB there was not a very good idea to begin with (but then I would be putting my own foot in my mouth a bit as a long time ago I actually used these signals to drive some highly experimental hardware to get more colors and resolution out of the 8301... but quickly went to picking them off the 8301 socket instead).

In fact, there are several ways a pin could have been freed for other purposes on the 8301.

The most direct one would have been to change the way the 8302 chip enable signal /PCEN is decoded inside the 8301. Choosing address lines that are in the column address part of the RAM address for that was a dubious choice, because it requires a separate signal to control the address multiplexer in order to tell it to pass the required address bits to the DA lines of the 8301. Usually the /RAS signal is used to flip from the row to the column address for the RAM as the row address is required to be stable when /RAS is high and persist for at most a very short time (shorther than it takes the multiplexer to actually replace the row address with the column address) when /RAS goes low, after which the column address can be put on the DA lines. As it is, a separate signal is needed because /RAS must stay high not to start an access cycle in RAM while the 8302 is being accessed.

However, one could argue that /RAS could still have been used as the multiplexer select signal anyway, since there is no /CAS generated when the 8302 is accessed so the RAM would just internally do a refresh while data was actually being read or written from or to the 8302. While it does increase the RAM current consumption a bit, the 8302 is accessed so infrequently compared to anything else that the trade-off would have been well worth the extra pin to use for much more clever things.

Be that as it may, let's explore the possibility of having a way of getting more colors without having to use an extra signal or turning the RGB lines into analog signals - which is actually not possible inside the type of chip used to implement the 8301 ULA.

The clue that points the way is there when one loads a MODE 8 screen but forgets to actually change the screen mode to MODE 8 and leaves it in MODE 4. Surprisingly, the picture is very much recognizable, except for flashing bits if there even are any.

So, the trick would be to display 512 pixels in a line instead of 256 but interpret the 2 bits for each of the 512 pixels using different color components for every even and odd pixel. There are numerous ways this can be implemented.

* Quick aside: such a trick was used on the original Apple II video board but on the level of luma/chroma component signals rather than RGB signals).

I am not certain that the following is not a particular quirk of the version of the 8301 I have explored, but there is a slight shift in the horizontal position of a MODE8 picture with respect to MODE4 which suggests that MODE8 is actually generated from mode 4 pixels internally to the 8301, rather than implementing a different way to shift bits of the 4-byte daya buffer into pixels, depending on the mode selected. The way FLASH works is also an indicator as the flash bit latches the rest of the bits concurrent with itself in the same MODE8 pixel, which ideally means it would have to be present a short while before the others, or there would be a slight delay due to the extra latch - in either case it implies logic that converts two MODE4 pixels to one MODE8 pixel.

Without such logic, the trick I mentioned above to get a semblance of more than 8 colors using only 3 digital signals limits the usability of the display as it actually produces a stippled pixel, which does not look 'smooth' enough for every combination of colors, so we would get a rather odd selection even if extra logic was used to do more complex mapping of bits to RGB components depending on the state of said bits and not only on even or odd pixel.

Depending on how far one can go, the results can actually be quite useful. One of the more creative ways to do this would be to 'abuse' the way the 10MHz mode 4 pixel clock is generated from the main 15MHz clock of the 8301. I mentioned a while back that 3 consecutive cycles of the 15MHz clock were used to generate 2 cycles of the 10MHz clock by putting the point where the 10MHz clock goes from an even to odd cycle in the middle one of every 3 cycles at 15MHz. When looking at the signals with a scope. one can see that the implementation is partially done with combinatorial logic as the periods of the even and odd 10MHz clock cycles are not exactly the same. A modification of the logic to deliberately produce different lenghth even and odd pixels (such as 2 15MHz clock cycles for even and 1 cycle for odd) can be used along with some mapping logic (usually with one or two pixels total delay) to implement a 'pulse width modulation' scheme on the RGB lines and get a decent representation of 16 colors. Actually 16 out of 64 discrete combinations can be chosen.

Downside: more logic but it is not overly complex, but the screen is a stippled one and without some form of filtering (which would BTW interfere with regular MODE4 making it blurry!) would produce moire patterns on a color monitor and dot crawl on a TV.

While the 'top end' implementation of this logic indeed is capable of generating 64 'colors' given only a single bit for RGB, I am virtually certain there would have been no way to implement palette functionality inside the 8301.

How do we know this? Well, I am sure some of us wished for a 4 out of 8 colors MODE4, and this only requires 4x 3-bit 'memory' to store the 4 pallete entries for MODE4, that's 12 bits of storage. If that did not fit, a 16x6 bit storage plus 64x6bit look-up table ('ROM') hardly could.

All that being said, there is another aspect of MODE8 which is actually very annoying and may have well contributed to the QL's market fail because it makes graphics for games slower if you want to have it smooth at the same time - MODE8 has a very odd bit to pixel mapping. While it could be said that MODE4 mapping makes sense once one looks at the MOVEP assembler command (which later comes back to haunt us on 68040 and 68060...) and saves us a number of fairly small lookup tables in the already chock-full original ROMs, it does complicate things for games. A simple chunky 4 consecutive bits per pixel, 'packed nibbles' organization would have made things much easier and faster. One thing that it certainly would have helped is already having a usable 16-color mode for business applications when higher resolution hardware became possible.

One could also argue using the flash bit as a 'hold' (no flashing) instead could have been much more useful, as it would give the QL the ability to very quickly create filled polygons on screen (though not in many colors...). So, I'll leave this bit up for discussion...

4) Why not 4 screen areas given that there are already 2 banks of RAM, 64k each?

Well, this is a true mystery to me because out of all of the improvements or alternative ways of implementing stuff, this one would have been the easiest. It does require an extra control bit in the MCR register (that's very little logic) and a line from it to the already existing decode logic for /CAS0 and /CAS1. The latter might actually be tha problem as this supplies an 'internal' bit for A16 to be used when the display data is read in order to refresh the screen. Routing is one of the BIG deciding factors on gate array utilization as it takes the same space that logic would normally take, made out of closely located gates connected by short 'wire' segments. Long wires (or busses) actually pass over un-committed gates that obviously can't then be used for logic, so this is a trade-off.

Granted, this is perhaps the least useful modification to be proposed, but then having two screen areas to begin with did not prove to be very useful.

One could argue, however, that since one is already using a given type of ULA and there is no added charge whatever the mask is that connects it's gates into a usable circuit (production costs don't depend on the pattern), one might as well put in small features even if they end up never been used as the only cost is a slight increase in engineer-hours used. Unfortunately, we all know that despite the huge delay, the QL was rushed to market. While I don't think that if any of the mods here proposed would have made a huge difference, things do add up and... who knows.

For instance, implementing all of the above would get us:

1) No horizontal overscan (though perhaps a more hazy picture in MODE4 on a TV), compatible with many monitors.

2) 63.7% memory speed utilization with the standard screen resolution, 59.95% with vertical resolution extended to 288 lines

3) 6.66% higher CPU speed, resulting in a speed improvement of 45.6% in speed of execution from RAM for standard and 37% for extended resolution, over stock QL.

4) Obviously 32 more display lines (12.5% improvement) and at least a more game friendly 8 color (if not 16 color) display. Pretty much cutting edge graphics in that price bracket at the time (business oriented, not game).

5) Full speed 'dark screen' mode when needed (given that it is a multitasking system, there could always be tasks that do things in the background when the screen saver goes up!).

6) The ability to do double-buffered flicker-free displays without having to dislodge the OS.

Would that have been of interest? Well, I'll leave this to you all.

Re: 8301
Postby Nasta » Sun Aug 12, 2018 12:57 pm

On average it should take the same as most of the time (4/5ths of it) the 8301 incurs at least an 8-clock wait. However, one could (very carefully) construct a limit case. This would happen every time the CPU attempts to write a byte just before the 8301 starts reading a long word from RAM. At this point it will ignore the CPU some clocks in advance of it's read slot if it deems there is no time to finish the CPU access before it has to read screen data. It only looks at /DS, so the fact it goes low later on write will make it skip a potentially valid access slot. A similar situation can happen if it is occasionally writing data to RAM while the visible portion of a line is in progress, interspersed

with accesses performed elsewhere - it can miss potentially valid access slots. This is indeed because the logic is based on /DS timing on read, where it comes 1 clock cycle early. On write, the 8301 counts on that one extra clock so it appears to it the cycle will take too long so it stalls the CPU.

Perhaps this is also a point where improvement could have been made, especially since the RAM actually latches the data provided by the CPU on write so looking at address lines and /WR could have been used to start a RAM cycle (/RAS goes low) and if /DS and /WR are found to be low right before /CAS is to be generated, it could safely proceed with the write counting on data having been latched on the falling edge of /CAS - even if the actual CPU cycle then extends into the next screen data read. If not, /CAS is not generated and the cycle ends up being a refresh which does not produce nor alter any data.

However, given how 8302 decoding is handled (as if it was RAM) obviously the logic was simplified as much as possible and re-used for both reading and writing RAM as well as the 8302 and 8301. Making it more sophisticated would have complicated things - but then one could argue there could have been logic to spare had the 8302 decoding been re-done to connect it directly to the CPU bus, as it could have been right from the start. 8301 MCR write would easily be catered for because it's write only and very simple.

In the post above I have only mentioned improvements that can be handled inside the 8301, without breaking compatibility with existing motherboard versions - If one could only count on the 8302 being directly on the CPU bus, there would have been savings in logic in the 8301, not to mention the existence of a 'HAL' logic chip to implement some small parts of logic that would save us pins on the 8301, for even more streamlining. But alas, it is what it is.

IF the motherboard was re-spun even with the original 8301 and 8302 many improvements can be made, one of the better ones being the ability to run the CPU from a clock independent of the 7.5MHz the 8301 provides and screen RAM shadowing.

Replacing the 8301 with a FPGA based PCB with RAM on-board (either the PCB or the FPGA or both) opens up a whole world of possibility, even just with the re-implementation of the basic functions. For instance, it's practically a given that full CPU access speed can be had AND line doubling VGA AND asynchronous/independent CPU clock all at once. Even a simpler non-VGA version based on an 128k x 8 static RAM would do a lot, as the common SRAM chip of that capacity can perform an entire access in a little more than half a CPU clock - so plenty of timing space to time-multiplex CPU and screen access. One could base the logic on alternate clock cycles where even is CPU and odd is screen access, at 7.5MHz a typical 80ns SRAM would cover the standard screen refresh needs without slowing down the CPU. A FPGA with 64k internal SRAM for both screens with an external 128k SRAM added as shadow and the other 64k of RAM (to emulate a 128k QL) would easily do QL video at VGA compatible timings with line doubling/tripling - and run the CPU with an independent clock and no waiting.