Greetings Karl,

Below are the questions I'm managed to pull out of my brain in the last two days (and a few from another 4A user). Thank you very much for taking time to do this! I've broken the questions down into 3 primary
topics: 99/4A, 9900 (or 9995 as it were), and the 9918. I tried to stay related to the topic as possible, but I'm also so interested in chip making that I may have strayed into a design question from time to time.
;-) Answer as much or little as you like and please take your time. I love stories and all the smallest details, technical or not (the more technical the better :-)), so be as verbose as you like.

I'm on my way to check out Joe Zbiciak stuff, thank you very much for the link. The Atari and Coleco seem to have a stronger following these days, at least from the new software development stand point, and I'd love to see some new stuff for the 4A too.

Thanks again!
Matthew

99/4A

Q: In your 1993 interview you mentioned that GPL was developed because the designers were hoping for a CPU that would execute GPL directly.
Was that GPL-CPU going to replace the main system CPU, or be a co-processor?

A: My understanding and I had this second hand, is that the 99/4 designers wanted to have their own custom CPU that ran GPL directly. When management wouldn't pay for a new CPU, they decide that they would do a real time GPL interpreter. Their hope was that the 99/4 would be so successful that they would get their own custom GPL CPU for the second generation.

By "hiding" the 9900 instruction behind GPL, they were trying to make it easy to transition to a native GPL CPU some day. But in trying to reserve room for what they really wanted, they severely impacted the performance of the first system.

Q: Do you know why the 4A was crippled so much? (For example, the interrupt pins from the 9901 are wired such that all interrupts appear as level 1. It seems that much of the possible functionality was disabled on purpose or a victim of poor design.)

A: To answer this it helps to have a little history of that time period. The 99/4 was originally slated to use the 9985 CPU and not need the 9901 at all. The 9985 CPU was going to be a variation of the 9940 "microcomputer" with embedded EEPROM. The 9985 replaced the EEPROM with 256 Bytes of on-chip RAM. The 256 bytes of on-chip RAM was for the GPL and Basic interpretation, and had to be shared between instructions and data.

The 9940 was a terrible design, it was big, slow and had a huge number of design bugs and to top it all off, there was huge pressure from management. So the 9985 was waiting behind getting the 9940 debugged and the 9940 was very hard to debug the way they designed it.

After many mask releases (masks used to make the silicon) of the 9940 and still with many bugs in the design, they made the modifications to release the 9985. The 9985 then came out and after several masks releases it was still full of bugs. During this time, we had finished the 9918 and I was assigned to help debug the 9985.

It became clear that the 9985 would not be ready anytime soon so the Home Computer group decided that they would have use the 9900 or 998~~0~~*[5]* (an~~d~~ 8-bit interface version of the 9900). They ended up deciding that the 998~~0~~*[5]* while cheaper would be too slow and so they went with the 9900. The 9900 was pretty old by this time and was not very integrated. The plan then became to build around the 9900 those features of a 9985 in hopes of cost reducing the computer with the 9985. So the feature set of the 9900 plus hardware around it was that planned for the 9985.

Now the 9900 had a 16-bit CPU with a 16-bit ALU and a 16-bit external interface but it was pretty inefficient and took 14 clock cycles to do a register to register add. The 9985 had only an 8-bit ALU and 8-bit I/O and took about 10 cycles to do a register to register add. Inside the 9985 the interface to the 256 bytes of RAM was also only 8-bits wide. To emulate the 9985, the 99/4 had to add hardware to covert the 16-bit bus coming out of the 9900 to act like the 8-bit bus of the 9900.

You have to remember that the 9900 architecture was a "memory to memory" interface in that the "register" resided in memory. So an 8-bit interface mean that it took 2 cycles to fetch an instruction, 2 cycles to fetch the source operation, 2 to fetch the designation, and 2 to save the result. Both the 9900 and 9985 had very poor instruction pipelines and so they Fetched and instruction, then took time to decode it, then fetched the operands, then did the operation, and then wrote the result.

The 9900 was not really complete without the 9901, in particular they wanted the CRU peripheral interface, but they didn't want to use any features of the 9901 that were not planned for the 9985.

So what you were left with was an old 16-bit CPU, that had hardware added that throttled its 16-bit interface to make it emulating the 99~~9~~*[8]*5 that was emulating the GPL CPU. Then you add to it the crazy way they had to go through the 9918 to get to any RAM other than the 256 bytes near the CPU. This is why the integrated circuit designers said "the 99/4 had the best bag of parts but they were put together wrong."

After the 9940/9985 debacle many designers had quit or were not trusted by management. The 9918 looking like a successful design and I was the "bright young kid" with some knowledge of CPUs. While helping debug the 9985, I figured out how to do a register to register add in only 4 cycles (versus 14 for the 9900 and 12 for the 9985) and that we could build a true 16-bit CPU in fewer transistors than it took to build the 9985. So even though I was only a year and a half out of school, I became the chief architect of what was first called the 9985A but was later renamed the 9995. The 9995 was designed in England (I got to spend 6 months there) and we only had one bug that could quickly be fixed. But while the 9995 got designed into the 99/2 and 99/8 and there were working prototypes, by that time TI's management had had enough and cancelled the home computer.

Q: Is is true that only 1 or 2 software titles for the 4A were actually written on the 4A, with most of the development taking place on a 990 Mini?

A: Sorry I can't help you with this one as I didn't have any direct involvement with the software development. The first game I ever saw was "W~~a~~*[u]*mpus" which was a sort of educational adventure game which if you failed these giant teeth of the W~~a~~*[u]*mpus would close down on you.

But just speculating, I can think that the 990 minicomputer could have been used in much the way people use a PC today for writing and assembling code. The 990 hardware itself behaved a good bit differently because it had a memory management unit and some other hardware. Back then the 990 while running the same assembly instruction had a lot of other hardware that meant almost no software written for the 990 would run on the 9900. We use to say "the only software that ever ran on a 9900 and a 990 was the assembler."

I don't know, but there was also a 990 board line that was based on the 9900 and did not have all the extra hardware of a 990 minicomputer. This is total speculation, but I would think that they might have used the 990 to write and assemble the software and then run it on a 900 board.

By the way, after the 9995, I was assigned to the 99000 which was used in the next generation 990 mini computer. We had to add a bunch of things to support the 990 operation, most importantly hooks for the memory management unit, some special 990 only instructions, and hooks for floating point.

Q: How many people worked on the 4A's design and implementation? Where you involved in the process in any way?

A: I don't know about the 4A's design. I know there were 6 Engineers the 9918 and we had about 8 on the 9995.

9900

Q: Can you describe the design goals of the 9900?

A: I was in college when the 9900 came out. I read a story that it was the basic instruction set was defined in less than a week. It was patterned after the DEC PDP-11 for the 990 minicomputer. TI was highly successful with the TMS1000 4-bit microcomputer using in all kinds of products from microwave ovens, to automobiles, to hand held game, but TI felt it had miss the boat for the 8-bit CPU and microcomputer (a microcomputer being defined as one with self contained memory).

So TI decided to leapfrog the 8-bit market and build a slightly cut down version of the 990 that became the 9900.

Q: What was the reasoning behind the off-die (is that the correct term?) registers? Many people sight the fast context switches, but there must be something else (it does not seem to be that much of an advantage.)

A: This is a little bit of "which came first the chicken or the egg?" I think they liked the idea of off chip memory for cost reasons and the fast context switch suggested it could be an advantage. My guess is that it was mostly for cost reasons and the "advantage" just help seal the deal. The concept was known as the Workspace Register File, with a Workspace Pointer (WP) register that pointed in memory to the registers.

As one that designed two instruction set compatible follow on chips, I can tell you that it was a performance albatross. It could be easily shown that after only a few instructions, you had spend more memory cycles going for registers than it would have taken to save the registers after an interrupt. So unless you were being constantly context switching (a context switch being an interrupt or major subroutine call) a pathological case were you don't do anything.

The other thing you have to realize is that most compilers of the era tended to save everything on stacks when doing a context switch because the stack could be any size were the register file is limited. So often the "free" saving of the entire register file was superfluous.

On the 99000 we considered a workspace "cache." This is what the higher performance 990 minicomputers used. In effect, there would be a hardware set of registers and you would keep track which ones had been modified. If you got a context switch then you would then save to memory which ones had been used and only bring in registers as they were needed. But you also had to have hardware to monitor the currently being cached workspace memory addresses because it was "legal" to directly address the workspace register in memory. Due to the cost/complexity of the workspace cache we didn't implement it on the 99000.

Very interestingly (and much to my surprise) the SUN SPARC architecture uses a "register window" concept that is essentially the same workspace concept. They did implemented a register file cache but they had several generations newer technology and could afford the transistors. I always thought the SUN SPARC was kind of dumb for going back to this concept having lived with it four about 4 years between the 9995 and 99000.

Q: If given fast enough RAM, can the register access ever be as fast as real internal registers? The 9995 has 256 bytes of internal RAM for the registers, but is that comparable to the way other CPU's of the day implemented their registers?

A: With the 9995 and 99000 we had internal RAM that would let us make an internal access in 1 cycle (where the 9985 with it 8-bit interface would take 2 cycles). The 256 bytes was shared between instruction and register.

With the 9995 I defined a pipeline that only took 4 cycles for a basic addition and we had a 16-bit bus to internal memory while supporting a 8-bit interface to lower

the external system cost of that era. We still had to Read the Instruction, Read the Source, Read the Second Source (same location at the Destination), and then Write the Destination. I pipelined the Read of the next instruction while the ALU was doing the ADD so that if there were a series of additions there would be solid memory cycles.

To get performance you would have to first split the instruction memory from the data memory so you could access them both in parallel. By having a small number of registers you could make a dual port register file and thus access both the source operands in a single cycle. With pipelining you could then get a register to register add down to 2 cycles which is basically what the Motorola 68000 could do. With even more hardware the operation could be pipelined down to 1 cycle with a 3-port register file, but that would take a lot more hardware (too much to be cost effective for that era).

As I said in the prior question, we could have built a workspace cache, and gotten the two cycle addition, but we would have been a lot more expensive than other CPUs as the workspace cache would have been expensive to implement.

The bottom line is that while the workspace concept made the earlier 9900 cheaper to implement, it was a performance bottleneck when we could a few years later afford to put a register file on the CPU but couldn't because of compatibility.

Q: The 9900 vs. 9995, other than clock speed (I believe the 9995 was designed to be clocked at 12MHz), in your opinion which one is a better CPU?

A: I'm certainly biased here, but the 9995 by a mile. First I had the opportunity to look at the 9900 and the 9985/9940 schematics and sadly the later 9985/9940 was so poorly architected that I just ignore them. The 9985/9940 had the false economy of an 8-bit ALU emulating a 16-bit CPU, but since all the registers had to be 16-bits and with the complexity of performing 16-bit operations on an 8-bit ALU, it actually would have been cheaper to have had a 16-bit CPU on the 9985/9940 which is what I did on the 9995.

The 9900 had a 14 clock pipeline even thought the 16-bit ALU could do an addition in 1 clock. In fact it really only did an operation every other color so it took 7 states with each state taking two clocks. To be fair, the external memory operation took two cycles so this probably propagated back into the 2 cycles for every state. The pipeline for a basic addition was (working from my memory) something like:

1. Read instruction
2. Decode instruction and Compute Source (WP + register number)
3. Fetch First source
4. Compute Destination Address
5. Fetch Destination
6. Do Addition
7. Write Destination

And remember each of the above "states" took two cycles.

The pipeline on the 9995 (and 99000) was:

1. Read instruction (while prior instruction doing and ALU operation)

   (Decode instruction while prior operation is writing result and computer source register address assuming there will be a source)

2. Fetch Source and compute destination register address
3. Fetch Destination

   (Do Addition in the ALU while fetching the next instruction, this "counts" as the first cycle of the next instruction)
4. Write Destination (while decoding the next instruction)

This let us do the basic register to register add continuously at a net of 4 memory operation and 4 cycles. By fetching the next instruction while we were computing the ALU operation and then decoding it while we were writing, we were not only doing

things in parallel, we were using different hardware so it was inexpensive to implement (we still only need 1 ALU). We knew were the source register field would be for either single two operand instruction so by default before we knew what the instruction was we went ahead and computed it just in case we needed it (which we would for many of the instructions).

Q: Personally I'm not very fond of the way the 9900 does I/O, what was the reasoning behind the design? Is it superior in some manner to the way other CPU's do their I/O?

A: I assume you are talking about the CRU I/O. Yes it was a very bad concept. It took the whole address bus, tide up the whole data bus and you only got 1 bit of I/O transferred per cycle. It was a very misguided idea to save cost. You basically had to add another chip, the 9901 to then have I/O. I think the "advantage" was that it would let you have a huge number of potential I/O location but only with the expense of many 9901's. The CRU instruction was the third most complicated instruction on the 9900 (after divide and multiply).

What you may not know is that originally they wanted the 9918 to have the CRU interface as well! Can you imagine fetching basic instruction 1 cycle at a time? Pete Macourek primarily (as I was too new at the time, but I agreed with him) fought putting CRU as the CPU interface on the 9918 and that is how we ended up with the 8-bit parallel CPU interface on the 9918. The 9918 was the first so called 9900 family peripheral to not have CRU by rather a parallel I/O interface.

Something else on the 9918, Pete (at bit of a "rebel") was really thinking about the 9918 as independent from the 9900 family. This actually led to much of the 9918 family success as it was used often with the Z80 (Colecovision and the MSX computer). The I/O including the CPU and DRAM interface was originally numbered "little endian" with the LS-bit numbered 0, where the rest of the 9900 family was big endian. At the end of the program they made us change the documentation to make it cosmetically "big endian" by renumbering the bits, but this caused a lot of confusion particularly on the way the DRAM addresses behaved (board designers were constantly hooking up the wrong address lines).

Q: Why is the address bus missing the least significant bit? (It seems, even on a 16-bit CPU, that it could have come in handy, and there was certainly room on the physical package, seeing as how there are several "not connected" pins.)

A: I assume you are talking about the 9900. I'm guessing they felt it was superfluous on the 9900 as it always fetch 16-bits. Saving 1 pin was considered a significant cost savings at that time. I think the 9980 and 9981 had the extra address bit.

Q: Why was a read-before-write design chosen over the ability to directly address individual bytes?

A: Sorry, I am a big confused by what you are referring to (remember this happened almost 30 years ago). Generally, they often did things inefficiently to save cost. They were building a 16-bit machine and the 8-bit mode was just an "option."

On the 9995, I believe we always fetched the one byte we needed. We also fetched the least significant byte first as I remember it. It turns out Home Computer designers did not like our improvements and it delayed the design of the 99/2 and 99/8 as they were emulating the old 9985 which fetched in the other order and did extra cycles.

Q: When a shift instruction has a count of zero (SLA R1,0) the lower 4-bits of R0 are used for the count. If those low 4-bits are zero, the count becomes 16, however is seems more logical that the count would just stay zero and the instruction would do nothing. Can you explain the reasoning behind this functionality?

A: This is one of those things were there are 17 possible operations and you don't want to spend another bit to get beyond 16 (2**4). I think the way they did it was biased in favor of getting the status bits set. I know I ran into this issue myself in designing the 340 Family and the 320C80 instruction sets. Without taking the time to look it up, I can't remember how it came down in those cases.

Q: Are there any undocumented opcodes or "features" for the 9900?

A: I don't remember any, but then I didn't design the 9900.

Q: If you could change anything about the 9900, what would it be?

A: There were 3 fatal flaws in the 9900 in order:

1. The workspace pointer architecture was a performance killer. While it saved some cost on the first 9900, it was a big bottleneck on subsequent generations.
2. The CRU I/O interface was both expensive and slow for I/O
3. The Memory address reach was too small with only 16-bits/64K bytes. This was almost forgivable as everyone including Intel and Zilog (but not the later Motorola 68000) stumbled with how to go beyond 64K. On the 99000 we ended up having two different memory management units. An expensive one for the 990 minicomputers that had their own more complex method and a simpler one for smaller application. But the time for address management only compounded the problems with the workspace pointer registers.

9918

Q: Can you describe the design goals when you set out working on the 9918? Were you able to start with a clean slate, or were there some existing criteria that had to be considered, possibly for compatibility?

A: For the most part, the 9918 was a "clean slate." I don't even remember the home computer being that much of a consideration at first.

I was only at TI about 3 months when I was assigned to the 9918. I have already related about the fact that Pete Macourek had to find to put a parallel rather than a CRU interface on the 9918. In hindsight, that was the most important design decisions because if it had had the CRU serial interface nobody would have used it.

As I wrote earlier, even the CPU and DRAM interfaces were designed "little endian" (Intel bit ordering) to be more compatible with the 8-bit CPUs of the era. We only cosmetically renumbered them for documentation as being "big endian" 9900 family members.

Pete worked on the CPU interface with the people from Home Computer. Granville Ott is the person we mostly interfaced with. Granville was very involved later in the video interface (I will write more about that in answering your next question).

They had already decided to have "Sprites" (a term coined by Dave Ackley a TI manager that started the program but soon when on to do other things). There were only going to be 4 total when I joined the program, but Pete and I worked out the Sprite processing logic with a "sprite stack" and "sprite pre-processing" that let us support 4 sprites on a line with 32-sprites on a screen.

I came up with the DRAM interface timing (I still have the timing diagram I drew to convince the managers that it was the way to go). The problem was that the bus turnaround time to go from reads to writes was too slow. We were given a write requirement by the Home Computer folks that we had to have so many data write slots for the CPU interface per unit of time and there just would not be enough time to turn around the bus. That is when I came up with the idea of having multiplexing the DATA Out on the Address lines since DRAMs at the time all has separate Data-In and Data-Out pins. This allowed us to quickly go from CPU writes to display reads.

When we did the DRAM interface, we were really pushing the DRAM cycle performance. In fact at the time (1977) running the 9918 at 5.4MHz was considered very fast (most other chips were 3MHz or slower) and doing a DRAM cycle every two clocks was really pushing it.

The original 9918 was designed for 4K DRAMs because we figured that they would become cheap soon for Video game. Supporting 16K DRAMs was almost an afterthought. But what happened is that 4K DRAMs became so cheap that everyone stopped making them, so all

the systems ended up using 16K. I don't know/remember, but this may have influence the Home Computer designers to use the 9918's RAM for main memory.

As a side note, my work on the DRAM interface for the 9918 in later years led to my helping define the Video RAM and later the Synchronous DRAM (SDRAM) commonly found in almost all computers today.

Using DRAM on the 9918 was pretty radical at the time. Other graphics chips (used in say Intellivision by Mattel and the older Atari chip sets) had a few hardware player graphics (similar in function to Sprites) and a very limited background that came from SRAM which was expensive and thus they could only afford simpler backgrounds.

Q: What were the limitations present when you were designing the 9918 that limited the resolution to 256x192?

A: In video we often talk about "magic numbers" that come from various multiplications of various numbers. Since the 9918 was going to drive the NTSC video directly (the 9928/9929 for Europe did it externally), we had to use the TI's color burst (color clock) which is 3.58xxxx (out to about 10 digits). If you multiply 3.58 by 3 and divide by 2 you get our clock frequency of about 5.4MHz. By the way if you multiply the 3.58 by 4 and divide by 3 you get 4.77MHz which is the clock rate of the original IBM PC (originally it was going to drive a TV as well but they changed their mind before going to market).

If you run a US TV in non interlaced mode (video games ran non-interlaced to avoid flicker) you will find that 5.4MHz will give you about 256 visible pixels per line. It also turns out that the pixels will be "square." Some companies (I think Commodore) in this era had 320 pixel per line by going to 2X 3.58 = 7.18MHz but the pixels will end up non-square.

We relied on the Home Computer group for our information on NTSC. It turns out that at 5.4MHz and 256 pixels per line while the pixels are all technically "visible," most TV makers had about 10% overscan (done to give a wide tolerance to make sure the picture fills the screen) which in turn cut off about 8 pixels in each side. The Zenith built TV based monitor was "tweaked" to not have the usual TV overscan.

At the time running at 5.4MHz was a huge challenge and unlike CPUs that could have different speed grades, it was 5.4MHz or nothing for the 9918.

Perhaps the biggest limitation was not memory storage but bandwidth into the DRAM. With 16K bytes of memory while we had enough storage to do a full bit-map but there was not enough bandwidth. This is the reason for the much maligned graphics mode 2. We wanted to put in full bitmap, but there was just not enough bandwidth to do 4 bits per pixel over an 8 bit bus. If you consider it took us 2 cycles per DRAM access we would have just enough time to read 8 bits/2 pixels but we would have to lock out all the CPU writes except during horizontal blanking and we would not have any time to do Sprite "pre processing" (described more below). So you would have had a 4-bit per pixel bitmap with no sprites and it would be very slow to change (because the CPU could only make about 1 access per line during the horizontal retrace.

By the way, the limitation of 4 sprites per line was also a result of bandwidth. We didn't have enough DRAM accesses to support a 5th Sprite without locking out the CPU for too long.

FYI, the basic processing scheme was that during the Horizontal retrace/blanking we would go though the list of 32 sprites and see which of 4 would be visible on a given line and as well as the 5th sprite and then fetch up to 4 sprite line segments that that would be displayed on a given line. During the active display, we fetched the background information and gave accesses to the CPU.

I remember one feature we debated adding was the scrolling background but did not do it because of the complexity of implementing it (it would also require one additional memory access).

Q: Are you aware of any hardware bugs, undocumented behaviors or glitches in the 9918A that made it to production? (My thinking being, maybe there's something we can

try to take advantage of. Glitches provide the most interesting opportunities to make hardware do new things. They only figured out how to glitch the Atari 2600 video circuitry for reliable interlaced video a few years ago, for example.)

A: I can't think of anything off the top of my head.

Q: Do you remember offhand whether any data from VRAM is cached in the chip? For example, we have docs from Paul Urbanus which note that the first four active sprites on a line are cached, is anything else cached or is it always fetched as needed?

A: Pretty much everything if fetch "just in time."

During the active display a memory cycle was as long as 2 pixels. The graphics modes required a NAME fetch, a Pattern fetch, and a Color Fetch or a total of 3 Fetches that would give 8 pixels of background and leave 1 fetch every 8 pixels for either a CPU access times) or **sprite preprocessing** access. During the active display there were 256 pixels, and 128 memory cycles. Of these cycles 3/4th we used by the background leaving a total of 32 to be shared between the CPU and the sprite preprocessing. I think we gave a few of these 32 to the CPU and had a few preprocessing accesses to do during blacking to get to 32 sprites.

During the active display we did **sprite preprocessing** where we went though the 32 sprites looking ONLY at the line number and the size (which was the same for all sprites because we didn't have the memory cycles to fetch the information). These sprite preprocessing access were done every 4th memory cycle (with some of these given to the CPU). This would determine if the sprite was active on the next line. If a Sprite was "active" we would save the sprite number and on a first in first out (FIFO) stack which was 5 entries long (4 active and the 5th sprite number.

When we hit blanking, we use the sprite number stack to go back and fetch the 4 bytes information for each sprite. We then used the size and starting line of the sprite and current line of the display to compute which the address of the line of the sprite.

Q: Can video memory be externally accessed during a scanline or do access periods occur only during hblank and vblank?

A: As stated above, we gave read and write access to the CPU during the active display. I would have to go back and look at the design note which I have boxed up some place, but I think we gave the CPU an access about 1 out of 16 cycle (with 12 of the 16 going to the background and 3 of the remaining 4 going to sprite preprocessing). The CPU also got a few accessed during horizontal blacking between the sprite processing and it of course got a lot of accesses during vertical blanking.

Q: Personally, I think multicolor mode (64x48) is great, and reminds me of the Apple2 lo-res mode. We know the Apple lo-res mode is there to play Breakout, similarly can you discuss why the 9918 has multicolor mode? (Is there a story there?)

A: We had the luxury of DRAM and thus a lot of storage relative to previous graphics chips. Unfortunately, as stated above we did not have enough bandwidth to support full bit map graphics. So the other modes were a way to give a compromise between resolution and numbers of colors.

Q: Are you aware of any systems that used the 9918's genlock/external video functionality? Were there any "gotcha"s to using it?

A: I know lots of people tried and I think there were a few products but. This was a much desired feature and was a pet project of Pete Macourek. When Pete left TI not long after the 9918, I took over supporting it in my "spare time."

The big problem originally was the relationship between our pixel clock and the 3.58MHz and the fact that they were tied together on the 9918. With the 9918, you could either LOCK to the dot clock OR lock to the color clock but not both at the same time. So you either got the colors flashing if you locked on the pixel, or you

got wiggly pixels if you locked on the color. With the 9928 and 9929 you could get it to work and several people did because the color burst was separate.

Q: Is there a story behind where the idea of sprites came from and what it took to implement them in the 9918?

A: Atari and some earlier graphics chips had hardware player graphics. The idea with sprites was to give more of these players. A big difference with the 9918 is rather than have a dedicated set of hardware for each sprite, we had sprite pre- and post-processing with hardware that was shared with all the sprites.

Pete and I worked out the sprite processing. I'm pretty sure I'm the one that came up with the "sprite pre-processing stack" but I couldn't swear to it.

Q: Why was the 9918 designed with an 8-bit bus to the host system vs. a 16-bit bus? (It seems that TI had already moved on to 16-bit systems by that time.)

A: We were really thinking in terms of the 8-bit CPUs that were used for games. Also remember that the home computer was going to originally use a 9985 that was going to have an 8-bit memory interface. Back then, 16-bit busses were expensive to support. Note that even IBM when with a 8-bit interfaced 8088 for their first PC.

Q: Why don't you see features of the old VDP's in new modern designs? (It seems that no matter how fast a system becomes, off-loading work to co-processors can always help.)

A: Basically everything went to Bit Mapped graphics with a thing called BitBlts. These are bit/pixel aligned transfers. In 1984 after working on the 9995 and 99000 I got back into computer graphics with the TMS340 family which included the 34010 and 34020. At first we called BitBlt'ing "bit mapped sprites" or "software sprites" as in may ways they behave like sprites on top of a background. During the design of the 34010 we learned about the work of Zerox Parc and they called them "bitblts" because they were black and white but because we were doing color pixels, we call them PixBlts.

The problem with hardware sprites is that you are always limited by the amount of hardware for processing each sprite. With BitBLIs you are only limited by the processor speed and your access rate to memory. You could also be totally flexible in terms of size and shape of the "sprites."

Almost all the hardware accelerators support color bitblt's so in a way they all are supporting software sprites in hardware. The last vestige of hardware sprites is cursors and I guess for compatibility reasons the graphics accelerator chips still support hardware cursors.

Q: If you could change anything about the 9918, what would it be?

A: You have to look at these kinds of questions in the historical context. We had started on AVDP that was going to add more sprites and bitmapped graphics and even got a chip out, but TI was behind in having CMOS process and the design used a very trick NMOS one that had a lot of bugs and they cancelled the program. At the same time Yamaha did a register level clone and superset that was similar to the AVDP. Yamaha was apparently doing the chip for Nintendo and being in Japan they had the inside track on that design (I don't know why TI never sued Yamaha based on our sprite patent, maybe we had a cross license, I don't know). I'm that their chip was register level compatible with the 9918 and suspect that Nintendo, which developed Donkey Kong for Colecovision, use the 9918 in there early game system development.

There certainly were some things we could have implemented more cleanly. We certainly wished we had put hardware background scrolling in the 9918. We wanted more sprites but we didn't have the memory bandwidth to support them (we used just about every memory cycle we had).

We were pushing the DRAM bandwidth as much as was practical at the time. We were pretty advanced for our day when you consider we had first silicon in mid 1978 and

the big volumes for the chip came in 1981 and beyond. In fact they though of cancelling the chip several times before the 99/4A and Colocovision happened.

I guess we could have gone with a 16-bit interface to the DRAM in order to support true bit maps, but that would have seemed ridiculously expensive when we started the 9918 in 1977. We were pretty walled in by the technology of the day. We could certainly see we wanted a simple bitmap but we couldn't see a way to get there cost effectively.

If things had taken off a bit sooner, we would have probably done the AVDP earlier fixing all the obvious things like more sprites and a full bitmap background mode. But so much time passed between finishing the 9918 (and it small variation of the 9118/9128/9129 that supported the by-4 DRAMS) that is was like starting over. None of the original designers worked on the AVDP and I had moved on to work on the 340 Family of Bitmapped graphics processors.

Future

Myself and a few other 4A die-hards are kicking around the idea of a 4A "upgrade". Basically we want to enhance the system to include some things we had always wished for. For me that is more colors per pixel, some hardware scrolling, and maybe some internal registers for the CPU (and other things, but those are the biggies). However, one of the primary goals is to make this an upgrade, not a totally new computer. Some backward compatibility is important, as is making sure we don't go overboard and destroy what I call the "magic" we all get from these classic computers.

Some of us have written emulators (and some are still working on theirs), and we would probably start with modifications to the emulators, and eventually I hope to help put the 4A on an FPGA as well.

Q: Do you have any words of wisdom for us on our adventure, or things you personally might like to see in the "upgrade"?

A: Man, I wish you luck. I will tell you what I told Joe Z. on his Intellivision hobby, "You have to put so much creativity and work into that project, its too bad you don't do something that could end up in a real product that would make you a lot of money." That said, I would be happy to help if you have the occasional question.

I love to study history and I try not to fault people and their hobbies (I collect Disney trains, I have a number of 1960's era Schuco Monorails, and enjoy Disney history as my hobby). But when it comes to doing "real work," I like to do things that have a chance of changing the world in some way.

BTW, I'm currently the CTO of a startup working on making an LCOS projector that is so small it will fit in a cell phone in a couple of year. Other than my work for about 4 years on 9995 and 99000, all my work for the last 30 years has in one way or another been manipulating pixels.

karl@kagutech.com wrote:
> Thank you for the kind words. I would be happy to try and answer your
> questions. A few things below you might find interesting came to mind with
> your email.
>
> The 9918 was the first chip I worked on at TI and so it has a special place
> in my heart. After the 9918 and after the 9918 was complete, just a year
> and a half out of school, I became the chief architect of the 9995
> microprocessor that was going to be using in the 99/2 and the 99/8 but they
> cancel the family before those products went to production. So I almost
> designed both the graphics chip and the CPU of the family.
>
> Related to your hobby, at TI I hired another person Joe Zbiciak who's first
> computer was a 99/4A when he was about 8 years old. Joe is extremely bright
> and his hobby is making software games for Mattel's Intellivision emulators
> (see: http://spatula-city.org/~im14u2c/intv/); he also had one of those as a

> kid. Mattel was interested in the 9918 for Intellivision, but we couldn't
> sell it to them because the Home Computer Group was suppose to need so many
> 9918 of them for the 99/4 (pre A), but as it turned out the Home Computer
> was delayed and Mattel went with an older less capable chip.
>
> We were about the cancel the whole 9918 program when Coleco came in with
> Colecovision and the TI Home Computer took off while at the same time the
> MSX computer in Japan (the "MS" standing for Microsoft which had a short
> lived joint venture in Japan) started selling big time using the 9918.
>
> Not many people know this but Nintendo's original game system was based on a
> register level compatible superset clone of the 9918 done by Yamaha.
> Nintendo had developed games for the Colecovision, including Donkey Kong,
> and they moved this to their new game system.
>
> In the early 1980's, I was a big fan of the show "Connections" by James
> Burke was a great program that showed (albeit loosely sometimes) how there
> are a series of connections between inventions and often times the person
> that becomes famous may have just been the one person that added one key
> improvement on top of a bunch of other ideas/inventions. I am a bit of a
> computer history buff (but less so as I go married and had children).
>
> Regards,
> Karl
>
> -----Original Message-----
> From: Matthew Hagerty [mailto:matthew@digitalstratum.com]
> Sent: Wednesday, January 16, 2008 3:08 PM
> To: karl@kagutech.com
> Subject: The 99/4A and 9918A.
>
> Greetings Mr. Guttag,
>
> First I'd like to say thank you for your contributions to the computer
> world, and specifically for the chips that make the 99/4A what it is.
>
> I'd like to introduce myself, my name is Matthew Hagerty and I've been a
> 99/4A user since 1983 when my Dad bought me one as my first computer.
> Once I managed to get the PEB, I learned assembly language and quickly
> became fascinated with the low level workings of the computer. That was
> all way back then, and now 25 or so years later, thanks to the Internet,
> much more information is available and programming my 99/4A is still
> something I enjoy as a hobby.
>
> Thanks to a post someone made on the Yahoo! TI forum, I found the
> transcript of an interview you gave in 1993 for the TI International
> Users Network. The information you provided was absolutely fantastic
> and cleared up many nagging questions and assumptions that have been
> made for years by the small remaining TI community. I was sad however,
> because I would have loved to have been able to participate in asking
> some questions. If you are not too busy, and if you would not mind, I
> was wondering if you would be willing to do one more interview? Nothing
> real-time or in person (although it would be great to have a real chat
> with you sometime), just simply if I could send you some questions via
> email that you could answer at your convenience? If not, that is okay
> too, I'm just glad I could say thank you for your contributions and hard
> work, the TI changed my childhood and my life.
>
> Thank you,
> Matthew
>
> ¸.·´¯`·.´¯`·.¸¸.·´¯`·.¸><(((º>
> .¸¸.·´¯`·.¸><(((º>
>