Magic-1 Memory Subsystem Redesign

The purpose of this document is to serve as a sounding board and design document for the redesign of Magic-1's memory subsystem. The reason for the change is that the SRAM to Memory Data Register (MDR) read path is the current speed-limiting path for Magic-1, and the redesign should dramatically shorten this path by splitting it across two halves of a clock cycle.

First, a quick overview of Magic-1's clocking, and some history about how the current inferior design came about.

Clocking

M-1 originally had a single clock, CLKS, with a rather simplistic notion of synchronization. On the falling edge of CLKS, the microcode control bits would flow through the system – enabling register operands and selecting the ALU operation. On the rising edge of CLKS, the ALU operation's result would be clocked into the target register. During the high period of CLKS, the next microinstruction to be executed would be fetched from the microcode store. On the falling edge of CLKS, those microcode control bits would again be released and this process would repeat.

Now, let's add memory accesses to this picture. For a memory read, the process is the same as for enabling register operands. Microcode bits signaling a read would flow out into the system on the falling edge of CLKS. Among those bits would be memory read enable signals. Besides knowing that we are doing a read operation, we also need to know the address to read from. This is computed by a combination of the virtual address contained in the Memory Address Register (MAR) and the process's page mapping stored in the Page Table SRAM. In all, the following signals are required to compute the physical read address to be placed on the address bus:

- Contents of MAR (16 bits)
- Contents of the Page Table Base register (16 bits)
- Mode bit in the Machine Status Word (1 bit)
- CODE_PTB microcode signal (1 bit)
- USER_PTB microcode signal (1 bit)

To successfully do the memory read, we need to translate the virtual address (MAR value) into the complete physical address (which will be placed on the 22-bit Address Bus), plus drive the signal that selects between SRAM and Device physical address spaces. Once that complete address is generated, the SRAM section must decode that address to enable the proper SRAM chip. The SRAM chip must then be given enough time to return the proper byte, which flows onto the Data Bus. And we're still not done. The bits must then flow through a bi-directional bus driver and a layer of multiplexers before arriving at the input pins of the Memory Data Register (MDR) with enough time to establish data setup prior to the rising edge of CLKS (which will clock the data into the MDR).

Critical Paths

Here's the longest read path, starting with the signal that enables the new microcode control signals to flow on the falling edge of CLKS:

Device	Signal/Description	LS delay	F delay
74x273 (Ctl U10)	USER_PTB microcode bit	20	7
74x02 (Mem U1A)	NOR USER_PTB, Mode bit	7	3.4
74x534 (Mem U17)	Page table base register enable	20	4.5
Fast SRAM (M U7)	18ns Page Table SRAM	18	18
74x10 (Mem U10A)	AND Mem/Dev & MSW paging enable	10	4.1
74x244 (Mem U2)	Bus driver for Dev/Mem space signal	12	4.5
74x138 (Mem U28)	Decoder for main SRAM	22	6
Big SRAM (Mem)	Any of 512K x 8 SRAM chips	70	70
74x245 (Ctl U61)	Bi-directional driver for MDR	17	5.5
74x157 (Ctl U52)	Mux for low byte of MDR	20	6.5
74x273 (Ctl U53)	Low byte of MDR setup time	20	4
Total Prop Delay:		**236**	**133.5**

This entire path must be accomplished in a half-cycle of the clock. At 3 Mhz, that means 166 nanoseconds, and at 4 Mhz, we only get 125 nanoseconds. If I went to all F parts on this path, the best I could do would be about 3.75 Mhz (though I believe I would run into noise problems with that many F parts).

The good news is that I don't really *have* to do all this in a half-cycle of the clock. The thing I didn't consider when I designed the memory subsystem the first time was that all of the signals that are used to translate the virtual address into the physical address could be made available a half-cycle early. In other words, we can split this path into two parts, and thus be limited only by the larger of the two. Here's what it will look like:

Device	Signal/Description	LS delay	F delay
74x273 (Ctl U10)	USER_PTB microcode bit	20	7
74x02 (Mem U1A)	NOR USER_PTB, Mode bit	7	3.4
74x534 (Mem U17)	Page table base register enable	20	4.5
Fast SRAM (M U7)	18ns Page Table SRAM	18	18
74x10 (Mem U10A)	AND Mem/Dev & MSW paging enable	10	4.1
Total Prop Delay:		**75**	**37**

Device	Signal/Description	LS delay	F delay
74x244 (Mem U2)	Bus driver for Dev/Mem space signal	12	4.5
74x138 (Mem U28)	Decoder for main SRAM	22	6
Big SRAM (Mem)	Any of 512K x 8 SRAM chips	70	70
74x245 (Ctl U61)	Bi-directional driver for MDR	17	5.5
74x157 (Ctl U52)	Mux for low byte of MDR	20	6.5
74x273 (Ctl U53)	Low byte of MDR setup time	20	4
Total Prop Delay:		**161**	**96.5**

By splitting this path, I could move my theoretical clock speed ceiling all the way to 5 Mhz. Further, I would also eliminate the need to use exotic 18ns cache memory for the page table. I could get away with using vanilla and easy-to-find 70ns 32Kx8 SRAMs.

Of course, making this change doesn't get me all the way to 5 Mhz – I believe my next speed path would be uncovered in instruction decode and microcode fetching before I got there. That path determines whether to take a microcode branch (starting with rising edge of CLKS):

- MSW S bit (ALU/Reg card, U26, 74x273: 20/7 ns)
- XOR w/ V bit (CTL card, U12B, 74x86: 20/6 ns)
- OR w/ Z bit (CTL card, U13B, 74x32: 10/4.2 ns)
- MUX (CTL card, U11, 74x151: 46/8 ns)
- XOR w/ NEGATE_BR (CTL card, U12A, 74x86: 20/6 ns)
- OR w/ DO_BRANCH (CTL card, U13A, 74x32: 10/4.2 ns)
- AND (CTL card, U44B, 74x11: 11/4.1 ns)
- MUX (CTL card, U18, 74x153: 30/7 ns)
- Microcode store (CTL card, U1-U4, 55ns EPROM or 45ns PROM, 55:55ns)
- Microcode gate setup (CTL Card, U6-U10, 74x273: 20/4 ns)

This adds up to 242ns with all LS parts, and 105.5 with all F parts. The big ticket items are the MUXes, and by making those F (which I already did) along with a few of the glue logic bits I shoudl be able to make the 125ns 4Mhz goal – but that's pushing it. 4 Mhz may be obtainable, but not much more. My other areas for potential speed paths are the register -> aluop -> Z-bit path and some of the more complicated microcode field decodes. A cursory examination suggests that neither of these will be a problem.

So, how do I make memory unit change? The first part – getting the MAR contents early – is already done. The MAR is treated as a normal register, and thus obtains new values on the rising edge of CLKS. Similarly, the page table base register (PTB) and mode bit (part of the MSW) are also set on CLKS rising. However, currently the microcode bits for USER_PTB and CODE_PTB are associated with the microinstruction that actually does the memory operation, and thus don't appear in the system until the falling edge of CLKS.

The first part of the solution is simply to rewrite the microcode to require that these bits be asserted 1 microinstruction prior to the memory reference. This is actually not a big deal, as this same rule already applies to the MAR. There is, however, a difference. The MAR is a register, and thus can be set any number of cycles prior to use. The microcode bits are ephemeral – and are set only when explicitly asserted. What needs to happen here is we will treat the setting of a new MAR register value as a general signal to capture the complete state of a virtual address. So, we not only latch in a new MAR value from the L bus, but we also take a snapshot of USER_PTB, CODE_PTB and (perhaps?) the mode bit from the MSW [need to think about that one. Don't really have to – but it might make mode transitions easier – examine the trap/reti microcode].

Now, assume that on the rising edge of CLKS when we set a new MAR value we also capture the rest of the bits needed for virtual to physical address translation. An issue here is that we cannot immediately start changing the current contents of the address bus. While memory reads happen on the rising edge of CLKS, memory writes complete on the falling edge of CLKS. Thus, the contents of the address bus must remain stable until that falling edge. So, what we do is have a set of registers that holds the contents of the address bus. This register will be clocked to take a new value on the falling edge of CLKS, but behind these registers the virtual to physical address translation will be taking place during the high period of CLKS.

As far as device counts go, I should end up with about the same number of parts. Right now there is a row of bus drivers that drive the address bus. Given them small number of devices connected to the bus, I expect I can get away with eliminating these drivers and just using the output of the address bus gating register to drive the address bus (but will wire things up so that I could easily add drivers if needed).