816 BEAT

Reprinted from Commodore World Issue 19

By Doug Cotton

CMD SUPERCPU RAM EXPANSION & TIMING

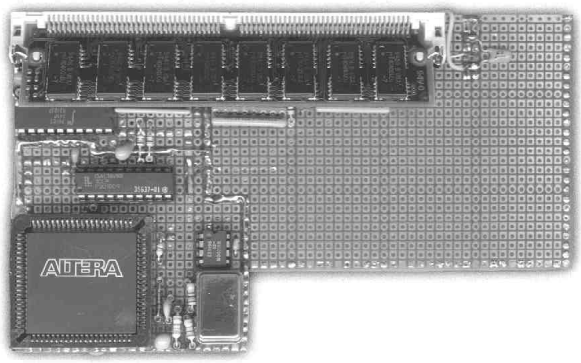
One of the more anticipated releases for the SuperCPU is just around the corner. I'm speaking of the SuperRAM card for the SuperCPU 64, long awaited by many of the developers involved in creating new programs for SuperCPU-enhanced systems. The card will allow larger programs or programs with extensive data to be fully loaded into memory (as opposed to bringing in separate modules from disk, a process that is both slow and inconvenient). New programs, written with the SuperCPU and SuperRAM card in mind, can offer more powerful features. But before we get into technical details, let's take a more general look at the SuperRAM card itself.

Pictured on the right side of this page is the prototype SuperRAM card which CMD has developed for testing. The board contains only a few components: a clock oscillator, bus driver, a reprogrammable array logic device (GAL), a digital delay chip, a complex programmable logic device (CPLD), and a handful of resistors and capacitors. There are also two connectors on the back of the circuit board (not shown) which attach the SuperRAM card to the SuperCPU main board, and a 72-pin SIMM (Single Inline Memory Module) socket where the RAM SIMM is installed.

As with the SuperCPU itself, the SuperRAM card's complex circuitry is mostly inside the CPLD, which contains most of the memory mapping, control and refresh circuitry. While this vastly decreases the amount of board space required, developing the logic equations needed to program the chip for a specific function often proves to be very time-consuming.

SuperCPU 64
RAM Expansion Card Prototype

Slated for release in the next few weeks is the RAM Expansion Card for the CMD SuperCPU 64. This card can contain from 1 to 16 Megabytes of RAM (using standard 72-pin SIMMs) that can be used by future applications. A GEOS driver is expected to ship with the card.



The SIMM socket can be fitted with 72-pin memory modules containing from one to 16 Megabytes of standard Fast Page DRAM. It's very important to make sure that the SIMM used is standard Fast Page; EDO and other 72-pin SIMM types are not compatible, and will not operate correctly. The memory must be rated at 70 ns or faster (the lower the number, the faster the speed rating), but bear in mind that faster RAM doesn't translate into faster access (the DRAM controller has fixed speeds for performing memory access).

For additional information on compatible SIMMs, see the SIMM Chart included with this article. The chart fully specifies all SIMMs approved for use with the SuperRAM card.

The General Memory Map

Since the 65816 processor can address up to 16 Megabytes of RAM, the SuperRAM memory is unlike previous RAM expanders (such as the Commodore 17xx series REU's) in that programs can actually execute directly from this memory. It's also important to note that programs don't have to use the 65816's native mode to be able to access this extra RAM, although there are some advantages to doing so. The program SUPERRAMFAKE, which accompanies this article, contains a subroutine that shows how extra memory can be accessed in 6502 emulation via "long" addressing modes. We'll discuss that more a little later, but we should first look at how the SuperRAM card fits into the SuperCPU memory scheme.

For a good overview, take a look at the "SuperCPU 64/128 Common Memory Map". The areas in white (Banks $00, $01 and $F8-FF) are the memory found in every SuperCPU, with or without memory expansion. Banks $00 and $01 are static RAM, while $F8-FF are used and reserved banks for the system ROM. This map is identical on both the 64 and 128 versions of the SuperCPU, though the 128 version will have two additional banks of static RAM which will be swapped in at Banks $00 and $01 as needed.

Given the general map, there is room for expansion RAM at Banks $02 through $F7. To avoid the need to translate addresses on all expansion RAM, SIMM memory addressing actually begins at Bank $00,

SIMM Chart

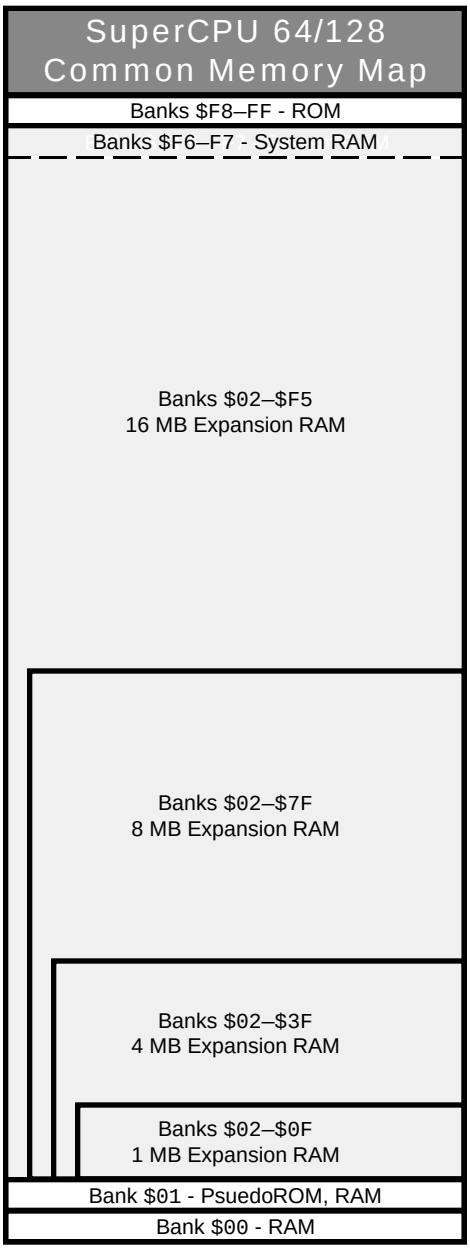
The chart below indicates the size and organization of 72-pin SIMMs supported by the SuperRAM card for the SuperCPU. All SIMMs must be Fast Page Mode type, 70ns or faster.

SIMM Capacity	Row Size	Row/Column Adr.
1 MB (256K x 32/36)	2 KB	9/9
4 MB (1M x 32/36)	4 KB	10/10
8 MB (2M x 32/36)	4 KB	11/10
16 MB (4M x 32/36)	4 KB	12/10
16 MB (4M x 32/36)	8 KB	11/11

although any expansion RAM that occupies the same address area as Static RAM (Banks $00 and $01) or ROM ($F8-FF) isn't accessible. The SIMM RAM Banks $00 and $01 have been moved to Banks $F6 and $F7, and reserved for future system use. This then means that any system with expansion RAM (any size) will have this extra RAM available for future OS capabilities, but it also means that the last Bank available for user programs will be Bank $F5 (on a system with 16 Megabytes of expansion RAM installed).

Detecting Expansion RAM

Naturally, one of the more important questions on programmer's minds is, "How do I detect RAM expansion, and how do I know how much there is?" Okay, that's two questions, but we've got answers for both.



First, any new SuperCPU sold after the new SuperRAM card becomes available will sport a new version of the SuperCPU ROM. Likewise, all SuperRAM upgrades sold to users with older SuperCPUs will include the new ROM. Thus, the first step in determining if RAM expansion is present will be to check the ROM version. This is accomplished by reading four bytes, beginning at $00E487 (64 mode only!). These four bytes contain the version number string in PETASCII. The version that will provide initial support for expansion RAM is "1.40". Read these bytes and compare for this number. If the version number is lower, there's no expansion; if it's the same or higher, there may be expanded memory, and you'll need to move on to the next step.

If you found a version that supports RAM expansion, read in four bytes beginning at $00D27C. These bytes contain the following information concerning RAM expansion:

$00D27C First Available Page
$00D27D Bank of First Available Page
$00D27E Last Available Page+1
$00D27F Bank of Last Available Page+1

If there isn't any extra RAM installed, all four bytes will contain zeroes. The BASIC program SUPERRAMDETECT provides an example of checking these parameters and calculating the available expansion memory. Please note that these variables are only valid in Bank $00 while I/O is switched in; should you need to check for expansion RAM with I/O out, these values are available in the same locations of Bank $01.

If your application needs to use some portion of expansion RAM, it must also update the memory variables. This requires switching in the SuperCPU H/W registers by storing any value at $00D07E (decimal 53374). After you have modified the variables, turn the SuperCPU H/W registers back off by storing any value to $00D07F (decimal 53375). Again, I/O must be enabled during any of these changes, or you'll need to change the variables directly in Bank $01.

It is very important that you pay attention to the expansion RAM variables, and that you don't make any assumptions with regard to RAM availability; some future system extensions or user programs may steal some of the RAM before your application is started. As a result, it would be wise to create your program code and/or data segments in a manner that allows them to be relocated. CMD is presently working toward standards and tools that will make writing and utilizing relocatable code less painful, but it will make the transition easier if 6502/65816 programmers start getting familiar with the techniques now.

To assist you in testing routines that detect RAM expansion, we've included the program SUPERRAMFAKE with this article. You may use this program to trick your SuperCPU into believing that it has RAM expansion available, as well as the proper OS version required to support it.

Speed Considerations

As you may already know, Dynamic RAM (DRAM) isn't as fast as Static RAM (SRAM), but it is far less expensive and available in larger capacities. This explains why DRAM was chosen for expansion memory.

Taking the speed into consideration, CMD employed special circuitry into the SuperRAM card's DRAM controller to help the DRAM keep up. Understanding how this controller 'thinks' is the key to optimizing the speed of expansion RAM accesses on the SuperCPU.

DRAM, unlike SRAM, must be pre-charged before valid data can be read from a specific address. DRAM also requires periodic 'refresh' in order to maintain its contents. These are the factors that add time to accessing the memory. The memory cells themselves in Dynamic memories are organized into an array of rows and columns. On memory modules such as the SIMMs used by the SuperRAM card, these rows and columns are combined in a way that allows

Expansion RAM Speed Characteristics at 20 MHz*

Sequential Read within Row[1]:	1 Cycle
Non-seq. Read within Column[2]:	1 Cycle
Non-seq. Read, new Column[2] in Row[1]:	2 Cycles
Read from new Row[1]:	3.5 Cycles
Write within Row[1]:	1 Cycle
Write in new Row[1]:	3 Cycles
Read during Refresh[3]:	up to 8.5 Cycles
Write during Refresh[3]:	up to 8 Cycles

[1]Rows are 2K, 4K or 8K Bytes, depending on the SIMM (see SIMM Chart).
[2]Columns are groups of four bytes each on supported 72-pin SIMMs (see SIMM Chart).
[3]Refresh occurs approximately every 10 microseconds.

*At 1 MHz all times are 1 cycle (synchronized to the computer's Phase 2 clock), refresh is hidden.

all the bits in a byte or a word to be precharged and accessed as a group.

If you look at the SIMM Chart in this article, you'll notice that we included the number of addressable bits for rows and columns, as well as the number of bytes within a specific row. Let's look at the 1 MB SIMM to understand how this information describes the SIMM.

There are 9 bits used to address rows, and another 9 bits for columns. Since 2^9=512, we can deduce that there are 512*512 array crosspoints, which gives us 262,144 unique addresses. Divide that by 1024 (1K), and you'll get 256... so there are 256K addresses on this SIMM. Since each address has 32 bits of data (or 36 on a parity SIMM), there are 4*256K, or one Megabyte (1,048,576 Bytes) of 8- or 9-bit memory locations.

Still with me? Okay, we can also deduce from the SIMM is that each row contains 2 KB ($800 bytes) of 8- or 9-bit data, since there are 512 columns of 4 bytes each per row.

Now let's look at how the SuperRAM memory controller handles things. Assume for a moment that you have a routine situated at $020000, the first available expansion memory location. This location is the first byte in a row ($020000/$800=$40 with no remainder), and also the first byte of the first column of that row (always the case at the start of any new row, though we can do the math $020000/$04=$8000 with no remainder). At this location you have the following code:

```
020000 A9 03        LDA #$03
020002 8F 00 03 02  STA $020300
```

Let's assume you jump to this code from another Bank or row, and it begins executing. Normally the LDA immediate would require 2 cycles to complete; 1 cycle to load the instruction, and 1 cycle to fetch the immediate

SUPERRAMDETECT

```
112  5 rem get version
243  10 v$=""
118  20 fori=585Ø3to585Ø6
43   3Ø : v$=v$+chr$(peek(i))
17Ø  4Ø next
136  5Ø v=val(v$)
118  6Ø :
16Ø  7Ø ifv<1.4Øthen2ØØ
138  8Ø :
32   1ØØ rem get ram size & location
7Ø   11Ø sp=peek(53884) : rem start page
89   12Ø sb=peek(53885) : rem start bank
153  13Ø ep=peek(53886) : rem end page
144  14Ø eb=peek(53887) : rem end bank
54   142 ifsb+sp=Øthen2ØØ
2Ø3  145 :
174  15Ø x=(eb*256+ep)-(sb*256+sp)
218  16Ø :
178  17Ø printx*256"bytes available"
23   18Ø print" starting at"(sb*256+sp)*256
63   19Ø end
25Ø  192 :
43   2ØØ print"no ram expansion"
83   21Ø end
```

SUPERRAMFAKE

```
2Ø1  1Ø print"{CLEAR/HOME}{CRSR DN}{14 SPACES
     }superramfake"
86   2Ø v$="1.4Ø":sp=Ø:sb=Ø:ep=Ø:eb=Ø
144  3Ø h$="Ø123456789abcdef"
148  9Ø :
2Ø   1ØØ print"{HOME}{3 CRSR DN}{15 SPACES}1.
     {2 SPACES}Ø mb"
117  11Ø print"{15 SPACES}2.{2 SPACES}1 mb"
147  12Ø print"{15 SPACES}3.{2 SPACES}4 mb"
38   13Ø print"{15 SPACES}4.{2 SPACES}8 mb"
78   14Ø print"{15 SPACES}5. 16 mb"
1Ø3  15Ø print"{15 SPACES}6. custom"
218  16Ø :
187  17Ø getk$:ifk$=""then17Ø
96   18Ø k=asc(k$+chr$(Ø))-48
92   19Ø ifk<1ork>6thenk$="":goto17Ø
3    2ØØ :
126  21Ø onkgoto3ØØ,31Ø,32Ø,33Ø,34Ø,35Ø
84   3ØØ sp=Ø:sb=Ø:ep=Ø:eb=Ø:goto5ØØ
174  31Ø sp=Ø:sb=2:ep=Ø:eb=16:goto5ØØ
11Ø  32Ø sp=Ø:sb=2:ep=Ø:eb=64:goto5ØØ
2Ø   33Ø sp=Ø:sb=2:ep=Ø:eb=128:goto5ØØ
243  34Ø sp=Ø:sb=2:ep=Ø:eb=246:goto5ØØ
```

SUPERRAMFAKE *(cont.)*

```
146  35Ø gosub4ØØ:goto5ØØ
163  36Ø :
126  4ØØ rem input custom values
213  41Ø :
91   42Ø print"{2 CRSR DN}enter values in hex
     !":print"note: end address is last addre
     ss+1{CRSR DN}"
54   43Ø input"starting bank (sb)";ui$:gosub4
     6Ø:sb=ui
24Ø  431 input"starting page (sp)";ui$:gosub4
     6Ø:sp=ui
1Ø1  432 input"ending bank{3 SPACES}(eb)";ui$
     :gosub46Ø:eb=ui
35   433 input"ending page{3 SPACES}(ep)";ui$
     :gosub46Ø:ep=ui
68   44Ø return
253  45Ø :
14Ø  46Ø ui=Ø
132  462 fori=1to16
141  464 : ifleft$(ui$,1)=mid$(h$,i,1)thenui=
     ui+((i-1)*16)
59   466 : ifright$(ui$,1)=mid$(h$,i,1)thenui
     =ui+(i-1)
9Ø   47Ø next
1Ø8  48Ø return
38   49Ø :
194  5ØØ rem store dummy values
45   5Ø1 :
11Ø  5Ø4 pl=124Ø39:fori=1to4:pv=asc(mid$(v$,i
     ,1)):gosub518:pl=pl+1:next
49   5Ø5 :
44   5Ø6 pl=119424Ø:pv=sp:gosub518
225  5Ø7 pl=pl+1:pv=sb:gosub518
222  5Ø8 pl=pl+1:pv=ep:gosub518
11   5Ø9 pl=pl+1:pv=eb:gosub518
58   51Ø :
211  516 sys64738
61   517 :
131  518 b=int(pl/65536):h=int((pl-(b*65536))
     /256):l=pl-((b*65536)+(h*256))
67   519 :
1Ø4  52Ø poke49152,169 : rem lda#
1ØØ  521 poke49153,pv  : rem value to store
23Ø  522 poke49154,143 : rem sta abs long
36   523 poke49155,l   : rem lo addr
168  524 poke49156,h   : rem hi addr
187  525 poke49157,b   : rem bank
23Ø  526 poke49158,96  : rem rts
75   527 :
188  528 sys49152
162  53Ø return
```

byte into the accumulator. But in this case it would require 4.5 cycles; 3.5 cycles to fetch the instruction from a new row in expansion RAM, then 1 more cycle to fetch the immediate byte. The latter took only 1 cycle because the row and column were already charged, and the controller knows this. If you're wondering how an operation can take an uneven number of cycles, you need to take a look at the sidebar on Clock Stretching.

Now the next instruction, STA, is fetched in 1 cycle, and the three address bytes are all fetched at 1 cycle for each. When the second byte of the address (the $03 at $020004) is fetched, we cross over into a new column. Normally this would require an extra cycle, due to column address access timing requirements; however, the controller outputs the next column address when the processor reads from $020003 by assuming that the next access will be in the following memory location. By always 'guessing' that the next access will be sequential, the DRAM controller saves time when this proves to be the case.

Back to the example, there's one operation left to perform: store the accumulator to memory. This usually takes 1 cycle, but the location where it is to be stored is in a distant column of the same row, so it takes 2 cycles.

This set of instructions would normally take 7 cycles in SRAM, but in expansion RAM, it requires 10.5 cycles. This may seem slow at first, but when contrasted with a stock 64 running at 1 MHz, we're still operating over 13 times faster (the throughput is approximately 13.4 MHz in this particular case). We could also modify the program so that the store instruction places the value into Static RAM instead of Dynamic, say at $003000, and save an additional cycle. This would kick the effective speed up to 14.7 MHz.

It's also important to note that most of our loss in throughput came when our code began executing in a new row. This doesn't happen often, since rows are at least 2 KB wide. Consider a whole 2 KB segment of contiguous code executing from DRAM, with all external reads and writes going to Static RAM. Under those circumstances you might achieve a throughput of over 19.9 MHz—not considering refresh or occasional jumps and branches.

Refresh? Yes, DRAM needs to be refreshed to maintain its contents, and at these speeds, it can no longer be 'hidden' as it commonly is at 1 MHz. Refresh occurs once approximately every 10 microseconds (about 200 cycles) and can cause a 1 cycle DRAM memory operation to take up to 8.5 cycles to complete. We could see up to 11 refreshes while executing a 2 KB segment of code, so if take this into consideration, our throughput drops to about 19.2 MHz.

Now if we also consider a branch or jump every 20 bytes (that's actually quite a high average), we get an overall throughput of around 18.3 MHz; still a remarkably good figure. Naturally, your own programs will vary from this mark, depending on how you write them, and how often you perform other accesses that can cause slowdowns (such as I/O access or frequent writes to mirrored memory).

Other SuperCPU Timing Issues

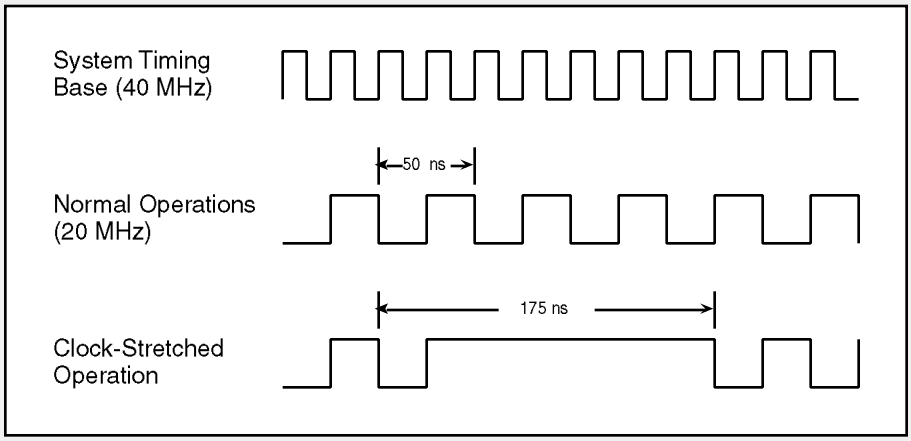
With the proverbial 'can of worms' now open, let's consider the other timing aspects of the SuperCPU. The SuperCPU Special Function Timing Chart will be our guide as we discuss the various functions. Please note that the signal relationships on the chart have been calculated using the NTSC dot clock frequency, but the times indicated are identical on PAL systems.

Access to Static RAM is always one cycle for reads. Writes also take one cycle—except under certain conditions. What can slow down a write is 'mirroring', where data is being written through to the RAM in the host computer. Mirroring is performed in order to make sure that the VIC chip, which reads screen and color data from the computer's own RAM, has proper data for the display. Since it isn't possible to detect in real time exactly where (in memory) the VIC will be looking for data, the SuperCPU's default is to mirror all writes to Bank $00 RAM.

A mirrored write doesn't automatically mean a speed penalty, however, since the SuperCPU employs a one-byte cache (buffer) for write-throughs. Refer to the Mirrored Memory Cache Latch timing. You'll see that the cache is cleared during the low phase of the first dot clock cycle following the

Clock-Stretching

While many of the operations within the SuperCPU occur at a normal 20 MHz rate, some operations may take longer than the single cycle in which they should usually occur on a 1 MHz Commodore computer. Under these circumstances, the high half of the clock signal is stretched to meet the requirements of the operation. In the example below we show the System Timing Base (40 MHz) and how the System Clock itself would look while executing instructions at full speed (Normal Operations). The bottom example shows what happens during an operation that requires 3.5 20 MHz clock cycles to complete. The low part of the cycle has a duration of 25 nanoseconds, or half of the duration of a 20 MHz cycle; the high portion of the cycle is stretched, giving it a duration of 150 nanoseconds. The total duration of this cycle is 175 nanoseconds, or 3.5 times the 50 nanosecond period of a standard 20 MHz cycle. You may note that this causes what may appear to be a phase inversion; that is to say, the clock signal is now low during a period in which it originally would have been high, and vice-versa. This factor is of no importance, however, and only becomes an issue when synchronization with the host computer's clock becomes necessary. When that occurs, clock stretching is used as required to bring the SuperCPU clock into phase with the host computer's clock.



computer's Phase 2 signal going low. The latch stays low for 25 ns, and the cache is then ready for another byte to be written through. Any byte must be in the cache at least 70 ns prior to the dot clock high transition that signals the computer's Phase 2 line to go high—any later than this and the cache mechanism has to wait until the Phase 2 clock cycles around again. In either case, the operation of the cache is transparent to program timing as long as no additional mirrored writes occur while the cache latch status is high.

This leads us to consider what happens if the cache is already full when a mirrored write occurs. The result is a clock stretch for the SuperCPU Phase 2 clock, which will stay high until the cache is cleared. Once this has occurred, the waiting byte can be put into the cache, and the SuperCPU returns to normal 20 MHz operation. Spacing writes to mirrored memory (as well as using the optimization modes to reduce mirroring) will help maximize program efficiency. With one mirrored store every 19th cycle you'll get maximum throughput of one cache write per 1 MHz cycle, provided there are no other special functions that slow things down.

The next area we'll look at is I/O access, which covers reads and writes to $00D000-$00DFFF with I/O switched in, and also includes a few miscellaneous locations. Most I/O reads and writes follow the timing specification shown as SCPU Phase 2 I/O Reads/Writes. Any store or load cycle to I/O causes the SuperCPU Phase 2 line to go high until the data can be written or read. The store or load must occur at least 70 ns prior to the dot clock high transition that signals the computer's Phase 2 line to go high in order to have the I/O access occur during the current 1 MHz cycle. If the access is to standard I/O, the SuperCPU Phase 2 will transition low about 105 ns after the rising edge of the dot clock cycle that signals the host computer's Phase 2 to go low. This timing of the SuperCPU's Phase 2 line also applies to cache full mirrored writes to RAM, memory location $000001, and reads from $00DF01, $00DF21, $00FF00. Furthermore, this timing is used to read from ROM cartridges installed in the $008000-$009FFF or $00A000-$00BFFF memory areas. An 8-cycle spacing of standard I/O access provides best throughput.

In addition to standard I/O reads and writes, there is a Long I/O Write timing specification that applies to locations $00DF01, $00DF21 and $00FF00. The long write has the same input deadline as all other special timing functions, but holds the SuperCPU Phase 2 line high 24 ns past the start of the fourth dot clock cycle after the computer's Phase 2 is signaled to go low. This timing was created to satisfy requirements of Commodore REU DMA operations.

The final special I/O timing specification only applies to writing to the CIA chips ($00DC00-$00DDFF). Here, a standard I/O write is performed, but the next processor cycle (usually a fetch of the next opcode) is stretched into the next computer Phase 2 cycle, and ends where a long I/O write would end. It was necessary to use this timing to make it impossible to read back from a CIA during the two 1 MHz cycles following the write. The reason? Because the CIA I/O lines are terminated with resistors, causing them to react slowly when going high. Reading too soon can generate erratic results.

Last of all, there is one final inconsistency in timing that isn't indicated on the chart. This applies to accessing the special SuperCPU RAM placed in the I/O area. Access to this RAM takes two 20 MHz cycles instead of one, because the SuperCPU needs to first decode that this area isn't actual I/O before it can perform the load or store function requested.

Conclusion

There are many factors to consider if your program is to achieve optimal throughput. Reducing mirroring, spreading special accesses, and optimizing routines that really need it will give you the most speed for your effort, without making the process excessively difficult and time-consuming.