Safely freezing the C64 on an asynchronous event

Introduction

If we want to safely freeze a C64 on the cartridge port, this should happen as software-transparent as possible. This allows the software to continue to run after the freezer software has done its job. Several freezers have been implemented throughout the years, and likewise, many freezer-protection tricks have been applied in sophisticated demos to avoid a freezer to work. Therefore, it is not an easy task. This document describes the procedure applied on the Ultimate 1541.

Analysis

As the Ultimate 1541 possesses its own processor, the freezing method that is under investigation is one that allows this processor to take over the control from the 6510 processor. The advantage is that neither the processor status, nor the stack is altered. Taking over control means that we *will* make use of the peripheral resources of the C64, so we need to restore their state as accurately as possible.

–DMA line

Because the C64 is taken over externally, freezing it and writing to it is done by using the –DMA line. Asserting –DMA causes two things: it will force AEC to the processor low, and it will also force RDY low. The first is to make the address bus (and R/–W) float, the second is to stop the execution of the processor. But, the –DMA line cannot be asserted at just any time.

First of all, setup and hold times need to be taken into account, so it needs to be synchronized with PHI2, which is easy. A measurement on the REU indicated that PHI2 is asserted 80 ns after the falling edge of PHI2, and released somewhere during the low phase of PHI2.

Secondly, it should be remembered that when the 6510 is doing a write, it does not 'listen' to the RDY line, and will just force the writes to take place. So, when –DMA is asserted during a write, the address bus will float, and the write cycle gets lost. Usually, this will cause the software to crash sooner or later.

The VIC does not have this problem while doing its bad-line or sprite DMA, because it generates AEC itself. First it will pull RDY low, but keeps AEC high for three more cycles. The maximum number of consecutive writes a 6510 can do is three; namely, when it is storing its state in order to serve an interrupt. So any write that is still pending will be carried out correctly before the processor comes to a full stop.

The REU does not have any problems either, because it always asserts the –DMA line right after an I/O write, namely the write that initiates the DMA transfer. There is no condition that the following cycle would have been a write as well, so the –DMA line is always asserted in a read cycle of the CPU.

The 1541 Ultimate wants to stop the C64 on an asynchronous event; when the freeze button is pressed.

Mission

In order to pull the –DMA line at the right moment, namely during a write cycle, it is necessary to look for a condition that indicates that it is safe to pull the DMA line.

Problem

Unfortunately we cannot pull the DMA line in the CPU cycle, in which we *know* that the CPU is performing a read, that is, while PHI2 is high. In case of a write, the CPU already pulls the R/–W low during the VIC half of the clock cycle (PHI2 is low), but because the VIC has AEC low, we cannot see it at the cartridge port. The VIC is doing a read, so R/–W is high during the first half of the clock cycle.

It was attempted to pull –DMA low around 250 ns before the falling edge of PHI2, so after the R/–W line had stabilized. This works perfectly on a 6510, but makes the 850x CPU in a C64c crash. Apparently, this CPU does not like to see RDY 'true' on a rising edge of PHI2, and 'false' on the subsequent falling edge.

Bad lines

A very simple solution is actually to listen to what the VIC is doing, rather than what the CPU is doing. When the VIC is doing DMA is pulls the BA line low on the cartridge port; indicating that that cycle is not available for external access. The VIC has already correctly stopped the CPU when it is doing its DMA, so we can just 'queue up', and extend the DMA cycle with our own for as long as we want. All we have to do is pull the –DMA line low somewhere within the bad line! Unfortunately, this trick does not work when the DEN bit in $D011 is false (blanked screen), because the VIC will not perform any DMA.

So what now?

We need to predict whether the CPU is going to do a write in the *next* cycle. Is this possible?

Yes it is! It can be proven by carefully analyzing the instruction set of the 6510 and the operation it performs on each clock cycle, plus all combinations of instruction pairs. The latter does not even turn out to be necessary, because it shows that *every* instruction, including the BRK instruction, start with two read cycles[1]. Some instructions do a write, some do two writes in a row (like JSR, but also any read-modify-write instruction), and BRK/interrupt does three. But when after a write a read is seen, then we know that the cycle following the read will *also* be a read!

But what if the program executing does not perform any writes? For example, it is in a loop, reading an I/O and branching back, while interrupts are disabled. The software also disabled the NMI by effectively pulling the line low and leaving it in that state. No writes will take place, so the condition to freeze will never occur. This can only be 'fixed' with a workaround; a timeout! When no writes occur within a certain period of time, it will be considered very likely that the next cycle will be a read as well.

---

[1] Note that handling an interrupt is in fact executing a BRK instruction with some variations.

Accurate resume

Some demos are totally synchronized with the screen and actually lock up when the program execution gets out of sync with the scanning of the VIC. The bad line method described above has the benefit that resuming can take place *at the same* bad line. DMA– can be released at the same condition that was used for freezing, and the software can do the rest. The software records the raster line at which the C64 entered the freeze-state, and waits for the previous raster line before releasing the C64 upon the next bad line. The 6510 CPU will then resume in *exactly* the same cycle in the raster scan[2].

There is a problem, however. It is not possible to read the value that was last written to $D012; since reading this register gives the actual raster line, rather than the raster match register. This problem can be easily fixed by clearing the raster interrupt flag and waiting for it to be set again, and then read the register again. However, this only needs to be done when the interrupt was enabled.

Solution

The previous paragraphs offer two solutions that both have benefits and drawbacks. The following procedure gives the best of both:

- Try to freeze upon the occurrence of a bad line. If this succeeds, read the raster line at which this occurred, determine the original content of the $D012 register and use the same condition upon exit.
- If this fails to occur within one frame, try the R/Wn method. Exit can be immediate at any cycle.
- If this fails, just perform a hard-stop, pulling the DMA line low upon the next falling edge of PHI2. Exit can be immediate at any cycle.

[2] The behavior of BA when sprites are enabled is not accounted for, and should be investigated in more detail.