

# The Evolution of CCD Clock Sequencers at MIT: Looking to the Future through History

John P. Doty, Noqsi Aerospace, Ltd.

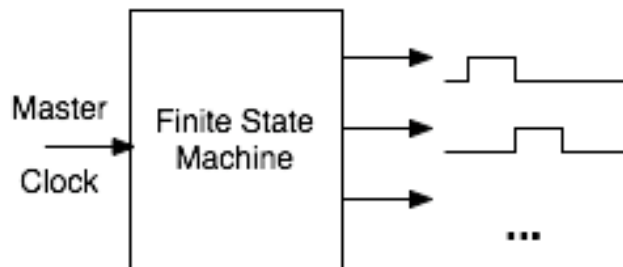
This work is Copyright 2007 Noqsi Aerospace, Ltd.

This work is licensed under the Creative Commons Attribution-Share Alike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

## Introduction

OK, you have a CCD imager, and you want to make it go. To do this, you must apply drive signals to the CCD electrodes ("clocks") in a complex organized sequence with well controlled timing. In addition, your video signal processing chain will need to be synchronized with the CCD, generally requiring more clocks. The "sequencer" is the circuit that controls all of this elaborate timing.

## Basic Sequencer



All of the designs I'll describe fit the definition of a "finite state machine", sometimes without inputs, and sometimes with inputs from a controlling processor (itself a finite state machine). However, in the descriptions below I'll reserve the term "state machine" for the portion of the sequencer where the designer actually has to deal directly with gates and registers: higher level components containing state machines will be called by their higher level names (e.g. "microcontroller").

Many of the designs below consist of a lower level "pixel" sequencer, and a higher control level. Sometimes the "pixel" level actually generates sequences for two or more pixels for each higher level operation, but I have omitted those details to concentrate on the critical architectural details.

## **Ad hoc State Machine**

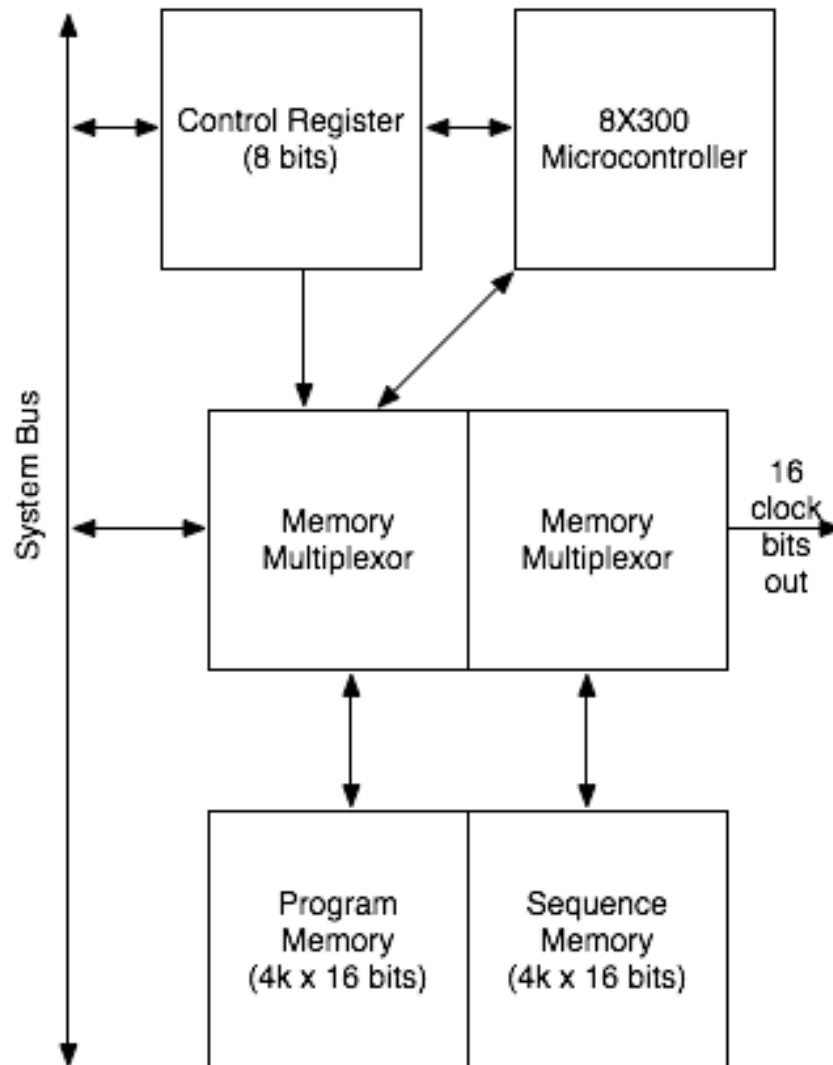
The first CCD project at the CCD Laboratory (at that time it was the Balloon Laboratory) was a prototype star tracker for a stratospheric balloon gondola (Jeffrey Bokor, S.B. thesis, 1974). Jeff and I designed a state machine from 4000 series CMOS that reproduced the timing diagrams in the data sheet. We didn't do this in any particularly organized way.

Subsequent students built several generations of similar clock state machines, with CCD format and clocking strategy hardwired.

## **Shadowed Microcontroller**

In 1982, Charlie Hulcher and I came up with a more flexible approach, using an 8X300 microcontroller (Charles Hulcher, unfinished SB thesis). The 8X300 had an unusual architecture for the time: it had separate instruction and data memory busses, and required a fixed time (two clock cycles) for every instruction.

## Microcontroller-based Sequencer



The only "data memory" we had was an 8 bit control register: program variables (loop counters) were kept in registers.

This worked by "shadowing" the program memory with 16 bits of sequence memory. For example, every time the processor executed location 17 in program memory, the contents of location 17 in sequence memory would appear on the output. So, we programmed it by arranging nested loops with the correct timing, while accompanying each each program instruction with the 16 bit output it was to generate.

To make this easier, I wrote a "compiler" in a local Forth dialect, LSE. Using this compiler, a piece of code like:

```
450 FOR
```

S1 HIGH 3 DELAY  
S1 LOW 3 DELAY

NEXT

would command the sequencer to set S1 high for three instruction cycles, then low for three instruction cycles, repeating 450 times. The host computer would configure the control register to give it access to the memory, compile the code into the memory, and then configure the control register to give the 8X300 access to its program memory and start it.

The compiler had to keep track of two programs: the 8X300 instructions for the loops and delays, and the sequence of 16 bit output states. These needed to be kept synchronized so that the correct state was associated with each instruction. Despite this complexity, the compiler was only 260 lines of LSE code!

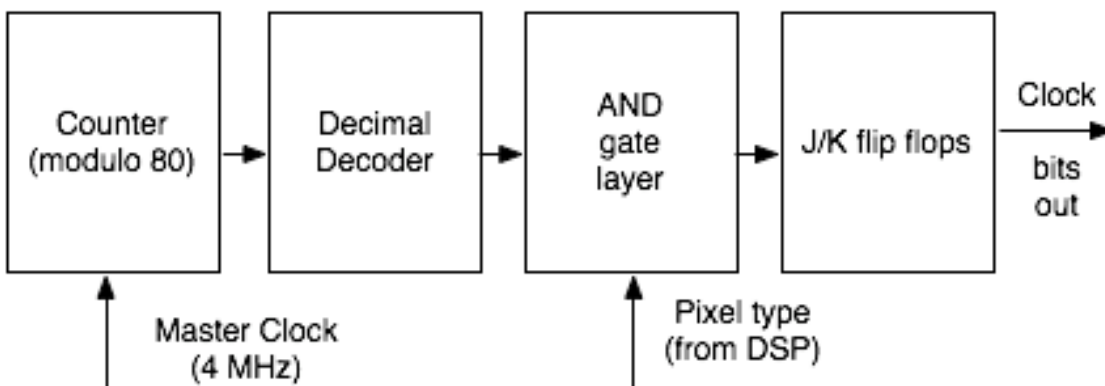
The 8x300's seven 8 bit registers sufficed for seven levels of loop nesting for loops of up to 256 iterations: the compiler supported up to 65536 iterations by automatically using two levels if necessary. An "infinite" outer loop didn't need a register. We never encountered a set of requirements too complex for this.

This sequencer proved a great advance: we could easily experiment with different clocking modes for subframes, pixel summation, continuous clocking, etc.

### Regular State Machine Plus Processor

When we began development of the Silicon Imaging Spectrometer for the ASCA satellite, we needed a new approach. While we would have liked to use the 8X300 design for the ASCA SIS, the 8X300 was very intolerant of radiation, unsuitable for space flight (and also obsolete). So, we adopted a mixed approach, with sequencing within a pixel interval handled by a hardwired state machine with a regular structure, but with higher level sequencing handled by a DSP within the digital electronics.

### State Machine/DSP Sequencer



The decimal decoder decoded the state number into 18 lines, ten for the lower decimal digit, 8 for the upper, only two of which were asserted at any time. This allowed a two-input AND gate to decode any specific cycle number. Each J/K flip flop was associated with two AND gates, one to determine which cycle it needed to be set to one, and the other for zero. Transitions that were dependent on pixel type (like parallel clocks) were associated with three-input AND gates: the extra input controlled which pixel type the transition was to be used for.

The DSP controlled the pixel type, potentially changing it as often as every 20  $\mu$ s, but in practice much more rarely.

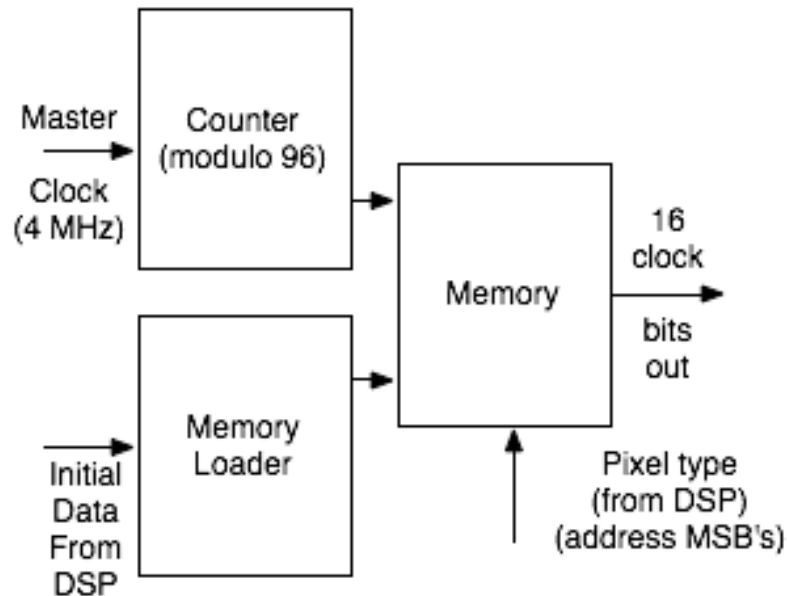
This had most of the useful flexibility of the shadowed microcontroller approach. While within a pixel, the possible clock sequences were fixed, different clocking strategies were still possible by changing the sequence of pixel cycles commanded by the DSP. Also, in the wire-wrapped prototype it was relatively easy to adjust the sequence within a pixel: the regular state machine structure meant that it was generally easy to identify a small change of wiring that would accomplish the desired change.

The MIT ground support setup for the ASCA SIS lacked the control DSP, but it proved easy to emulate its sequencer controlling function with one of the old shadowed microcontroller sequencers, programmed as described above. But for the flight system, there was no special programming support: the assembly language code issued pixel types as part of its pixel processing loop.

### **Sequential Memory Plus Processor**

When we were working on the ASCA SIS, Keith Gendreau commented to me that he visualized these sequencers as large sequential memories. Given that memory chips were continuing to advance, this seemed to me a good way to actually structure the sequencer for our next CCD project: the UV camera for the HETE-1 satellite. The first version of this was similar to the SIS version, but with memory substituted for the decoder and flip flop stages.

## Sequential Memory/DSP Sequencer

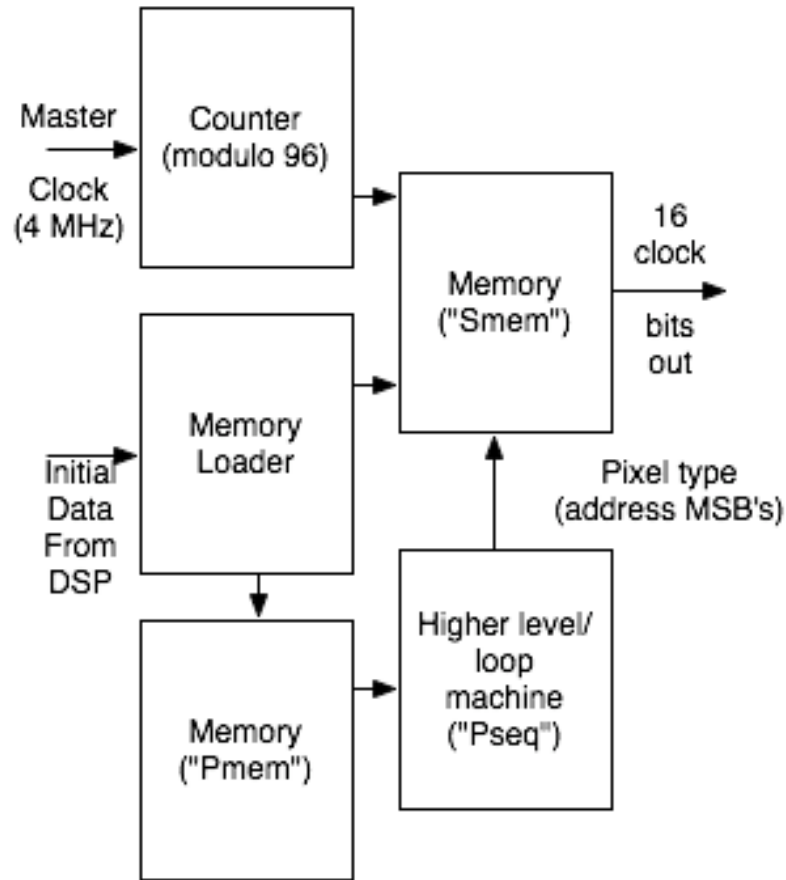


The memory just holds bit vectors that correspond to the state at each step. The DSP loads these in before starting the sequencer. The DSP then starts the sequencer and provides a new pixel type every 24  $\mu$ s. This sequencer was our first to use FPGA technology: Tye Brady integrated everything except the memory into an Actel FPGA.

For programming, a student named Cotton Seed wrote a graphical editor for the arrays of pixel-level bit vectors, so one could simply edit a timing diagram. I improved this and maintained it for a few years, but it has unfortunately fallen into disuse. For the higher level, I wrote a threaded code interpreter for a tiny subset of Forth in C (just two pages of code!). A simple program in this language issued a pixel types into a queue. An interrupt handler synchronized to the pixel rate transferred these from the queue to the sequencer. The threaded code performed the counting and looping.

Steve Kissel and Tye Brady found the requirement for active supervision by a DSP burdensome in testing, so they offloaded that function into a programmable device, the "Pseq". This was like a computer CPU, but with no data manipulation capability, only loops and output. It generated the bits to select the pixel type according to a program in a memory chip. This approach required an extra memory chip, but the rest of the logic was in the FPGA. Here, the DSP loaded both chips before starting the sequencer, but then it could just let it run.

## Sequential Memory/Loop Machine Sequencer

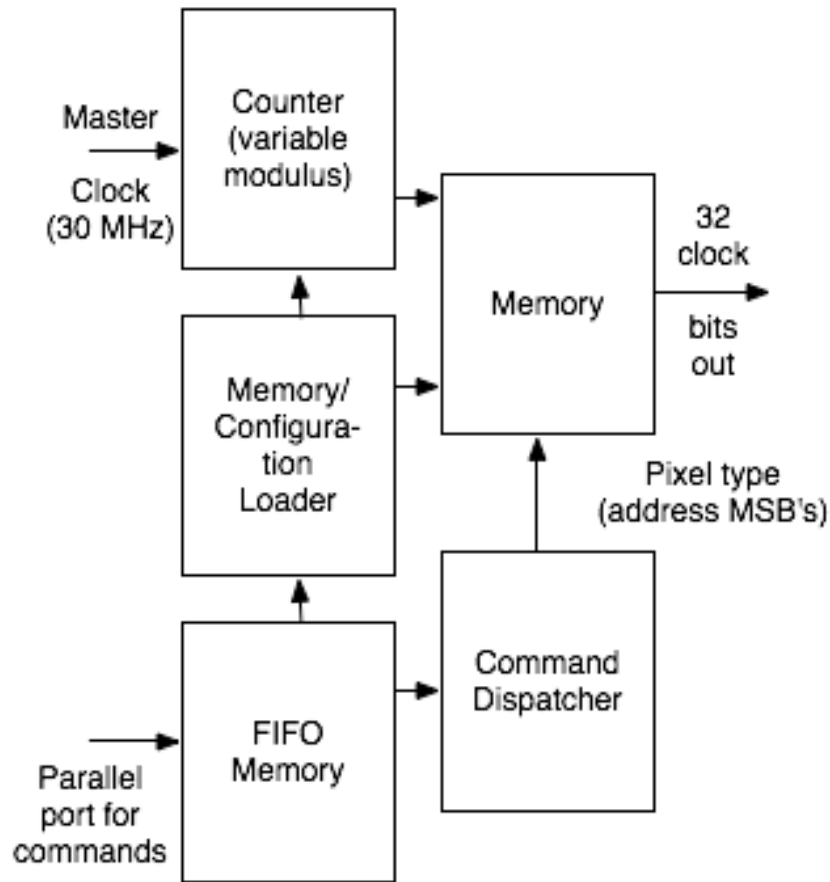


This design was used for both HETE-1 and HETE-2, and variants by Dorothy Gordon and Frank LaRosa also flew in space on Chandra, ASTRO-E1, and Suzaku.

### Sequential Memory Sequencer with Simple Loops and FIFO

After the Suzaku development, we wanted a versatile sequencer for laboratory development. I wanted to avoid the specialized Pseq microcode, but also to reduce traffic and synchronization requirements between the control processor and the sequencer. We were working with more complex CCD's and with faster systems, so a 32 bit sequencer (up from 16) with a 30 MHz clock (up from 4 MHz) seemed desirable.

## Sequential Memory/FIFO Sequencer



Frank LaRosa did the FPGA design for this, while a student, Mike Trakimas, designed the circuit board. The control interface is a PC parallel port. The computer loads memory and configuration registers and then starts to issue commands to the sequencer. Each 16 bit command contains the address MSB's that identify the pixel type, and a repeat count. This greatly reduces the load on the control computer, as it can issue clocks for a whole serial row with a single command. The first-in first-out (FIFO) memory relieves the requirement that the computer issue its commands in synchrony with sequencer operation. These features greatly reduce the computational load on the computer without requiring a complex "Pseq" loop machine and its microcode.

I rewrote LSE in C to program this (available from <http://noqsi.com/software.html>).

### Future Possibilities

With block memory now available in FPGA's, we may want to consider eliminating the external memory and making a sequencer in just one chip. However, the sequential memory approach tends to require relatively large memories. Each control signal requires a bit in memory for each time step and pixel type. The shadowed

microcontroller approach requires less memory and is more flexible, but requires the most complex programming system.

The structured state machine approach from the ASCA SIS looks like a simple and attractive possibility with a reprogrammable FPGA. For the pixel level it needs only few flip flops, and perhaps a second (CCD row level) layer could relieve the burden on the processor (although a dedicated microcontroller within the FPGA could easily handle the higher levels). A reprogrammable FPGA would eliminate its inflexibility.

For programming, I note that all designs since the shadowed microprocessor work with a set of pixel-level sequences that are all the same length, while the higher levels are relatively simple counted loops. Even programs for the more flexible shadowed microprocessor design tended to follow these rules. It seems that a graphical editor for the pixel timing blocks would be very handy, while the higher levels could be represented by a trivial programming language. This suggests that a universal programming system for future sequencers should be relatively simple. Back ends could emit bit vectors, HDL, or microcontroller code as appropriate to the specific design.