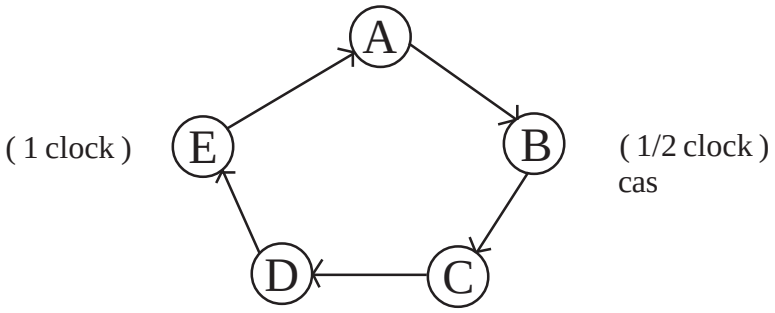
Auxiliary Information

Candidate: James T. Battle

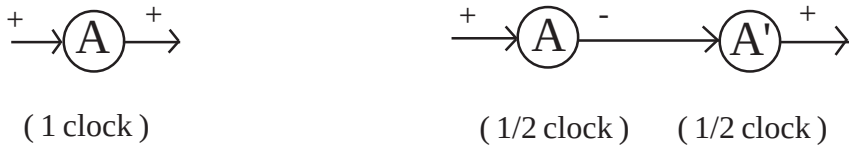
Sample of Original Work

Two years ago I was given the task of designing a DRAM controller for a 16.6 MHz 68000 microprocessor. I felt that it would be possible to use ordinary 120ns dynamic RAM (DRAM) chips and achieve 1 wait-state (w.s.) performance. On the way to a solution, I developed a technique for systematically generating a synchronous state machine which operates at twice the system clock rate. This is done by allowing the state machine to make transitions on both positive-going and negative-going clock edges. This means that some state bits can change only on positive clock edges, and the other bits on negative edges. Another constraint was that states which were adjacent in time must be adjacent in the state space. This allows glitch-free decoding required to control the DRAMs. Advantages of this technique are a simplified clock distribution scheme, no clock skew problems as with a multi-frequency clock, and a "tighter" design than an ad-hoc solution. This technique could be generalized to a multiphase clocking scheme.

To explain the idea, I will describe one loop of the DRAM controller state graph, the refresh cycle. All DRAM chips need a periodic access sequence to maintain the contents of its memory. This is performed in this instance by asserting the "CAS*" signal, waiting, asserting the "RAS*" signal, waiting, releasing "RAS*", waiting, releasing "CAS*", and waiting a precharge period. The exact duration of the wait isn't critical, but it must meet a minimum specified by the maker of the DRAM. The state graph would look something like this:



In actually implementing the graph, each logical state is implemented as one or more actual states. This is because actual states have no knowledge of time; passage of time is represented by changing state. For instance, if a logical state persists for 1/2 clock period, there is only one way to implement this. If a logical state persists for one clock period, there are two ways to implement this:



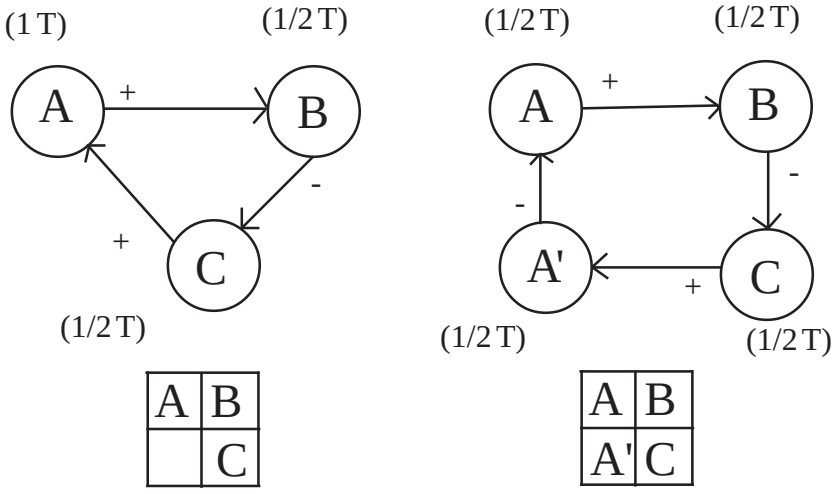
For 1 1/2 clocks, there are three ways, for 2 clocks there are five ways. When turning a logical state diagram into an actual state diagram, there are many degrees of freedom in the implementation. There were two constraints that limited the choices so only a few implementations were valid. The first constraint is that in a given loop of the graph, there must be an even number of positive edge transitions and an even number

of negative edge transitions. This is because to complete a cycle in the graph, the state bits must return to their starting value. Thus, each time a state bit changes away from its value in the starting state, it must late make a transition back.

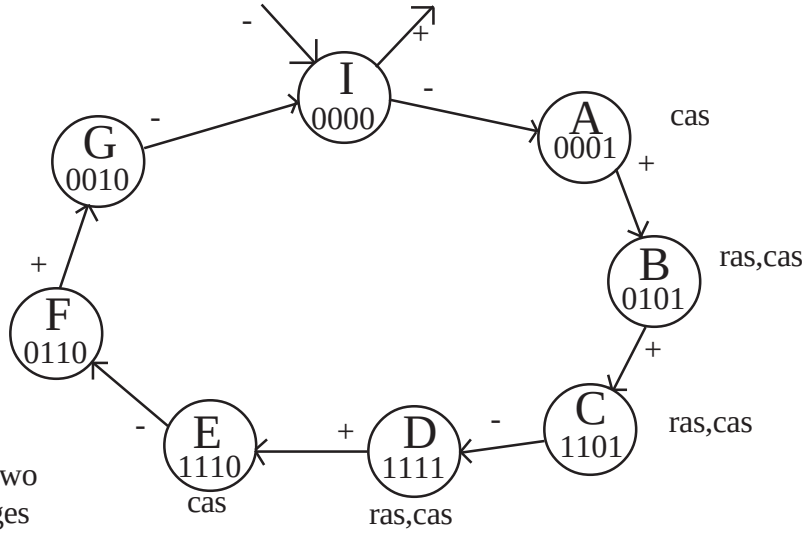
The other constraint was that imposed by system requirements. States adjacent in time must be adjacent in the state space to allow glitch-free decoding.

Some of the design process is trial and error. Knowing how many state bits to assign for positive edges and negative edges to allow for a conflict-free solution and choosing which sequence of actual states to implement a logical state require experimentation.

I modified the usual technique of using Karnaugh maps to make the bit assignment for the states. I arranged the maps to that horizontal motions on the graph correspond to transitions on positive clock edges, and vertical motions negative edges. This graphical technique makes generating assignments and detecting conflicts easier, and helps suggest how to fix any problems. A simple example is:



After an hour's work, the solution was to use two bits which were clocked on positive edges, and two bits which were clocked on negative edges. For the part of the graph in exposition, this was the solution:



Note: first two state bits change on positive edges and the second two change on negative edges

The graph of the state adjacency looks like this: