# Clocked Latched Fedback Memory Finite State Engine. I.

## A Novel Technique for the Design of Arithmetic and Control Processors

John H. Letcher
Department of Mathematical and Computer Sciences
University of Tulsa
Tulsa, Oklahoma 74104

## A B S T R A C T

A technique has been developed to reduce a finite state diagram for a proposed processor and to implement this as a very fast, very inexpensive processor. The device consists of a combination of latches and memories, only, to effect anything that can be stated by the situation calculus. These processors exhibit the unusual behavior that the clock may be entirely asynchronous from some extremely fast level down to essentially zero frequency. The outputs from this processor are absolutely stable (do not vary with time) except for a brief time interval immediately following the clock pulse. Because of this characteristic, multiple devices may be used in parallel and can be globally synchronized. In these multiple applications, it is possible to define a sequence of processors that are clocked at precisely the same rate. Because of the wide range of clock rates possible, it is possible to define parallel designs that slow down, and therefore dramatically reduce the power dissipation of a processor when it is not needed. After construction, this design may be dramatically modified with only the reprogramming of the memory which is at the heart of the processor. It is possible to design and implement the firmware for these processors through the use of high level computer languages, only. This design technique is general enough to form a basis of a new class of VLSI components.

## INTRODUCTION

Professor Carver Mead([1] ) stated that he felt that a 10,000 fold increase in the cost effectiveness in computing would occur in the next decade if the ability to produce custom processors are placed in the hands of all design engineers including those who do not work for the large silicon vendors. His vision has led to the development of silicon compilers, which are computer programs which translate the definition of a processor into a form that makes it possible for a computer to construct custom integrated circuits. He proposes stating the processor problem in terms of a set of equations; then, a custom processor can be produced by sending machine readable medium to a company that will build the custom processor.

Clearly, the statement of the function of a control processor must be established before the design work may begin. However, it would be preferable to use a suitably general technique that allows this specification to be implemented through the use of commercially available microcircuits and through the use of the equipment found in essentially all electronic development laboratories (PC computers and PROM programmers). This paper describes the development of such a technique.

## The Definition of a Finite State Machine

A finite state machine is a 5-tuple:

$$M = (S,X,Y,g,S_0)$$

where
- **S** is a finite state set of states $\{S_i\}$ ;
- **X** is a finite set of input symbols $\{X_i\}$ ;
- **Y** is a finite set of output symbols $\{Y_i\}$ ;
- **g:** $S \times X \Rightarrow S$ , the next state function ;
- $S_0$ is the designated state called the initial state.

Note: 1) All states are recognizing states.
2) The values of all $X_i$ and $Y_i$ are either 0 or 1.

There always exists a complete State Table:

| g | $X_1$ | $X_2$ |
|---|---|---|
| $S_0$ | $S_i$ $Y_j$ | ... |
| $S_1$ | ... | |

The input to the above defined machine is a finite sequence of input symbols, $X_i$ $X_j$ ... .

Let us define the number of input values $X_i$ to be $N_I$, the number of output values $Y_i$ to be $N_0$, and the number of states to be $N_s$. It is convenient to redefine **X** and **Y** as binary numbers, a concatenation of the binary values of $X_i$ and $Y_i$, taken as bits, to construct unsigned binary numbers, i.e.,

$$X = \sum_{i=1}^{N_I} X_i * 2^{i-1} \quad \text{and} \quad Y = \sum_{i=1}^{N_0} Y_i * 2^{i-1} .$$

The maximum value of **X** is $2^{N_I} - 1$ and of **Y** is $2^{N_0} - 1$. The minimum values **X** and **Y** are zero.

## Finite State Diagrams

The action of a proposed processor must be defined, a priori. This description may be given in several convenient ways. The first is a graphical approach and the result is a finite state diagram. This diagram gives a visualization of the action of the processor and its response to all possible combinations of inputs thereby producing the desired outputs.

These diagrams are given in terms of defined ***states*** and ***activities***. The processor will be driven and synchronized with other activities by its being fed a sequence of clock pulses. Between each sequential pair of clock pulses, an ***activity*** occurs. This activity calculates the values of the output signals which will be presented as outputs (simultaneously, at the next clock pulse). At the start of each activity the processor is found in a ***state***. Multiple states (i.e., different states) are required in situations where the recipes used to calculate output values from input values may differ from activity to activity depending upon what has happened before.

The action of a processor can be described as an activity which occurs within the processor after each clock pulse. This activity involves using the observed input signals and, with full knowledge of the state in which the processor found itself at the time of initiation of this activity, the output signals of the processor can be calculated. After the activity has been completed, all signals within the processor are quiescent (not changing in value). Therefore, at any time from the instant at which all signals stabilize until the next clock pulse, the internals signals of the processor do not change. A snap shot of the internal values may be taken anytime during this time interval. The configuration given by a subset of these signals defines a state. That there are different states within a processor implies that, for a given set of inputs, the desired outputs will perhaps vary, that is, from state to state.

Graphically in a finite state diagram, a state is represented by a circle and an activity is represented by a directed arrow which extends from one state to another. A complete finite state diagram is the set of all possible states of a processor and all possible activities that could occur with all combinations of inputs. Each activity ***belongs to*** a state. In graphical form, each activity starts in one state (the foot of the arrow) and ends in a state (the head of the arrow). These two states may or may not be the same. To complete the definition, it is desirable to label each activity, **X:Y**, where numerical values for **X** and **Y** are given. As will be shown later, this diagram alone, is sufficient to design and build the proposed processor.

## Combinatorial Logic

The production rule $Y(t+\varepsilon)::=f(X(t))$ is an algorithm that states that in combinatorial logic (which is normally made up of the usual AND, OR, NOT, XOR, etc.) that the input signals to this logic module (which may be represented by a set, **X**, of binary values) produces a set, **Y**, which represents the binary output values. $\varepsilon$ is a non zero time delay. Combinatorial logic may be represented by a finite state diagram with only ***one state*** and a set of activities extending from this state folding back onto this state, one for each of the defined input values. We find it convenient to label each of the activities with a number, according to the binary value, **X**, and listing as a part of the label the binary value of the output values, **Y**, thus: **X:Y**.

## The Finite State Equation

It is seldom, if ever, possible to design an arithmetic or control processor through the use of combinatorial logic, only. Under certain circumstances, depending upon what has happened before, the processors response to an input may depend to a great extent on its history. Under the action of its inputs, a processor will move from state to state and

can best be represented by the situation calculus([2] ) in which $S^1$=result(e,s). To keep a consistent nomenclature, we prefer to express our finite state equation thus:

$$Y(t+\tau) ::= f(X(t),Y(t)).$$

This states that the outputs of a processor at time, t, plus an increment, $\tau$, is obtained from the values of the output at time t combined with the inputs at time t according to some defined recipe, f. This production rule is sufficient to represent most arithmetic and control processors. A finite state diagram for such processors would consist of a set of activities and states. Again, the activities are labeled with each of the defined inputs and the defined outputs. For any activity, the calculation of the outputs from the inputs is given by the production rule, f, which is a function of the input values **X** and the state number to which the activity belongs.

The finite state diagram can be drawn for the convenience of the hardware designer. Each processor must be initialized and this state in which the processor finds itself at the start of time shall be numbered, 0. All of the others may be labeled with a number starting from one up through as many states as exist. An important point that can be made is that any finite state diagram may be linearized. That is, each can be redrawn so that the states are shown along a single line. No new information is added and the fact that one state is tagged with one number or another is not important at all.

A finite state diagram may be reduced if there exist two states in which the action of all of the activities belonging to state is identical to those of another state. The diagram is reduced by redefining the activities pointing to the higher number state to point to the lower number state. The higher number state is simply erased.

A finite state diagram can be ***abstracted*** by saying that this is a processor characterized by three numbers: the number of input states, $N_I$, the number of output states and $N_0$, and the number of states in the diagram, $N_S$, and the production rule. Therefore, the firmware for the device which implements such a processor can be calculated in an automatic way from $N_I$, $N_0$, $N_S$, and a statement of the production rule, f.

## The Hardware Device([3] )

Let us assume that the designer has in hand the three parameters, $N_I$, $N_0$, $N_S$, and the production rule, f. It is possible to reduce this to hardware by the addition of two input signals: a clock and a reset signal. Now, our device can be represented as a function module with inputs, **X** (as before) plus the clock and the reset pulse. The outputs from this device is the number, **Y**, as before. In our production rule, a defined time delay, $\tau$, was defined. In order to implement this in practice, it is necessary to present to our device a clock, which is a sequence of pulses with any reasonable duty cycle (i.e. it may be a square wave with a 50% duty cycle, or a sequence of extremely narrow pulses with a very low duty cycle). The distance between the rising edge of the pulses, however, is defined to have the value of $\tau$. This is represented simply in hardware as shown in Fig.1. This consists schematically of only two devices, a latch and a memory.

To refresh the readers memory, a latch is a device with multiple matched inputs and outputs. In operation, it is assumed that the input values to this device are stable over a very brief time interval before and after the rising edge of a clock pulse that is fed to this latch. During other times, the input values to this latch may do anything they please. On

the rising edge of the clock pulse, the circuitry inside the latch sets the values of the outputs (within a small but finite time delay) to match the values that the inputs had over the brief interval before the clock pulse. After the time delay of the latch, the outputs of the latch are set and fixed. Therefore, they remain invariant until the latch receives another clock pulse. The reset pulse overrides the normal action of the latch and sets the output values of the latch to be all zeros. This signal is used only to initialize a processor.

The memory represented in Fig.1. is any device which has a set of inputs and a set of outputs where the set of inputs represents a binary number, a location within the memory, which, after a time delay $\lambda$, will cause the outputs of the memory to be set to the defined contents of that memory. This deceptively simple looking hardware device will implement any processor that can be described by the situation calculus. The latches shown in Fig.1., may be implemented through the use of shift registers, parallel entry shift registers, or a member of other types of devices that carry out this function. The memory, shown in Fig.1., may be implemented through the use of commercially available ROMs, PROMs, EPROMs, EEPROMs, RAMs, PALs, HALs, PLAs, and other devices that carry out this function.

## Properties of the Device

We may view the action of this device in terms of the timing diagram given in Fig.2. The clock is an asynchronous or synchronous stream of pulses subject only to the restriction that the time period, $\tau$, must be greater than $\varepsilon + \lambda$. Certain important facts should be recognized: 1) the output signals are stable except from time $t_0$ to $t_0 + \varepsilon$ after the clock. (Here, $t_0$ is defined as the time at which the rising edge of a clock occurs during this cycle.) 2) input signals need only be stable between $(t_0 - \varepsilon)$ to $(t_0 + \varepsilon)$ which is a small fraction of total time. 3) the clock must have a period greater than the sum of the delays of the latch and that of the memory $\lambda + \varepsilon$, but it may have ***any*** larger value.

It should be noted that such a device can operate over a very wide range of clock frequencies and that the clock may be entirely asynchronous in its operation. It is perfectly allowed to use the output of a finite state engine to adjust the clock ***rate*** for later tasks to produce a desirable effect of giving a processor longer to carry out certain tasks that it does to carry out others.

It should be noted that a device of this sort has the ability to ***regularize*** signals. This means that over a time period between clock pulses an input can be delayed and presented with the new value at precisely a defined time value. This makes it possible to define sets of finite state engines, processors working independently and in parallel, yet with the ability to be easily synchronized.

## Instruction Encoding of the Finite State Engine

Given a finite state diagram for a proposed processor, it is possible to use this diagram to calculate the contents that the memory must have in order to carry out the function of the proposed processor. First, we will describe how this is done by hand. Then, we will show how this can be automated to produce memory contents on an engineering work station in an almost automatic way.

Remember that the initial state was defined equal to 0 and that the remaining states of the diagram, which has been reduced as much as possible, are number 1, ..., $N_s$-1. For each activity, we have labels that state for a given value, **X**, we have a defined set of output values, **Y**. We extend this label to append the binary value of the state from which the activity starts, $n_1$, to produce a binary number, **L**, which is a concatenation of the binary numbers n1 and **X**. Similarly, for a binary number, **C**, by concatenating the binary value of the state to which the activity points, $n_2$ to the defined output values for that activity, **Y**. That is,

$$\mathbf{L} = \mathbf{X} + (n_1 * 2^{N_I})$$

$$\mathbf{C} = \mathbf{Y} + (n_2 * 2^{N_O}) .$$

Once this is performed for all activities, we should have a set of tuples of numbers which we may sort, using a numeric sort on **L**, thereby producing a table where **L** is the value starting with zero incrementing by one. The values, **C**, are similarly ordered according to its counterpart in **L**. The table so generated may be viewed as follows: **L** is the index (or memory location) of the engine memory and **C** is its contents. Using this process, we have generated the contents of the memory to be used in the proposed processor! In the event our sorted values of **L** have multiple entries for a given value of **L**, the finite state diagram can be reduced.

In the event all possible values of **L** do not occur in the table, then the finite state diagram was not complete and should be completed. Even though the designer is convinced that an activity is unreachable, it is good practice to signal some form of indication that an impossible combination of inputs, coupled with the known state, has occurred.

## Preparation of Finite State Engine Firmware With High Level Languages

For large processors, it is ofttimes awkward to actually draw and label a finite state diagram for a proposed processor simply because it is too complicated. Yet a designer knows the number of inputs, $N_I$, the number of outputs, $N_0$, the number of states. $N_s$, and in terms of some machine digestible algorithm, a proposed production rule, f. It is possible for the designer to write a high level language subroutine (in the fortran language) which takes the following form: using statement labels to represent each of the defined states, it is possible to write using logical IF statements, of the form:

<state>IF(X) THEN Y=f(X,<state>) GOTO <next> ENDIF

<state is the statement label for the state, n, (to which the activity belongs) and <next> is the statement label for the next state, $n_2$. When this is done for all activities, the desired processor can be defined to the computer.

Computer software([4] ) has been written by this author ***to call*** the subroutine written by the processor designer to go through all possible combination of states and input values, **X**, and to calculate an array which, in turn, is passed to routines which carry out the task of placing the memory contents into an ASCII text file written in the Intel hexadecimal object file format (Intel Hex). In the event the firmware is to be placed in a PROM, EPROM, EEPROM, PLA, etc, it is only necessary to ***copy*** this Intel Hex file to a properly

equipped commercial device, a PROM programmer, for the implementation of the firmware.

It should be pointed out that a printed circuit card layout for a finite state engine can be constructed with only the knowledge of the number of $N_I$, $N_O$, and $N_S$ with ***no knowledge*** of the production rule, f. With a proposed processor, the circuit card layout is independent of the algorithm that is used for calculating the output values in the next state. This means that in the event that a mistake has been made in the statement of f, the change in the processor can be done entirely through the use of an engineering work station, using high level languages, thereby producing new memories which are inserted into the device. These require no hardware modification, i.e., no circuit changes whatsoever. This can be done without restriction as long as any of the numbers of $N_I$, $N_O$, and $N_S$ is not ***increased.***

The PROM programmer used by the author is a (DATA I/O Model 29 with Unipak 2B). This is capable of programming in literally thousands of different types of devices to implement the very smallest engines to the very largest. Research and development tasks can be carried out through the use of very modest equipment, indeed.

## Reductions To Practice

So far, the finite state engine approach has been used to implement a number of different processors. The first([5] ) was a device to continuously sample the video signal produced by an RS 170B camera (in real time). The finite state engine was to capture these data and, at the same time, convert the binary values to a modified run length encoded format (thereby compressing the data and storing this in a memory for later use. The second([5] ) of these devices was the counterpart of the video input stage. That is it was designed to continuously read from the serial access memory, decompress the modified run length encoded data and to generate the composite video which met RS 170B standards. Each of these two processors were built out of components that were available from local suppliers at a cost of under $10.

The third application([6] ) was a particularly taxing application of finite state engines. This was used in the production of a data acquisition system for the receipt and orchestration of the analog to digital conversion process in a nuclear resonance imager. Here, signal conditioning, signal transmission, and preparation for use in the computer was done through the use of three finite state engines. It has made it possible to use a PC/AT for the entire process of signal acquisition, image reconstruction and image display. This is carried out with the accuracy and speed comparable to the finest of the commercially available medical imagers. The components required to carry this out, not including digital to analog conversion modules and the computer itself, were assembled for a cost of under $200.

## Discussion

In order to realize the potential capable in this technique we must consider the use of multiple finite state diagrams which are implemented as multiple processors. It should be understood that the output of one processor can be the input to any number of others. This technique is suitable for massively parallel operations with global synchronization. This is possible because of the fact that clock pulses to one device may be fed to any number of others. The fact that additional circuitry has been developed and described elsewhere(7 ) that allow the shaping and lengthening of extraordinarily quick pulses. It should be pointed out that by defining the number of states to be 1, any combinatorial logic can be represented. It is true that this can also be achieved in devices such as PALs, yet, it should be pointed out that, at times, combinatorial logic modules are needed and by using a finite state engine it will assure the designer that the combinatorial logic will execute ***at the same speed*** as other processors.

In processors with multiple finite state engines, the clock speeds need not be the same. This allows for design ***module abstraction.*** One of the important features of the language, LISP, is that it allows for abstraction in function definition. Here we have a technique that gives to circuit design, a design module abstraction that allows a finite state engine to be considered an individual module in yet a larger finite state diagram.

Unlike most other CPU design techniques, the signals are stable except after the rising edge of the clock pulse. This makes simple the act of designing error detection monitoring circuitry in redundant systems.

The fact that the clocks may be asynchronous and controlled by the processor allows (by way of the use of CMOS logic) low powered dissipation in a processor except when it is needed.

Finally, it is proposed that the finite state engines described herein be used as a basis for a new class of VLSI components. Imagine an integrated circuit having input/output connections, power, reset and clock pulses that can be bought as different members of a class of generic devices. Then, through the use of an engineering work station, (a PC and a PROM programmer), alone, a circuit designer with only a simple work station and modest equipment can design and build a large class of custom processors in his own laboratory. This capability is what Prof. Carver Mead thinks will bring about an increase in cost effectiveness in computing. This author will add to this by adding the convenience of performing the design in the designers own laboratory, a larger number of designs will be tried. This will certainly contribute to the development of more efficient and powerful custom processors.

Entire new class of computer designs are possible. Hopefully, these will consist of the use of many different little ones, rather than the microprocessor designers current apparent preoccupation of building ever stronger, ever faster, monolithic (single) devices.

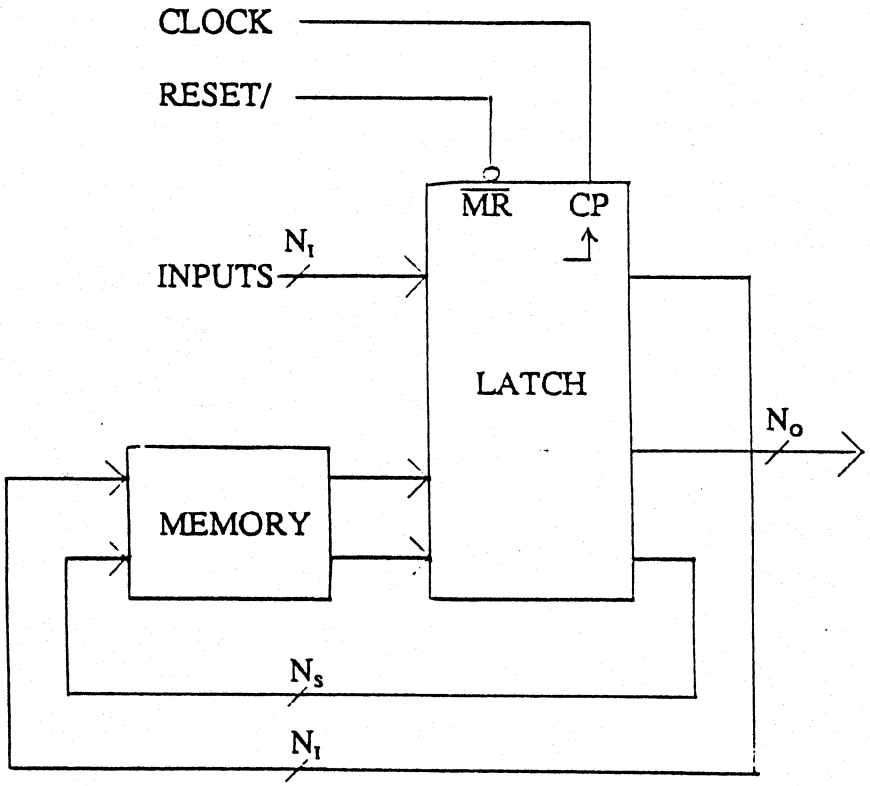


**Figure 1. The Schematic Diagram for a Finite State Engine.**

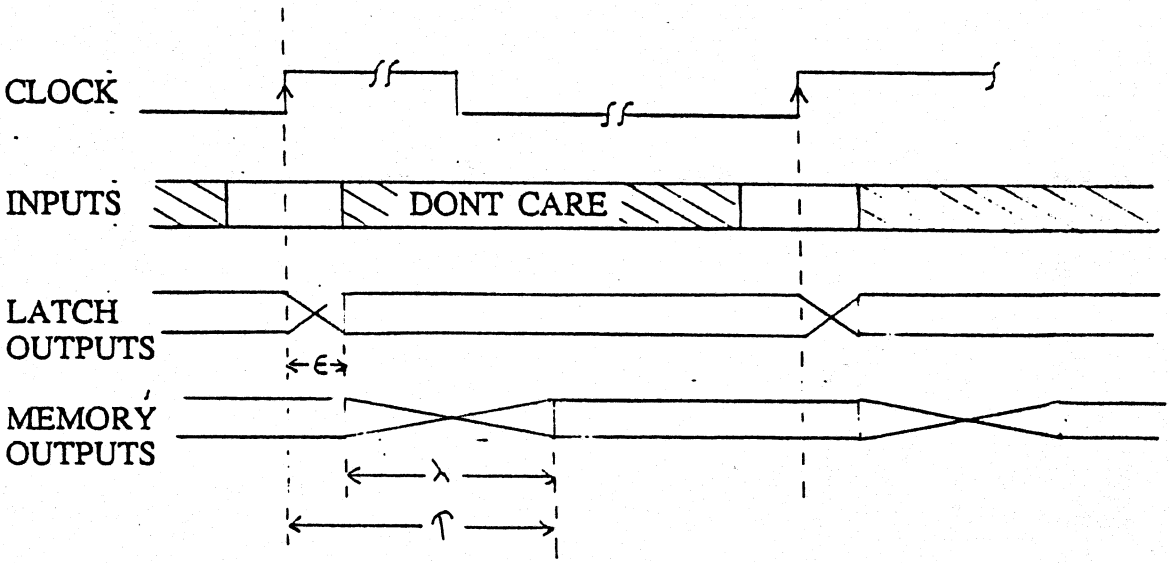


**Figure 2. The Timing Diagram for a Typical Finite State Engine.**

Note: $\varepsilon$ = 6.5 nsec for a 74F374 latch and $\lambda$ < 20 nsec for MB7112Y PROM, Therefore, $\tau$ may be 26.5 nsec!

1. Carver Mead, Forbes, April 4, 1988. Pages 88-93.

2. J. McCarthy and P.J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in Machine Intelligence 4, edited by D. Michie, New York: American Elsevier (1969)

3. J.H. Letcher, Latched Fedback Memory Finite State Engine". Patent Number 4786829. Patent and Trademark Office. (1988)

4. J.H. Letcher, "FLIBSCI, The Fortran LIBrary of Synergistic Consultants Incorporated" (1987)

5. Henry Tromp, MSEE Thesis. University of Tulsa, Pages 20-36 (1987)

6. J.H. Letcher, "The Use of Weiner Deconvolution (An Optimal Filter) in Nuclear Magnetic Resonance Imaging", 74th Scientific Assembly and Annual Meeting, Radiological Society of North America. Chicago (November 30, 1988)

7. Henry Tromp, MSEE Thesis. University of Tulsa, Pages 50-57 (1987)