A Technique for the Design and Construction of Arithmetic and Control Processors

John H. Letcher
Department of Computer Sciences
University of Tulsa
600 South College Ave.
Tulsa, Oklahoma 74104

Introduction

We have seen the development and use of both Complex Instruction Set (CIS) and Reduced Instruction Set Computers (RISC). These CPUs have been made the basis of most of our workstations and personal computers. These machines are constantly being made larger and faster. It would seem that we are driven by an insatiable thirst for such things. However, if we look critically at what we are doing, we would find that we typically have bought far too much computer for most of the jobs that we perform. Upon reflection, we find that it is only a few highly intensive tasks that drive us to buy the more expensive machines. Is there another way?

We have seen the development of parallel processing computers imbedded with sets of identical units communicating within rather constraining architectures. Is this the direction that parallel processing is going to achieve greater performance?

Why not combine these ideas and produce a system with a number of different processors; yet, select the processors so that these are balanced one with the other and that each suited to its own tasks. Some of these may be the general purpose computers that we buy today, but for other tasks, custom processors are needed. Probably, these custom devices have not yet been designed.

This paper was written with two purposes in mind. These are:

1) To show how *one* custom processor is specified, designed and implemented in hardware. This will show that the design of truly custom processors is easy and practical.
2) To show how *sets* of these processors (which may be quite different one from another) can be connected and made to play in concert one with the others with absolute assurance that a type of error called metastability *cannot* occur.

A custom processor that we propose consists of a function module completely surrounded by extremely high speed latches. The custom hardware function module could be build entirely of memories yet the function modules could be *anything* that obeys a simple set of rules. These do not need to be constructed out of the customary digital logic circuits, AND, OR, NOT, etc.

How the function modules (the part that does the work) are implemented in hardware, although interesting, is not the subject of this paper; rather, as long as *any* function unit can accept a stable set of inputs and produce a stable set of outputs at a known time after the inputs become stable, then the function module is suitable for our purposes. The technique which is to be presented herein is to do away with the nemesis of sequential circuits, metastability.

I doubt that many DDJ readers would want to employ the ideas in this document to reproduce a 80486 chip, yet perhaps many would like to design and build a single small inexpensive processor suited to a users needs. As an added benefit, it would be desirable to employ techniques that allow the processor to be incorporated easily in a larger unit that employs a number of other free running devices.

Why Custom Processors?

Analyze the tasks performed by your workstation. If you characterize the tasks performed by the workstation or personal computer class machine, then you could probably divide all of these tasks into two categories: 1) those that take a great deal of time to execute (relatively speaking) or those that require a modest amount of time yet are executed over and over, and 2) all those tasks that do not fall into category 1. Chances are, those tasks in category 2 will be handled well by commercially available CIS or RISC CPUs and that high level language compilers are quite adequate for the task of preparing the executable programs for them (even though the compilers, themselves, may fall into category 1). If you determine the total time required to perform all of the tasks in category 2 you will see that you could have gotten along very well with a small and inexpensive CPU. We see now that it is the category 1 tasks that drive us to want faster devices. Then, why not perform these tasks on separate asynchronous processors that can be designed from the ground up to perform their tasks well and coexist within the same system as the conventional (primary) CPU?

All processors are not well suited to all tasks. It can be shown that many of the commercially available processors are not ideally suited for well known tasks; quite the contrary, in application such as the execution of LISP, FORTH and Postscript programs, CPUs such as the 80386, Sun Sparc and Vax computers are profoundly trounced (by order of magnitude) by the TI Explorer which employs a chip specifically designed *for its own task.* Adding a custom processor to this system was certainly worth the effort.

There are many jobs that we ask our computers to perform that are too small or too simple for large and expensive CIS or RISC CPUs, yet make enormous demands upon system resources. An example is the service requirement of a serial port communicating with another device at 115000 baud*. Here, interrupts are arriving at a rate of 60 µsec per character. Servicing an interrupt every 60 µsec consumes most of a 80386 computer. An ancillary 8088 computer or even smaller control processor would handle the job nicely, producing only one interrupt for the 80386 at the end of a long block or none if the software is status driven. Adding a custom processor could relieve the burden from the primary CPU; its presence would be desirable.

**Even the original IBM serial port boards as well as most later boards use the 8250 Universal Asynchronous Receiver Transmitter (UART). Other boards use the 16450 or equivalent. All of these are pin for pin compatible and software compatible. A clock exists on the board that is divided by the contents of a UART register to produce all of the customarily employed baud rates, for example 9600 baud. This divisor is contained in a register which may be written by a user program. By setting this divisor to 1 the baud rate of 115000 baud is achieved. This produces rather snappy inter processor communication and this is very very inexpensive to implement.*

Some newly designed RISC chips have produced programming headaches. In the attempt to achieve outstanding performance levels, designers are producing devices that are not suited to a broad spectrum of jobs, yet are sold as general purpose devices. Let's look for a moment at one of the newer RISC processors, the Intel i860. This processor is capable of outstanding performance if the software for a task is perfectly prepared to account for the fact that the i860 can perform a pipelined floating point (FP) load (a core instruction) plus a dual pipelined FP add and a FP multiply, *every clock*. If you count FP load as a floating point instruction (which this author does), then it is possible to perform 100 million FP instructions per second (MFLOPS) in a 33.3 Mhz i860. However, no one is achieving anywhere near this number from software produced from high level languages. So what is the problem? One of the answers is that the pipelining structure of the i860 has made it extremely difficult to translate high level languages into code that even comes close to keeping the pipelines full. As a result, performance suffers. Seldom is 10 MFLOPS achieved in practice. This is less than 10% of theoretical performance. Is it possible to reorganize (redesign) the instruction set (and some of the internal data paths) to give better access by high level languages? We believe so, even though it has not been done. Nevertheless, specialized jobs such as performing wavelet and fourier transforms justify the effort to produce software specifically tuned to perform these tasks. If the i860 is relegated to the task of performing only a few important and complicated tasks, then a companion general purpose CPU is totally freed to do other things. Some people feel that the i860 is a failure as a general purpose computer. If this is true, then this can be attributed only to the difficulty of adapting general purpose software to run well on it. No one faults the high quality of the hardware itself. If we were to look at processors like this as individual function modules that we will place, as needed, in a larger parallel environment, then in this role the i860 and other custom function modules will prove to be stunning performers. Furthermore, the possibilities for other custom processors designed for a predefined task appears to be endless.

There are many tasks which conventional CPUs are asked to perform that could be relegated to a large number of separate parallel processors. The interaction protocol could be a simple BUSY-DONE semaphore. Candidates for such units include the calculation of fourier transforms, the calculation of wavelet transforms and carrying out complicated manipulation of graphics pixel values. Imagine preparing a data set of values and placing it somewhere accessible to the new custom processor. A BUSY flag is set and the new processor reads the input values and calculates a set of output values, placing these either on top of or next to the input values; then, the DONE flag is returned. How the new processor carries out its tasks is of *no concern whatsoever* to the requesting processor. The new processor may possess whatever kind of clock it needs and this may be totally unrelated to that used by any other. It may go as fast or as slowly as the function module designer wishes and the host cares only about when the other processor is finished.

Incorporation of custom processors into existing workstations could be accomplished easily. This need not be by the conventional coprocessor route. Additional boards with shared memory with the host CPU seems to offer the most promising method of incorporating a custom processor to perform a specific task. The design and implementation of these processors is within the reach of many DDJ readers.

Custom Processors

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. Herein, we show a way to accomplish the same task without the services of the manufacturing company.

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).

In order to successfully employ these techniques it is necessary to review some basic definitions. Many words are used in different ways by different people. The definitions that will be employed are given below.

Process Abstraction

A lesson to be learned from LISP and FORTH (and their descendants, for example POSTSCRIPT) is the power of function abstraction by allowing definition of new functions or words as time goes on. In terms of a small set of primitive operations, it is possible to define new, more generalized tasks in terms of the already defined ones. Soon, we may forget that the original primitives exist and deal only with the definitions of the new functions. We wish to do the same for hardware entities except that our primitives are a set of discrete asynchronous processors. We now wish to define a set of characteristics of an engine so that we may employ the same techniques to build this module into a larger framework.

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.,

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

Multiplying by 2^{i-1} shifts the value of X_i into the proper position so that all of these terms may be XORed (which is an ADD without carry) to produce the value of X, a number.

The maximum value of X is $2^{N_I} - 1$ and of Y is $2^{N_0} - 1$. The minimum values of X and Y are zero. The states are numbered starting with 0 (for the initial state) and extending through the positive integers as far as needed.

Finite State Diagrams

The action of a proposed processor must be defined, a priori. This definition or 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, S: X:Y, where numerical values for S, 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 $\mathbf{Y}(t+\varepsilon)::=f(\mathbf{X}(t))$ 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. ε 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: 0:X:Y.

The Finite State Equations

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: $\mathbf{Y}(t+\tau) ::= f(\mathbf{X}(t),\mathbf{Y}(t))$.
This states that the outputs of a processor at time, t, plus an increment, τ, 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.

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 that 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][4]

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, τ, 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 or even very high duty cycle). The distance between the rising edge of the pulses, however, is defined to have the value of at least τ. This is represented simply in hardware as shown in Fig.1. This consists schematically of only two devices, a latch and a function module which may be 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 to the appropriate values 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 will be used by us 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 λ, 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, τ, 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 $(t_0 - \varepsilon_1)$ to $(t_0 + \varepsilon_2)$ 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. Furthermore, these may be cascaded to produce pipelines.

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 n_1 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,

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

and

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

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 must 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_O, 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=N) THEN Y=f(X,<state>) GOTO <next> ENDIF

<state> is the statement label for the state, n_1, (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[5] 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, 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_0, 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 workstation, 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_0, and N_S is not *increased.*

Applications of the FSE

The finite state engine approach has been used to implement a number of different processors. The first[6] 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[6] 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[7] was a particularly taxing application of finite state engines. This was used within the production of a data acquisition system for the receipt and orchestration of the analog to digital conversion process in a nuclear magnetic resonance imager (MRI). 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 class computer 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 and analog to digital conversion modules and the computer itself, were assembled for a cost of under $200.

Metastability

Metastability is a type of failure of a circuit that can occur when digital circuits try to capture the value of a digital signal through the use of a clock. An example of this is a latch with a digital input that does not obey the rule that the input must be in a known and stable state for a period of time ε_1 before the rising edge of the clock and a time ε_2 after the rising edge of the clock. When this is not the case the output of the latch can drift into the indeterminate range, or as bad, go into oscillation for an extended period of time. This will cause your processor to get the wrong answer! Engineers calculate the probability that this will occur by a measure, the Mean Time Between Failure (MTBF). A formula for this is[8]

$$MTBF = (\exp ((1 / F_C) - T_D) / G) / 2 F_C F_D T_P$$

where: F_C is the clock frequency.
F_D is the asynchronous data rate of the input.
T_P is the flip flop (latch) propagation delay.
G is the flip flop inverse gain-bandwidth product.
T_D is the total gate delay between the clock edge and any destination flip flops that receive the output data.

Reference 8[9] gives a nice example for a single D type flipflop where the propagation delay equals 7 nsec, the setup time = 1.5 nsec, the hold time = 2.5 nsec, the inverse gain-band width product G = 2.1 nsec max. Using these numbers it is calculated that T_P = 7.0 nsec and T_D = 7 + 2.5 + 1.5 = 11 nsec. When the clock frequency F_C is 8 Mhz and the asynchronous input data rate is 4 Mhz, the MTBF calculates out to be 27 billion years. However, if you double both the clock frequency and the asynchronous data rate, the calculated MTBF falls to 6.9 hours! In our quest to force our processors to go ever faster, we run the risk of failure if we do not account for the possibility of metastability.

A large system consisting of many finite state engines can be viewed as a directed *graph*, in the mathematical sense, with clocks being presented to each FSE and the outputs of one engine being presented as inputs to another. As long as the clocks to each FSE are adjusted so that the rules are followed (and whether or not this is so can be calculated), then *within the system*, the MTBF of the entire system is infinite! Stated in another way, the probability of failure due to metastability is identically zero. Only with the signals coming in asynchronously from the outside does any possibility of metastability occur. A synchronizer for asynchronous input data can be formed by a FSE with a nul function module giving in essence two latches in series. Reference 8 proposes this as a good way to produce a circuit that produces a calculated MTBF in excess of a billion years. This might be sufficient.

Rules for the Design of the FSE

Looking at the FSE inside and outside the halo which the latches on input and output provide, we can define a set of rules which, if obeyed, allow the design of a stable device with well behaved attributes. Only the external attributes are important in the use of this engine. The Inputs -> LFL -> Outputs can be used a representation for an engine. On the inside the function module, F, is bracketed by latches, L.

1) The function module, F, must exhibit the characteristics that within a time period, λ, after its inputs become stable, the outputs of F have been determined and remain so until an input changes. The characteristic time, λ, may be in nanoseconds if F carries out a Floating Point (FP) multiply, or it may be measured in microseconds if F carries out a fourier transform or it may be measured in milliseconds if it carries out a complete image reconstruction of a magnetic resonance image or an ultrasound image.

2) The latch, L, is designed to have the characteristic that its input is sampled sometime within the interval of ε_1 before and ε_2 after the rising edge of the clock

that drives the latch. The times ε_1 and ε_2 should be *extremely* short, preferably measured in picoseconds, and ε_2 should be as close to zero as possible. In the event the latch sees a transition within this brief time interval, the latch should be allowed *to choose* either level it wishes (remember the input was in both states, else there would not be a transition so why should it not choose one or the other). In any case, after a time, ε, the outputs of the latches are set correctly and held absolutely stable until the rising edge of the next clock pulse. The characteristic time, ε, should be short, measured in only a few nanoseconds, at most.

It should be noted that if several FSEs are used in tandem, then the representation Input --> (LFL)(LFL) --> Outputs may be reduced. The interior latches are not necessary (unless needed to introduce a desired time delay) and may be *virtual*, that is, these need not be included in the hardware at all. The characteristic of F_1 F_2 is simply $\lambda_1 + \lambda_2$.

Endogenous Clocking of Processors

It should be obvious to any reader that to multiply 8.7×10^{15} by 1.98×10^{87} should take a longer time than multiplying 3.0 by 0.0. Work is going on by many vendors to try to make the finest of their processors perform each operation in one clock. Even then, why should the multiply by 0.0 take a *full* clock. If a function module can recognize that its task is complete within this time cycle, it should be possible to offer the signal line that states that it is complete to allow this to be the clock begin immediately the next step of a sequence. A properly designed function module could supply the clock to its latches as well as to supply an indication to other processors that its outputs are stable. Dead or idle time in a processor because of a regular clock is no longer necessary.

Conclusions

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[10] 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.

Unlike many 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 (and possibly correction) circuitry in redundant systems.

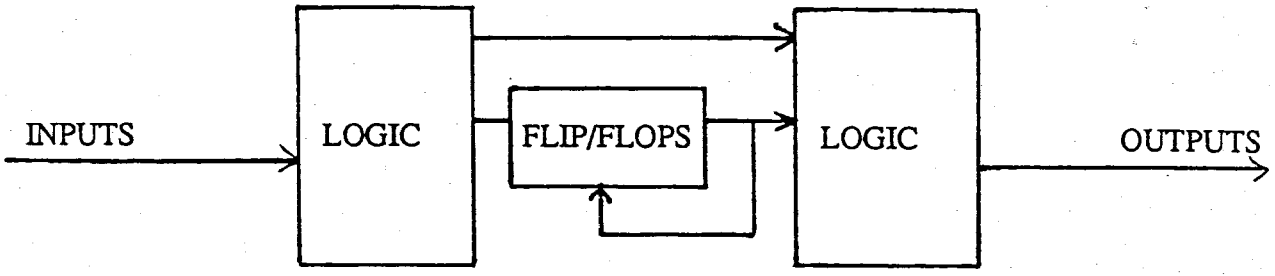
The fact that the clocks may be asynchronous and controlled by the processor allows (by way of the use of CMOS logic) low powered dissipation in a processor except when it is needed.

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.

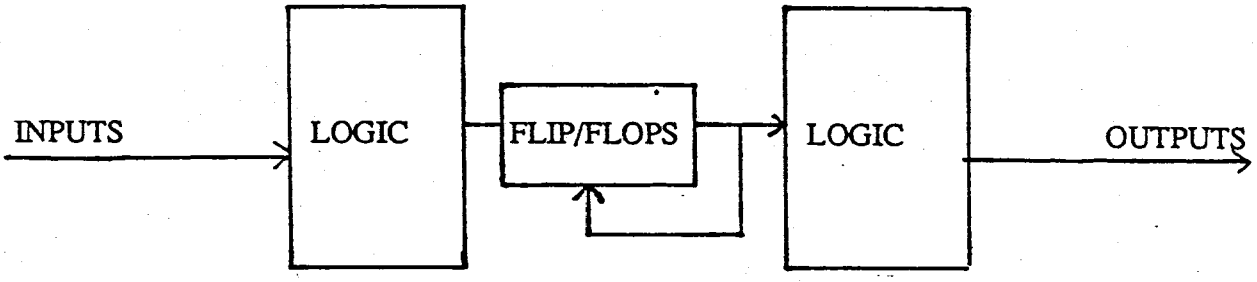
According to the book, state machines come in just two varieties:

1. **The Mealy State Machine**



A state machine with pulse outputs.

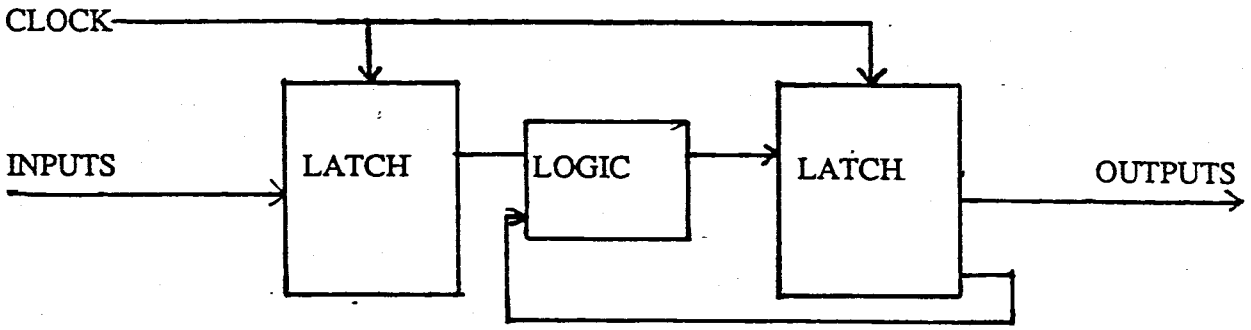
2. **The Moore State Machine**



A state machine with level outputs.

There is yet another:

3. **The Letcher State Machine**



A state machine with level output.

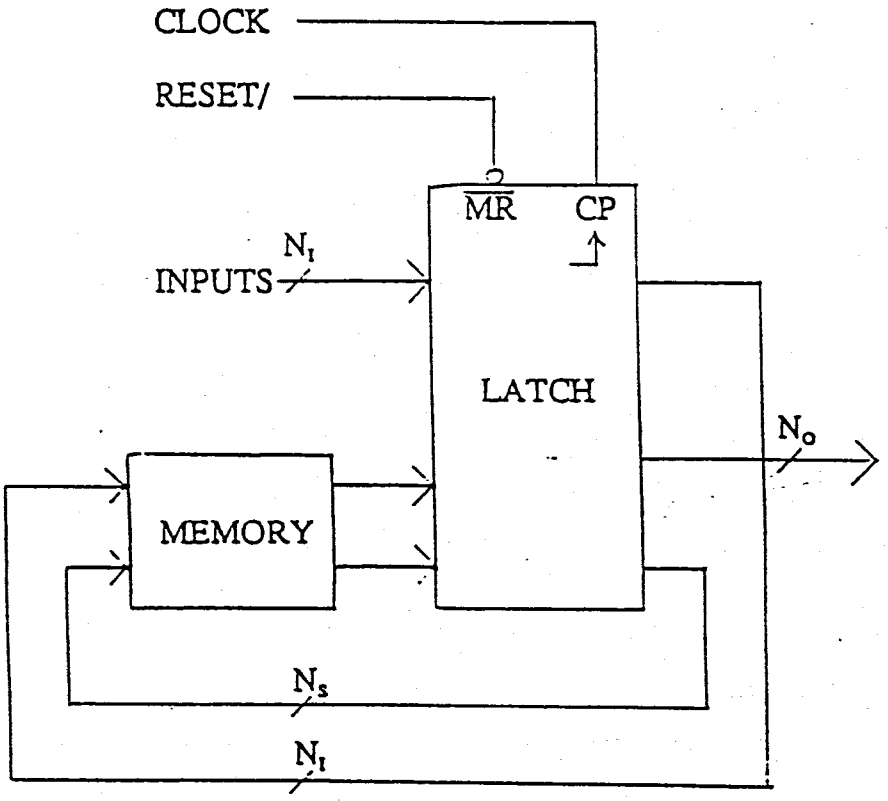


Figure 1. The Schematic Diagram for a Finite State Engine.

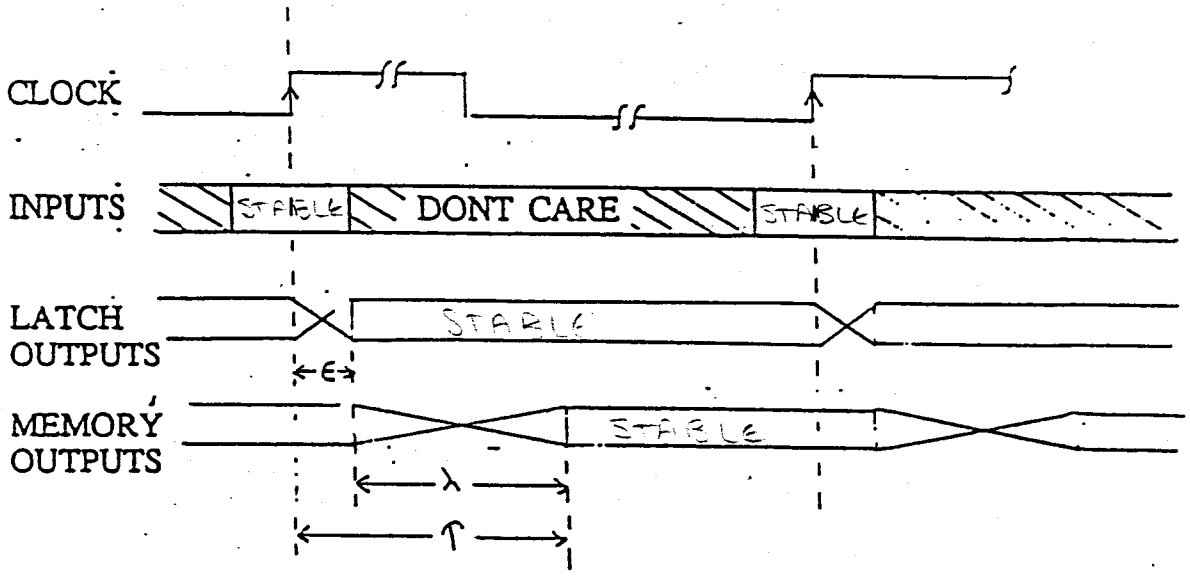


Figure 2. The Timing Diagram for a Typical Finite State Engine.

Note: ε = 6.5 nsec for a 74F374 latch and λ < 20 nsec for MB7112Y PROM, Therefore, τ 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, Latched Fedback Memory Finite State Engine". The Second Symposium on Artificial Intelligence. Norman, Oklahoma (November 3 & 4, 1988)

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

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

7. 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)

8. John Wakerly, "Designers Guide to Synchronizers and Metastability", Parts 1 and 2. Microprocessor. Report Vol 1, Number 1. Pages 4-8 (September/October 1987)

9. Stephen R. Masteller, EDN. Pages 169-174 (April 25, 1991)

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

With the publication (Letcher,1988) and patent application (Letcher, 1987) a new model was presented for finite state machines. This model is different from all of the others as the function module (which may be combinatorial logic, but need not be) is surrounded by "latches" whereas the bi-stable memory devices (which may be latches) of both the Mealy and Moore machines (as well as the intermediate or mixed models) are shown being entered into combinatorial logic. The latter machines use combinatorial logic to modify the machine output signals as well. The Letcher model turned the Moore model inside out. This was not arbitrary and without purpose. A number of distinct advantages were obtained over the previous models. These differences (advantages) include the following:

1. The probability of circuit malfunction of Letcher machines due to *metastability failure* is identically zero as long as certain well defined timing conditions are met. This is not true for the other models.

2. No *hazards* exist in Letcher machines whether these be static, dynamic or essential.

3. The model of the Letcher machine permits the calculation of the *upper bound* on the *clock frequency* and still preserve absolute freedom from metastability failure and deleterious actions due to hazards.

4. The model of the Letcher machine permits total freedom to *slow clocks to any degree*. This may or may not be shared by the other machine models. This is useful in the design of laptop computers to run in low power mode while not being used.

5. The model of the Letcher machine shows where research and development activities could be concentrated, notably the *development of latch circuits* that minimize the effect of the timing restrictions on these machines.

6. Because the maximum clock frequency of a Letcher machine is dependent upon the worst case propagation delay in the function module (e.g., an arithmetic unit), the possibility exists to rethink existing designs with the purpose of reducing only this number at the cost of additional circuit complexity. (normally circuit complexity is minimized to produce the fastest unit possible). It might be possible to produce a faster unit by redesign and making the unit more complicated.

7. Any function module may be used in a Letcher machine as long as the following are true:

   a. the module has digital inputs
   b. the module has digital outputs
   c. the module produces stable outputs at a finite (which is usually small) propagation delay time λ, after all of the inputs have become stable, no matter what the input values are.

This makes it possible to produce, among other things:

a. *fuzzy logic engines* where the module circuity may or may not be digital logic
b. hybrid *(analog)* function modules for real time solution of differential equations in guidance systems
c. *ultra high speed* arithmetic units. Using commercially available components, engines can be built which can be clocked at 400 Mhz.

The Letcher machine *requires* (according to the patent) that the inputs be stable within a short time interval before the rising edge of the clock) plus a short time interval after the rising edge of the clock. In practice, this can be *relaxed* as long as the input latches are designed so that a stable output is obtained even when input levels change during the critical interval. The output level which is chosen by the latch does not matter.

The latch used in Letcher machines should require that the low to high transition of the clock be crisp. There is no apparent reason to try to relax this requirement.

The Letcher machine has no hazards, wither static or dynamic as long as the logic contained with the latches obeys the maximum propagation delay criterion.

All of the machines considered are built from a set of circuit *elements* (transistors, etc). No claim is made by anyone with regard to the novelty of any of these elements. Instead, the use of the elements in ways that match *model templates* make the distinction of one invention from another. Trying to use the argument that each machine is equivalent because a common set of elements is employed is certainly similar to proving the equivalence of a Hereford bull and a Billy goat (after all, each is made up of cells and these are organized into systems that are surprisingly similar).

In summary:

1. *prior art* for pulse and level output synchronous state machines fall into three categories: the Mealy Machine, the Moore Machine and mixed types such as the Signetics/Data I/O Moorly Machine. The Letcher machine is different from any of these, *not because of clocking action* rather because of the placement of the latches (bi-stable memory devices).

2. Clocks for any of these machines all appear to be edge triggered even though almost all of the literature is silent on the issue of clocking type. Whether a clock is edge triggered or "level triggered" *does not change the taxonomy of the machine classification.*

3. Models of all prior art show combinatorial logic used on input signals. This is a clear distinction from the Letcher machine.