MICROPIPELINES

IVAN E. SUTHERLAND

The pipeline processor is a common paradigm for very high speed computing machinery. Pipeline processors provide high speed because their separate stages can operate concurrently, much as different people on a manufacturing assembly line work concurrently on material passing down the line. Although the concurrency of pipeline processors makes their design a demanding task, they can be found in graphics processors, in signal processing devices, in integrated circuit components for doing arithmetic, and in the instruction interpretation units and arithmetic operations of general purpose computing machinery.

Because I plan to describe a variety of pipeline processors, I will start by suggesting names for their various forms. Pipeline processors, or more simply just pipelines, operate on data as it passes along them. The latency of a pipeline is a measure of how long it takes a single data value to pass through it. The throughput rate of a pipeline is a measure of how many data values can pass through it per unit time.

Pipelines both store and process data; the storage elements and processing logic in them alternate along their length. I will describe pipelines in their complete form later, but first I will focus on their storage elements alone, stripping away all processing logic. Stripped of all processing logic, any pipeline acts like a series of storage elements through which data can pass.

Pipelines can be clocked or event-driven, depending on whether their parts act in response to some widely-distributed external clock, or act independently whenever local events permit. Some pipelines are inelastic; the amount of data in them is fixed. The input rate and the output rate of an inelastic pipeline must match exactly. Stripped of any processing logic, an inelastic pipeline acts like a shift register. Other pipelines are elastic; the amount of data in them may vary. The input rate and the output rate of an elastic pipeline may differ momentarily because of internal buffering. Stripped of all processing logic, an elastic pipeline becomes a flow-through first-in-first-out memory, or FIFO. FIFOs may be clocked or event-driven; their important property is that they are elastic.

I assign the name micropipeline to a particularly simple form of event-driven elastic pipeline with or without internal processing. The micro part of this name seems appropriate to me because micropipelines contain very simple circuitry, because micropipelines are useful in very short lengths, and because micropipelines are suitable for layout in microelectronic form.

I have chosen micropipelines as the subject of this lecture for three reasons. First, micropipelines are simple and easy to understand. I believe that simple ideas are best, and I find beauty in the simplicity and symmetry of micropipelines. Second, I see confusion surrounding the design of FIFOs. I offer this description of micropipelines in the hope of reducing some of that confusion.

The third reason I have chosen my subject addresses the limitations imposed on us by the clocked-logic conceptual framework now commonly used in the design of digital systems. I believe that this conceptual framework or mind set masks simple and useful structures like micropipelines from our thoughts, structures that are easy to design and apply given a different conceptual framework. Because micropipelines are event-driven, their simplicity is not available within the clocked-logic conceptual framework. I offer this description of micropipelines in the hope of focusing attention on an alternative transition-signalling conceptual framework.

We need a new conceptual framework because the complexity of VLSI technology has now reached the point where design time and design cost often exceed fabrication time and fabrication cost. Moreover, most systems designed today are monolithic and resist midlife improvement. The transition-signalling conceptual framework offers the opportunity to build up complex systems by hierarchical composition from simpler pieces. The resulting systems are easily modified. I believe that the transition-signalling conceptual framework has much to offer in reducing the design time and cost of complex systems and increasing their useful lifetime. I offer this description of micropipelines as an example of the transition-signalling conceptual framework.

Until recently only a hardy few used the transition-signalling conceptual framework for design because it was too hard. It was nearly impossible to design the small circuits of 10 to 100 transistors that form the elemental building blocks from which complex systems are composed. Moreover, it was difficult to prove anything about the resulting compositions. In the past five years, however, much progress has been made on both fronts. Charles Molnar and his colleagues at Washington University have developed a simple way to design the small basic building blocks [9]. Martin Rem's "VLSI Club" at the Technical University of Eindhoven has been working effectively on the mathematics of event-driven systems [6, 10, 11, 19]. These emerging conceptual tools now make transition signalling a lively candidate for widespread use.

TWO CONCEPTUAL FRAMEWORKS

In the clocked-logic conceptual framework, registers of flip flops operating from a common clock separate stages of processing logic. Each time the clock enters its active state a new data element enters each register. Data elements march forward through successive registers in lock step, each taking a fixed number of clock cycles to pass through the fixed number of registers and intervening logic stages built into the system. The clocked-logic conceptual framework is widely used 1) because it offers a simple way to design computing equipment, 2) because it is widely taught and understood, 3) because parts that operate with clocks are widely available, and 4) because system noise has died away by the time a clock event occurs.

To build the micropipelines described here we must discard the clocked-logic conceptual framework and think instead about a different but equally simple form of control called transition signalling. In return for moving into the transition-signalling conceptual framework, we are rewarded with three new types of flexibility. In hardware design we attain a new flexibility to compose systems from small parts previously designed and tested; in software, we achieve a new flexibility to handle vectors of variable length; and in systems we enjoy a new flexibility to extend system life by replacing isolated parts whenever components with improved speed or cost become available.

It is often hard to discard a conceptual framework. The well-known puzzle shown in Figure 1 illustrates this difficulty by asking us to draw four straight lines through nine dots without lifting our pencil from the paper. Our natural conception of figure and ground in looking at this puzzle suggests that the lines to be drawn should stay within the square of dots, a conceptual framework that renders the task impossible. The puzzle can be solved only by drawing outside the boundary of the dots.

Similar difficulty in discarding a conceptual framework can be seen in the design of FIFOs. Conventional wisdom in the clocked-logic conceptual framework says that each stage of a flow-through FIFO should have a clocked register that feeds its output to the input of the next stage. Now recall that a FIFO must be elastic; it must be able to store a variable amount of data; and if it has a fixed number of stages, some of them may be unoccupied. Continuing with the clocked-logic conceptual framework, a "full" or "empty" clocked flip flop for each stage is required to make the FIFO elastic.

Each stage must also have logic involving the state of its full or empty flip flop and the states of other stages to decide when to capture new data. One simple control rule operates as follows: a) Each stage captures new data and sets its full flip flop to the full state whenever

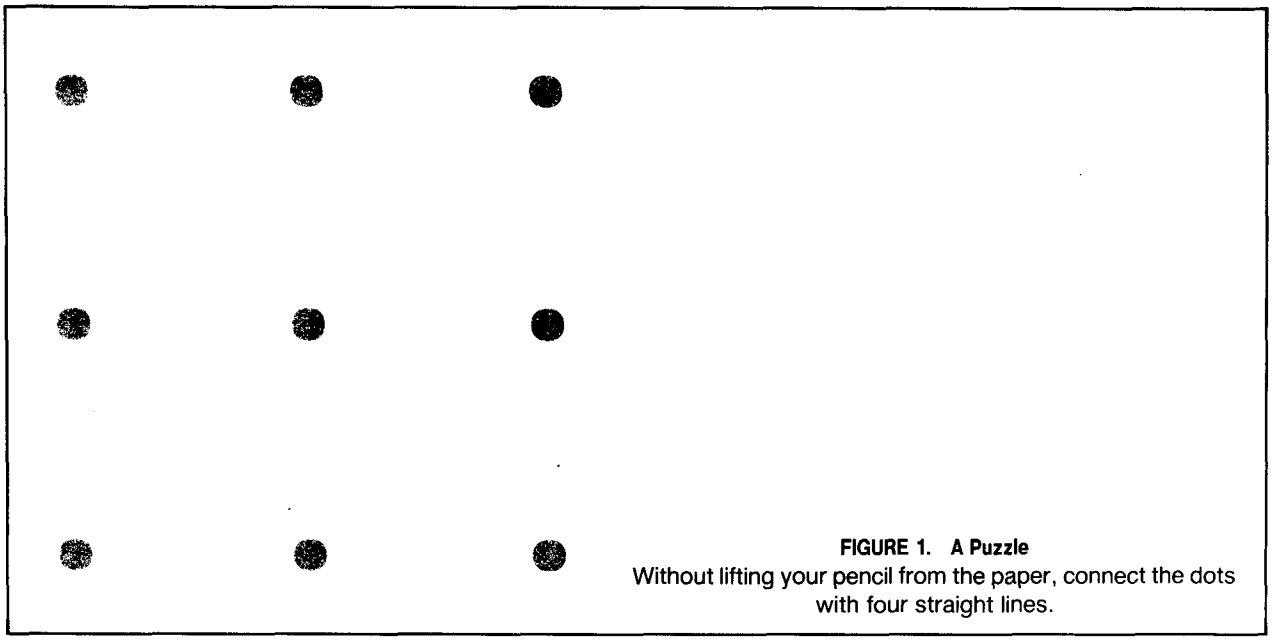

FIGURE 1. A Puzzle
Without lifting your pencil from the paper, connect the dots with four straight lines.

it is empty and its predecessor is full. b) Each stage sets its full flip flop to the empty state whenever it is full and its successor is empty. This rule delivers output data only on alternate clock cycles. More complex rules for the control of synchronous FIFOs get better performance by looking ahead many stages to decide if an entire block of data can move forward during the forthcoming cycle. The clocked-logic conceptual framework itself creates this complexity, because all registers must act together at once; any that fail to act now must suffer a complete cycle of delay for their next opportunity.

The clocked-logic conceptual framework is poorly matched to FIFO design for another reason as well: FIFOs often connect senders and receivers that have separate clocks. The difficulty of designing a FIFO with separate clocks at input and output is strikingly evident when one asks whether to use the input or the output clock to control the internal stages. There is no natural point in the FIFO to transfer control from the input clock to the output clock. Should the transfer occur near the beginning, at the middle, or near the end of the pipeline? Why?

If conventional clocks are used at the input and output of a FIFO and the two clocks are separate, then arbitration or synchronization between these two clocks is required somewhere in the design. Arbitration or synchronization is necessary because the data must pass from control by the input clock to control by the output clock somewhere, even though the phase relationship between the clocks is unknown and variable. There is always some phase relationship between the separate clocks that violates the setup or hold time requirements of some latch or flip flop.

The fact that arbitration or synchronization is required somewhere in a clocked FIFO introduces a host of problems [1]. It is not possible to make an arbiter or synchronizer that is perfectly reliable; instead one must design a circuit for which the probability of failure is so low as to be unimportant. Sadly, although the problems inherent in synchronizers and arbiters have been known for many years, synchronizer failures still cause difficulty, and remarkably, discussions of arbitration and synchronization are largely absent from the descriptions of FIFOs now on the market. Solutions to the inherent synchronizer problems are left to the users.

The internal stages of the micropipelines I shall describe here get their timing signals neither directly from the input control signals nor directly from the output control signals. Rather, each internal stage captures a new data value whenever its successor stage has accepted the present value and its predecessor stage has the new data ready. Each stage operates at its own pace, using control information only from adjacent stages. Letting each stage operate separately, without a common clock, avoids the need for arbitration and simplifies the design. Although the design of FIFOs and other micropipelines is very difficult within the clocked-logic conceptual framework, it is easy once one abandons that framework in favor of transition signalling, as I shall do throughout this lecture.

TRANSITION SIGNALLING

In transition signalling any transition, either rising or falling, has the same meaning, as illustrated in Figure 2; either kind of transition is called an event. As indicated in the figure, and suggested by its name, transition signalling avoids distinguishing between the two types of transitions even though they may look quite different. In effect, all responses to transition signals are edge-triggered, and are triggered on both rising and falling edges. Because transition signalling uses both rising and falling edges as trigger events, it may offer twice the speed potential of conventional clocking.

FIGURE 2. Two Equivalent Transistors
Rising and falling transitions on signalling wires have the same meaning. They are called events.

Transition signalling avoids assigning meanings to the absolute high or low state of control signals. I will use the state of a control signal only relative to the states of other related control signals; the state of a control signal may be the same as or different from that of another, but its absolute state will never matter. Because the absolute state of a transition control signal is unimportant, there is no need to return it to some neutral state between events. By avoiding such returns to a neutral or low state, transition signalling saves the time and energy costs of the return transition, as well as the design confusion of an unnecessary event. Transition signalling is much like non–return-to-zero (NRZ) magnetic recording.

Many people find it difficult at first to grasp the notion that both rising and falling edges should have the same meaning. This is not surprising because a change in conceptual framework is required. Most people are accustomed to differentiating high and low levels and to a clock that returns to a neutral state between actions. But even though it may seem hard at first, the transition-signalling conceptual framework quickly becomes very easy to use. Abandoning the clocked-logic conceptual framework in favor of the transition-signalling conceptual framework can provide real advantages in speed and simplicity that are particularly striking in micropipelines.

Transition signalling circuits must be symmetric with respect to the high and low states of control signals, since both rising and falling transitions have the same meaning. Look for this symmetry throughout this lecture. Notice also that my descriptions of circuit action speak of control signal levels only in relative terms as being the same or different rather than in absolute terms as being high or low, again to preserve symmetry. I use conventional levels, high or low, only for data values. The symmetry of transition signalling provides appealing simplicity in complementary metal oxide

semiconductor (CMOS) circuits because it fits well with the symmetry of the complementary transistors from which CMOS circuits are built.

THE TWO-PHASE BUNDLED DATA CONVENTION

If a sender and a receiver communicate using transition signalling, there will be two control wires and many data wires between them, as illustrated in Figure 3. The data wires carry conventional high or low states to convey true or false Boolean data. The sender places a data value on the data wires *and then* produces an event on its control wire, called "request," to indicate that valid data are available. In some cycles the request event will be a rising transition and in some it will be a falling transition; we make no distinction between them. The receiver accepts the data *and then* produces an event on its control wire, called "acknowledge," to indicate that the data have been accepted. The three events, data change, request event, acknowledge event, always follow in cyclic order, as illustrated in Figure 4. Successive cycles may take different amounts of time, as suggested by the difference in length of the cycles in the figure. Sometimes names other than "request" and "acknowledge" are used for the two control wires [7]. You may wish to compare this protocol to the non-overlapping clock protocol of Figure 5.

Seitz [13] describes the protocol of Figure 4 that we have come to call the two-phase bundled data convention. The "two-phase" part of this name indicates that only two phases of operation are distinguished: the sender's active phase and the receiver's active phase. An event terminates each phase: the request event terminates the sender's active phase, and the acknowledge event terminates the receiver's active phase. The sender is free to change the data during its active phase and makes an event on the request wire after it has made the data valid; it must then hold the data constant during the receiver's active phase. The "bundled data" part of this name indicates that the data wires and the request signalling wire must be treated as a bundle; delays in data transmission must be less than delays in transmitting the request event lest the request event reach the receiver prior to valid data. The ac-

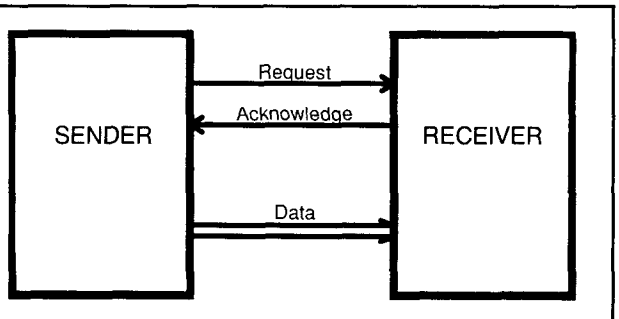

FIGURE 3. A Bundled Data Interface

In addition to an arbitrary number of data wires, an interface following the two-phase bundled data convention has two signalling wires here called "request" and "acknowledge." The sender puts valid data on the data wires *and then* produces an event on the acknowledge wire. The receiver takes the data *and then* produces an event on the acknowledge wire. The request and acknowledge wires are sometimes given other names.

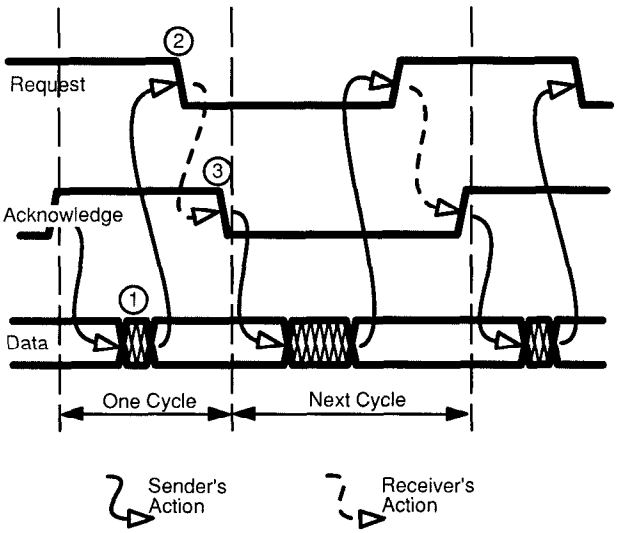

FIGURE 4. The Two-phase Bundled Data Convention

The three events per cycle in the two-phase bundled data convention occur cyclically in the sequence shown. They are: ① During the sender's active phase, shown with solid arrows, the sender may change the data at will. ② After the sender has established correct data values, it ends its active phase by producing the request event. During the receiver's active phase, shown with dotted arrows, the sender must hold the data constant. ③ After the receiver no longer needs these data values, it ends its active phase by producing the acknowledge event. Either phase may last for as much or as little time as its controlling unit decrees. Note that request and acknowledge events alternate.

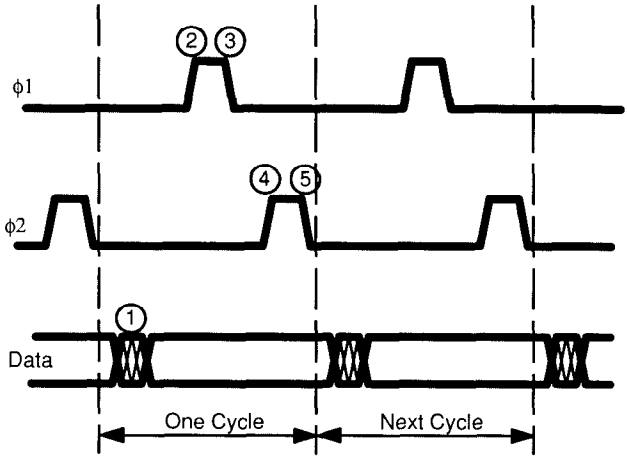

FIGURE 5. Nonoverlapping Clocks

Conventional non-overlapping clocks, $\phi 1$ and $\phi 2$, require 5 events per cycle. They are: ① data change, ② $\phi 1$ rises, ③ $\phi 1$ falls, ④ $\phi 2$ rises, and ⑤ $\phi 2$ falls. Compare this with the three events per cycle of Figure 4.

knowledge wire need not be bundled.

In addition to several data wires, an interface using the two-phase bundled data convention requires two control wires. One may think that this is more expensive than a conventional clocking system, which requires only a single clock wire. The two wires, however, serve to replace not only the clock wire, but also at least two additional wires that would be required between stages in a clocked system to make the pipeline elastic.

EVENT LOGIC

Control circuits for transition signalling are built out of modules that form various logical combinations of events. Here are a few samples:

The exclusive or (XOR) circuit acts as the OR element for events. When either input of an XOR circuit changes state, its output also changes state. Thus an event received on either the first input OR the second input of the XOR will produce an output event. For more than two inputs, XOR generalizes to parity; parity circuits act as the multiple-input OR for events. We use the standard XOR logic symbol with two or more inputs to represent these OR elements for events. Such elements are sometimes called MERGE elements, because they merge two or more event streams.

The Muller C-element [8], for which I will shortly show circuits, acts as the AND element for events. When both inputs of a Muller C-element are in the same logical state, the Muller C-element's state and its output are copies of that state. When the two inputs differ, the Muller C-element uses internal storage to retain its previous state and hold its output unchanged. Thus only after an event takes place on both of its inputs will a Muller C-element produce an event at its output. The Muller C-element generalizes easily to three or more inputs, requiring that all of them reach a new logical state before copying that new state as output. We use the standard AND logic symbol with a large C inside it to represent Muller C-elements that implement logical AND for transition events. Such elements are sometimes called RENDEZVOUS elements, because they act only after all input events have arrived.

In CMOS an appealingly simple dynamic implementation of the Muller C-element is possible, as illustrated in Figure 6. This circuit uses the electrical capacitance of an internal node as the storage element required in the Muller C-element. If a static Muller C-element is required, the node capacitance must be augmented with static logic, one form of which is illustrated in Figure 7.

Although the absolute state of a transition signal does not matter, its state relative to other related signals does matter. Thus it is sometimes important to invert transition signals. We use "bubbles" on inputs or outputs of logic symbols to represent such inversions, as illustrated in Figure 8. Every loop around which events flow must contain an odd number of inversions. Such loops are, in effect, oscillators whose oscillations are coordinated with those of other loops by the actions of Muller C-elements or other modules at loop junctions.

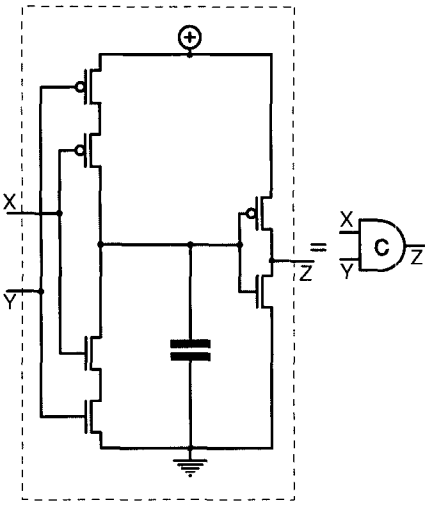

FIGURE 6. A Dynamic Muller C-Element

In CMOS the Muller C-element has a particularly simple dynamic implementation that uses the electrical capacitance of an internal node as the storage element. Transistors to initialize the Muller C-element during master clear are not shown.

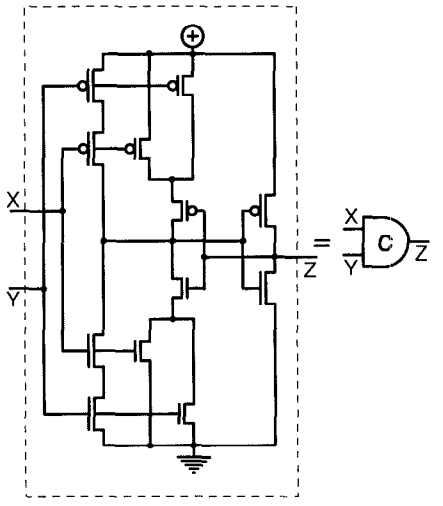

FIGURE 7. A Static Muller C-Element

Replacing the capacitor of Figure 6 with transistors as shown produces a static Muller C-element. In an integrated circuit layout, the transistors shown here with smaller symbols can be very narrow, since they serve only to retain an already-established value. This circuit generalizes in an obvious way to three or more inputs.

Figure 9 shows block symbols for some other useful event logic circuits that implement elemental operations, some of which are familiar to programmers. The TOGGLE circuit produces events alternately on its two outputs in response to events at its input; the first event after some master clear signal and every other subse-

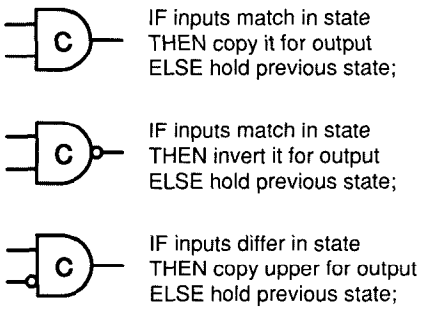

FIGURE 8. Muller C-Elements with Inverters

Muller C-elements contain storage to hold a previous state on some input conditions. When inverters are included in input or output wires, as indicated by the bubbles in this figure, the actions are as listed. Muller C-elements provide the AND function for events.

quent event pass through it to the output with the dot. The SELECT module steers an incoming event to one output or the other depending on the value of a data input; it serves for testing the Boolean condition in conditional expressions. The Boolean value must be available before the incoming event that it steers, a requirement similar to the bundling condition in the protocol of Figure 4. The CALL element remembers which of its inputs most recently received an event, and returns an event to the matching output terminal after a called procedure has finished. The memory in the CALL element serves the role of subroutine return address. The CALL element operates properly only if each call completes before a subsequent call occurs. The ARBITER decides cleanly between two events whose arrival sequence is unknown, producing a grant event for only one of them even if they arrive at very nearly the same time. Like a semaphore in programming, it delays subsequent grants until after receiving an event on the done wire corresponding to an earlier grant so that only one grant at a time is ever outstanding. An ARBITER can be connected directly to a CALL element to permit two entirely independent processes to call on a single shared procedure.

CONTROL FOR MICROPIPELINES

A string of Muller C-elements with inverters interposed, as illustrated in Figure 10, is the only logic required to control the micropipelines described in this lecture. I find it remarkable that the only distinctions between the forward and reverse direction of the pipeline to be found in this circuit are the delay required for bundling and an inversion in the reverse signal path. Observe in Figure 10 that a request and an acknowledge signal pass between adjacent stages of this control. Data wires also pass between stages, as I shall shortly describe, but these are not shown in Figure 10. At each interface between stages the request and acknowledge signals and the data values follow exactly the two-phase bundled data convention of Figure 4.

Notice also in Figure 10 that the request and acknowledge wires at the input interface are identical to those at the output interface. The similarity of the signalling form at the input and output ends of the control ensure that any number of such control systems, even ones that differ markedly in raw speed, will operate properly when connected in series, albeit at the speed of the slowest. This composability of micropipelines makes it easy to assemble long signal-processing pipelines; one simply connects the request, acknowledge, and data wires at the output of one micropipeline to the corresponding wires at the input of the subsequent micropipeline. The composite control is just a further repetition of the control system illustrated in Figure 10.

One way to see how the control circuit of Figure 10

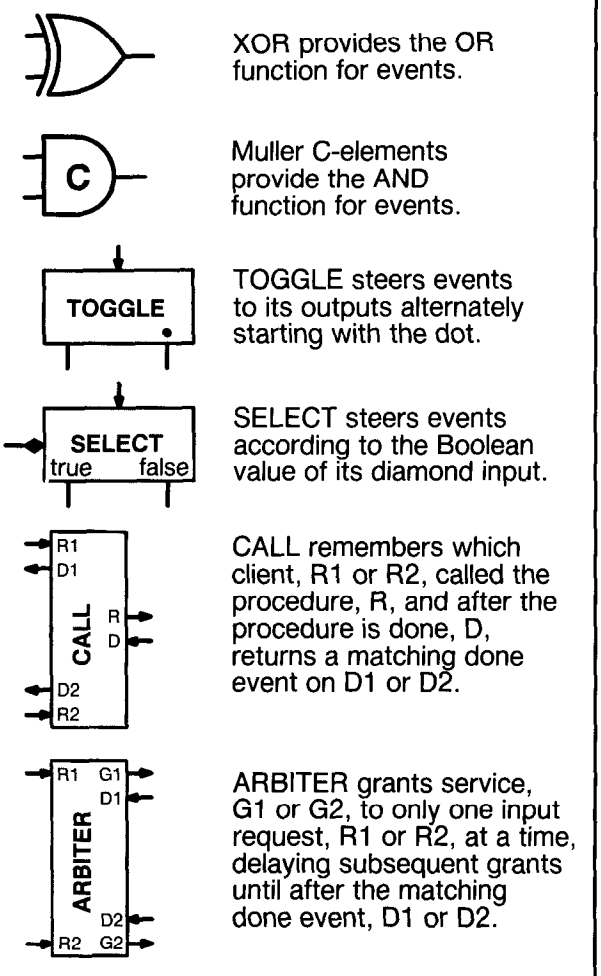

FIGURE 9. Logic Modules for Events

Modules of 10 to 100 transistors can perform useful logical functions on events. The modules whose symbols are shown provide the functions indicated. Note the similarity of these functions to the basic structures used in programming. One might think that the arbiter would require 8 terminals, since the request signals at the left seem to lack corresponding acknowledge signals. Either the grant or the done signals are used to acknowledge incoming requests, depending on the application.

works requires us to focus on its behavior as a series of loops around which events flow. There is a single inverter in each loop, and so each loop will oscillate. The Muller C-elements coordinate the oscillations in adjacent loops. This view makes it easy to see that the request and acknowledge events at each interface must alternate.

Another way to see how this circuit works requires us to focus on the state of each Muller C-element relative to the states of predecessor and successor Muller C-elements. Remembering the behavior of a Muller C-element with one inverted input, we can easily see that each stage of the control of Figure 10 follows a very simple stage state rule:

IF predecessor and successor differ in state
THEN copy predecessor's state
ELSE hold present state.

This stage state rule makes the control system stable both when all stages are in the same state and when alternate stages are in opposite states. The condition in which all control elements are in the same state corresponds to an empty pipeline, and the condition in which alternate stages are in opposite states corresponds to a filled pipeline. There are other stable conditions with stages near the input end in the same state and stages near the output end in alternating states; these conditions correspond to a partly filled pipeline. The stage state rule also makes the control system unstable in some states; such unstable states change immediately as events propagate through the stages of the pipeline. To initialize a micropipeline to the empty condition, its Muller C-elements may all be set to the same state by a master clear signal. I have omitted the reset circuits required to do this from Figures 6 and 7.

The stage state rule, described above in IF THEN ELSE form, is the digital equivalent of the differential equations that describe ocean waves and electromagnetic waves. In a wave equation a time derivative, in this case copy predecessor's state, is set equal to a space derivative, in this case IF predecessor and successor differ in state. Like the rules of physics described by the differential wave equation, the stage state rule results in wave propagation.

Like ocean and electromagnetic waves, events can propagate in either direction through the control of Figure 10. If all Muller C-elements are initially in the same state, an event at the input end of the control will propagate forward through the control from input to output. If the control elements are initially in alternate states, an event introduced at the output end of the control will propagate backward through the control from output to input. It is interesting that so simple a circuit should exhibit wave propagation in both directions.

Both forward and reverse propagation of events in the control system of Figure 10 are useful in controlling micropipelines. Forward propagation of events through the control circuit will force information forward through the micropipeline, much as an ocean wave pushes a surfer toward the shore. Reverse propagation will sweep empty data spaces created at the end of the pipeline back through occupied sections toward the beginning. Empty spaces move through occupied sections much as holes move in a semiconductor or air bubbles rise through water.

AN EVENT-CONTROLLED STORAGE ELEMENT

This section introduces a storage element suitable for use with a transition signalling control system. In order to make these circuits easier to understand, I will use a switch symbol to represent any one of several configurations of transistors, one of which is shown in Figure 11. As you can see in the figure, the transistor implementation of this switch makes use of both the true and the complement form of its control signal, C and ~C, which implies an inversion of the control signal not shown explicitly in the figure. Notice that the circuit is entirely symmetric with respect to high or low values of its control signal, selecting one input when its control is high and the other when its control is low. I will always draw such switches in the position they assume when their control signals are low.

A suitable storage element for use with a transition

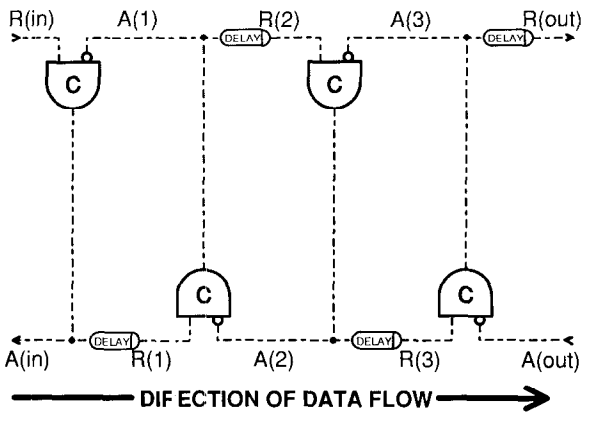

FIGURE 10. Control Circuit for a Micropipeline

With data paths omitted, the control circuit for a micropipeline is a string of Muller C-elements. In this figure one of four identical stages is shaded and alternate stages have been drawn upside down. At the input and output to each stage there are request, R(n), and acknowledge, A(n), signals. Inverters in the acknowledge paths are represented by "bubbles" at one input of each Muller C-element. The delays shown explicitly here may not be required for simple data paths. Notice that each loop in this circuit contains exactly one inversion, the bubble, and is therefore an oscillator. The Muller C-elements retard the oscillation in each loop to coordinate it with the actions of adjacent loops. In this and other figures, dotted wires carry event signals.

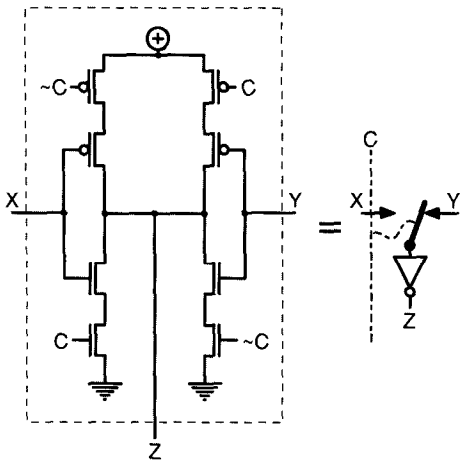

FIGURE 11. Circuit for the Switch Symbol

The double-throw switch symbol at the right of this drawing represents the transistor circuit shown inside the dotted line. When the control wire, C, is low, the output terminal, Z, is controlled by the Y input, as shown. When the control wire is high, the switch flips to the X input. The output of this form of switch is controlled by its selected input, but inverted in value. Other implementations of such a switch using pass transistors are also possible.

signalling control system must respond to transition events. Unlike a conventional latch in which the "high" and the "low" state of the clock signals can perform different functions, an event-controlled storage element must give similar responses to rising and falling transitions. Suitable circuits that use two control wires called "capture" and "pass" are illustrated in Figure 12. Each of these two circuits uses two latches side by side and activates them alternately. In the circuit with only three inverters, the output inverter is shared between the two latches. Notice that because of the inverters implied in the control of the switches shown, both of these circuits are entirely symmetric with respect to high and low values of their control signals. You may wish to compare these circuits with the conventional D flip-flop circuit shown in Figure 13.

The behavior of an event-controlled storage element is easy to describe using only the relative states of its two control signals. When its two control signals are in the same state, the condition shown in Figure 12, the event-controlled storage element is transparent and delivers its input data directly to its output, not acting as a storage element at all. You can see a path through the switches and inverters leading directly from input to output. When its two control signals differ in state, one or the other of the switch sets will be flipped from the position shown in Figure 12. As you can imagine from the figure, if one of the switches is flipped, a loop is formed containing two inverters. Such a loop captures and retains the data value. If one switch is flipped to form such a loop, no path exists from input to output, the event-controlled storage element is rendered insensitive to changes on its data input terminal, and it reports at its output only the data captured in the loop.

The behavior of an event-controlled storage element can also be described in terms of events. Let us assume that the event-controlled storage element is initially transparent, as it is shown in Figure 12 and that the capture and pass control signal events always alternate. An event on its capture control wire flips the two switches to which the capture wire is connected, and thus causes the storage element to capture and hold the data value then passing through it. This event isolates the output value of the element from changes at the

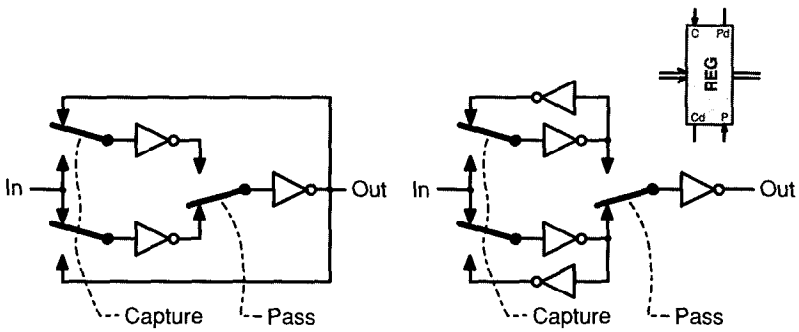

FIGURE 12. Event-Controlled Storage Elements

An event-controlled storage element responds to events on its two control wires, called "capture" and "pass" in this drawing. Two different configurations are shown. The form on the right, with five inverters, is slightly faster than the form on the left, with only three inverters, because its feedback paths contain only one switch rather than two. After master clear the switches will be in the position shown, making a direct connection without loops between input and output, a state in which the storage element is said to be transparent. Storage elements of either type are formed into registers just as are flip flops by connecting their capture and pass control wires in parallel. The register symbol includes control outputs, Cd and Pd, which are amplified, and thus necessarily delayed, versions of the control input signals, C and P. Cd and Pd, named for "capture done" and "pass done," deliver output events after the register has done its action.

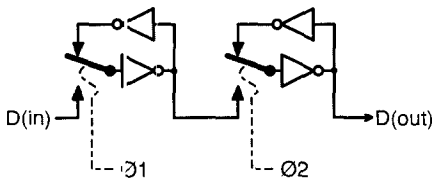

FIGURE 13. Conventional D Flip-Flop

A conventional D flip flop is controlled by non-overlapping clock signals $\phi 1$ and $\phi 2$ illustrated in Figure 5. Compare this circuit to the event-controlled storage element of Figure 12.

element's input but does not change the output value. A subsequent event on the pass control wire flips the other switch, returning the element to the transparent state, permitting the next data value to appear as its output, and possibly changing its output value. After each event on the element's pass control wire a new output value appears. This is exactly the behavior required to make micropipelines with the control system already described.

Event-controlled storage elements are connected in groups to form event-controlled registers. Each such register consists of a number of event-controlled storage elements with their capture and pass control wires connected in parallel. Because the capture and pass wires drive many transistors, suitable amplifiers are included in the register design. These amplifiers have some unknown delay, and there is further delay introduced by the physical length of the wires. I therefore include pass done, Pd, and capture done, Cd, event outputs on every event-controlled register, as shown on the register symbol in Figure 12. Events on these outputs follow exactly the events on the capture and pass control inputs, but are delayed to account for the amplification and wiring delays in the register. The explicit done signal outputs permit subsequent actions to be further delayed if required. Such a delay might be needed if the register is composed from simpler parts, as we shall shortly see, or performs some side effect. It is interesting to note that if two or more event-controlled storage registers are connected so that their data paths are in series and are provided with the same control signals, the result is indistinguishable, except for overall delay, from a single event-controlled storage register.

If wide words are involved and small size is required rather than high speed, one may use the circuit of Figure 14 as an event-controlled storage register. This circuit uses only a single latch per bit, but requires extra equipment for control. The extra control equipment is required because the latches have only a single control wire in which both capture and pass events must flow. Naturally, the delays introduced by the extra equipment also delay the capture done and pass done control outputs.

The operation of the circuit of Figure 14 is easy to understand. The XOR circuit shown at the top of the figure merges the capture and pass control events from the two control inputs onto one wire. Because capture and pass events alternate, each capture event will make the output of the XOR high, flipping the switches from the position shown, and causing the latches to capture data. Each pass event will make the output of the XOR low, returning the switches to the position shown, and making the latches transparent again. The TOGGLE module shown at the bottom of the figure separates the capture and pass events on the common wire onto the "capture done" and "pass done" control outputs labeled Cd and Pd in the figure. The toggle suffices for this purpose because capture and pass events alternate. The capture done and pass done control outputs indicate completion of any internal delays involved in the XOR and TOGGLE modules and in driving the control wire for the many latches in the register.

MICROPIPELINES WITHOUT PROCESSING

A micropipeline with no processing in it, which is a FIFO, can be built by combining the control of Figure 10 with the storage registers of Figure 12 or Figure 14. A set of event-controlled storage registers in series

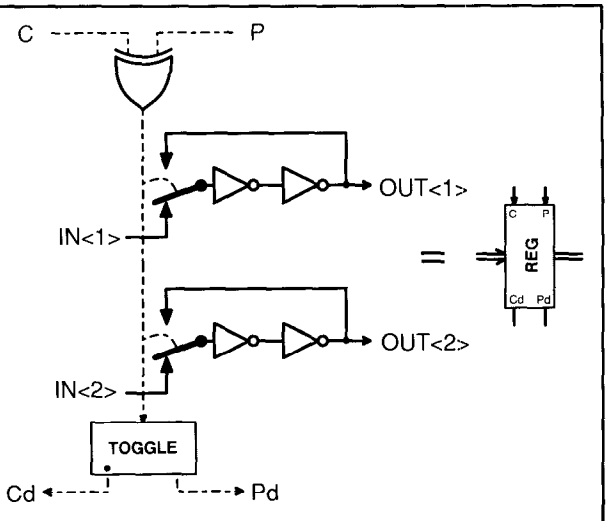

FIGURE 14. Latches Used as an Event-Controlled Storage Register

An event-controlled register made from ordinary latches requires an XOR module and a TOGGLE module for control. A 2-bit register is shown; dashed wires carry events. Capture and pass events arrive alternately at the separate control inputs, C and P, but the XOR merges them onto one wire. At the XOR output, each capture event becomes a rising transition in the latch control wire and flips the switches, causing the latches to capture data. Each pass event becomes a falling transition in the latch control wire and flips the switches back to the position shown, making the latches transparent again. The TOGGLE module separates the capture and pass events back into two separate output paths, Cd and Pd, after the register has done its action. This circuit is slower than the event-controlled register of Figure 12 and delays its output events, Cd and Pd, accordingly, but except for delay provides exactly the same function.

serves as its data path while a string of Muller C-elements serves as its control, as illustrated in Figure 15. Each event-controlled storage register uses the control signal from its stage of the control as its capture control signal, and the control signal from the successor stage as its pass control signal. When this FIFO is empty, all of its storage registers are transparent, and so a path exists through it directly from its data input terminals to its data output terminals.

I have arranged the layout of Figure 15 not only to make it easy to read but also to suggest a layout for an integrated circuit implementation. The Muller C-elements are located at either ends of the registers, just as shown, so that control signals zigzag across the chip. The wires that control the registers are driven from one side of the register and are used to control the Muller C-elements of adjacent stages at the other side of the register. Because the control signals for the register must be amplified to drive all the switches in the many storage elements involved, and because the wires that carry control signals across the register are long, there is always some delay in controlling the register. The arrangement of driving registers from one side and sensing their control signals at the other side accommodates not only the delay in the driving amplifiers but also any delay in the wires themselves.

If no processing is required in the micropipeline, i.e., for a FIFO, the simpler data path circuit of Figure 16 will serve [17]. In this circuit the side-by-side latch configuration of the event-controlled storage element is extended between stages. The two separate data paths are brought together again only at the output end of the FIFO by an output selector switch very similar to that used in the event-controlled storage element. The first, third and other odd-indexed data values pass through the upper data path while even-indexed values pass through the lower data path.

Look at all the symmetry in Figure 16. Except at its output, shown at the right of the figure, there is no distinction at all in its data path between the forward and reverse directions of the FIFO. It seemed at first to me that this must indicate a flaw in its design. There is no flaw; the circuit of Figure 16, though unconventional, works well. Abandoning the conventional notion

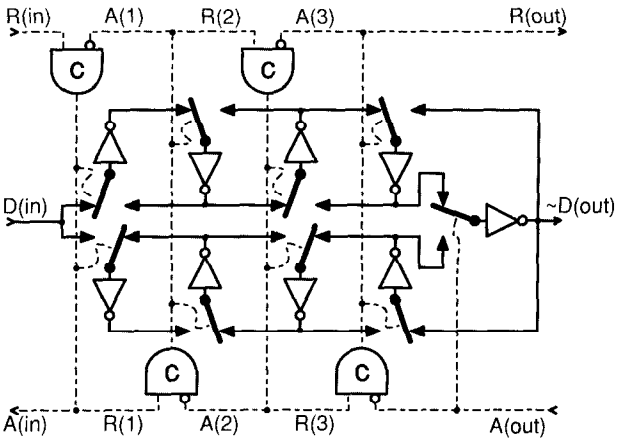

FIGURE 16. A FIFO Circuit

If no processing is required, as in a FIFO, the event-controlled storage elements in Figure 15 can be replaced with this simpler circuit [17]. In this figure, dashed lines carry control signals, solid lines carry data values, and four stages are shown. The switches are drawn as they would be in an empty FIFO. The FIFO illustrated is one bit wide and can therefore store four bits; more length or width comes with further repetition of the internal parts of this data path. Alternate inputs pass through the upper and lower rails of the data path and merge again only at the output. When the FIFO is empty, as it is illustrated, it is transparent; one can trace a direct path from data input to data output. When each switch changes, identical data is presented to each of its inputs; thus the switches may momentarily short their two inputs together when changing. The micropipeline shown has an odd number of inversions and thus inverts its data value.

of a latch produces a simple and effective circuit for a FIFO.

Naturally, data must propagate through a micropipeline faster than the control events propagate through its control. This is usually assured by three factors. First, the Muller C-element used in the control circuit is more complex than the storage element used in the data path and therefore inherently slower. Second, since each single stage of the control system must drive the many storage elements that hold a parallel word in each register, the control signals must be amplified to

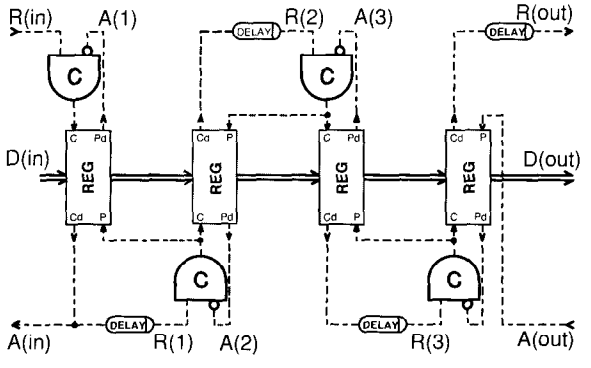

FIGURE 15. Micropipeline without Processing

A micropipeline without processing has event-controlled registers for data path and Muller C-elements for control. Four stages are shown; one of them is shaded. Each interface between stages conforms to the two-phase bundled data convention of Figure 4. This drawing suggests the form of an integrated circuit layout; the control signals pass back and forth across the data path to accommodate transmission delays.

drive multiple loads. This amplification inevitably delays the control signals. Third, the layout suggested in Figure 15 ensures that the zigzag path of the control signals has longer wires in it than those in the data path. Of course, when the circuit of Figure 14 is used as the register, the capture done and pass done control outputs from each register should be used to drive the Muller C-element inputs of adjacent stages so that any delays in the TOGGLE and XOR modules are included in the overall control signal delay. If no significant processing logic is placed in the data path, one can easily develop confidence that the data path is faster than the control path. Special delays are required in the control path only when significant processing logic is put between storage cells in the data path.

MICROPIPELINES WITH PROCESSING

The micropipeline framework provides a basis for a variety of pipeline processors [18]. My colleagues and I have designed multipliers, binary to one-out-of-N decoders, a memory controller, and other circuits using the micropipeline framework. In each case the micropipeline control template of Figure 10 provides the basis for the design. In some cases this simple template is embellished with circuits composed from the event logic modules of Figure 9 to provide more complex control functions. For example, using only one TOGGLE module and one XOR module it is easy to construct a circuit that performs two operations for each input event. The same two modules connected differently form a circuit that performs an operation only for every other input event.

When logical processing is required, suitable combinatorial circuits are placed between the storage registers, as illustrated in block form in Figure 17. One can trade off the number of stages of storage and the complexity of the intervening logic to obtain a suitable balance between latency and throughput rate. With less combinatorial logic between stages and more stages of storage, one obtains higher throughput rate at the cost of greater latency. The decoder circuits of Figures 18 and 19 perform the same function using different amounts of storage.

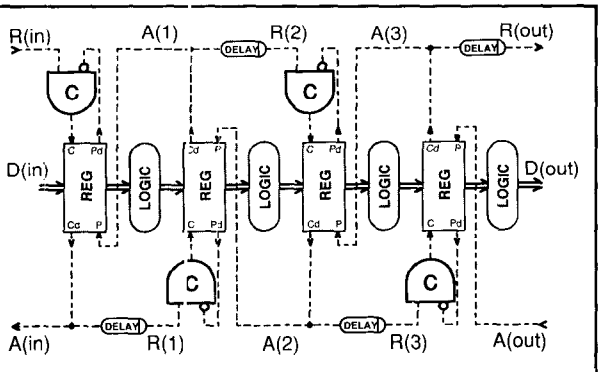

FIGURE 17. Micropipeline with Processing

A micropipeline with processing uses combinatorial logic between the event-controlled registers of Figure 15. Four stages are shown; one of them is shaded. The delay elements in the request event path model the processing logic delay to preserve the bundling convention. All interfaces between stages, taken either before or after the logic circuits, conform to the two-phase bundled data convention of Figure 4. The capture done, Cd, output of each register is shown connected to the pass, P, input of its predecessor, a more conservative connection than was used in Figure 15; either connection works.

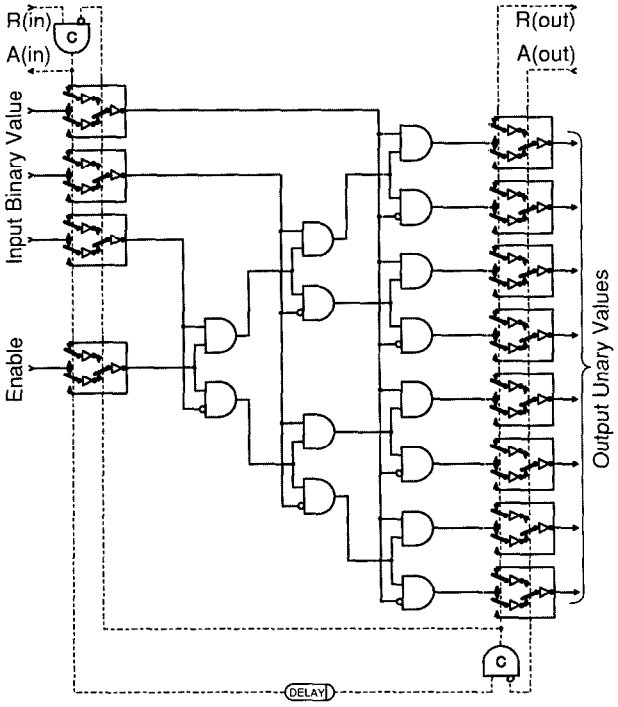

FIGURE 18. A Decoder with Two Micropipeline Stages

This two stage micropipeline decodes three binary input bits into eight unary output bits. It can store two values, one not yet decoded and the other fully decoded. The processing logic at the center of the figure is formed from three ranks of ordinary AND gates. The delay at the bottom of the figure must delay the request event at least as much as the three ranks of combinatorial logic delay the data.

The number of bits of storage in the registers of successive stages in a micropipeline may vary widely according to the needs of different processing steps. For example, the decoder of Figure 19 has 3 binary inputs but 8 unary outputs, and increases the width of the data word as each internal stage decodes an additional bit of the input. The 12-bit × 12-bit micropipeline multiplier whose layout is illustrated in Figure 20 has 24-bit data paths at input and output. It uses 24 stages of micropipeline: 12 to do the multiplication and 12 to resolve the carry-save form of product that results. At the center of the pipeline, 36-bit registers are required, since half of the product is in a carry-save form that requires two bits of data to represent each bit of product.

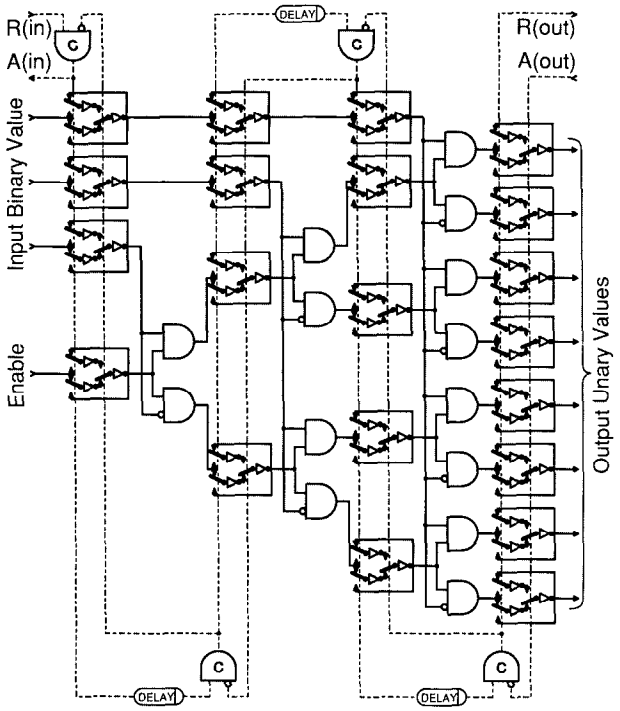

FIGURE 19. A Decoder with Four Micropipeline Stages

This four stage micropipeline also decodes three binary input bits into eight unary output bits. It does the same thing as the circuit of Figure 18, but it has more storage for partially decoded results. It can store four values, the first not yet decoded, two partially decoded values, and the final one fully decoded. The three delays in these request paths can each be shorter than the one in Figure 18, because each delay models only a single rank of combinatorial logic. This decoder has a higher throughput rate than the functionally equivalent decoder of Figure 18.

Because it has 24 stages, this multiplier can hold as many as 24 partially processed products. It can also hold fewer. It automatically processes any partially complete products as much and as fast as possible, considering the products already queued for output. At full operating speed, it provides the very high throughput of a pipeline process. When empty, however, it has no storage and acts as a combinatorial multiplier to produce individual products. It is never necessary to insert dummy data to flush previously entered information out of a micropipeline.

As a more complex example, we designed a memory controller using the micropipeline framework. This memory controller is intended for byte-serial access to a dynamic random access memory (DRAM) of 2^{24} words of 16 bits each. Its input and output registers are 8 bits wide to accommodate the byte-serial data format. We used the two-phase bundled data convention of Figure 4 as the byte transfer protocol at the input and output of this memory controller. It contains seven parts: four event-controlled storage registers and three stages of logic between them.

Each stage of control in the memory controller operates much like one of the simple stages in micropipeline control of Figure 10. Like those of Figure 10, each stage includes a Muller C-element and each stage communicates only with adjacent stages using exactly the two-phase bundled data convention of Figure 4. The control for each stage is composed from the event logic elements shown in Figure 9: XORs, Muller C-elements, SELECTs, TOGGLEs, and CALLs. In some stages these elements are connected in loops to permit several actions to take place within the stage before it acknowledges data from a previous stage or requests service from a subsequent stage. Such loops pack and unpack data. A separate memory refresh procedure interrupts normal operation using an ARBITER.

The logic in each stage of the memory controller performs a different function. The first stage decodes byte-serial input from the 8-bit input register, converting it into a 54-bit parallel word containing all of the address, data, and control information required for a memory cycle. This stage accepts and acknowledges several bytes of input before requesting action from the next stage. Between the first and second logic stages is a 54-bit event-controlled register. The second stage uses each 54-bit parallel word to control one access to the external DRAM chips. When reading from memory, this access converts the 54-bit address and control information into a 16-bit data value. The control includes a timing model for the memory chips and waits for the memory cycle to finish before requesting action from the next stage. Between the second and third stage of logic is a 16-bit event-controlled register that captures the data output from the DRAM chips. The third stage repacks the 16-bit output data into byte-serial form and presents it at the output terminals through the 8-bit event-controlled output register.

This memory controller, operating as a pipeline, can be carrying out a memory access while concurrently packing up the previously accessed data and unpacking the byte-serial address and control information for the next access. Because the stages are free of a common clock and each runs at its own pace, the pipeline is elastic. The elasticity permits a memory cycle to occur whenever a single set of address and command values is presented at the input, which may require several input bytes, even if no further input is provided.

The behavior of micropipelines is a blend of combinatorial behavior and pipeline processing. Remember that event-controlled storage elements are transparent when empty, and can behave like combinatorial circuits, storing nothing. Thus when a micropipeline is empty it behaves just like a combinatorial circuit. After their data path delay, the decoders of Figures 18 and 19, if empty, faithfully report as output the correct one-out-of-N code for any given binary input. You can confirm this by examining Figures 18 and 19 and remembering that the switches are all drawn in the positions they occupy when the data path is empty. Notice that complete paths involving no storage are available from input to output. Similarly, the multiplier of Figure 20, if

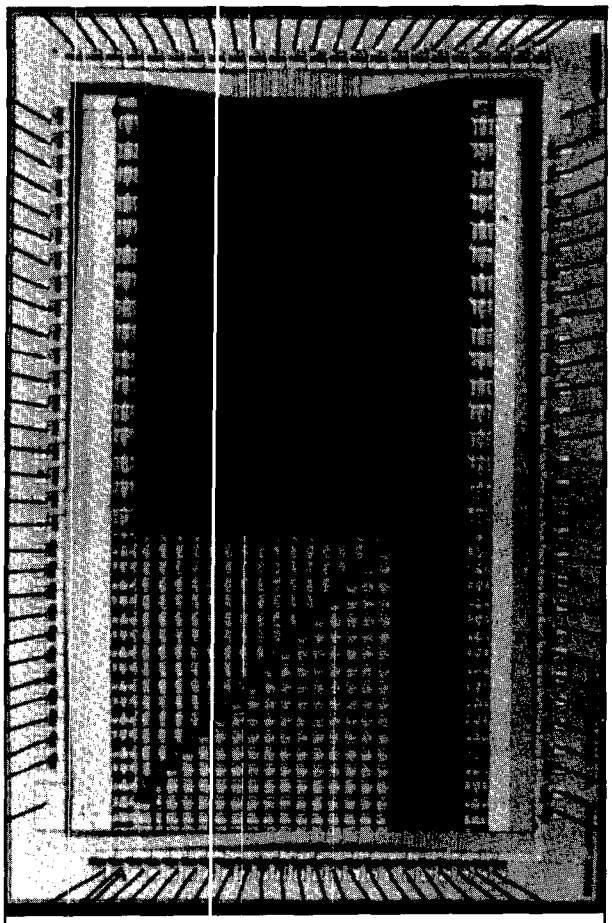

FIGURE 20. A Micropipeline Multiplier Chip

The experimental micropipeline multiplier shown in this photograph was built by Austek Microsystems. It multiplies pairs of 12-bit numbers using 24 stages of micropipeline; the first 12 stages are in the multiplication array, and the final 12 resolve the carry–save form of multiplier output. When empty, the multiplier acts just like a storage-free combinatorial multiplier, but when used as a pipeline it can accept up to 24 operand pairs before delivering its first product.

empty, faithfully reports as output the product of its input operands after its data path delay. This behavior makes the data path of a micropipeline easy to test.

The pipeline behavior of micropipelines is evident when they are given several inputs in rapid succession. Transition events on the request and acknowledge wires at the input end of the micropipeline serve to separate one input data element from another according to the two-phase bundled data convention of Figure 4. The "handshake" events on the request and acknowledge wires are like the rubber rods used in a grocery store check-out line to separate one customer's groceries from another's. Each request-acknowledge pair of events separates one data set from preceding or following data sets. The wave propagation properties of the Muller C-element control system move these data-separation events forward through the control circuits, and the control events force the data forward through the event-controlled storage registers, just as motion of the conveyer belt in the grocery store moves rubber rods and groceries forward toward the cashier.

The pipeline behavior of micropipelines is also evident when they deliver several outputs in rapid succession. Again the "handshake" events on the request and acknowledge wires at the output end of the micropipeline serve to separate one output value from another. Again the two-phase bundled data convention of Figure 4 is used, each handshake bringing a new value to the output data terminals. Micropipelines can exhibit very high burst input and output data rates.

The two-phase bundled data convention of Figure 4 automatically takes care of the "full" and "empty" conditions of the micropipeline. If the micropipeline becomes full, it will delay the acknowledge event on its input end, thus preventing further input. Remember that the device feeding the micropipeline must conform to the two-phase bundled data convention of Figure 4, and therefore cannot change the input data until after it receives an acknowledgment for the present data. Similarly, if the micropipeline becomes empty, it will delay the request event at its output end, thus preventing the output device from taking erroneous data. At every internal stage of the micropipeline the same signalling convention applies. Thus if a section is full, it will automatically delay new data from earlier sections in the micropipeline.

OTHER DEVICES USING THE SAME PROTOCOL

The two-phase bundled data convention used in micropipelines can be applied to other types of devices as well. For example, one can build a ring-buffer FIFO whose interface characteristics are the same as those of the micropipeline FIFOs of Figure 15 or Figure 16. Such a device might use an external random access memory for the required storage and two address counters as pointers, one for reading and one for writing, to treat the memory as a ring buffer. It would compare the values of the two pointers to recognize if the ring buffer were full or empty, and if so to delay the handshake signals at its terminals. The ring buffer pointers and the full and empty signals that result from comparing them could be private internal signals not available outside the control.

If a single port memory is used in such a ring-buffer FIFO, an arbiter must be used to decide whether the next memory cycle will be devoted to reading or to writing. Arbitration is required because a single resource, namely the memory access port, must be shared between two independently timed processes, the input process and the output process. If, at precisely the same instant, the input process delivers a new input value and the output process asks for a new output value, the arbiter must decide cleanly which request to service first. A transition logic control for such a ring-buffer FIFO is shown in Figure 21.

Although the circuit of Figure 21 looks like a flow diagram, it is in fact a circuit. It is composed of simple transition control modules, all of which use transition

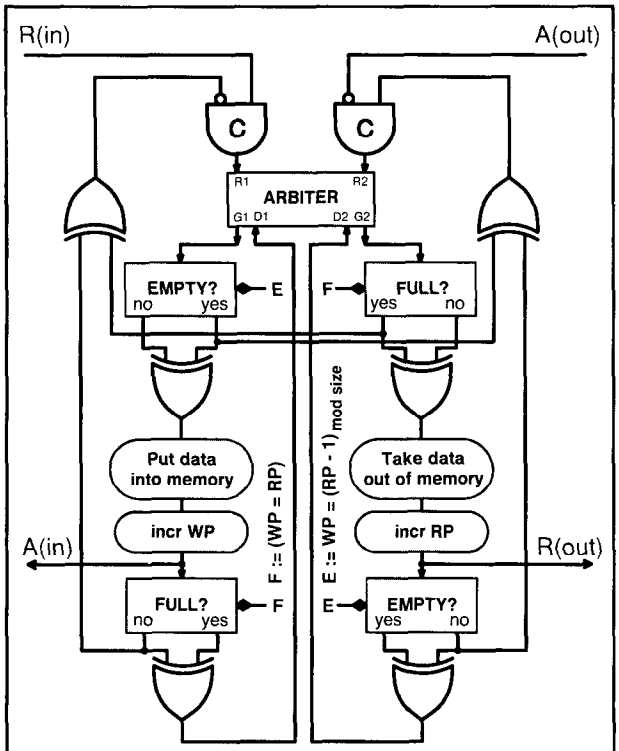

FIGURE 21. Ring-Buffer FIFO Control Logic

The control logic for a ring-buffer FIFO can be composed from the event logic modules shown in Figure 9. Except for the test values, all wires shown here carry event signals; the data path, the address pointers and the memory are not shown. In each of the four SELECT modules I have written the name of its test; the wires labeled "E" and "F" carry the required Boolean values. The functions described in the four lozenges include memory access and incrementing the read and write pointers, RP and WP. Although this figure looks like a block diagram, it is actually a circuit ready for direct implementation. It has been proven [5] than an external observer cannot distinguish this ring-buffer FIFO control circuit from the micropipeline control circuit of Figure 10.

signalling. Because these modules are insensitive to delay, composing them into circuits is much like drawing flow diagrams. Using tools developed by David Dill, my colleague, Bob Sproull, proved that if the FIFO controls of Figures 10 and 21 work at all, then for equivalent memory sizes they are functionally equivalent [5]. We can be assured, therefore, that such a ring-buffer FIFO and the micropipeline FIFO are interchangeable. The ability to make such proofs is one of the appealing things about the transition-signalling conceptual framework.

Using the two-phase bundled data convention of Figure 4 between micropipeline stages leaves wide latitude to make individual stages perform their functions in diverse ways. For example, a pipeline device for arithmetic normalization can be built with many stages or with a single stage. The multi-stage version performs a single bit shift in each stage, has very high throughput, exhibits long latency, and provides much buffer space. The single stage device performs its shifts sequentially, has reduced throughput and buffer space, but requires substantially less circuitry.

Three bits of a data path for such a single stage sequential normalizer are shown in Figure 22. Two registers of the form illustrated in Figure 14 are used in series to capture and hold the data. Switches at the top of the diagram select whether input data or shifted data enter the registers. The XOR and TOGGLE modules at the left of this circuit serve a similar role to those in Figure 14. Each event on the wire labeled "start latch data procedure" produces two events on each latch control wire and thus flips the register switches out of the position shown and then back into the position shown, capturing a new data value in the register. You should think of Figure 22 as the definition of the LATCH DATA PROCEDURE. This procedure has one input parameter, the "shift control" signal shown, and one output parameter, the "normalized or all zero" signal, which is generated by circuits omitted from the figure.

The sequential form of normalizer operates just as would a normalization program for a computer able to shift left only one place at a time. The control circuit is shown in Figure 23. At the top of the figure is a Muller C-element, similar to those we have seen in other micropipeline stages. After an event leaves the Muller C-element, its first action is to capture an input datum by using the upper client terminal, R1, of the CALL module to access the latch data procedure represented by the lozenge and defined in Figure 22. When the latch data procedure is done, it returns an event to the D terminal of the CALL module, which in turn returns an event to its D1 terminal. Thus shortly after the data are captured and before they are normalized, the control produces an event on its input acknowledge wire, A(in). From the point of view of a micropipeline stage, the rest of the algorithm below A(in) is just a delay before R(out).

After capturing the input datum, the control uses a while loop to shift the data into normalized form. The while loop contains an XOR module, a SELECT module, and the latch data procedure via the lower client terminals of the CALL module. An event circulates around the while loop and through the latch data procedure as long as the data are not yet normalized, causing one shift per trip around the loop. The time between shifts is established by the loop delay, and may be as fast or as slow as the circuits involved. When the while loop finishes, the event exits from the loop via the "true" output of the SELECT module. Thus when shifting is complete the control makes an event on the output request wire, R(out).

The shift control wire shown at the left of Figure 23 deserves special mention. We can think about it in two ways. First, thinking in terms of events, we should put an event on this control wire just before the while loop starts to flip the shift control switches into the shift position, and we need another event on it when the while loop finishes to flip the switches back into the

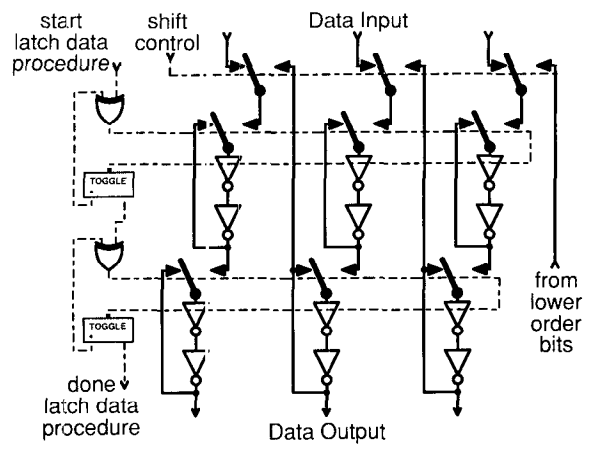

FIGURE 22. Data Path for Sequential Normalizer

The data patch for a sequential normalizer defines the LATCH DATA PROCEDURE. Only three bits of the register are shown. Circuits to detect the all zeros case or that the number is correctly normalized are not shown. The switches at the top of the data path select whether the register gets input data or shifted data. The TOGGLE and XOR modules at the left of the diagram are connected into two "do twice" loops in sequence. After an event arrives on the terminal labeled "start latch data procedure," the two ranks of latches in turn capture the data. After the data are latched, the final TOGGLE module delivers a single event to the terminal labeled "done latch data procedure."

input position. The two inputs to the XOR element that drives the shift control wire serve to bracket the while loop and thus deliver the two required events. The other way of thinking about the shift control considers the value of the XCR module output. So long as the while loop is active, its input and output control terminals will be in different states, and thus the output of the XOR module will be high, setting the switches in the correct position for shifting.

Designing control circuits like the ones illustrated here is rather like making block diagrams for programs. Not only do the event logic modules provide conditionals, procedure call, and other elements familiar to programmers, but also their response to events makes them easy to compose into loops and other structures similar to those found in programs. Using the form of the micropipeline control, it is also easy to build concurrent processing devices. We and others have built and tested libraries of such event logic modules and found them remarkably easy to use; the similarity of composing event-driven modules and programming has been recognized and used to advantage in a few places at least since the macromodule project [2, 3] during the 1960s. It provides, I think, an exciting alternative to conventional hardware design.

MICROPIPELINES IN GENERAL PURPOSE COMPUTING

General purpose computing machines use pipelines for two purposes: computation data paths and instruction decoding. They could also use pipelines in memory fetch operations if common memory parts and controllers were built using the micropipeline framework. Let us consider each of these three applications in turn to see how the micropipeline framework might improve system performance or usability.

Let us imagine a general purpose computing machine with micropipelines for arithmetic vector processing. Because the micropipelines provide an amount of storage that varies on demand, there need be no fixed vector length built into the machine. The program would be free to load vectors of any length, up to a maximum, into such a micropipeline, and subsequently unload the results. Using a micropipeline adder, for example, a program might pile in a set of address and offset addition tasks required to compute indexed memory references and use the internal storage of the micropipeline to hold the resulting sequence of addresses until needed. A program for multiplying short vectors by small matrices, an operation useful in computer graphics, might load the vector and matrix elements into a micropipeline multiplier followed by an accumulator. Vectors of 2, 3, or 4 components could be handled easily and efficiently by the same equipment. Moreover, because input and output operations might be separated in time, the indexing required for memory access might be simplified.

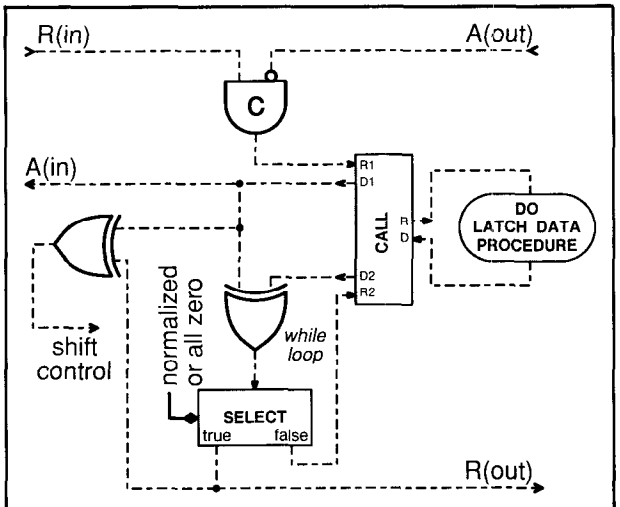

FIGURE 23. Control Circuit for a Sequential Normalizer

The control circuit for the sequential normalizer uses a call module to make the latch data procedure available for two separate purposes. The upper client uses the latch data procedure to capture input data with the shift control switches in the position illustrated at the top of Figure 22. The lower client is part of a while loop containing also an XOR and a SELECT module. An event circulates around this loop while the value is not yet normalized. Notice that an event is given to the shift control wire when the while loop starts, thus flipping the shift switches in the data path to the shifting position. Another even passes to the shift control wire when the while loop finishes, thus returning the shift switches to the input data position.

Perhaps the most important applications of micropipelines will involve operations in which the vector length changes. One such example is the clipping operation widely used in computer graphics [14, 15]. The clipping operation removes the parts of a set of objects that lie outside a reference window. Clipping may result in an increase or decrease in the number of objects in the set. Because whole objects may be removed, there may be less output than input, but because connected edges may also be broken into multiple pieces, there may also be more output than input. Such a clipping device with very simple interface characteristics can be built using the micropipeline framework.

Sorting is another important application for micropipelines in which the vector length changes. Micropipelines can be applied to both the partitioning and merging operations used in sorting. For partitioning, suppose that two micropipelines are connected to the outputs of a rapid micropipeline partitioner. Given a vector of input values, the partitioner can separate them into two output vectors according to some partitioning criterion, delivering elements from each vector into the corresponding output micropipeline. Of course, the number of elements in each of the two output vectors is data dependent, but the micropipelines are elastic and can easily accommodate variable length vectors by increasing or decreasing on demand the amount of storage available. The outputs of the two micropipelines can deliver the partitioned values without any need for priming or flushing.

Micropipelines can be applied to the merging operation as well. In this case two micropipelines for storing input vectors are connected to the two inputs of a merging device. This merging device can make whatever comparison is appropriate between the data values it is presented and select one of them for output. The elastic property of the input micropipelines permits the merging device to take data from either of them in whatever sequence the data values require. Partitioning and merging devices can be useful in signal-processing pipelines as well, for example, to divide a workload between several parallel pipelines.

Let us now turn from arithmetic to instruction processing. Pipeline instruction processing has become very common, and with reduced instruction set computer (RISC) architectures, is by now very well understood. One of its side effects is called "delayed branch." This name describes the fact that some precise number of instructions, for example exactly 2, will be performed in sequence after each jump or conditional jump instruction before the branch actually takes effect. These "overhang" instructions are necessary to keep the inelastic instruction processing pipeline busy while the new jump address takes effect. If nothing useful can be done in these overhang instructions, NOPs must be inserted as input to the instruction processing pipeline while it completes work on the jump instruction.

Let us imagine a micropipeline instruction processor. Such a processor can avoid the requirement for overhang instructions, but permit them to be included for additional speed. Although the micropipeline latency may create a time delay equivalent to two instructions after a jump, such a processor need not impose the storage cost of NOPs. If there is nothing useful to do, the NOPs can be omitted to save the storage. If some other number of instructions can usefully be done after the jump, for example, one or three, they may be inserted. By expanding or contracting the amount of storage used in the instruction processing pipeline on demand, the micropipeline framework can increase the programmer's flexibility. No longer does the pipeline have to contain exactly a fixed number of storage cells.

Condition codes can usefully be passed through a micropipeline. Conditions such as arithmetic comparison or parity for which a pipeline offers high throughput can be computed in vector fashion. For maximum throughput, the program should insert other operations between computing a condition and testing its result. If there is nothing useful to do between computing a condition and testing its result, intervening instructions may be omitted and the condition micropipeline behaves like a combinatorial circuit. Such a program may suffer the delay of the micropipeline latency, but will work properly.

With a micropipeline for storing condition codes, a program can compute several conditions before testing the first of them. The condition codes remain in the micropipeline in first-in-first-out sequence until tested. This is particularly useful in multi-way decoding trees, for example where three conditions control an 8-way branch. Instructions to compute each condition are required only once, and the three codes thus generated are stored in the micropipeline until tested in the branch tree. In conventional machines the instructions that compute the second and third conditions must be duplicated in each branch of the test tree.

Finally, memory systems obviously fit well into the micropipeline framework. One might design a dynamic random access memory (DRAM) part using a micropipeline. Such a memory part can provide at least a factor of two improvement in throughput over conventional DRAM parts. This improvement comes about because such a memory part can access its memory array concurrently with decoding the next address and with driving its data output pin or pins with the previously retrieved data. Such an improved part requires relatively little additional circuitry, since many of the actions in a DRAM are already driven from an internal timing chain. Only suitable event-controlled latches and Muller C-elements to form a micropipeline need be added to the existing DRAM logic, control mechanisms, and delay models. When concurrency is not needed, the micropipeline will be empty, making the event-controlled storage elements transparent, and permitting the micropipeline DRAM part to behave much like the one-cycle-at-a-time DRAM parts now in widespread use.

One might worry that a DRAM part with a micropipeline would require four control wires, two at the

input port and two at the output port, where existing parts have only two, called RAS and CAS. This is not so, because it will prove better to use an external timing model of the DRAM behavior, based on the manufacturer's worst case specifications, rather than to have each and every DRAM part in a system report completion on its own. The pins for completion signals at both input and output port can be omitted from the individual DRAM parts, because the external timing model provides the two missing completion signals on behalf of the entire memory system, using its model of the DRAM behavior to provide suitable delays. The input port of individual DRAM parts needs only the request wire, and the output port of individual DRAM parts needs only the acknowledge wire. Events on these two control wires respectively tell the DRAM part when to accept new address information or data to be written, and when to present a new output value. The external timing model will itself be a micropipeline built with stage delays that equal or exceed, stage by stage, the corresponding delays in the micropipeline in the DRAM parts.

Cache memories also fit well into the micropipeline framework. A very high throughput cache built within this framework can perform decode, detect "hits," and drive its output all concurrently. Such a cache memory has two interface pairs, both using a signalling convention similar to the two-phase bundled data convention of Figure 4. One pair of interfaces connects the cache to the processor and the other pair connects it to memory, as shown in Figure 24. The pair of interfaces between the processor and the cache should be identical to the pair of interfaces between the cache and the memory, so that the system can operate with or without the cache as shown in Figures 24 and 25.

The processor, cache, and memory, taken together form a micropipeline. Memory requests from the processor flow into the cache and back in micropipeline fashion, going to memory and back only when necessary. The processor can give the cache several memory requests concurrently before getting any data back. For highest throughput, the processor should deliver a continuous stream of memory requests, but it operates correctly, albeit at reduced throughput, if it gives only one request at a time. Because the two-phase bundled data convention of Figure 4 permits either sender or receiver to delay the next transaction arbitrarily, cache or memory access delays automatically delay subsequent requests from the processor. Similarly, the part of the processor that consumes memory data waits however long is required for the events that signal the presence of valid data.

If the cache does not contain the required information, it passes the request on to the memory. In this case the processor suffers the additional delay required to fetch information from memory. Because the processor accepts data from the cache only when it detects a validating request event, the processor easily accommodates to any additional memory delay. In fact, if the

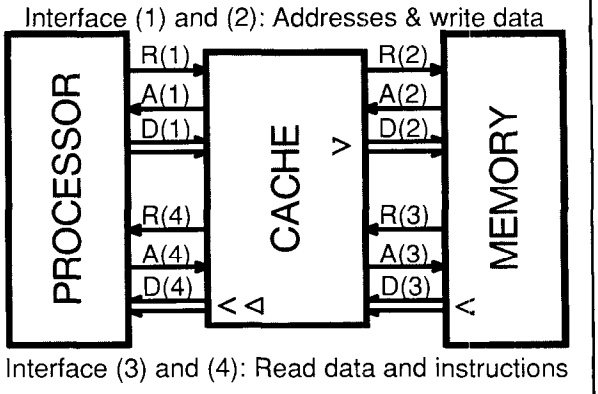

FIGURE 24. Memory Interface with Cache

A processor, cache, and memory system can use interfaces that follow the two-phase bundled data convention of Figure 4. Interfaces 1 and 2 carry addresses and data to be written; interfaces 3 and 4 carry data and instructions. The entire system can operate as a micropipeline at the maximum throughput rate of its slowest participant.

cache sent every request to the memory, or if the cache were omitted, the system would still operate properly, albeit at reduced throughput.

This leads me to the most important implication of micropipelines. Because they use event-controlled interfaces rather than a common clock, micropipelines with different inherent speeds can be composed directly into systems that function correctly, albeit at the speed of the slowest part. If the cache of Figure 24 were omitted or replaced with a cache with different cost and performance characteristics, the system would still operate correctly. Similarly the processor performance or the memory performance can be upgraded and the system will still work, taking advantage of any available speed improvements.

CONCLUSION

FIFOs and pipelines are simple to design and easy to understand in the transition-signalling conceptual framework. They are relatively difficult to design within the clocked-logic conceptual framework. By abandoning clocked logic in favor of transition signalling, one is able to make very simple micropipelines that assemble easily into larger structures. The change in conceptual framework suggested here simplifies system design because simple modules and compositions of them can be further composed into large systems.

The composability offered by micropipelines and transition signalling may be their most important property. Complex functions are easy to compose from simple modules that provide basic functions already familiar in programming. More complex systems can be built by composing them as a hierarchy of the basic modules and previously designed compositions. Even if the basic building blocks were hard to design, and they no longer are, they would be worthwhile, because they are so

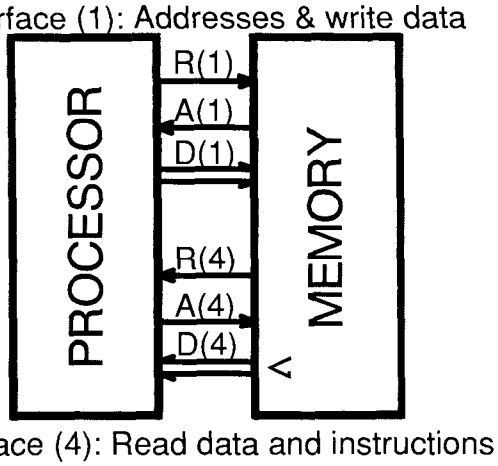

FIGURE 25. Memory Interface without Cache
With the cache omitted, the processor and memory can still function together. Because each interface accommodates arbitrary delay, no further design change is required when the cache is omitted or when substituting a cache with different performance characteristics.

easy to compose into systems.

This same composability offers a simple way to upgrade system performance as improved circuitry becomes available. Event-driven interface protocols permit old components to be replaced by new ones with improved throughput, latency, or cost characteristics. Because the handshake used here automatically takes care of delays in delivering or making use of data, such replacements can be made with assurance that the system will still operate properly. On the other hand, large systems built in the clocked-logic conceptual framework resist incremental improvement, because any increase in clock speed must be accommodated throughout the system. As improvements are made to systems built as I have outlined here, one can expect that the slowest or most expensive parts of a system will be replaced first, and thus that each replacement will improve system performance or decrease system cost. Thus the transition-signalling conceptual framework, micropipelines, and the two-phase bundled data convention of Figure 4 taken together not only simplify initial system design but also permit rapid mid-life upgrade of systems as new technology becomes available.

I hope that this lecture may help system design to keep pace with advancing component technology. Today, new integrated circuit technology makes available significant improvements in cost or performance every six months or so. It is often difficult to make use of such improved performance, because speeding up the clock in an entire system is a formidable task fraught with dangers. Today's system designers, constrained by the clocked-logic conceptual framework, take several years to produce a new system. Thus the systems being sold may lag by several years the potential speed or cost benefits offered by the most modern technology. I believe that the micropipeline framework that I have described here can reduce the opportunity cost imposed by the clocked-logic conceptual framework.

Acknowledgments. The transition-signalling conceptual framework has been used in a few places over a long period of time. I know of early work at the University of Illinois by David Muller, at the University of Utah by Al Davis, and at the Massachusetts Institute of Technology by Jack Dennis; I apologize in advance for omitting mention of other projects.

I owe my own education in the transition-signalling conceptual framework to a few able people. I first became aware of transition signalling in the early 1960s when a group at Washington University in Saint Louis, led by Wes Clark, used it in the design of a set of macromodules [2,3]. I learned much more about it from Charles Seitz, now on the faculty at Caltech, over a dozen years starting in 1966 when, as an MIT graduate student, he taught me most of what I know about digital design. We worked together at Harvard and at the Evans and Sutherland Computer Corporation using an almost-correct version of micropipelines in processors for computer graphics, including the original "clipping divider" [14]. His chapter in the well-known Mead and Conway book on VLSI is one of the best presented and most accessible references on transition signalling [13].

Two other people have been important to my education. Most important to me over a long period of time is Bob Sproull, from whom I first started a lifelong education twenty-five years ago and of whose knowledge and ability I remain in awe. He has regularly fixed my thinking when it was fuzzy, and has made this lecture more accurate than I alone could have. During the past five years he and I led an "Asynchronous Systems Study" to learn and teach transition signalling. As part of it we designed micropipelines for various purposes, including those I have described here. We have taught our subject to a few hundred people, and we have two books in preparation. Finally, I want to mention Charles Molnar, whose group at Washington University continues the pioneering work I mentioned before. He has given unstintingly to my education not only his time and ideas but also his enthusiasm. His contributions to the transition-signalling conceptual framework include not only the absolutely essential synthesis method for logic modules [9] without which the new framework was difficult to use, but also many important parts of an overall mathematical theory, and new conceptions of useful circuits [12]. The theoretical work [6,10,11,19], in which he collaborates with Martin Rem's group in Eindhoven, is beginning to prove theorems about the correctness of systems designed in the transition-signalling conceptual framework.

The work reported here led to a broader "Asynchronous Systems Study," conducted by Sutherland, Sproull and Associates, Inc., and supported by six industrial sponsors: Apple Computer, Austek Microsystems Ltd., Digital Equipment Corp., Evans and Sutherland Com-

puter Corp., Floating Point Systems, and the Schlumberger Research Laboratory. We did the work with the cooperation of Carnegie Mellon University and Imperial College of the University of London. Erik Brunvand, Ed Frank, Ian Jones, Charles Molnar, and Bert Sutherland collaborated with us. We were able to test micropipeline circuits fabricated for us by the MOSIS integrated circuit fabrication service operated by the Information Sciences Institute of the University of Southern California. We are proud to have been the very first commercial MOSIS client.

REFERENCES

1. Chaney, T.J., and Molnar, C.E. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput. C-22*, 4 (Apr. 1973), 421–422.
2. Clark, W.A. Macromodular computer systems. In *Proceedings of the Spring Joint Computer Conference*, AFIPS, April 1967.
3. Clark, W.A., and Molnar, C.E. Macromodular computer systems. *Computers in Biomedical Research*, Vol. 4, R. Stacy and B. Waxman, Eds., Academic Press, New York, 1974, 45–85.
4. Dally, W.J., Seitz, C.L. Deadlock-free message routing in multiprocessor interconnection networks *IEEE Trans. Comput. 36*, 5 (May 1987), 547–553.
5. Dill, D.L., Nowick, S.M., and Sproull, R.F. Specification and automatic verification of self-timed queues. Computer Systems Laboratory Report, Stanford University, 1988.
6. Ebergen, J.C. Translating programs into delay-insensitive circuits. Ph.D. dissertation, Eindhoven University of Technology, 1987.
7. Levy, J.V. Buses, the skeleton of computer structures. In *Computer Engineering*, C.G. Bell, J.C. Mudge, and J.E. McNamara, Eds., Digital Press, 1978.
8. Miller, R.E. "Sequential Circuits", Chapter 10, In *Switching Theory*, Vol 2, Wiley, NY, 1965.
9. Molnar, C.E., Fang, T.P., and Rosenberger, F.U. Synthesis of delay-insensitive modules. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, H. Fuchs, Ed., Computer Science Press, 1985.
10. Rem, M., van de Snepscheut, J.L.A., and Udding, J.T. Trace theory and the definition of hierarchical components. In *Proceedings of the Caltech Conference on VLSI*, 1983.
11. Rem, M. Trace theory and systolic computations. In *Proc. PARLE (Parallel Architectures and Languages Europe)*, Vol 1, J.W. deBakker, A.J. Nijman, and P.C. Treleaven, Eds, Springer-Verlag, 1987, pp. 14–34.
12. Rosenberger, F.U., Molnar, C.E., Chaney, T.J., et al. Q-modules: Locally clocked delay-insensitive modules. *IEEE Trans. Comput. 37*, 9 (Sept. 1988), 1005–1018.
13. Seitz, C.L. System Timing. In *Introduction to VLSI Systems*, C.A. Mead and L.A. Conway, Eds., Addison-Wesley, 1980.
14. Sproull, R.F., and Sutherland, I.E. A clipping divider. *FJCC 1968*, Thompson Books, Washington, D.C., 765.
15. Sutherland, I.E., and Hodgman, G.W. Reentrant polygon clipping. *Commun. ACM 17*,1 (Jan. 1974), 32–42.
16. Sutherland, I.E. Asynchronous queue system, U.S. Patent 4,679,213, July 7, 1987.
17. Sutherland, I.E., Asynchronous first-in-first-out register structure. US Patent Pending.
18. Sutherland, I.E. Asynchronous pipelined data processing system. US Patent pending.
19. Udding, J.T. A formal model for defining assifying delay-insensitive circuits and systems. *J. Distrib. Comptg. 1*, 1986, 197–204.

CR Categories and Subject Descriptors: B.2.1 [**Arithmetic and Logic Structures**]: Design Styles—*pipeline*; B.5.1 [**Register Transfer-Level Implementation**]: Design—*styles(e.g., parallel, pipeline, special-purpose*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Input/Output circuits*
General Terms: Design
Additional Key Words and Phrases: Asynchronous handshake, FIFO, pipeline processing, transition signalling