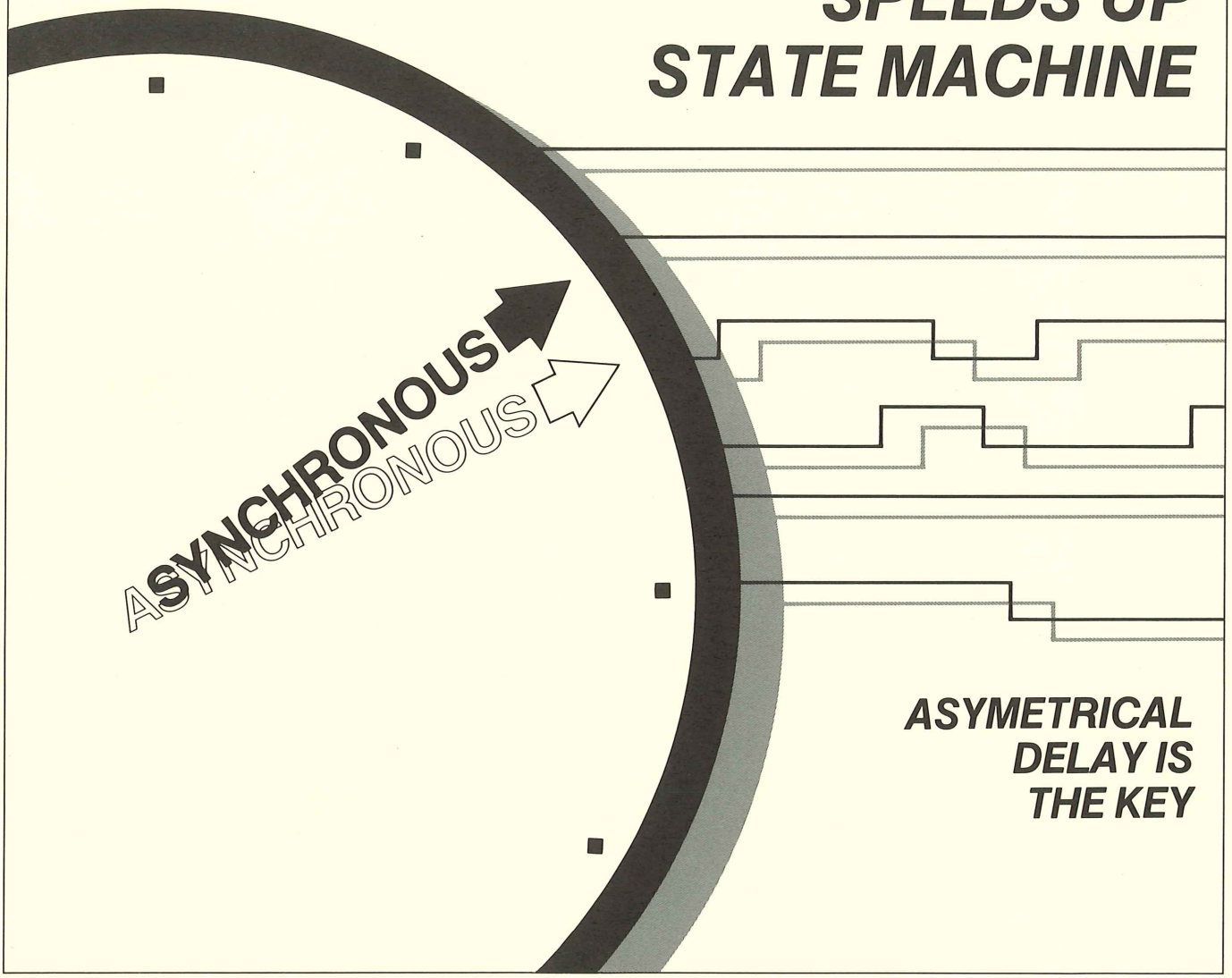
TEK INFORMATION FOR TECHNOLOGISTS

MARCH/APRIL 1986

TECHNOLOGY report

COMPANY CONFIDENTIAL

SELF-SYNCHRONIZATION SPEEDS UP STATE MACHINE



ASYMETRICAL DELAY IS THE KEY

Tektronix®
COMMITTED TO EXCELLENCE

CONTENTS

Asymetrical Delay Makes Self-Synchronized State Machine Faster 3

Engineering Forum Aims at Building Business Excellence 13

Testing the Large Chip: Artificial Intelligence Shows Promise in Reconciling Functional with Gate Level Testing 14

Tolerances Too Tight: Aluminum-Extrusion Vendors Shy Away from Doing Business with Tek 17

CONNECTIONS. . . Lookahead Carry Speeds Up Binary Addition/Subtraction and Binary/BCD Addition—Less Logic Too 18

Standards Review Board 28

Volume 8, No. 2, April/May 1986. Managing editor: Art Andersen, 642-8934, d.s. 53-077. Cover: Monica Kaul, Graphic illustrator: John Kennedy. Composition editor: Sharlet Foster. Published for the benefit of the Tektronix engineering and scientific community.

This document is protected under the copyright law as an unpublished work, and may not be published, or copied or reproduced by persons outside TEKTRONIX, INC., without express written permission.

Why TR?

Technology Report serves two purposes. Long-range, it promotes the flow of technical information among the diverse segments of the Tektronix engineering and scientific community. Short-range, it publicizes current events (new services available and notice of achievements by members of the technical community at Tektronix).

HELP AVAILABLE FOR PAPERS, ARTICLES, AND PRESENTATIONS

If you're preparing a paper for publication or presentation outside Tektronix, the Technology Communications Support (TCS) group of Corporate Marketing Communications can make your job easier. TCS can provide editorial help with outlines, abstracts, and manuscripts; prepare artwork for illustrations; and format material to journal or conference requirements. They can also help you "storyboard" your talk, and then produce professional, attractive slides to go with it. In addition, they interface with Patents and Trademarks to obtain confidentiality reviews and to assure all necessary patent and copyright protection.

For more information, or for whatever assistance you may need, contact Al Carpenter, 642-8955.

WRITING FOR *TECHNOLOGY REPORT*

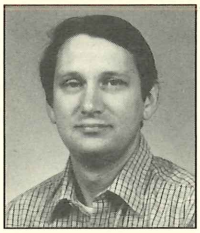
Technology Report can effectively convey ideas, innovations, services, and background information to the Tektronix technological community.

How long does it take to see an article appear in print? That is a function of many things (the completeness of the input, the review cycle, and the timeliness of the content). But the minimum is six weeks for simple announcements and as much as 14 weeks for major technical articles.

The most important step for the contributor is to put the message on paper so we will have something to work with. Don't worry about organization, spelling, and grammar. The editors will take care of that when we put the article into shape for you.

Do you have an article to contribute or an announcement to make? Contact the editor, Art Andersen, 642-8934 (Merlo Road) or write to d.s. 53-077. □

Asymetrical Delay Makes Self-Synchronized State Machine Faster



Donald C. Kirkpatrick is a senior hardware/software engineer in the Logic Analyzer Division. Don joined Tek in 1972 from General Telephone where he was an engineer. Don holds a PhD in electrical engineering from Oregon State University.

Synchronous machines have many advantages over asynchronous machines, but asynchronous designs can run faster. Don Kirkpatrick, Logic Analyzer Division, set out to combine the best of both in one scheme. The result, a major improvement to product-line functionality that will be announced this summer.

Where speed is critical, an asynchronous machine has the distinct advantage of not having to wait for the next clock pulse. On the other hand, in synchronous machines state assignment is efficient and less logic is required.

Since synchronous machines have many advantages over asynchronous machines, the Logic Analyzer Division needed to combine the best of both machines in one scheme. We did this by using *self-synchronized* asynchronous design, thus retaining the speed advantage of the asynchronous machine while gaining the efficient state assignment and logic reduction of the synchronous machine. The price for this is a clock generator.

Our problem was to implement correctly the state-transition function and output function of the machine as specified by a flow table. A logical model for all state machines is shown in Figure 1. This model is completely general. Any sequential state machine can be built in this form.

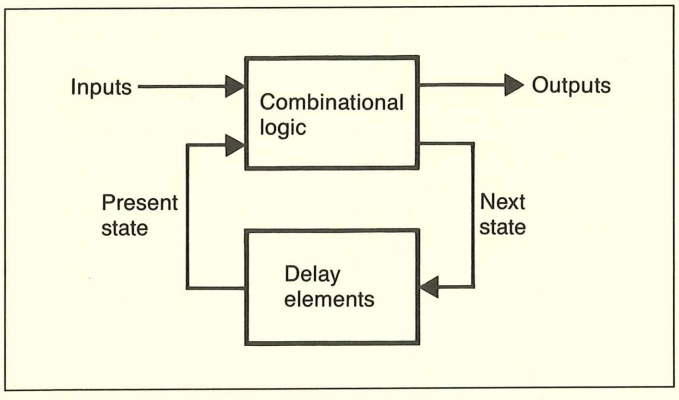


Figure 1. Huffman-Moore model finite state machine.

The combinational logic block, in the model, contains no memory and the delay-element block contains only memory devices. The input signal combination presented to the machine is called the *input state*. The output signal combination produced by the machine is called the *output state*. The state-variable signal combination is called the *internal state*. The individual input, output, or internal state-variable signals themselves will be referred to as inputs, outputs, or state variables. Together, the internal and input states form the *total state* (or just *"state"*).

The usual approach to designing simple asynchronous machines, after the style of the Huffman-Moore machine (Figure 1), is to base the design on broad conditions called *operating assumptions*. Within these conditions proper functional behavior can be assured.

For example, the most-often-required environmental condition is that the machine be allowed to work in *fundamental mode*; that is, a final stable state is reached between input state changes. Machines operate either in fundamental or non-fundamental mode. Some, such as UIC machines, operate in non-fundamental mode.

The new next-state value is fed back to become the new present state. The delay elements in the feedback path are one of several classes of devices that delay feedback. These delay elements compensate for the finite and differing speeds of the various paths from inputs to next-state terminals.

The appropriate delay times depend on the particular circuitry in the combinational logic. While the circuit technology determines the ultimate minimum delay time and maximum operating speed, the choice of a machine structure may adversely impact this ultimate speed.

If there are several input signals to an asynchronous machine, a distinction is made between *single input change* (SIC) and *multiple input-change* (MIC) modes. In MIC mode, more than one input is allowed to change, and all changes within some small interval are accepted as if they were simultaneous. In the SIC mode, of course, only one input is allowed to change.

Design methods for SIC machines are well established, although not as direct and easy as methods for synchronous machines. MIC mode machines are a bit more complex. In either case, proper operation depends upon proper behavior of the input. One way to make asynchronous-machine behavior predictable and designable is to use speed independent machines. Input changes for a *speed independent* machine are permitted only when the machine indicates, through special outputs, that it is ready to accept the next input change.

According to the flow table of the machine, a single input change may cause one or more changes of state. Since the output is a function of the state, state changes may or may not cause changes in the outputs. If no allowed input change causes more than one output state change, the machine is a *single output-change* (SOC) machine. If the number of output state changes is bounded but sometimes more than one, the machine is in the *multiple output-change* (MOC) category; otherwise it is in the *unbounded output-change* (UOC) category. An SIC, SOC machine in fundamental mode is a *normal* fundamental mode machine.

Delay elements used previously in self-synchronized machines exhibit one of three behaviors: (1) The *monostable multivibrator* outputs an edge at a fixed time, D, after the input edge that triggers it. (2) The *pure delay* shifts the entire input signal in time by a fixed amount, D. (3) The *inertial delay* echoes the input transition only after it has persisted for some delay time, D. The inertial delay does not pass to its output any change that does not last for the delay time, D.

One important specification for any asynchronous MIC machine is a time interval, δ_1, during which several input signals may change. The machine is to consider these input changes to be simultaneous. That is, these input-signal changes together are to be considered as only one input-state change. Given this specification and the required machine behavior, a circuit is designed to realize the machine. One result from the design is the determination of a second time interval, δ_2. The inputs must remain stable during this second interval while the machine perambulates from one state to the goal state. If the inputs do not remain stable, unpredictable behavior will result. The minimum time between input state changes is the sum of the two intervals, $\delta_1+\delta_2$. A MIC machine with no specified δ is known as Unrestricted Input Change (UIC). Any input may change at any time.

Timing Analysis

Over the years, the following notation has evolved as conventional when writing timing expressions:

- D : Delays through delay elements.
- d : Stray delays through combinational logic.
- s : Set-up times for flip-flops.
- f : Propagation delays through flip-flops.

Subscripts M and m represent maximum and minimum values respectively. This notation will be used throughout the timing analysis that follows.

Because of functional hazards in the combinational logic, a MIC Huffman-Moore machine having a proper critical race-free state assignment will, in general, still require delay elements for proper operation. During a multiple input change, the earliest a change can reach the logic output is d_m; the latest a change can reach the logic output is δ_1+d_M. The combinational logic may exhibit spurious output pulses for a period as long as the difference between these two times. The delay elements must filter out such spurious output. Thus the minimum delay element value is:

$$D_m \geq \delta_1 + d_M - d_m.$$

When any machine generates multiple output states, it does so by "perambulating" through intermediate total states and generating output states. *If the inputs do not remain stable until the final and stable state is reached, the fundamental-mode assumption is violated.* (Lift the fundamental-mode restriction and the machine is in UIC mode.) The times between successive intermediate states (and thus successive output states) are determined by the propagation delays through the combinational-logic block and the delay element. The time for one intermediate state transition (D+d) is bounded by a minimum of D_m+d_m and a maximum of D_M+d_M.

The last changes caused by the final input change of an input state, including any state variable change, must reach the combinational-logic outputs before the first change of the next input state. If n is the number of intermediate internal state transitions required to produce all the output states, then

$$\delta_2 + d_m \geq d_M + n(D_M + d_M).$$

Thus the time between input states must satisfy the inequality

$$\delta_1 + \delta_2 \geq \delta_1 + n(D_M + d_M) + (d_M - d_m)$$

If the machine is designed to operate in single output-change mode, then n=1. If the transition function has no essential hazards, then state assignments exist[2] that can result in a delay-free realization ($D_M=0$). For any level-sensitive Huffman-Moore machine, a proper state assignment must be found. This assignment is customized, based on the transition function, using the techniques developed by Liu and others.[2,3,4]

Self-Synchronized Machine Structures

While the Huffman-Moore model (Figure 1) can describe a self-synchronized machine, it is better to augment the model slightly as shown in Figure 2. This was first done about fifteen years ago.[5]

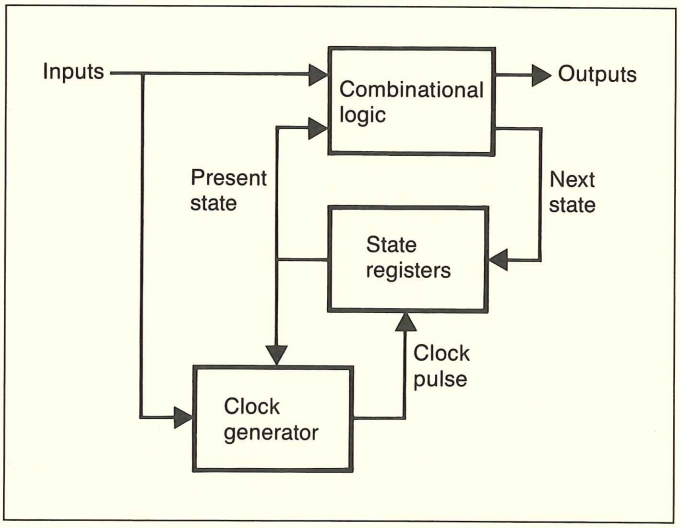


Figure 2. MOC clock generator—input and present state.

In the augmented model, edge-triggered flip-flops—organized as state registers—replace generalized delay elements and a clock generator is added. With proper clock-generator design, only one delay element is required (inside the clock generator). This delay element times the pulse edge that clocks the flop-flops. The first self-synchronized machine was built in just this fashion.[5] It operated only in normal fundamental mode.

By using this augmented model, the designer can design a "standard" clock generator, without considering the behavior of the machine itself. This universal approach reduces design effort. In this model, the clock generator monitors the inputs and state variables to produce a clock pulse when an input or state-variable changes. This structure was first used in a multiple output machine by Rey and Vaucher.[6]

Self-synchronized timing analysis

In a self-synchronized machine using this augmented structure, the first clock-pulse edge must not reach the flip-flops before the input-generated changes have gone through the combinational logic, reached the state-variable flip-flops, and met the set-up time requirements. Thus,

$$D_m \geq \delta_1 + d_M + s.$$

The state-transition time is the sum of the delays through the flip-flops and the combinational logic, and the set-up time for the flip-flops ($f_M + d_M + s$). If the machine is to operate only in single output-change mode, the inputs are permitted to change after the state transition. However, to operate in the multiple output-change mode, a clock pulse must be generated after every state change, because more states might follow. Thus, for MOC machines, n state transitions will generate n+1 clock pulses (since the machine does not "know" no more states will follow)

$$\delta_2 + D_m \geq (n+1)(f_M + d_M + s) + D_M$$

and input state changes are separated by

$$\delta_1 + \delta_2 \geq \delta_1 + (n+1)(f_M + d_M + s) + (D_M - D_m).$$

The time between input states is proportional to n+1, but in an optimum machine, delay is proportional to n. For a SOC mode machine, n=0, since it is known *a priori* no more states will follow (only one clock pulse is ever needed).

For the single output-change operation, we can compare the speed of the self-synchronized machine (n=0) with the level-sensitive Huffman-Moore SOC machine (n=1 state transition).

Assuming equivalent technologies, the combinational-logic delays (d_M) should be equal. The uncertainty terms are simply the difference between the fastest and slowest state-variable change ($d_M - d_m$) and the difference clock-pulse ($D_M - D_m$) respectively. The two uncertainty terms should also be nearly equal for equivalent technologies. The two machines operate at the same speed when the right-hand side of the input-state timing inequalities are equal. Equating the two right-hand sides and canceling these approximate equalities results in

$$D_M = f_M + s.$$

A Huffman-Moore machine operating SIC is always faster if it does not have an essential hazard since a delay-free ($D_M = 0$) state assignment can be made.[2] In the self-synchronized MIC machine, flip-flop set-up-time (s) and propagation-delay (f_M) depend on technology, D_M in the Huffman-Moore machine increases with δ_1. *Conclusion: the greater the* δ_1, the greater the advantage for self-synchronization when operating in MIC mode.

The designer should note that $f_M + s$ can be very small. Set-up for the 74F374 and 10H131 are 4 nanoseconds and 1.4 nanoseconds; typically maximum propagation delays for these commercial latches are 10 and 2.1 nanoseconds respectively.

The designer can also customize clock-generator logic to the behavior of the machine.[7] Since both clock generator and combinational logic can access the same information, the clock can compute the next state and generate a clock pulse. However, such a behavior-customized architecture can't reach ultimate speed, because the clock generator must be able to detect when one intermediate-state transition is complete. Since a clock pulse is generated (after a suitable delay) only when the next state becomes different from the present state, the logic condition must reach next-state and present-state equal before another clock pulse can be issued. In an intermediate-state transition during a multiple output-change perambulation, an equality of states is often brief.

The next-state-must-equal-present-state restriction is crucial. Consider what might happen if the delay through the clock-pulse-required combinational logic is less than the delay through the state-transition-complete (next-equals-present) combinational logic. In that case, at time f_M after the clock pulse changes state s_i to s_j, state s_j appears at the inputs of both combinational-logic blocks. If the request for the next clock pulse comes out of the combinational logic before the next-equals-present state is sensed, the clock generator may never detect that transition s_j is complete. This will cause the machine to lock up in intermediate state s_j and subsequent input changes may not dislodge it.

Since at least 1962,[8] many designers have known generating a pulse each time an input changed could combine the best features of asynchronous and synchronous machines. However, they could not develop a clock that would not compromise the inherent speed advantage of an asynchronous machine.

The clock generator

Our clock generator has two parts: a change detector to determine when a clock pulse is needed and a delay element to generate the clock pulse (figure 3). As discussed earlier, the change detector could be customized to a machines clock-pulse needs; this introduces timing restrictions that slow the machine. It's better to use a generalized change detector.

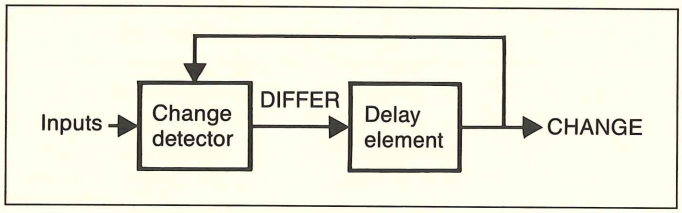


Figure 3. Clock generator expanded.

The change detector outputs the signal DIFFER when the input signals change. DIFFER propagates through the delay element, emerging a predictable time later as the signal CHANGE. This signal may be fed back to the change detector, which turns off DIFFER; a short while later, DIFFER-off propagates through the delay element and CHANGE goes off. By using a monostable multivibrator (as an alternative delay element) the designer can eliminate the need for feedback. In either case, it is usually the final transition on CHANGE that clocks the state-register flip-flops. If, for some special reason, the flip-flops are clocked from the leading edge of CHANGE, then the timing analyses must account for the interval in which CHANGE is high after the flip-flops have been clocked.

Clock Generator Design

Change detectors for single input-change mode

The first practical clock-pulse generator was developed for a normal fundamental-mode machine.[5] Since only a single input is permitted to change in this machine, the modulo 2 sum of the input vector components changes for each input state. This change-detector method is not suitable for a multiple output-change machine unless there is only a single change of state variable for each state change. One reason for using a self-synchronized machine is to simplify the state assignment. Constraining state encoding erases this advantage.

Change detectors for multiple input-change mode

Previous investigators have proposed two ways to generate clock pulses for MIC machines. The change detector in the first approach was a combinational-logic network customized for the required machine behavior.[7] This approach is slower than a generalized change detector.[8]

The second approach uses monostable multivibrators on the inputs to convert changes in the level-sensitive inputs into pulse-mode inputs.[6] These pulse-mode inputs are then combined in an OR gate, which triggers another monostable multivibrator to form the clock pulse. Although such change detectors are fundamentally sound, implementation can be difficult. Since each input has an edge-to-pulse converter, each signal input requires one multivibrator. Building accurate multivibrators for narrow pulse widths is difficult—and high operating speeds require short clock pulses.

Employing a *digital differentiator* as the change detector solves the problems of the second approach. It converts a change in input level into a level. To understand its operation, assume the present input state is stored in the latch (figure 4) and the enable (CHANGE) is off. When one or more inputs change, the appropriate exclusive-or gate outputs a high and DIFFER goes high. After a time determined by a delay element (not shown), CHANGE goes high and the latch is opened. When the latch outputs match the input state, DIFFER and (eventually) CHANGE again go low. The change detector can now accept the next input state. Figure 5 shows the timing relationships for this change detector. The first input change (I_1) starts the cycle while I_n is the last change to be part of this input-state change.

This clock generator is particularly economical. A clock generator that accepts up to eight inputs can be built with only two parts: one 74F373 eight-bit latch and one 74F521 eight-bit equality comparator. The minimum and maximum propagation delays from input to DIFFER are 3 and 11 nanoseconds; the delays from CHANGE to DIFFER are 8 and 24 nanoseconds. Thus the change detector's minimum and maximum propagation delays are 11 and 35 nanoseconds.

But one more obstacle blocks ultimate speed.

If a self-synchronized machine with a symmetrical delay element uses this clock generator, a timing restriction remains to force the machine to operate at less than ultimate speed. The

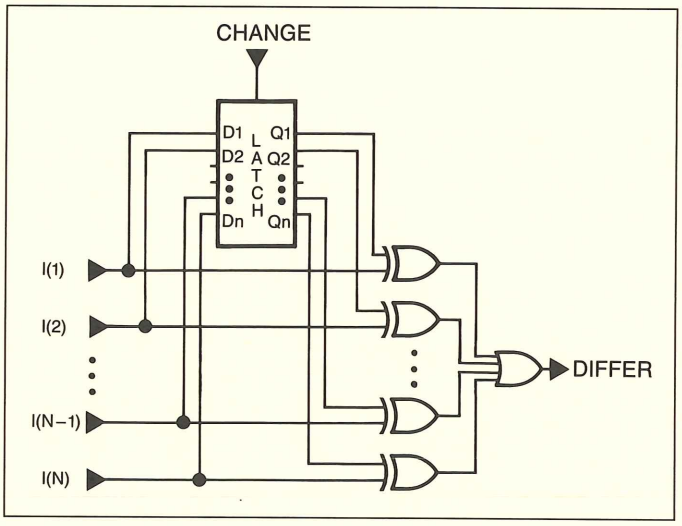


Figure 4. Digital differentiator as the change detector.

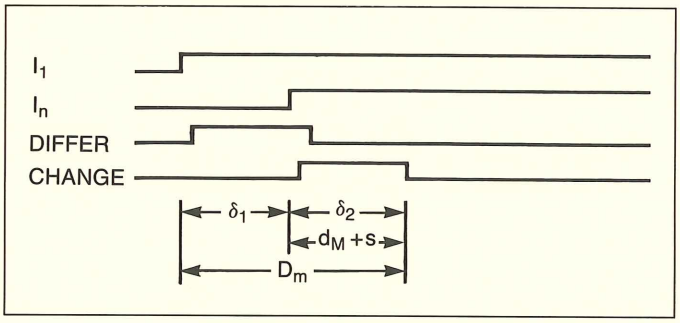


Figure 5. Symmetrical delay element MIC timing diagram.

trailing edge of CHANGE, which clocks the flip-flops, is delayed by D_m. However, the delay element itself only delays a signal by $D_m/2$. If the delay element is a pure delay, then DIFFER had better not go low while the inputs are still permitted to change. If DIFFER does go low, any input signal changing near the end of the delay period will drive DIFFER high again and generate additional clock pulses that will pass through a pure delay. Thus, if a pure delay is used, we have the additional restriction

$$D_m/2 \geq \delta_1,$$
$$\delta_1 + \delta_2 \geq 2\delta_1.$$

By using an *inertial delay*, DIFFER can be permitted to change $D_m/2$ before the end of δ_1. No pulse shorter than $D_m/2$ will pass through the delay element. But since the delay is inertial, any input change at the end of δ_1 will extend the trailing edge of the clock pulse by $D_m/2$. Thus, using an inertial delay element, we have the additional restriction:

$$\delta_2 \geq D_m/2,$$
$$\delta_1 + \delta_2 \geq (3/2)\delta_1.$$

Up to now, these delay-induced limitations have always prevented self-synchronized machines from operating at maximum speed.

An optimum clock generator

To achieve maximum operating speed, the designer must overcome three problems caused by clock structure:

(1) The symmetrical delay-element limitation

(2) the extra clock-pulse generated when the final stable state is entered

(3) the fixed (constant) time between intermediate internal-state transitions during a multiple output-change perambulation

Solving the first problem, the limitation imposed by the symmetrical-delay element: As we saw earlier, DIFFER should be high for the entire input-change period δ_1 to prevent inputs changing late in δ_1 causing DIFFER to go high more than once in a single input-state change. With a symmetrical-delay element, the period DIFFER is high is also the minimum period that DIFFER must remain low. By separating these two times, we can optimize each for speed. In fact, the optimum low time is zero and the machine would be "instantly" ready for the next input state.

We can separate DIFFER high and low by delaying the rising and falling edges of CHANGE by different amounts. In an asymmetrical delay element the design sets rising-edge delay but the falling-edge delay is "set" only by the technology. Figure 6 shows the timing for a MIC machine using such an asymmetrical-delay element. This element represents the optimum choice for speed. It solves the delay-element limit of previous clock generators.

The beauty of this asymmetrical-delay element is that the designer can base the D_m on the problem specification and the technology. It is easy to design the asymmetrical delay element so that DIFFER remains high throughout the time the inputs are permitted to change, without forcing an performance-degrading extension of the period between input-state changes.

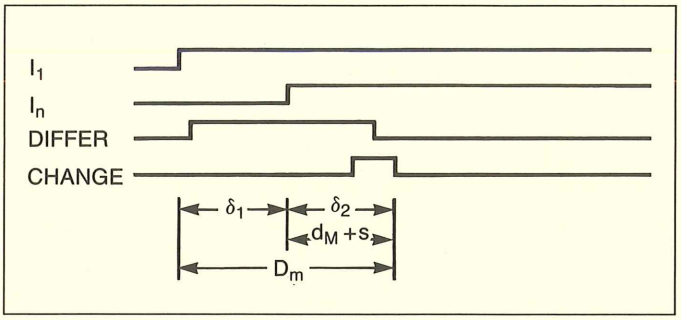


Figure 6. Asymmetrical delay element MIC timing diagram.

Solving the second problem, the extra clock pulse that occurs at the end of an MOC internal-state change:

An early indication that the next state is stable is needed. It's easy to determine a transition into a stable state from the flow table. If the present and the next internal state, are equal, the total state is stable; if they are not equal, then more states follow.

As the first step in solving the extra-clock-pulse problem, we added an output called MORE to the combinational-logic block. Figure 7 shows the resulting machine architecture. If more transitions are required, MORE will be high, otherwise MORE will be low. This "early finish" indication (low) needs to fed to the change detector with minimum impact on speed. Thus the second step is to connect MORE to the T input of a T flip-flop that is clocked by the trailing edge of CHANGE. When the machine is clocked with MORE high, the flip-flop's output changes. Such changes on a signal are exactly what the change detector is designed to process. This change of flip-flop output causes another clock pulse. If a clock pulse occurs with MORE low, then the T flip-flop does not change and the sequence ends.

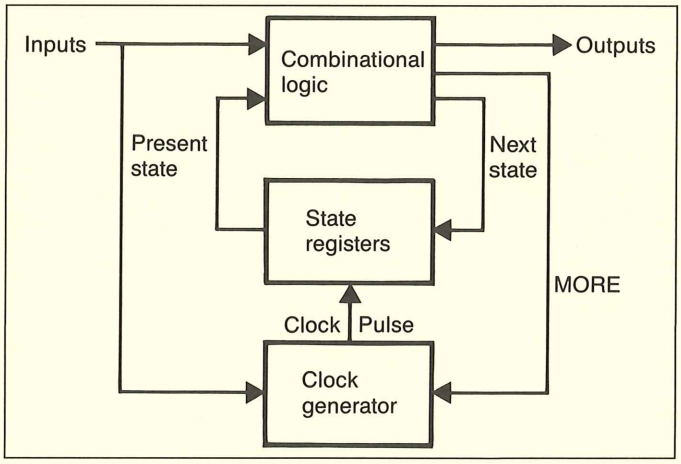


Figure 7. MOC machine with early final state indication.

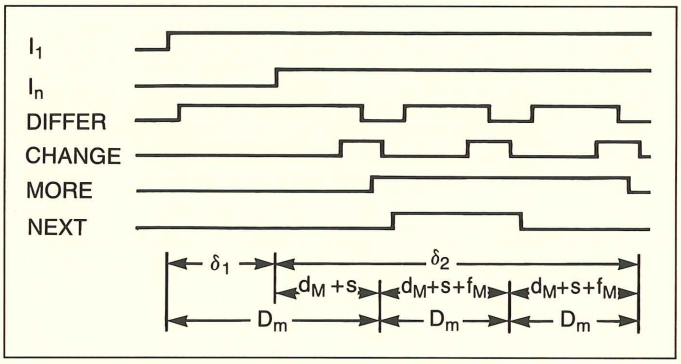


Figure 8. Early final state indication timing diagram.

It's essential that the T flip-flop be clocked on the trailing edge of CHANGE to guarantee the clock generator will accept the next T flip-flop output change, if there is one. Figure 8 shows one state transition caused by an input-state change and two state transitions caused by the T flip-flop. The output of the T flip-flop is named NEXT.

Figure 8 also shows the solution to the third problem—*optimizing the internal-state transitions during a multiple output-change perambulation*.

The delay-element time D_m for a state transition *may not need to be a constant*. We selected the first D_m value to insure that the first state transition would not malfunction. The machine designer can completely control all state transitions except the first. Since the input state must be stable, every state transition except the first appears as a single input-change transition to the clock generator; only the NEXT clock-generator input changes. For the combinational-logic block, only the state variables change, and since these variables come from the state register clocked by CHANGE, they arrive at the combinational-logic block simultaneously. For all state transitions after the first, D_m can be reduced. To exploit such reduction, we need to make our D_m variable.

An Asymmetrical Variable-Delay Element

Our design scheme centers on the use of an asymmetrical-delay element. Delay elements comprise circuitry inserted (usually in the feedback path) to slow a signal change. In previous asynchronous sequential machines, the delay elements have been one of three types: a monostable multivibrator, a pure delay, or an inertial delay.

There are several ways to create an asymmetrical delay element. One is the resistor-capacitor-diode combination shown in Figure 9.

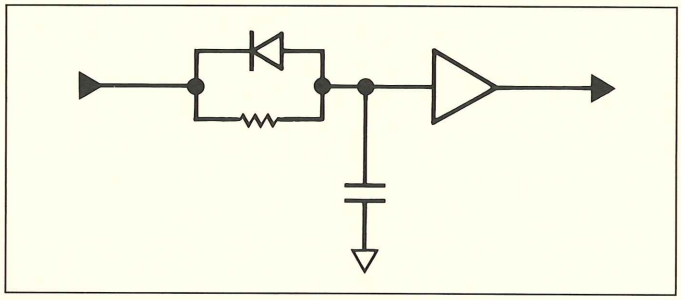


Figure 9. Simple asymmetrical delay.

Although simple, this method has serious limitations. Large delays require large capacitor values and the driving-circuit output impedance becomes important. Because capacitor voltage rises slowly, a buffer with hysteresis must be used. With the input low, the circuit's noise immunity is degraded by the diode forward-voltage drop. With the input high, noise is coupled into the node due to the high source impedance. Designers find nominal delay time is difficult to control.

The circuit in Figure 10[9] is an improvement over the simplistic circuit of Figure 9. The delayed (rising) edge propagates along the serial path while the non-delayed (falling) edge goes directly to the output gate. The falling edge of CHANGE lags that of DIFFER by only a technology-dependent gate delay (figure 6).

Total delay is spread over many smaller delays. This helps solve problems caused by noise and slow rise time. The input low-noise margin is restored since the second input of each AND gate is connected to DIFFER. We built and tested the scheme of figure 10 using 74F08s as AND gates. The performance was excellent, with one exception. Propagation delay varied significantly with temperature due to input-threshold drift of the 74F08 (−4.4 millivolts per degree C). Propagation delay varied 20% over 0 to 70 degrees C.

We solved the problem by replacing the delay AND-gate string with a shift register (or counter), (figure 11). A low on DIFFER holds the shift register in a reset condition. When DIFFER changes to a high, the reset is removed and the register begins to shift the high on the serial input down the register. Only clock-oscillator accuracy now limits delay accuracy. Since DIFFER also turns the oscillator on and off, the time to the first shift is a predictable and constant.

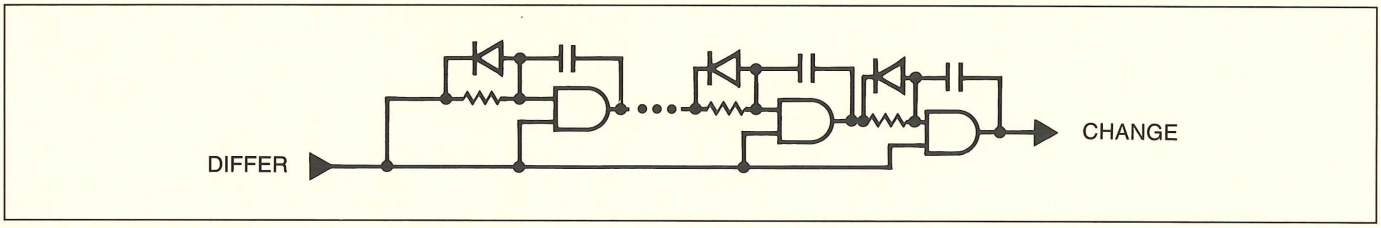


Figure 10. Series/parallel asymmetrical delay.

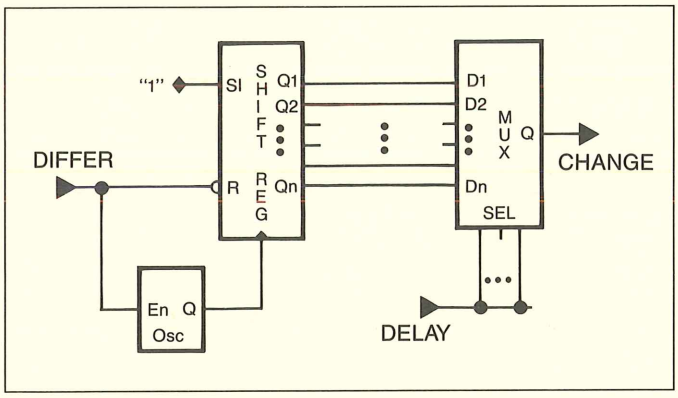


Figure 11. Improved asymmetrical delay element.

Programmable delay enables the designer to increase machine throughput by customizing the delay to machine state. The delay is changed using the binary vector DELAY to select the proper input in the N-to-1 multiplexer. The improved asymetrical delays can be much longer than those reasonable for the resistor-capacitor-diode network.

With programmable asymetrical-delay the designer has a new freedom to optimize performance using auxiliary information about the machine being designed. For example, the designer may know that when state s_i is reached, only one input will need to change to cause the next-state transition, but in state sj, the next-state transition will be caused by a multiple input change.

Suppose an additional set of output signals, called DELAY, is introduced just when the state variables are produced. DELAY is a function of the states and inputs just as are the state variables. We use these additional outputs to control the delay time D_m. Designers can customize the delay time leaving each state, state by state, for the machine behavior required. The number of different D_m times needed determines the number of bits in DELAY. The timing diagram in figure 8 shows two values for D_m, a long time for the first state transition following an input-state change and a short time for all subsequent state transitions, but each state transition could have had different delay times.

Extending Self-Clocked Machines

Designers can easily extend the self-synchronized MIC machine to either the unrestricted input-change (UIC) mode or the speed-independent mode.

Unrestricted input-change mode

Most asynchronous designs assume the machine will operate in the fundamental mode—once the machine perceives a change of input-state, it will reach a final stable state before permitting the next input state-change. In the UIC mode, however, this assumption of fundamental mode is violated. Since timing relationships between the input changes are not constrained, ambiguous input states result. For example, when one input change is followed closely by another, should there be one or two input states? Or what should the machine do when an input changes during a state transition?

The designer's problem is to describe what is a satisfactory outcome when faced by these ambiguities.

As a first step in defining a satisfactory outcome, the concept of an *n-cube* and *spanning* must be employed. In this concept, there are 2^n binary vectors with n components. This set of all 2^n vectors forms an n-cube. A subset of 2^m n-dimensional vectors having the same value in n-m locations form a subcube. Given a set of n-dimensional binary vectors, V, the set of vectors *spanned* by V is the smallest subcube containing every member of V. This set of vectors spanned is written T(V). The concept of spanning is best illustrated by example. If

$$V=\{01001,01100\},$$

then T(V) would be

$$T(V)=\{01000,01001,01100,01101\},$$
$$=\{01-0-\}.$$

(This definition of spanning differs from the definition of spanning used in linear algebra.)

$T(i_a,i_b)$ would be all the input states that might be passed through as an input vector changed from i_a to i_b. The machine will not necessarily respond to all input states in the set of states spanned. For example, with T as above, the machine may respond as if any of these input sequences occur: (01001, 01000, 01100), (01001, 01101, 01100), or (01001, 01100). But it will not respond so to the sequence: (01001, 01000, 01101, 01100) since once an input changes, it will not change back. If more than one input changes, some input states in the set of states spanned must be skipped. However, observe that the states in the input sequence are not necessarily all single input changes (Hamming distance one).

A *satisfactory outcome*[10] of an input change from i_1 to i_n with initial state A is any stable state (s,i_n) that could have been reached by a sequence of input changes i_1, i_2, ... i_n where, for j=2 to n, i_j is a member of $T(i_{j-1},i_n)$.

Extending the MIC machine to the UIC mode is straightforward. All the inputs pass through a transparent latch before presentation to the machine. While the machine is in a stable state, the gate signal for the latch is true and the latch inputs pass through to the machine. While the machine is in transition to a final stable state, the gate signal is false and the latch outputs are frozen. Since the machine is busy if DIFFER, CHANGE, or MORE are true, the latch-gate signal is the complement of (DIFFER OR CHANGE OR MORE). When some input changes, the gate is turned off, freezing the input state in the latch. This input state is processed and the latch is opened to capture another input state. As far as the machine is concerned, it sees only multiple input changes and operates in the fundamental mode. Thus, the input latch groups the input changes into a sequence of batches for the machine to process, and any such sequence eventually leaves the machine in a satisfactory outcome.

	x_1 x_2				y_1 y_2
	0 0	0 1	1 1	0 0	
1	1,0	2,0	4,0	1,0	0 0
2	2,0	2,0	3,1	3,1	0 1
3	1,0	2,0	3,1	3,1	1 0
4	2,0	2,0	4,0	4,0	1 1

Figure 12. Crumb Road problem flow table.

Designers should note that the UIC latch may exhibit metastable behavior; input changes may violate the set-up or hold times for the latches used. To compensate for this, extend δ_1 to allow the latches to resolve the metastable condition. The probability of non-resolution of a metastable condition is the negative exponential of the delay interval. Using moderate values of delay, designers can achieve very reliable resolution by employing high gain/bandwidth circuitry.[11]

Speed independent mode

Speed-independent designs are characterized by employing completion handshakes. Such a design is based on a chain of subcircuits, each subcircuit sending completion signals to its predecessor and responding to completion signals from its successor. Each subcircuit is free to operate at its own speed. In addition to being separated by the special completion signals, the input states are separated by a special input state called a *spacer.*

Each subcircuit cycles continuously through the sequence: output data, request spacer, output spacer, request data. When a subcircuit has responded to the input data from its predecessor, and its output data has been accepted by its successor, the subcircuit sends a completion signal (traditionally called S) to its predecessor requesting a spacer. A subcircuit outputs a spacer when one is requested by its successor and its predecessor has output a spacer. When a subcircuit has had a spacer input and its successor is requesting data, the subcircuit itself requests data by sending a signal (traditionally called D) to its predecessor.

Although the self-synchronized machine itself is not speed independent, it can be made to operate in a chain of speed-independent subcircuits. For a self-synchronized machine to operate in speed-independent mode, S and D completion signals must be added and sent to the predecessor subcircuit and the machine must respond to completion S and D signals from the successor subcircuit. Since there are no timing relationships between subcircuits, a self-synchronized sub-machine (subcircuit) must operate in UIC mode.

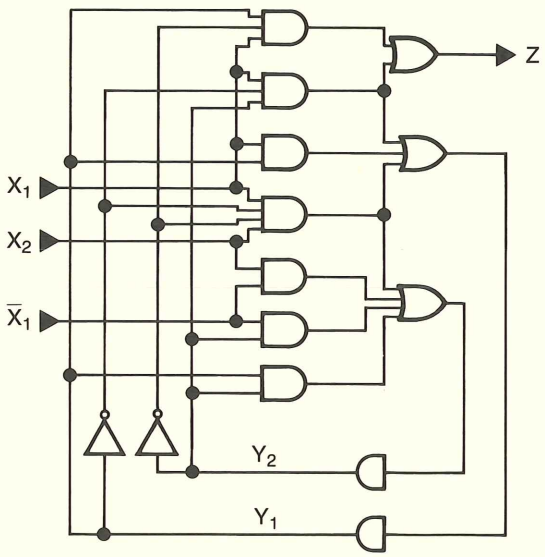
To build integrated circuits vastly larger than those of today, we must solve the problems of distribution delay and skew among various data and clock lines. One way to do this is to organize the system into pools of circuitry interconnected according to the spirit of speed-independence. Alternative conventions for interlocked signaling among self-timed units of a speed-independent organization have been proposed.[9] .

For the near future, we feel that it's feasible to organize a large chip as a single, globally equipotential region. Technology-dependent designs and conservative system-clock rates can compensate somewhat for the fact that an ideally synchronized, lockstep operation runs with a period that increases in proportion to the delay diameter representing the differences in the various distribution paths. Tightly synchronized and highly regular large arrays are an important approach to the parallel resolution of large problems. In one-dimensional arrays of processors, clocks can be pipelined at a rate independent of the array size, but arrays of higher dimension can't. "The only [practical] means of improving performance [in really big ICs] are technology improvement, clever design, and self-timing."[12]

A Design Example: The Crumb Road Traffic Control Machine

Unger's problem[1] is a well known example of all the things that can go wrong in asynchronous design. His problem also demonstrates how simple design can be with self-synchronization.

This example employs, in one coherent design, nearly all the ideas we have presented. The problem (see [1] for a complete description) involves designing a sequential machine to control the traffic at the hypothetical intersection of Crumb Road and Route 1. After describing the problem with words, Unger developed a flow matrix (figure 12) as a first attempt to solve the problem.



$$z = x_1 \bar{y}_1 y_2 + x_1 y_1 \bar{y}_2$$

$$Y_1 = x_1 x_2 \bar{y}_1 \bar{y}_2 + x_1 \bar{y}_1 y_2 + x_1 y_1$$

$$Y_2 = \bar{x}_1 x_2 + y_1 y_2 + \bar{x}_1 y_2 + x_1 x_2 \bar{y}_1 \bar{y}_2$$

Figure 13. Crumb Road problem sequential machine.

From this flow matrix, he derived logic equations and developed the sequential circuit of Figure 13. He intentionally ignored the design issues of the unrestricted input-change mode, critical races, essential hazards, and logic hazards. He then demonstrated how proper operation of this sequential circuit depends upon the relative magnitudes of the stray delays.

However, if this circuit did operate correctly, what would be its speed? To answer this assume the gates and inverters have a minimum/maximum propagation delays of 3 and 7 nanoseconds and the delay element's maximum delay is 1.5 times its minimum. These values fairly represent the real times for 74Fxx series parts. Since the two input variables, x_1 and x_2, can change at any time, the machine operates in unrestricted input-change mode and δ_1 is 0. Using these values and the previously developed timing analysis, the minimum delay-element time is now calculated as

$$D_m \geq \delta_1 + d_M - d_m = 0 + (7+7) - (3+3) = 8 \text{ ns},$$

and the time between input states calculates as

$$\begin{aligned}\delta_1 + \delta_2 &\geq \delta_1 + n(d_M + D_M) + (d_M - d_m), \\ &\geq 0 + 1(21 + 1.5\times 8) + (21 - 6) = 48 \text{ ns}.\end{aligned}$$

We will convert this abysmal failure of an asynchronous machine (as Unger demonstrated) into a practical design by simply adding a change detector, UIC latch, and state registers. The result (shown in figure 14) demonstrates the power of self-synchronized design.

Again, assuming the flip-flops have a set-up time of 2 nanoseconds and a maximum propagation delay of 10 nanoseconds, with the gate delays the same as the previous case, the delay element time is:

$$D_m \geq \delta_1 + d_M + s = 4\times 10 + (7+7) + 2 = 56 \text{ ns}.$$

To to compensate for a potential metastable condition in the UIC latch, we assign δ_1 the value of four times the flip-flop propagation delay. Metastability occurs when the latch inputs change at a time that violates the latch's specified set-up or hold time. Since the change detector's combinational logic has a minimum propagation delay of 6 (=3+3) nanoseconds, the actual minimum delay-element value is 50 nanoseconds. Thus the minimum time between input states is

$$\begin{aligned}\delta_1 + \delta_2 &\geq \delta_1 + f_M + d_M + s + (D_M - D_m), \\ &\geq 40 + 10 + (14) + 2 + ((75+14) - (50+6)) = 99 \text{ ns}.\end{aligned}$$

If simultaneous input changes are improbable, the UIC latch is unnecessary (reasonable in this design problem) and the minimum delay-element time can be reduced from 50 to 10 nanoseconds. This also reduces the time between input states from 99 to 39 nanoseconds. The resulting self-synchronized design, without the UIC latches, will then essentially match the speed of the Huffman-Moore machine without self-synchronization. Even more important, it will work.

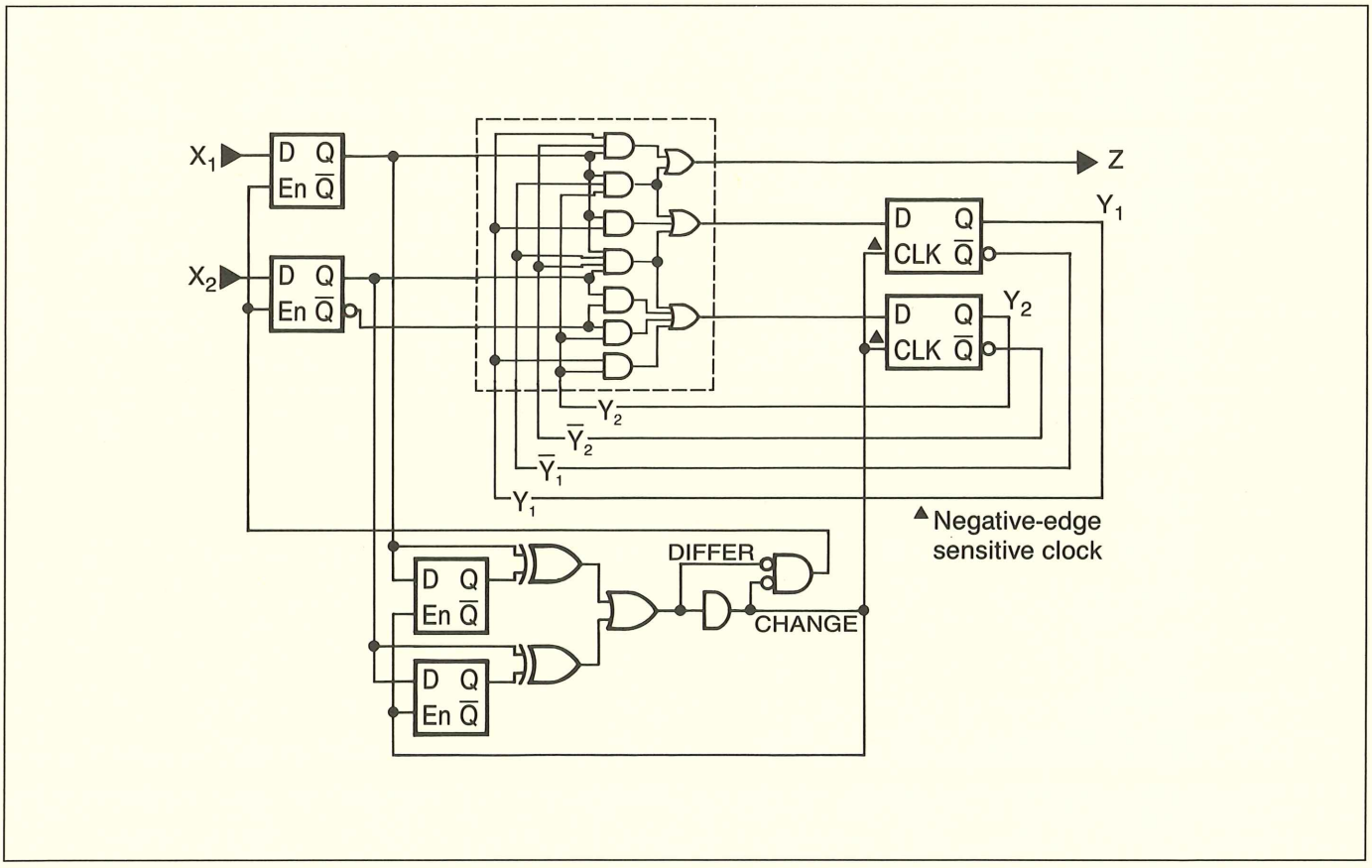


Figure 14. Crumb Road problem self-synchronized machine.

Conclusion and Summary

In a digital system, asynchronous design is called for when two modules do not share a common clock, or when a rapid machine response to input changes is required. In doing such designs, designers usually run into behavioral anomalies such as races and essential hazzards and have to employ involved methods to avoid them. Our approach simplifies all that. This approach to self-synchronization enables the designer to negate the anomalies that the differing and finite response times of real circuits introduce.

This self-synchronized machine can operate with the speed of a classic Huffman-Moore asynchronous machine. Its implementation can be as simple as a clocked machine. This design style adapts to a almost any real-life implementation—boards, arrays, even vastly separated elements on extremely large integrated-circuit chips or wafers.

The clock is the key, the clock generator and its change detector and delay element. Our generator for multiple inputs is a digital differentiator with programmable assymetrical delay. We found this to be the best organization for change detection in self-synchronized machines running in the multiple input-change (MIC) mode.

The asymmetry permits internal clocking at the maximum rate for a chosen technology; no superfluous clock pulses are generated at the end of a multiple output-change (MOC) state sequence; and each clock pulse can be tailored to the minimum duration necessary for a particular state transition. Thus, the design's ultimate speed is limited only by the speed of the logic components being used and the durations and complexities of the state-transition sequences necessary to the application.

Our self-synchronized design is suitable for a small, localized circuit module. For a larger network of communicating modules, two extensions to the design style can be considered: unrestricted input-change (UIC) mode and speed-independent networks.

Real inputs may not maintain the intervals between input changes fundamental-mode operation require. A self-synchronized-machine can quarantee a satisfactory, although possibly non-deterministic outcome of an input sequence. The designer can do this using the UIC technique of applying the local clock to an input latch (shown in the Crumb Road design).

The self-synchronized machine can operate in a network of speed-independent subcircuit modules by means of hand-shaking signals such as space/data control tokens, or a similar protocol (for example, see [13]). In this mode, the UIC mode machine would be an excellent choice as an I/O module for a larger network.

For More Information

For more information call Don Kirkpatrick, 629-1236, 92-716. □

Bibliography

[1] Unger, S.H., *Asynchronous Sequential Switching Circuits*, Wiley Interscience, New York, 1969.

[2] Tracey, J.H., "Internal state assignments for asynchronous sequential machines," *IEEE Trans. Electronic Computers*, vol. EC-15, pp. 551–560, Aug. 1966.

[3] Liu, C.N., "A state variable assignment method for asynchronous sequential switching circuits," *J. ACM*, vol. 10, pp. 209–216, 1963.

[4] Tan, C.J., "State assignments for asynchronous sequential machines," *IEEE Trans. Comput.*, vol. C-20, pp. 382–391, April 1971.

[5] Bredeson, J.G., and Hulina, P.T., "Generation of a clock pulse for asynchronous sequential machines to eliminate critical races," *IEEE Trans. Comput.*, vol. C-20, pp. 225–226, Feb. 1971.

[6] Rey, C.A., and Vaucher, J., "Self-synchronized asynchronous sequential machines," *IEEE Trans. Comput.*, vol. C-23, pp. 1306–1311, Dec. 1974.

[7] Chuang, H.Y.H., and Das, S., "Synthesis of multiple-input change asynchronous machines using controlled excitation and flip-flops," *IEEE Trans. Comput.*, vol. C-22, pp. 1103–1109, Dec. 1973.

[8] Unger, S.H., "Self-synchronizing circuits and nonfundamental-mode operation," *IEEE Trans. Comput.*, vol. C-26, pp. 278–281, March 1977.

[9] Seitz, C.L., "Chapter 7, System Timing" in Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison-Westley, Reading, Massachusetts, 1980.

[10] Unger, S.H., "Asynchronous sequential switching circuits with unrestricted input changes," *IEEE Trans. Comput.*, vol. C-20, pp. 1437–1444, Dec. 1971.

[11] Flannagan, S.L., "Synchronization in a CMOS technology," M.S. Thesis, Oregon State University, May, 1982.

[12] Fisher, A.L., and Kung, H.T., "Synchronizing large VLSI processor arrays," *IEEE Trans. Comp.*, vol. C-34, pp. 734–740, Aug. 1985.

[13] Franklin, F.A., and Wann, D.F., "Asynchronous and clocked control structures for VLSI based interconnection networks," *Proc. Symp. on Computer Architecture*, IEEE and ACM, Apr. 26–29, 1982, Austin, Texas, pp. 50–59.

Engineering Forum Aims at Building Business Excellence

Tek's ambitious business-excellence renaissance is off to a successful start. In manufacturing, the company has eliminated more than $100 million of inventory, increased inventory turns by better than 50%, installed MRP and won Class-A certification for a score of plants, embraced just-in-time systems—and lowered the manufacturing cost of sales.

The engineering community is also responding, with higher levels of involvement: getting closer to the customer, helping their colleagues in manufacturing and marketing, and—the theme of Tek's first Engineering Excellence Forum—recognizing engineering's opportunities to stimulate business results.

The Forum, held March 3 and 4 at the Greenwood Inn, focused on our critical need to really understand customer needs. After President/CEO Earl Wantland got the Forum started with opening remarks, Executive Vice President Wim Velsink laid down specific objectives. The overview was spelled out this way:

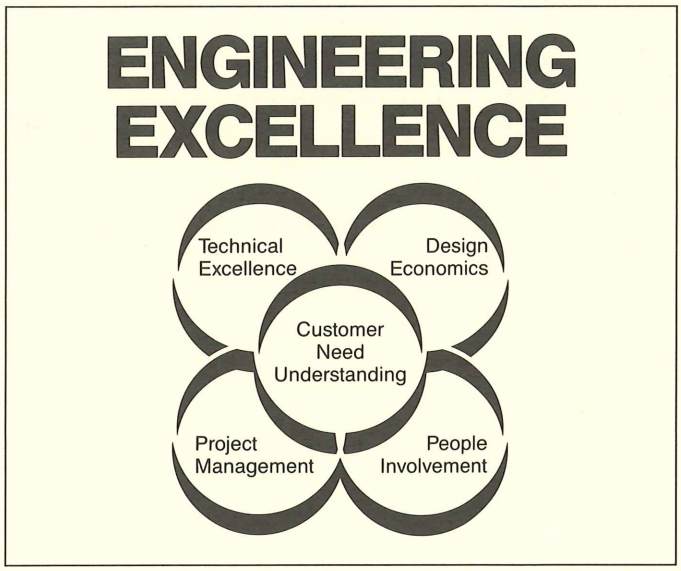
To "clearly establish" what engineering contributions are essential to achieving ongoing business excellence. . .(and) to communicate individual and organizational challenges of continued engineering development. . .

In his keynote address, Wim cited Tek's historic reputation for across-the-board excellence, and positioned the Forum as an opportunity to build on that.

The Forum was geared toward a three-fold program to dramatically reduce time-to-market through 1) improved customer-need understanding, 2) project management disciplines for improved execution, and 3) increased use of CAE tools. The big target: Reducing average time-to-market of successful products by 50% within three years.

In Wim's words: "A major, and perhaps the most fundamental, building block of business excellence is achieving a continuous flow of products that serve our customers' needs better than any alternative."

In addition to asking for drastic cuts in cycle times, Wim also challenged the Tek's engineering community to get customer



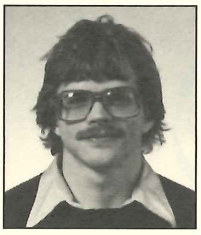
requirements defined before allocating major resources, and ensure that post-DC engineering milestones are met.

Presentations on increasing customer involvement and understanding customer needs were made by Dennis Brunnenmeyer (GVG), Wendell Damm (DAG-Logic Analyzers), Sue Grady (IDG-Interactive Display Systems, Graphic Workstations), Jerry Ashley (IDG-Graphic S/W Products, Graphics Terminals), and Dave Squire (IDG-Artificial Intelligence Machines).

Internal CAE tools were surveyed, and a panel (Rick Potter, Rick LeFaivre, Chuck Saxe, Jack Hurt, Jim Carden and Dave Brown) dealt with "Future Directions in CAE." Jerry Sullivan moderated.

Other Tek people contributing to the two-day Forum included Dick Knight, Denton Tarbet, Jim Carden, Tom Long, Dave Moser, John McCormick, George Kersels, Harry Anderton, John Sonneborn, Steve Ratner, Dave Friedley, Larry Kaplan, Gary Andrews, Jon Reed, Rose Marshall, Dave Cassing, Fred Hanson, Lyle Ochs, Mayer Schwartz and Binoy Rosario. □

Testing the Large Chip: Artificial Intelligence Shows Promise in Reconciling Functional with Gate-Level Testing



Eirik Fuller is a software engineer II in the Computational Algorithms Department of the Computer Research Laboratory. He joined Tektronix in 1985. Eirik holds a BS and an MS in math from Rensselaer Polytechnic Institute, Troy, NY.

Richard Sheng is a software engineer III in the Computational Algorithms Department of the Computer Research Laboratory. He joined Tek in 1984. While studying for his PhD in electrical engineering and computer science at the University of California, Berkeley, Richard was a summer intern at Fairchild's Artificial Intelligence Laboratory. Richard's BSEE is from National Taiwan University, Taipai, Taiwan.

Balaji Krishnamurthy is the manager of the Computer Algorithms Department of the Computer Research Laboratory. He joined Tek in 1984 from General Electric's Research Center in Schenectady, New York. Balaji has an MS in math from the Birla Instititue of Technology and Science, India. He also has an MS and PhD in computer science from the University of Massachusetts.

In the Computer Research Laboratory, we are developing an AI-based hierarchical test-generation system. By using high-level design information to guide the search process, we expect to speed up the automatic test-generation process to the extent that it becomes feasible for the large, complex circuits typical of today's VLSI designs.

What We're Trying to Do

With VLSI, digital testing confronts a wall of intractability. The complexity of advanced integrated circuits is far beyond the capabilities of current generation programs. Because test generation belongs to a class of intractable problems, there is no hope for an efficient deterministic algorithm. However, problems in this class can frequently be solved by heuristic algorithms. These algorithms are guided by heuristics intended to make them finish test generation quickly for inputs of practical interest. (However, they won't finish quickly for all possible inputs.)

The goal in digital testing is a test set, a sequence of input patterns. These patterns must, when applied to the circuit under test, detect all possible faults as discrepancies in the output values. Implicit in test generation is a fault model, which defines the list of all possible faults in a circuit. For reasons to be discussed, no fault model actually accounts for *all possible* faults. Incidentally, the test-generation problem is not concerned with fault isolation. In filtering out defective ICs, the aim is not to repair them. In process control or the repair of discrete circuits, fault isolation is a different problem.

The usual fault model for test generation is the single stuck-line fault model. In this model, a fault manifests itself as a single lead stuck at a logical value. Since each lead can be stuck at one of two values, there are two faults for each lead in the circuit. There are several reasons that this model is incomplete. There might be more than one fault. More to the point, individual faults might not manifest themselves as stuck leads. The circuit is modeled as interconnected logical gates; a physical defect within a gate might not produce the same logical behavior as one of its leads being stuck.

These are reasonable objections, yet the single stuck-line fault model is widely accepted. Multiple faults are unlikely, and it's not clear that including them in the fault model is worth the added complexity. Modeling digital circuits with gates has the advantage of being useful for all logic families; a fault model too closely related to any one logic family defeats this advantage. Informally, this fault model can be justified: A test set which tests all single stuck-at faults in a circuit can, in some sense, fully exercise the circuit.

The problem of generating a test set eventually reduces to finding tests for individual faults. This problem, a search on the input space, is intractable. The real expense in this search is backtracking, the process of restoring the circuit to a previous state after an inconsistency is detected. The expense of backtracking can be reduced in two ways: make correct decisions as often as possible, and detect inconsistencies as soon as possible. Conventional test generation programs use heuristics based on controllability values to guide the decisions in the search process.

As an example, consider the circuit fragment shown in figure 1. Suppose we want to generate a test for lead *E* stuck at one. This requires two independent tasks. One is to justify a value of zero on *E*. The other is to push the fault-sensitive value forward from *E* along a path to some output.

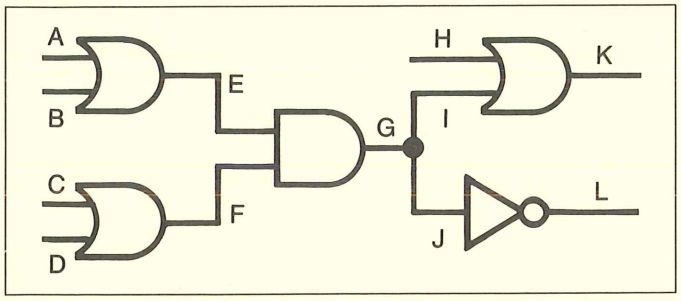


Figure 1. Suppose we want to generate a test that will determine if lead *E* is stuck at one. To do this we would have to justify a value of zero on *E* and to push the fault-sensitive value forward to some output.

In general, each task generates others as it is performed. For instance, sensitizing a path from *E* requires a one on *F*, as well as sensitizing a path from *G*. Also, there are decisions to make when there are multiple ways of performing tasks; for instance, a path sensitized from *G* can go through either *I* or *J*.

Justifying a one on *F* requires a one on either *C* or *D*. A typical heuristic for deciding which to try first would use a controllability measure, with the idea that the lead on which it is easiest to justify a one is the least likely to require backtracking. On the other hand, justifying a zero on *E* requires zeros on both *A* and *B*. In this case, the least controllable lead would be processed first, since backtracking, if it must occur, should occur as early as possible to minimize the wasted time.

In essence, the problem with typical test-generation heuristics is that they work as well on random circuits as they do on useful ones. Many intractable problems of practical interest are solvable by algorithms which fail on contrived counter examples of no practical value. To be useful, heuristics for test generation should work on well-designed circuits; that they fail on useless, arbitrary circuits is a reasonable price, in fact a bargain. It is simply impossible for a program which treats a circuit as an unstructured collection of interconnected logical gates to make this distinction.

While a gate-level description of a large digital circuit is technically accurate, it is worthless for someone trying to understand how the circuit works. A good design is a structured, layered hierarchy, in which each level consists of a relatively simple configuration of well-defined modules. This is the way an engineer perceives a circuit when designing it, or when manually generating a test set for it. We think a test-generation program should be able to guide its search by using this information about the structure of a circuit, the semantics of the design.

This idea is not new. Treating a circuit as a collection of high-level modules for the purpose of test generation is called *functional testing*. The foremost objection to this approach relates to its fault model. As discussed previously, the inability of the single-stuck-line fault model to account for all possible faults is excused by its reasonably uniform coverage. At higher levels, this uniformity is progressively diminished.

How We Do It

Our approach to test generation is based on the goal of reconciling high-level reasoning with a gate-level fault model. For high-level reasoning, we use a new control structure, in which operations on gates are replaced by operations on arbitrary modules. These operations are supported by an expandable rule base. For this rule base, we are developing a language in which a designer can encode instructions to the test-generation program for performing operations required for the fault-searching process.

Some examples of the rules are shown in figure 2; these examples are taken from our prototype (described later). We concocted the voter circuit out of comparators and multiplexers, also shown. As indicated, even gates are treated as modules, with their own rules. The informal language used in

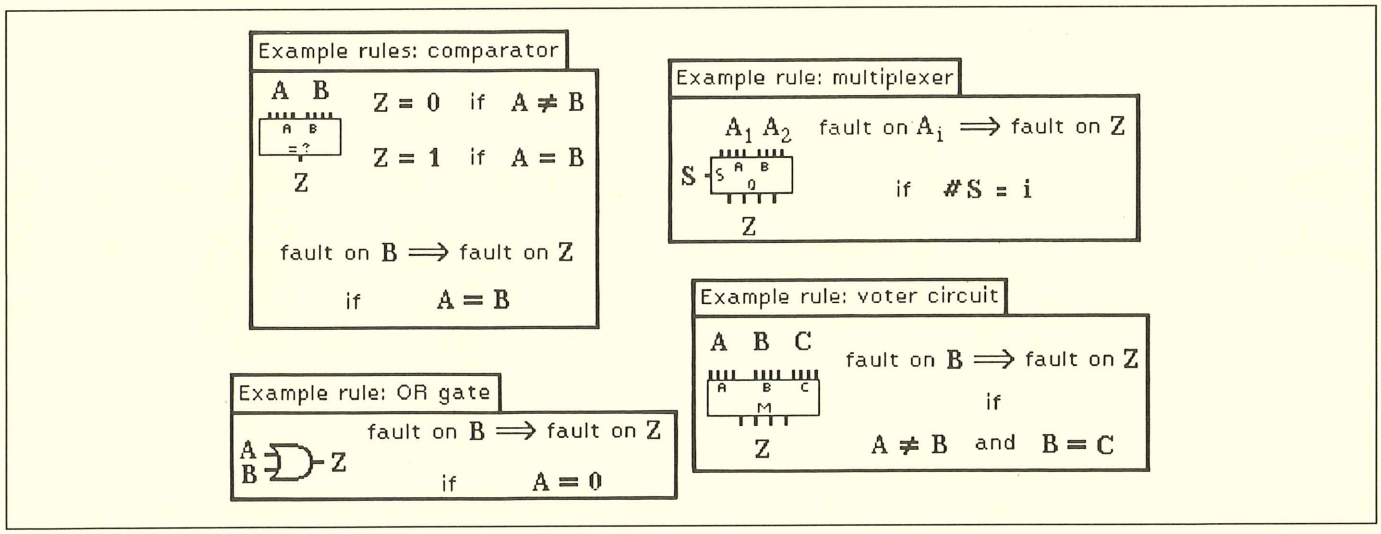


Figure 2. Some of the rules from CRL's test system prototype. The language used is an informal prototype of the *formal* language expected for a production system. The voter circuit shown in the lower right window is made up of the comparator, multiplexor, and OR gate shown in the other windows.

these examples is a rough indication of the language we expect to be using for writing actual rules.

To reconcile this high-level reasoning with a gate-level fault model, we use a variable representation of the circuit structure, in which a module can be replaced by its innards.

To test a fault which is inside of a module, we perform this replacement as many times as it takes to make the faulty lead visible. The circuit is represented at a mixed level, since there is no immediate need to open any other modules. While slightly more complicated than a highest-level representation, this mixed-level view of the circuit is far less complicated than a purely gate-level view, and there is no reduction in fault coverage.

This ability to open a module and reason about its innards (which allows a gate-level fault model) has a useful side effect. The rules for high-level modules, used by the control structure in its searching, are typically incomplete; that is, the rules ignore some possibilities. By enumerating the most likely alternatives and neglecting the more obscure possibilities, the rules implicitly define the heuristics for the search. Without the capability of opening modules, this would make the search nondeterministic. With this capability, an incomplete rule can be completed by adding a final case which says, in effect, "if all else fails, open my box." This "trick" bottoms out at the gate level, where all rules are complete.

This approach to test generation yields two immediate improvements in efficiency. First, the effective size of the circuit is reduced (the number of modules at a high level is far less than the number of gates); even for internal faults, most of the circuit is represented at the highest level. Second, backtracking, which would otherwise occur inside the module boundaries, is sidestepped (bypassed by the rules for that module). These improvements are useful, but the real hope is that better heuristics will reduce external backtracking as well.

Our Prototype

We demonstrate these concepts with a prototype of our test-generation system. In addition to the test-generation program, written in C, we have a user interface, written in Smalltalk. Consisting of a rudimentary schematic editor with a subtasking facility, the interface animates the test-generation program. The test-generation program is run as a subtask, using pipes to send and receive commands. The interface interprets the commands by highlighting the specified circuit elements in the schematic view.

The interface is one entry in a long and growing list of examples at the Computer Research Lab that show the power of the Smalltalk programming environment in the rapid development of graphics prototypes. The interface, in addition to serving its purpose of demonstrating our research, might help us to evaluate heuristics as we develop them for the test-generation program. Our prototype, including the C program, runs on a Tektronix 4404 workstation.

Figure 3 shows a "snapshot" from the animation interface. The top-level circuit, a voter circuit, is in the window labeled "43major." Its three modules are detailed in the other three windows. The highlighted circuit elements show the progress

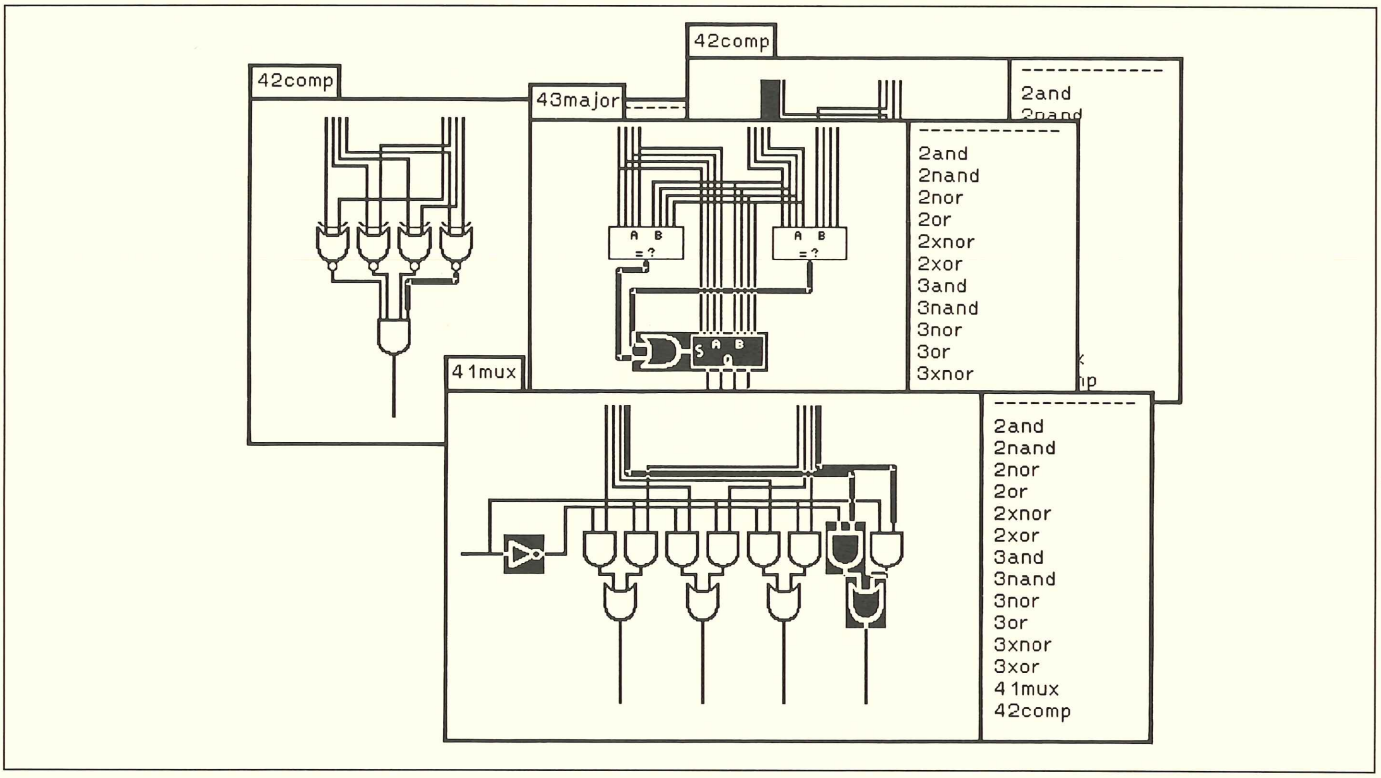


Figure 3. Animation shows, by dynamic highlighting, how the test-generation algorithm is progressing in the voter circuit and its major segments.

of the test-generation algorithm. The voter circuit is a circuit we devised at lunch one day. Its output is the most frequently occurring bit pattern on the inputs.

The Future

Many research issues remain to be explored. The rules we used in our example were manually coded at a low level, with little in the way of heuristics. We expect to be able to encode these rules in a higher-level language, with provision for more heuristics. After a compiler is written for this language, these rules can be compiled into a rule base, for interpretation by the control structure.

We also want to extend our ideas to sequential circuits; at present they work only for combinational circuits. For this extension we are particularly optimistic about the usefulness of high-level reasoning. We are also developing refinements to the search process, which we expect to be useful even for gate-level reasoning. Finally, we intend to enhance the animation interface.

For More Information

For more information call Eirik Fuller, 627-6410 (50-662). □

Tolerances Too Tight: Aluminum-Extrusion Vendors Shy Away From Doing Business with Tek

Dick Borts
Metals Materials Management

Two major vendors of aluminum extrusions have told us they will no longer accept orders from Tektronix. In addition, two other vendors have said they would prefer not doing business with Tek. These four vendors are the cream of the crop in extrusions.

The reasons given are Tek's unusually tight dimensional and finish requirements. They feel that Tek designers are not well informed about extrusions and their inherent limitations. Two vendors say that 60 percent of all Tek-ordered material is rejected at the extrusion press, creating a situation in which doing business with Tektronix is not feasible. These four companies do business with H-P, Xerox, IBM and other companies with few problems.

We suspect most tolerances specified on Tek drawings are tighter than they need to be. We have found tolerances of as tight as 0.005" where ±0.020" would have been acceptable.

To salvage our business relationships with other extrusion vendors, we need to ease our unnecessarily tight dimensional and finish specifications.

We have a list of specific part numbers which we have been asked to provide relief. Tek users of these parts will be contacted soon. We ask your cooperation in working with us to solve this problem.

Melvin Krusell, general manager of Futura Home Products, has offered to conduct a seminar on the design, fabrication, and application of aluminum extrusions. If you are interested, please contact Dick Borts, 627-2741, or Bill Gilbert, 627-1154. Both are located at MS 16-298. □

CONNECTIONS. . . Lookahead Carry Speeds Up Binary Addition/Subtraction and Binary/BCD Addition—Less Logic Too

By Virgil LaBuda, staff applications engineer, Motorola, Inc.

By employing the logic of *lookahead carry* in any of its many forms, the designer can significantly shorten the time needed to complete addition/subtraction. In doing so, the designer faces various trade-offs of amount of the additional logic required versus amount of computation time saved.

This article in the *Connections* series (see box) demonstrates the optimum use of components to realize binary addition/subtraction and binary/BCD addition within minimal space and with the shortest signal delays. By using *lookahead carry*, as detailed here, designers can significantly shorten addition/subtraction time.

Although the internal logic of most computers employs binary numbers, input/output equipment generally use decimal numbers. Because most logic accepts only two-valued signals, decimal numbers must be encoded in binary. This has caused a family of binary codes that represent decimal numbers to evolve. While the only requirement for a code to be valid is that each decimal digit be represented by a unique combination of binary digits, those codes adopted industry-wide are distinquished by having some specific purpose rather than employing an arbitrary assignment.

The sets of binary codes for decimal digits are classified as either weighted or non-weighted. In a weighted code, the position of each code bit is assigned a weighting factor. The code is converted to its decimal equivalent by summing the products of each positional weight times the value at that position. For example, in a 4-bit weighted code with a weighting scheme of W(3), W(2), W(1), and W(0), the decimal value represented by the coded representation N(3)N(2)N(1)N(0) would be N(3)W(3) + N(2)W(2) + N(1)W(1) + N(0)W(0).

Two examples of weighted codes are the binary-coded decimal (BCD) and the 6-3-1-1 code. The 8-4-2-1 BCD code is the simplest, with each decimal digit represented by a 4-bit-wide binary equivalent; the weighting factors are 8,4,2, and 1, proceeding from the most-significant bit (MSB) to the least-significant bit (LSB). In the 6-3-1-1 code, the weighting factors work the same way.

Three examples of non-weighted codes are the Excess-3, 2-out-of-5, and Gray. Excess-3 is formed from the BCD code by adding 3 (0011) to each decimal representation. The 2-out-of-5 code is used primarily in error checking. In the Gray code each successive decimal digit differs at exactly one bit position. This article will focus on using the BCD code and straight-binary representation for adder/subtractor implementations.

For numbers of even modest size (e.g., 16-bit binary) most of the addition/subtraction time is consumed in delivering a carry-in bit to each bit in the number. This is especially true when a carry is rippled across the entire length of the number. By employing the logic of lookahead carry in any of its many forms, the designer can significantly shorten the time needed to complete the addition/subtraction. In doing so, the designer faces trade-offs of the amount of additional logic required versus computation time saved.

> The Connections Program is a critical part of Tek's strategy for the computer aided engineering market. But "Connections" is more than a marketing stategy, it's a way for Tek circuit designers to access the products and the know-how of a score of IC vendors. Wes Brunning, Connection's program manager (629-1488) can provide additional information on how Connections can help Tek designers.

Motorola's MCA2500ECL Macrocell Array library contains half adders, full adders, as well as building blocks useful in generating these computation elements. This article demonstrates the optimum use of components in realizing binary addition/subtraction and binary/BCD addition with minimal space and with the shortest signal delays. Comparable building blocks for Motorola's other semicustom macrocell array are available in their respective array libraries.

The Lookahead Carry Concept

Let's examine what occurs at the K-th bit position during the addition of two binary numbers, each larger than 'K' bits. The two bits being added will be called A(K) and B(K) (referred to as augend and addend). Let C(K) represent a possible carry-out generated by such an addition, and C(K^{11}) represent a

possible carry-in to the K-th bit position. A quick reflection on the situation reveals that a carry-out, C(K), is generated when there is either a carry-in of Logic "1," C(K¹1), and at least one of the two bits being added, A(K) or B(K), is a Logic "1"; or both A(K) and B(K) are Logic "1". A truth table reveals this relationship:

$$C(K) = A(K) \bullet B(K) + (A(K) \oplus B(K)) \bullet C(K-1) \quad (1)$$

This truth table reveals that a carry-out will be produced if either 1) A(K) and B(K) are each Logic "1" (A(K)•B(K)) irrespective of the value of the carry-in, or 2) either A(K) or B(K) are a logic "1" and a carry-in of logic "1" exists (A(K)⊕B(K)•C(K¹1)). The principle of lookahead carry is that the carry-out is determined from an examination of the addend and augend. But how can the concept of lookahead carry be extended to number systems of other than base 2?

Clearly, the above relationship was developed by considering the events occurring in a binary system. If instead these events are viewed from a general stance, it can be said that a carry-out is produced when either 1) a carry-out is generated solely by the two numbers at the K-th bit position or 2) a carry-out is generated solely by the presence of a carry-in to the K-th bit position. Case 1 occurs when the sum of A(K) and B(K) equals or exceeds the base of the number system in question. Case 2 occurs when the sum of A(K), B(K), and C(K-1) equals the base of the number system in question. It has proven convenient to express these concepts as follows:

$$C(K) = G(K) + P(K) \bullet C(K¹1) \quad (2)$$

This expression states that a carry-out from the K-th position, C(K), is produced when 1) it is "generated" at the K-th bit position when the sum of A(K) and B(K) equals or exceeds the base of the number system, or 2) it is produced by the presence of a carry-in of logic "1" to the K-th bit position, C(K¹1), and the sum of A(K) and B(K) is one less than the base of the number system. The second case can be thought of as the sum of the addend and augend at the K-th bit position "propagating" (P(K)) the carry-in through to the carry-out position. In the case of a binary (base 2) system, one can intuitively generate the boolean expressions which fit the general form of equation 2. Reference to equation 1 shows that G(K) = A(K)•B(K) and P(K) = A(K)⊕B(K). Extending the concept of lookahead carry to other number systems, notably binary-coded decimal (BCD), involves developing the proper boolean expressions to satisfy the generic definition of the generate (G(K)) and propagate (P(K)) (see references).

Equation 2 focuses solely on the events that occur at the K-th bit position, and as such represents a lookahead expression spanning one numeral (for whatever numeric radix of interest). This recursive relationship is easily expanded to provide a lookahead carry expression spanning any number of numerals. That is, for a number 'N' numerals long, with the first numeral being the least-significant numeral and the N-th numeral the most significant, the lookahead expression providing the carry-out from the N-th numeral is:

$$C(N+1)=G(N)+P(N)\bullet G(N-1)+P(N)\bullet P(N-1)\bullet G(N-2)+\ldots+P(N)\bullet P(N-1)\bullet\ldots P(1)\bullet C(1) \quad (3)$$

where 'C(1)' is the carry-in to the least-significant numeral.

Clearly this only provides the carry-out for the N-th numeral, whereas every numeral must receive a carry-in, whether produced from a ripple or lookahead source. For most numerals in a number, supplying the carry-in with the lookahead technique is the most expedient approach. Supplying each and every numeral in a number with its carry-in in such a fashion is termed *100% lookahead*. However, this necessitates the production of lookahead logic blocks whose sizes rapidly become onerous as their number lengths increase; 100% lookahead is unrealistic with the number lengths routinely used by current computational architectures. Fortunately, adaptations of the lookahead concept can significantly reduce add time over the time achieved with a 100% ripple of the carry through a number without using 100% lookahead.

Providing 100% lookahead carry for even a modest-sized number is not practical because doing this requires a large amount of logic as well as large input gates, with the concurrent requirement for signals driving very large fanouts. Examining the boolean expressions for a variety of lookahead lengths reveals that realizing lookahead beyond a length of four numerals becomes unattractive, whether dealing with macrocells available in Motorola's MCA2500ECL macrocell library or available discrete SSI/MSI packages. (Of course, the designer determines what is "unattractive" as he trades speed for logic.) Faced with this subjective limitation the designer could elect to:

(1) Implement lookahead using generates (G(K)) and propagates (P(K)) derived directly from blocks of addend and augend bits (A(K) and B(K)), providing lookahead carries between these blocks of bits wherein the carry is rippled through.

(2) Employ multi-level lookahead wherein the generates and propagates that will form the lookahead carry are derived in turn from the generates and propagates derived directly from addend and augend bits, or even from other intermediate generate and propagates.

(3) Employ a combination of methods 1 and 2 tailored to the designer's immediate need.

It is left to the reader to perform the boolean algebraic manipulation to convince himself of the validity of such multi-level structures. A lookahead carry that is generated from generates and propagates that are derived directly from the addend and augend bits is usually called a "zero-level lookahead." Higher multi-level lookahead carries utilizing one, two,...., N levels of intermediate generates and propagates are called first, second,....,(N¹1)-th level lookahead schemes respectively.

A family of lookahead subunits has been developed to support Motorola's MCA2500ECL (MCA II) Macrocell Array semi-custom program. These lookahead subunits optimumly use those macros (macrocells) available in this macrocell library, delivering minimum propagation delays across bit spans of various lengths, while consuming minimum chip space.

Binary Addition/Subtraction

In representing signed binary numbers, three coding systems dominate: sign-and-magnitude, 1's-complement, and 2's-complement. Although the sign-and-magnitude representation is the easiest to understand, designing logic networks to do arithmetic with sign-and-magnitude binary numbers is very awkward. Performing arithmetic employing 2's complement representation, on the other hand, is straightforward.

In the 2's-complement system, positive numbers are represented by a zero followed by the number's magnitude (just as in the sign-and-magnitude system). Negative numbers are represented by the 2's complement of their positive counterpart, which, for binary numbers, is formed by inverting each bit and adding '1' to the least-significant-bit position. (This methodology is not the definition of how to form the 2's complement of a number for any base. However, the definition of 2's complement can be extended to any number system.) Forming the 2's complement of a number currently represented in 2's-complement notation is equivalent to changing the sign of that number. Of course, the result remains in 2's-complement notation. To add two numbers in 2's-complement notation does not require manipulation of either number prior to performing the addition. Subtracting one number from another, each in 2's-complement notation, is carried out by forming the 2's complement of the number to be subtracted (changing its sign) and then adding the two numbers. In either case, addition is carried out just as if each number was positive, but with any carry from the sign position ignored. This will always give the correct result except when an "overflow" condition occurs.

An overflow is said to occur if, when adding two numbers each with a length of 'N' bits, the correct representation of the sum, including the sign, requires more than 'N' bits. An overflow condition is easy to detect: it is indicated when the addition of two positive numbers would otherwise indicate a negative result (a '1' in the sign bit position) or the addition of two negative numbers would otherwise indicate a positive result (a '0' in the sign bit position). The proper representation of a sum, when an overflow occurs, uses the overflow bit as the sign bit, with the 'N' bits, previously sized to contain both the sign and magnitude segments of augend and addend, representing solely the sum's magnitude. There is no need to examine the carry from the sign bit position (N-th bit for 'N' bit addends and augends) because ignoring any carry always gives the correct answer. Addition of 1's-complement numbers is the same as addition using 2's complement numbers, except that instead of discarding the carry from the sign position, the carry is added to the sum at the least-significant-bit position ("end-around carry"). Because this involves more logic, we will focus on using the 2's-complement system for addition or subtraction.

Forming the 2's complement of a binary number is most efficiently done by exclusive-or gating to each bit input of the adder structure of choice, using one of the exclusive-or inputs as a control line to select bit inversion. An appropriate scheme must also be realized as input to the carry-in to the least-significant bit of the adder structure, providing for the carry-in of '1' when needed to generate the 2's complement of a number as well as providing the usual carry-in capability. Finally, provision must be made for generating the overflow bit.

Quick reflection on the logic realized by a full adder reveals that:

$$S(i) = A(i) \oplus B(i) \oplus C(IN) \quad (4)$$

$$C(OUT) = A(i) \bullet B(i) + A(i) \bullet C(IN) + B(i) \bullet C(IN) \quad (5)$$

Where: A(i) = addend bit at the i-th position in a number
B(i) = augend bit at the i-th position in a number
S(i) = sum bit at the i-th position in a number
C(IN) = carry-in to the i-th position in a number
C(OUT) = carry-out from the i-th position in a number

Equation 4 can be rewritten as:

$$S(i) = P(i) \oplus C(IN) \quad (6)$$

and, from the discussion of lookahead techniques, equation 5 can be expressed as:

$$C(OUT) = G(i) + P(i) \bullet C(IN) \quad (7)$$

In designing an adder, the most expedient way to employ a lookahead scheme is to focus on a realization that provides the generates and propagates for each sum-bit rather than first realizing the sum bit directly. Fortunately, the M283 2-bit Lookahead Carry macro does this, supplying the individual propagates for each of two bits (labeled 'H(0)' and 'H(1)' in the design manual's library rather than 'P(0)' and 'P(1)', the carry-out 'C(0)' for one pair of bits to be added, as well as the (group) propagate and (group) generate for a two-bit span.

This important building block (M283) is extremely fast and extremely space-efficient. By itself, this macro can form the core of a two-bit-adder structure significantly faster and smaller in real estate than any other MCA2500ECL structure, including those macros designed to provide that specific function—while producing all the terms needed for lookahead carry generation.

Figure 1 shows this basic unit adapted for use in a binary adder/subtracter of arbitary size.

Figure 2 shows a higher-level building block built from this basic unit for use in the realization of larger binary adders/subtractors as well as the amendment necessary to provide a carry-in of '1' when the 2's complement of a number is formed, in addition to the usual provision of carry-in capability.

Figure 3 shows an 8-bit-wide block ($SUBU BIN8TERM) to be used as the terminal segment of a larger binary adder/subtractor structure, as evidenced by the propagation of the carry with the aid of lookahead logic rather than a straight ripple between M283 blocks. This particular 8-bit-wide block was developed because, in an optimally designed adder/subtractor, the longest delay path should terminate in the most significant bit (assuming that the same logic is required at each bit position to produce the sum bit, as is the case here). Anticipating that 8 bits would be the minimum binary length of interest to the reader, 8-bit binary adder/subtractor blocks were used instead of individual bits or M283 2-bit units in conjunction with

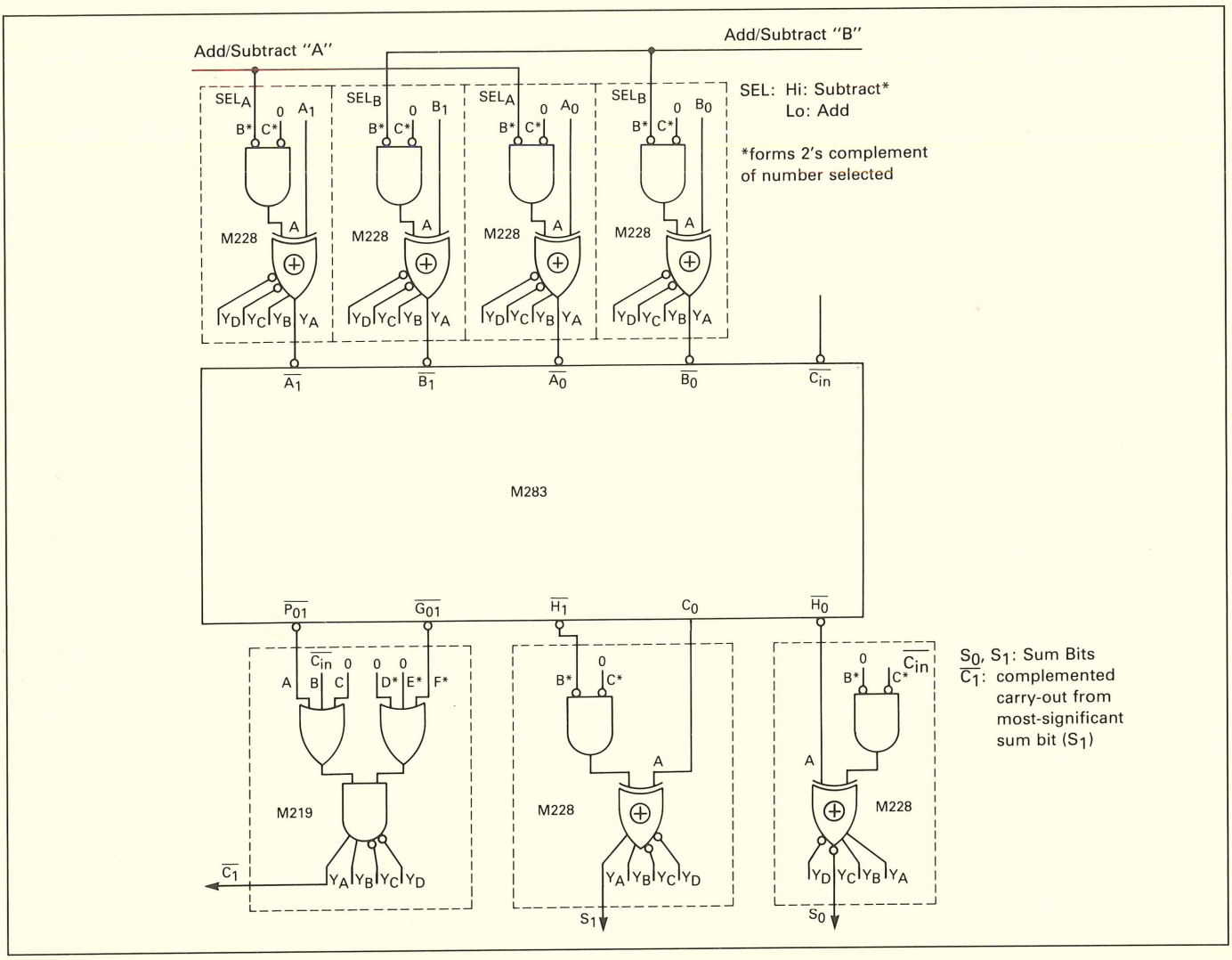


Figure 1. Binary Adder/Subtractor Basic Unit.

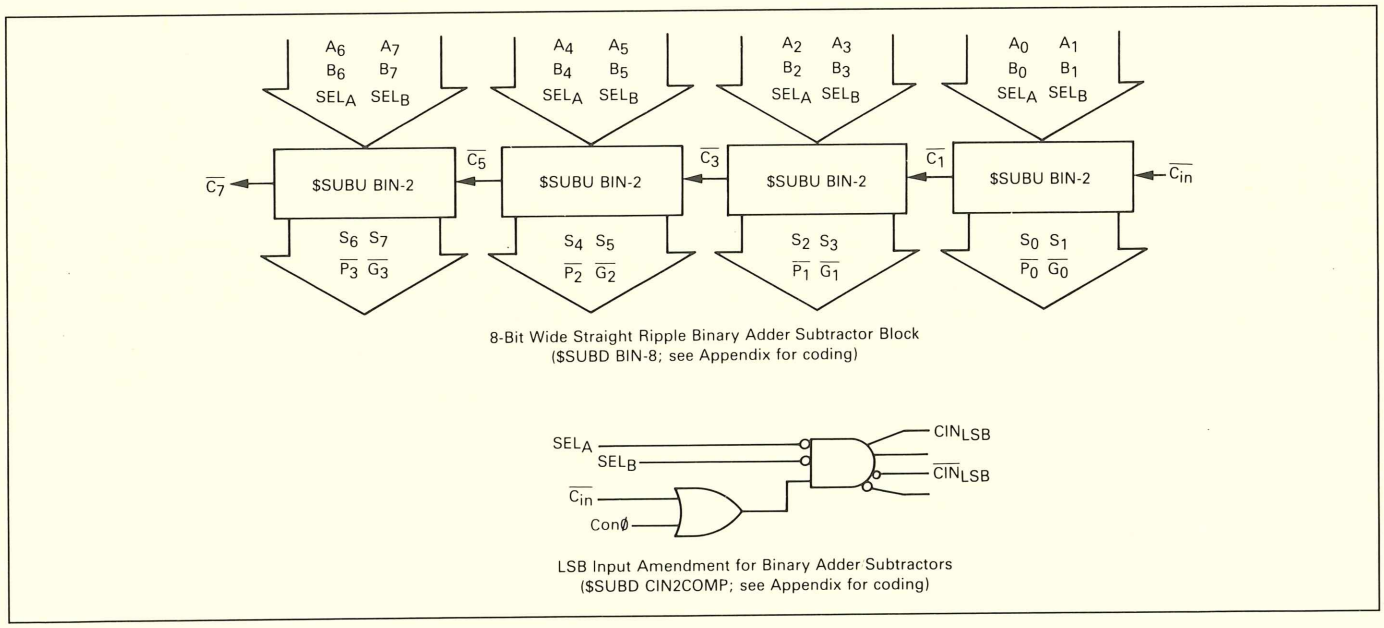


Figure 2. Intermediate-level building blocks for binary adder/subtractor realizations.

various lookahead units. These blocks are used to construct binary adder/subtractors of varying sizes, with ripple through the 8-bit-wide slice propagating the carry-in to that slice's bits. Because a significant amount of the overall longest-path propagation delay was utilized in the ripple through the terminal adder/subtractor block, speed would be significantly increased by employing lookahead logic within this segment as well.

The proper use of the binary adder/subtractor constructs presented relies on the following constraints:

- Both augend and addend are supplied to the unit in 2's complement representation.
- At most, only one of the two numbers (addend or augend) will have its sign changed (2's complement formed).
- An additional carry-in of 1 to the least-significant bit will not be provided when a 2's complement of either the addend or augend is being performed.

Binary/BCD Addition

Although representing decimal digits in binary code is not an efficient way to communicate information (i.e., information density), there are cases where other (display/interface) considerations must be met. One could achieve BCD addition by first performing binary addition on the two strings of BCD numbers, then adding six (0110) to each 4-bit BCD string that exhibits a carry-out of 1 from the most-significant bit.

In such a scheme two additions are being performed rather than one. This can be avoided by developing the appropriate expressions for the propagate and generate terms for each BCD digit, then employing the same lookahead techniques used for the binary adder/subtractor. Such a straightforward approach will increase the amount of logic and propagation delay time, but not as much as necessitated by dual addition. Figures 4 and 5 show the logic for each binary/BCD unit.

Again, the M283 2-bit Lookahead-Carry is used as the core construct of the binary/BCD adder because it can produce intermediate propagate and generate terms.

Binary Adder/Subtractor and Binary/BCD Adder Implementations

Figures 3 and 6 show realizations of binary adder/subtractors of various bit lengths. Figure 7 shows an 8 bit adder. Table 1 lists specifications for each, both in longest propagation delay as well as size. Because it is probable that the user of Motorola's MCA2500ECL Macrocell Array will employ additional logic when using these adders, the propagation delay times represent an elapsed time with t^50 being the arrival of all of the addend and augend bits at the first gates in the network, and the final elapsed time the arrival of the last sum bit from the final network gate with 1) fanout of zero in the case of the binary adder/subtractors, and 2) fanout of four in the case of the binary/BCD adders. Further, delay times use nominal metal lengths between various building blocks (macros), as supplied by the CAD system, rather than values after a place and route of the final network on the arrived at MCA2500ECL array.

All realizations were built solely with (internal) M cells, but user needs may dictate otherwise. Because the specifications assume the simultaneous application of all addend and augend bits, any select lines should be developed with sufficient fanout and delivered with appropriate timeliness as not to be a rate limiting path.

The binary adder/subtractor implementations all employ lookahead between units of 8 bits. The carry is rippled across each 8-bit-wide block except for that block comprising the most-significant-eight bits; that block employs internal lookahead to maximize performance. These adder/subtractor designs were implemented with lookahead schemes chosen to cause the longest propagation delay to terminate in the most

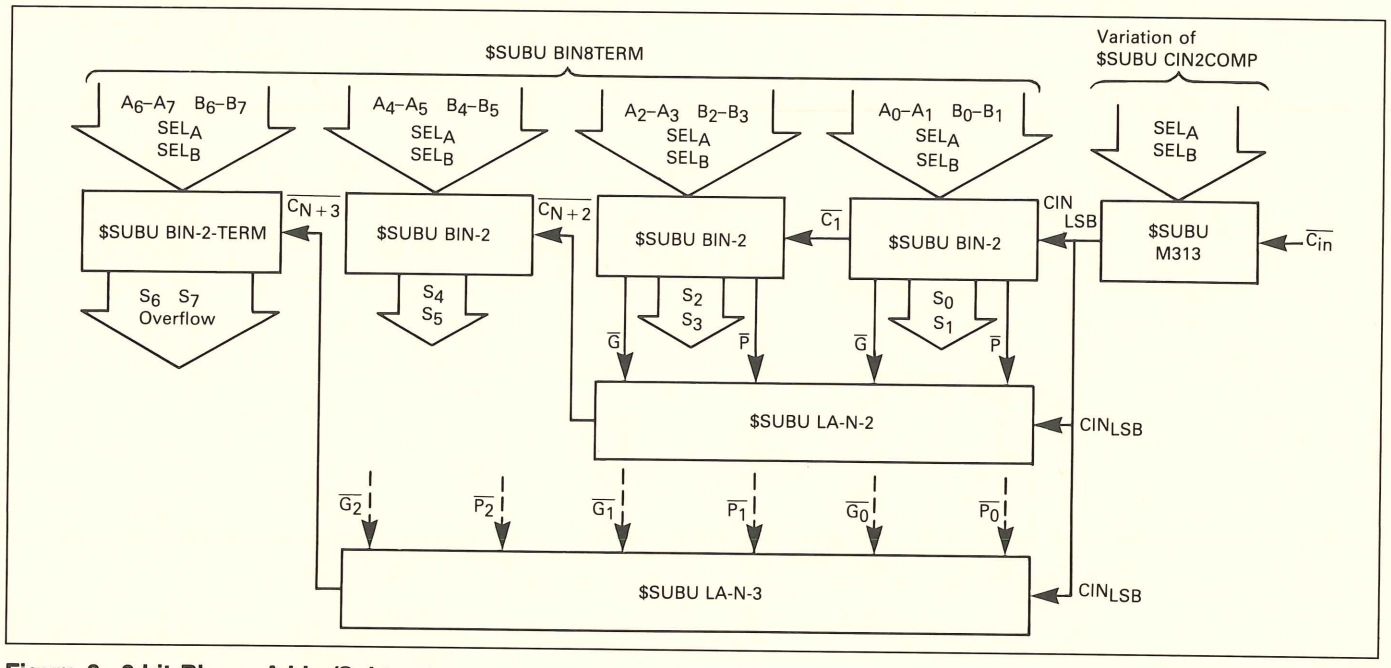


Figure 3. 8-bit Binary Adder/Subtractor.

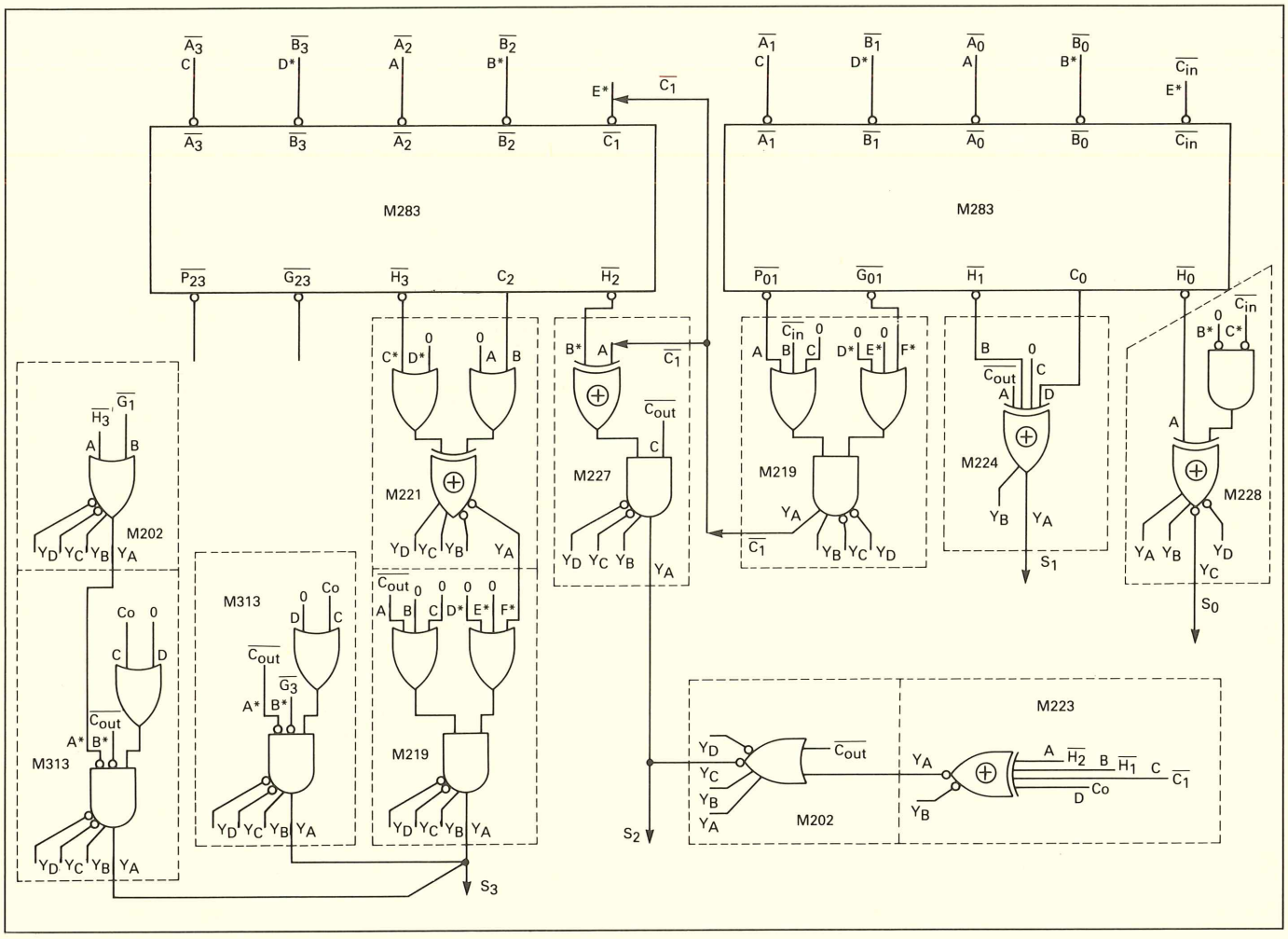


Figure 4. Binary/BCD Adder Basic Unit.

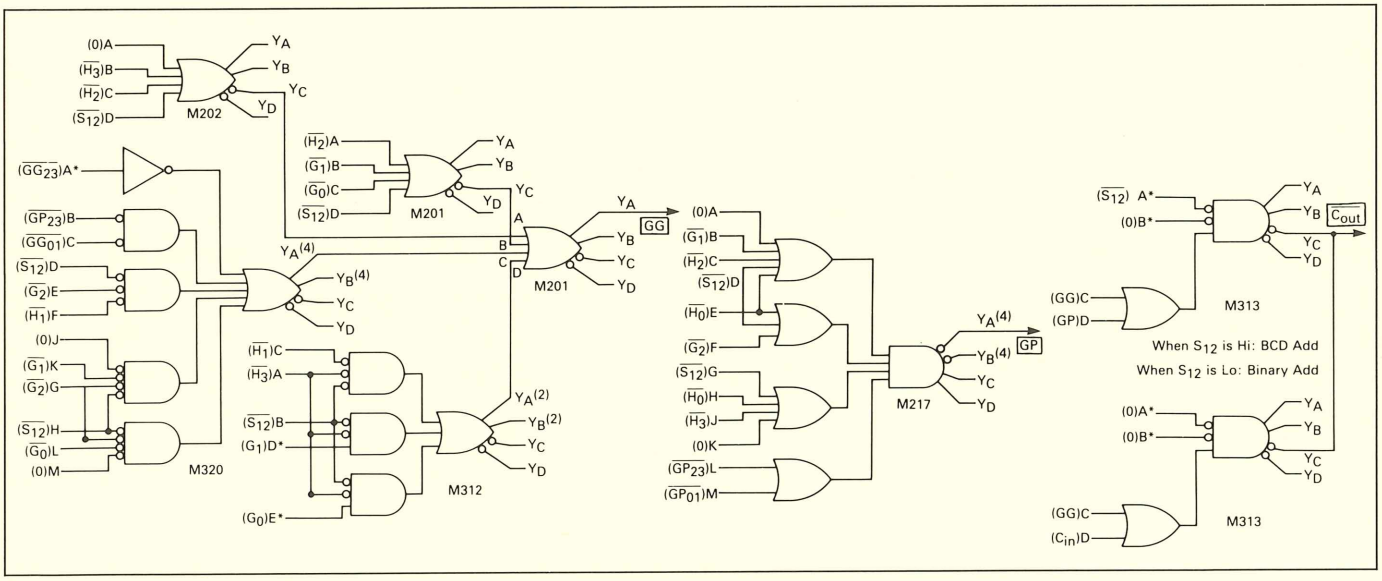


Figure 5. Realization of $\overline{C_{out}}$ for use with Binary Adder/Subtractor Basic Unit.

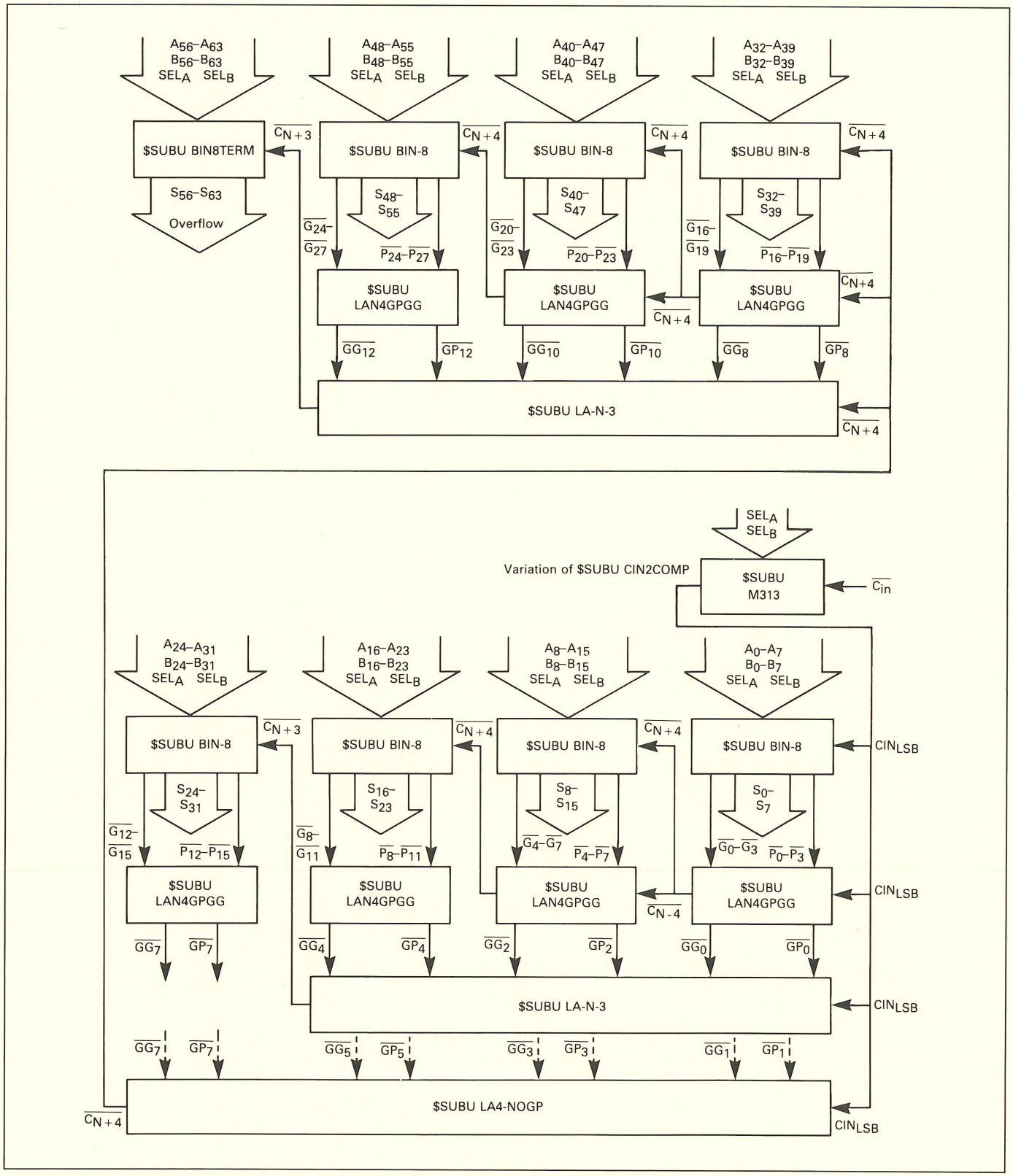


Figure 6. 64-bit Binary Adder/Subtractor.

Number of Bits	Binary/BCD Adder Max P_D (ns)	Binary/BCD Adder Size (M cells)	Binary Adder/Subtractor Max P_D (ns)	Binary Adder/Subtractor Size (M cells)	Comments re Binary Adder/Subtractor
8	6.950**	21	4.500**	13.50	Selects have fanout of 5.
16	8.375	43	5.975	25.75	Longest delay path is ripple through first block of 8 (Δ = 0.225 ns); Selects have fanout of 8.
24	8.625	64.25	7.475	38	Longest delay path is ripple through second block of 8 (Δ = 0.275 ns); Selects have fanout of 8.
32	9.900	86	8.400†	51.25	Longest delay path is ripple through third block of 8 (Δ = 1.800 ns); Selects have fanout of 8.
64	Exceeds Macrocell Array Size Limitations		10.275†	102.50	Longest delay path is ripple through seventh block of 8 (Δ = 2.200 ns); Selects have fanout of 8.

** 100% Fault graded (see text for additional detail).
† Potential for increased speed performance (see text).

Table 1. Binary/BCD Adder and Binary Adder/Subtractor Realizations Utilizing MCA2500ECL "M" Cells.

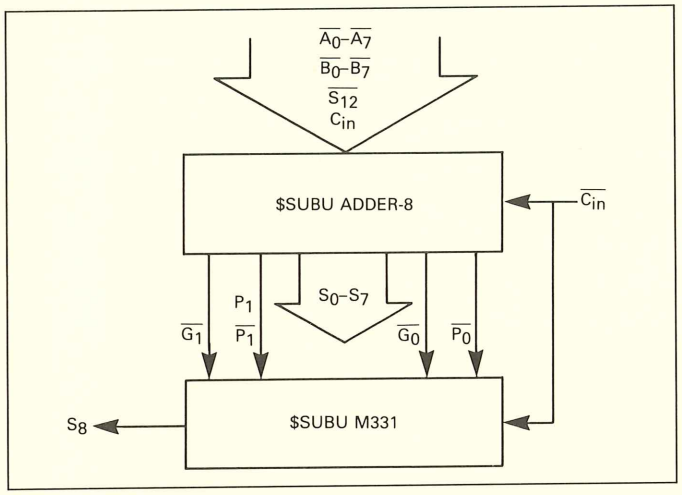


Figure 7. 8-bit Binary/BCD Adder.

significant bit. However, analyses of the longest propagation delay in each design revealed that, except in the eight-bit adder/subtractor, the longest delay terminated instead through the next-to-least-significant block; this delta is due to the long ripple of the carry across eight bits. This delta (indicated in table 1), seems to be significant only in large structures. Additional delay reductions could be realized in these cases by adding lookahead for that block if the performance gain would justify the additional logic.

The binary/BCD adder implementations are somewhat more straightforward, with lookahead logic needed to support (at most) eight binary/BCD digits (each 4 binary-bits wide). Figure 8 details an intermediate building block called "ADDER-8", which comprises two BCD digits with lookahead transporting the carry from one to another. This block is utilized for all the structures, easing the design task.

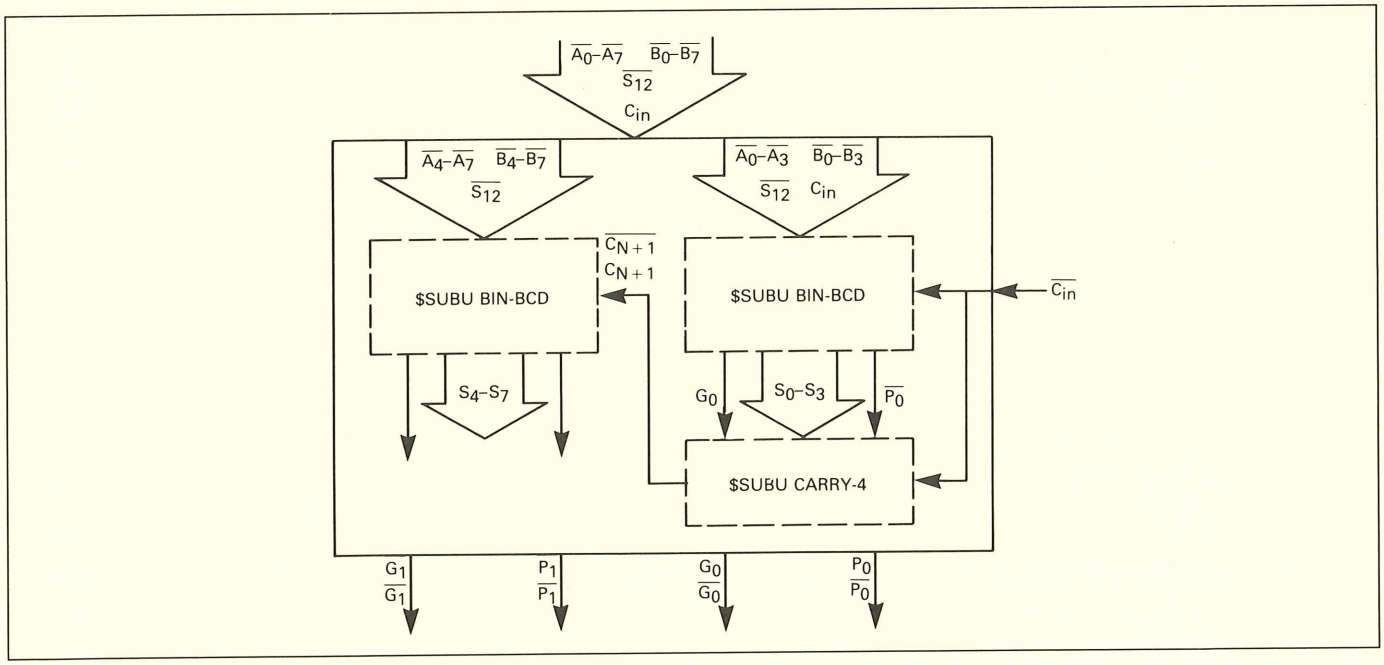


Figure 8. $SUBD ADDER-8 detail. (See figure 7.)

Summary

Using Motorola's MCA2500ECL Macrocell library, binary/BCD adders and binary adder/subtractors employing lookahead carry have been implemented in popular bit widths. To optimize for speed and part count, a family of lookahead units spanning varying bit/BCD-numeral lengths has been developed. Adder/subtractor structures were built using stepwise-refinement with intermediate-level building blocks developed to perform intermediate levels of logic complexity. Each lookahead subunit as well as the binary/BCD and binary adder/subtractor cells have been 100% fault graded for accuracy. However, some adder/subtractor implementations may be limited by the size or pinouts available in the chosen array.

Using multi-level lookahead for carry transport can produce significant speed advantages over single-level lookahead. In multi-level lookahead, the number of bits spanned by the lookahead construct equals the product of the bits spanned by lookaheads at each level (when all lookaheads at a given level have equal lengths). This proves quite useful when spanning a large number of bits. For example, the 64-bit binary adder/subtractor, shown in figure 6, employs three levels of lookahead; per 32-bit slice, with lookahead spans of 2, 4, and 4—realizing a final lookahead length of 2×4×4 or 32 bits.

The lookahead carry subunits were developed with two primary constraints: 1) what propagates/generates were available (true or complemented form), and 2), if both forms were available, what consideration should be given to their time-of-arrival differences. For example, in the binary adder/subtractors only the complemented terms are available (from M283). Implementing the binary/BCD adders allowed all four possible terms to be available, but the true forms were noticeably faster. Another constraint was the desire not to supply a logical '1' to any inputs (this would require more logic).

Other concerns: Should the lookaheads provide propagates/generates as well as carries? Is either the true or complemented form of the carry needed? What is the initial versus succeeding time delay to conduct the carry through the lookahead structure(for multilevel lookahead)? These lookahead structures were developed to satisfy specific criteria, criteria not necessarily the same for all design situations. This caveat and figure 9 should help the designer decide the appropriate choice of lookahead subunits for implementations other than presented in table 1. Figure 9 shows that a constant "bit-transport" rate is not achieved irrespective of what lookahead structure is chosen. This fact is evident in "logic overhead" embodied in the lookaheads spanning shorter

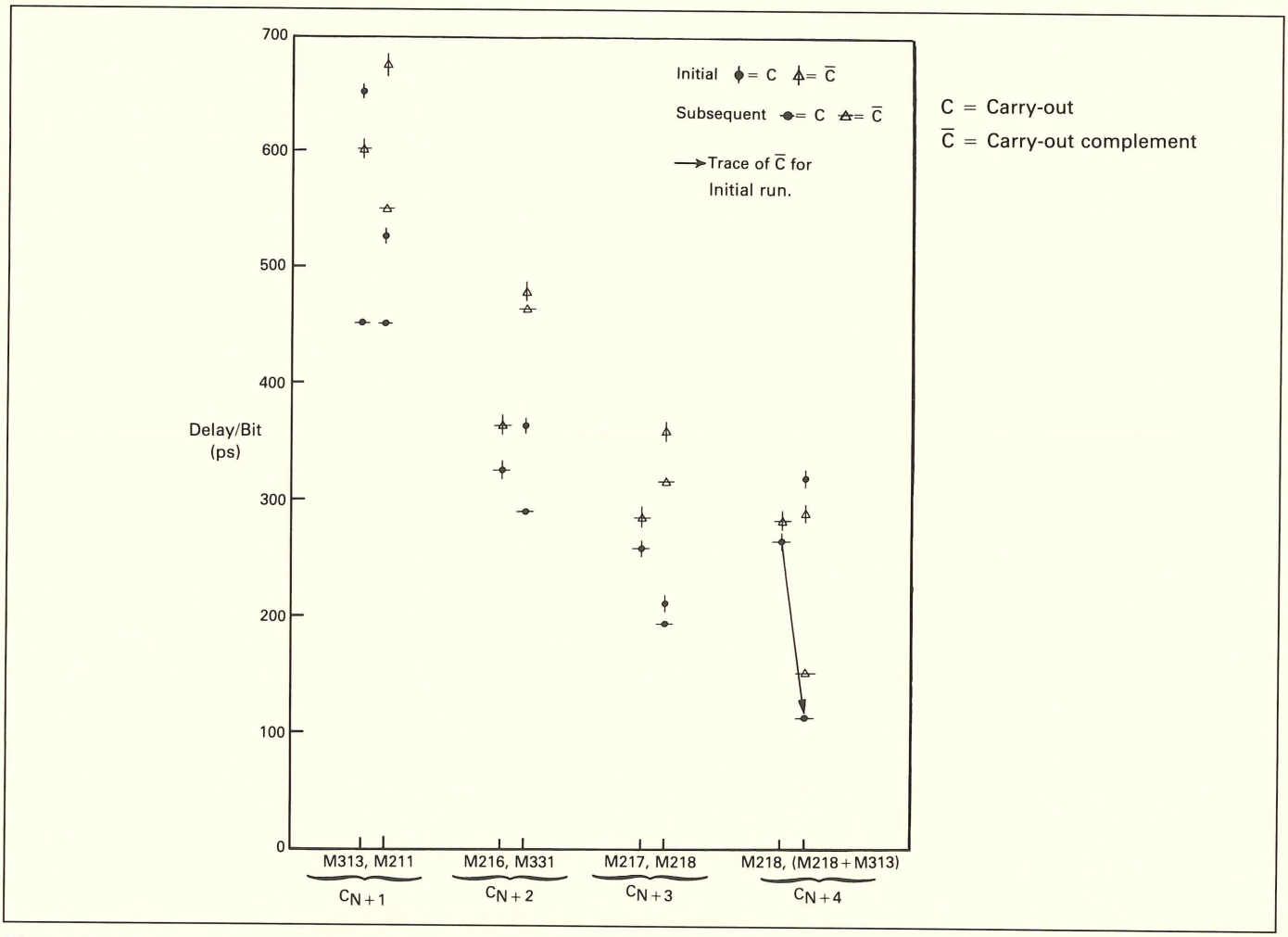


Figure 9. Lookahead carry propagation delays for various bit spans.

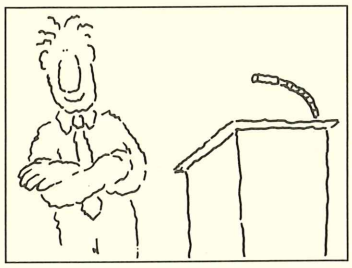
lengths. Because there are but a few adder/subtractor structures of interest, and lookahead performance is nonlinear, it is best to use an empirical approach when working up a lookahead implementation. This last sentence gives the bottom line: No formula will automatically dictate what form of lookahead should be used for an adder/subtractor of radix 'x', 'y' bits long. One must look at the lookahead units available (see figure 9) then choose the units and scheme appropriate for the problem.

For More Information

An application note, AN953, shows these and additional lookahead implementations, along with the code needed to build any of adder/subtractors. For this application note or more information about Macrocell Arrays, call Katie Hamilton, sales engineer, Motorola, 503-641-3681.

For more information about the Connections program, call Wes Brunning 629-1488 (92-824). ☐

Been Asked to Organize a Session or to Talk at a Conference?



Professional conferences are an integral part of the information-transfer process. The success of such conferences depends on many things. The theme, the facilities, even the weather are important, but the speakers and the organizers are critical.

To help session organizers and speakers, Technology Communications Support has prepared two guides, each based on extensive experience in preparing and supporting professional communications.

If you've been invited to talk, *So You're Giving a Talk at a Professional Conference* will help you organize your talk and prepare effective slides to go with it. The rules in this booklet are few and simple, but they can help you look like a "pro" instead of an ill-prepared amateur.

If you've been invited to organize a session, you'll be expected to be not only the manager of a team but the coach, scheduler, and expediter as well. So *You're Organizing a Session for a Professional Conference* gives the basic rules and guidelines.

To get a copy of either or both guides, and to get professional support in preparing talks and slides, contact Technology Communications Support d.s. 53-077 (642-8920). ☐

Technology Report
MAILING LIST COUPON

☐ ADD
☐ REMOVE

Not available to field offices or outside the U.S.

MAIL COUPON TO 53-077

Name:_____ D.S.: _____
Payroll Code: _____
(Required for the mailing list)

For change of delivery station, use a directory change form.

Standards Review Board

Organization	Appointing Manager	SRB Member/Alternate	Location	Phone
Instrument Systems	Frank Hermance, LI Jim Koehn, ISI Sal Kadri, Accessories	Joel Swanno/John Hazard Jim Brammer Rick Wilson/Bill Dippert	39-113/39-113 C1-469 C1-708/C1-775	627-3055/627-3053 253-5770 253-5415/253-5394
Portable Inst. & Mfg.	Fred Hanson, VP&GM Dick Knight Soren Vestergaard George Kersels, VP EMCG	Merle Nielsen/Pam Weightman Jeff Allen/Rick Cummings Jim Herinckx/Paul VanDomelen Robin March	47-670/47-664 F1-186/F1-186 16-285/16-285 19-290	627-2981/627-3008 640-2288 x4248 627-7814/627-0763 627-5070
Technology	George Kersels, VP	Maria Lochmann	50-252	627-1242
Communications	Morris Engelson, FDI Larry Kaplan, TV	Fran DiGiorgio Ken Durk/Steve Kvavle Bob Melton	R1-000 58-620/58-585 58-594	923-4442 627-1398/627-1254 627-1345
Information Display	Geo. Rhine, Actg GWD Roy Barker, PD Jerry Ramey, TD Geo. Rhine, Actg GWD	Ron Murrey (East) Marlene Conklin Bill DeVey/Lee Winer Mike Massey (Central)	63-281 63-223 63-083/83-416 60-559	685-3230 685-3511 685-3520/685-3048 685-2478
Design Automation	Tom Clark, SDP Dick Lemke, LA Vince Lutheran, STS	Brent Anderson Larry Larson Haydn Piper/Alice Houtsager	92-551 92-703 94-442/94-442	629-1677 629-1883 629-1240/629-1125
Corp. Services & U.S. Field Support	Jim Baker, Mgr. Factory Service	Del Knapp	56-125	642-8658
Tek's Standards Review Board reviews all proposed group and company-wide standards. The Board also reviews revisions to existing Tek standards.				

COMPANY CONFIDENTIAL
NOT AVAILABLE TO FIELD OFFICES

TECHNOLOGY REPORT
RICHARD E CORNWELL
mail label goes here
19-285

DO NOT FORWARD

Tektronix, Inc. is an equal opportunity employer.