Configurable Array Logic Circuits for Computing Network Error Detection Codes

Gordon Brebner

Department of Computer Science
University of Edinburgh
James Clerk Maxwell Building
Mayfield Road
Edinburgh EH9 3JZ
Britain

Abstract

Configurable Array Logic (CAL) has a basic architecture which is a cellular array with nearest neighbour connections. The cells in the array are dynamically programmable using transistor switches controlled by static RAM cells. Each cell can realise any two-input Boolean operation or act as a simple latch, as well as providing routing for pass-through connections to allow non-neighbour inter-cell connections. In this paper, we demonstrate the versatility of the CAL technology by presenting efficient CAL circuits for computing all of the major error detection codes now in use for world-wide computer networking; these include CCITT, IEEE, Internet and ISO standard codes. The circuits, each having a version which comfortably fits on to a single 32 x 32 cell CAL chip, are appropriate for use as hardware accelerators to help computers to deal with the ever-increasing rates of data transmission over networks. The first class of error detection codes described are the *cyclic redundancy codes* (CRCs), which are in virtually universal use for bit serial transmission over physical links. The other class of error detection codes described are the 'modulo $2^n - 1$' checksums, which are in common use for byte transmission over networks and inter-networks.

1. Introduction

The Bit Stream Enhancement (BSE) project in the Department of Computer Science at the University of Edinburgh is investigating the application of Configurable Array Logic (CAL) technology to the evaluation of various compute-intensive functions used by computer networking protocols. As the name of the project suggests, the functions under consideration are used to improve the quality of communication of bit sequences over less than ideal transmission media; the functions can be categorised as:

- compression functions, which reduce the quantity of transmitted data used to represent information, so that higher throughput is possible;
- encryption functions, which scramble transmitted information, so that it is intelligible only to the intended recipient;
- error detection functions, which add codes to transmitted data, so that transmission errors can be detected by the recipient.

Compression functions will be considered in a later paper, the aim being to design logic circuits to implement a variety of compression functions, appropriate for different types of information distribution, and then to embed these in a configurable system which can adjust dynamically to changing information distributions during communication. Encryption functions will also be covered later: a CAL circuit design to implement the U.S. Data Encryption Standard (DES) can be found in [1]. Error detection functions are examined in this paper. Given that choices of error detection function are normally fixed over prolonged time periods, it is not likely that dynamic circuit reconfiguration during communication is a useful property here except, perhaps, for specialist devices such as protocol analysers. Instead, configurable logic can be viewed just as a simple way of building hardware accelerators to tackle communication-related functions; in the longer term, such accelerators can be easily reconfigured if function requirements change.

Two classes of error detection code are considered here. The first class of codes are prevalent amongst communication protocols which can be classified within the Data Link layer of the ISO reference model for Open Systems Interconnection (RM/OSI), that is, for communication over physical channels within local area or wide area computer networks. These are the *cyclic redundancy codes*, and they are typically computed by hardware. The second class of codes are prevalent amongst communication protocols which can be classified within the Inter-network sub-layer, or the Transport layer, of RM/OSI, that is, for communication over logical channels across one or more computer networks. These are the *modulo* $2^n - 1$ *checksums*, and they are typically computed by software. Particular examples include both the error detection code used by the ISO protocols (the official international standards), and the error detection code used by the Internet protocols (the main alternative, *de facto,* international standards).

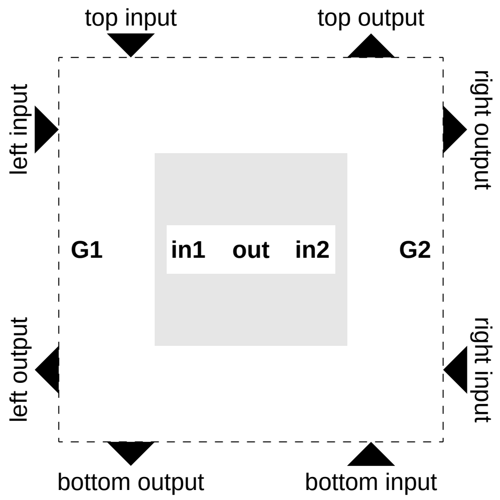

Fig. 1. Schematic diagram of a CAL cell.

2. Configurable Array Logic

Configurable Array Logic (CAL) [1,2] has a basic architecture which is a programmable cellular array with nearest neighbour connections. Cells and connections are very simple, supporting only single-bit operations and transfers. Figure 1 shows a schematic diagram of the components of a single cell. There is a function unit, shown in grey, together with input and output ports around the perimeter; also, G_1 and G_2 are two global signals which are distributed to all cells in the array. The function unit can be programmed either to perform one of the 16 possible two-input Boolean operations, or to act as a clocked one-bit latch; if it is not programmed, the cell only through-routes connections. More precisely, the possible functions are:

- constant value: zero, one (ignoring both inputs);
- unary operator: identity, **not** (of either input);
- binary operator: **and**, **or**, **exclusive-or** (with optional negation of inputs);
- C-D latch (with optional negation of clock and/or data inputs).

Within-cell routing can be programmed in order to inter-connect the function unit inputs and output, and the input and output ports. Possible sources for connections are:

- the input ports at the top, bottom, left and right of the cell;
- the function unit output;
- the global signals G_1 and G_2

and possible destinations for connections are:

- the output ports at the top, bottom, left and right of the cell;
- the function unit inputs.

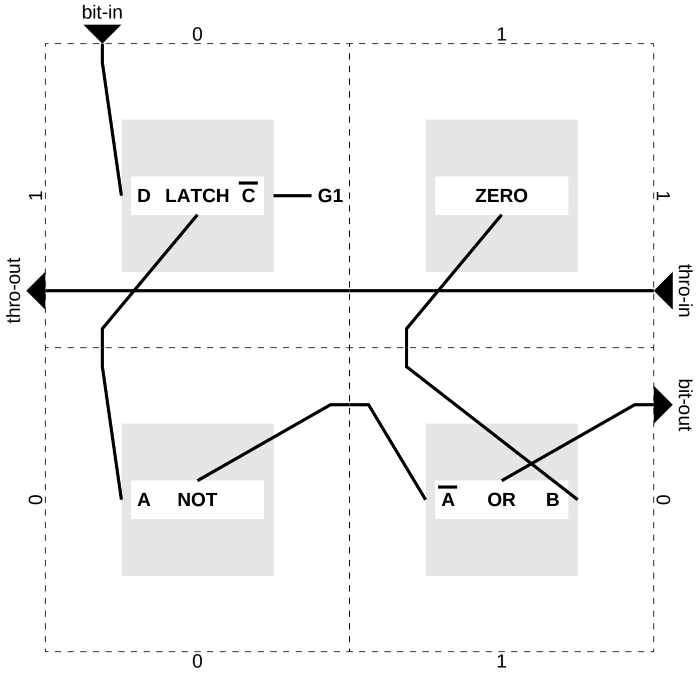

Fig. 2. Example of programmed CAL array.

Some routing possibilities are not allowed: function unit output to function unit input, and global signals to the in_2 function unit input. One source can be routed to more than one destination, but more than one source cannot be routed to the same destination. A (non-meaningful) example array containing four programmed cells is shown in Figure 2; this array has two external input ports "bit-in" and "thro-in", and two external output ports "bit-out" and "thro-out'. To see further (meaningful) examples, the reader should glance ahead to the figures in Sections 3 and 4.

A graphical design tool is available to aid the production of CAL circuits. However, for the work described here, a library of C procedures was constructed, which allows the assignment of a CAL array to be specified using the full power of the C programming language, rather than the more modest constructions of the graphical tool. The two outputs from the library procedures are (a) a file containing a description of the CAL array in the 'cfg' format which is used as input to the supplied tool for programming CAL arrays, and (b) a file containing a pictorial representation of the CAL array in the Postscript language. The latter capability was used to generate all of the CAL-related figures in this document. To give the merest whiff of the usage of this library, the C procedure calls to specify the details of the bottom right cell in Figure 2 were:

```
block_port ("bit-out", out_port, 1, 0, right);   /* Port at (1,0) */

begin_cell (not_a_or_b, 1, 0);                   /* Cell at (1,0) */
   from_to (left, a_in);
   from_to (top, b_in);
   from_to (f_out, right);
end_cell ();
```

(of course, this illustrative example makes no use of the ability to include C variables and control structures).

The arguments for and against graphical design tools and programming language-embedded design tools have been well-rehearsed (in the case of CAL, see [1]). For this work, the primary reason for choosing the latter approach was that *families* of CAL circuits were being designed, with the intention that a final system would allow the dynamic selection and configuration of particular family members under program control. This stems from the nature of configurable hardware, which allows the circuit designer to avoid focussing on one particular family member, and instead to produce genuinely parameterised designs. Thus, it is desirable to make the full power of a high-level programming language available to express the structure of circuits.

The Algotronix CAL1024 chip is a 1.5 micron CMOS realisation of the CAL concept and, as seen by its user, it contains a 32 x 32 array of cells with the above capabilities. In fact, the actual on-chip realisation is a little different, but a tool is supplied to perform the necessary adjustments automatically prior to cell programming. All cell input and output ports on the array boundary are accessible from chip pins via programmable input-output blocks on the chip periphery. The Algotronix CHS2x4 Custom Computer is a board-level system compatible with the PC/AT bus, and contains up to nine CAL1024 chips, together with up to two megabytes of static RAM. One of the CAL1024 chips interfaces the board to the host computer; the others are available for general-purpose use. A CHS2x4 board was used for the work described here, and it contained seven CAL1024 chips; however, all of the CAL circuits here fit on to a single chip. Some of the features of the CHS2x4 system have influenced the details of the CAL circuit design, but not in such a way as to make the designs specific to the CHS2x4 system.

The CHS2x4 board has external connectors allowing direct access to the CAL array but, for testing the circuits described here, all contact between testing programs running on the host computer and the CAL array took place via the CHS2x4 on-board memory. The overall sequence of operations for running a CAL circuit was:

(1) program the CAL array cells using data from host computer memory;

(2) copy a sequence of circuit inputs from host computer memory to on-board memory;

(3) activate the CAL array in a sequence of steps, with one new circuit input being read from on-board memory at each step, and one new circuit output being written to on-board memory at each step; and

(4) copy the sequence of circuit outputs from on-board memory to host computer memory for checking against correct values computed by software.

This style of working reflects the capabilities of the PC/AT interface mechanism supplied for the CHS2x4 board, and allows the behaviour of the CAL array to be checked, albeit at a slow operating rate determined by the speed of interaction between the PC/AT and the array.

The interaction between the CAL array and the on-board memory is controlled by the control CAL chip. In the mode used here, an eight-bit value is read from the memory and supplied to fixed input ports on eight fixed CAL cells and, at the same time, an eight-bit value is supplied by eight fixed output ports on eight fixed CAL cells to be written to the memory; during this time (approx. 110 ns), the global signal G_1 is asserted. Afterwards, the read and write memory addresses are both auto-incremented by the control CAL, in readiness for the next operational step.

The impact of this discipline on the circuits designed here was that the global signal G_1 is used only to control latches. An operational step of any circuit consists of input and output when G_1 is high, followed by computation, as appropriate, when G_1 is low. Input latches are clocked by G_1 and storage/output latches are clocked by $\overline{G_1}$. A further convention adopted, not related to the test system, is that the global signal G_2 is used only during the computation ($\overline{G_1}$) phase, and that it has a initialising rôle in the sense that it always forces the supply of a zero operand instead of a stored operand to one or more function units. The state of G_2 can be controlled from the testing program. A separate input signal could have been used for this purpose (and the circuit layouts all allow for the wiring needed for such a signal) but, in the absence of any other use for G_2 in the circuits here, it was convenient to harness it in this way.

In order to analyse the maximum operating rates of the circuits, given that it was not possible to exercise them to the full using the available test system, a simple model of circuit behaviour was used. In this, an operational delay for each cell can be determined, and then the operating time for the circuit can be determined by summing the cell delays along critical paths. Values for CAL1024 cell delays under normal operating conditions were obtained after discussions with the chip designer; these were suggested by simulation results, but checked by measurements on particular chips [3]. They are over-estimates of the real observed delays, a fact which should help to reassure any reader who, in the absence of real test results for the overall circuits, is concerned that the timing analyses are unduly optimistic. Particular values of interest here are:

- maximum time before new G_1 or G_2 values propagate to all cells = 10 ns;
- delay through a connecting cell = 2 ns;
- delay between input to a Boolean function cell and output = 8 ns;
- delay between input to a latch cell and output = 8 ns.

For the global signal conventions noted above, one other useful fact is that the maximum delay before an input latch cell can supply its output is 18 ns: 10 ns for the propagation of G_1 being high, followed by a 8 ns latch cell delay. It is also the case that propagation delays for G_2 during the computation phase of an operational step can be ignored, since new G_2 signal values can be allowed to propagate during the input phase.

3. Cyclic Redundancy Codes

Cyclic Redundancy Codes (CRCs) are in widespread use for performing error detection on bit-serial channels. These codes are usually used at a low level of communication, being computed by hardware as part of the process of transmission over a noisy channel. The operation of CRCs can be understood in terms of polynomials over the field $GF(2)$, i.e., in an algebraic system in which 0 and 1 are the only values, and in which addition and subtraction involve no carries and are both the same as the logical exclusive-or operation. If a message of length b bits: $m_{b-1}, \ldots, m_1, m_0$ (from most to least significant) is to be transmitted along a channel then, for coding purposes, it is considered as representing a $GF(2)$ polynomial:

$$M(x) = m_{b-1}x^{b-1} + \ldots + m_1 x + m_0.$$

The CRC used on any particular channel has a fixed $GF(2)$ *generator polynomial* $G(x)$, of degree k say, associated with it. The encoding process consists of dividing the polynomial $x^k M(x)$ by the generator polynomial $G(x)$ to find the remainder polynomial $R(x)$ (the quotient polynomial is ignored), and taking the coded form of the message as being the bits which are the coefficients of the polynomial $x^k M(x) - R(x)$.

Thus, the coded message consists of the original b message bits, followed by k additional bits corresponding to $R(x)$. Viewed as a polynomial, the message has the property that it is divisible by $G(x)$, and therefore a recipient can check for transmission errors by dividing any message it receives by $G(x)$ to check that there is no remainder; if there is, a channel error must have occurred. The generator polynomial is carefully chosen so that almost all errors will cause this division test to fail; for example, one simple fact is that using a generator of degree k allows the detection of all burst errors affecting up to k consecutive bits. Two well-known generators are the one chosen by the CCITT and used extensively on wide area networks:

$$x^{16} + x^{12} + x^5 + 1,$$

and the one chosen by the IEEE and used extensively on local area networks:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1.$$

At first sight, the central operation involving the division of polynomials may seem complicated, but in the primitive world of $GF(2)$ polynomials, the necessary long division can be implemented in hardware simply.

The details of long division of $GF(2)$ polynomials can be found in [4], but the general method is the same as normal long division, except that the intermediate division and subtraction operations are very simple. A well-known suitable piece of hardware [4] just consists of a k-stage shift register with, for $0 \leq i < k$, an **exclusive-or** gate inserted prior to the i-th stage wherever the generator $G(x)$ has a non-zero coefficient for x^i. One input

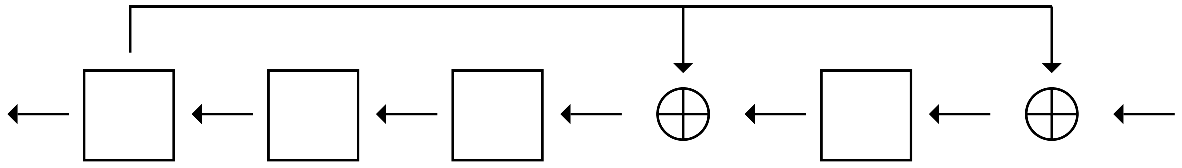

Fig. 3. Feedback shift register for simple generator.

of each such **exclusive-or** gate comes from the previous register stage (or from the circuit input in the case of the first stage), and the other input comes from the output of the final stage, causing feedback. In fact, the fed-back bit represents the result of an intermediate division, and the **exclusive-or** gates implement an intermediate subtraction, at each step of the long division. For example, a feedback shift register corresponding to the very simple generator $G(x) = x^4 + x + 1$ is shown in Figure 3.

To perform the required coding operation, the shift register is first initialised to contain zeroes, and then the message bits $m_{b-1}, \ldots, m_0$ are input in turn, followed by k zero bits, at which point the register contains the required k-bit remainder, which can be appended to the message bits for transmission.

We have produced a tool which inputs a description of a generator polynomial $G(x)$, in the form of a set $\{i : G(x)$ has a non-zero coefficient for $x^i\}$ of integers, and then outputs a CAL circuit which implements an appropriate feedback shift register in the above style. Each shift register stage is implemented by a pair of latch cells clocked by $\overline{G_1}$ and G_1 respectively. Where appropriate, **exclusive-or** function cells are inserted before register stages. An input latch cell clocked by G_1 is inserted before the first register stage. The $\overline{G_1}$ latch cell of the final register stage also acts as an output latch. An **and** function cell is inserted after the final register stage so that feedback may be inhibited by the assertion of G_2; this is desirable when initialising the shift register or when reading out the result from the shift register. The circuit produced for the simple generator $G(x) = x^4 + x + 1$ (from an input file containing "4 1 0") is shown in Figure 4.

Note that the shift register is implemented not just as a single row of cells, with data bits passing from right to left, and fed-back bits returning from left to right, but instead is folded about its mid-point to produce a double-height array of cells. Folding, which in this simple case involved only one fold, is a general feature of the CAL circuits produced. There are two main reasons for folding: (a) a unit-height array is liable to be excessively wide (e.g., 37 cells for the CCITT generator and 80 cells for the IEEE generator); and (b) given that all other wiring is very local, the propagation delay along feedback connections determines the rate at which the shift register can be operated and, because all real-life generators include a "+1" term, a unit-height array would have a delay proportional to the

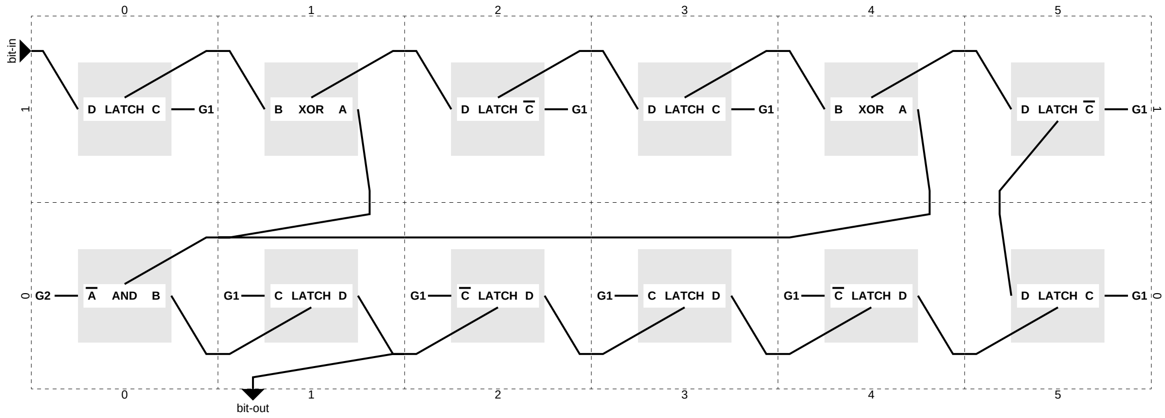

Fig. 4. CAL circuit for simple generator.

total number of circuit cells. The design tool has an adjustable parameter to specify the maximum allowable width of the circuit, and hence also its aspect ratio. The register is folded sufficiently many times to fit within this width constraint. During experimentation with various CRC circuits, a shortcoming of the CAL1024 chip was discovered, which limits the maximum width possible. The problem is that the global clock G_1 can only drive at most five latches on any one physical row of the chip (the chip has 64 physical rows of 16 cells each — the 32 cells in each logical row alternate between two adjacent physical rows). This means that it is unsafe here to have a circuit width larger than ten cells; the design tool enforces this limitation.

The folding process always uses an odd number of folds, so that the circuit has an even number of rows, with the first stage of the shift register at the same (left) side of the circuit as the final stage. This means that the feedback connections can be routed up the leftmost column, and then routed from left to right along those rows in which the data flows from right to left; given this, the maximum progagation delay is proportional to the sum of the circuit height and width. This folding arrangement also has the advantage that the output port is at the same side of the circuit as the input port, which is likely to be of use if the CRC circuit is incorporated into a larger circuit for constructing and transmitting messages. Figures 5 and 6 show the circuits produced for both the CCITT and the IEEE generator polynomials and, although the functions of individual cells are not visible, should make the above remarks clear. In the case of the IEEE circuit, the number of cells happens to be a multiple of the maximum width (10) and so the circuit area is fully utilised. This is not true in the case of the CCITT circuit, and a little area is wasted at the right of the top two rows, together with a single cell near the left of the bottom row; the latter is because the number of cells is odd.

The operation of the CRC circuits is straightforward. In each operational step, an input bit is clocked in when G_1 goes high, and then a 'computation' (in the form of the **exclusive-or** operations) is performed when G_1 goes low. The shift register can

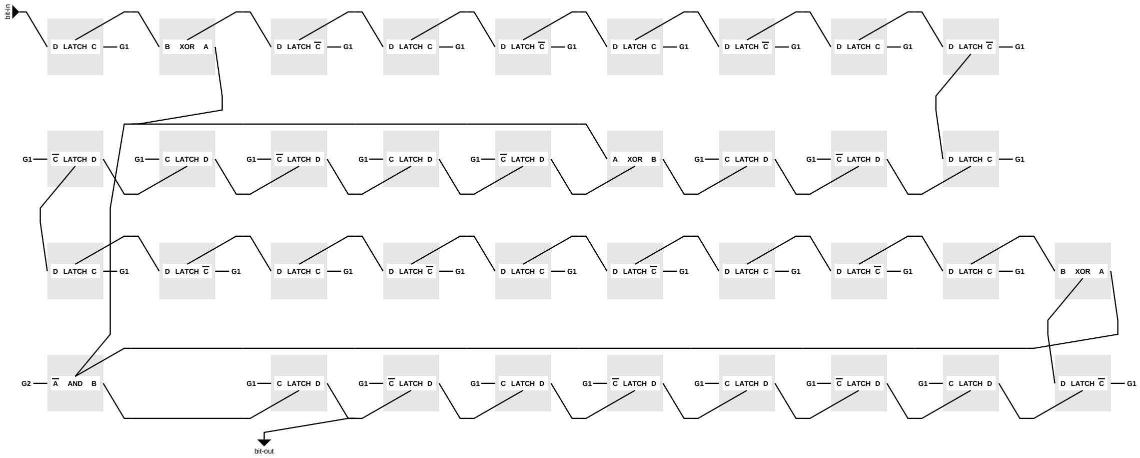

Fig. 5. CAL circuit for CCITT generator.

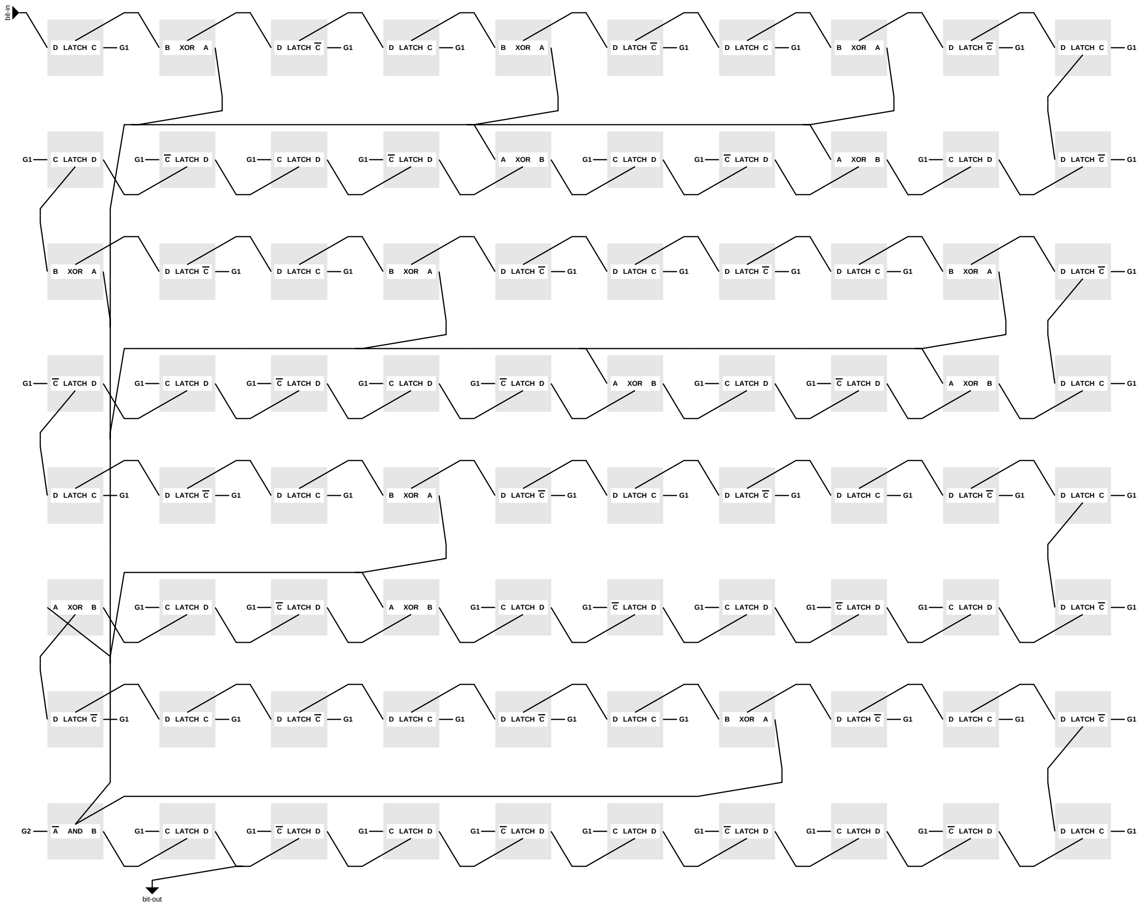

Fig. 6. CAL circuit for IEEE generator.

be initialised by supplying an appropriate initial pattern as sequence of k bits, while holding G_2 high to inhibit feedback to the **exclusive-or** gates. In fact, in the case of CRC computation as described earlier, it is unnecessary to explicitly initialise the register to all zeroes, since the first k message bits can be input while holding G_2 high, to achieve the same effect as feedback of a sequence of k zero bits. Such a saving of k initial steps is not possible for the computation variants prescribed both by the CCITT for its X.25 wide area network standard and by the IEEE for its 802 series local area network standards, because the shift register must be initialised to all ones. After k initial operational steps, the remaining message bits, followed by k zeroes, are input in subsequent steps while holding G_2 low, after which the register will contain the required result. To extract this result, a further k steps while holding G_2 high are necessary; however, the input bits supplied at this stage can be either the initialisation pattern for the next computation or, if this pattern is all zeroes, the first k bits of the next message if available. Thus, these circuits efficiently implement the computation of CRCs. (Note that the CCITT and IEEE variants also prescribe complementation of the k result bits before attachment to the message: this is assumed to be performed elsewhere by the circuit user.)

It is relatively easy to estimate the operation time for this circuit, although this time depends upon the generator polynomial used. The only operation during G_1 is the latching of input bits in the case of the first register stage, and of data bits from preceding latches in the case of subsequent stages. Input latching is potentially the more time-consuming and, as noted in the previous section, can be done in 18 ns. During $\overline{G_1}$, the time-dominant operation is feedback, and comprises (i) an **and** operation for possible inhibition (8 ns), (ii) propagation over the longest feedback connection of length ℓ say (2ℓ ns), and (iii) an **exclusive-or** operation (8 ns). Also, in all cases, there is the latching of data bits from preceding latches or from **exclusive-or** gates, which can be done in a further 8 ns. Thus, the minimum duration of $\overline{G_1}$ is $24 + 2\ell$ ns. In the case of the CCITT circuit, $\ell = 9$, and so the maximum operational rate of the circuit is 60 ns per bit. In the case of the IEEE circuit, $\ell = 13$, and so the maximum operational rate of the circuit is 68 ns per bit. In general, it is the case that *any* CRC with a generator of degree less than 64 can be computed in less than 100 ns per bit.

These rates are compatible with those of some CRC chips (e.g., Motorola M8500 or National Semiconductor 9401C with a maximum rate of 100 ns per bit), so it is the case that the CAL implementation can match fixed hardware implementations. As part of a general hardware component to accelerate communication functions, the CAL circuits are attractive, since they occupy only a fraction of a CAL1024 chip. As far as software implementations (usually byte-oriented rather than bit-oriented) are concerned, the VAX-11 computer has a special instruction which can compute a CRC, given a table of coefficients in memory, and it requires around 350 ns per bit; this compares with around 1000 ns per bit for a hand-coded VAX-11 assembler implementation [5]. More modern computers might offer rather faster rates, given suitably tailored assembly language implementations.

4. Modulo $2^n - 1$ Checksums

Modulo 2^n-1 checksums are in widespread use for performing error detection on byte-serial network connections or transport connections. They are usually computed by software as part of the process of packet construction. These checksums are not normally regarded as forming a class of codes, but a general mechanism can be found underpinning two error detection codes, which have emerged from the Internet community, and from the ISO international standardisation effort, respectively.

The first error detection code is the Internet checksum used both for IP protocol message headers and for TCP and UDP protocol messages [6]. In all cases, if a message of length w 16-bit words: $m_{w-1}, \ldots, m_0$ is to be checksummed, the one-word checksum is just the complement of

$$\sum_{i=0}^{w-1} m_i \pmod{65535}$$

and so the main operation required for this is summation modulo 65535 (i.e., summation using one's complement word addition). The checksum is included within the transmitted message and a recipient can check for transmission errors by performing a similar summation over the received data to check that the sum is zero; if it is not, a channel error has occurred. The reason for the use of 16-bit words is that the protocols were designed at a time when most switching nodes were 16-bit minicomputers. A discussion of the rationale behind the use of this error detection code can be found in [7].

The second error detection code is the ISO checksum used both for ISO 8473 connectionless network protocol message headers and for ISO 8073 transport protocol messages [4]. It employs an algorithm due to Fletcher [8]. If a message of length b bytes: $m_{b-1}, \ldots, m_0$ (from most to least significant) is to be transmitted along a connection, the major operation involved in determining the two-byte checksum for the message is the computation of the two one-byte quantities:

$$C_0 = \sum_{i=0}^{b-1} m_i \pmod{255} \quad \text{and} \quad C_1 = \sum_{i=0}^{b-1} (b-i)\, m_i \pmod{255}.$$

Given C_0 and C_1, the actual checksum can be simply computed; the details can be found in [5]. The checksum is included within the transmitted message and a recipient can check for transmission errors by performing a similar summation over the received data to check that both C_0 and C_1 are zero; if not, a channel error has occurred. A discussion of the rationale behind the use of this error detection code can be found in [8]. There is a good case for computing it as fast as possible, since it has been found that computation time for the ISO checksum can be a rate-determining step for communications throughput [5,9].

Algorithms to compute C_0 and C_1, with various degrees of bit-twiddling and tight programming, have been suggested [5,9,10] but the simplest, if not the most efficient, is just:

$$
\begin{array}{l}
C_0 := 0; \quad C_1 := 0; \\
\textbf{for } i := 0 \textbf{ to } b - 1 \textbf{ do} \\
\qquad C_0 := C_0 + m_i \pmod{255}; \\
\qquad C_1 := C_1 + C_0 \pmod{255};
\end{array}
$$

and so the main operation required for this is summation modulo 255 (summation using one's complement byte addition). Of course, addition modulo 256 (two's complement byte addition) is rather easier to implement, but ISO chose the modulus 255 rather than 256 because its error-detecting properties were deemed superior; this decision was perhaps questionable [5].

Given the above two error detection codes, it is clear that a hardware unit would usefully, for some n, accumulate the modulo $2^n - 1$ sum of a sequence of n-bit integers supplied as input and, when required, have its current sum available as output. In the Internet protocol case, a single 16-bit unit of this type is adequate, with data words as input. Then, the final sum output is ready for adjustment to produce the required checksum for incorporation into a transmitted message. In the ISO protocol case, a pair of 8-bit units is adequate: one unit has data bytes as input, and it outputs its intermediate sums as input to the second unit. Then, the final sum output by the first unit is C_0 and the final sum output by the second unit is C_1; these are ready for adjustment to produce the required checksum for incorporation into a transmitted message.

We have designed a CAL implementation of such units. The basic component is a standard element six cells wide and two cells high which implements a one-bit slice of a modulo $2^n - 1$ addition, and is shown in Figure 8. The idea is that an n-bit adder is formed by stacking n such elements on top of one another, with the least significant bit element at the bottom and the most significant bit element at the top. Variants of the standard element are used for the most, and the least, significant bit elements, and are shown in Figures 7 and 9 respectively. In this initial design, the circuitry was made simple and compact, with the inclusion of features such as a pipelined carry-chain handling being deferred for later consideration.

A one-bit input port (labelled "bit-in") is located on the left of each element at cell $(0, 0)$, connected to a latch cell clocked by G_1. A one-bit output port labelled "bit-out" is located on the right of each element at cell $(5, 0)$, connected to a latch cell clocked by $\overline{G_1}$. Except for the least significant bit element, the group of five cells at $(0, 1)$, $(1, 0)$, $(1, 1)$, $(2, 0)$ and $(2, 1)$ form a full adder operating on the input bit, a stored sum bit, and a carry bit from the preceding bit element. Cells $(0, 1)$ and $(2, 1)$ form a half adder operating on the input bit and the stored sum bit, and cells $(1, 0)$ and $(2, 0)$ form a half adder operating on the first half-sum bit and the carry bit. The stored sum bit is held in the one-bit register formed by the pair of latch cells at $(5, 1)$ and $(4, 1)$ in Figure 8, clocked by $\overline{G_1}$

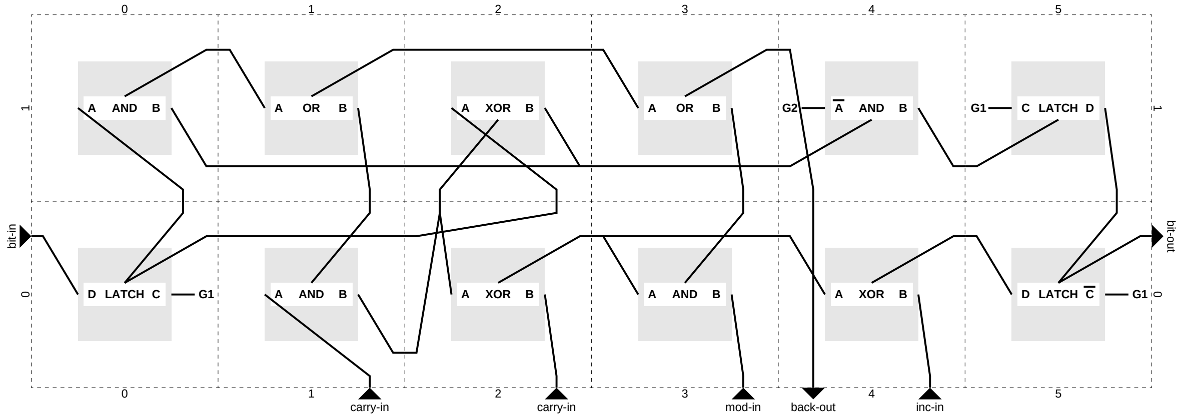

Fig. 7. CAL circuit for most significant bit adder slice.

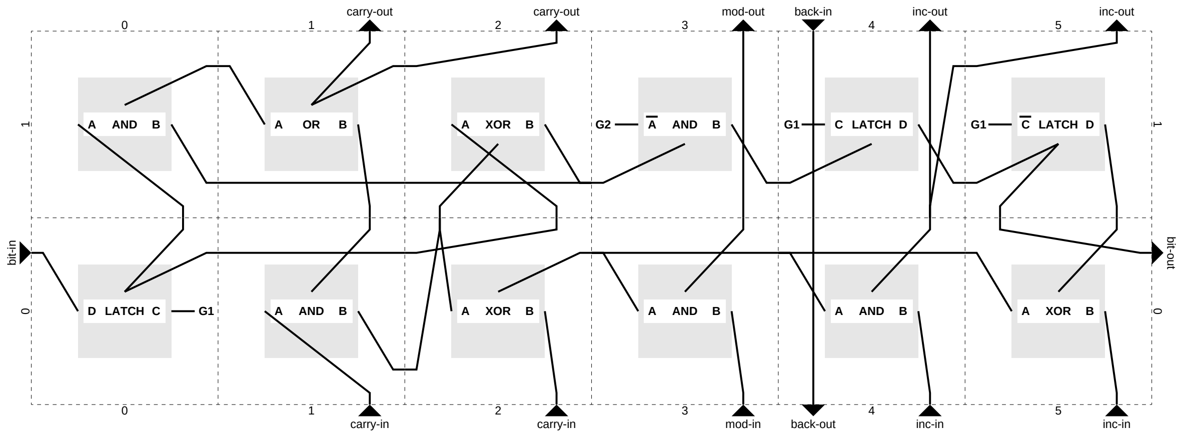

Fig. 8. CAL circuit for standard one-bit adder slice.

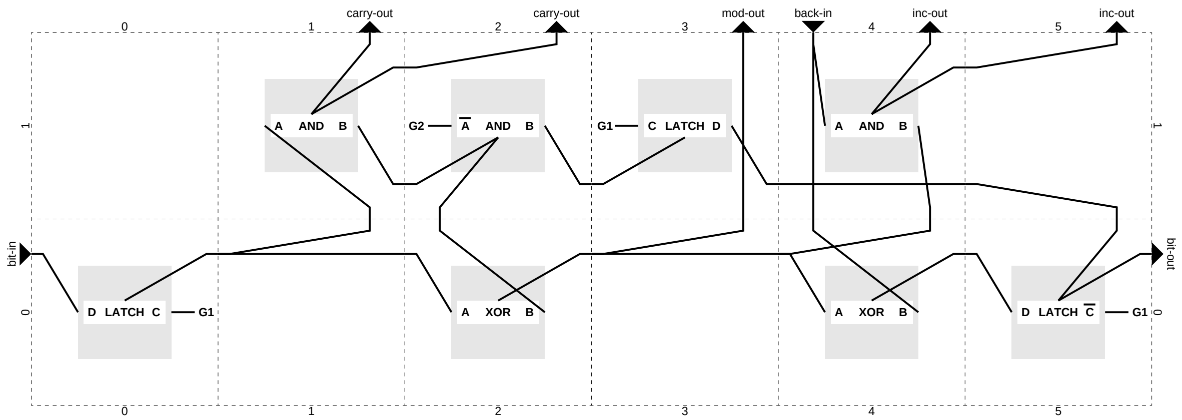

Fig. 9. CAL circuit for least significant bit adder slice.

and G_1 respectively. There is also an **and** function cell at $(3, 1)$ in Figure 8 so that a zero 'stored bit' can be supplied by the assertion of G_2; this is desirable when starting a new summation. The carry input bit is supplied at ports labelled "carry-in" at the bottom of both cell $(1, 0)$ and cell $(2, 0)$, and the carry output from the full adder is supplied to ports labelled "carry-out" at the top of both cell $(1, 1)$ and $(2, 1)$ so that it is available to the succeeding bit element above. The least significant bit element, which does not have a carry input, has only a half adder formed by cells $(1, 1)$ and $(2, 0)$. Note that both the most, and the least, significant bit elements have the two latch cells and the 'zero stored' **and** function cell in different positions from the standard element; these can be seen in Figures 7 and 9.

The circuit, as described so far, is appropriate for a modulo 2^n adder. The remaining cells are concerned with making the circuit implement a modulo $2^n - 1$ adder. In order to detect sums which are equal to $2^n - 1$, and should thus be adjusted to 0, the **and** function cell at $(3, 0)$ operates on the sum bit from the full adder, and an 'all less significant bits equal to one' bit which is supplied at a port labelled "mod-in" at the bottom of that cell. The result is supplied to a port labelled "mod-out" at the top of cell $(3, 1)$ so that it is available to the succeeding bit element above. The element used for the most significant bit of the adder differs from the other bit elements because, rather than outputting a carry bit and an 'all ones' bit to a succeeding bit element, it computes the logical or of these two bits using the **or** function cell at $(3, 1)$ in Figure 7. If the result is equal to one, then the sum computed by the stack of n full adders is equal to $2^n - 1$ or more, and so it must be incremented (ignoring any carry out from the most significant bit) in order to derive the correct sum modulo $2^n - 1$. (For example, with $n = 4$, $8 + 7 = 15$ (all ones) and incrementing gives 0, or $8 + 9 = 1$ (with carry out) and incrementing gives 2). This 'increment bit' is passed downwards to the least significant bit element. In the standard element, the increment bit is supplied at a port labelled "back-in" at the top of cell $(4, 1)$ and is supplied to a port labelled "back-out" at the bottom of cell $(4, 0)$ so that it is available to the preceding bit element below.

The least significant bit element does not pass on the increment bit to a preceding bit element, but instead uses it as an 'increment carry' bit to a half adder formed by cells $(4, 0)$ and $(4, 1)$ in Figure 9, the other addend being the sum bit from the other half adder. In the standard element, the two cells at $(4, 0)$ and $(5, 0)$ form a half adder, which operates on the sum bit from the full adder and an increment carry bit which is supplied at ports labelled "inc-in" at the bottom of both cells. The sum output from the half adder is stored in the one-bit register and the increment carry output from the half adder is supplied to ports labelled "inc-out" at the top of both cell $(4, 1)$ and $(5, 1)$ so that it is available to the succeeding bit element above. The most significant bit element does not produce an increment carry output, and so has only a 'quarter adder' at cell $(4, 0)$ in Figure 7.

Using these elements, it is possible to construct an n-bit modulo $2^n - 1$ summation circuit using a six cell wide and $2n$ cell high CAL array with no wasted area, except for

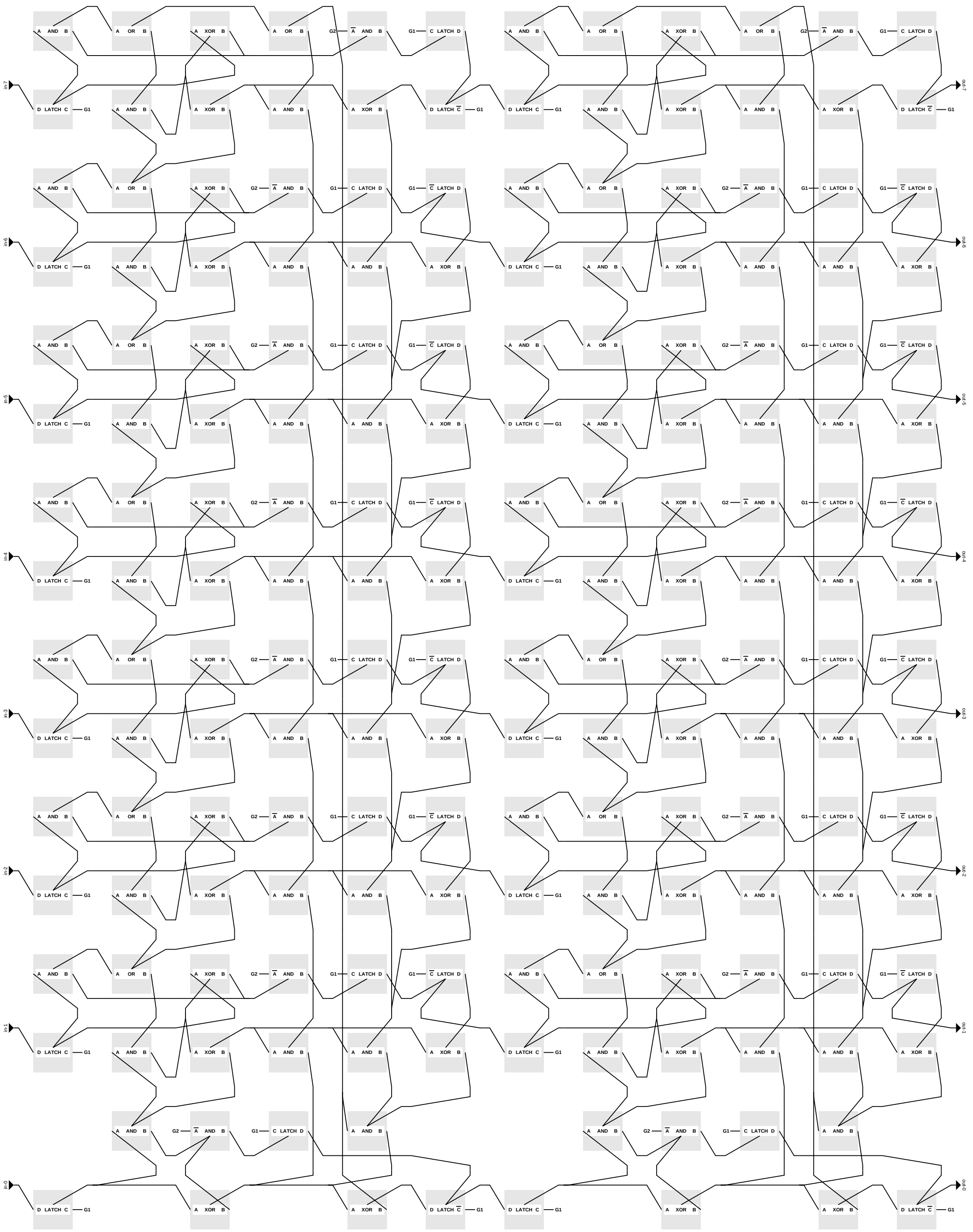

Fig. 10. CAL circuit for ISO summation.

two cells in each of the two bottom rows. One such circuit is adequate to perform the necessary summation for the Internet checksum and uses 6 x 32 = 192 cells. Two side-abutting circuits are adequate to perform the necessary double-stage summation for the ISO checksum and use 12 x 16 = 192 cells also. The overall layout of the latter circuit is shown in Figure 10; eight data bits are input in parallel at the left side of the circuit, and the eight bits of the 'C_1' sum are output in parallel at the right side of the circuit. The 'C_0' sum is apparently 'inaccessible' in the middle, where it supplies the input to the second summation stage, but a way to access it will emerge when the operation of the circuit is examined.

The operation of the basic summation circuit is straightforward. In each operational step, n input bits are clocked in in parallel and the n bits of the sum from the previous step are available as output in parallel when G_1 goes high. Then the computation of the new sum is performed when G_1 goes low. After all of a sequence of summands have been input, one further step is necessary to obtain the final sum output. The input at this step is unimportant, but it may be convenient to supply input bits which can re-initialise the accumulator for the next summation. The accumulator can be zeroed by supplying zeroes as input while holding G_2 high so that a zero 'stored bit' is added. In fact, it is unnecessary to explicitly zero the accumulator, since a similar effect can be achieved by holding G_2 high when supplying the first of a sequence of summands.

In the case of the ISO double-summation circuit, judicious operation at the end of the sequence of summands allows the value for C_0 to be obtained from the circuit output. At the step after the final (b-th) message byte has been input, a zero byte is input (and, meanwhile, the left summer outputs C_0 over all b bytes and the right summer outputs C_1 over the first $b-1$ bytes). At the next step, a zero byte is input with G_2 held high (meanwhile the left summer still outputs C_0 over all b bytes and the right summer outputs C_1 over all b bytes; this value of C_1 can be taken by the circuit user). Finally, at the next step, a zero byte (or the first byte of a new sequence of summands) is input with G_2 held high and, meanwhile, the left summer outputs zero and the right summer outputs C_0 over all b bytes; this value of C_0 can be taken by the circuit user. Furthermore, the subsequent addition during $\overline{G_1}$ zeroes the right accumulator in readiness for a new C_1 summation. Thus, the circuit is an efficient pipe-lined implementation of the main part of the ISO checksum algorithm.

The operation time of the summation circuit can be estimated. The only operation during G_1 is the latching of input bits and, as noted in Section 2, this can be done in 18 ns. During $\overline{G_1}$, the modulo $2^n - 1$ addition is performed; the timing analysis of this is a little intricate, and involves identifying the critical timing paths.

First, there is the carry chain progressing up the cells in column 1 of the circuit: the least significant bit element can supply its carry out after $8+8=16$ ns [path $(2,1)$;$(1,1)$], the second least significant bit element can supply its carry out after $8+8+2+8+8=34$ ns [path $(3,1)$;$(2,1)$;$(1,1)$;$(0,1)$;$(1,1)$], and each other bit element can supply its carry out

$8+8=16$ ns [path $(1,0)$;$(1,1)$] after receiving its carry in. Also, at the same time, there is the 'all ones' chain progressing up the cells in column 3 of the circuit: the least significant bit element can supply its output after $8+8+2+2=20$ ns [path $(2,1)$;$(2,0)$;$(3,0)$;$(3,1)$], and each other bit element can supply its output $8+8+2=18$ ns [path $(2,0)$; $(3,0)$;$(3,1)$] after receiving its column 2 carry in.

Second, there is the 'increment bit' path progressing down the cells in column 4 of the circuit: the most significant bit element can supply the result of its increment test $8+8+2+8+2+2=30$ ns [path $(1,0)$;$(1,1)$;$(2,1)$;$(3,1)$;$(4,1)$;$(4,0)$] after receiving its column 1 carry in (the later arrival of the 'all ones' input is not significant: the carry chain is actually the critical path above), and each other bit element can supply its output $2+2=4$ ns [path $(4,1)$;$(4,0)$] after receiving its input.

Third, there is the increment carry chain progressing up the cells in column 4 of the circuit: the least significant bit element can supply its output 8 ns [path $(4,1)$] after receiving its 'increment bit' input, and each other bit element can supply its output $8+2=10$ ns [path $(4,0)$;$(4,1)$] after receiving its increment carry bit input.

Finally, the most significant bit element requires 8 ns [cell $(4,0)$] after receiving its increment carry bit input, and then 8 ns to latch its final sum bit. The total time required for the sequential operation of the three critical paths above is

$$\Big(34+16(n-3)\Big)+\Big(30+4(n-2)\Big)+\Big(8+10(n-2)\Big)\text{ns}$$

and so the minimum duration of $\overline{G_1}$ is $30n+12$ ns. Therefore, the maximum operational rate of the circuit is $30(n+1)$ ns per n-bit input.

In the case of the Internet checksum, the maximum operational rate of the circuit is 510 ns per word and, in the case of the ISO checksum, the maximum operational rate of the circuit is 270 ns per byte. Due to its two-stage pipelining, the CAL implementation operates at approximately the same byte rate for both types of checksum, whereas experiments indicate that efficient software implementations of the ISO checksum may be around five times slower than those of the Internet checksum (although the ratio varies with message size) [10]. (The relative simplicity of the Internet checksum computation reflects a characteristic difference between the pragmatic attitudes of Internet implementors and the more idealistic attitudes of ISO specifiers.)

The circuits operate at a rate compatible with a communication speed of around 30 Mbits per second, which is in excess of the capacity which can be made available on most present-day inter-network or transport level connections. However, these rates may appear sluggish, compared with some rates reported for hand-crafted assembler language implementations of the Internet checksum; for example, around ten times faster than the CAL circuit on an IBM 370, and around twice as fast as the CAL circuit on a 20 MHz Motorola 68020 [11]. A reported rate for fast ISO checksum software for a (comparatively slow) VAX-11 is around ten times slower than the CAL circuit [6]. Typical high-level language implementations of both checksums are very much slower.

Viewed as part of a general hardware component to accelerate communication functions, the circuit is certainly not unreasonable, since it occupies less than one-quarter of a CAL1024 chip, and so the remainder of the chip, and other CAL chips on a board, can be used to compute the same, or other, functions in parallel. In contrast, software implementations tie up the entire processing capacity of a computer.

Fast software implementations of both checksums employ techniques which are not reflected in the above circuits. In general, modulo $2^n - 1$ adjustment is not performed after each addition step: instead, addition is performed using integers of length $2n$, for example, and adjustment is performed infrequently, sometimes only once per message; the addition operations themselves are fast since they can harness the normal optimised adder hardware in the general-purpose processor. These observations motivate an improved family of CAL circuit designs which, although less compact, offer greatly improved computation speeds.

A major improvement which is possible at this level is to introduce bit pipelining into the summation circuitry to avoid the delays due to carry propagation. Note that the complete CAL circuits described above cannot be pipelined straightforwardly because of the existence of the modulo $2^n - 1$ increment bit which is passed *back* from the most significant bit element to the least significant bit element. To eliminate this problem, the new circuits have two stages: a pipelined modulo 2^{kn} summation circuit for some small $k > 1$ (or a linked pair of such circuits in the case of the ISO checksum), and a modulo $2^n - 1$ conversion circuit which sums the k base-2^n digits of the accumulator value, modulo $2^n - 1$, to compute the equivalent value modulo $2^n - 1$. The second conversion circuit must be used before overflow occurs in the first circuit: the necessary frequency of use is dependent on the choice of k. The frequency is reduced by increasing k, but at the expense of increasing the required number of cells.

In the case of a simple accumulator, as used for the Internet checksum or for the C_0 computation of the ISO checksum, where the maximum value added at each step is $2^n - 1$, it is possible to add $(2^{kn} - 1)/(2^n - 1)$ values before overflow might occur. In the simplest case when $k = 2$, this means that 2^n new values can be added to the initial value stored in the accumulator. For the Internet checksum, where $n = 16$, up to 65536 words could be added: this is double the maximum number of words allowed in a single packet. For the ISO checksum, where $n = 8$, this is less attractive because only 256 bytes could be added: rather less than many typical packet sizes. In fact, the situation is far worse when the second accumulator for the C_1 computation is taken into account. Given that the initial values in the C_0 and C_1 accumulators are less than 255 then, after b bytes have been added, the value in the C_1 accumulator is less than or equal to:

$$254 + 254 \times b + 255 \times \sum_{i=1}^{b} i$$

which is only less than 65535 for $b \leq 21$. This means that the maximum number of consecutive additions in the first stage circuit is only slightly larger than the length of its

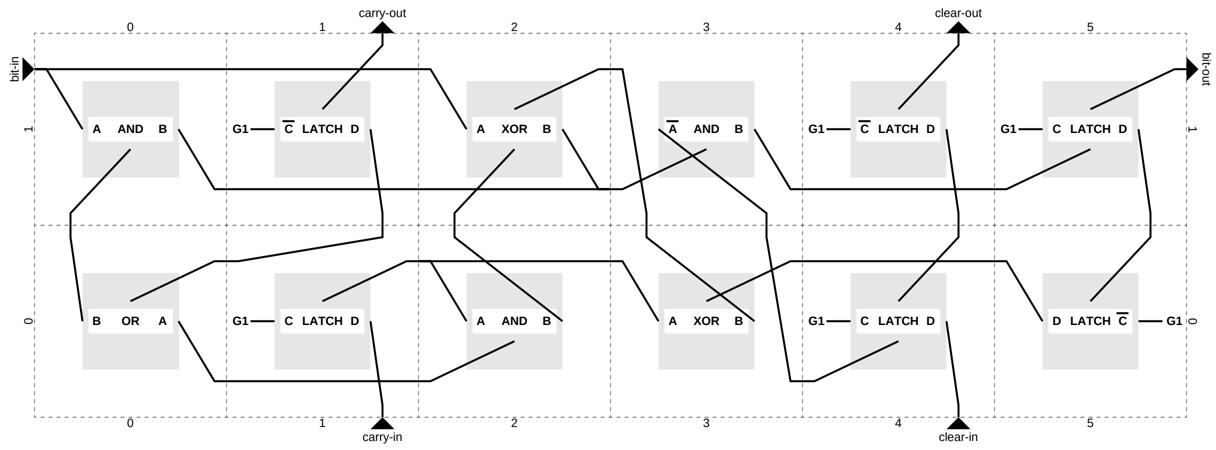

Fig. 11. CAL circuit for pipelined bit adder slice.

pipeline (16 stages). However, choosing $k = 4$, the position is much improved: up to 5802 bytes can be added before overflow might occur in the C_1 accumulator.

The basic component of the pipelined accumulator is a standard element six cells wide and two cells high which implements a one-bit slice of a pipelined 2^{kn}-bit addition, and is shown in Figure 11. The group of five cells at $(0,0)$, $(0,1)$, $(2,0)$, $(2,1)$ and $(3,0)$ form a full adder operating on an input bit arriving at the left of cell $(0,1)$, a stored sum bit from latch $(5,1)$, and a carry bit from latch $(1,0)$. The resulting sum bit is stored in latch $(5,0)$ and the carry bit is stored in latch $(1,1)$. As in the earlier basic element, the **and** function cell at $(3,1)$ allows zero to be added instead of the stored sum bit; the difference is that this mechanism is not triggered by the global signal G_2, but by a pipelined clear signal which arrives at cell $(4,0)$ and leaves at cell $(4,1)$. Another difference from the earlier element is the absence of a latch for the input bit and, matching this, the output bit departing from an 'input' latch. A timing analysis of this element reveals that the minimum duration of G_1 is 18 ns (as usual), and of $\overline{G_1}$ is 46 ns, giving a total time of 64 ns per input bit.

Variant components were designed for the least and most significant bit elements, and for the $2^{(k-1)n}$ more significant bit elements, which have no input bits; full details of these are omitted (but they can be viewed to small scale in Figure 12). The effect of appropriately stacking kn elements is to construct a pipelined 2^{kn}-bit adder which is clocked by the global signal G_1. A complete circuit for the ISO checksum, incorporating a 16-stage pipeline (i.e., $k = 2$), is shown in Figure 12.

Eight data bits are input at the lower left-hand side, and the eight bits of the C_0 and C_1 sums are output at the upper right-hand side. The paired summation circuits form a main 12 x 32 cell unit in the left-hand part of the circuit. A diagonally-placed unit in the top-right part of the circuit is a pipelined adder with an additional "all ones" carry; this is the main part of the second circuit stage to add the two halves of the accumulator outputs. A small 8 x 2 cell unit at the extreme top right is the only non-pipelined unit,

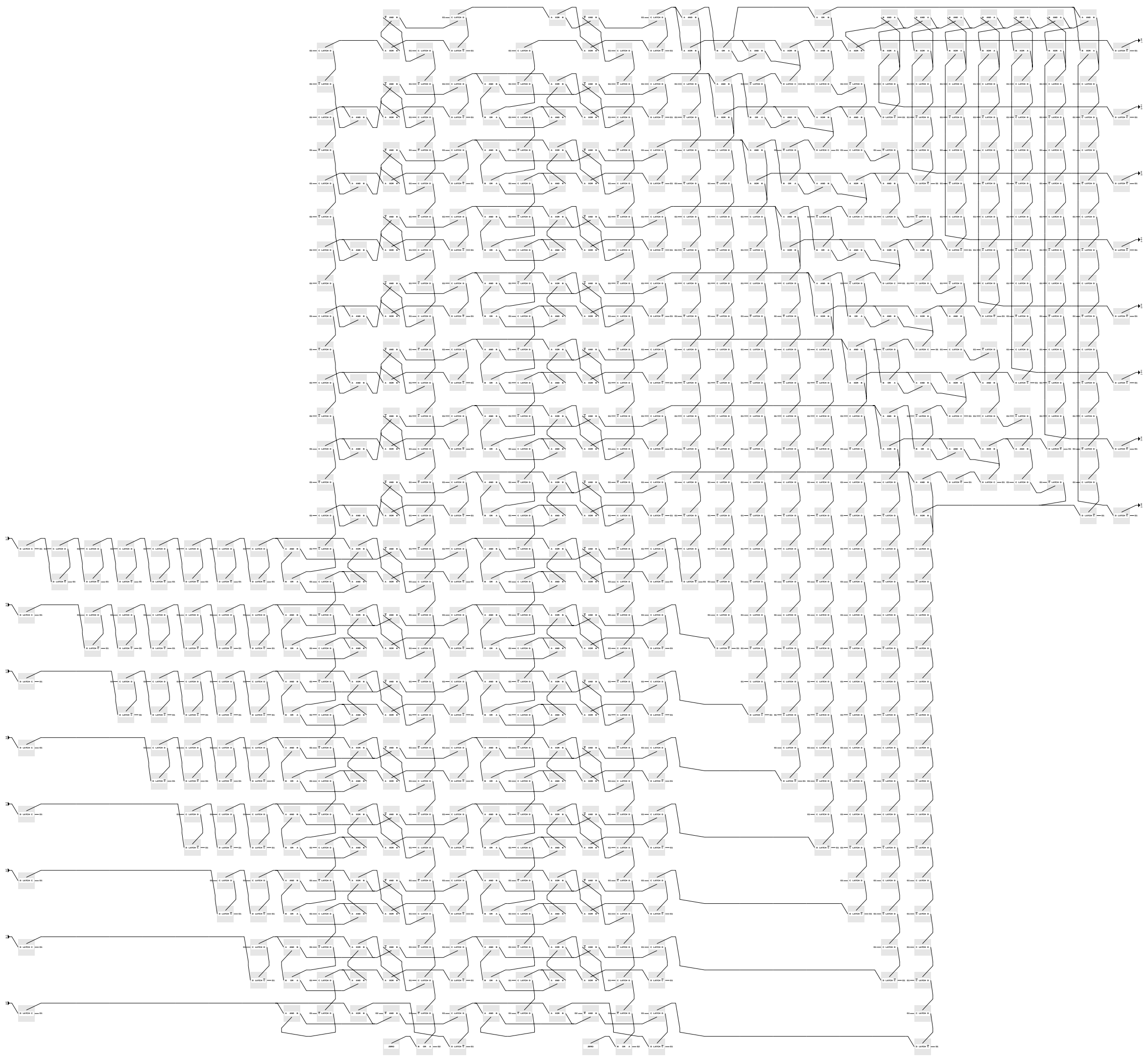

Fig. 12. CAL circuit for pipelined ISO summation.

and performs the final modulo 255 correction (if necessary). All other components are shift registers to provide the appropriate delays for correct pipeline operation. Some of these act as interfacing 'glue circuitry', being concerned with skewing inputs and deskewing outputs; however, by employing slightly more complex circuitry in the second summation stage (e.g., using carry-save addition), these shift registers could be omitted and some space would be saved. Full details of the extra units are omitted here, since their nature is not a major influence on the overall performance of the circuit. A slight variant of the circuit was successfully tested using the CHS2x4 board. This variant took account of the limitation on the number of latches placed on each physical row of a CAL1024 chip (mentioned in Section 2), and involved spacing the circuit across three adjacent chips by inserting appropriate segments of horizontal wiring.

A timing analysis of the additional circuitry reveals two critical features. First, two of the stages in the pipelined modulo 255 adder require a slightly longer duration of $\overline{G_1}$ than the time mentioned above: 66 and 68 ns respectively. More significantly, the final modulo 255 correction unit requires a minimum duration of 144 ns before all of the circuit outputs have been latched. The minimum duration of G_1 is not affected. Given these figures, the overall pipelined operation of the circuit can be analysed, when it adds b data bytes ($b \leq 21$) to the sums C_0 and C_1. The circuit is used in the following phases:

1. 2 cycles to initialise the accumulators: 128 ns;
2. b cycles to add in data bytes: $64b$ ns;
3. 8 cycles to complete the least significant half pipeline: 512 ns;
4. 8 cycles to add the accumulator halves: 544 ns; and
5. 2 cycles to perform modulo 255 correction: 324 ns

giving a total of $64b + 1508$ ns. When $b = 21$, this gives a maximum operational rate of approximately 136 ns per byte: almost exactly twice the speed of the earlier non-pipelined ISO summation circuit. However, the circuit is significantly larger, and only just fits on to a single CAL1024 chip when the input and output latches are disregarded.

The major flaw in the circuit is the fact that the pipeline has to be drained after every 21 data bytes. An improved version was designed with a 32-stage pipelined adder unit. The modulo 255 correction circuitry was extended to add together the four bytes of the accumulators modulo 255 (this extension was not performed with particular care for efficiency). The resulting circuit could be accomodated on a 2 x 2 array of CAL1024 chips. The new circuit operation time when adding b bytes ($b \leq 5802$) was found to be $64b + 2774$ ns. When $b = 5802$, this gives a maximum operational rate of less than 65 ns per byte, which is a further doubling of computation speed. It was not possible to test this circuit on the CHS2x4 board, because of the increased number of latches per row of cells; however, assuming the advent of a corrected version of the CAL1024 chip, no functional problems are envisaged.

In the case of the Internet checksum, we note that a 32-stage pipelined adder unit can also be used to improve the computation speed significantly. Here, all of the arithmetic is performed on 16-bit quantities, yielding a modulo 65535 result, so the circuitry is essentially just a double-length version of that used for the ISO checksum computation. Such a circuit can be accomodated on a 2 x 2 array of CAL1024 chips. The operation time of the complete circuit, when adding w 16-bit words ($w \leq 65536$), was found to be $64w + 3472$ ns. For realistic values of w, this is almost eight times the speed of the original Internet summation circuit, which is a significant improvement.

The basic performance gain from pipelining can be derived in a relatively technology-independent manner. For simplicity, the inflated timings of the first and last units of the original sequential adder will be ignored, as will the timings of the first and last few cycles of the pipelined adder. Suppose that g is the settling time for a logic gate. Then, in the case of a one-bit slice in the sequential adder, its computation time is dominated by $2g$ to complete carry computation in the full adder, plus (later) g for increment carry computation in the half adder. In the case of a one-bit slice in the pipelined adder, its computation time is dominated by g for optional clearing, plus $3g$ for carry computation in the full adder, plus g for latching the carry output. In both cases, these times can be assumed to dominate any wire propagation delays, given a relatively dense tiling of components. Now, if the adders are handling n-bit values, the time for the sequential adder is $3ng$, whereas the time for the pipelined adder remains $5g$. Finally, to process a sequence of ℓ n-bit values, the total times are $3ng\ell$ and $5g\ell$ respectively plus, in each case, a further $g\ell$ for input latching, giving a total of $(3n+1)g\ell$ and $6g\ell$ respectively. Thus, the speed-up from pipelining is approximately $(3n+1)/6 \approx \frac{n}{2}$. This matches the earlier observations from more detailed CAL timing estimates.

At the expense of using more cells, the basic pipelined unit can be further exploited to achieve faster speeds. The above circuits all have single byte (ISO checksum) or single word (Internet checksum) inputs. If two consecutive inputs are supplied in parallel to a doubled-length adder unit then, for large enough packets, the per-byte or per-word rate can be effectively doubled. For example, such a version of the above Internet checksum unit could achieve a rate of about 32 ns per word, which comfortably exceeds the fastest hand-tailored software implementations running on powerful processors.

5. Conclusions

We have produced working CAL circuits which can compute all of the major types of error detection code used for present-day computer networking. In their simplest forms, each of the circuits can be accommodated on a single CAL 1024 chip containing a 32 x 32 array of programmable cells. These circuits could be used as components of more complex communications hardware, implemented with CAL or otherwise. In the case of cyclic redundancy codes, which are popular because of their ease of implementation in hardware, the CAL circuits can perform at a level comparable with specialist chips. In the case of modulo $2^n - 1$ checksums, which are popular because of their ease of implementation in software, simple CAL circuits can perform at a level far superior to general-purpose software implementations, and more complex circuits can perform at rates in excess of special-purpose software for sophisticated processors. As an alternative to forming part of larger communications hardware, the circuits also have a rôle as co-processing accelerators, particularly in the case of the ISO checksum.

This work has indicated that, by using more complex circuit designs, it is possible to trade-off speed and circuit area. This is by no means an original observation, but the novel angle is that trade-offs can be accomodated using the same physical piece of hardware. The configurability of the CAL array allows an appropriate level of complexity to be selected dynamically. Thus, it is possible to envisage a general-purpose communication function accelerator which not only allows an appropriate range of functions, but also a range of performances. A run-time 'real estate' manager might organise an appropriate layout of components on the CAL array area available for use by the accelerator, in a manner akin to memory managers or window managers.

Apart from any advantages of using configurable hardware as part of a final production system with flexible characteristics, it was the case that using configurable hardware made design experiments particularly easy during this work. The design cycle time would have been considerably lengthened had other types of hardware been used. The circuits described in this paper were the author's first experience of CAL (or, indeed, any kind of) circuit design. Given this, the central problem for someone from a software background was dealing with timing issues. The $G_1/\overline{G_1}$ mechanism which pervades the circuits described here has emerged as the central feature which allows trouble-free CAL circuit design and testing, and it has been distilled out to yield a simple guide to CAL work [12].

In particular, use of this methodology made the pipelining of the carry chain in the modulo $2^n - 1$ adder particularly straightforward. Although the redesigned unit here was produced by hand, there seem to be no major conceptual barriers to the automation of such a pipelining process, when circuits follow this $G_1/\overline{G_1}$ mechanism. Essentially, an automated system would insert latch pairs between output and input ports of 'appropriate' adjacent functional units (here, between the one-bit slice adder units) which form part of a hierarchical design. (Note that dealing with more subtle matters like the factoring out of the backwards increment bit in the modulo $2^n - 1$ adder would be distinctly harder to automate.) To determine automatically where simple pipelining is possible, some circuit analysis would be necessary. The beginnings of such circuit analysis are embodied in a design tool already produced which, assuming the above timing methodology, finds critical paths to estimate circuit operation times. Of course, a disappointment is the current inability to check that circuits operate at their purported maximum speeds. However, it is hoped that a modified version of the controller on the CHS2x4 board can be produced to rectify this shortcoming.

Aside from problems with time-related matters, the author found little trouble in producing functionally-correct circuits with array-style connectivity, the main interesting intellectual puzzles being involved with producing compact and fast processing elements. The basic designs for these were done using paper and pencil, but it would have been possible to use the graphical CAL design tool instead.

The performance available from CAL will improve with the use of better CMOS technology. For example, the suppliers suggest that the next generation of CAL will use 0.8 to 1.0 micron CMOS, and is likely to work around three times faster, as well as containing four times the number of cells. Given this, all of the circuits described here would fit on a single chip, in addition to the obvious speed improvement.

References

[1] T. Kean, "Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation," University of Edinburgh Ph.D. Thesis, 1989.

[2] J. Gray and T. Kean, "Configurable Hardware: A New Paradigm for Computation," *Proc. of Decennial Caltech Conference on VLSI,* Pasadena, CA, 1989.

[3] T. Kean, personal communication, 1991.

[4] W. Stallings, *Data and Computer Communications* (3rd edition), New York: Macmillan, 1991.

[5] A. Nakassis, "Fletcher's Error Detection Algorithm: How to implement it efficiently and how to avoid the most common pitfalls," *Computer Communications Review,* vol. 18, 1988, pp 63-88.

[6] D. Comer, *Internetworking with TCP/IP,* London: Prentice-Hall International, 1991.

[7] W. Plummer, "IEN-45: TCP Checksum Function Design," *Computer Communications Review,* vol. 19, April 1989, pp 95-101.

[8] J. Fletcher, "An Arithmetic Checksum for Serial Transmission," *IEEE Trans. on Communications,* vol. COM-30, 1982, pp 247-52.

[9] A. Cockburn, "Efficient Implementation of the OSI Transport Protocol Checksum Algorithm Using 8/16-bit Arithmetic," *Computer Communications Review,* vol. 17, July 1987, pp 13-20.

[10] K. Sklower, "Improving the Efficiency of the OSI Checksum Calculation," *Computer Communications Review,* vol. 19, October 1989, pp 32-43.

[11] R. Braden, D. Borman and C. Partridge, "Computing the Internet Checksum," *Computer Communications Review,* vol. 19, April 1989, pp 86-94.

[12] G. Brebner, "CAL on CHS2x4 Boards Made Simple," University of Edinburgh Department of Computer Science Technical Report, 1992.