[4] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer-Aided Design*, pages 1062–1081, November 1987.

[5] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, 1976.

[6] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[7] S. Devadas and K. Keutzer. Design of Integrated Circuits Fully Testable for Delay Faults and Multifaults. In *Proceedings of the Int'l Test Conference*, pages 284–293, October 1990.

[8] S. Devadas and K. Keutzer. Synthesis and Optimization Procedures for Robustly Delay-Fault Testable Logic Circuits. In *Proceedings of the 27th Design Automation Conference*, pages 221–227, June 1990.

[9] H. Fujiwara. *Logic Testing and Design for Testability*. MIT Press, Cambridge MA, 1985.

[10] G. D. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On the Relationship Between Area Optimization and Multifault Testabilty of Multilevel Logic. In *Int'l Conference on Computer-Aided Design*, pages 422–425, November 1989. (Extended version submitted to IEEE TCAD).

[11] N. K. Jha and S. Kundu. *Testing and Reliable Design of CMOS Circuits*. Kluwer Academic Publishers, 1990.

[12] S. Kundu, S. M. Reddy, and N. K. Jha. On the Design of Robust Multiple Fault Testable CMOS Combinational Logic Circuits. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 240–243, November 1988.

[13] C. Y. Lee. Representation of Switching Circuits by Binary Decision Programs. In *Bell System Technical Journal*, volume 38, pages 985–999, July 1959.

[14] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 6–9, November 1988.

[15] A. Pramanick, S. Reddy, and S. Sengupta. Synthesis of Combinational Logic Circuits for Path Delay Fault Testability. In *Int'l Symposium on Circuits and Systems*, pages 3105–3108, May 1990.

[16] K. Roy, K. De, J. A. Abraham, and S. Lusky. Synthesis of Delay Fault Testable Combinational Logic. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 418–421, November 1989.

[17] A. Saldanha, K-T. Cheng, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization and Testability Considerations. In *the Int'l Conference on Computer-Aided Design*, November 1990.

Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)

Mark E. Dean[1], Ted E. Williams and David L. Dill[2]
Computer Systems Laboratory
Stanford University

ABSTRACT

This paper proposes a dual-rail signalling method which allows an efficient implementation of Boolean functions as a self-timed logic module. This signalling method, called Level-Encoded 2-Phase Dual-Rail (LEDR), differs from the traditional dual-rail signalling methods by not requiring spacer tokens between data tokens. A LEDR design style can increase throughput in a pipeline structure by as much as 100% over traditional 4-phase, dual-rail design styles. A logic function is also simpler to implement in the LEDR design style than with transition signaling methods. Several LEDR design styles are provided. Use of LEDR signalling in pipelined and feedback logic structures is discussed. Transistor and gate level logic implementations are provided for comparison.

1 INTRODUCTION

Self-timed logic provides a method for designing asynchronous logic circuits such that their correct behavior is independent of the speed of their components or signal wire delays. [SEIT80] gives an extensive discussion of self-timed logic and its advantages over globally clocked, or synchronous, logic. One important advantage of self-timed logic functions is they can signal when a computation completes, rather than waiting for the worst-case delay. In synchronous designs this worse-case delay is factored into the clock cycle time.

Dual-rail signalling is one widely used style of self-timed circuit design. In the traditional style of dual-rail design, called **4-phase dual-rail**, three "logical" values are used: **0**, **1**, and **Invalid**. Every logical variable **x** is encoded using two wires, $\mathbf{x}^0$ and $\mathbf{x}^1$, called an **encoding pair**. The protocol for dual-rail signalling requires that the signal return to the invalid value after taking a 0 or 1 value. In essence, the invalid logic values serve as **spacer tokens** which separate the valid tokens in the data stream. This provides a means for the

[1]Mark E. Dean is supported by IBM.
[2]David L. Dill is supported by NSF under grant number MIP-8858807.

self-timed logic to detect completion of a logic function for each data token. Otherwise, it would not be possible to separate two consecutive tokens that happen to have the same value.

Several methods of designing self-timed logic can be found in [SEIT80], [ANAN86], [SING81], and [DGY89]. In these design styles the functional delay through a logic block for a spacer token is approximately the same as for a data token. Other dual-rail design styles use a control signal to reset or precharge all gates in the function block in parallel before accepting the next data token. This type of pre-charged, dual-rail logic implementation [MENG88] reduces the delay required to process a spacer token.

Spacer tokens reduce throughput. One alternative is to use **transition signalling**. Perhaps the most obvious way to implement transition signalling is to interpret a transition (inversion of signal) on wire x^i (i = 0, 1) of an encoding pair as a new datum with value i. Unfortunately, it is difficult to implement functional units that use transition signalling: to decode a token value from x^0 and x^1, one needs to know not only the current value, but the previous value as well. For example, the encoding $x^1x^0 = 00$ could mean either "0" or "1", depending on whether the previous state was 01 or 10.

This paper proposes an alternate dual-rail signalling method called Level-Encoded 2-Phase Dual-Rail or LEDR. The LEDR signalling method uses two wires to encode data values for each input/output variable, like the other dual-rail signalling methods. LEDR encoding does not require a spacer token in its data codes, as in 4-phase dual-rail, and it is much easier to implement than transition signalling. Data tokens are encoded into two possible **phases**, EVEN or ODD. Each data token in a data stream must have the opposite phase of the data token preceding it. Table 1 lists the variable encoding scheme used in the LEDR implementations described in this paper. Note that the Z^1 wire always carries the logic state of the data token, while the Z^0 wire is equal to Z^1 on even phases and inverse of Z^1 on odd phases. This encoding insures that exactly one of the dual-rail encoding wires changes value for every successive data token.

Section 2 gives a detailed definition of the guidelines for implementing LEDR logic structures. Characteristics of LEDR signalling are discussed in Section 3. Several design styles can be used to implement LEDR logic, each providing different levels of synthesis complexity, performance and implementation size. These design styles are described in Section 4. Section 5 describes how LEDR function blocks can be configured within a pipelined structure. The performance of the design styles of Section 4 is analyzed in Section 6. Section 7 summarizes the LEDR approach.

Z^1	Z^0	Phase	State
0	1	ODD	0
1	0	ODD	1
1	1	EVEN	1
0	0	EVEN	0

TABLE 1: Dual-Rail encoding for LEDR implementations.

Z^1	Z^0	State
0	1	0
1	0	1
1	1	unused
0	0	invalid (spacer token)

TABLE 2: Dual-Rail encoding for 4-phase dual-rail implementations.

2 DEFINITION OF LEDR IMPLEMENTATION OF BOOLEAN FUNCTIONS

A LEDR logic function **F** has **n**, 2-phase inputs and **m**, 2-phase outputs each with 2 possible value states per phase. Each input and output variable is made up of an encoding pair. The phases are defined as ODD and EVEN while the value states are binary values 0 and 1. The LEDR encoding of signals in Table 1 can be contrasted with 4-phase encoding of the referenced dual-rail implementations shown in Table 2.

If all inputs of a LEDR function are in the same phase, the outputs will transition to that phase and the logic states of the outputs will be defined by the Boolean transfer functions of **F**. A "strict" implementation of **F** is defined such that the outputs variables will remain in their previous state and phase until all input variables have changed to the alternate phase. An "eager" implementation is defined such that an output variable can change phase and state when enough input variables have changed to the alternate phase and the appropriate states to guarantee the correct output value independent of the other inputs. This paper will primarily discuss strict implementations. Strict design styles guarantee there are no internal nodes still transitioning when all the inputs and outputs are in the same phase.

The "strict" sequential behavior of the logic function F is specified by the following cycle of activities (which directly correspond to the "weak conditions" in [SEIT80]):

I All inputs and outputs are in the same phase, p_i.

II Some inputs are set to the alternate phase, p_{i+1}. All outputs remain in a p_i phase.

III All inputs are eventually set to the alternate phase. Their logical state may or may not have changed. Outputs remain in a p_i phase.

IV The outputs begin to change phase to p_{i+1}. Their final states will depend on the input variable's state and the Boolean function implemented.

V All inputs and outputs are in the same phase, p_{i+1}.

Each subsequent data token must have the opposite phase of the preceding token presented to the function. The phase of the input variables must not change again until the output phase matches the input phase. In a single thread network each token maintains the same phase as it passes through each function in the network. Selective phase inversion may be required in multi-threaded networks with branches.

Each output variable must make only one transition, to its final phase and logic state, for each group input variation. This is implied in the definition of sequential behavior given above. Therefore a LEDR design style must be free of any functional hazards or race conditions. The logical implementation can thereby avoid any false completion detection at the output of each functional block.

3 CHARACTERISTICS OF LEDR SELF-TIMED CIRCUITRY

One important characteristic of the 4-phase dual-rail implementations is that only one signal wire within an encoding pair changes from one token to the next. The spacer token ensures this characteristic. This makes completion detection simple. LEDR implementations also have this characteristic. Because a token, T_{i+1}, must have a different phase than token T_i, only a single wire changes independent of the logic states of T_i and T_{i+1}. Figure 1 illustrates the differences between single-rail encoding, 4-phase dual-rail encoding, transition signalling encoding and LEDR encoding relative to single bit data stream.

Logic State	0		0		1		0		0		1		1		1	
4-Ph. Dl-Rl, x^1x^0	01	00	01	00	10	00	01	00	01	00	10	00	10	00	10	
Trans. Sig., x^1x^0	00		01		11		10		11		01		11		01	
LEDR, x^1x^0	00		01		11		01		00		10		11		10	

FIGURE 1: Single variable data stream example.

The LEDR encoding also allows simple completion detection. An exclusive-or gate can be used to detect the change in phase from one token to the next. A C-element can then be used to combine the outputs of the exclusive-or gates of all the outputs of the function block to signal when they all reach the same phase. (A C-element is a storage element which generates a logic 1 at its output when all its inputs are 1 and a logic 0 when all its inputs are 0. When the C-element inputs are not equal, it holds its output at the previous logic state.) Figure 2 shows the completion detection circuitry for a 3-output, 4-phase functional block. Figure 3 illustrates how the completion detection circuitry is built for a 3-output LEDR function block. The completion circuit for the 4-phase function block generates a 4-phase signal, while the completion circuit for the LEDR function block generates a 2-phase signal.

4 IMPLEMENTATION ALTERNATIVES

We will discuss three implementation alternatives for LEDR function blocks. The first, called **LEDR-PLA**, will be based on a PLA structure which is modified to operate as a LEDR logic block. The **LEDR-EVAL** design style uses a pseudo-domino logic structure with dynamic storage. The last design style, called **LEDR-SS**, uses dynamic storage and decodes of the active high and active low logic terms to implement a LEDR structure.

4.1 LEDR-PLA Design Style

One of the simplest ways to implement a logic function is to use an "AND-OR" network to decode the desired input variable combinations for output variable generation. The is usually accomplished by using a programmable-logic-array (PLA) structure. By using a modified PLA structure, Boolean functions can be implemented with LEDR signal encoding. Figure 4 shows a basic block diagram of a LEDR-PLA structure. Each input signal pair is inverted and buffered. This

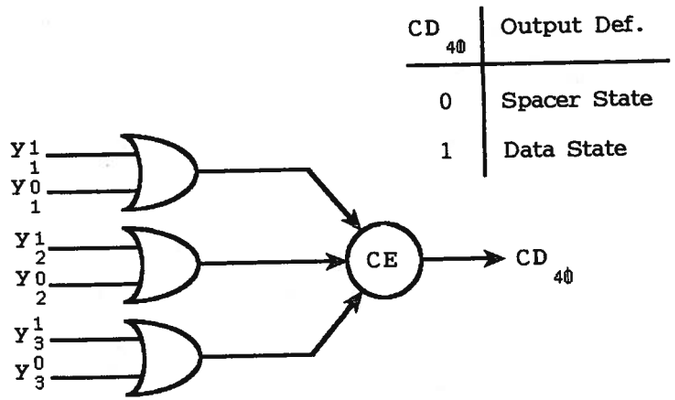

FIGURE 2: 4-Phase Completion Detection (3-variable).

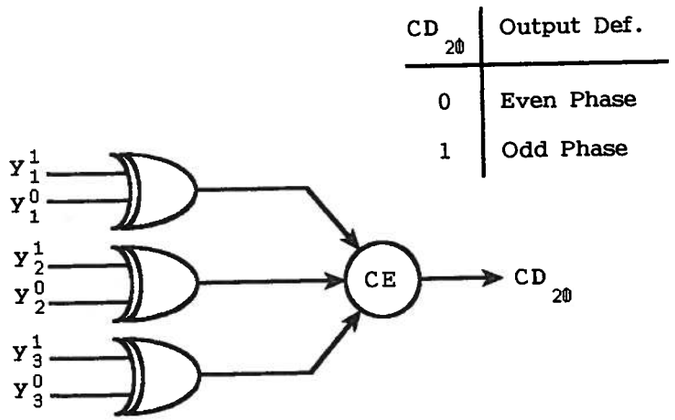

FIGURE 3: LEDR, 2-Phase Completion Detection (3-variable).

provides the true and complement of each input signal to the "AND" array. The "AND" array is constructed from a pseudo-NMOS "NOR" network. The "NOR" network generates both the Even Phase Product Terms (EPPTs) and the Odd Phase Product Terms (OPPTs). The complement of these product terms is also generated. With n input variables and 2n input wires, 2^{n+1} product terms must be generated. The product terms do not need to be generated for input variable combinations which are out of phase with one another. The "OR" array holds the value of the last output token until all the input variables have transitioned to the alternate phase. Figure 5 shows the general transistor level diagram for the even-phase product terms of the "AND" array. The transistor network required for the odd-phase product terms would have a similar arrangement except the odd-phase decodes of the input encoding pairs would be used.

The "OR" array combines the active high and active low product terms for each output function into an "OR" network of pullup and pulldown transistors. This configuration is shown in Figure 6. Each output function is composed of Active High Product Terms (AHPTs) and Active Low Product Terms (ALPTs). For a given output signal, the AHPTs correspond to the product terms which force the signal HIGH and the ALPTs correspond to the product terms which force the signal LOW. The transistor arrangement in the "OR" array also acts as a half-latch to provide strict operation of the LEDR function block. This dynamic storage is required during the time when the inputs are not all in the same phase. A weak feedback inverter or negative resistor [SEIT80] could be used to hold the output constant if the input transition time is significant as compared to the dynamic storage time of the half-latch. Each output variable requires two PLA transistor networks for generation of the LEDR encoding pair.

The PLA design style provides a systematic approach to the implementation of LEDR logic blocks. The resulting logic function will be hazard-free. Although the LEDR-PLA is significantly larger than a single-rail implementation of the same function, the physical layout is still well structured and regular. The strict design constraint causes each output function to require approximately the same inter-array loading. This causes the PLA output variables to evaluate in approximately the same delay time, independent of the input token. The use of a pseudo-NMOS "OR" network will cause the implementation to have static power dissipation. This may be a problem in low-power applications. Pre-decoding of the input variables would reduce the number of transistors required in the "AND" array, thus reducing the parasitic capacitance on each product term signal. Because the half-latch is built into the logic structure, no external latch is required in the output path. This is similar to [WILL87], but different from most

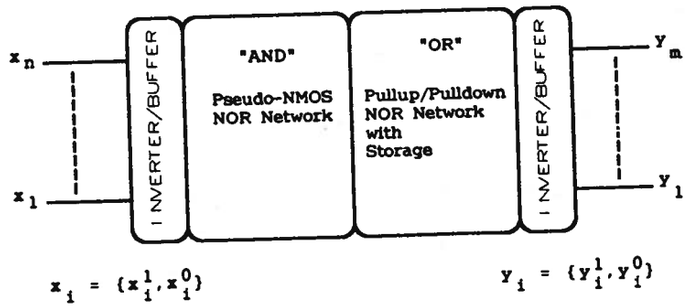

FIGURE 4: LEDR-PLA Design Style block diagram.

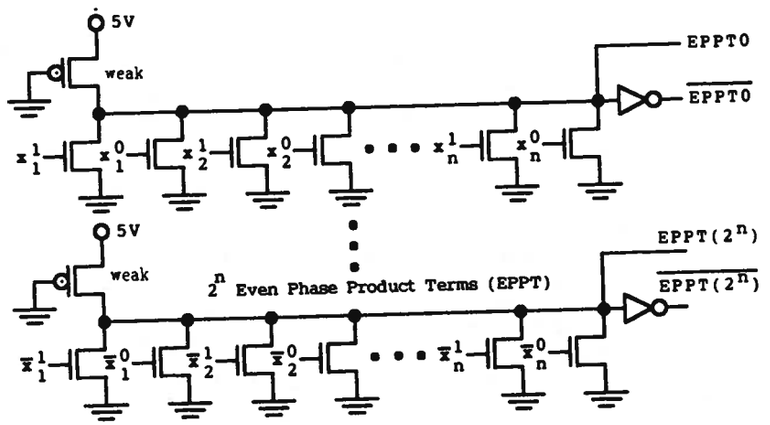

FIGURE 5: "AND" network, Even-Phase product terms.

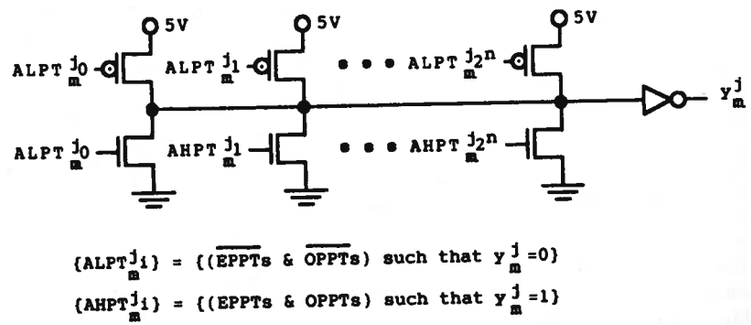

FIGURE 6: General Configuration of "OR" network.

other traditional dual-rail designs such as [JACO88] and [MENG88].

4.2 LEDR-EVAL Design Style

The LEDR-EVAL design style is a self-timed Domino logic structure with dynamic storage. Figure 7 illustrates the general logic structures used to implement the LEDR-EVAL design style. Each process logic block, P_i, consists of combinatorial logic which is activated when the $eval_i$ signal is active. The $eval_i$ and $eval\_b_i$ signals are generated by comparing the phase of the input variables with the phase of the output variables. If the input variables all have the same phase and are out of phase with the output variables, the eval control allows the process to evaluate the input token. When the output variables reach the same phase as the input variables, the eval signals are driven inactive, holding the output state until the input variables contain the next alternate phase data token. The expected processing frequency and the structure of the functional network will control whether dynamic or static storage is required. A weak feedback inverter can be used to provide static storage if required. Figure 7 shows three LEDR-EVAL structures using enable control and pre- and post-processing transmission gates to control the processing phase for the function block.

The **eval** and **eval_b** signals are generated by EXCLUSIVE-ORing and EXCLUSIVE-NORing the completion detection control signals from the previous process with the present process. Figure 8 is a logic diagram illustrating the **eval** and **eval_b** signal generation for process "i". For unequal function block delays and multiple tokens in a single pipe, an additional C-element per stage would be needed to guarantee correct operation. The generation of the evaluation control signals can add additional delay to the processing time of the function block.

The post-processing configuration shown in Figure 7 is totally speed independent, requiring no knowledge of the processing delay. If the processing time is known to be less than the completion detect plus the eval signal generation delay, the enable-control configuration shown in Figure 7 can be used. This configuration can reduce the processing overhead caused by the generation of the evaluation controls. The pre-processing configuration has the same timing constraints as the enable-control configuration.

One possible configuration not shown is when a process can be divided into two parts. The sub-process with a functional delay less than the evaluate control circuitry could be placed before the eval transmission gate while the remaining sub-process would be located after the eval gate. This

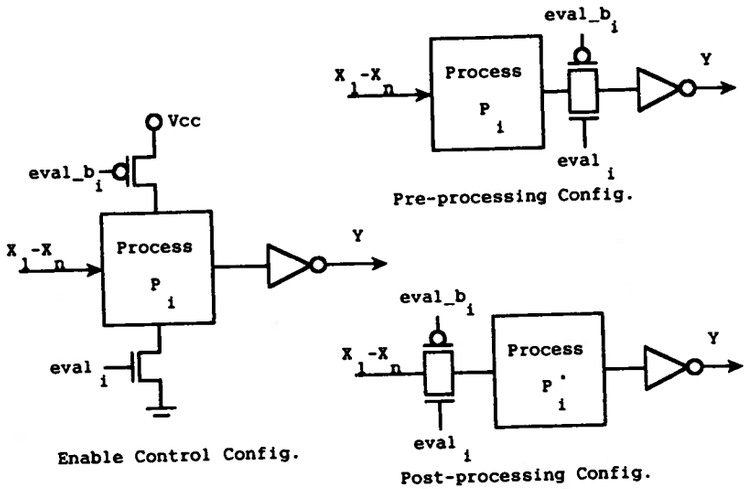

FIGURE 7: LEDR-EVAL Design Style Configurations.

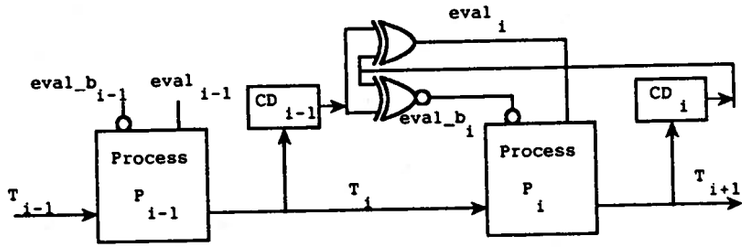

FIGURE 8: Evaluation Signal Generation

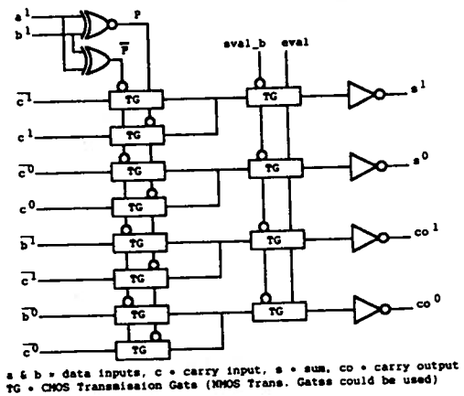

FIGURE 9: Full Adder implementation (LEDR-EVAL)

configuration may improve the total latency and throughput.

The following is a list of features and attributes of the LEDR-EVAL design style:

1. No static power dissipation.

2. Since the eval control logic guarantees strict operation of the logic function, fewer transistors or gates are required to realize the desired Boolean function. The gate/transistor level implementation of the Boolean function need not be hazard-free.

3. The pre-processing and enable control configurations allow some of the process delay to be overlapped with the eval control delay. Knowledge about function block delays is required to guarantee a correct logic implementation.

4. Care must be taken to avoid output corruption due to charge sharing.

5. In a multi-token, pipelined structure the network latency may be greater than and throughput may be less than equivalent LEDR implementations using one of the other design styles. The amount of concurrency is reduced due to the interlocks inherent in the LEDR-EVAL design style.

6. The use of dynamic storage elements implies the token repetition rate must be greater than the minimum storage refresh rate. Token data may also be maintained by toggling the phase of a given token without changing its state. This assumes the process does not maintain internal state depending on the number of tokens passing through it.

Figure 9 shows a transmission gate full adder implemented in the LEDR-EVAL design style. CMOS transmission gates are shown but NMOS transmission gates could be used and the output inverters ratioed to handle the degraded input high level, $V_{EVAL(H)}$-V_T.

4.3 LEDR-SS (Series Stack) Design Style

The last design style for LEDR logic implementations uses a series stack of transistors to provide a full decode of the active high and active low terms for each output signal. The active low terms are decoded using PMOS transistors and the active high terms are built from NMOS transistors. A full decode is required for all logic states if strict LEDR operation is required. The LEDR-SS design style is best suited for logic networks which can be built with an eager design style,

allowing the series stack decoders to be minimized. Because of feed-forward branches which may exist in some logic implementations, an eager design style may have transitioning logic blocks even though the output variables are stable. Care must be taken to guarantee all logic blocks within a process are stable before the next input token arrives.

Figure 10 is a LEDR-SS (eager) implementation of an "AND" function. Because of the amount of parasitic capacitance created by the series stack, a MOS only configuration may not be feasible. The use of BiCMOS transistor structures or buffers should make this design style more feasible.

5 PIPELINED STRUCTURES

Figures 11 and 12 illustrates how pipelined structures could be implemented using either the LEDR-SS or LEDR-PLA strict design styles to implement the function blocks. These design styles have less communication overhead than the LEDR-EVAL pipeline structure (not shown). Figure 11 shows a pipeline construction without latches between function blocks. Function block FB_i holds its output valid until FB_{i+1} has completed its evaluation of that data. The true and complement outputs of completion detector CD_{i+1} are used as inputs to FB_i. These signals are used as an extra input to the function block which allows the function block to selectively evaluate even or odd phase input signals. The CD outputs will select the opposite phase from the present data phase at the function block's inputs.

Figure 12 adds identity function latches between each function block. The identity function latches are storage elements which selectively store even or odd phase data. The **select** input controls the phase of the data stored. The **select** input for latch L_i comes from completion detector CD_{i+1}. The **select** input for function block FB_i is now the output of completion detector for latch L_i, CDL_{i+1}. This configuration allows data hold states to reside in the latches, freeing the function blocks for evaluation states. Figure 13 shows the structure of the identity function latch for wire "0" of the encoding pair. The identity function latch for wire "1" of an encoding pair has a similar configuration.

6 PERFORMANCE EVALUATION

We used **Dependency Graphs** [WILL90] to determine the throughput of each pipeline structure. Dependency Graphs represent communication dependencies between function blocks within a pipeline structure. They allow the maximum loop delays, which limit the throughput of a pipeline, to be readily identified and calculated. Table 3 lists the throughput and

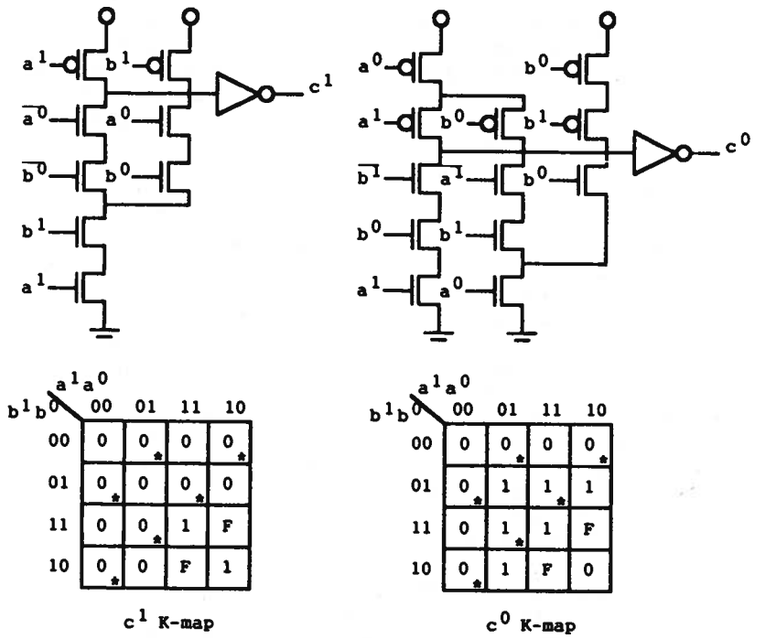

FIGURE 10: 2-Input "AND" using LEDR-SS eager Design Style.

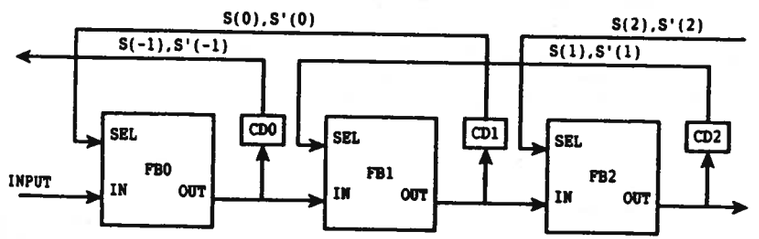

FIGURE 11: PTDR0 Pipeline Structure for LEDR logic blocks.

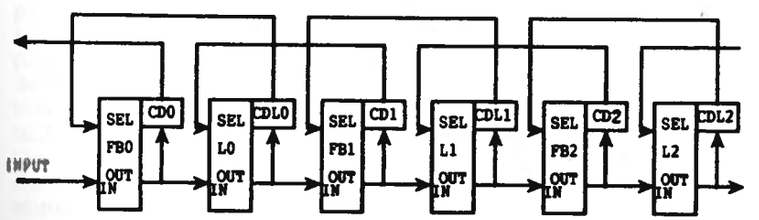

FIGURE 12: PTDR1 Pipeline Structure for LEDR logic blocks.

latency formulas per data token for the LEDR pipeline structures. PTDR0 corresponds to the LEDR (SS and PLA) pipeline without identity latches and PTDR1 corresponds to the LEDR (SS and PLA) pipeline with identity latches. PEVL0 and PEVL1 correspond to an LEDR-EVAL pipeline with and without identity latches. For comparison, Table 3 also includes the equivalent formulas for the 4-phase dual-rail pipeline structures as calculated in [WILL90]. PS0 is the 4-phase equivalent to PTDR0 and PS1 is the 4-phase equivalent to PTDR1.

Assuming the LEDR function block delay, t_F, is equal to the 4-phase function block delay, t_{FE} and the 4-phase delays, t_{FR} and t_{FE} are equal, the PTDR0 structure shows 2x throughput improvement when compared to PS0. Intuitively, this can be explained since a function block delay and a reset delay are required for each data token computation in the 4-phase pipe. The LEDR pipeline only requires a function block delay for each token. The equations show that because explicit latches are not used in these structures one function block must hold its output valid while the next function block is executing. This is accounted for in the $2t_F$ factor in the PTDR0 formula and the $3t_{FE} + t_{FR}$ factor in the PS0 formula. If $t_{FE} >> t_{FR}$ then the throughput improvement of the PTDR0 structure approaches 1.5x.

The throughput of the PTDR1 structure is twice that of its 4-phase equivalent, PS1, for equal function block delays, $t_F = t_{FE}$. This also assumes that the latch delays are equivalent. The addition of latches allow all function blocks within the pipeline to be executing, increasing the overall throughput. Also note that the reset delays in the 4-phase pipeline, PS1, are not in the worse case control path.

The latency of each structure is equivalent in both PTDRx and PSx configurations assuming $t_F = t_{FE}$.

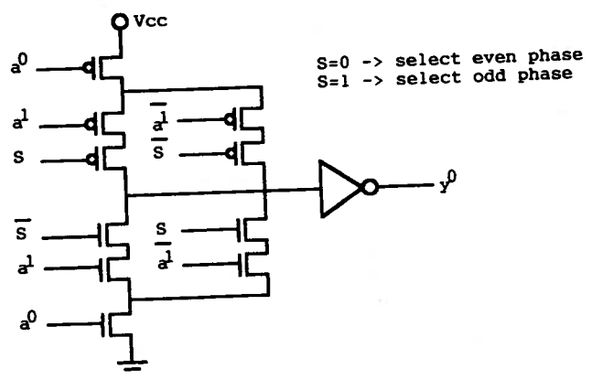

FIGURE 13: Identity Function Latch for LEDR Networks.

Pipeline Structure	Throughput	Latency
PTDR0	$1/(2t_F + t_D)$	t_F
PS0	$1/(3t_{FE} + t_{FR} + 2t_D)$	t_{FE}
PEVL0	$1/(2t_F + 2t_D + 2t_X + 4t_C)$	$t_F+t_D+t_X+t_C$
PTDR1	$1/(t_F + t_C + t_D)$	t_F+t_C
PS1	$1/(2t_{FE} + 2t_C + 2t_D)$	$t_{FE}+t_C$
PEVL1	$1/(t_F + 2t_D + 2t_X + 5t_C)$	$t_F+2t_D+2t_X+3t_C$

Legend:
t_F - LEDR function block processing delay
t_{FE} - 4-phase function block processing delay
t_{FR} - 4-phase function block reset delay (spacer token)
t_D - completion detector delay
t_C - LEDR identity or 4-phase C-latch delay
t_X - XOR delay
PTDR0 - LEDR (SS and PLA) pipeline without explicit latches
PTDR1 - LEDR (SS and PLA) pipeline with explicit latches
PEVL0 - LEDR-EVAL pipeline without explicit latches
PEVL1 - LEDR-EVAL pipeline with explicit latches
PS0 - 4-phase pipeline structure without explicit latches
PS1 - 4-phase pipeline structure with explicit latches

Table 3: 2-phase and 4-phase pipeline throughput and latency formulas.

7 CONCLUSION

LEDR signalling is an alternative to 4-phase dual-rail signalling for self-timed logic networks, which can increase throughput from 1.5x to 2x in pipelined networks by eliminating the need for spacer tokens between data tokens. The increase in throughput assumes equal function block, C-element, and latch delays between the LEDR and 4-phase implementations. When compared to transition signalling methods, LEDR provides a simpler means of implementing logic functions.

We have proposed several design styles, allowing the implementation to be tailored for specific design goals. The proposed design styles do require more transistors per function block than traditional dual-rail approaches, due to its two phase encoding scheme. Further research must be done to develop alternate design styles to reduce the number of transistors required per logic block. Implementation in other technologies (BiCMOS, ECL, etc.) may provide alternatives to the design approaches used in the CMOS implementation described. Possible logic synthesis methods must also be studied to make LEDR implementations more feasible.

REFERENCES

[ANAN86] Anantharaman T.S., "A Delay Insensitive Regular Expression Recognizer," IEEE VLSI Technical Bulletin, September 86.

[DG85] Dobberpuhl D.W., Glasser L.A., The Design and Analysis of VLSI Circuits, Addison-Wesley, 1985.

[DGY89] David I., Ginosar R., Yoeli M., "An Efficient Implementation of Boolean Functions As Self-Timed Circuits," Technion and Israel Institute of Technology, 1989.

[JACO88] Jacobs G., Brodersen R., "Self-timed Integrated Circuits for Digital Signal Processing Applications," Proceedings of Third Workshop on VLSI Signal Processing, Monterey, California, September 1988.

[MENG88] Meng T., "Asynchronous Design for Programmable Digital Signal Processors," Ph.D. Thesis, UC Berkeley, 1988.

[SEIT80] Seitz C., "System Timing," Chapter 7 in Introduction to VLSI Systems, eds. Mead C. & Conway L., Addison-Wesley, 1980.

[SING81] Singh N.P., "A Design Methodology for Self-Timed Systems," M.Sc. Thesis, MIT Laboratory for Computer Science Technical Report TR-258, MIT, Cambridge, Mass., February 1981.

[SUTH89] Sutherland I., "Micropipelines," Communications of the ACM, vol. 32 no. 6, July 1989.

[WE85] Weste N., Eshraghian K., Principles of CMOS VLSI Design, A Systems Perspective, Addison-Wesley, 1985.

[WILL87] Williams T., Horowitz M., et.al., "A Self-Timed Chip for Division," Advanced Research in VLSI, Proceedings of the Stanford Conference, pp. 75-96, March 1987.

[WILL90] Williams T.E., "Latency and Throughput Tradeoffs in Self-Timed Asynchronous Pipelines and Rings," Computer Systems Laboratory Technical Report CSL 90-431, Stanford University, Stanford, California, May 1990.

Performance Analysis and Optimization of Asynchronous Circuits

Steven M. Burns and Alain J. Martin
Computer Science Department
California Institute of Technology
Pasadena, CA 91125 USA
{steveb,alain}@vlsi.cs.caltech.edu

Abstract

We present a method for analyzing the time performance of asynchronous circuits, in particular, those derived by program transformation from concurrent programs using the synthesis approach developed by the second author. The analysis method produces a performance metric (related to the time needed to perform an operation) in terms of the primitive gate delays of the circuit. Such a metric provides a quantitative means by which to compare competing designs. Because the gate delays are functions of transistor sizes, the performance metric can be optimized with respect to these sizes. For a large class of asynchronous circuits—including those produced by using our synthesis method—these techniques produce the global optimum of the performance metric. A CAD tool has been implemented to perform this optimization.

1 Introduction

Performance analysis of a synchronous computer system is simplified by an external clock that partitions the events in the system into discrete segments. In asynchronous systems, no such quantization exists. Instead, the operation of the system proceeds at a rate determined by the speed of its individual components, and the sequencing of the operation of the components. Unlike the synchronous case, the time needed to perform an asynchronous computation cannot be determined by merely counting the number of clock cycles required and multiplying by the clock period. Instead, to determine the time required to perform the computation as a whole, the times of those individual components of the computation that must occur sequentially are summed.

The techniques required to analyze asynchronous systems resemble those used to determine the clock period of a synchronous system, that is, summing the delays along the longest path through the combinational logic connecting adjacent latches. In the clocked case, the critical path has a clear beginning and a clear end because all paths are broken by latches. No clear separation is available in asynchronous systems. Analysis procedures must deal directly