Asynchronous Datapaths and the Design of an Asynchronous Adder

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

June 1991/October 1991

Abstract

This paper presents a general method for designing delay insensitive datapath circuits. Its emphasis is on the formal derivation of a circuit from its specification. We discuss the properties required in a code that is used to transmit data asynchronously, and we introduce such a code. We introduce a general method (in the form of a theorem) for distributing the evaluation of a function over a number of concurrent cells. This method requires that the code be "distributive." We apply the method to the familiar example of a ripple-carry adder, and we give a CMOS implementation of the adder.

1 Introduction

A circuit is said to be delay-insensitive when its correct operation is independent of the delays in the operators and in the wires, except that these delays are positive and finite. Obviously, such circuits do not use clocks for the sequencing of actions, and are therefore a special class of asynchronous circuits. Delay-insensitive circuits are interesting for two main reasons: First, they are more robust and potentially faster than their clocked counterparts, since their correct operation does not rely on worst-case delay assumptions. (The speed advantage will be clearly demonstrated by the ripple-carry adder example, where we exploit the

variation in carry-chain lengths to reduce an algorithm that is linear in the worst-case assumption, to an algorithm that is logarithmic in the average case.)

Second, delay-insensitive circuits are more suitable for formal treatment since they can be designed and analyzed entirely within the algorithmic domain, up to electrical optimizations like transistor sizing. A delay-insensitive circuit can be formally derived by program-transformation from a "high-level" program description. If the original program has been proven correct, the resulting circuit will be correct by construction. (For a description of the method, see, for instance, [4], and [5].)

In spite of the intense activity in the area of high-level synthesis of delay-insensitive circuits, most published research so far has concentrated on the design of "control circuits," i.e., circuits that realize the sequencing of actions of a computation. The other type of circuits, called "data paths," are those that deal with the manipulation and transmission of data.

Datapath design raises issues very different from, and in several respects more difficult than, that of control circuitry. First, for reasons of efficiency, all sequencing circuitry should be eliminated from the datapath implementation. Second, the evaluation of a function should be distributed. Ideally, we want each bit of the output to be produced by a "cell" that depends only on a limited number of bits of input. All cells operate concurrently.

This paper presents a general method for designing delay-insensitive datapath circuits. Its emphasis is on the formal derivation of a circuit from its specification. We first discuss the properties required in codes used to transmit data asynchronously between two concurrent processes, and we introduce one of these codes. We then introduce a general method (in the form of a theorem) for distributing the evaluation of a function over a number of concurrent cells; this method requires that the code be "distributive."

Next, we apply the method to the familiar example of a ripple-carry adder. This example uncovers another difficulty of datapath design: In order to reduce the fanin of each cell, some information computed by one cell is used in another cell—the carry in the case of an adder. But this extra communication may reduce the concurrency between cells.

We therefore introduce and apply optimization rules that reduce the dependencies between input and output.

Finally, we show how a monotonicity property of guard evaluation (called "stability") makes it possible to implement the final program directly as a transistor network in CMOS. This mapping is particularly efficient since, unlike earlier implementations of the adder, it does not require translation to standard cells.

The paper is reasonably self-contained: The whole design of the adder from program to CMOS circuit is explained and justified.

2 Delay-insensitive Communication

Consider a system consisting of two communicating processes: a producer/sender of data words, and a consumer/receiver of the data words. The data words are binary encoded and transmitted on a set of wires. For the purpose of making this paper self-contained, we view a wire shared by two processes as being a boolean variable assigned by one process and read by the other process. There is an important restriction, however, to the use of wires as program variables: Because of the delay-insensitive nature of the transmission of data, the order in which the wires of a set are assigned by the sender cannot be maintained on the receiver side; they can be observed by the receiver to change value in any order.

Because signals (assignments to wires) cannot be ordered, it is impossible to use an extra signal—clock or control signal—to encode the information (to be used by the the receiver) that the set of data wires contains a valid value. Instead, this information has to be encoded in the data that is transmitted between sender and receiver.

2.1 Delay-insensitive codes

Let us discuss first the transmission of one data word. The sender assigns values to all the wires concurrently, since order is irrelevant. The receiver reads the data wires in any order or concurrently. Concurrent reading and writing of a wire is possible: We may assume without loss of generality that the value read is either the old or the new value. Concurrent writes are not allowed.

Let $\mathcal{B}$ be the set of data words to be transmitted. A data value to be transmitted is encoded using the coding function $\mathcal{C} : \mathcal{B} \mapsto \mathcal{X}$. Set $\mathcal{X}$ is the set of all code words. Let $\mathcal{V}$ be the set $\mathcal{C}(\mathcal{B})$. $\mathcal{V}$ is called the *valid* set (or the set of *valid values*).

The code has to be chosen such that there is a *non-empty* set, $\mathcal{N}$, the *neutral* set (or the set of *neutral values*), such that $\mathcal{N} \subseteq \mathcal{X} - \mathcal{V}$.

Hence, a code value cannot be both neutral and valid. For a code word X, the predicate $v(X)$ stands for "X is a valid code word." The predicate $n(X)$ stands for "X is a neutral code word." The code has to be chosen such that:

Property 1 *For any code word* X *:* $\neg v(X) \vee \neg n(X)$.

Furthermore, $\mid \mathcal{X} \mid > \mid \mathcal{B} \mid$. (Typically, each data word is an array of n booleans, and each code word is an array of m booleans, with $m > n$.)

The transmission of a data word, B, by the sender is the assignment of a valid code word, X, to the set of wires such that $\mathcal{C}(B) = X$. If the assignment also implies that the wires change from a neutral value to a valid value, we can construct a communication protocol in which the receiver can detect that the value read on the wires is the data sent by observing a change from a neutral value to a valid value.

Once a valid value has been assigned to the wires, sending the next code word requires either that all wires first be reset to a neutral value or that the coding function, $\mathcal{C}$, be changed such that the final, valid, value of any communication can be interpreted as the initial, neutral value of the next communication.

The first solution is a straightforward extension of the four-phase handshake protocol; the second solution is a straightforward extension of the two-phase handshake protocol. Since we usually prefer to use a four-phase protocol, we choose the first solution in this paper. The extended four-phase protocol between the producer and the consumer can be described as follows:

$$\begin{aligned} \text{producer} &\equiv *[\ [ci];\ \text{produce } X;\ X\Uparrow;\ [\neg ci];\ X\Downarrow\ ] \\ \text{consumer} &\equiv *[\ [n(X)];\ ci\uparrow;\ [v(X)];\ \text{consume } X;\ ci\downarrow\ ] \end{aligned}$$

Initially, $\neg ci \wedge n(X)$ holds.

The general notation used is explained in the appendix. $X \Uparrow$ is the concurrent assignment of some bits of X such that the result is a valid value, and $X \Downarrow$ is the concurrent assignment of some bits of X such that the result is a neutral value.

In the consumer, the test $[v(X)]$ is needed to guarantee that the consumed value is a valid value, and the test $[n(X)]$ is needed to guarantee that the next valid value produced by the producer is separated from the previous one by a neutral value.

3 Separable Codes

Because the assignments to the wires used to communicate a code word are concurrent, any transition from a neutral value to a valid value or from a valid value to a neutral value can go through a number of intermediate values. When executing the waits $[v(X)]$ and $[n(X)]$, the receiver can read several intermediate values for X, i.e., values that are obtained by changing only some of the wires of X. To avoid premature completion of the waits, we must ensure that none of the intermediate values generated during a transition from neutral to valid is valid, and that none of the intermediate values generated during a transition from valid to neutral is neutral. A code with this property is said to be *separable*.

3.1 Intermediate Values

We require that assignments $X \Uparrow$ and $X \Downarrow$ each contain at most one assignment to each boolean variable x of X (an "elementary assignment"). Since any valid value is distinct from any neutral value, the assignment $X \Uparrow$, which realizes the transition from a neutral value Xn to a valid value Xv, contains at least one elementary assignment. If $X \Uparrow$ contains more than one elementary assignment, the set of elementary assignments of $X \Uparrow$ can be partitioned into two non-empty subsets, $S1$ and $S2$. The set $S1$ realizes a transition from Xn to a value Z, called an *upward intermediate* value.

Similarly, $X \Downarrow$, which realizes the transition from a valid value to a neutral value, contains at least one elementary assignment, and we define *downward intermediate* values in the same way as upward inter-

mediate values.

We require that the following property hold:

Property 2 (Separable code) *A code is separable if no upward intermediate value is valid, and no downward intermediate value is neutral.*

Obviously, if a code contains exactly one neutral value, no downward intermediate value is neutral.

3.2 Dual-Rail Code

A simple code that satisfies both Property 1 and Property 2 is the so-called *dual-rail* code[7]. To each bit, b_k, of a word, B, correspond two bits, xt_k and xf_k, of the code word, X, encoding B. We define:

$$\begin{aligned} n(X) &\stackrel{\text{def}}{=} (\forall k : 0..N-1 : \neg xt_k \wedge \neg xf_k) \\ v(X) &\stackrel{\text{def}}{=} (\forall k : 0..N-1 : xt_k \neq xf_k) \quad . \end{aligned}$$

Hence, the neutral value is unique. The coding of a value word, B, as a valid code word, X, is simply:

$$\forall k : 0..N-1 : xt_k, xf_k := b_k, \neg b_k \quad . \tag{1}$$

Observe that a code word, X, for which $xt_k \wedge xf_k$ holds for some value of k, is neither valid nor neutral, and is therefore not in $\mathcal{X}$.

Proof of Property 1 By the definition of $n(X)$ and $v(X)$, we have:

$$n(X) \Rightarrow \neg v(X)$$

which establishes Property 1. □

Proof of Property 2 Since the code contains only one neutral value, no downward intermediate value is neutral.

We prove that an upward intermediate word, Z, is not valid. Because of the coding (1), any valid dual-rail code word differs from the neutral word in exactly N bit positions. By definition, Z differs from the neutral value in a number, m, of bit positions equal to the size of $S1$. Hence, $m < N$, and Z is not valid. □

3.3 One-Hot Code

Another commonly used delay-insensitive code is the so-called *one-hot* code. For a data word B of n bits, the one-hot code X is the word of 2^n bits with exactly one bit true in the position corresponding to the decimal value of B. We have:

$$\begin{array}{lll} n(X) & \stackrel{\text{def}}{=} & (\forall k : 0..2^n - 1 : \neg x_k) \\ v(X) & \stackrel{\text{def}}{=} & ((\mathrm{N}k : 0..2^n - 1 : x_k) = 1) \quad . \end{array}$$

Since $X \Uparrow$ and $X \Downarrow$ both contain exactly one elementary assignment, no intermediate value can be generated; thus, the one-hot code is separable.

4 Function Evaluation

We want to construct a process, F, that repeatedly takes separable code word, X, and produces a separable code word, Y, such that $Y = f(X)$ for a given function f. The process behaves as both the consumer of argument X and the producer of the result Y. Combining the two protocols gives the function-evaluation process

$$F \equiv *[\ [v(X)]; Y\Uparrow; [n(X)]; Y\Downarrow\ ] \quad , \tag{2}$$

such that $v(Y) \wedge (Y = f(X))$ holds as a postcondition of $Y\Uparrow$, and $n(Y)$ holds as a postcondition of $Y\Downarrow$.

The environment behaves as the producer of X and the consumer of Y, and fulfills the protocol

$$E \equiv *[\ \text{produce } X; [n(Y)]; X\Uparrow; [v(Y)]; X\Downarrow; \text{consume } Y\ ] \quad , \tag{3}$$

such that $v(X)$ holds as a postcondition of $X\Uparrow$, and $n(X)$ holds as a postcondition of $X\Downarrow$.

The initial conditions are $n(X)$ and $n(Y)$.

If X and Y are dual-rail code words, each pair yt_k, yf_k of bits of Y is assigned by the two commands $Y_k\Uparrow$ and $Y_k\Downarrow$, with

$$\begin{array}{lll} Y_k\Uparrow & \equiv & [Bt_k \rightarrow yt_k\uparrow \,[]\, Bf_k \rightarrow yf_k\uparrow] \\ Y_k\Downarrow & \equiv & yt_k\downarrow \| \; yf_k\downarrow \quad , \end{array}$$

where Bt_k and Bf_k are two boolean expressions that depend on a subset, X_k, of the bits of X.

5 Distributive Codes

Essential to the introduction of concurrency in the implementation of process F is the ability to distribute the global tests, $v(X)$ and $n(X)$. Codes with this property are called *distributive.*

A *subcode* of code word X is a word formed from a proper subset of the set of bits of X.

Definition 1 (Distributive Code) *A code is distributive if any code word, X, can be partitioned into a set, S, of subcodes such that*

- *$n(Y)$ and $v(Y)$ are defined for any Y, $Y \in S$,*
-

$$(\forall Y : Y \in S : n(Y)) = n(X) \tag{4}$$

$$(\forall Y : Y \in S : v(Y)) = v(X) \tag{5}$$

Theorem 1 *The dual-rail code is distributive.*

Proof We construct a set, S, of subcodes, $X_0, X_1, \ldots X_{p-1}$, of code word X as follows. First, each subcode contains any number (larger than 0 and less than N) of pairs, xt_k, xf_k. Hence, any such subcode, X_j, is itself the dual-rail code of the subset of $\mathcal{B}$ consisting of the bits of $\mathcal{B}$ with the same indices as the pairs in X_j, and thus, $n(X_j)$ and $v(X_j)$ are defined.

Secondly, S is chosen such that $(\bigcup k : 0..p-1 : W_k) = X$, and thus (4) and (5) hold. □

However, the one-hot code is *not* distributive.

6 The Main Theorem

Next, we show how to implement the function evaluation process, F, with a set of concurrent *cells,* each dedicated to assigning one bit of the function. We present the result in the form of a theorem. Although the method is applicable to all distributive codes, we prove the theorem for dual-rail codes.

We first distribute the (dual-rail) input code word X in the following way: We construct a set of N subcodes, W_k, with $0 \leq k < N$, where

N is the size (number of bits) of the data output. The construction of the code follows the two rules introduced in the proof of the previous theorem. Hence, we have

$$(\forall k :: v(W_k)) \equiv v(X) \tag{6}$$

$$(\forall k :: n(W_k)) \equiv n(X) \tag{7}$$

We add one extra requirement: Let S_k be the set of bits of X used in Bt_k and Bf_k. We require that W_k be chosen such that $S_k \subseteq W_k$.

Hence, the function evaluation can be distributed only if the algorithm used for evaluating the function satisfies the locality property that the number of bits of X used in Bt_k and Bf_k is (significantly) smaller than N.

This extra requirement ensures that the validity of W_k implies the validity of the bits of X used in Bt_k and Bf_k.

With this distribution of the input code, X, we will establish

Theorem 2 *The function evaluation process F can be implemented as the parallel composition of N function-cells C_k, where C_k is the program:*

$$\begin{array}{l}(*[[Bt_k \wedge v(W_k) \rightarrow yt_k\uparrow]] \\ \|*[[Bf_k \wedge v(W_k) \rightarrow yf_k\uparrow]] \\ \|*[[n(W_k) \rightarrow yt_k\downarrow \| \; yf_k\downarrow]] \\ ) \quad ,\end{array}$$

where N is the number of data bits of the output Y of F.

Proof We are going to produce the solution by successive program transformations.

The function evaluation process, F, and the environment, E, share variables in a restricted form. Process F sets the output variables, Y, and observes the input variables, X. Process E sets the input variables, X, and observes the output variables, Y. The correctness of any implementation relies on an important property of the guard evaluations, called *stability*.

Definition 2 (Stability) *Let G be a guard containing shared variables assigned by another process. The evaluation of G is stable if, once G is evaluated to true, it remains true at least until the process containing G changes some variable.*

Theorem 3 *All guards are stable in the initial version of F and in E.*

(The proof is immediate from the properties of a separable code.)

We shall maintain the stability of the guards as an invariant of all further versions of F.

We can now introduce and justify the successive transformations of F. (In the proof, the range of k is from 0 to $N-1$ and is omitted.)
Transformation 1 replaces the global waits with conjunctions of local waits:

$$*[\ [\forall k :: v(W_k)];\ Y\Uparrow;\ [\forall k :: n(W_k)];\ Y\Downarrow\ ]$$

The correctness of the transformation is immediate from (6) and (7).
Transformation 2 distributes the waits:

$$*[(\|k :: [v(W_k)]); Y\Uparrow; (\|k :: [n(W_k)]); Y\Downarrow]$$

The correctness of this transformation follows from the stability of $v(W_k)$ and $n(W_k)$ for all k in F, which follows from the stability of $v(X)$ and $n(X)$ in F, since $v(X) \Rightarrow v(W_k)$ and $n(X) \Rightarrow n(W_k)$.

In view of the next transformation, we rewrite this version as:

$$*[(\|k :: [v(W_k)]); (\|k :: Y_k\Uparrow); (\|k :: [n(W_k)]); (\|k :: Y_k\Downarrow)]$$

Transformation 3 eliminates the "global" semicolons between the concurrent waits and the following concurrent assignments:

$$*[(\|k :: [v(W_k)]; Y_k\Uparrow); (\|k :: [n(W_k)]; Y_k\Downarrow)]$$

This tranformation is justified as follows: In the new program, $Y_k\Uparrow$ can be executed before the completion of a wait action $[v(W_j)]$ for $j \neq k$; however, the assignment still follows the wait $[v(W_k)]$. Since $S_k \subseteq W_k$, the assignment depends only on the validity of variables in W_k; and since $v(W_k)$ is stable, the net effect of the assignment $Y_k\Uparrow$ is not changed by the transformation.

The net effect of the assignment, $Y_j\Uparrow$, is not changed either. This assignment depends only on the validity of the variables in the set, W_j.

Since all bits of X are assigned concurrently by the environment, if $v(X_k)$ is true in a state of F, we can conclude that eventually, $v(X_j)$

will hold; and similarly for the downgoing transitions. Hence, the assignment, $Y_j\Uparrow$, will be correctly executed in the new program.

The other half of the transformation is justified in the same way. Now, F has the structure:

$$*[(\|k :: T_k); (\|k :: T'_k)] \quad .$$

Transformation 4 eliminates the last global synchronization points, leading to the program:

$$(\|k :: *[T_k; T'_k])$$

where $*[T_k; T'_k]$ is the program of function-cell C_k.

This transformation potentially eliminates the sequencing between an action, T_k, and the following T'_j, and between an action, T'_j, and the following T_k, for $k \neq j$.

However, T'_j is conditional to $n(W_j)$ holding. And the environment establishes $n(W_j)$ as a result of $X\Downarrow$, which is conditional to $v(Y)$ holding as a postcondition of the preceding T_k for all k. Hence, the sequencing between a T_k action and the following T'_j is enforced by the environment even for $k = j$.

The other half of the proof is similar.

We have also established that the sequential composition between T_k and T'_k inside the same cell (i.e., for the same value of k) is also superfluous, which justifies the next transformation.

Transformation 5 rewrites the program of function-cell C_k as

$$(*[T_k] \parallel *[T'_k])$$

since the sequencing between a T_k action and the following T'_k is enforced by the environment.

Transformation 6: For X and Y dual-rail codes, the program of C_k is

$$\begin{array}{l}(*[\,[v(W_k)]; [Bt_k \rightarrow yt_k\uparrow \\ \qquad\qquad\quad [\!]\, Bf_k \rightarrow yf_k\uparrow \\ \quad]\qquad\quad] \\ \|*[\,[n(W_k) \rightarrow yt_k\downarrow\|\, yf_k\downarrow]] \\ ) \quad .\end{array}$$

We eliminate the last semicolon by moving the test, $[v(W_k)]$, inside the guard of the selection command. We get:

$$
\begin{array}{l}
(*[\,[Bt_k \wedge v(W_k) \rightarrow yt_k\uparrow \\
\quad [\!]\, Bf_k \wedge v(W_k) \rightarrow yf_k\uparrow \\
\quad]\,] \\
\|*[\,[n(W_k) \rightarrow yt_k\downarrow \| \; yf_k\downarrow]] \\
) \quad .
\end{array}
$$

This transformation is valid if we assume that the implementation of the guard evaluation uses the same value of X for both $v(W_k)$ and Bt_k in the first guard, and the same value of X for both $v(W_k)$ and Bf_k in the second guard. This requirement is relatively easy to meet in VLSI, but we will not elaborate any further, as we can justify the transformation in another way—thanks to a property of Bt_k and Bf_k that we will introduce for optimization purposes.

Transformation 7 replaces the selection command

$$
\begin{array}{l}
*[\,[Bt_k \wedge v(W_k) \rightarrow yt_k\uparrow \\
\quad [\!]\, Bf_k \wedge v(W_k) \rightarrow yf_k\uparrow \\
\quad]\,]
\end{array}
$$

with the parallel command

$$
\begin{array}{l}
(\,*[[Bt_k \wedge v(W_k) \rightarrow yt_k\uparrow]] \\
\|\,*[[Bf_k \wedge v(W_k) \rightarrow yf_k\uparrow]] \\
) \quad .
\end{array}
$$

The transformation is an application of

Theorem 4 *The programs* $*[[A[\!]B]]$ *and* $*[[A]] \parallel *[[B]]$ *are equivalent if and only if* A *and* B *are mutually exclusive.*

Proof It is obvious that A and B being mutually exclusive is a necessary condition for the equivalence of the two programs.

Assume that A and B are mutually exclusive. Any finite execution of either program is an interleaving of a finite number of executions of A and B. (An execution of A or B is a "step of the interleaving.") Assume that the two interleavings are identical up to and excluding the n-th step, $n \geq 0$. Since the selection command is deterministic, the nth step is unique, and is therefore identical for both interleavings. □

This completes the proof of the main theorem. □

Corollary 1 *All guards of a cell are stable.*

7 Binary Addition

As an example of an application of the method, we will now implement the process, F, whose function, f, is the addition of two N-bit integers, A and B. The output is an $N+1$-bit integer, S. We want to select an algorithm for binary addition in which the functions, Bt and Bf, as introduced in the previous sections, depend only on a few bits of A and B. "Ripple-carry addition" is such an algorithm.

7.1 Ripple-Carry Addition

The value of bit s_k of S can be expressed as a function of bits a_k and b_k of A and B, and of the carry-in bit, c_k. More precisely, the postcondition of the addition can be expressed as:

$$\neg c_0 \wedge (\forall k : 0..N-1 : sum_k) \wedge s_N = c_N \quad ,$$

where each sum_k is the conjunction of the three predicates:

$$(\neg a_k \wedge \neg b_k) \Rightarrow (s_k, c_{k+1} = c_k, false)$$

$$(a_k \wedge b_k) \Rightarrow (s_k, c_{k+1} = c_k, true)$$

$$(a_k \neq b_k) \Rightarrow (s_k, c_{k+1} = \neg c_k, c_k)$$

The computation of bit s_k of the sum requires the previous computation of carry bit c_k, and therefore also produces carry bit c_{k+1}. Hence, we are faced with a new problem: The adder-cell, add_k, for $k > 0$, requires as input the carry-in c_k produced by cell add_{k-1}.

7.2 "Magic" Inputs

First, let us assume that the carry-in bits are provided "by magic" by the environment as normal inputs, and that each cell computes its carry-out, d_k, as a normal output. We can then apply our main theorem and construct an adder as the concurrent composition of N adder-cells.

The inputs, A, B, and C, and the outputs, S and D, are dual-rail encoded: To bit a of data input A correspond bits at and af of the dual-rail code; and similarly for the other inputs and outputs. For the construction of a generic adder-cell, add, we can omit the subscript

k. The guards, Bt and Bf, of the commands that set the two output bits to true in the main theorem have to be replaced with two sets of guards, as we have two different output bits per cell.

Guards St and Sf are used to assign bits st and sf, respectively. Guards Dt and Df are used to assign bits dt and df, respectively.

We have:

$$\begin{array}{rcl} St & \equiv & (ct \wedge eq(a,b)) \vee (cf \wedge dif(a,b)) \\ Sf & \equiv & (cf \wedge eq(a,b)) \vee (ct \wedge dif(a,b)) \\ Dt & \equiv & (at \wedge bt) \vee (dif(a,b) \wedge ct) \\ Df & \equiv & (af \wedge bf) \vee (dif(a,b) \wedge cf) \end{array}$$

where

$$eq(a,b) \quad \equiv \quad (at \wedge bt) \vee (af \wedge bf)$$

and

$$dif(a,b) \quad \equiv \quad (at \wedge bf) \vee (af \wedge bt) \quad .$$

The smallest set, W, of input bits used in any of the guards is $\{a, b, c\}$. Hence, the validity test, $v(W)$, is

$$v(a) \wedge v(b) \wedge v(c)$$

with

$$v(x) = (xt \wedge \neg xf) \vee (xf \wedge \neg xt) \quad .$$

For dual-rail codes, this expression can be simplified as

$$v(x) = xt \vee xf \quad ,$$

since $xt \wedge xf$ never holds.

7.3 Eliminating the Magic

Since all input transitions are delay-insensitive, we can restrict the "magic" to producing a valid input, c_{k+1}, only after output d_k is valid, and to producing a neutral input, c_{k+1}, only after output d_k is neutral, for $0 \leq k < N-1$. The environment originally produces input, $c0$, which is false. The solution is still correct although the concurrency between cells has been restricted.

Next, we observe that since, for $k < N-1$, the valid value of d_k is the same as the valid value of c_{k+1}, we can eliminate the magic and

connect d_k with c_{k+1}. We also eliminate the d outputs, except for d_{N-1}, which is s_N. The first carry-in is no longer produced by magic but more prosaically by the program

$$
\begin{array}{l}
*[\,[at_0 \vee af_0 \rightarrow cf_0\uparrow \\
\quad []\neg at_0 \wedge \neg af_0 \rightarrow cf_0\downarrow \\
\,]\,] \quad .
\end{array}
$$

The program of a cell is:

$$
\begin{array}{l}
(*[[St \wedge v(a) \wedge v(b) \wedge v(c) \rightarrow st\uparrow]] \\
\|*[[Sf \wedge v(a) \wedge v(b) \wedge v(c) \rightarrow sf\uparrow]] \\
\|*[[Dt \wedge v(a) \wedge v(b) \wedge v(c) \rightarrow dt\uparrow]] \\
\|*[[Df \wedge v(a) \wedge v(b) \wedge v(c) \rightarrow df\uparrow]] \\
\|*[[n(a) \wedge n(b) \wedge n(c) \rightarrow st\downarrow \| sf\downarrow \| dt\downarrow \| df\downarrow]] \\
) \quad .
\end{array}
$$

The solution obtained is completely sequential since the validity of s_k depends on the validity of the carry-in, c_k. The solution can be greatly improved by reducing these dependencies and simplifying the guards.

8 Optimization

An important property of the dual-rail code is that the tests $v(W_k)$ can be simplified and often even eliminated. Simplifying or eliminating these tests may eliminate some of the sequential dependencies between the validity of an input and the validity of an output, hence reducing the number of steps required to compute the function in the average case.

We will also simplify the remaining expressions. These transformations will reduce the number of conjuncts in boolean expressions, hence reducing the number of transistors in series in a pullup or pulldown chain of a CMOS inplementation. (In the worst case, the switching delay is quadratic with the number of transistors in series.)

8.1 Simplifying the Validity Conditions

The validity tests, $v(W_k)$, can be simplified by application of

Theorem 5 *Let B be a guard Bt or Bf of a cell. For B in disjunctive-normal form (sum-of-products), let T be a term of B, i.e. $B \equiv T \vee B'$, and T be a conjunction. If W_t is the set of input booleans used in T, we have, for dual-rail encoded inputs*

$$T \Rightarrow v(W_T) \quad .$$

Proof B is derived from a condition Bd on the data words, also in disjunctive normal form, by applying the dual-rail coding assignment (1): For each data input x in Bd, all literals, x, are replaced with x_t and all literals, $\neg x$, are replaced with x_f. Hence, if term T of B contains either x_t or x_f, we have $T \Rightarrow x_t \vee x_f$. □

In other words, if a guard Bt or Bf of a cell is true, the inputs used to established the truth of the guard are valid and thus the guard is stable. (As was suggested earlier, Transformation 6 can be justified by means of this property of dual-rail codes.)

8.2 Validity of Transient Inputs

Although all guards Bt or Bf of a cell are stable, we cannot always eliminate the validity tests altogether, because of the possible existence of so-called *transient inputs*.

It may occur that, for some value of the inputs, some input bits are not used to establish the validity of the output Y, and therefore the function-evaluation process can complete the handshake protocol without waiting for these input bits to be valid. However, we have to see to it that those input bits still go through the valid/neutral cycle before they are used in a subsequent function evaluation. Such input bits are called *transient* inputs. Let us look at a simple example.

The function to be implemented is the AND-function:

$$[a \wedge b \rightarrow y\uparrow \,[\!]\, \neg a \vee \neg b \rightarrow y\downarrow] \quad .$$

The dual-rail translation of this program gives:

$$[at \wedge bt \rightarrow yt\uparrow \,[\!]\, af \vee bf \rightarrow yf\uparrow] \quad .$$

We observe that because of the disjunction in the second guard, both a and b are transient inputs. Hence, the second guard has to include the validity test for the transient inputs.

8.3 Simplification of the Adder

We first eliminate the validity tests from the guards. We then simplify Dt and Df. Finally, we check that there is no transient input in the new guards. We leave it to the reader to verify that Dt

$$(at \wedge bt) \vee (dif(a,b) \wedge ct) \quad ,$$

can be simplified as:

$$(at \wedge bt) \vee ((at \vee bt) \wedge ct) \quad .$$

Df can be simplified similarly as

$$(af \wedge bf) \vee ((af \vee bf) \wedge cf) \quad .$$

We cannot simplify the expressions for St and Sf. With this new set of guards, we check that all inputs are used in St and Sf, i.e.,

$$st \vee sf \Rightarrow v(a) \wedge v(b) \wedge v(c)$$

holds, and thus there is no transient input.

8.4 A Graphical Analysis

A graphical analysis can be helpful in identifying the transient inputs and at the same time in evaluating the efficiency of the algorithm. In the case of the adder, we construct the following graph: To each cell correspond four nodes in the graph—one for input c, one for inputs a and b together, one for output s, and one for output c.

A solid arrow from node x to node y means that the validity of y is established by a command with guard G such that $G \Rightarrow v(x)$. The dotted arrow from c to d indicates that the validity of d is established by a command with guard G' such that $G' \Rightarrow v(c)$ only for certain inputs, namely when $a \neq b$. Figure 1 shows the dependency graph for three cells.

The graph shows that the validity of a, b, and c is required for s and d to be valid. Each directed path from an input to an output indicates that the validity of each node but the last one on the path is required for the next node on the path to be valid. Hence, the length of the

longest path gives an upper bound of the number of steps necessary to compute the outputs.

An inspection of the graph shows that the longest path is proportional to the largest number of contiguous cells with the arrow from c to d—the dotted arrow—present. Hence *The number of steps required to compute the output of the ripple-carry adder is proportional to the maximal number of contiguous binary positions in which one input bit is different from the other.*

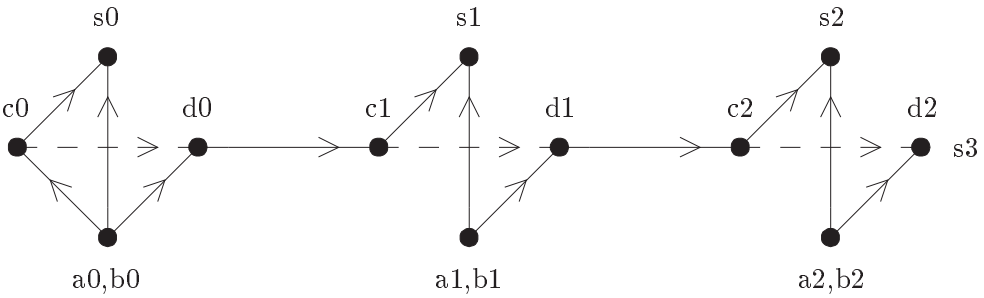

Figure 1: The validity graph for three cells

8.5 Distributing the test $n(W)$

The assignments of $Y \Downarrow$ are unconditional: All variables of Y are reset to the neutral value. We can therefore distribute the test $n(X)$ in any way we want. In the case of the adder, we can split the test $n(W)$ into $n(a) \wedge n(b)$ on the one hand, and $n(c)$ on the other hand. We can associate either guard with the transitions $st\downarrow, sf\downarrow$ or $dt\downarrow, df\downarrow$. The two choices are expressed in the dependency graphs of Figure 2, in which an arrow from x to y means that the neutrality of y depends on the neutrality of x. It is clear that the solution of Figure 2(a) is more efficient since all paths have constant length. This choice corresponds to the guarded commands:

$$\begin{aligned} \neg at \wedge \neg af \wedge \neg bt \wedge \neg bf &\rightarrow dt\downarrow \| \; df\downarrow \\ \neg ct \wedge \neg cf &\rightarrow st\downarrow \| \; sf\downarrow \quad . \end{aligned}$$

The final program of a cell is

$$
\begin{array}{ll}
(*[[(ct \wedge eq(a,b)) \vee (cf \wedge dif(a,b)) & \rightarrow st\uparrow]] \\
\|*[[(cf \wedge eq(a,b)) \vee (ct \wedge dif(a,b)) & \rightarrow sf\uparrow]] \\
\|*[[(at \wedge bt) \vee ((at \vee bt) \wedge ct) & \rightarrow dt\uparrow]] \\
\|*[[(af \wedge bf) \vee ((af \vee bf) \wedge cf) & \rightarrow df\uparrow]] \\
\|*[[\neg ct \wedge \neg cf & \rightarrow st\downarrow \| \; sf\downarrow]] \\
\|*[[\neg at \wedge \neg af \wedge \neg bt \wedge \neg bf & \rightarrow dt\downarrow \| \; df\downarrow]] \\
) \quad .
\end{array}
$$

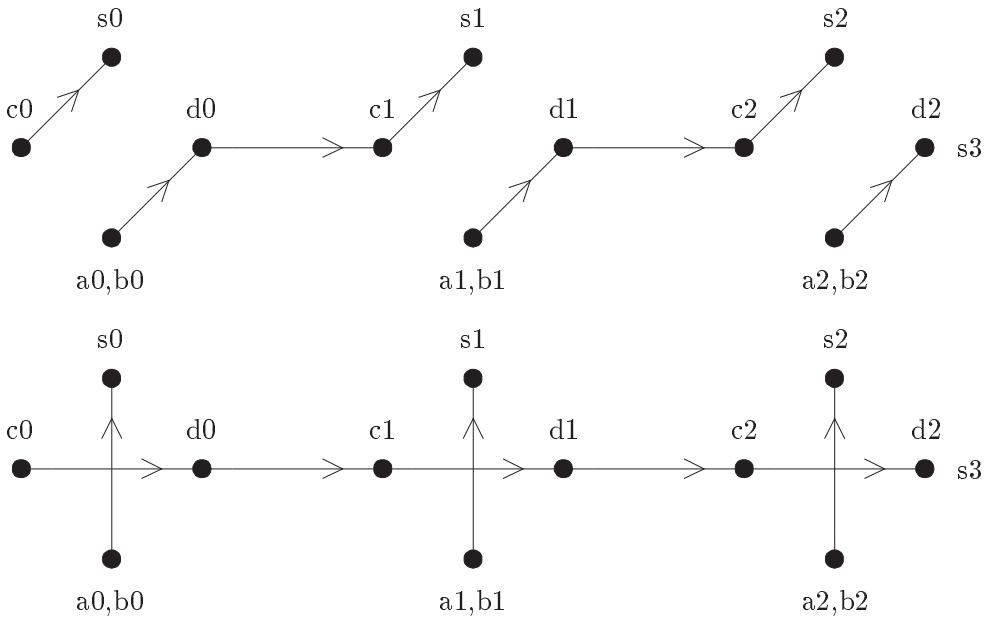

Figure 2: Two ways to distribute the neutrality test

9 CMOS Implementation

A program $*[[B \rightarrow x\uparrow]]$ or $*[[B \rightarrow x\downarrow]]$, with B stable can be implemented directly in CMOS. (We call such a program a *production rule*.) Hence, the whole adder can be implemented directly in CMOS without further transformation into "standard cells."

To the expression, B, corresponds a series-parallel switching network, $N(B)$. Each switch is implemented with an n-transistor or a p-transistor whose gate is a literal of B. Hence, the predicate, *there is a conducting path between the two terminal nodes of* $N(B)$, has the same value as B. We limit ourselves to two types of switching networks: A "pullup" circuit has for terminal nodes the high-voltage constant, *VDD*, and the output node, x, of the program. A "pulldown" circuit has for terminal nodes the low-voltage constant, *GND,* and the output node, x, of the program. Hence, a pullup circuit implements the program $*[[B \rightarrow x\uparrow]]$, and a pulldown circuit implements the program $*[[B \rightarrow x\downarrow]]$.

For reasons of efficiency particular to the CMOS technology, we restrict a pullup circuit to containing only p-transistors, and a pulldown circuit to containing only n-transistors. A p-transistor is a conducting switch when the gate voltage is low; an n-transistor is a conducting switch when the gate voltage is high.

Hence, we can choose to implement the first four guards of the adder-cell as pulldown circuits since they do not have inverted literals, and the last two guards of the adder-cell as pullup circuits since they have only inverted literals; but then, all ouputs of the cell are inverted.

Adding an inverter to each output is expensive since the carry chain may include up to N inverters in series in addition to the N carry gates. A better solution is obtained by alternating cells that produce negated outputs—the even-numbered bits—with cells that produce straight outputs—the odd-numbered bits.

A CMOS implementation of a cell with inverted outputs is shown in Figure 3. The only noticeable disadvantage of this design is the long pullup chain (4 transistors) for the carry circuitry. We can reduce the length of these pullup chains from 4 to 3 by distributing the neutrality test even more evenly. For instance, we can choose the following distribution:

$$\begin{aligned} \neg ct \wedge \neg cf \wedge \neg at &\rightarrow st\downarrow \| \; sf\downarrow \\ \neg bt \wedge \neg bf \wedge \neg af &\rightarrow dt\downarrow \| \; df\downarrow . \end{aligned}$$

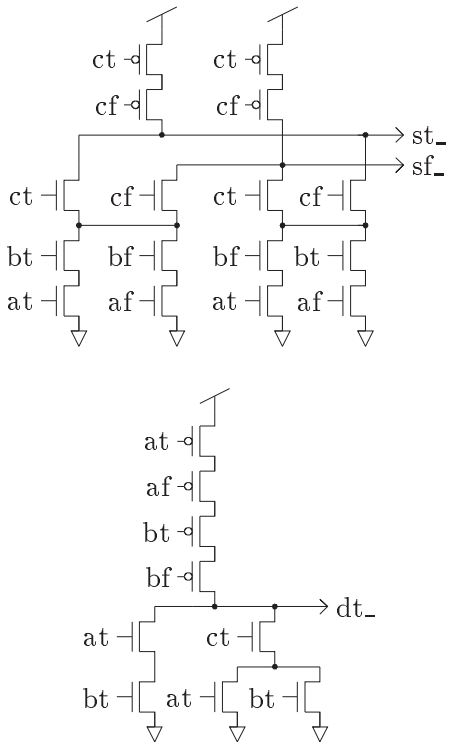

Figure 3: CMOS implementation of an adder cell

10 Performance and Comparisons

The transistor count per cell is 34. If one includes the inverters needed to invert the inputs and the outputs of every other cell, the transistor count is 42, as compared to the 40 transistors needed for an equivalent (no pass-transistors) cell design in clocked logic. Hence, contrary to common belief, the asynchronous solution is hardly larger than the clocked one, in spite of the use of dual-rail logic.

In evaluating the performance of the adder, it is important to realize that only the transitions from neutral to valid values are critical in the type of protocol (lazy-active) used. From equation (3) describing the environment protocol, we see that the environment consumes the result,

Y, and produces the next output X before testing that Y has been reset to the neutral value by the function-evaluation process. Hence, the resetting of Y to the neutral value is not on the critical path.

As we have seen, the length of the longest carry chain is proportional to the maximal number, n, of contiguous binary positions in which one input bit is different from the other. In the HP CMOS 40 process provided by MOSIS (1.6 micron feature size), the delay (in nanoseconds) for an addition is $6 + 1.2*(n-1)$. This delay includes the completion-tree delay required for the environment to detect the completion of an addition. It is usually believed that, statistically, n is about $\log N$. Hence, for $N = 32$, an adder delay is about 11 nanoseconds in the average case.

If we had to adjust the delay to the worst case, as is required in clocked logic, we would have to stretch each addition delay to accommodate the delay corresponding to $N = 32$, i.e., 40 nanoseconds, or four times the average delay!

Comparison to the similar adder designed by C.L. Seitz in [7] seems unavoidable. Seitz's adder cell contains more than 100 transistors, without counting the inverters. Hence, it is about three times larger, and also three times slower, than the adder cell presented here.

11 Conclusion

We have presented a method for the formal derivation of asynchronous datapath functions. First, an algorithm with reasonable distributive properties has to be chosen for the function evaluation; and, for that matter, ripple-carry is not the only choice for the adder. After that choice has been made, the rest of the derivation is almost automatic. Apart from some simplification of the guards, which can be important, the main decision left to the designer is how to distribute the validity test for the transient inputs, if any, and the neutrality test.

In the method presented, the validity and neutrality tests are included in the evaluation of the function output variables. Another, quite different, approach is to keep the function evaluation proper separate from the validity and neutrality tests, and to perform them concurrently.

For the method used, dual-rail coding is almost ideal because of

its distributivity property. Other codes may be better suited for the alternative method mentioned.

The adder described here has been used in a slightly different form (the inputs A and B are not dual-rail encoded as they are part of the same process as the adder) as a basis for the different asynchronous arithmetic units in the Caltech Asynchronous Microprocessor [2]. The performance of the ALUs in general has been surprisingly good [3].

Acknowledgments

I am indebted to Tony Lee for designing several beautiful asynchronous ALUs that were an inspiration for this paper. Acknowledgment is also due Ralph Back and Mark Josephs, and to my students Dražen Borković, Marcel van der Goot, Pieter Hazewindus, Tony Lee, Christian Nielsen, and José Tierno for their comments on several versions of the manuscript. The referees' comments were appreciated.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, and monitored by the Office of Naval Research.

Appendix: The Notation

- $b\uparrow$ stands for $b := \mathbf{true}$, $b\downarrow$ stands for $b := \mathbf{false}$. Those assignments are called *simple assignments*.

- The execution of the *selection* command $[G_1 \rightarrow S_1 [\!] \ldots [\!] G_n \rightarrow S_n]$, where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts (following Edsger W. Dijkstra [1], G_i is called a *guard*, and $G_i \rightarrow S_i$ a *guarded command*), amounts to the execution of *the* S_i for which G_i holds.

  Unlike Dijkstra's guarded commands, this selection is deterministic: At most one guard is true. If no guard is true, the execution of the command is suspended until some guard is true.

- *Sequencing*: Besides the usual sequential composition operator $S1; S2$, we use the concurrent composition, $S1 \parallel S2$. The concurrent composition is weakly fair.

- $[G]$, where G is a boolean expression, stands for $[G \rightarrow \mathbf{skip}]$, and thus for "wait until G holds." (Hence, "$[G]$; S" and $[G \rightarrow S]$ are equivalent.)
- $*[S]$ stands for "repeat S forever."
- Hence, the operational description of the statement $*[[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]]$ is "repeat forever: wait until some G_i holds; execute the S_i for which G_i holds."

References

[1] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs NJ, 1976.

[2] Alain J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI*, ed. C.L. Seitz, MIT Press, 351–273, 1989.

[3] Alain J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus. The First Asynchronous Microprocessor: The Test Results. *Computer Architecture News*, 17 (4):95-110, June 1989.

[4] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, 1(4), 1986.

[5] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-insensitive Circuits, *in* C.A.R. Hoare (ed), *UT Year of Programming Institute on Concurrent Programming*, Addison-Wesley, Reading MA, 1989.

[6] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*, Addison-Wesley, Reading MA, 1980.

[7] Charles L. Seitz. System Timing. in [6].