The Power of Carry-Save Addition

D. R. Lutz
D. N. Jayasimha
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210
{lutz-d,jayasim}@cis.ohio-state.edu

March 31, 1994

Abstract

A *carry-save adder* (CSA), or 3-2 adder, is a very fast and cheap adder that does not propagate carry bits. Historically, carry-save addition has been used for a limited set of intermediate calculations, with the most common example being the accumulation of the partial products of a multiplication. We examine carry-save addition from a new perspective: as a binary operation in which one of the operands is a number in carry-save form. From this perspective we develop five new uses for CSAs: (1) as fast adder-comparators for evaluating whether or not $X + Y = Z$; (2) as the basis of an expanded instruction set that can reduce branch and data hazards and decrease the cycles per instruction (CPI) on superpipelined architectures; (3) as linear-time multipliers for very large integers; (4) as arbitrary-length, constant-time, synchronous, up-down counters; and (5) as extremely fast frequency dividers.

Index terms: carry-save adders, computer arithmetic, superpipelined architectures, pipeline hazards, multipliers, counters, frequency dividers.

1 Introduction

A *carry-save adder* (CSA), or 3-2 adder, is a very fast and cheap adder that does not propagate carry bits. The usual sort of adder that propagates carry bits is called a *carry-propagate adder* (CPA). Historically, carry-save addition has been used for a limited set of intermediate calculations, with the most common example being the accumulation of the partial products of a multiplication. In this paper we examine carry-save addition from a new perspective: as a binary operation in which one of the operands is a number in carry-save form. From this perspective we develop new and interesting applications for CSAs which have the potential to dramatically improve the performance of commonly occurring arithmetic calculations on advanced computer architectures.

Except for a brief discussion of some future work in the conclusion, this paper is concerned with two's complement integers. The organization of the paper is as follows: Section 2 formally defines carry-save addition and discusses some of its interesting properties. The notion of carry-save form as an alternate number representation is introduced, and a fast method of adding and comparing numbers for equality or inequality is proved. Section 3 proposes the introduction of CSA-based instructions into the instruction set of a superpipelined architecture. These instructions can be used to reduce or eliminate many control and data hazards, this improving the performance of superpipelined machines. Section 4 discusses the problem of multiplying large integers, and shows how to construct a linear time multiplier for these integers using CSAs. Section 5 presents a new perspective on counting, and shows how very fast counters and frequency dividers can be constructed with CSAs as building blocks. Finally, Section 6 discusses the future directions of our research. In particular, we elaborate on how floating point numbers could be added with no accumulated round-off errors using CSAs. This makes floating point addition associative without sacrificing speed.

2 Properties of CSAs

An n-bit CSA consists of n independent full adders. It takes three n-bit two's complement numbers as inputs, and produces two outputs: an n-bit sum and an n-bit carry. Let $X = x_{n-1} \ldots x_1 x_0$, $Y = y_{n-1} \ldots y_1 y_0$, and $Z = z_{n-1} \ldots z_1 z_0$ be n-bit words with low order bits x_0, y_0, and z_0. An n-bit CSA produces a carry word $C = c_{n-1} \ldots c_1 0$ and a sum word $S = s_{n-1} \ldots s_1 s_0$ such that

$$c_i = (x_{i-1} \wedge y_{i-1}) \vee (x_{i-1} \wedge z_{i-1}) \vee (y_{i-1} \wedge z_{i-1}) \quad (1)$$

$$s_i = x_i \oplus y_i \oplus z_i \quad (2)$$

Note that c_0 is always 0, and that $C + S = X + Y + Z$. The high order carry bit, c_n, provides no useful information when adding signed numbers, so it is discarded. Figure 1 shows a four-bit CSA together with its inputs and outputs.

The representation of a sum by the ordered pair (C, S) is sometimes called *redundant*, because there are many values of C and S that produce the same sum [6]. In order to avoid confusion

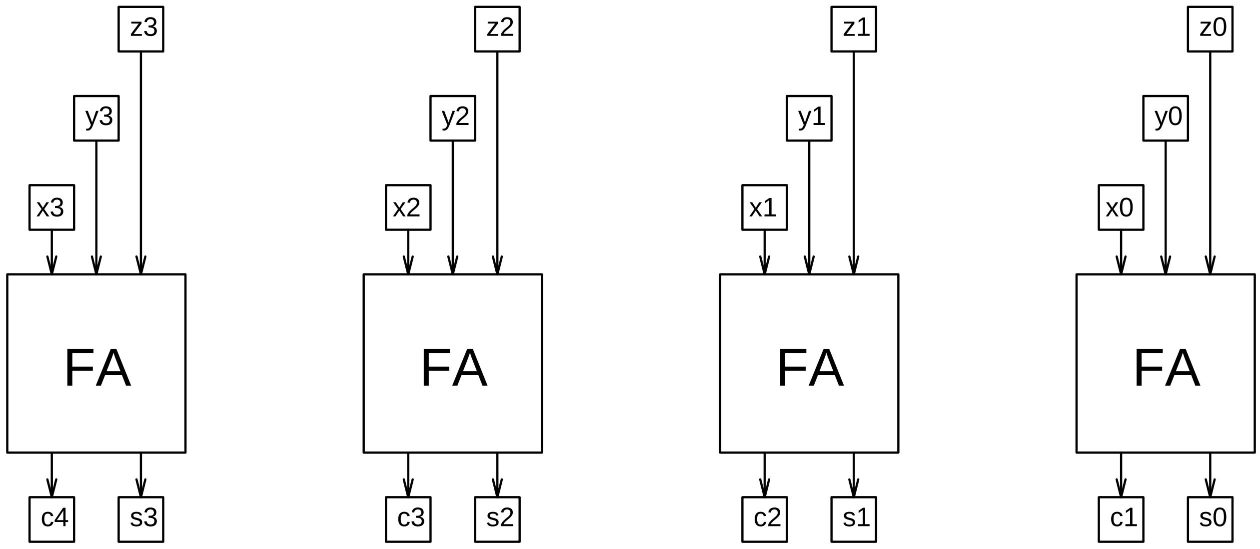

Figure 1: Basic 4-Bit CSA

with other redundant representations, we will say that (C, S) is the *carry-save form* (or *c-s form*) of the sum. When C and S are added with a CPA, the resulting number $C + S$ is the *carry-propagate* form (or *c-p form*) of the sum.

A major disadvantage of most alternate number systems, including the widely-studied residue number systems, is the difficulty of conversion between the alternate system and the two's complement system(for examples see [14]). In contrast, conversion between c-s form and c-p form is easy. Numbers in c-s form are converted to c-p form by adding them using a CPA. Numbers in c-p form are converted to c-s form by adding them to (0, 0) using a CSA.

We are interested in comparing CSAs and CPAs, and to do this requires a change in perspective. We usually think of a CSA as a circuit that adds three numbers and produces two outputs. Instead, *consider a CSA to be a circuit that adds two numbers*, one in c-s form and one in c-p form, and produces a single output in c-s form. From this point of view, both CSAs and CPAs are two-input, one-output functional units, and it is possible to compare their properties. Three major differences are apparent.

The first, and most striking difference, is the fact that a CSA produces its sum in only two gate delays, *regardless of the length of the words being added* [8], while the fastest n-bit CPA requires $\Omega(\log n)$ gate delays [5]. For 32 or 64 bit words, the fastest CPA requires around 14 gate delays, giving the CSA roughly a 7-fold speed advantage [7]. For longer words, the speedup is even more dramatic.

The second major difference is that a CSA is a much simpler circuit than a fast CPA, with hardware complexity and area growing only linearly with the size of the input. This means that large CSAs are practical (unlike large, fast CPAs), enabling us to solve some interesting problems involving large numbers.

The third major difference is that the CSA leaves its output in a somewhat inconvenient form. Determining the sign of (C, S) requires $\Omega(\log n)$ gate delays (if we could compute the sign any faster then we could make a faster CPA). A number (C, S) in c-s form requires twice as much memory as the same number $C + S$ in c-p form. Finally, it is not obvious how to determine equality or inequality between two numbers when one is in c-s form and the other is in c-p form. Fortunately, there is a solution to this last problem, beginning with an easy test to see if $(C, S) = -1$.

Lemma 1 *Let (C, S) be a number in c-s form. Then $C + S = -1 \Leftrightarrow C$ and S differ in every bit position.*

Proof:

[$\Leftarrow$] In two's complement numbers, -1 is represented by a word in which every bit is 1, so this part of the proof is immediate.

[$\Rightarrow$] Let i be the lowest order bit such that $c_i \oplus s_i = 0$. Then the ith bit of $C + S$ is 0 unless there is a carry from the next lower order bit. The sum of two bits generates a carry if and only if both bits are 1. But i is the lowest order bit with $c_i \oplus s_i = 0$, so there can be no carry into i. Since the ith bit of $C + S$ is 0, we have $C + S \neq -1$, contradicting our hypothesis. $\Box$

The time to compute the test in lemma 1 is simply the time to compute a 2-input xor plus the time to compute an n-input and. This is only one 2-input xor more delay than the time it takes to determine if a number in c-p form is zero or nonzero.

We would like use our fast test for -1 as part of a fast test for an arbitrary value. In other words, how can we use a fast comparison with -1 to tell if $(X, Y) = Z$ for some arbitrary Z? Lemma 2 tells us how to accomplish this.

Lemma 2 *Let X, Y, and Z be n-bit two's complement numbers. Then $X + Y = Z \Leftrightarrow X + Y + \overline{Z} = -1$*

Proof:

$$\begin{aligned} X + Y = Z &\Leftrightarrow X + Y - Z = 0 \\ &\Leftrightarrow X + Y - Z - 1 = -1 \\ &\Leftrightarrow X + Y + \overline{Z} = -1 \end{aligned}$$

The last line holds because Z is a two's complement number and so $-Z = \overline{Z} + 1$. $\Box$

The time to compute $X + Y + \overline{Z}$ in c-s form is the time to compute a *not* plus the time to compute a carry-save add.

Theorem 1 follows immediately from lemmas 1 and 2.

Theorem 1 *Let X, Y, and Z be n-bit two's complement numbers, and let* $(C, S) = X + Y + \overline{Z}$. *Then* $X + Y = Z \Leftrightarrow C$ *and* S *differ in every bit position.*

Theorem 1 gives us a very fast method to determine whether or not $X + Y = Z$. Note that the proof did not require that any of the inputs be in c-s form. In fact, the method is broadly applicable to a very frequently occurring comparison: a comparison that occurs, among other places, in the execution of iterative loops. A four-bit design incorporating the method of theorem 1 is given in figure 2.

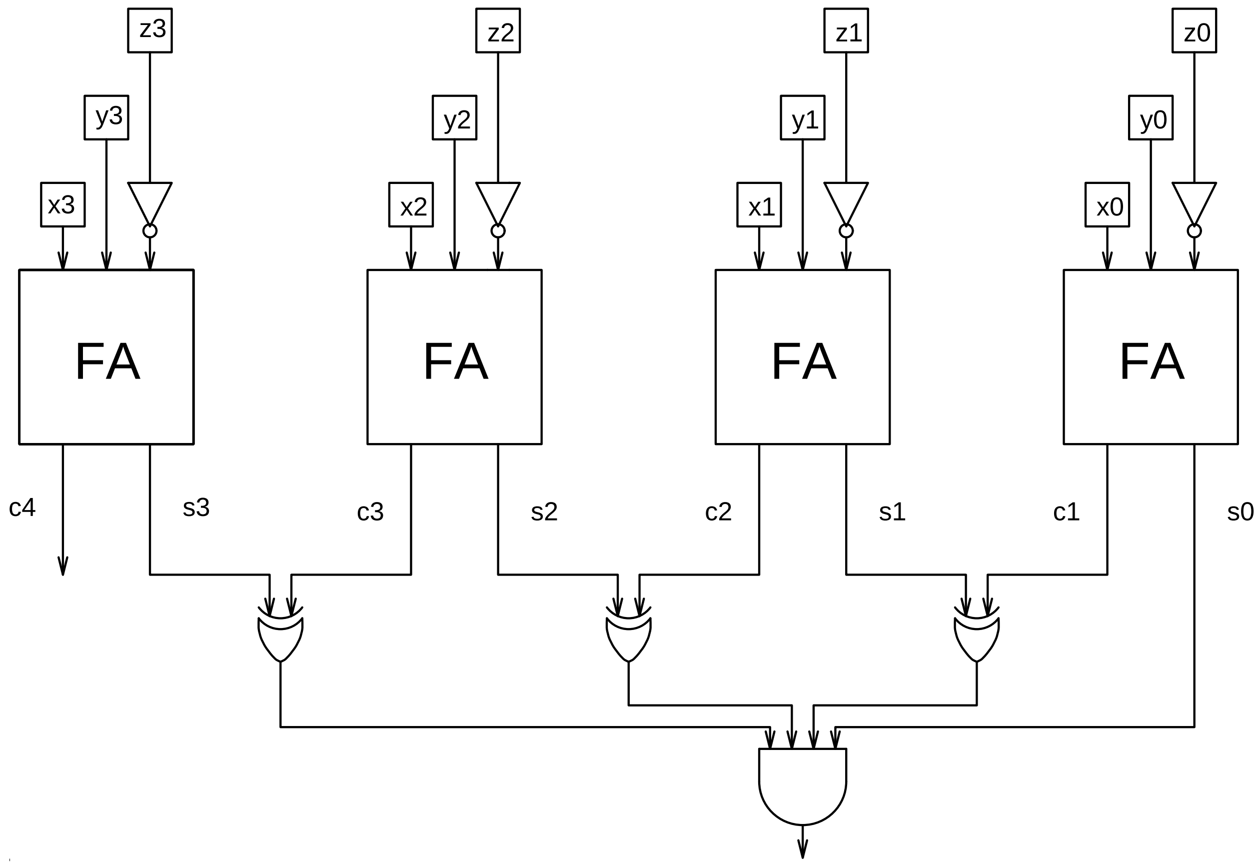

Figure 2: CSA-based comparator for determining if $X + Y = Z$

Cortadello and Llaberia were the first to give a method for evaluating whether $X + Y = Z$ without the need for carry propagation [4]. Our design is different from theirs, and has some advantages. First, it is considerably easier to understand and modify. For example, in section 5 we will modify the design to produce a fast programmable frequency divider. Second, it is easy to separate into two useful functions: carry-save add and compare with -1. The benefits of these functions will be discussed in section 3. Finally, our design may also be easier to implement, because its basic component, the full adder, has been very well studied [12].

Our design is much faster than any design employing CPAs to add and perform comparisons. The fastest CPA-based design to perform this test would add X and Y with a CPA and compare each bit of the sum with the corresponding bit of Z, as in figure 3. For a 32- or 64-bit architecture, the CPA-based comparator would take 14 gate delays for the carry-propagate add, and then 4 gate delays (assuming a fan-in of 4 per *and* gate) to complete the comparison. Under the same assumptions, our CSA-based comparator can complete the comparison in only 6 gate delays.

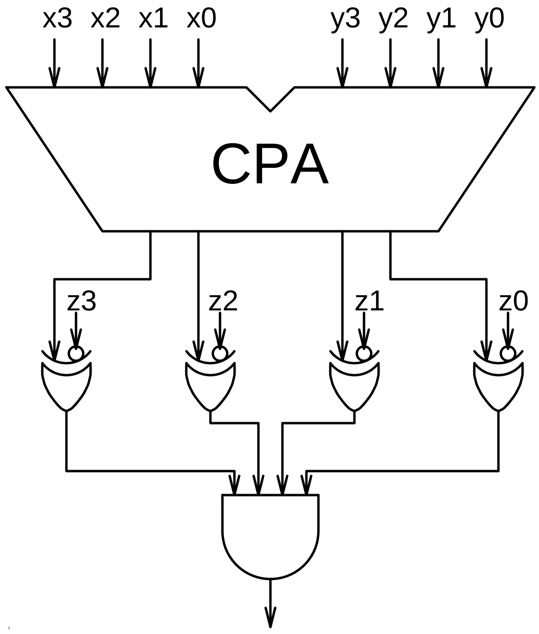

Figure 3: CPA-based comparator for determining if $X + Y = Z$

We summarize this section with the following observation:

Observation 1 *If we consider a CSA to be a functional unit that operates on numbers in c-s form, then we have an adder with four remarkable properties:*

- *constant, two gate-delay add time, regardless of word length;*
- *hardware cost that grows only linearly with word length;*
- *an output format that converts easily to ordinary two's complement numbers; and*
- *an output format that can easily be tested for equality or inequality with a two's complement number.*

3 Reducing the CPI of Superpipelined Computers

A superpipelined machine is defined by comparison with a contemporary pipelined *base machine*. We will consider the base machine to be Hennessy and Patterson's DLX pipelined machine [6]. DLX has five pipelined stages: instruction fetch, instruction decode, execute, memory, and writeback. The machine cycle in DLX is long enough for the "execute" stage to complete a carry-propagate add.

A *superpipelined machine of degree* m is a pipelined machine in which the cycle time is only $1/m$ as long as the cycle time of the base machine, and the pipeline has about m times as many stages [9, 10]. Examples of superpipelined machines include the Cray-1 and the Cray Y-MP, which are superpipelined of degree 3.

While a carry-propagate add takes one cycle to complete on the base machine, it takes m cycles to complete on the superpipelined machine. The advantage of the superpipelined machine is that given enough instruction-level parallelism, it is about m times as fast as the base machine.

In an ideal pipeline, an instruction would be completed on every cycle. This ideal pipeline would have *cycles per instruction* (or *CPI*) = 1. This does not happen because of three *hazards*: *structural hazards*, which occur when there are insufficient hardware resources to support a given sequence of instructions; *data hazards*, which occur when an instruction cannot continue until it obtains a result produced by some previous instruction that is still in the pipeline; and *control hazards*, which occur when the normal flow of control is changed by an instruction [6]. Structural hazards can be eliminated by adding sufficient hardware. In this paper we are concerned only with data and control hazards.

When an instruction encounters a hazard, the pipeline is stalled for one or more cycles until the hazard is resolved. On superpipelined architectures, hazards are especially problematical. Because instruction level parallelism is limited, and because branch statements are so common, the longer pipelines are more likely to contain hazards. Because the instructions have longer latencies with respect to the cycle time, hazards can take more cycles to be resolved. Thus reducing data and control hazards is particularly important for superpipelined machines.

One way to reduce these hazards is to replace high-latency instructions with low-latency instructions. Consider the problem of adding three numbers, i.e., calculating

```
I = J + K + L
```

on a superpipelined machine of degree 3. We assume that J, K, and L are already in registers. Two instructions are needed: the first adds J and K, and the second adds $J+K$ and L. The two instructions can be issued on consecutive cycles, but at the execution stage the second instruction must stall for at least two cycles because of a data hazard ($J+K$ has not been produced). Sometimes the compiler can rearrange instructions so that useful work can be done during these two cycles, but this is by no means guaranteed.

On the base machine, this situation does not result in a data hazard because of a technique called *data forwarding*. In data forwarding, the result of the execution stage is forwarded to subsequent instructions so that they need not wait for the value to be stored in a register. Data forwarding is also useful in the superpipelined machine, but it still cannot prevent a data hazard because *the execution now occupies three cycles instead of one*. Until the three execute cycles have completed, there is no data value to forward.

In the above example, the value $J + K$ is never needed, at least in c-p form. The two-cycle stall is spent waiting for a value that is of no use once we have calculated $J + K + L$. This discussion leads us to make the following observation.

Observation 2 *When adding three or more numbers together, we usually do not care about the values of the intermediate sums, and so it is a waste of computational effort to compute these values with a CPA.*

This wasted effort is hidden on machines such as the base machine whose cycle time is long enough to complete a carry-propagate add, but becomes apparent on superpipelined machines. One way to get rid of this wasted effort is to add a carry-save add instruction to the instruction set. A carry-save add instruction would use a CSA to add the contents of three registers, sending the outputs to two registers. Unlike the carry-propagate add instruction, the carry-save add instruction would complete in a single cycle.

There are many possible ways to design such an instruction, but for our present purposes we will use the following simple design. The instruction

```
CSA R1,R2,R3
```

will mean to do a carry-save addition on registers R1, R2, and R3, putting the results of the carry-save addition in R1 and R2, with the carry going into R1 and the sum going into R2.

In contrast, the carry-propagate add instruction

```
ADD R1,R2,R3
```

means to add R2 and R3 with a CPA and put the result in R1.

Given these instructions, we can now calculate $I = J + K + L$ without the 2-cycle stall. The first instruction adds $J + K + L$ with a CSA, and the second adds the two outputs with a CPA.

Other applications arising from observation 2 include:

- counting – counter values are usually incremented much more often than they are used. The carry-propagate add can often be delayed until the final value is needed. For example, a program that counts the number of characters in an input stream could maintain its count in c-s form.

- calculating array or structure addresses – this is usually done with a sequence of shifts and adds, and only the final sum is useful. All of the adds except the last could be done as carry-save adds.
- maintaining loop indexes – we will show how to do this in an example later in this section.

A compiler can detect when variables are used by the standard technique of examining def-use chains [1]. We also can determine which kinds of uses require their inputs in c-p form. For example, if the use is a multiplication then the inputs are required to be in c-p form, but if the use is a sum then the inputs could be in either form. To save register space, the compiler would only want to use carry-save instructions when it is necessary to avoid a data hazard.

Given a carry-save add function, we would probably also want a carry-save subtract function

```
CSS R1,R2,R3
```

that would subtract R3 from the sum of R1 and R2. The carry-save subtraction can be performed by inverting the bits of register R3; placing a 1 in the low order bit of R1; performing a carry-save addition on R1, R2, and R3; and placing the results of the carry-save addition in R1 and R2, with the carry going into R1 and the sum going into R2. We need to place a 1 in the low order bit of R1 in order to complete the negation of R3. The reason that we can place a 1 in the low order bit of R1 is that the carry word will never have the low order bit set, at least not after the first carry-save add. The compiler can keep things straight, and if there has not been a carry-save add into R1 and R2 it can perform one, e.g.

```
CSA R1,R2,0
CSS R1,R2,R3
```

This method allows for very fast subtraction (three gate delays). Observe that we have once again substituted a high-latency instruction with a low-latency instruction. If the output from the subtraction is needed as part of a sum, there will be no data hazards as there might be with a carry-propagate subtraction.

Another high-latency instruction is multiplication. If we assume a very fast multiplier, then significant savings can be achieved by leaving the output of the multiplier in c-s form when it is not needed in c-p form. For example, the calculation of many array or structure addresses involve a multiplication (to find the starting address) followed by an addition (to find the offset). In this case, the address can be calculated more quickly (and stalls can be avoided) if the multiplier does not propagate the final carry.

The CSA-based instructions can also help with certain control hazards, such as those introduced in loops. Loop overhead is the time required to check whether a loop has terminated, and to branch back to the top of the loop if it has not. For example, consider the following loop, that arises in many numerical packages such as LINPACK:

```
    DO 10 I=1,N
10  Y[I] = A * X[I] + Y[I]
```

There are currently two fast ways to handle the overhead for this loop. The first method is to set a register to the value -N (or +N), and then increment (or decrement) and test for 0. The test for 0 can be very fast (approximately log N gate delays), but the increment or decrement operation requires a full carry propagate add. The second method is to increment a pointer to X and a pointer to Y and compare one of the pointers to a terminating value (say the address of Y[N]). The normal method for comparing two numbers is to subtract them, so even though we have discarded the variable I we still have the overhead of one subtraction per loop iteration. In both cases, we have to wait 3 cycles before we can determine whether to branch or not.

If we have the CSA instruction, we could implement the loop by setting I to $-N$ (possibly in c-s form) and then incrementing and testing for equality to -1 on each iteration. As shown in Lemma 1, this is particularly easy to test. One way to implement the test would be to have a branch on carry-save equal to -1 instruction. This instruction

```
    BCSEQM1 R1,R2,label
```

would mean to branch to "label" if $R1 + R2 = -1$. Putting these instructions together we get the code for the loop overhead shown below, in which the branch decision can be made in the cycle following the carry-save adds.

```
        LWI R1,0            # R1 <- 0
        LWI R2,0            # R2 <- 0
        LWI R3,N            # R3 <- N
        CSS R1,R2,R3        # (R1,R2) <-  -N
label:  . . .               # code for Y[I] = A * X[I] + Y[I]
        CSA R1,R2,1         # increment loop counter (R1,R2)
        BCSEQ R1,R2,label   # branch to label if R1 + R2 = -1
```

There are many details to be worked out before we can unequivocally recommend a particular set of carry-save functions. However, the basic principle, as given in the following observation, is clear.

Observation 3 *Replacing high-latency carry-propagate additions (and subtractions, comparisons, multiplications, etc.), with their low-latency carry-save equivalents can eliminate data hazards.*

How much time can we save by adding carry-save instructions to superpipelined machines? Unfortunately, there is no data on how many carry-propagate instructions can be replaced by their carry-save equivalents in a typical program mix. We do know that control and data hazards are a big problem. In [15], a simulation of a superpipelined scalar version of the Cray 1 with no structural hazards still showed a CPI of more than 2 on the first 14 Lawrence Livermore Loops (The ideal CPI would be 1). This means that more than half of the available cycles were spent on stalls because of control and data hazards. Most of these stalls were probably related to loop overhead, because the CPI in a similar simulation was close to 1 when the loops were unrolled eight times [17].

Some measurements on DLX show that conditional branches based on a simple test for equality or inequality with zero account for more than half of all conditional instructions, and approximately 11 percent of all instructions executed (The data used in this paragraph are taken from appendix C in [6]). It is likely that most of these tests were preceded by carry-propagate additions or subtactions, many of which could cause a data hazard on a superpipelined machine. Replacing these instructions with their CSA-based equivalents could save two stall cycles on each of these branches. Additions account for 20 percent of all DLX instructions executed, but there is no indication as to how many of these additions could profitably be replaced with carry-save additions. We plan to do simulations at the instruction level on a variety of applications before recommending CSA-based instructions.

4 Multiplying Large Integers in Linear Time

So far, we have examined CSAs in the context of ordinary length words. What use can we make of the fact that CSAs are equally fast for large words?

One application that uses long words, and in particular long multiplications, is encryption. The RSA encryption algorithm, for example, requires repeated multiplication of numbers that are hundreds of bits in length [13].

It is easy to construct an n-bit multiplier that uses a single CSA to multiply in linear time. The design (figure 4) is essentially the same as the multiply-step design used in some RISC architectures, with the accumulation done in c-s form. Initially, the multiplicand is loaded into register X, and the multiplier into register M. Registers C and S are set to 0. Then we examine each bit $m_i (0 \leq i \leq n-1)$ in sequence. If m_i is 1, then register X is added to C and S with a CSA. Whether or not m_i is 1, X is shifted left 1 bit. After n steps, C and S contain the product in c-s form. The c-p form of the final product can be obtained by adding C and S with a CPA.

Note that this design is scalable, and that it yields a cheap linear time multiplier even when n is large. Each of the multiply/add steps can be performed in constant time, and all of the data that is needed is in registers, so the cycle time can be much faster than is usual for a CPU. The final addition needed to convert from c-s form to c-p form could be performed by repeated calls to a 32 or 64-bit CPA.

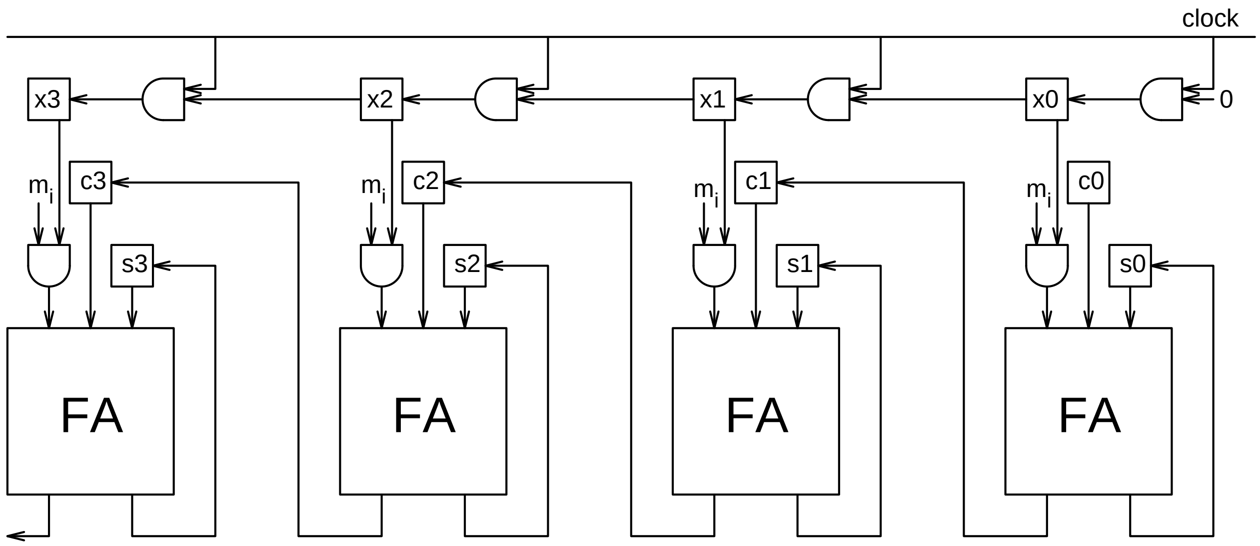

Figure 4: First 4 Bits of a Simple CSA-Based Multiplier

The current method for multiplying large integers is to make repeated calls to a 32 or 64-bit multiplier, which is a slow process. For numbers in the 300 to 7000-bit range, the Karatsuba algorithm gives the best results ($O(n^{\log_2 3})$ multiplies), while 7,000 or more-bit numbers are best multiplied using an FFT-based algorithm ($O(n \log n)$ multiplies) [11]. Both of these methods have fairly high overhead, and they would be considerably slower than the low-overhead linear-time multiplication provided by the special-purpose hardware proposed here.

For an example of the possible savings, our best hand-tuned algorithm for multiplying 1000-bit numbers on a SPARC SLC (done as part of [11]) consumed over 330,000 cycles. Given the same technology, but ignoring I/O, our 1000-bit multiplier could produce the same output in c-s form in 1000 cycles. Given the slowest possible carry-propagation, which would be to propagate the carries by running the multiplier for another 1000 cycles, the output would be available in c-p form in 2000 cycles. Since the cycle time of our multiplier could probably be 1/4 of the cycle time of a processor using the same technology, let us assume that we can do the multiplication in 500 processor cycles. The time to read the multiplicand is 32 cycles. An additional cycle is required to read the LSB of the multiplier (the rest of the multiplier could be read during the multiplication). The output begins during the the carry-propagation stage, and completes one cycle after the carry-propagation is completed. Thus the total multiplication time (including I/O) would be something like 534 cycles, more than 600 times faster than the best method using repeated calls to a fast fixed-length multiplier.

This long word multiplier could also be used to multiply a vector by a scalar. Suppose we had a 4k-bit multiplier. Then we could multiply a vector of 64 32-bit numbers by a 32-bit scalar by loading the 64 numbers into the multiplicand register X with 32 zeros in between each number. Then after 32 (fast) cycles, we could read all 64 64-bit products.

5 A New Perspective on Counting

There are many different circuits that are called counters. We focus on two of them: (1) a circuit that counts the number of input pulses it receives; and (2) a circuit that provides an output pulse that is some multiple of its input pulse. This second type of counter is sometimes called a *frequency divider* or a *scaler*.

For the first type of counter, current designs update the count in $O(\log n)$ gate delays, and then allow access to the count (i.e., allow the count to be read) in constant time. We propose that this is exactly the opposite of what a fast counter should do.

Observation 4 *Counter values are updated much more often than they are retrieved, so for fast counters, we should emphasize the speed of updating over the speed of retrieval.*

Figure 5 shows a counter that maintains its count, (C, S), in c-s form. The count is updated by adding the value in register X to (C, S) in response to an input pulse (the input pulse is not shown in the figure). For the simplest type of counter that just counts input pulses, then X contains the value 1. The count is retrieved by adding C and S with a CPA. Unlike the usual design, updates occur in constant time, and the slow carry-propagation only takes place when the count is retrieved.

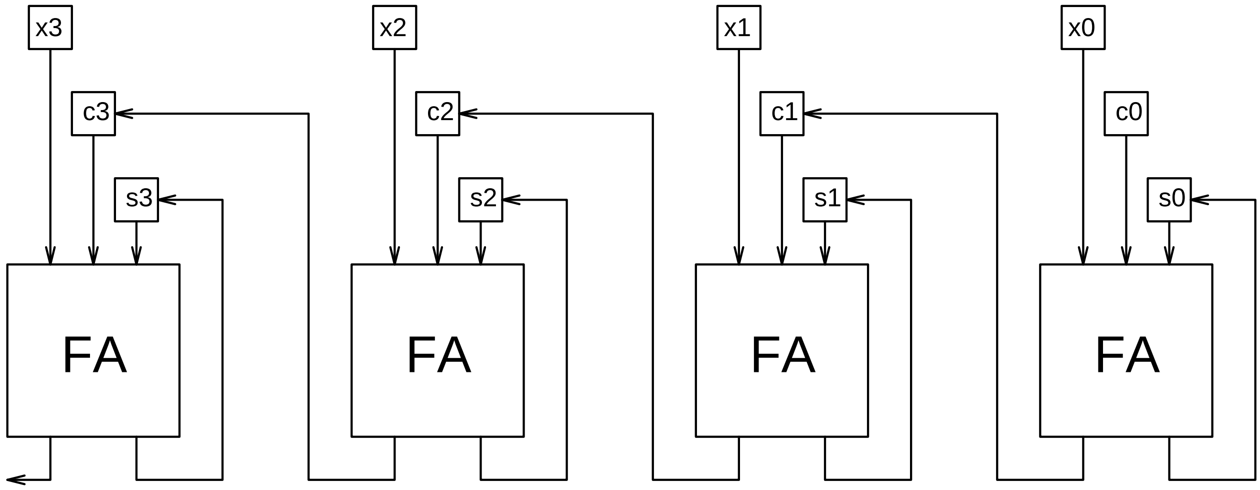

Figure 5: 4-Bit CSA-Based Counter

In [16], Vuillemin presents the following open question: "is it possible to design a synchronous, arbitrary length, constant time up-down counter"? If we accept the notion that there is no need to maintain the count in c-p form, then figure 5 is such a counter. The counter is synchronous, because each of its bits are updated simultaneously. We can count down by setting $X = -1$. In fact, we can count up or down by an arbitrary value by setting

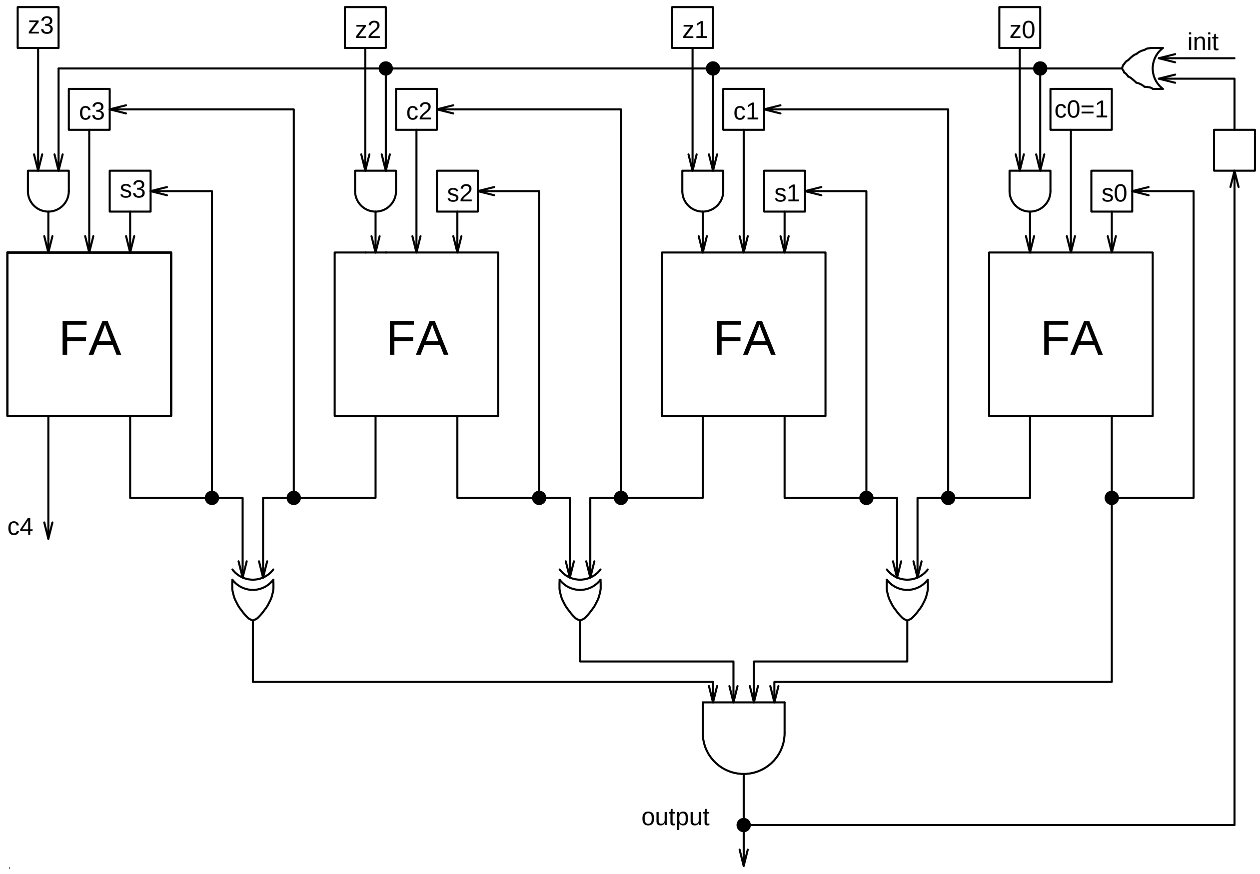

Figure 6: 4-Bit CSA-Based Frequency Divider

X to that value. The time to increment (or decrement) the counter is only the delay of a full adder plus the length of time it takes to store the outputs.

For the second kind of counter, the frequency divider, the carries never have to be propagated. Figure 6 is a programmable frequency divider based on Theorem 1. The counter works by counting up until it reaches -1. Since we want to output a 1 after T cycles, the sum of the original inputs should be $-T$. We achieve this by setting $Z = -T$, $C = 1$, and $S = -1$. The first input signal (the input signal is not shown) causes the adders to add C, S, and Z. The second and succeeding signals add 1 to this sum (because of the 1 in c_0). After each addition, the c-s sum is compared to -1, using the fast comparison given in lemma 1. When -1 is attained, an output pulse is produced. The output pulse also allows the value in register Z (which is still -T, unless some other frequency is desired) to become part of the next input cycle, which brings us back to the initial state.

The frequency divider can handle any frequency in the range from 1 to 2^n.

6 Future Directions

We are currently investigating new uses for carry-save addition in floating point arithmetic. One interesting possibility is given in [2] and [3]. In these papers, Cappello and Miranker present designs for systolic super summers, which are adders that add floating point numbers in fixed point form. We believe that these designs could be simplified by using the carry-save addition concepts in this paper.

The basic idea is to convert the floating point numbers to fixed point form. For a number with M mantissa bits and E exponent bits, the conversion can be done in E stages. The initial stage consists of the M mantissa bits. Each subsequent stage involves the examination of an exponent bit e, and the placement of all of the bits from the previous stage to one of two new locations, depending on whether e is set or not. The number of bits in each stage grows, until the final stage contains $2^E + M - 1$ bits, 1 of which is a sign bit. For example, an IEEE 754 single precision number could be represented by a 279-bit fixed point number. Each stage of this conversion requires only two gate delays, and the process would be easy to pipeline.

After this conversion, we can accumulate the numbers in carry-save form. An interesting simplification to Cappello's and Miranker's designs is to use the sign bit to indicate whether to add or subtract each new number N from the carry-save sum (C, S). Using the method for carry-save subtraction given in section 3, we would perform the subtraction by inverting the bits of N, placing a '1' in the low order bit c_0 of C, and adding with a CSA. It is safe to replace c_0 with 1 because c_0 is always 0 for a number in c-s form. The addition or subtraction would only cost two or three gate delays, respectively, which is fast enough to keep up with any superpipelined machine.

When the last number has been added, the carry bits have to be propagated and the sum converted back to fixed point form. We can do this using an ordinary-length CPA by locating the highest-order significant bit in $(C \vee S)$, and then adding M bits of C and S starting with this position. The position of this bit will also provide us with the exponent for the final sum. While this procedure is somewhat complicated, it only needs to be done once for each sum, irrespective of the number of terms that are being added.

There are at least four advantages to such a scheme:

- Addition by this method is associative (unlike ordinary floating point addition), which would greatly simplify numerical programming and numerical analysis.
- Accumulated roundoff error is eliminated. Sums are only rounded once, instead of being rounded after each addition.
- the whole process can be pipelined so that it can keep up with vector superpipelined machines.
- In a multiprocessor environment, multiple processors could compute very large sums in parallel with no accumulated roundoff error. The method would be to have each

processor maintain its sum in fixed-point c-s form, and then combine the sums in the usual binary fashion to get the final sum. Only this final sum would have the carries propagate and be converted back to a floating point representation.

There are a number of other interesting questions that we are currently addressing, particularly with regard to the work in section 3. How can carry-save instructions best be incorporated into an architecture? Should there be dedicated carry-save registers, or should we use normal registers? What is the best instruction set? How can compilers best take advantage of the carry-save instructions? We need to run simulations to see what kind of savings are generated by the various proposals, and to see what effect the CSA-based instructions have on the CPI of superpipelined machines.

An unresolved issue concerns overflow. It may not be possible to detect overflow when a number is in c-s form. This is probably not a major concern for most applications, but certain programs may have to avoid CSA-based instructions in certain situations– the compiler could detect such situations and generate the proper code.

We have not yet considered superscalar architectures. What effect would carry-save instructions have on these architectures? Jouppi and Wall showed that superscalar and superpipelined machines have roughly the same performance [10]. If it turns out that superpipelined machines can be made significantly faster with the addition of CSA-based instructions, then it is likely that superpipelined machines will outperform superscalar machines.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] Peter R. Cappello and Willard L. Miranker. Systolic super summation. *IEEE Transactions on Computers*, 37:657–677, June 1988.

[3] Peter R. Cappello and Willard L. Miranker. Systolic super summation with reduced hardware. *IEEE Transactions on Computers*, 41:339–342, March 1992.

[4] Jordi Cortadella and Jose M. Llaberia. Evaluation of $a + b = k$ conditions without carry propagation. *IEEE Transactions on Computers*, 41:1484–1488, November 1992.

[5] Barry S. Fagin. Fast addition of large integers. *IEEE Transactions on Computers*, 41:1069–1077, September 1992.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Palo Alto, California, 1990.

[7] F. Hill and G. Peterson. *Digital Systems: Hardware Organization and Design*. John Wiley and Sons, Inc., New York, New York, third edition, 1987.

[8] Kai Hwang. *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley and Sons, Inc., New York, New York, 1979.

[9] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., New York, New York, 1993.

[10] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *Third Symposium on Parallel Algorithms and Architectures*, pages 272–282, 1989.

[11] W. Kuechlin, D. Lutz, and N. Nevin. Integer multiplication in parsac-2 on stock microprocessors. In *Ninth International Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes*, October 1991.

[12] Tso-Kai Liu, Keith R. Hohulin, Lih-Er Shiau, and Saburo Muroga. Optimal one-bit full adders with different types of gates. *IEEE Transactions on Computers*, 23:63–69, January 1974.

[13] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

[14] Norman R. Scott. *Computer Number Systems and Arithmetic.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[15] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39:349–359, March 1990.

[16] J. E. Vuillemin. Constant time arbitrary length synchronous binary counters. In *10th IEEE Symposium on Computer Arithmetic*, pages 180–183, June 1991.

[17] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–109, 1987.