

confirms that the lower cost star graph cannot perform better than the hypercube on these communication problems, and so lower cost brings lower performance.

#### REFERENCES

- [1] Y. Saad and M. Schultz, "Topological properties of hypercubes," *IEEE Trans. Comput.*, vol. 37, no. 7, pp. 867-871, July 1988.
- [2] W. Dally and C. Seitz, "The torus routing chip," Dept. of Comput. Sci., California Inst. Technol., Tech. Rep. 5208:TR:86, Jan. 1986.
- [3] D. Agrawal and L. Bhuyan, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Trans. Comput.*, vol. C-33, no. 4, pp. 323-333, Apr. 1984.
- [4] S. Lakshminarayanan and S. Dhall, "A new hierarchy of hypercube interconnection schemes for parallel computers," *J. Supercomputing*, vol. 2, pp. 81-108, 1988.
- [5] S. B. Akers and B. Krishnamurthy, "A group-theoretic model for symmetric interconnection networks," *IEEE Trans. Comput.*, vol. 38, no. 4, pp. 555-566, Apr. 1989.
- [6] S. Johnsson and C. Ho, "Spanning graphs for optimum broadcasting and personalized communication in hypercubes," *IEEE Trans. Comput.*, vol. 38, no. 9, pp. 1249-1268, Sept. 1989.
- [7] D. C. MacKenzie and S. R. Seidel, "Broadcasting on three multiprocessor interconnection topologies," Dept. Comput. Sci., Michigan Technol. Univ., Tech. Rep. CS-TR-89-01, Jan. 1989.
- [8] B. Helminen and D. Poplawski, "A performance characterization of the FPS T-Series hypercube," Dept. Comput. Sci., Michigan Technol. Univ., Houghton, Tech. Rep. CS-TR-87-06, 1987.
- [9] D. K. Bradley, "First and second generation hypercube performance," Dept. Comput. Sci., Univ. of Illinois, Tech. Rep. UIUCDCS-R-88-1455, Sept. 1988.
- [10] D. A. Poplawski, "Node and communication performance of the Ametek Series 2010," Dept. Comput. Sci., Michigan Technol. Univ., Houghton, Tech. Rep. CS-TR-89-02, Feb. 1989.

### Designing Optimum One-Level Carry-Skip Adders

Vitit Kantabutra

**Abstract**—We show how to design one-level carry-skip adders that attain very high speeds in practice. One-level carry-skip adders are very fast adders that are hardly more complex than the much-slower ripple adders. Our design procedure allows the use of realistic component delays obtained by simulation, and is technology-independent. An example of a 64-bit, 1  $\mu\text{m}$  CMOS adder is given, which achieves an add time of 6.23ns, measured by SPICE simulation with realistic loads. This delay figure excludes sum buffering delays, which depend on the particular application of the adder. The combination of high-speed and simplicity makes one-level carry-skip adders attractive for applications in highly parallel systems.

**Index Terms**—Adders for control units, adders for parallel computers, addition, area-efficient circuits, carry-skip adders, computer arithmetic, delay optimization, digital circuits, low-power circuits, VLSI.

Manuscript received December 6, 1991; revised June 16, 1992 and September 30, 1992. This work is based on "Designing Optimum Carry-Skip Adders" by Vitit Kantabutra, in the *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, June, 1991.

V. Kantabutra is with the Department of Computer Science, State University of New York, Brockport, NY 14420.

IEEE Log Number 9207289.

#### I. INTRODUCTION

Adders are very important because they play several roles in every CPU. They are used not only for addition, but also for other operations such as subtraction, multiplication, division, and address computation.

Speed is not the only important criterion in judging an adder. There are many other important criteria, such as power consumption and layout area. In CMOS technology, Lynch and Swartzlander's 56-bit "Spanning-Tree Carry-Lookahead Adder" [1] achieves a delay of only 3.2 ns with AMD's 1  $\mu\text{m}$  design rule process. According to an earlier version of their paper [2], a traditional carry-lookahead adder, believed by many to be the fastest known adders, would be about twice as slow as their adder. However, their adder is large because it employs three levels of Manchester carry chains, as well as 104 ripples cells and 48 multiplexers. Traditional carry-lookahead adders are also large.

Recently *carry-skip adders*, invented by C. Babbage in the 1800's [3], are becoming popular because they are fast and yet in CMOS technology they require low layout area and power. In CMOS technology, carry-skip adders with two or more levels of skip are already known to be comparable in speed with traditional carry-lookahead adders [4]. However, it was thought that one-level carry-skip adders were much slower.

In this paper we will show that tightly-designed one-level carry-skip adders can be comparable in speed to the speeds achieved by traditional carry-lookahead adders in CMOS, which is about 6-7 ns for 64 bits, when using 1  $\mu\text{m}$  CMOS, and have the advantage of being quite small and power (energy)-efficient. See Fig. 1 for an 18-bit example.

We can easily project that when 0.5-0.6  $\mu\text{m}$  drawn-channel-length CMOS technology becomes standard for logic circuits in 1993, one-level carry-skip adders will be at least as fast as the 5 ns, 64-bit adders that some leading companies plan to offer.

Interest in carry-skip adders after Babbage's time didn't seem to have started until Lehman and Burla [5] wrote a paper on delay optimization in the early 1960's. After that there have been several other papers on delay optimization [6]-[11], where [9] is actually a former version of this paper. Lehman and Burla [5] discussed how to divide the operand bit positions into blocks to attain the fastest adder speed possible with *equal block sizes*. They also speculated that unequal blocks could yield even faster adders. Oklobdzija and Barnes [10] proved them right and published a procedure to obtain the best block sizes<sup>1</sup> if the skip time (time for skipping a block) is a constant, and is an *integer multiple* of the ripple time (time for rippling through a bit position, which is also assumed to be a constant). Guyot *et al.* [8] presented a method of finding near-optimum block sizes while allowing the skip time to be an integer or a noninteger multiple of the ripple time. Their method involves a very nice, intuitive process of containing graphs of block sizes in triangles, and is computationally very efficient in practice. Their method is also generalizable to multiple-level carry-skip adders and to more complex delay models. Kantabutra [9], in an earlier version of this paper, tightened Guyot *et al.*'s result to achieve optimality, while still using the triangle method. Chan *et al.* [6], [7] presented algorithms for computing the best block sizes for a large variety of one- and multiple-level carry-skip adders, and even for block carry-lookahead adders. The methods of Chan *et al.* involve dynamic programming, are computationally intensive, and do not yield a simple, intuitive

<sup>1</sup> Best block sizes = block sizes that yield the fastest adders.

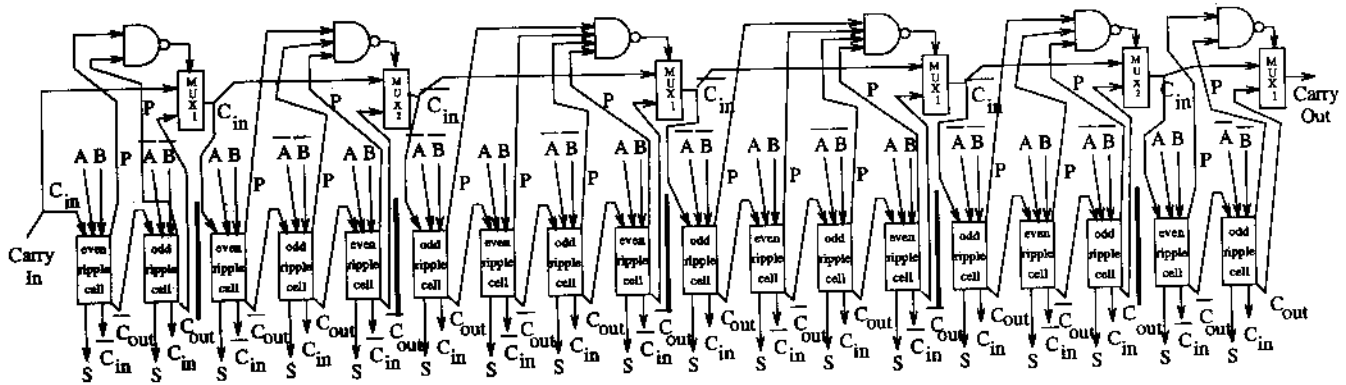


Fig. 1. A one-level carry-skip adder.

understanding of optimum block sizes. Turrini [11] also showed how to optimize block sizes for multilevel carry-skip adders.

This paper, in its current version, departs from all the earlier papers (including the earlier version [9]) in that the circuit components' delays to be used in the circuit design procedure are obtained by simulating (with SPICE) such components using the same loads that would appear in a real adder. Previously, simulation was only used for finetuning the design of adders, rather than for obtaining an optimum design in the first place. Thus, an adder obtained by using the procedure presented here is very likely to be closer to optimum than the adders obtained without using simulated or measured data. Use of simulated data requires a slight extension of the theory in [9] due a greater complexity in the component delay models.

## II. THE COMPONENTS

We now present the components of a carry-skip adder. To make our discussion concrete, we will assume that our adders use the same components as in Fig. 1. In an adder there are two kinds of ripple cells (full adders), which are used alternately in the adder: *even* and *odd* ripple cells, both shown in Fig. 2. These cells are the same as those used in [8], but with a slight correction made.

Fig. 3 shows the inverting tristate gate, the XOR gate, as well as the multiplexers, MUX1 and MUX2.

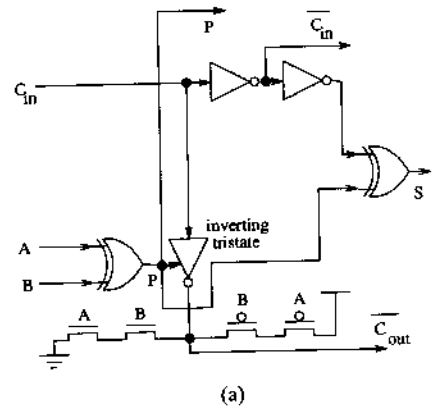
MUX1 is the usual 1-select-line multiplexer, whereas MUX2 inverts one of its inputs to accommodate the fact that our ripple cells invert their carry signals. Most gates we use here are the same as or similar to the ones used in [8]. The transistors used in the main description of the design process are the ones described by the pss and nss models presented in Glasser and Dobberpuhl [12], and are of  $2\ \mu\text{m}$  channel length and  $15\ \mu\text{m}$  channel width. Another, more modern example adder is also given in this paper: a 64-bit adder using HP CMOS26B components ( $1.0\ \mu\text{m}$  drawn channel length), whose parameters are those which were measured by MOSIS. Note that in the  $1\ \mu\text{m}$  case, the transistors will also be connected into functional components just like in the  $2\ \mu\text{m}$  case (as shown in Figs. 1-3).

In the next section we simulate the components, properly loaded, with the SPICE simulator to find the delays.

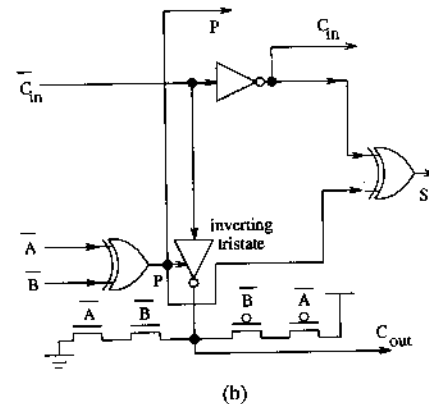
## III. SIMULATING THE COMPONENTS

We will discuss the simulation details using  $2\ \mu\text{m}$  technology, and then also give results for  $1\ \mu\text{m}$ , which was simulated in the same fashion.

We will first simulate the circuitry for skipping a block of bit positions. The skip circuit for a block is either a MUX1 multiplexer or MUX2 multiplexer, respectively, depending on whether the block to be skipped has an even or an odd number of bit positions.



(a)



(b)

Fig. 2. Ripple cells. (a) Even ripple cell. (b) Odd ripple cell.

Let us first consider the case of a MUX1. To measure (in simulation) the delay of a MUX1, we string ten of them together as shown in Fig. 4.

Each multiplexer except for the last one is loaded with the "if select = 0" input of the next multiplexer, as well as with a ripple cell. This load yields realistic delay figures, because an actual multiplexer would see exactly this load in an adder. The outputs of the ripple cells are loaded with reasonable RC loads. These loads are buffered away from the multiplexers because of the way the components are designed, and thus the delays of the multiplexers don't depend on the value of the load. In our test of the MUX1 chain, the select lines of all the multiplexers are 0, and the carry input line to the chain was raised linearly from 0 to 5 volts between  $t = 2\ \text{ns}$  and  $t = 3\ \text{ns}$ . The response times of the multiplexer output lines are as recorded in

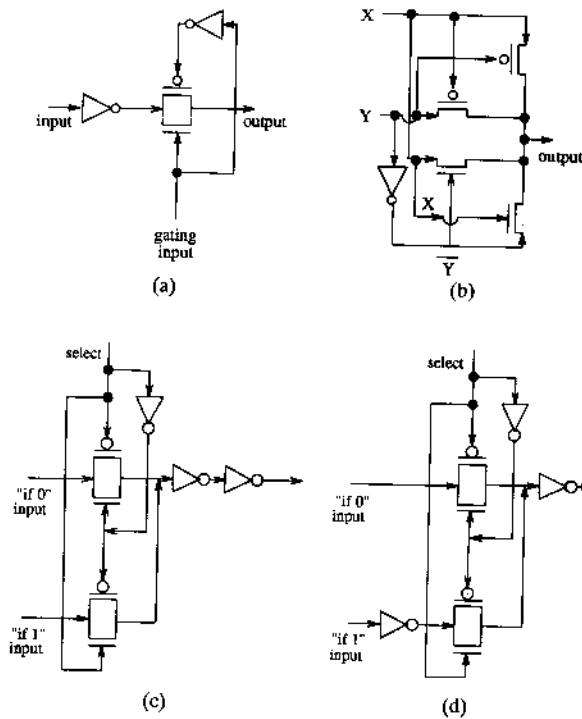


Fig. 3. Gates and multiplexers. (a) Inverting tristate. (b) XOR [8]. (c) MUX1. (d) MUX2.

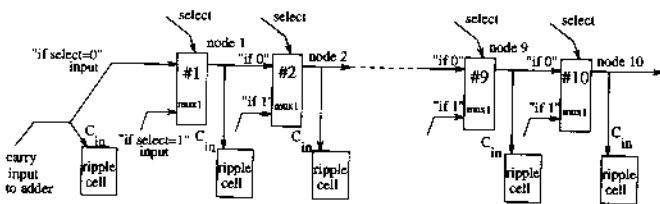


Fig. 4. Simulating a chain of ten MUX1's.

Fig. 6. To get the delay for each multiplexer, we simply subtract the time at which each multiplexer ends its voltage level change from the corresponding time of the multiplexer after it.<sup>2</sup> We conclude that each MUX1 has a delay of 1.24 ns.<sup>3</sup>

A chain of ten MUX2's was tested in the same fashion as the MUX1's. It turns out that the delay of each MUX2 is not much less than the delay of each MUX1. Thus we will use MUX1's delay, 1.24 ns, as MUX2's delay. This figure would be slightly too high.

<sup>2</sup>That the delay of the first multiplexer seems greater than the rest is simply because our input is a sharp ramp. In actual operation the input would probably not be a sharp ramp, and the first multiplexer's delay would be the same as that of the rest. In the 1  $\mu$ m case, where we care even more about accurate speed measurements, we will assume that all multiplexer delays are the same. Note also that the last multiplexer has a smaller delay because its load is light. We will disregard its delay.

<sup>3</sup>The timestep we used in our SPICE simulations in the 2  $\mu$ m case was 0.1 ns, and according to Fig. 6, some multiplexer delays are 1.2 ns and some are 1.3 ns. So we took the average over the 2nd through the 9th multiplexer. Keeping 3 significant figures, we obtained 1.24 ns. We also tested the same circuit with the input going from 5 to 0 volts. However, the delays were less than 1.24 ns, and so we will not use those test results. In the 1  $\mu$ m case we will size the transistor to obtain more equal up and down transition delays.

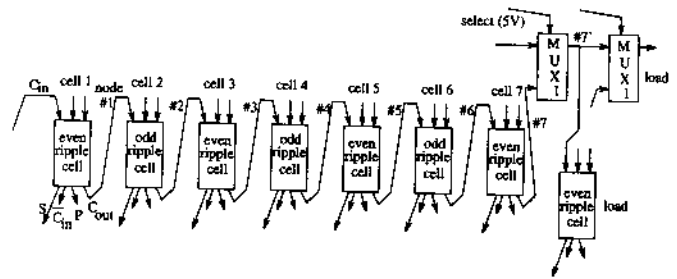


Fig. 5. Simulating a chain of seven ripple cells.

node number	begin change	end change	node number	begin change	end change
carry in	2.0ns (0V)	3.0ns (5V)	carry in	2.0ns (5V)	3.0ns (0V)
1	3.0 (0V)	6.2 (5V)	1	2.7ns (0V)	5.2ns (5V)
2	4.2 (0V)	7.4 (5V)	2	3.3ns (5V)	5.0ns (0V)
3	5.4	8.7	3	4.1ns (0V)	6.8ns (5V)
4	6.6	9.9	4	4.8ns	6.5ns
5	7.9	11.2	5	5.7ns	8.3ns
6	9.1	12.4	6	6.3ns	8.1ns
7	10.3	13.6	7	7.1ns (0V)	9.0ns (5V)
8	11.6	14.9			
9	12.8	16.1	7'	8.0ns (0V)	11.4ns (5V)
10	14.1	16.6 (light load)			

(a)

(b)

Fig. 6. SPICE simulation results for the 2  $\mu$ m case. (a) Test results for a string of ten MUX1's. (b) Test results for a string of seven ripple cells.

Next we test a string of ripple cells. For this test we connect in series seven ripple cells of alternating parity, as shown in Fig 5.<sup>4</sup>

The carry output of the last ripple cell is fed into a multiplexer, to simulate the end of a block of bit positions. The output of the multiplexer has a realistic load; namely, a ripple cell and another multiplexer. The test results are given in Fig. 6. Note that a pair of ripple cells possesses a delay of 1.55 ns, taking the average over several pairs. Note that the delay between the 5th ripple cell's output and the multiplexer's output is 3.1 ns, because the signal path is longer than just two ripple cells. This constitutes the delay of the last pair of cells plus the multiplexer.

To get a slightly-high figure for the delay of one ripple cell, we use the same string of ripple cells, but this time we insert an inverter between each adjacent pair of ripple cells. We found the average delay for 1 ripple cell (except the last one) to be 0.95 ns.

All the test results that will be used in the 2  $\mu$ m case are summarized in Table I.

Note that some of the delays are approximated and are a little too high, and we therefore would try to avoid using them. For example, if we have a pair of ripple cells in series, then we would use 1.55 ns, and not 0.95 + 0.95 ns.

Now we simulate components made from 1  $\mu$ m (drawn channel length) HP CMOS26B devices. The ripple cells are connected and tested like for the 2  $\mu$ m case, with results not quite 3 times as fast; that is, 0.55 ns delay for a pair. (The figure for the 2  $\mu$ m was 1.55

<sup>4</sup>It is not important to alternate the parity of the ripple cells in the simulation, since both even and odd cells obviously have the same delay through the carry path.

TABLE I  
SUMMARIZING THE SIMULATION RESULTS

Circuit Description	Delay (2 micron)	Delay (1 micron)
MUX1	1.24 ns	0.44 ns
MUX2, pair	2.03 ns (not used)	(not used)
MUX2, single	$\approx 1.24$ ns	$\approx 0.44$ ns
Ripple cell pair, not at end of block	1.55 ns	0.55 ns
Single ripple cell, not at end of block	$\approx 0.95$ ns	0.4 ns
Ripple cell pair, end of block (including MUX delay)	3.10 ns	0.8 ns
Single ripple cell, end of block (for 1-cell blocks)	$\approx 3.1$ ns (not used)	(not used)

ns.) For a ripple cell pair at the end of a bit block, the delay for the  $1 \mu\text{m}$  case is 0.8 ns. A single ripple cell not at the end of a block has a delay of 0.4 ns.

We will assume that the carry input signal is not an abrupt ramp, and so it is reasonable to say that all ripple cell pairs not at a block's end have the same delay of 0.55 ns. This assumption is better than the one used in the  $2 \mu\text{m}$  case.

A MUX1 (noninverting multiplexer) has a delay of 0.44 ns for an upward transition and 0.4 ns for a downward transition. So we will use the 0.44 ns figure. This again is not quite three times as fast as the  $2 \mu\text{m}$  case. Note that we sized the widths of the transistors to balance the upward and downward transition times to a certain extent. A MUX2's delay will be less than a MUX1's delay, and so we will just use MUX1's delay for a MUX2.

#### IV. DESIGN THEORY

The design theory for optimum one-level carry-skip adders that we will present here is a slight generalization of the one presented in an earlier version of this paper [9]. The generalization is needed because the time to skip a block and the time to ripple through a bit position are no longer constants.

The idea behind the main loop of the design process is the following:<sup>5</sup> Instead of asking *how  $n$  bit positions can be partitioned to yield an optimum-speed adder*, we first ask *how to build the biggest<sup>6</sup> possible adder for a given carry-propagation delay*. In other words, we will exhibit a procedure P that takes a number  $d$  as input and produces the biggest one-level carry-skip adder with carry propagation delay no more than  $d$ . Now, we can use procedure P to solve our original problem of designing the fastest adder for a specified number of bit positions  $n$  in the following way: for the specified value of  $n$ , search for the *minimum* carry propagation delay  $d$  such that the biggest adder with carry propagation delay  $d$  has at least  $n$  bit positions. This search for a minimum  $d$  may be conducted like a *binary search* or a *bisection*, which means the following: We know that  $d$  must lie somewhere between 0 and the carry propagation delay time  $D$  for  $n$  ripple cells. Thus the *search range* is now between 0 and  $D$ . Now let us use procedure P to determine whether the biggest adder with carry propagation delay up to  $D/2$  has at least  $n$  bit positions. If the answer is "yes," then we now know that the minimum  $d$  that we are looking for must be between 0 and  $D/2$ . But if the answer is "no," then  $d$  must be between  $D/2$  and  $D$ . Thus, in either case we would have narrowed the search range by a factor of 2. We can now repeat this iteration until  $d$  is found that is accurate enough for practical purposes.

<sup>5</sup>This idea itself is similar to the one used by Turrini [11]. However, the contents of the main loop itself are quite different here.

<sup>6</sup>Biggest = having the largest number of bits.

Before we continue, we note here that in order to optimize the adder speed, we only need to be concerned with the carry propagation delay. This is because in practice, it takes a very short time to set up the multiplexer select lines at the beginning of the addition process. It also takes a very short time to compute the adder outputs once the carries into all the ripple cells are known.

Now we need to describe the procedure P that specifies the bit blocks for the biggest adder with a specified carry propagation delay  $d$ .<sup>7</sup> Before going further, we will introduce a convenient representation of block sizes on a plane, which will be equivalent to the representations used in [8], [9]. The nonnegative X axis represents the block number, while the Y axis represents the number of bit positions in a block. We will number the least significant block with any integer  $i$ , and number the next block with the integer  $i + 1$ , and the next blocks would be numbered  $i + 2, i + 3$ , etc. Thus an adder can be represented as a set of markers that indicate the number of bits in a block, so that if block  $j$  has  $k$  bits, then block  $j$  would be represented with a marker at point  $(j, k)$ .

Our procedure P starts off by finding the largest integer  $m$  such that if we connect two  $m$ -bit blocks together, then the delay of a carry signal generated from the least significant bit (lsb) of the less significant block and terminated at the most significant bit (msb) of the more significant block is no more than  $d$ . Note that this carry signal has to go through a multiplexer between the two blocks. We will further describe the action of P by means of a typical example. In this example, we will use the component delay figures for  $2 \mu\text{m}$  CMOS presented in the last section (as summarized in Fig. 6). For this example, let's suppose  $d$  is 12 ns, and then obtain  $m$  by trial and error.<sup>8</sup> Is  $m = 6$ ? To answer that, recall that a pair of ripple cells normally have a delay of 1.55 ns except an end-of-block pair would have a delay of 3.1 ns. Thus the less significant block, being a block consisting of 3 pairs of cells, would have a delay of  $2 \times 1.55 + 3.1 = 6.2$  ns.<sup>9</sup> The more significant block consists of three pairs of ripple cells, hence the delay caused by that block (which doesn't include the multiplexer at the end) is  $3 \times 1.55 = 4.65$  ns. So if  $m = 6$ , then the total delay would be 10.85 ns, which is significantly less than  $d$ . So we ask if we could increase  $m$  to 7. If  $m$  is 7, then to each block we would have to add a delay of 0.95, which is the delay for a single ripple cell. But  $10.85 + 2 \times 0.95 = 12.75$ , which is more than  $d$ . So  $m$  is 6 and not 7.

P built this two-block adder because P will use these two blocks as the *nucleus* of the biggest one-level carry-skip adder with a carry propagation delay of at most  $d$ . But before we describe P's action any further, let us construct a representation of the two-block adder in the style described a little earlier. This representation would consist of two markers, one at the point  $(0, m)$ , and the other one at the point  $(1, m)$ , as shown in Fig. 7(a).

It is convenient to use this representation to represent also the *trips* of carry signals in a graphical way—the carry signal from lsb to msb in the two-block adder just mentioned can be thought of as making a trip from the marker at  $(0, 1)$ , then going down vertically six units (representing six ripple cells) to the X axis, then going through  $(0.5, 0)$  (which represents the output of the multiplexer), and finally

<sup>7</sup>In the case in which the skip and ripple times are constant, P consists of sliding an isosceles triangle on the X axis of a plane marked with "bit position markers" [9]. However, in the more realistic case in which we are considering, a more intricate process must be used. This process is so fast that it can be done by hand in a few minutes, and is still intuitively appealing.

<sup>8</sup>Clearly there is a way to obtain  $m$  systematically, but we will not consider such minor details here.

<sup>9</sup>Note that the carry signal originates at the lsb, and doesn't really propagate through it. However, the carry signal does experience some delay through lsb's circuitry, and thus practically speaking we might as well do our calculations as if the signal had propagated through the lsb.

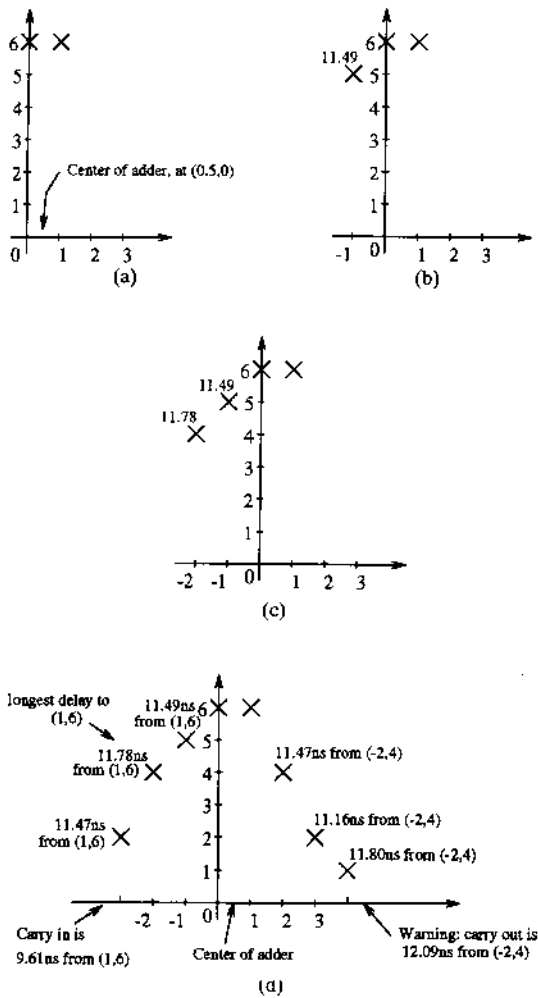


Fig. 7. Illustrating how P works. (a) Markers representing the nucleus of the adder. (b) Adding a new block to the left of the nucleus. (c) Adding yet another block to the left of the nucleus. (d) Adding as many blocks as possible to both sides of the nucleus.

going vertically upwards from (1,0) to the marker at (1,6). The point (0.5,0) shall be called the *center* of the adder.

Now we will continue to describe P's actions. Now that P has created the adder's nucleus, P will try to add blocks to the left and to the right of the nucleus. P must add blocks that are as big as possible without making the worst delay path longer than  $d$ . What if P adds a 5-bit block to the less significant end of the adder? This new block would be represented by a marker at point (-1,5), as shown in Fig 7(b). Consider the carry path from the adder's new lsb to the msb. This carry path first goes through a ripple cell plus two cell pairs, one of which ends with a multiplexer. This part of the trip takes  $0.95 + 1.55 + 3.1 = 5.6$  ns, and takes it to the point (-0.5,0), which represents the output of the multiplexer just mentioned. The next part of the trip involves skipping a block (the left block of the nucleus) via a MUX1, which takes 1.24 ns, and brings the signal to the center (0.5,0) of the adder. The rest of the trip is a ripple through three pairs of ripple cells without a multiplexer at the end, which takes 4.65 ns. So the total delay is  $5.6 + 1.24 + 4.65 = 11.49$  ns, which is less than  $d$ . If we calculate the lengths of other carry paths in the adder so far, we will also find that they are all no more than  $d$ .

P can usually add even more blocks to both sides of the nucleus and not increase the worst-case carry delay to more than  $d$ . There are

many ways in which P can go about adding blocks. One good way is to continue adding the biggest possible blocks to the left until no more blocks to the left can be added. Fig. 7(c) shows the first such block. The delay from the lsb of this block to the adder's msb is 11.78 ns, and we note this in Fig. 7(c) with the number 11.78. The left half of Fig. 7(d) shows all the left blocks.

After all the left blocks have been added, P will add the biggest possible blocks to the right of the nucleus. To do this, first note that the worst delay path of the adder P has obtained so far is from the point (-2,4) to the point (1,6). To ensure that each new (right) block is not too big, it suffices to make sure that it is short enough so that the path from (-2,4) to the msb of the block is limited by  $d$ . The resulting adder is shown in Fig 7(d). Note that the path from (-2,4) to the carry output of the entire adder would be 12.09, slightly larger than  $d$ , if the multiplexer at this output has the same load as the other multiplexers in the adder. However, this multiplexer may have a lighter load, and hence a smaller delay than the other multiplexers. If this multiplexer is very slow, however, then we would omit the rightmost block of one ripple cell.

The adder in Fig. 7(d) that we have just built is, for practical purposes, just about the biggest adder possible with delay  $d$  or less. Stating this more precisely, we can't get a bigger adder of delay  $d$  or less if we keep the same nucleus. However, if we are willing to change the nucleus, then we *might* be able to pack in very few more bits while maintaining a worst-case delay of  $d$ . However, in order to add bits to one side of the adder, we would have to *reduce* the size of the nucleus of the other side by one or two bits, and we may also have to reduce the size of some other blocks on that other side as well. For this reason, we conclude that this final adjustment of blocks by taking away bits from one side and adding some bits to the other sides is of no practical value. Note that we can't take away too many bits from either side of the adder, because doing so would make the adder unbalanced. And an unbalanced adder can't be the biggest adder for its delay, because we would be able to add bits to the scanty side and not increase the delay.

Thus we have built an 30-bit adder with 12 ns carry propagation delay using the  $2 \mu\text{m}$  nss and pss transistor models from [5]. Thirty-bit adders are sometimes used in address generation in a CPU's control unit.

Now we discuss the  $1 \mu\text{m}$  case. Given the simulation results in Fig. 6, we would expect a 30-bit adder in  $1 \mu\text{m}$  CMOS to be about three times as fast as a  $2 \mu\text{m}$  one. Thus we would expect a 64-bit adder in  $1 \mu\text{m}$  to be just a bit slower (since the adder delay is a square-root function of the number of bits). Indeed this is the case. Using the procedure discussed above, we can construct a 64-bit adder shown in Fig. 8, where each block's marker is labeled with the maximum delay between that block and the center of the adder. The worst-case carry propagation delay is thus  $3.0 + 2.78 = 5.78$  ns. We found out by SPICE simulation that 0.45 ns more is needed to obtain the msb of the sum, and thus the total worst-case adder delay is 6.23 ns, not including output buffering.<sup>10</sup>

## V. CONCLUSION

We presented a new procedure for designing optimum one-level carry-skip adders. This procedure departs from existing ones in the sense that it allows the use of realistic component delay figures obtained by simulation. This departure makes it more likely that the adders obtained are optimum in practice. We gave two examples, a 30-bit adder in  $2 \mu\text{m}$  CMOS and a 64-bit adder in  $1 \mu\text{m}$  CMOS. The former adder has a 12ns carry propagation time, while the latter has a 6.23 ns addition time.

<sup>10</sup>Output buffering is needed in some applications, and takes amounts of time that varies according to the implementation.

## On Asymmetric Invalidation with Partial Tests

Vijay Raghavan

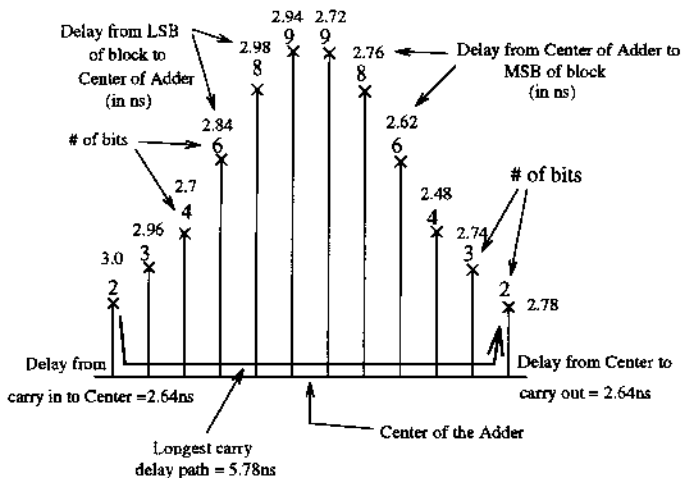


Fig. 8. Diagram representing a 64-bit one-level carry-skip adder in  $1\ \mu\text{m}$  (drawn channel length) CMOS, with 6.23 ns addition time.

## ACKNOWLEDGMENT

The author would like to thank the anonymous referees and the editor of this paper (Prof. P. Kornerup) for suggestions on how to strengthen the results of this paper, so that the main result is now of real practical value.

## REFERENCES

- [1] T. Lynch and E. E. Swartzlander, Jr., "A spanning tree carry lookahead adder," *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 931-939, Aug. 1992.
- [2] T. Lynch and E. E. Swartzlander, Jr., "The redundant cell adder," in *Proc. 10th IEEE Symp. Comput. Arith.*, Grenoble, France, 1991.
- [3] *The Origins of Digital Computers*, 3rd ed., B. Randell, Ed. New York: Springer-Verlag, 1982.
- [4] Anonymous Referee "C" (private communication), 1992.
- [5] M. Lehman and N. Burla, "Skip techniques for high-speed carry propagation in binary arithmetic units," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 691-698, Dec. 1961.
- [6] P. K. Chan and M. D. F. Schlag, "Analysis and design of CMOS manchester adders with variable carry-skip," *IEEE Trans. Comput.*, vol. 39, pp. 983-992, Aug. 1990; an earlier version appeared in *Proc. 9th IEEE Symp. Comput. Arithmetic*, 1989.
- [7] P. K. Chan, M. D. F. Schlag, C. D. Thomborson, and V. G. Oklobdzija, "Delay optimization of carry-skip adders and block carry-lookahead adders using multidimensional dynamic programming," in *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 920-930, Aug. 1992.
- [8] A. Guyot, B. Hochet, and J.-M. Muller, "A way to build efficient carry-skip adders," *IEEE Trans. Comput.*, vol. C-36, pp. 1144-1151, Oct. 1987.
- [9] V. Kantabutra, "Designing carry-skip adders," in *Proc. 10th IEEE Symp. Comput. Arithmetic*, Grenoble, France, 1991.
- [10] V. G. Oklobdzija and E. R. Barnes, "Some optimal schemes for ALU implementation in VLSI technology," in *Proc. 7th Symp. Comput. Arithmetic*, 1985.
- [11] S. Turrini, "Optimal group distribution in carry-skip adders," in *Proc. 9th IEEE Symp. Comput. Arithmetic*, 1989.
- [12] L. Glasser and D. Dobberpuhl, *The Design and Analysis of VLSI Circuits*. Reading, MA, Addison-Wesley, 1985.

**Abstract**—Gupta and Ramakrishnan proposed a general model for fault diagnosis which uniformly handles intermittent faults, masking, and partial testing. In this model, the diagnosability problem is open if asymmetric invalidation is used, although a polynomial time algorithm was once incorrectly claimed. We show that the problem is really co-NP complete. We also give an efficient diagnosis algorithm for this model.

**Index Terms**—Asymmetric invalidation, co-NP completeness, diagnosability, partial testing.

## I. INTRODUCTION

System-level fault diagnosis has its origins in a seminal paper by Preparata, Metze, and Chien [1]. The paper introduced a simple graph-theoretical model for fault diagnosis, which has been the subject of intense research in the past twenty years. The model has undergone many changes to accommodate intermittent faults [2], partial testing [3], fault masking [4] and practical restrictions which lead to asymmetric invalidation [5] of test results. This has given rise to many offshoots of the original model. Recent research has focused primarily on probabilistic diagnosis algorithms [6], [7] diagnosis for systems with a regular structure or sparse interconnection [8], [9]. Some of the recent work has been in issues of computational complexity arising from the search for efficient diagnosis and diagnosability algorithms under different measures of diagnosability [10]-[13]. A good survey of the research in this area can be found in [14].

Many different approaches have been taken to unify this proliferation of models. One approach, due to Somani and Agarwal [15], was to provide a characterization theorem that is generally applicable for symmetric and asymmetric invalidation under both permanent and intermittent faults. Blough, Sullivan, and Masson [8] and Rangarajan and Fussell [7] model faults and test results probabilistically and show that highly reliable diagnosis can be achieved in systems with relatively sparse testing connections. Yet another approach, and the one we are concerned with in this paper, is due to Gupta and Ramakrishnan [16]. They use a single parameter, called the test multiplicity, that can be set to different values to handle intermittent faults, partial testing and fault masking. This generalized framework therefore handles many of the features of earlier models in a uniform manner.

Gupta and Ramakrishnan deal primarily with symmetric invalidation, although they claimed a polynomial time algorithm for diagnosability when asymmetric invalidation is used in their generalized framework. This claim was later withdrawn [17] and the problem is mentioned as being open in Gupta's Ph.D. thesis [18]. We show here that the problem is really co-NP complete, in spite of the fact that the diagnosability number must be one of only two possible values. We also give an efficient diagnosis algorithm for this model.

This paper is organized in four sections, including the introduction. Section II develops the formal concepts used in the model and defines

Manuscript received April 24, 1989; revised January 26, 1991 and September 30, 1992.

Dr. Vijay Raghavan is with the Computer Science Department, Vanderbilt University, Nashville, TN 37235.

IEEE Log Number 9207297.