A Design for an Efficient NOR-gate only, Binary-ripple Adder with Carry-completion-detection Logic

A novel way of designing a ripple-carry adder is presented. The new design uses only NOR gates. It is both economical, using only six NOR gates per stage, and efficient, since a carry-completion-detection circuit can easily be integrated into it at a cost of (approximately) one NOR gate per two adder stages.

Received January 1986, revised September 1986

1. Introduction

Adders are important components of arithmetic units. Many designs have been developed described in the literature,[1, 2] and used in actual computers. The ripple-carry (or parallel) adder (fig. 1) is one of the oldest and simplest designs.[3] It consists of n identical stages, each a full adder, with carry propagation from each stage to the next.

Each stage receives two input bits a_i, b_i, and a carry bit c_{i-1} from the preceding stage. It produces a sum bit s_i and a carry bit c_i according to Table 1.

$$s_i = a_i' b_i' c_{i-1} + a_i' b_i c_{i-1}' + a_i b_i' c_{i-1}' + a_i b_i c_{i-1}$$

$$c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1} = a_i b_i + (a_i + b_i) c_{i-1}$$

The ripple carry adder is not parallel, since each stage must wait for the carry from its predecessor. A major improvement was suggested by Gilchrist *et al.*[4] They proposed a carry-completion-detection circuit (fig. 2) that outputs a signal (carry-completed) when all stages have generated their carries.

When all adder stages have settled down, some stages will produce a carry (1-carry) and others will produce no carry (0-carry). To detect completion it is necessary to indicate the presence of either a 1-carry or a 0-carry from each stage.

Table 1

a_i	b_i	c_{i-1}	s_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The two quantities

$c1_i = a_i b_i + (a_i' b_i + a_i b_i') c1_{i-1}$ (the 1-carry or normal carry)

$c0_i = a_i' b_i' + (a_i' b_i + a_i b_i') c0_{i-1}$ (the 0-carry, or absence of a carry)

indicate either the presence of a carry ($c1$) or its absence ($c0$). A detailed discussion of carry-completion detection can be found in Lewin,[5] but two facts should be noted here.

(1) The carry-completed gate must have a fan-in of n.

(2) The average maximum carry length has been estimated by Hendrickson[6] to be $\log_2(5n/4)$.

Thus the average speed of the entire adder is $D \log_2(5n/4)$, where D is the carry-propagation time of a single stage.

The design presented here is for a ripple-carry adder to which a simple, fast, carry-completion-detection logic can easily be added. The main features of the design are as follows.

(1) Only NOR gates are used, with only six gates per stage.

(2) Almost all signals necessary for carry-completion detection are automatically generated by each stage. The only additional logic necessary for carry-completion detection is one NOR gate for each pair of stages, plus another NOR gate for the entire adder. This translates to approximately half a gate per stage.

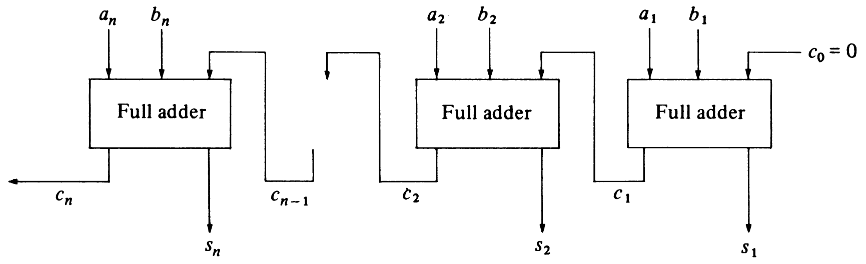

Figure 1. A ripple-carry adder.

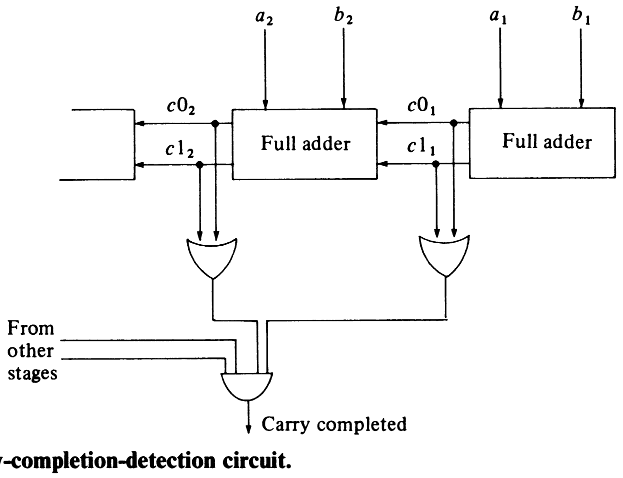

Figure 2. Carry-completion-detection circuit.

2. A Single Adder Stage

The design is based on the following logical relations, derived from Table 1.

$$\begin{aligned} s_i &= a_i' b_i' c_{i-1} + a_i b_i c_{i-1} + a_i' b_i c_{i-1}' + a_i b_i' c_{i-1}' \\ &= (a_i' b_i' + a_i b_i) c_{i-1} + (a_i' b_i + a_i b_i') c_{i-1}' \\ &= p_i' c_{i-1} + p_i c_{i-1}' = (p_i c_{i-1} + p_i' c_{i-1}')' \\ &= [p_i c_{i-1} + p_i'(a_{i-1} b_{i-1} + p_{i-1} c_{i-2})']' \\ &= [p_i c_{i-1} + (p_i + a_{i-1} b_{i-1} + p_{i-1} c_{i-2})']' \\ &= [p_i c_{i-1} + (p_i c_{i-1}' + a_{i-1} b_{i-1} + p_{i-1} c_{i-2})']' \end{aligned} \quad (1)$$

Also from Table 1, $c_i = a_i b_i + p_i c_{i-1}$. Thus $c_{i-1} = a_{i-1} b_{i-1} + p_{i-1} c_{i-2}$ and

$$\begin{aligned} p_i c_{i-1} &= (a_i' b_i + a_i b_i')(a_{i-1} b_{i-1} + p_{i-1} c_{i-2}) \\ &= (a_i' b_i + a_i b_i')(a_{i-1}' b_{i-1}')'(p_{i-1} c_{i-2})' \\ &= (a_i b_i)'(a_i' b_i')'(a_{i-1}' b_{i-1}')'(p_{i-1} c_{i-2}')' \\ &= (a_i b_i + a_i' b_i' + a_{i-1}' b_{i-1}' + p_{i-1} c_{i-2}')' \end{aligned} \quad (2)$$

Similarly

$$p_i c_{i-1}' = (a_i b_i + a_i' b_i' + a_{i-1} b_{i-1} + p_{i-1} c_{i-2})' \quad (3)$$

Fig. 3 shows a single adder stage. It is clear that producing the sum s_i requires two NOR gates (equation 1), producing $p_i c_{i-1}$ and $p_i c_{i-1}'$ (equations 2 and 3) requires two more gates, and producing p_i (rather a_i and b_i) requires the other two gates.

3. The Complete Adder

Fig. 4 shows the complete adder with carry-completion detection. The carry-completion-detection circuit is based on signals T_i defined by

$$\begin{aligned} T_i &= (c0_i + c1_i)' \\ &= (a_i b_i + p_i c_{i-1} + a_i' b_i' + p_i c_{i-1}')' \end{aligned} \quad (4)$$

which are implemented for every other stage. Thus $i = 2, 4, 6, \ldots, m$ where $m = 2\lfloor n/2 \rfloor$. The carry-completed signal is $T = (T_2 + T_4 + \ldots + T_m)'$. It goes high when all T_i go low.

An additional control signal t is necessary to prevent false indication of carry completed. It is included in the T_i signals whose definition thus becomes

$$T_i = (a_i b_i + p_i c_{i-1} t' + a_i' b_i' + p_i c_{i-1}' t'')' \quad (5)$$

Initially t is set high (to indicate 'carry not completed'), and the numbers to be added a_i, b_i are fed into all the stages. It takes two propagation delays of a NOR gate ($2D$) to generate $p_1 c_0$ and $p_1 c_0'$ in the first stage, at which point the carry-propagation process starts and t should be switched low. With t low, the T_i signals depend on $p_i c_{i-1}$ and $p_i c_{i-1}'$ as in equation 4. When all the stages have stabilised, each stage outputs either $c1_i$ high or $c0_i$ high, and all T_i become low.

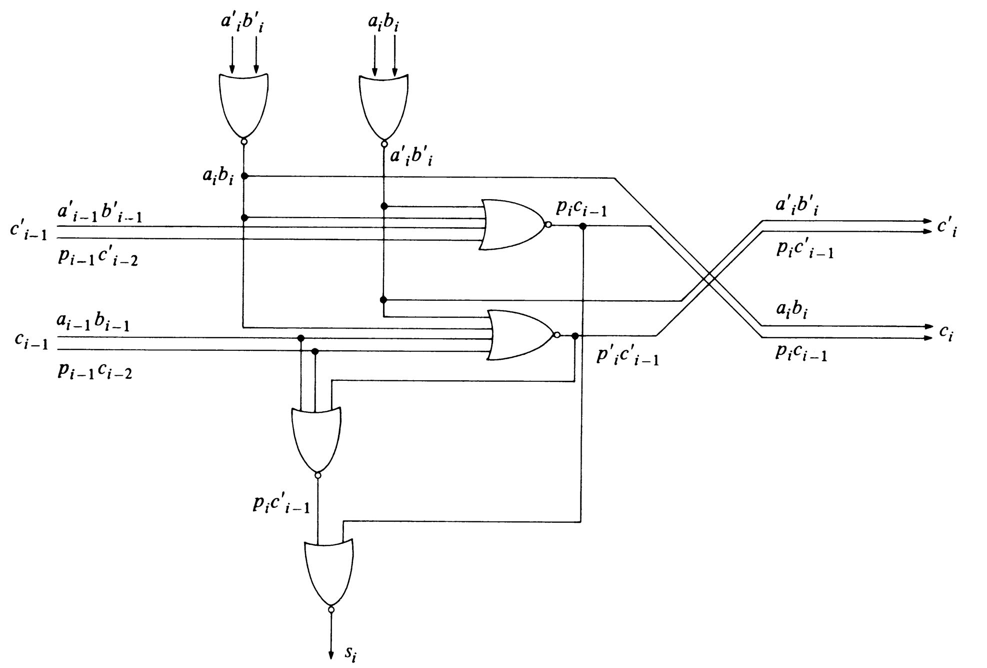

Figure 3. A single stage.

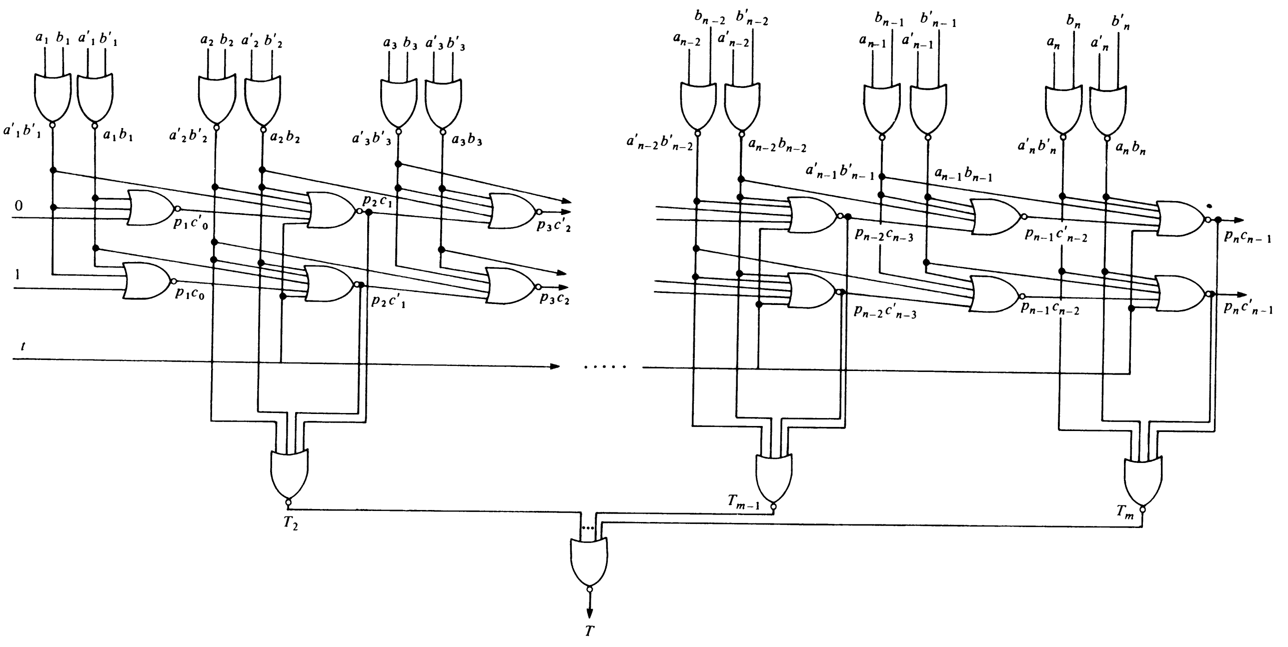

Figure 4. The complete adder.

4. Notes

(1) The T_i signals are only generated by every other stage for economy reasons. This, however, implies that the carry-completed signal T may be generated too early. After T is generated there is still the possibility that a stage i that does not generate a T_i will change its output carry c_i. The actual carry completion would, in such a case, take place D time units after T is generated. This would normally cause no problem, since the sum typically remains in the adder until the next clock pulse and is not moved out of the adder immediately. If this is a concern, however, the design should be changed so that every stage generates a T_i.

(2) The maximum fan-in in Fig. 4 is $\lfloor n/2 \rfloor$. If fan-in is a concern, the adder can be designed with a T_i generated for every kth stage. The number of T_i signals (and thus the maximum fan-in) in such a case would be $\lfloor n/k \rfloor$. In such an adder, however, T could precede the actual carry completion by up to $(k-1)D$ time units.

Another solution to a fan-in problem is replacing the single NOR gate at the bottom of Fig. 4 with several gates. Still another solution is to use wired logic instead of gates, but this is possible only if $T_2, \ldots, T_{m-1}, T_m$ are open-collector.

(3) A look at Fig. 4 shows that the design is modularised. Each group of two successive stages can be considered a module. Such design can easily be described as an iterative array and thus lends itself to easy implementation in VLSI.

(4) The adder proposed here is self-timed (asynchronous) and the author considers this an advantage. In wafer-scale integration (WSI) there is a need for self-timed circuits because it is hard to bring clock signals to the centre of the die. The clock signals get skewed when passing through a wafer due to the large diameter (3–4 inches) of the wafer. This is why experts predict more asynchronous circuits in the future.

(5) It is common practice to add another control signal E to such a circuit to completely disable it. Fig. 5 shows how the design can

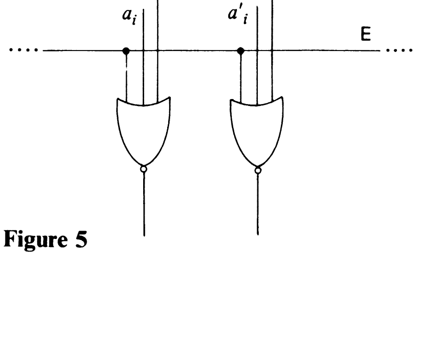

Figure 5

easily be modified to include such a signal. When E = 1 the circuit is disabled. After feeding-in the inputs, E is pulled down to zero, enabling the device.

Acknowledgement
The author would like to thank Dr Robert Burger for his help in this research.

D. SALOMON
Computer Science Department, California State University, Northridge, CA 91330, USA.

References

1. M. Abd Alla and A. Meltzer, *Principles of Digital Computer Design*. Prentice-Hall, Englewood Cliffs, N.J. (1978).
2. J. Cavanagh, *Digital Computer Arithmetic*. McGraw-Hill, New York (1984).
3. J. Cavanagh, *Digital Computer Arithmetic*, p. 105.
4. B. Gilchrist, J. H. Pomerence and S. Y. Wong, *Fast Carry Logic for Digital Computers. IRE Trans. on Electronic Computers*, vol. EC-4 (Dec. 1955), pp. 133–136.
5. D. Lewin, *Theory and Design of Digital Computers*. Halstead Press, New York (1972).
6. H. C. Hendrickson, *Fast, High-Accuracy Binary Parallel Addition. IRE Trans. on Electronic Computers*, vol. EC-9 (Dec. 1960), pp. 465–469.

Correspondence

Leith and Legal Logic Programming

Dear Sir,

In his article 'Fundamental errors in legal logic programming' (vol. 29, no. 6, 1986), Philip Leith attacks our use of PROLOG to formalise the provisions of the British Nationality Act 1981. The basis of his attack is the claim that we identify the legal process with the rigid, formal application of rules embodied in legislation. Leith is quite mistaken. Nowhere in our original draft or in the later published paper[3] do we ourselves make such a claim. In the paper we state explicitly that our British Nationality Act program can only be used to determine what follows if the rules of the Act are applied literally. Since there is more to legal reasoning than the literal application of the letter of the law, we did not propose that our program could be used to decide questions of British citizenship autonomously. On the contrary, we explain why it could not be used for this purpose, except in quite unrealistic circumstances. Elsewhere we have consistently emphasised the need for embedding the use of logic within a flexible framework for assimilating knowledge, for revising beliefs and for comparing alternative systems of belief.[1, 2, 4] It is precisely because reasoning in law demands such great flexibility that we believe it is an ideal domain in which to test the application of these developing techniques. We do contend, however, that systems like our British Nationality Act program can be of substantial use, even though they address only one relatively trivial aspect of the whole legal process. Some of these uses are described in our paper. The possibilities are argued at greater length in more general accounts of our work.[4]

Leith's central contention seems to be that there is no such thing as a clear legal rule, or more accurately that there is no set of circumstances in which a legal rule could be applied routinely. In making such a claim, Leith is attacking not simply the use of logic programming techniques in law, but the very idea that computer programs of any kind could ever be used in the routine administration of law, whether these programs are written in PROLOG, in FORTRAN, in COBOL, or whatever. If Leith is right, then there are no clear legal rules. The mistake must be in thinking that there are, and not in choosing some particular programming language to express them. However, in the day-to-day practice of law, there are mundane and routine tasks that have to be performed. That is why there are computer programs, like payroll systems for example, which are used every day to perform these tasks. To suggest that the administration of law can be reduced to some routine application of fixed legal rules is a massive oversimplification. To suggest that legal rules are never applied routinely is to oversimplify to the opposite extreme.

Philip Leith's attack seems to be based on a mistaken impression of our work. We would argue that it is also based on a mistaken impression of the jurisprudential material which he cites. Less easy to overlook, however, is the general tone of the article. We take particular objection to several quite outrageous claims that are made in the article, none of which Leith has bothered to substantiate. Leith draws attention to 'racist implications' of the British Nationality Act and suggests that our work gives support to the racist cause. He goes so far as to hint that this might explain in some part the funding that our work on logic programming has attracted. If the Act is indeed racist, then a rigorous derivation of its logical consequences can only make its racist character more apparent. In fact, it was precisely such ethical considerations that contributed to our initial choice of the British Nationality Act project.

Philip Leith concludes

> 'I believe that I have cast substantial doubt on the claimed success of the Imperial team in their use of logic programming in law. The next question should be whether the team have made similar claims in other areas outside computer science which are open to the same challenge.'

Leith is, of course, entitled to uncover and expose errors in our work. We welcome such challenges when they are based on technical considerations. However, articles such as this one by Leith, with its personal overtones, contribute little to what would otherwise be an important and stimulating debate.

Yours faithfully

R. KOWALSKI and M. SERGOT
Imperial College of Science and Technology, University of London,
180 Queen's Gate, London SW7 2BZ

References

1. R. A. Kowalski, *Logic for Problem Solving*. North Holland-Elsevier, New York (1979). (See especially Chapter 13.)
2. R. A. Kowalski, *Logic-based Open Systems*. Department of Computing, Imperial College (1985).
3. M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond and H. T. Cory, The British Nationality Act as a logic program. *Comm. ACM* **29** (5) (1986).
4. M. J. Sergot, *Representing Legislation as Logic Programs*. Department of Computing, Imperial College (1985). To appear in *Machine Intelligence* 11, Oxford University Press.

Editor's note:

The above letter draws attention to a comment made in the paper by Leith that there are racist implications in the British Nationality Act. Authors and potential authors should remember that *The Computer Journal* is devoted solely to reports of new technical developments in computer science and computer applications. In general, items which are not directly relevant to the technical aspects of work undertaken should not be included in papers submitted to the *Journal*.

Ada's Fixed-point Types

Dear Sir,

I feel that I must make a comment on the fragment of Ada which appears in A. J. Cowling's paper on type checking in *The Computer Journal*, **29** (6) 541, where the fixed-point type 'currency' is declared. (A minor point is that 'delta' should be in the bold font, not in italics.) I believe that this fragment perpetuates a misunderstanding about Ada (which fortunately in no way invalidates the technical content of the published paper), which I should like to clear up.

Currency comes in indivisible units. If I divide £100 between 3 Scotsmen, each gets £33.33 and I keep 1 penny. If I divide 1 000 000 lire between 3 Italians each gets 333 300 lire and