ALGORITHMS FOR PARALLEL ADDITION AND PARALLEL POLYNOMIAL EVALUATION

Christos A. Papachristou

Department of Electrical and Computer Engineering

University of Cincinnati

Cincinnati, Ohio 45221

ABSTRACT

This paper presents two related algorithms for implementing parallel n-bit binary addition and evaluating n-th degree polynomials, respectively. The approach taken makes use of an iterative construction, the computation tree. The algorithms are particularly effective for moderate values of n and are in accord with well-known asymptotic bounds. In the case of n-bit addition, the implementations constitute lookahead tree circuits of r-input standard logic elements. Extensions to modular tree structures for lookahead adders are also considered. In the case of parallel polynomial evaluation, the operations of ordinary addition and multiplication are assumed with the capability to employ r arguments simultaneously.

I. INTRODUCTION

It is known that the carry propagation time is a limiting factor on the speed with which two binary numbers are added in parallel [1]. Many techniques for reducing the carry propagation time of n-bit binary adders are described in the literature [2-4]. The most widely used method employs the concept of lookahead carry [5]. This, in principle, can produce all n carries within two logic levels at the expense of increased hardware complexity. The latter may no longer be an issue because of the rapid progress in integrated circuit (IC) technology. However, existing fan-in limitations severely restrict two-level n-bit lookahead adders to $n \leq 8$. Multilevel implementations have been considered in [6] on the basis of a simple splitting of the lookahead adder into groups. A more sophisticated partitioning was used in [7] to find an asymptotic estimate on the addition time. However, the latter is far from the best estimate for moderate values of n. General bounds on the time required to perform addition are given in [8] and [9]. A method for implementing lookahead addition as a first order linear recursion is developed in [10].

In this paper we first present a technique for efficient implementation of n-bit carry lookahead addition as far as both time and complexity are concerned. The implementations are tree circuits which may consist of 1) either standard gates with limited fan-in and fan-out, or 2) carry lookahead modules with limited number of terminals (pins). Our method is based on a partition of the lookahead adder with the aid of a computation tree at level t+1, on the basis of the trees at level $\leq$ t, inductively. The proposed technique is consistent with existing asymptotic bounds for addition. However, our method is particular effective when the number of bits is rather moderate, which usually occurs in practice.

Since the carry lookahead formula resembles a polynomial-like expression, it is not difficult to show that the previous method, after some modification and notation adjustments, applies to the problem of parallel evaluation of polynomials. The results obtained by this approach appear to provide the greatest known degrees of polynomials that can be evaluated in given steps.

II. FORMULATION OF THE METHOD

We first need to recall the essentials of carry lookahead addition [5-6]. Let $A = a_1,\ldots,a_n$ and $B = b_1,\ldots,b_n$ be two n-bit binary numbers with a_1, b_1 being their most significant bits, respectively. The i-th carry-generate and carry-propagate functions are respectively defined by

$$g_i = a_i b_i \quad \text{and} \quad p_i = a_i \oplus b_i \; ; \; i = 1,\ldots,n$$

Then, the i-th stage carry and sum bits may be expressed by the following iterative relations (see Fig. 1)

$$c_i = g_{n-i+1} + p_{n-i+1} c_{i-1}$$

$$s_{n-i+1} = p_{n-i+1} \oplus c_{i-1} \qquad (1)$$

where i=1,...,n and $c_0 = 0$. Consequently, the n-th stage carry is given by (2)

$$c_n = g_1 + p_1 g_2 + p_1 p_2 g_3 + \ldots + p_1 p_2 \cdots p_{n-1} g_n$$

Clearly the n-bit addition time depends entirely on the depth, i.e., the number of logic levels required to implement the carry c_n. For small n, e.g., $n \leq 6$, it is not difficult to achieve constant addition time by implementing all carries in 2-level logic. However, this is not feasible to do when n has rather moderate values, e.g., $n > 8$, due to fan-in limitations. For high speed addition we will consider parallel implementation

of the carries c_i i=1,...,n by multilevel tree circuits composed of standard gates (AND, OR, EX-OR, etc.) with fan-in r, where r < n.

To formulate our method we need the following definitions and notation. Given two integers q and m, m ≥ q, the generating and propagating functions, G(q,m) and P(q,m), respectively, are defined as follows

$$G(q,m) = g_p + p_q g_{q+1} \cdots p_{m-1} g_m$$
$$P(q,m) = p_q p_{q+1} \cdots p_m \quad (3)$$

We define the size of G(q,m) (or P(q,m)) to be equal to m-q+1, i.e., the number of integers in the closed interval q ≤ m. Generating (or propagating) functions of equal size are equivalent in the sense that they may be implemented by the same circuit by a trivial relabeling of the input terminals. It is clear from the preceding definitions that the addition carries are related to the generating functions as follows

$$c_k = G(n-k+1,n) \quad k=1,\ldots,n \quad (4)$$

therefore, the addition carry c_k is equivalent to the 1-oriented generating function G(1,k) with

$$c_n = G(k,n) \quad (5)$$

Evidently it is sufficient to restrict our attention to implementing 1-oriented generating functions. Straightforward multilevel implementation of G(1,n) by r-input gates requires $2\lceil \log_r n \rceil$ logic levels [11]. Note $\lceil \alpha \rceil$ ($\lfloor \alpha \rfloor$) represents the greatest(smallest)integer ≤(≥) α, respectively. The objective of our method is to achieve considerable reduction in logic depth by appropriate restructuring of the lookahead formula G(1,n). First we will establish decomposition properties for generating functions, next.

Lemma 1: For any set of integers $s_1, s_2, \ldots s_m$ m ≥ 2, with $s_1 < s_2 < \ldots < s_m$ we have

$$G(s_1,s_m) = G(s_1,s_2) + P(s_1,s_2)G(1+s_2,s_3) + \ldots + P(s_1,s_{m-1})G(1+s_{m-1},s_m) \quad (6)$$

Proof: It can be easily proven by induction on m.

Let $T_\alpha G(1,k)$, k≥1, denote the depth required to implement G(1,n) with r-input gates by an algorithm or scheme α. Then we have

Lemma 2: For m >0, $T_\alpha G(1,m+1) \leq T_\alpha G(1,m) + 1$

Proof: From Lemma 1, G(1,m+1) = G(1,m) + P(1,m)G(m+1,m+1); it is sufficient to show that, under scheme α, P(1,m)G(m+1,m+1) = V requires no more depth than G(1,m) does. Clearly, V is a conjunction with length, i.e., number of literals, equal to m+1; V can thus be implemented in depth $\lceil \log_r(m+1) \rceil$ levels of r-input gates. Since G(1,m) contains 2m independent variables $T_\alpha G(1,m) = \lceil \log_r(2m) \rceil$ (see [5]). It can now be easily shown by induction on m that $\lceil \log_r(2m) \rceil \geq \lceil \log_r(m+1) \rceil$ for m >0, which proves this lemma.

As a result of the previous lemma, the set of generating functions may be partitioned in equivalence classes $\Gamma_0, \Gamma_1, \ldots, \Gamma_t, \ldots$ such that Γ_t contains all generating functions realizable in depth t by r-input gates under scheme α. With the exception of Γ_1, the equivalence classes defined above are non-empty; Γ_1 is empty since every generating function with size >1 requires at least two logic levels under any realization scheme.

The class Γ_t is characterized by the maximum size N_t among all sizes of the generating functions which are in Γ_t. It is convenient to label N_t maximal size for level t, or simply t-maximal. A function with size N_t, for example, $G(1,N_t)$ or $G(1+N_t,2N_t)$, is called prime for level t, or simply t-prime. Our proposed method is inductive in that it produces the (t+1)-prime function $G(1,N_t)$ on the basis of the already known implementations of the lower level k-primes, k ≤ t. Furthermore, using only the primes for level lower than t, non-prime generating functions can also be produced during the implementation of $G(1,N_t)$. The details of this process are described next.

III. CONSTRUCTION PROCESS

We first note that it is not difficult to find the 1-oriented 2-prime and 3-prime functions for a given fan-in r. For example, for r=2 they are G(1,2) and G(1,3), respectively; for r=3, G(1,3) and G(1,5). To compute $G(1,N_{t+1})$ we will attempt to restructure the basic formula (6) in order to satisfy the following conditions: 1) the righthand side of (6) should contain only prime functions; 2) the only t-prime to be used is $G(1,N_t)$, labeled head prime; 3) the rest of the primes are each ANDed with an appropriate propagating function such that the size of the propagating functions associated with a j-prime, j < t, should be ≤ r^j.

As a result of the preceding conditions the lowest level prime function that can be used for constructing $G(1,N_{t+1})$ is equivalent to $G(1,N_d)$ where $d = \lceil \log_r N_t \rceil$. This statement can be shown using formula (6), as follows: $G(1,N_{t+1}) = G(1,N_t) + P(1,N_t)G(1+N_t,N_t+N_d) + \ldots$ Clearly, $G(1+N_t,N_t+N_d)$ is equivalent to $G(1,N_d)$ and, hence, d-prime. According to the third condition above, $P(1,N_t)$ should have size $N_t \leq r^d$, and thus $d = \lceil \log_r N_t \rceil$. The following example illustrates the above points. Given t=3 and N_2=3 for r=3, we have $G(1,N_4) = G(1,5) + P(1,5)G(6,8) + P(1,8)G(9,11)$, and therefore, $N_4 = 11$.

Let h_{d+k} denote the maximum number of (d+k)-primes in a restructured formula for $G(1,N_{t+1})$, where $0 \leq k \leq t-d-1$, and $h_{d-1}=0$.

The following two lemmas are consequences of the preceding definitions and conditions.

Lemma 3: The (t+1)-maximal size is given by

$$N_{t+1} = N_t + h_d N_d + \ldots + h_{d+k} N_{d+k} + \ldots + h_{t-1} N_{t-1} \quad (7)$$

Proof: We will prove this lemma by inductively constructing $G(1,N_{t+1})$ on the basis of formula (6). Using the latter and the previous three conditions we can compose $G(1,N_{t+1})$ of the head prime $G(1,N_t)$ and the following OR series of terms:
$S(d) + S(d+1) + \ldots + S(d+k) + \ldots + S(t-1)$.
In the above series, $S(d+k)$ consists of h_{d+k} (d+k)-primes, each ANDed by an appropriate propagating function, $0 \leq k \leq t-d-1$. We will form the terms of the above series inductively on k. For k=0 we have $S(d) = P(1,N_t)G(1+N_t,N_t+N_d) + P(1,N_t+N_d)G(1+N_t+N_d;N_t+2N_t) + \ldots + P(1,N_t + (h_d-1)N_d)G(1+N_t+(h_d-1)N_d;N_t+h_dN_d)$
We may define for convenience the following recursive function

$$M_{d+k} = M_{d+k-1} + h_{d+k}N_{d+k}, \text{ with } M_{d-1} = N_t \quad (8)$$

Using the above function we can easily see that S(d) spans between the integers N_t and $N_t + h_dN_d = M_d$. Using a similar composition we can easily verify that S(d+1) spans between the integers M_d and M_{d+1}. The composition of the general term S(d+k) is as follows

$$S(d+k) = \sum_{q=1}^{h_{d+k}} P(1;M_{d+k-1}+(q-1)N_{d+k})G(1+M_{d+k-1}+(q-1)N_{d+k};M_{d+k-1}+qN_{d+k}) \quad (8a)$$

and therefore S(d+k) spans between M_{d+k-1} and M_{d+k}. We can easily show by induction that S(t-1) spans between the integers M_{t-2} and M_{t-1}. In conclusion we have established that $N_{t+1} = M_{t-1}$; this also established (7) after a recursive summation of (8). (End of Proof).

The following lemma provides the numbers h_{d+k}.

Lemma 4: To maximize N_{t+1} it is required that h_{d+k} satisfies the relation

$$h_{d+k} = \frac{r^{d+k} - M_{d+k-1}}{N_{d+k}} + 1 \quad (9)$$

Proof: To prove this lemma we will use the maximum size propagating function contained in term S(d+k), given in (8a) above. This function is

$$P(1;M_{d+k-1}+(h_{d+k}-1)N_{d+k}).$$

According to condition 3), mentioned previously, the size of the latter function should satisfy

$$M_{d+k-1}+(h_{d+k}-1)N_{d+k} \leq r^{d+k}$$

Solving the last inequality for h_{d+k} we obtain its maximum value given by (9).
(End of Proof).

Our major concern in developing $G(1,N_{t+1})$ has been to be able to place the restructured formula on a computation tree described, with the aid of Fig. 2, as follows: 1) The computation tree of $G(1,N_{t+1})$ begins at level d+1 and ends at level t+1. 2) The input nodes at each tree level all correspond to prime functions. 3) The h_{d+k} (d+k)-prime functions, due to propagating "ANDings" are to be placed at the input tree nodes in the d+k+1 level. 4) As shown in Fig. 2, the tree merges to at most r-1 nodes at the t-th level, with the r-th node being occupied by $G(1,N_t)$. Thus, the number of tree nodes at the d+k level is at most

$$L_{d+k} = (r-1)\, r^{t-d-k}; \quad k=0,1,\ldots,t-d \quad (10)$$

The following iterative rules are useful in employing the tree computation process. 1) The internal tree nodes at the d+k+1 level, $0 \leq k \leq t-1$, are produced by r-ary merging the actual tree nodes at d+k level. 2) The actual tree nodes at d+k level are composed of both the internal and the input tree nodes at d+k level.

If H_{d+k} denotes the number of actual tree nodes at d+k level, then apparently we have

$$H_{d+k} = h_{d+k-1} + (1/r)H_{d+k-1};$$

where $H_{d-1} = h_{d-1} = 0$

However, for efficient merging, H_{d+k} is restricted to be a multiple of r; on the other hand we have $H_{d+k} \leq L_{d+k}$, where the latter is defined in (10). This reasoning results into the lemma next.

Lemma 5: The number of actual nodes at the d+k+1 level of the computation tree for $G(1,N_{t+1})$ satisfies the iterative relation

$$H_{d+k+1} = \min\{L_{d+k+1}\,;\, r\lfloor(1/r)h_{d+k}\rfloor + (1/r)H_{d+k}\} \text{ with } H_{d-1} = 0 \quad k=0,1,\ldots,t-d-1 \quad (11)$$

Proof: The proof of this lemma follows inductively on the basis of the restrictions on H_{d+k}.

Some maximal sizes for r=2 and r=3 are derived by applying the previous rules and Lemma 5; they are listed in Table I. The computation tree for $G(1,N_7)$, with r=3, is shown in Fig. 3.

IV. DISCUSSION

We now consider the implementation of non-prime generating functions. Let $G(1,n_1)$ be a non-prime function where $N_{t_1} < n_1 < N_{t_1+1}$. In other

words, n_1 can always be included in an interval of two consecutive level maximals. To implement $G(1,n_2)$ we have $G(1,n_1) = G(1,N_t) + P(1,N_t)G(1+N_t;n_1)$, where $G(1+N_t;n_1)$ is, in general, a non-prime function and, at the same time, equivalent to $G(1,n_2)$ with $n_2 = n_1 - N_t$. Similarly we have, $N_{t_2} < n_2 < N_{t_2+1}$ and $G(1,n_2) = G(1,N_t) + P(1,N_t)G(1+N_t;n_2)$. Since $n_2 < n_1$, $N_{t_2} \leq N_{t_1}$ (because otherwise, $N_{t_2} \geq N_{t_1+1} > n_1$ which contradicts the preceding relations). If we continue this process we will establish two decreasing sequences, namely $n_1 > n_2 > n_3 > \ldots$ and $N_{t_1} \geq N_{t_2} \geq N_{t_3} \geq \ldots$ until some $n_\ell = N_{t_\ell}$. It is not difficult to prove that the latter will always occur at some step ℓ of the above process since the preceding sequences are both convergent to integer 1, i.e., the trivial 0-prime function $G(1,1)$.

The starting step for producing the computation tree of $G(1,N_{t+1})$ is the head prime function $G(1,N_t)$. However, we can use as head some other z-prime with $z < t$, and then apply the rest of the steps of the above construction process in the same way. It can be proven that the size of $G(1,N_{t+1})$ will not be affected if we use any head prime function $G(1,N_z)$, where $z \leq t$.

The last remark is important because we can apply our method to modular tree implementations of carry lookahead addition [11]. Suppose each such module (so-called carry lookahead generator [12] has 2R input terminals, namely $(p_1,g_1),(p_2,g_2),\ldots,(p_R,g_R)$, and R output terminals, namely the carriers $c_1,c_2,\ldots,c_R$ with $c_R = G(1,R)$. Then to construct a modular computation tree for $G(1,n)$, $n >> R$, we first construct the computation tree of $G(1,n)$ using as head prime the function $G(1,R)$. Since the carry lookahead module provides $G(1,R)$ and all lower level carries, it follows that the computation tree of $G(1,n)$ can be entirely composed of $G(1,R)$ carry lookahead modules. The last point requires additional effort in a continuing research work-project. New results will be reported in the near future.

We conclude this section by noting that the results of this research have been substantially better than the other results on multilevel carry lookahead addition schemes referenced in the literature (see [5], [6], and [7]).

V. APPLICATION TO PARALLEL POLYNOMIAL EVALUATION

It is not difficult to observe that the carry lookahead formula (2) resembles strong analogy to a polynomial-like expression. Therefore, the previously developed parallel lookahead addition method is also applicable to the problem of parallel evaluation of polynomials. This problem has been studied first by Estrin [13] and Dorn [14] who generalized Horner's rule for polynomial evaluation (for a description of the latter see Knuth [15]). Later Muraoka [16], using a tree folding approach, improved Estrin's algorithm. Further improvements were obtained by Maruyama [17], and by Munro and Paterson [18]. Additional improvement is possible by employing a modified version of our previously described tree lookahead algorithm.

First note that there is a similarity in format between a (m+1)-size generating function $G(1,m+1)$ given by (3), and an m-th degree ordinary polynomial

$$Q(x) = q_0 + q_1x + q_2x^2 + \ldots + q_mx^m$$

More particularly, the following one-to-one correspondence between $Q(x)$ and $G(1,m+1)$ may be assigned: q_k corresponds to g_{k+1} $k=0,1,\ldots,m$; the power x^k corresponds to the logic product $p_1p_2\cdots p_k$, i.e., the propagating function $P(1,k)$; the arithmetic addition and multiplication operators correspond to the logical OR and AND, respectively; then, the polynomial $Q(x)$ corresponds to the generating function $G(1,m+1)$. A t-maximal will now be defined to be the maximum degree of a polynomial that can be evaluated in t steps using the arithmetic addition and multiplication operators. Allowing for the above notation adjustment and correspondence, the previous lookahead binary addition algorithm can now be employed for parallel evaluation of polynomials. Results of this algorithm are given in Table II, last column. These results appear to provide the greatest known degrees of polynomials that can be evaluated in given steps.

An additional point to be noted is that the existing polynomial evaluation algorithms assume addition and multiplication processors (operators) with capabilities of handling two arguments (operands) at a time. However, our algorithm allows simultaneous multi-argument processing capability for both arithmetic addition and multiplication.

ACKNOWLEDGEMENT

The author gratefully acknowledges the support for this work from the College of Engineering, University of Cincinnati, under a 1980 Junior Morrow Research Chair grant.

REFERENCES

[1]. K. Hwang, Computer Arithmetic, John Wiley & Sons, New York, 1979.

[2]. O.L. Mac Sorley, "High speed arithmetic in binary computers," Proc. I.R.E., pp. 67-91, 1961.

[3]. J. Sklansky, "Conditional-sum addition logic," I.R.E. Trnas. Comp., Vol. 9, No. 2, pp. 213-216, June 1960.

[4]. H. Ling, "High-speed binary parallel adders," IEEE Trans. Comp., Vol. EC-15, No. 5, Oct. 1966, pp. 799-802.

[5]. I. Flores, The Logic of Computer Arithmethic, Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[6]. F.J. Mowle, A Systematic Approach to Digital Logic Design, Addison-Wesley, Reading, Mass., 1976.

[7]. R. Brent, "On the addition of binary numbers," IEEE Trans. Comp., Vol. C-19, No. 8, August 1970, pp. 758-759.

[8]. S. Winograd, "On the time required to perform addition," J. ACM, Vol. 12, No.2, April 1965, pp. 277-285.

[9]. P.M. Spira, "Computation times of arithmetic and boolean functions in (d,r) circuits," IEEE Trans. Comp., Vol. C-22, No. 6, June 1973, pp. 552-555.

[10]. S.C.Chen and D.J. Kuck "Combinational circuit synthesis with time and component bounds," IEEE Trans. Comp. Vol. C-25, No. 8, August 1977, pp. 712-726.

[11]. C.A. Papachristou, "Parallel implementation of binary comparison circuits," Int. Journal of Electronics, Vol. 47, No. 2, August 1979, 187-192.

[12]. Advanced Microdevices, The AMD 2900 Familty Data Book, Advanced Micro Devices, Inc. Sunnyvale, California, 1978.

[13]. G. Estrin, "Organization of computer systems--the fixed plus variable structure computer," in Proc. Western Joint Comp. Conf., May 1960, pp. 33-40.

[14]. W. Dorn, "Generalization of Horner's rule for polynomial evaluation," IBM J. Res. Develop., Vol. 6, April 1962, pp. 239-245.

[15]. D.E. Knuth, Seminumerical Algorithms: Vol. 2 of The Art of Computer Programming, Addison-Wesley, 1969, pp. 422-444.

[16]. Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. Dissertation, Dept. Comp. Science, University of Illinois, Urbana, 1971, pp. 33-41.

[17]. K. Maruyama, "On the parallel evaluation of polynomials," IEEE Trans. Comp., Vol. C-22, No. 1, Jan. 1973, pp. 2-5.

[18]. I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," in Proc. IEEE 12th Annual Symp. Switch. Automata Theory, October 1971, pp. 132-139.

TABLE 1

size of t-maximals with fan-in r=2 and 3

level t	Size N_t r=2	r=3
2	2	3
3	3	5
4	5	11
5	8	21
6	21	51
7	37	105
8	63	231
9	105	537
10	184	1215

TABLE II

Degrees of polynomials that can be evaluated in given steps

Step	Degree		
	Folding [16]	Multifolding [17]	Proposed algorithm
1	0	0	0
2	1	1	1
3	2	2	2
4	4	4	4
5	7	7	7
6	12	12	12
7	20	20	21
8	33	36	37
9	54	62	63
10	88	104	107
11	143	183	187
12	230	320	327
13	376	572	577
14	609	992	1,009
15	986	1,728	1,763
16	1,596	3,059	3,123
17	2,583	5,489	5,578
18	4,180	9,767	9,950
19	6,764	17,454	17,828
20	10,945	31,286	31,940
21	17,710	55,915	57,278

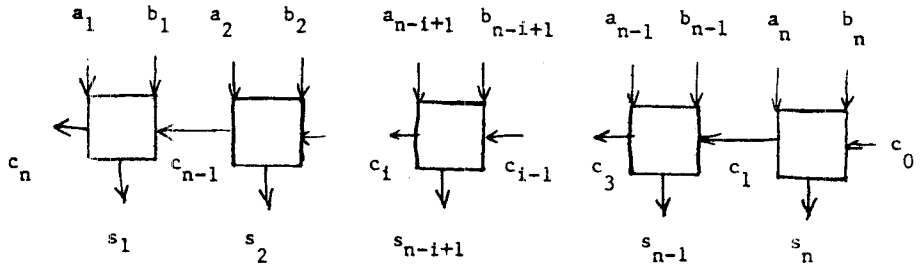

Fig. 1 : A general n-bit adder

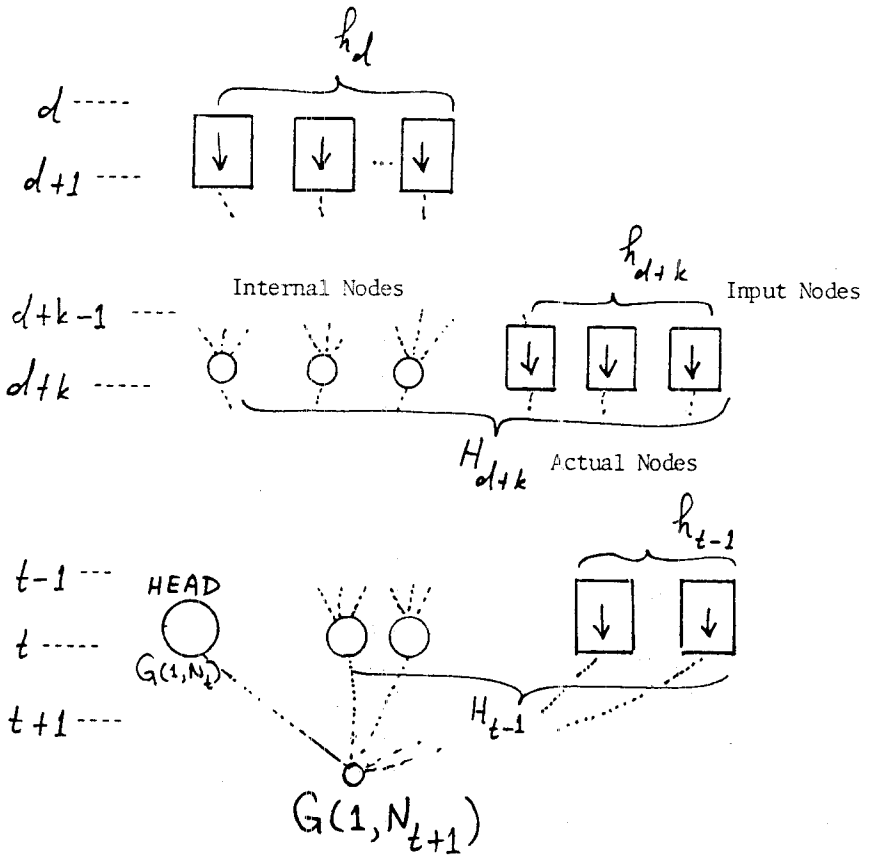

Fig. 2 : The general computation tree of $G(1,N_{t+1})$

Arrow at d+k level indicates a (d+k)-maximal which is ANDed by an appropriate propagating function.

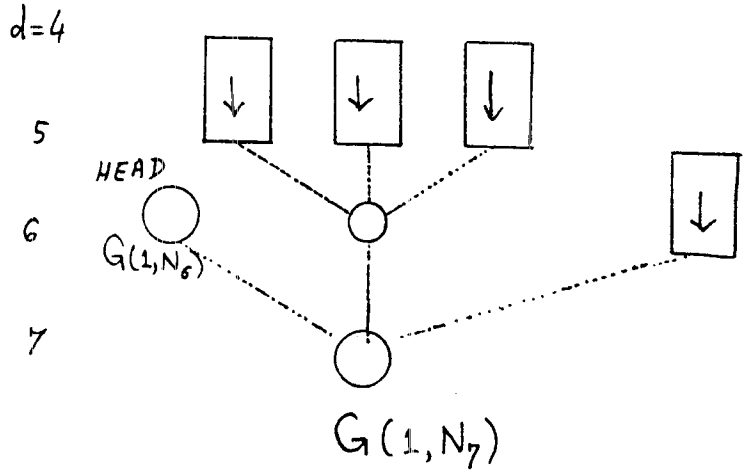

Fig. 3: The computation tree of $G(1,N_7)$ with head maximal $G(1,N_6)$.

$$G(1,N_7) = G(1,N_6) + P(1,N_6)G(1+N_6,N_6+N_4) + P(1,N_6+N_4)G(1+N_6+N_4,N_6+2N_4) +$$
$$+ P(1,N_6+2N_4)G(1+N_6+2N_4,N_6+3N_4) + P(1,N_6+3N_4)G(1+N_6+3N_4,N_6+N_5+3N_4$$

$$N_7 = N_6 + 3N_4 + N_5$$