Università degli Studi di Padova

Tesi di Laurea in

Ingegneria dell'Informazione

Study and Design of a 32-bit High-Speed Adder

Relatore
Andrea Neviani

Laureando
Matteo Stangherlin

Anno Accademico 2012/2013

Contents

Introduction . . . 7

1 Study of Adders **9**

1.1 Full Adder . . . 9

1.2 Ripple Carry Adder . . . 12

1.3 Carry Lookahead Adder . . . 13

1.3.1 Monolithic Carry Lookahead Adder . . . 13

1.3.2 Logarithmic Carry Lookahead Adder . . . 14

1.4 Recurrence Solver Adders . . . 19

1.5 Radix-4 implementation of adders . . . 27

2 Design of a 32-bit High-Speed Adder **31**

2.1 Circuit implementation . . . 31

2.1.1 Blocks implementation . . . 32

2.1.2 32-bit Kogge-Stone adder implementation . . . 37

2.1.3 32-bit Brent-Kung adder implementation . . . 44

2.2 Transistor sizing . . . 49

2.3 Simulation . . . 51

Conclusion . . . 63

Introduction

Addition is the most basic and most frequently used operation in digital circuit design. Due to this reason, it often represents the limiting factor for the computational speed of a system. The accurate optimization of adders holds a role of major importance in the architecture of more complex structures such as arithmetic logic units of microprocessors. Optimization proceeds at both logical and circuit levels. Since hardware can only perform a relatively simple set of Boolean operations, arithmetic calculations are based on a hierarchy of operations that are built upon simpler ones in such a way that a fast and a low area-occupying circuit is obtained. On the other hand, as far as the circuit level is concerned, optimization can be reached by manipulating the size of transistors and the topology of the logic gates so that every single element of the adder is optimized. In the following paragraphs, various different architectures for adders will be discussed, with particular attention to speed, power and chip area in order to measure the efficiency of each device.

Chapter 1

Study of Adders

1.1 Full Adder

The basic building block on which more complex adding structures are derived is the one-bit full adder. Operation of a full adder is defined by the Boolean equations for the sum and the carry signals:

$$S_i = A_i \oplus B_i \oplus C_i = A_i\overline{B}_i\overline{C}_i + \overline{A}_iB_i\overline{C}_i + \overline{A}_i\overline{B}_iC_i + A_iB_iC_i \qquad (1.1.1)$$

$$C_{i+1} = A_iB_i + B_iC_i + A_iC_i \qquad (1.1.2)$$

where A_i and B_i are the two bits that must be summed and C_i is the input carry; S_i and C_{i+1} represent the sum and the carry outputs from the *i-th* stage, respectively. Since the previous equations require two *XOR* gates to be implemented, it is customary to define sum and carry output signals as functions of two intermediate operations: *generate* (G_i) and *propagate* (P_i). The former processes the generation of the carry when both the bits A_i and B_i are set to 1, independently of the value of C_i; the latter, when set to 1, carries the value of C_i to C_{i+1}, propagating the carry. The Boolean expressions of these two functions are the following:

$$G_i = A_iB_i \qquad (1.1.3)$$

$$P_i = A_i \oplus B_i \qquad (1.1.4)$$

Note that generate and propagate terms are functions of the entries A_i and B_i only and do not depend on the carry signal C_i. Generate and propagate can be used to redefine sum S_i and carry C_{i+1}:

$$S_i = P_i \oplus C_i \qquad (1.1.5)$$

$$C_{i+1} = G_i + P_i C_i \tag{1.1.6}$$

The logical implementation of a full adder is shown in Figure 1.1.

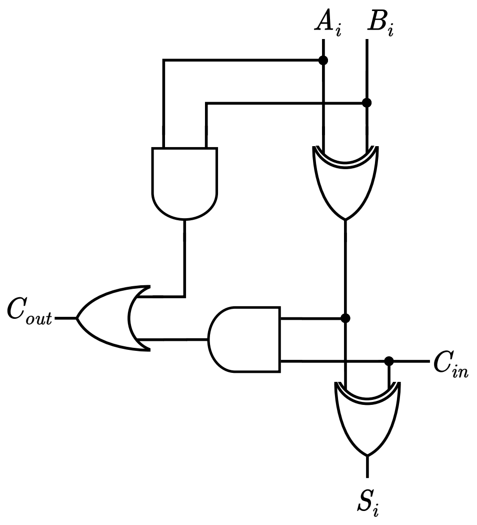


Figure 1.1: Logical implementation of a full adder.

Since the delay from $A_i$or B_i to S_i is two *XOR* delays and the delay from C_i (or any of the inputs) to C_{i+1} is given by an *AND* and an *OR* delay, a full adder is not efficiently implemented by using static circuits. Conversely, pass-transistor circuits can implement the functions more efficiently in terms of speed, even though they have more area occupation. A possible implementation of a pass-transistor full adder is presented in Figure 1.2.

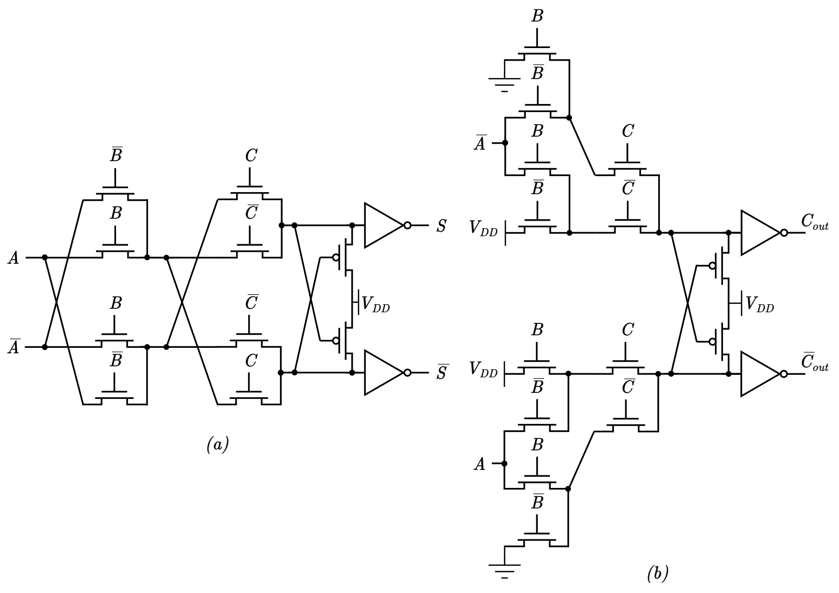


Figure 1.2: Pass-transistor implementation of a full adder: *(a)* sum and *(b)* carry signals.

1.2 Ripple Carry Adder

A simple way to implement an adder for N-bit numbers is to concatenate N full adders, connecting the output carry of the $k-1$ block ($C_{out,k-1}$) to the input carry of the k block ($C_{in,k}$), with $k = 1, 2, ..., N-1$, while in general the input carry of the first full adder ($C_{in,0}$) is set to zero. Each full adder block computes the sum of the $i-th$ bits of the two numbers and its carry, which is transmitted to the following full adder block. This kind of architecture is called a *Ripple Carry Adder* (*RCA*), since the carry signal propagates from the least significant bit position to the most significant one. The structure of a *RCA* is shown in Figure 1.3.

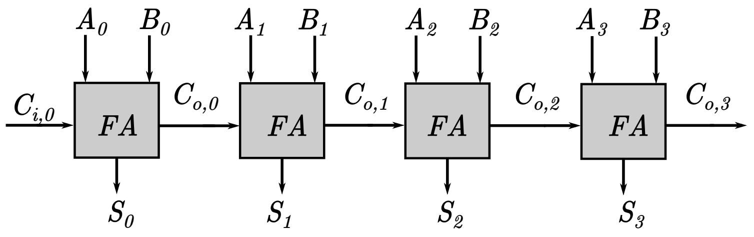


Figure 1.3: Structure of a Ripple Carry Adder

The delay introduced by the circuit depends on the number of full adder blocks that must propagate the carry, which in turn depends on the configuration of the bits given as input to the adder block. The path from the input to the output signal that is likely to take the longest time is designated as a "*critical path*". In the case of the *RCA*, this is the path from the least significant input bits A_0 or B_0 to the last sum bit S_N. Therefore, delay is proportional to the number N of bits of the input words. Assuming that every full adder block is implemented as shown in Figure 1.1, the delay introduced by the *RCA* is given approximately by the following equation:

$$t_{RCA} \approx t_{XOR} + (N-1)t_{AO} + t_{GP} \tag{1.2.1}$$

where t_{GP} is the delay introduced by computing the generate and propagate functions for each *FA* block, t_{XOR} is the delay originated by calculating the sum S_i for each *FA* block and t_{AO} is the delay introduced by the computation of the carry from $C_{in,i}$ to $C_{out,i}$, which corresponds to the sum of an *AND* and *OR* gate delay. Since every full adder block has likely the same inner architecture, t_{AO} is assumed to be the same for each block.

As equation (1.2.1) shows, the delay of a *RCA* grows linearly with the number N of *FA* stages, that is, the number N of bits of the input words. That is an important aspect to consider when designing for digital systems that require a sum of large number of bits. In particular, it is customary to avoid the use of *RCA* topology for adders that compute operations involving 16 binary digits or more, even though the area occupation is much smaller than other more performing adders that will be introduced in the following paragraphs.

1.3 Carry Lookahead Adder

1.3.1 Monolithic Carry Lookahead Adder

As seen in the previous paragraph, adder performance is strongly influenced by the carry-propagating process. In order to achieve higher computational speed, it is thus necessary to make the sum independent from carry propagation. The principle on which monolithic *Carry Lookahead Adder* (monolithic *CLA*) is based offers a possibility to solve such issue. As previously stated in equation (1.1.6), in an N bit adder, each carry bit can be expressed as:

$$C_{out,i} = G_i + P_i C_{out,i-1} \tag{1.3.1}$$

The dependence of $C_{out,i}$ from $C_{out,i-1}$ can be eliminated by expliciting the dependence of $C_{out,i-1}$ from the input signals G_{i-1} and P_{i-1}:

$$C_{out,i} = G_i + P_i(G_{i-1} + P_{i-1}C_{out,i-2}) \tag{1.3.2}$$

The previous equation states that a carry signal exits from stage i if (a) a carry is generated in the stage i; (b) a carry is generated at stage $i-1$ and propagates across stage i; or (c) a carry enters stage $i-1$ and propagates across both stages $i-1$ and i.

By proceeding recursively, we obtain the following equation:

$$C_{out,i} = G_i + P_i(G_{i-1} + P_{i-1}(\cdots + P_1(G_0 + P_0 C_{in,0}))) \tag{1.3.3}$$

where $C_{in,0}$ is typically set to zero. Equation (1.3.3) can be used to implement an N bit adder. For every bit, the result of the sum is independent from the carry of the previous one, thus eliminating the problem of the carry propagation. The delay introduced by the monolithic *CLA* should be in theory constant and independent from the number N of bits.
In practice, the implementation of the circuit, as shown in Figure 1.4 for a 4-bit adder, points out that the delay is not constant, but grows linearly with the number N of bits. In fact, even though the carry $C_{out,i}$ does not depend on the previous carry outputs, it depends on the respective generate and propagate functions, making the delay proportional to $N+1$, i.e. the number of parallel branches of the circuit. Moreover, the circuit features, for a large N, an high fan-in and fan-out. For example, we see that the signals P_0 and G_0 appear in the logical expression of every other signal, significantly increasing the capacity associated to each connected line. Because of these reasons, monolithic *CLA* is useful only for small numbers of bit (typically not more than four).

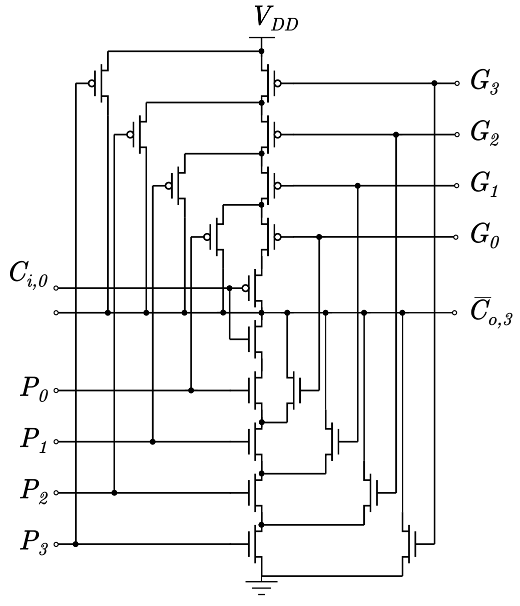


Figure 1.4: CMOS implementation of a 4-bit Monolithic Carry Lookahead Adder

1.3.2 Logarithmic Carry Lookahead Adder

The performance of monolithic *CLA* makes it suitable only for adding small groups of bits. In order to enhance the performance of the *CLA* it is necessary to modify the architecture of the adder by reorganizing it in a hierarchical structure, as in *Logarithmic Carry Lookahead Adder* (commonly referred to as *CLA* since its monolithic version is not particularly efficient).
CLA uses generation and propagation modules for each bit position and lookahead modules which are used to generate carry signals independently for a group of k-bits. In addition to carry signal for the group, lookahead modules produce group carry generate and group carry propagate outputs that indicate that the carry is generated within the group, or that in incoming carry would propagate across the group. In this way, it is possible to know more in advance whether carry is generated or propagated inside a large group of bits, decreasing the delay due to the carry computation. Such modules are implemented by extending equation (1.3.2) for a group of k-bits (for reasons that will shortly be explained, the index i is zero or an integer multiple of four):

$$\begin{aligned} C_{out,i} &= G_i + P_i C_{in,i} \\ C_{out,i+1} &= G_{i+1} + P_{i+1} C_{out,i} \\ C_{out,i+2} &= G_{i+2} + P_{i+2} C_{out,i+1} \\ C_{out,i+3} &= G_{i+3} + P_{i+3} C_{out,i+2} \end{aligned} \tag{1.3.4}$$

Substituting $C_{out,i}$ into $C_{out,i+1}$, then $C_{out,i+1}$ into $C_{out,i+2}$, then $C_{out,i+2}$ into $C_{out,i+3}$ yields the expanded equations:

$$\begin{aligned}
C_{out,i} &= G_i + P_i C_{in,i} \\
C_{out,i+1} &= G_{i+1} + P_{i+1} G_i + P_{i+1} P_i C_{in,i} \\
C_{out,i+2} &= G_{i+2} + P_{i+2} G_{i+1} + P_{i+2} P_{i+1} G_i + P_{i+2} P_{i+1} P_i C_{in,i} \\
C_{out,i+3} &= G_{i+3} + P_{i+3} G_{i+2} + P_{i+3} P_{i+2} G_{i+1} + P_{i+3} P_{i+2} P_{i+1} G_i \\
&\quad + P_{i+3} P_{i+2} P_{i+1} P_i C_{in,i}
\end{aligned} \tag{1.3.5}$$

As seen in the previous paragraph, each additional stage increases the size of the logic gates in terms of number of inputs. Therefore, it is not appropriate to create a group that calculates the carry for a large number of bits. The maximum number of inputs per gate for current technologies is four. In order to continue the process, *CLA* collects carry, generate and propagate signals into lookahead modules that compute *group-generate* and *group-propagate* signals, respectively $G_{i+3:i}$ and $P_{i+3:i}$ (where i is zero or an integer multiple of four), over a four-bit group. Every lookahead module is able to anticipate whether a carry entering the module will be propagated all the way across it or generated within it four times faster than a normal ripple process. $G_{i+3:i}$ and $P_{i+3:i}$ are described by the following equation:

$$\begin{aligned}
G_{i+3:i} &= G_{i+3} + P_{i+3} G_{i+2} + P_{i+3} P_{i+2} G_{i+1} + P_{i+3} P_{i+2} P_{i+1} G_i \\
P_{i+3:i} &= P_{i+3} P_{i+2} P_{i+1} P_i
\end{aligned} \tag{1.3.6}$$

The notations $G_{j:i}$ and $P_{j:i}$ denote respectively *group-generate* and *group-propagate* for the group that includes bit positions from i-th to j-th. The carry equation can be expressed in terms of the 4-bit *group-generate* and *group-propagate* signals:

$$C_{out,i+3:i} = G_{i+3:i} + P_{i+3:i} C_{in,i+3:i} \tag{1.3.7}$$

where $C_{in,i+3:i}$ is the input carry of the i-th stage. Usually, carry input for the first block is set to zero. The extended expressions of the carry equations can be obtained by substitution as done for equation (1.3.4).
In a recursive fashion, from group generate, propagate and carry, it is possible to create a "*group of groups*" or "*super group*". The inputs of the "*super group*" are $G_{i+3:i}$ and $P_{i+3:i}$ signals computed by all the groups within it. Each "*super group*" produces propagate $P^*_{j:i}$ and generate $G^*_{j:i}$ signals that indicate that the carry signal will be propagated across all the groups belonging to the "*super group*" or will be generated in one of them. Similarly to the group, a "*super*

group" produces an output carry signal as well as an input carry signal for each of the groups in the level above:

$$G^*_{j:i} = G_{i+3:i} + P_{i+3:i}G_{i+2:i} + P_{i+3:i}P_{i+2:i}G_{i+1:i} + P_{i+3:i}P_{i+2:i}P_{i+1:}G_{i:i}$$

$$P^*_{j:i} = P_{i+3:i}P_{i+2:i}P_{i+1:i}P_{i:i}$$

$$C_{out,j} = G^*_{j:i} + P^*_{j:i}C_{in,i} \tag{1.3.8}$$

Depending on the number of bits of the sum, it is possible to build a group of "*super groups*" and so on. In particular , a 32-bit adder needs 8 4-bit groups, 2 "*super groups*" and a final group that conveys carry, generate and propagate signals from the "*super groups*", producing a final carry and the intermediate one for the levels below.
The main advantage of the hierarchical configuration is that the critical path does not travel in horizontal direction, as happens in *RCA*. Because of the tree structure, the delay of *CLA* is not directly proportional to the number of bits N, but to the number of levels used. Therefore, the delay of *CLA* is proportional to the *log* function of the number of bits N.

The delay introduced by *CLA* can be evaluated by observing that each lookahead level takes one gate delay in order to compute $P_{i+3:i}$ and two gate delays in order to compute $G_{i+3:i}$ and $C_{out,i+3:i}$. Therefore, a lookahead module introduces a two-gate delay. Moreover, each generate and propagate function G_i and P_i takes one gate delay, that must be added to the delay of the *XOR* gate which processes the sum at the bit level. Finally, being the number of lookahead modules given by the logarithm to the base k (where k is the number of bits in a group) of the total number of bits N minus one, it is possible to compute the delay of a *LCLA*:

$$t_{CLA} = 1 + 1 + 2(\log(\lceil N \rceil) - 1) = 2\log(\lceil N \rceil) \tag{1.3.9}$$

The logarithmic dependence of the delay on the number of bits of the sum makes the *CLA* one of the theoretically fastest structures for addition. Unfortunately, in practice the efficiency of *CLA* is much lower than expected. This is due to the fact that the model used to compute the delay does not take into account fan-in and fan-out dependencies of the logic gates. In fact, as shown in Figure 1.6 where a domino realization of a *CLA* is presented, logic gates do not have a constant fan-in, making the delay of group and "*super groups*" modules much greater than the ones of the individual generate and propagate functions. Nevertheless, *CLA* can reach a remarkable computational speed if properly implemented. An example was already presented in Figure 1.6, which shows a

Manchester Carry-Chain (*MCC*) implementation of a CMOS Domino realization of a 64-bit adder by Motorola. Due to the properties of pass-transistor logic which characterize *MCC*, the adder can reach the remarkable speed of 4.5 ns at $V_{DD} = 5V$ and temperature of 25°C.

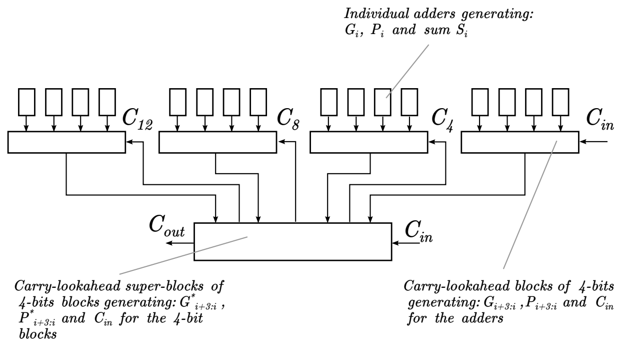

Individual adders generating: G_i, P_i and sum S_i
C_{12}
C_8
C_4
C_{in}
C_{out}
C_{in}
Carry-lookahead super-blocks of 4-bits blocks generating: ${G^*}_{i+3:i}$, ${P^*}_{i+3:i}$ and C_{in} for the 4-bit blocks
Carry-lookahead blocks of 4-bits generating: $G_{i+3:i}$, $P_{i+3:i}$ and C_{in} for the adders

Figure 1.5: Block structure of a 16-bit Carry lookahead adder.

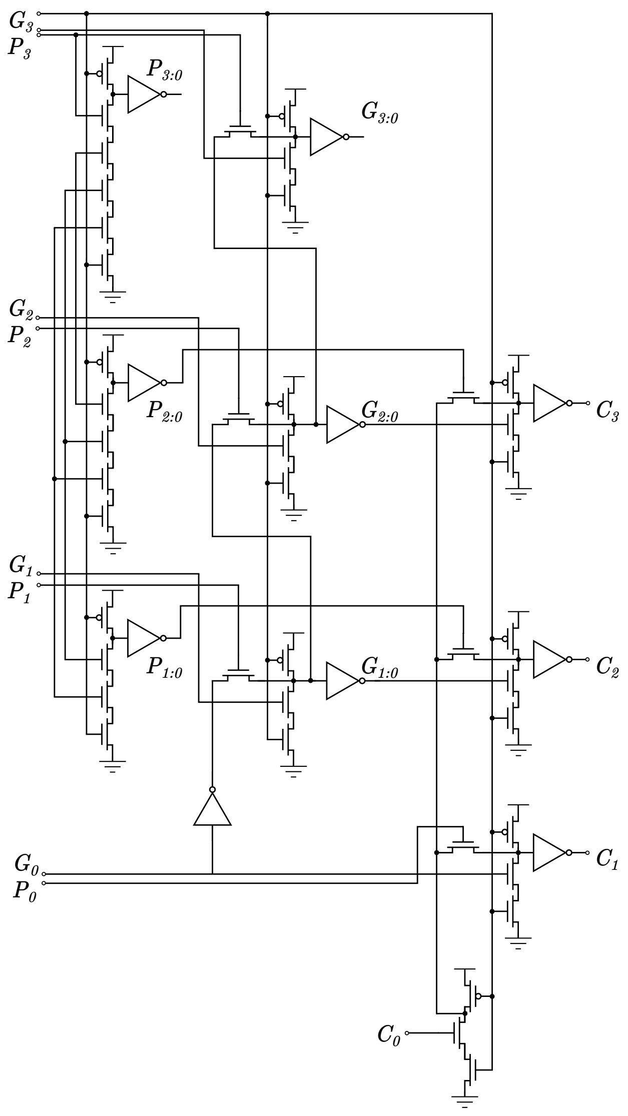


Figure 1.6: CMOS Domino realization of a 4-bit module of a 64-bit adder by Motorola.

1.4 Recurrence Solver Adders

A particularly efficient class of adders is the one based on solving recurrence equations that was introduced by Brent and Kung, based on the previous work by Kogge and Stone.
They realized that any first order recurrence problem can be written in an alternate form by using a concept called *recursive doubling*, which consists of breaking the calculation of one term into two equally complex subterms. In particular, as seen in the previous paragraphs, carry output at the i-th stage can be written as a linear combination of generate and propagate functions at i-th and $i-1$-th stages:

$$C_{out,i} = G_i + P_i G_{i-1} + P_i P_{i-1} C_{out,i-1} \tag{1.4.1}$$

The previous equation can now be divided into two subterms by defining the $\bullet$ operator, termed "*dot*", as follows:

$$(G_i, P_i) \bullet (G_{i-1}, P_{i-1}) = (G_i + P_i G_{i-1}, P_i P_{i-1}) \tag{1.4.2}$$

where G_i, P_i, G_{i-1} and P_{i-1} are all Boolean functions. By using the *dot* operator, *group-generate* and *group-propagate* functions can be easily written for a group of two bits. For example, given generate and propagate functions for the two least significant bits, it is possible to obtain the *group-generate* and *group-propagate* functions for the two-bit group:

$$(G_1, P_1) \bullet (G_0, P_0) = (G_1 + P_1 G_0, P_1 P_0) = (G_{1:0}, P_{1:0}) \tag{1.4.3}$$

Moreover, *dot* operator is associative. This property can be used in order to compute generate and propagate functions over a group of k bits:

$$(G_{k:0}, P_{k:0}) = (G_k, P_k) \bullet (G_{k-1}, P_{k-1}) \bullet \ldots \bullet (G_0, P_0) \tag{1.4.4}$$

Since the *dot* operator is associative, every $(G_{k:0}, P_{k:0})$ can be computed in the order defined by a binary tree. This is shown in Figure 1.7, for $k = 16$. In the figure each black dot processes the *dot* operator, while the white circles process generate and propagate functions for a single bit. Furthermore, it is possible to calculate carry signal for every bit at positions $2^n - 1$, where $n = 1, 2, \ldots, \log_2(N)$, by simply extending equation (1.4.4). In fact we have:

$$(C_{o,k}, 0) = (G_k, P_k) \bullet (G_{k-1}, P_{k-1}) \bullet \ldots \bullet (G_0, P_0) \bullet (C_{i,0}, 0) \tag{1.4.5}$$

Calculating carry output only for the $2^n - 1$ bit positions is though not sufficient to perform a correct sum. It is thus necessary to compute $(G_{k:0}, P_{k:0})$ for

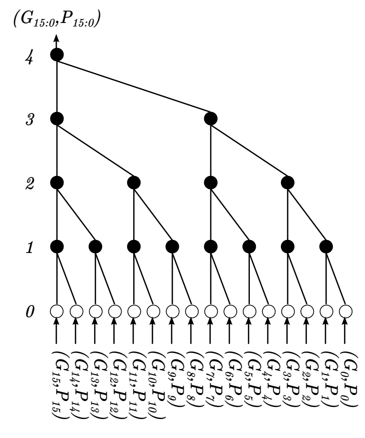


Figure 1.7: Computation of (G_{15}, P_{15}) using a tree structure.

$0 \leq k \leq N$, where N is the number of the bits of the sum, and then evaluate the carry output by implementing equation (1.4.5). The computation can be performed by extending the tree structure in Figure 1.7, so that it is possible to obtain $(G_{k:j}, P_{k:j})$ for every k and j. This process is illustrated in Figure 1.9, for $N = 16$. In addition to the black dots and white squares that compute *dot* operator and (G_k, P_k) respectively, Figure 1.9 also features sum operator (depicted as a white diamond), obtained by implementing equation (1.1.5). This particular architecture is called Kogge-Stone adder.

Although being a variation of a *CLA*, recurrence solver adders feature several properties that make them preferable for implementation, such as:

(a) a regular layout that allows easier implementation

(b) a fan-out that can be controlled and limited to no more than 2

(c) possibility to operate trade-offs between fan-in, fan-out and hierarchical topology

Moreover, since they belong to the class of binary trees, recurrence solver adders can compute output carry with complexity proportional to the number of levels of the adder, which is variable and depends on the particular architecture. As far as Kogge-Stone adder is concerned, its number of levels is given by $\log_2(N)$.

A possible implementation of Kogge-Stone adder can be realized in *CMOS* technology by exploiting the properties of dynamic logic. As already pointed out in Figure 1.9, Kogge-Stone adder consists in the recursive repetition of simpler

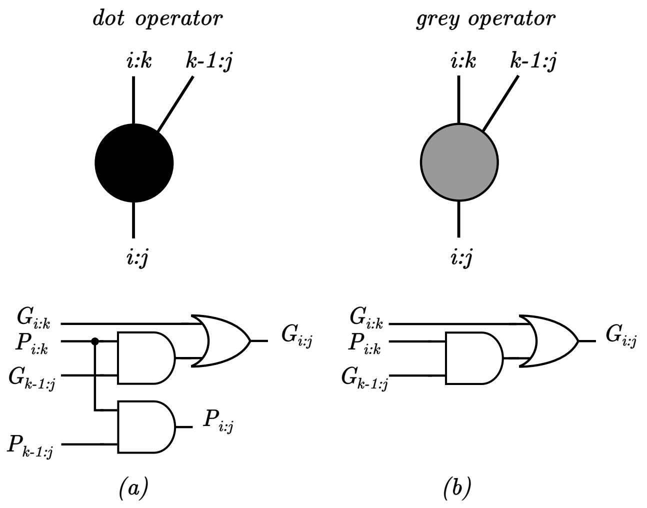


Figure 1.8: Logical structure of *(a) dot* and *(b) grey* operators.

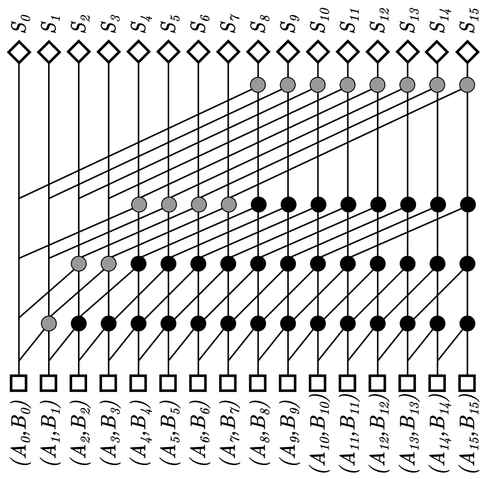
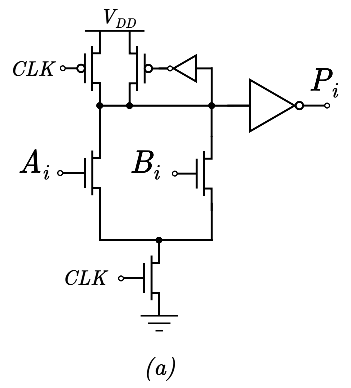

S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15}
(A_0,B_0) (A_1,B_1) (A_2,B_2) (A_3,B_3) (A_4,B_4) (A_5,B_5) (A_6,B_6) (A_7,B_7) (A_8,B_8) (A_9,B_9) (A_{10},B_{10}) (A_{11},B_{11}) (A_{12},B_{12}) (A_{13},B_{13}) (A_{14},B_{14}) (A_{15},B_{15})

Figure 1.9: Tree structure of a 16-bit Kogge-Stone adder.



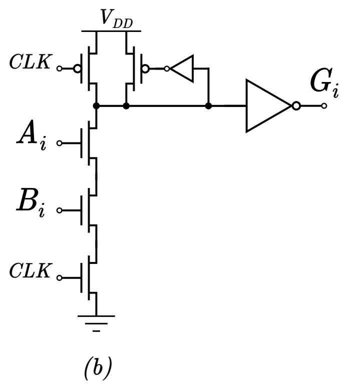
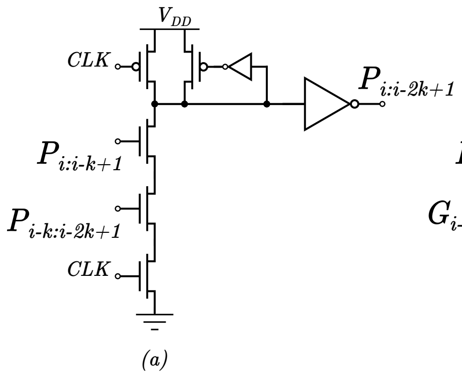


Figure 1.10: CMOS Domino implementation of *(a)* propagate and *(b)* generate signals.



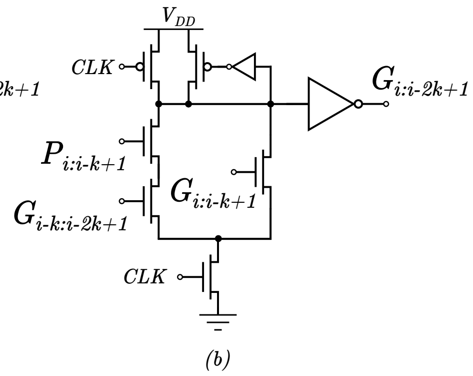


Figure 1.11: CMOS Domino implementation of the *dot* operator: *(a) group-propagate* and *(b) group-generate* signals.

modules, which realize *dot* operator, generate, propagate and sum functions.
The Domino *CMOS* implementation of generate and propagate functions from equations (1.1.3) and (1.1.4) is shown in Figure 1.10. Note that propagate function does not actually implement a *XOR* gate, but the following equation:

$$P_i = A_i + B_i \tag{1.4.6}$$

which represents a *OR* gate. This is an alternate definition of the propagate function that allows carry to be forwarded also when both A_i and B_i assume logical value 1. Due to the way generate and propagate bits are used by *CLA* and recurrence solver adders, propagate definition given by equation (1.4.6) is equivalent to the one given by equation (1.1.4) [1]. Because of this, propagate function can be easily implemented in Domino logic. In addition, both generate and propagate functions feature a bleeder circuit whose function is to eliminate every static power consumption originated by the pull-down network. Every bleeder p-MOS has a small inverter connected to its gate in order to decouple it from the output load, allowing a fast switching off.
Figure 1.11 shows the Domino logic implementation of the *dot* operator. In order to implement equation (1.4.2), every *dot* operator consists of two modules: the first computes *group-propagate* and the second *group-generate*. The inputs of *group-propagate* modules are given by the signals $P_{i:i-k+1}$ and $P_{i-k:i-2k+1}$, where i represents the bit position of the operand and k represents the logic level from which the input is produced. Similarly for *group-generate* the imputs are $G_{i-k:i-2k+1}$, $G_{i:i-k+1}$ and $P_{i:i-k+1}$. The outputs of the two modules will be $P_{i:i-2k+1}$ and $G_{i:i-2k+1}$ respectively. In addition, since *group-propagate* and *group-generate* functions do not represent the first stage of the circuit, it would be possible remove the evaluation transistor from the PDN in order to reduce the logical effort from 1 to 2/3. Note that by doing this it would also be necessary to delay the clock signal of stage k from the one of stage $k-1$ in order to eliminate the direct path from V_{DD} to ground. In this way the clock signal is delayed by a constant time from a stage and the following one.
After computing all generate and propagate functions it is also necessary to calculate the sum of the i-th bit of the two operands by implementing the following equation:

$$S_i = A_i \oplus B_i \tag{1.4.7}$$

A static implementation of equation (1.4.7) is not particularly efficient, having implemented the previous stages in Domino logic. In fact a static gate could increase the delay of the circuit and cause unwanted transitions whether the following stage is implemented in dynamic logic. A possible way to compute the sum using Domino logic is to precompute the two possible outputs for S_i, $S_i^0 = \overline{A_i \oplus B_i}$ and $S_i^1 = A_i \oplus B_i$, and select the proper output by exploiting

[1]Note that equations (1.4.6) and (1.1.4) are not equivalent for full-adders or *RCA*.

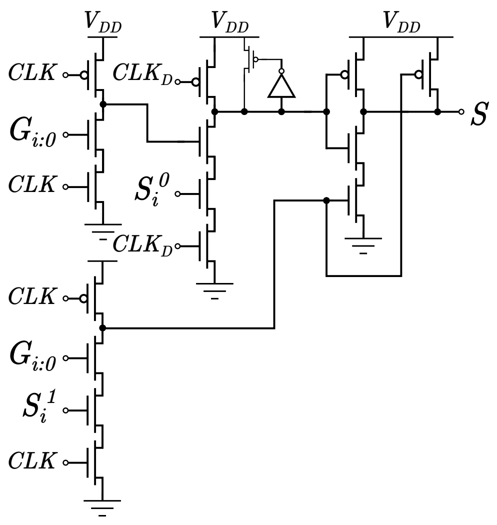
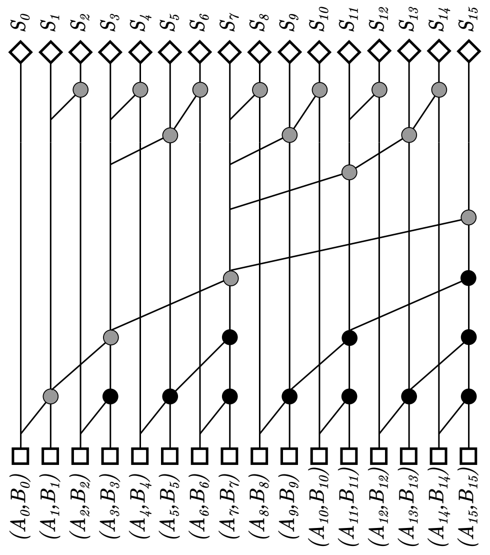


Figure 1.12: CMOS Domino implementation of the sum selector.

$G_{i:0}$ function. The implementation of the sum operator is shown in Figure 1.12. Since domino logic does not provide the negated value of the inputs, the sum operator is implemented by three logic stages. Note that the first two stages, namely *NOT* and *NAND* gates, do not feature an inverter at the output and this may cause unwanted glitches at the output. This problem can be solved by delaying the clock for the second *NAND* gate so that the delayed clock edge follows every commutation of the inputs from the previous stage.

As pointed out in Figure 1.9, the implementation of a 16-bit Kogge-Stone adder requires 49 modules in order to realize all the *dot* operators, 16 modules in order to implement generate and propagate functions and finally 16 more logic gates for the implementation of the sum. If domino logic is used as suggested above the adder would need a total of 1507 transistors (this is the worst case in which every gate in the *dot* modules has the evaluation transistor in the *PDN*). Moreover, more than one wiring track is required between two logic levels.
Because of these reasons, sometimes it is preferable to reduce the computational speed in order to improve area occupation and decrease power consumption, as suggested by Brent and Kung.
In fact, it is possible to exploit the properties of the *dot* operator in order to implement a structure that computes the sum of two operands with smaller area occupation and power consumption than Kogge-Stone adder. In particular, it was previously stated that every instance $(G_{k:0}, P_{k:0})$ can be computed in the order defined by a binary tree, giving an example of the computation in Figure 1.7 for $k = 16$. Note that the binary tree allows the carry computation only for the bits at positions $2^n - 1$, with $n = 1, 2, \ldots, \log_2(N)$. For example, referring to



S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15}
(A_0,B_0) (A_1,B_1) (A_2,B_2) (A_3,B_3) (A_4,B_4) (A_5,B_5) (A_6,B_6) (A_7,B_7) (A_8,B_8) (A_9,B_9) (A_{10},B_{10}) (A_{11},B_{11}) (A_{12},B_{12}) (A_{13},B_{13}) (A_{14},B_{14}) (A_{15},B_{15})

Figure 1.13: Tree structure of a 16-bit Brent-Kung adder.

the structure in Figure 1.7, it is possible to compute the following carry outputs:

$$\begin{aligned}
(C_{o,0},0) &= (G_0,P_0) \bullet (C_{i,0},0)\\
(C_{o,1},0) &= [(G_1,P_1) \bullet (G_0,P_0)] \bullet (C_{i,0},0) = (G_{1:0},P_{1:0}) \bullet (C_{i,0},0)\\
(C_{o,3},0) &= [(G_{3:2},P_{3:2}) \bullet (G_{1:0},P_{1:0})] \bullet (C_{i,0},0) = (G_{3:0},P_{3:0}) \bullet (C_{i,0},0)\\
(C_{o,7},0) &= [(G_{7:4},P_{7:4}) \bullet (G_{3:0},P_{3:0})] \bullet (C_{i,0},0) = (G_{7:0},P_{7:0}) \bullet (C_{i,0},0)\\
(C_{o,15},0) &= [(G_{15:8},P_{15:8}) \bullet (G_{7:0},P_{7:0})] \bullet (C_{i,0},0)\\
&= (G_{15:0},P_{15:0}) \bullet (C_{i,0},0)
\end{aligned} \tag{1.4.8}$$

Since binary tree itself does not allow to compute the complete sum, it is necessary to juxtapose another structure to it so that the carry can be calculated for every position of the two operands. Kogge and Stone's solution to this problem was to reproduce the tree structure for every bit position, significantly increasing the number of *dot* operators and wiring tracks.

Instead of reproducing the tree structure, it is possible to compute the remaining carry outputs by simply adding a second tree to the structure in Figure 1.7, as suggested by Brent and Kung. The resulting scheme shown in Figure 1.13 is known as Brent-Kung adder. The additional structure, called *inverse binary tree*, combines the intermediate results in order to compute the remaining carry outputs.

In this way, the number of *dot* operators implemented in the adder is significantly reduced: referring to a 16-bit adder, instead of the 49 requested by a Kogge-Stone adder, a Brent-Kung adder only needs 27. Moreover the number of wiring tracks between two logic levels is limited to one.
Note that the inverse binary tree consists of a different class of *dot* operators, depicted as grey dots in Figure 1.13. Since the implementation of the sum only requires the *group-generate* signal $G_{i:0}$ in addition to the two possible values of the signal S_i, *grey* operators implement only the following equation:

$$G_{i:i-2k+1} = G_{i:i-k+1} + P_{i:i-k+1}G_{i:i-2k+1} \tag{1.4.9}$$

instead of equation (1.4.2). Because of this, only three inputs are needed and only one output is produced. The implementation of the *grey* operator can be seen in Figure 1.11 b. Note that *grey* operators are also used in the implementation of Kogge-Stone adder for the nodes directly connected to the modules that compute the sum.
The *width* w of an adder can be defined as the number of bits it accepts at one time from each operand. For the parallel adders considered so far, $w = N$ was assumed. For any Brent-Kung adder it can be proven that all the carries in an N-bit addition can be computed in time proportional to $(N/w) + \log(w)$ and in area proportional to $w \log(w) + 1$, and so can the addition. The application of the previous bound to the case $w = N$ leads to the conclusion that the computation delay for a N-bit Brent-Kung adder is proportional to $\log(N) + 1$. Nevertheless, Brent-Kung adder does not reach the computational speed of a Kogge-Stone adder. This is due to the fact that the structure of the wiring tracks is less regularly distributed and the fan-out is variable for each logic gate. In particular, referring to the 16-bit adder in Figure 1.13, the fan-out of the node associated to the intermediate carry output $(C_{o,7}, 0)$ is given by five *dot* operators and a sum function, significantly increasing the delay.

1.5 Radix-4 implementation of adders

Another factor that contributes to the enhancement of the delay is the number of logic levels of the adder which, in the case of a recurrence solver adder, corresponds to the number of carries (or equivalently *dot* operators) grouped in each step of the computation. This parameter is often referred to as the *depth* of the adder. Considering that in a recurrence solver adder all the *dot* operators have the same architecture and therefore the same delay, to a greater depth corresponds greater delay. Moreover, a great number of subsequent logic gates significantly increases the fan-out associated to each bit of the operands.
A possible way to reduce the depth of an adder is the implementation through radix-4 architecture. All the recurrence solver adders analyzed so far were implemented by using radix-2 architecture, which means coupling two signals into a *dot* operator so that it is possible to compute generate and propagate functions over a group of two bits. As the name suggests, radix-4 architecture groups four signals into a single *dot* operator, computing generate and propagate for a group of four bits. It is thus possible to redefine equation (1.4.2) for a group of four bits as follows:

$$\boldsymbol{Dot_4}\Big((G_{i+3}, P_{i+3}), (G_{i+2}, P_{i+2}), (G_i + 1, P_{i+1}), (G_i, P_i)\Big) = \\ (G_{i+3} + P_{i+3}G_{i+2} + P_{i+3}P_{i+2}G_{i+1} + P_{i+3}P_{i+2}P_{i+1}G_i, P_{i+3}P_{i+2}P_{i+1}P_i) \tag{1.5.1}$$

Consequently recurrence solver adders can be implemented by a *quaternary tree* instead of a binary one. Figure 1.14 shows a 16-bit radix-4 Kogge-Stone adder. Note that in addition to the radix-4 *dot* operators the adder also features radix-2 *dot* operators (depicted as a white circle) and radix-4 *grey* operators (depicted as a grey circle). The former implement equation (1.4.2) as for radix-2 adders, the latter implement the following equation:

$$\boldsymbol{Grey_4}\Big((G_{i+2}, P_{i+2}),(G_i + 1, P_{i+1}), (G_i, P_i)\Big) = \\ = (G_{i+2} + P_{i+2}G_{i+1} + P_{i+2}P_{i+1}G_i, P_{i+2}P_{i+1}P_i) \tag{1.5.2}$$

A possible implementation in CMOS Domino logic of radix-4 *dot* and *grey* operators is shown in Figure 1.15 and Figure 1.16 respectively. As for the implementation of the radix-2 *dot* operator presented in the previous paragraph, both modules feature a bleeder in order to eliminate every static power consumption originated by the pull-down network.
By applying radix-4 architecture it is possible to reduce the depth of the adder and therefore decrease the number of carries grouped in each step. For example, for a 16-bit Kogge-Stone adder, the depth decreases from four to two.

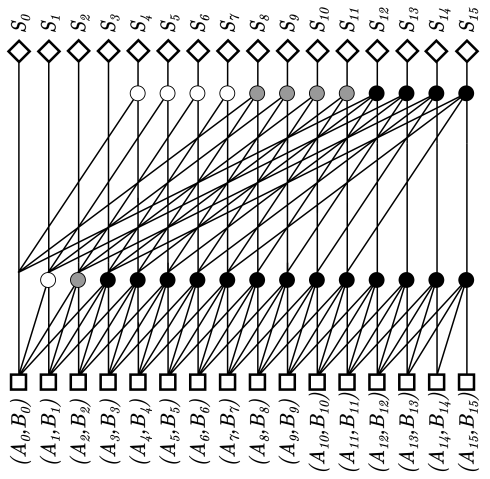


Figure 1.14: Tree structure of a 16-bit radix-4 Kogge-Stone adder.

Nevertheless, it is necessary to consider the fact that radix-4 nodes are more complex than radix-2 ones and the resulting quaternary-tree-based adder could result slower than the binary-tree-based one. Moreover, despite the computational speed, radix-4 gates require greater power consumption than radix-2 ones, making radix-4 adders more power-consuming.

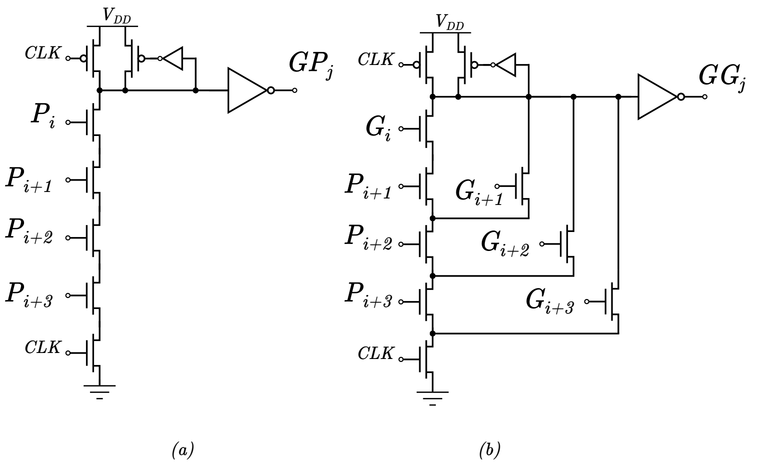


Figure 1.15: CMOS Domino implementation of the radix-4 *dot* operator: *(a) group-propagate* and *(b) group-generate* signals.

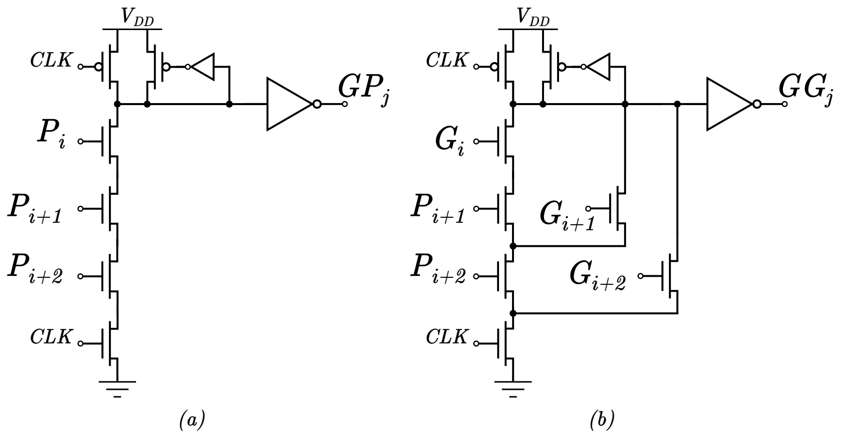


Figure 1.16: CMOS Domino implementation of the radix-4 *grey* operator: *(a) group-propagate* and *(b) group-generate* signals.

Chapter 2

Design of a 32-bit High-Speed Adder

Since adders occupy a critical position inside arithmetic logic units of microprocessors, it is very important to ensure that their performance adequately meets given specifications on speed, power and area occupation. The optimization of such specifications can be achieved at both logical and circuit level. The previous chapter featured the analysis of the logical level, presenting many efficient topological solutions in order to implement high-speed adders. The purpose of the second part of this work is to verify the results presented in the previous chapter by implementing notable adder structures and verifying their performance through simulations.
The adder topologies chosen for simulation are the following: 32-bit Kogge-Stone radix-2 adder; 32-bit Brent-Kung radix-2 adder; 32-bit Kogge-Stone radix-4 adder and 32-bit Brent-Kung radix-4 adder. Kogge-Stone topology was mainly chosen because it ensures high performance despite a considerable power consumption and area occupation, while Brent-Kung topology ensures lower computational speed but also lower area occupation and power consumption. Moreover, a further comparison can be made between radix-2 and radix-4 architectures can be made in terms of overall computational speed and area occupation. Circuit simulation for verification of performance was carried out by using *Cadence Design Framework II* ®, an electronic design automated software for digital, analog and mixed circuits.

2.1 Circuit implementation

Because of its peculiar properties, for circuit implementation Dynamic Domino logic was chosen. In particular, since each operation is free from glitches as each gate can make only one $0 \rightarrow 1$ transition in evaluation, Domino logic is particularly apt to implement cascades of logic gates as the ones of recurrence

solver adders. Moreover, as all Dynamic logic does, the area occupation is much smaller and computational speed is much faster than the conventional CMOS logic.
Given the modular structure of recurrence solver adders, many circuit blocks are shared between different adder topologies. For example *dot* operator is identically implemented for both Kogge-Stone and Brent-Kung adders. Because of this reason, this paragraph will deal with each module separately before introducing the proper adder topology. All the following blocks were implemented by using the $0.35\mu m$ CMOS C35 process.

2.1.1 Blocks implementation

The first stage for each recurrence solver adder is the block that computes both *generate* and *propagate* functions from the operands bit values A_i and B_i. Since in the tree diagrams of recurrence solver adders this function was depicted as a square, it will be henceforth called *square* operator. Its implementation derives directly from equations (1.1.3) and (1.4.6), and is shown in Figure 2.1. Note that, differently from the implementation already presented in paragraph 1.4, the circuits presented in this paragraph do not include a *bleeder* p-MOS as its presence may influence the evaluation of computational speed of the adder.
Depending on the topology of the adder, *square* block is followed either by *dot* or *grey* operators. Radix-2 and radix-4 implementations of the former are shown in Figure 2.2 and Figure 2.3 respectively; while the implementation of the latter is shown in Figure 2.4.
The final stage of the adder is the one that computes the sum from *group-generate* and *group-propagate* signals. Similarly to *square* operator, since in the tree diagrams it is shown as a diamond, this block will be henceforth referred to as *diamond* operator. Differently from the implementation presented in paragraph 1.4, the one used for simulations was realized from equations (1.1.5) and (1.4.5) and is shown in Figure 2.5. Basically, *diamond* operator consists in the cascade of two logic gates: the first computes the carry output at the k-th stage $C_{o,k}$ while the second computes the sum as an exclusive or of the *group-propagate* signal $P_{k:0}$ and the carry output signal.

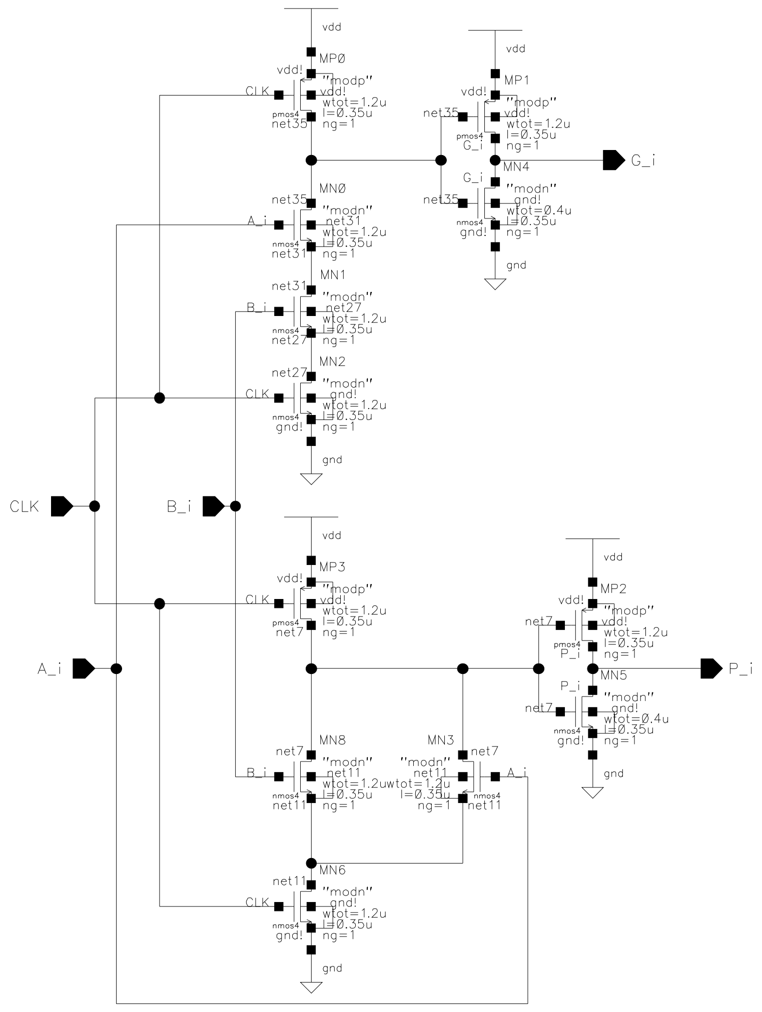


Figure 2.1: Schematic view of the *square* block.

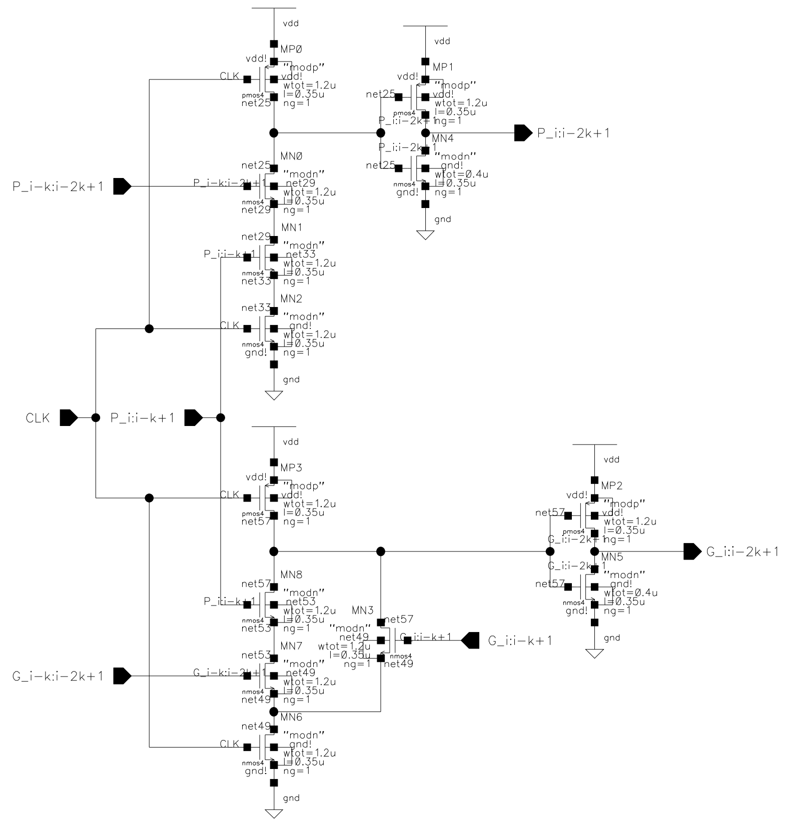


Figure 2.2: Schematic view of the radix-2 *dot* operator.

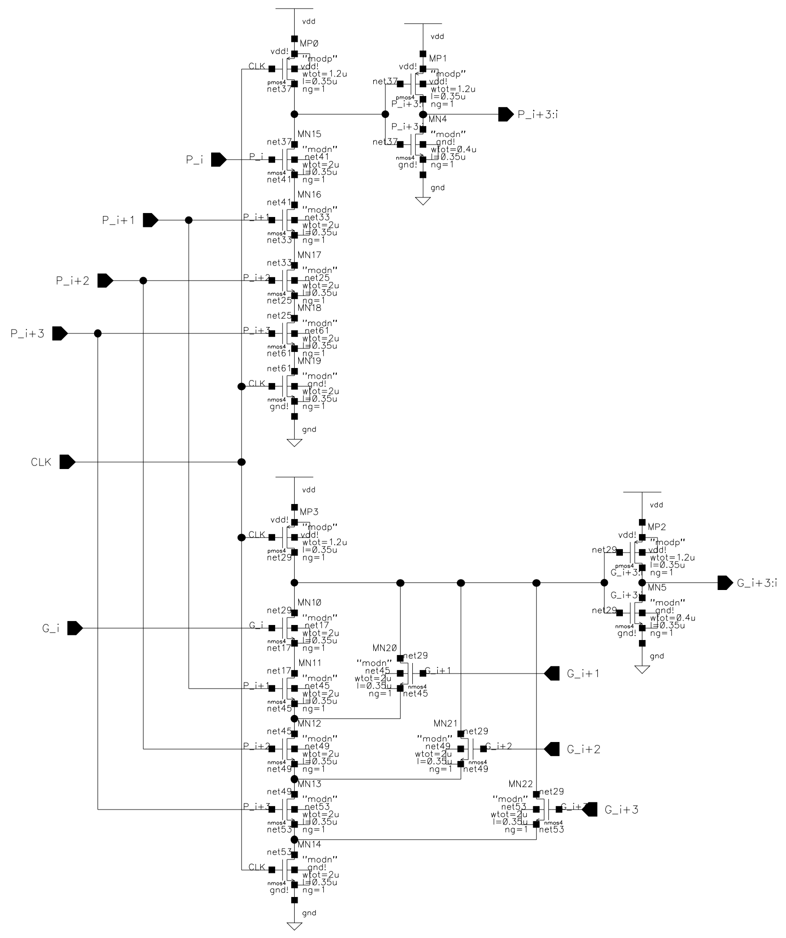


Figure 2.3: Schematic view of the radix-4 *dot* operator.

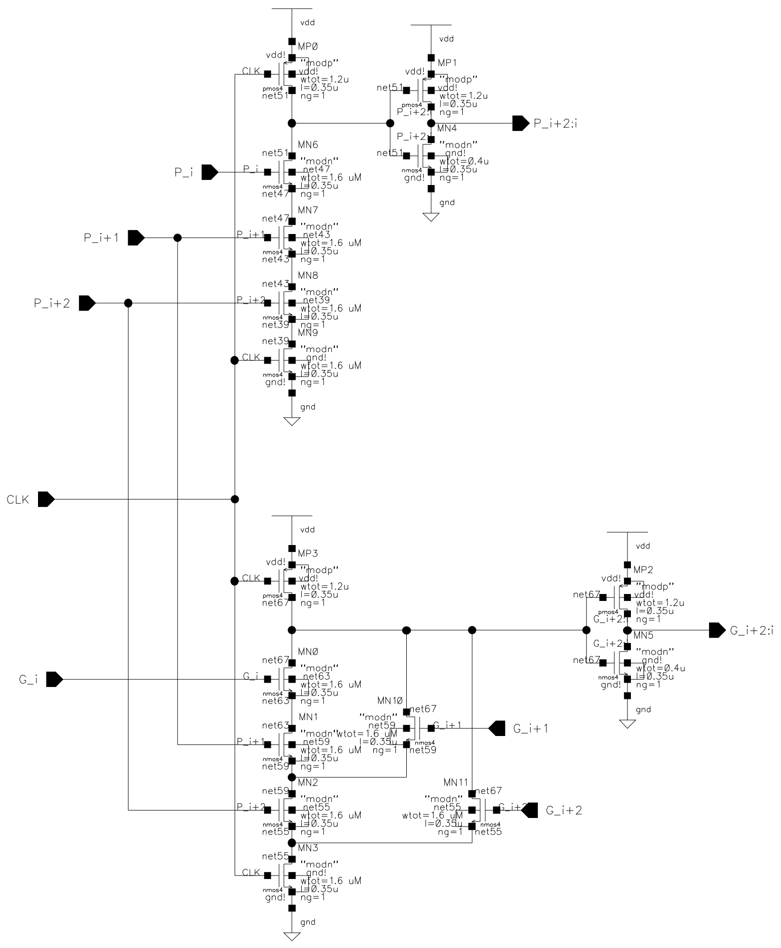


Figure 2.4: Schematic view of the radix-4 *grey* operator.

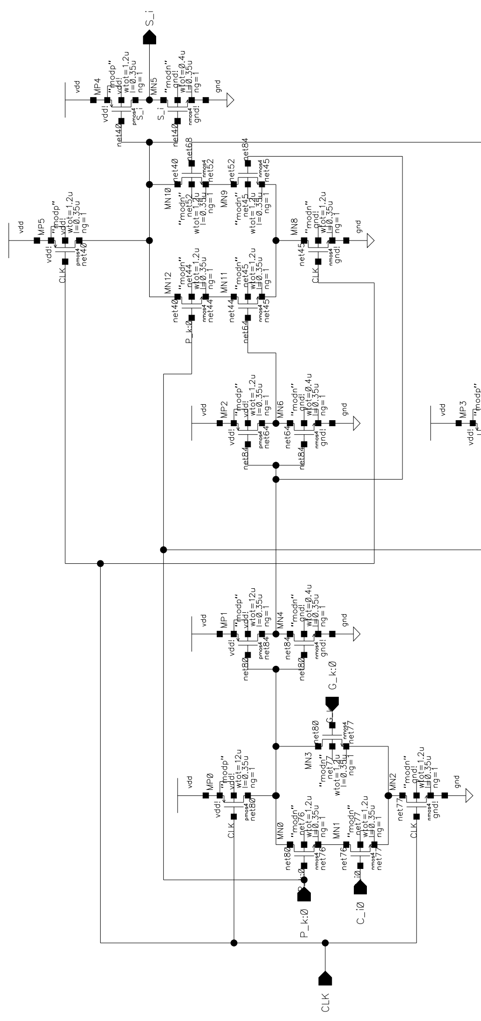


Figure 2.5: Schematic view of the *diamond* block.

2.1.2 32-bit Kogge-Stone adder implementation

The implementation of a 32-bit Kogge-Stone radix-2 adder directly derives from the tree structure presented in Figure 2.6. Since the implementation chosen for the sum requires both *generate* and *propagate* signals, all the *grey* operators, which normally compute only the *generate* signal, are replaced with common *dot* operators. Figure 2.7 shows the full 32-bit Kogge-Stone radix-2 adder circuit, while Figure 2.8 and Figure 2.9 show details of the interconnections among the various blocks.

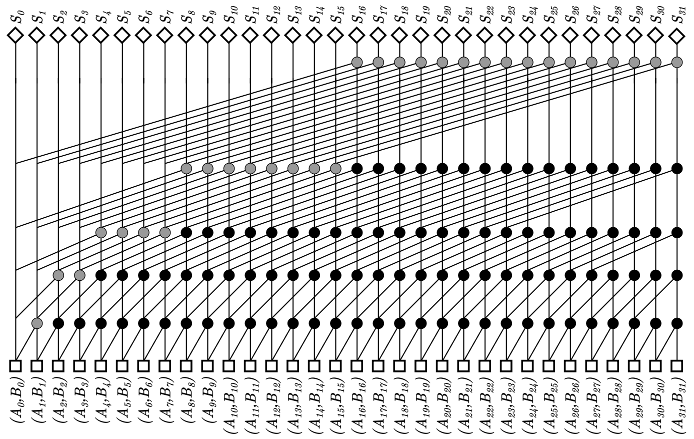

S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15} S_{16} S_{17} S_{18} S_{19} S_{20} S_{21} S_{22} S_{23} S_{24} S_{25} S_{26} S_{27} S_{28} S_{29} S_{30} S_{31}
(A_0,B_0) (A_1,B_1) (A_2,B_2) (A_3,B_3) (A_4,B_4) (A_5,B_5) (A_6,B_6) (A_7,B_7) (A_8,B_8) (A_9,B_9) (A_{10},B_{10}) (A_{11},B_{11}) (A_{12},B_{12}) (A_{13},B_{13}) (A_{14},B_{14}) (A_{15},B_{15}) (A_{16},B_{16}) (A_{17},B_{17}) (A_{18},B_{18}) (A_{19},B_{19}) (A_{20},B_{20}) (A_{21},B_{21}) (A_{22},B_{22}) (A_{23},B_{23}) (A_{24},B_{24}) (A_{25},B_{25}) (A_{26},B_{26}) (A_{27},B_{27}) (A_{28},B_{28}) (A_{29},B_{29}) (A_{30},B_{30}) (A_{31},B_{31})

Figure 2.6: Tree structure of a 32-bit Kogge-Stone radix-2 adder.

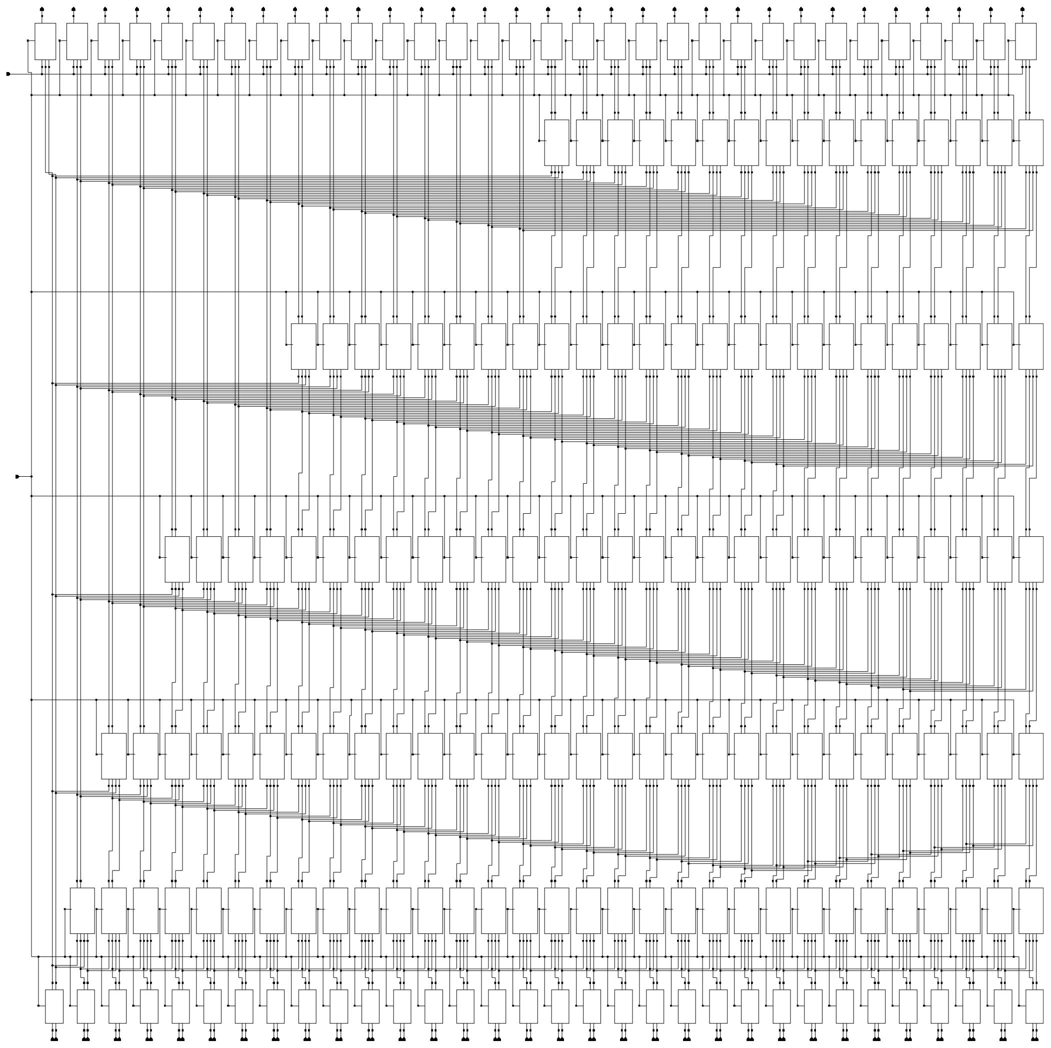


Figure 2.7: Schematic view of a 32-bit Kogge-Stone radix-2 adder.

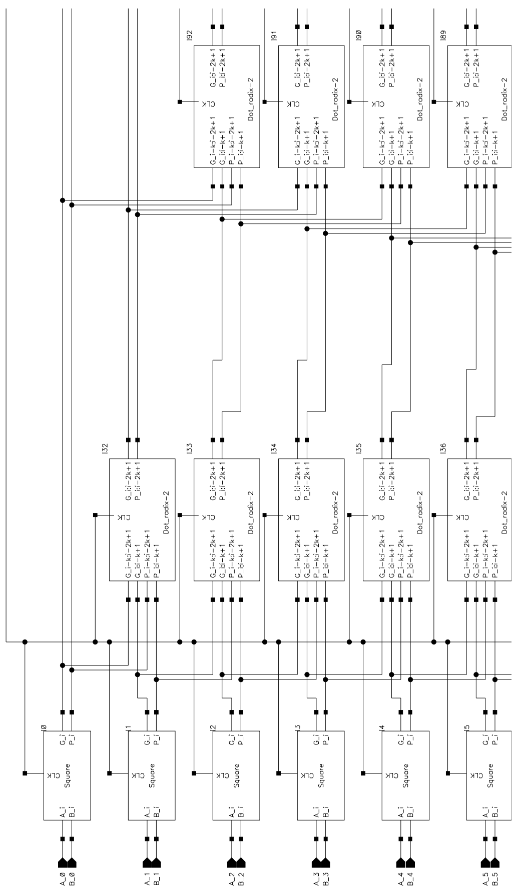


Figure 2.8: Detail showing the first four imputs of of Figure 2.7.

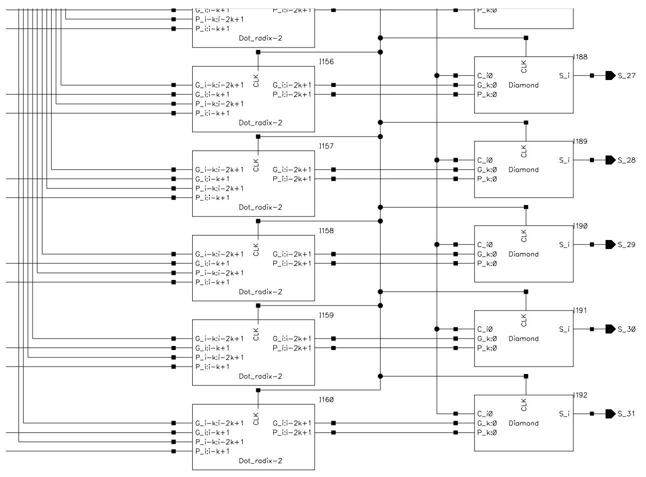


Figure 2.9: Detail showing the last five outputs of Figure 2.7.

Figure 2.10 shows the tree structure of a 32-bit Kogge-Stone radix-4 adder. The white circles represent radix-2 *dot* operators, the grey circles represent radix-4 *grey* operators and the black ones represent radix-4 *dot* operators. The adder implementation on *Cadence* is shown in Figure 2.11, while Figure 2.12 and Figure 2.13 show details of the interconnections among the various blocks.

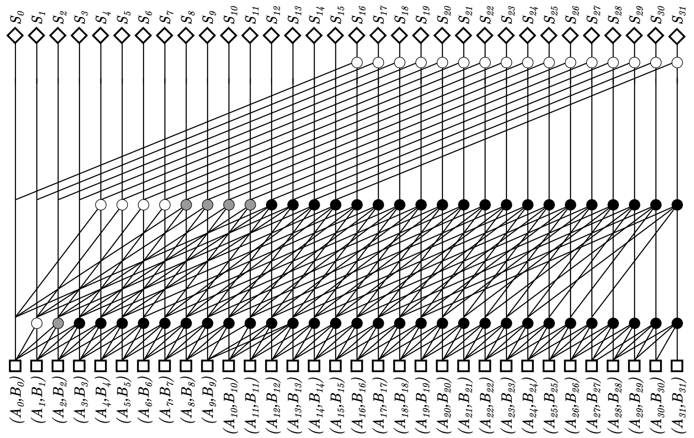

S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15} S_{16} S_{17} S_{18} S_{19} S_{20} S_{21} S_{22} S_{23} S_{24} S_{25} S_{26} S_{27} S_{28} S_{29} S_{30} S_{31}
(A_0,B_0) (A_1,B_1) (A_2,B_2) (A_3,B_3) (A_4,B_4) (A_5,B_5) (A_6,B_6) (A_7,B_7) (A_8,B_8) (A_9,B_9) (A_{10},B_{10}) (A_{11},B_{11}) (A_{12},B_{12}) (A_{13},B_{13}) (A_{14},B_{14}) (A_{15},B_{15}) (A_{16},B_{16}) (A_{17},B_{17}) (A_{18},B_{18}) (A_{19},B_{19}) (A_{20},B_{20}) (A_{21},B_{21}) (A_{22},B_{22}) (A_{23},B_{23}) (A_{24},B_{24}) (A_{25},B_{25}) (A_{26},B_{26}) (A_{27},B_{27}) (A_{28},B_{28}) (A_{29},B_{29}) (A_{30},B_{30}) (A_{31},B_{31})

Figure 2.10: Tree structure of a 32-bit Kogge-Stone radix-2 adder.

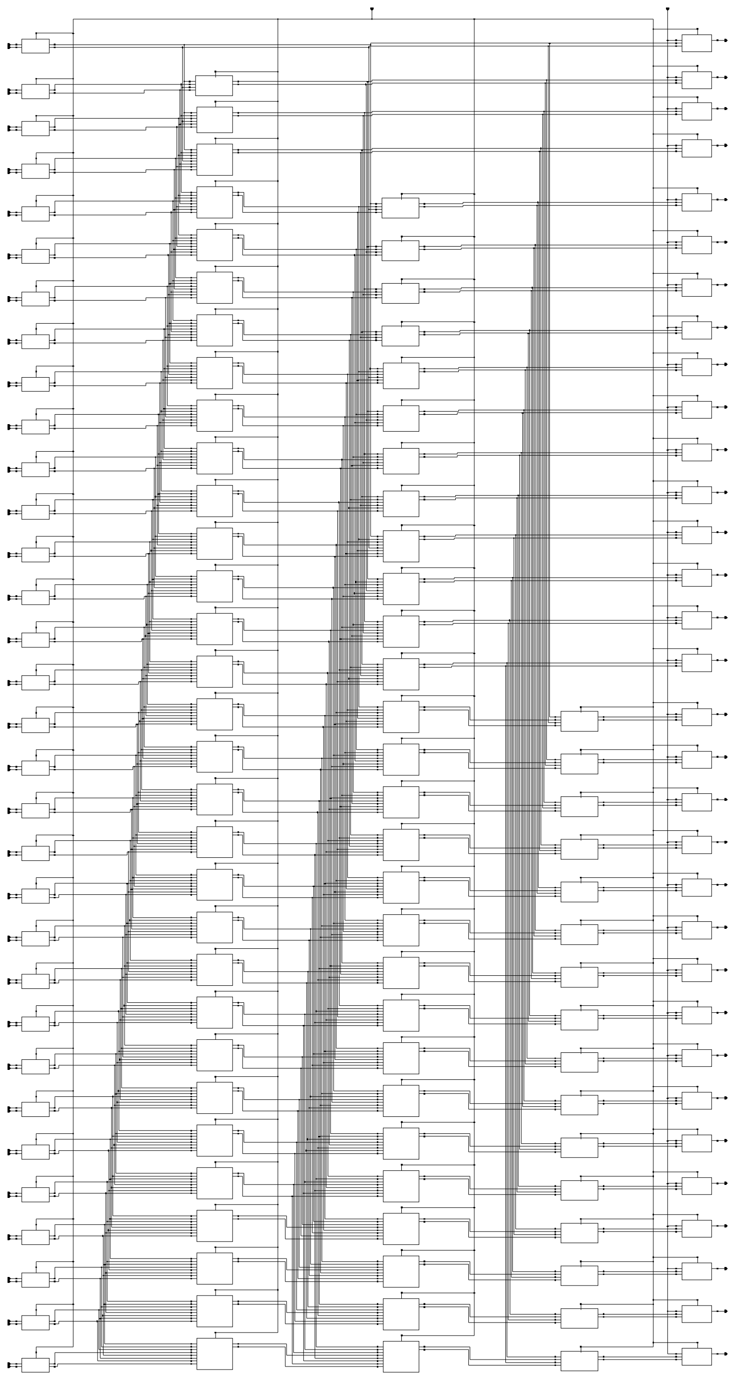


Figure 2.11: Schematic view of a 32-bit Kogge-Stone radix-4 adder.

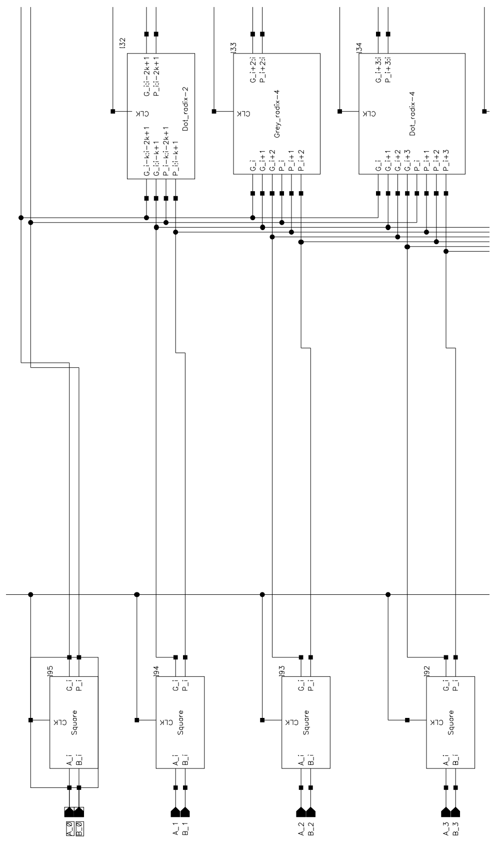


Figure 2.12: Detail showing the first four imputs of of Figure 2.11.

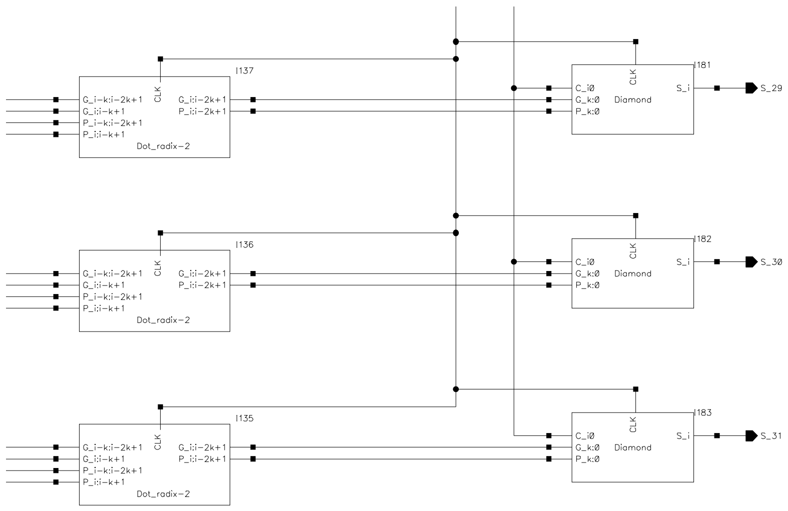


Figure 2.13: Detail showing the last three outputs of Figure 2.11.

2.1.3 32-bit Brent-Kung adder implementation

The implementation of a 32-bit Brent-Kung radix-2 adder directly derives from the tree structure presented in Figure 2.14. Similarly to Kogge-Stone radix-2 implementation, all *grey* operators are replaced with *dot* operators. Figure 2.15 shows the full 32-bit Brent-Kung radix-2 adder circuit, while Figure 2.16 and Figure 2.17 show details of the interconnections among the various blocks.

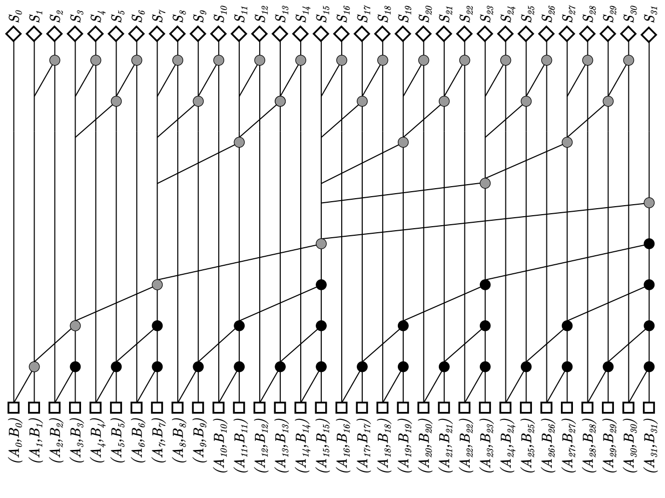


Figure 2.14: Tree structure of a 32-bitBrent-Kung radix-2 adder.

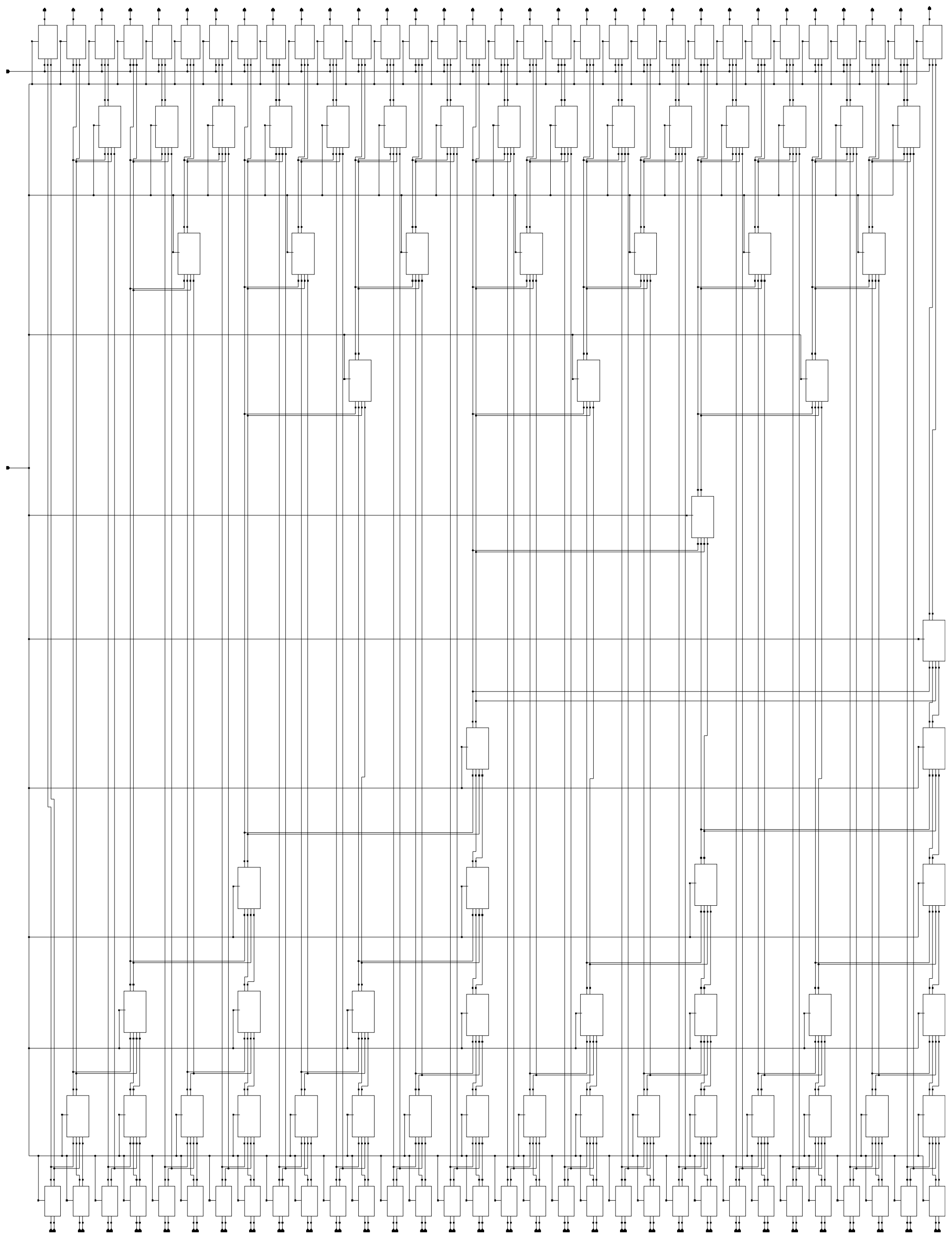


Figure 2.15: Schematic view of a 32-bit Brent-Kung radix-2 adder.

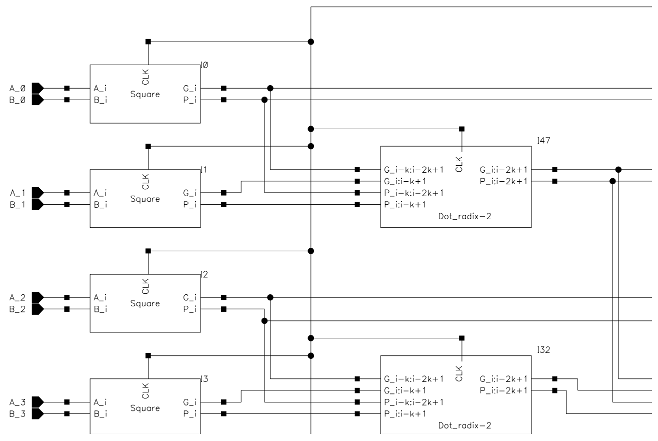


Figure 2.16: Detail showing the first four imputs of of Figure 2.15.

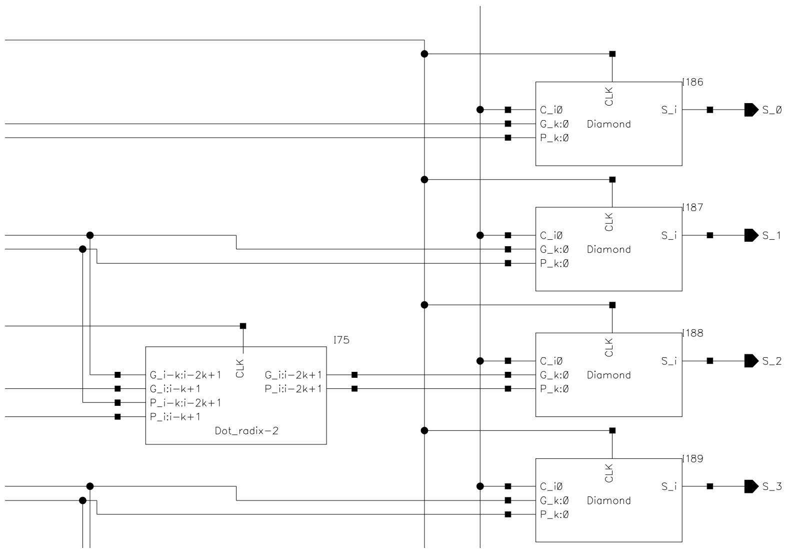


Figure 2.17: Detail showing the first four outputs of Figure 2.15.

Figure 2.18 shows the tree structure of a 32-bit Brent-Kung radix-4 adder. The representation of operators is the same as that used for Kogge-Stone radix-4 adder. The adder implementation on *Cadence* is shown in Figure 2.19, while Figure 2.20 and Figure 2.21 show details of the interconnections among the various blocks.

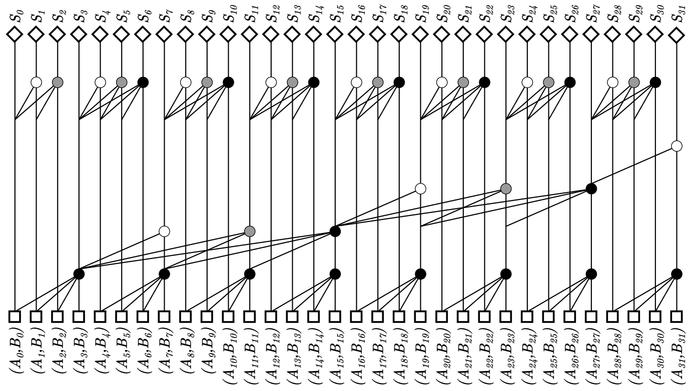

S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13} S_{14} S_{15} S_{16} S_{17} S_{18} S_{19} S_{20} S_{21} S_{22} S_{23} S_{24} S_{25} S_{26} S_{27} S_{28} S_{29} S_{30} S_{31}
(A_0,B_0) (A_1,B_1) (A_2,B_2) (A_3,B_3) (A_4,B_4) (A_5,B_5) (A_6,B_6) (A_7,B_7) (A_8,B_8) (A_9,B_9) (A_{10},B_{10}) (A_{11},B_{11}) (A_{12},B_{12}) (A_{13},B_{13}) (A_{14},B_{14}) (A_{15},B_{15}) (A_{16},B_{16}) (A_{17},B_{17}) (A_{18},B_{18}) (A_{19},B_{19}) (A_{20},B_{20}) (A_{21},B_{21}) (A_{22},B_{22}) (A_{23},B_{23}) (A_{24},B_{24}) (A_{25},B_{25}) (A_{26},B_{26}) (A_{27},B_{27}) (A_{28},B_{28}) (A_{29},B_{29}) (A_{30},B_{30}) (A_{31},B_{31})

Figure 2.18: Tree structure of a 32-bitBrent-Kung radix-4 adder.

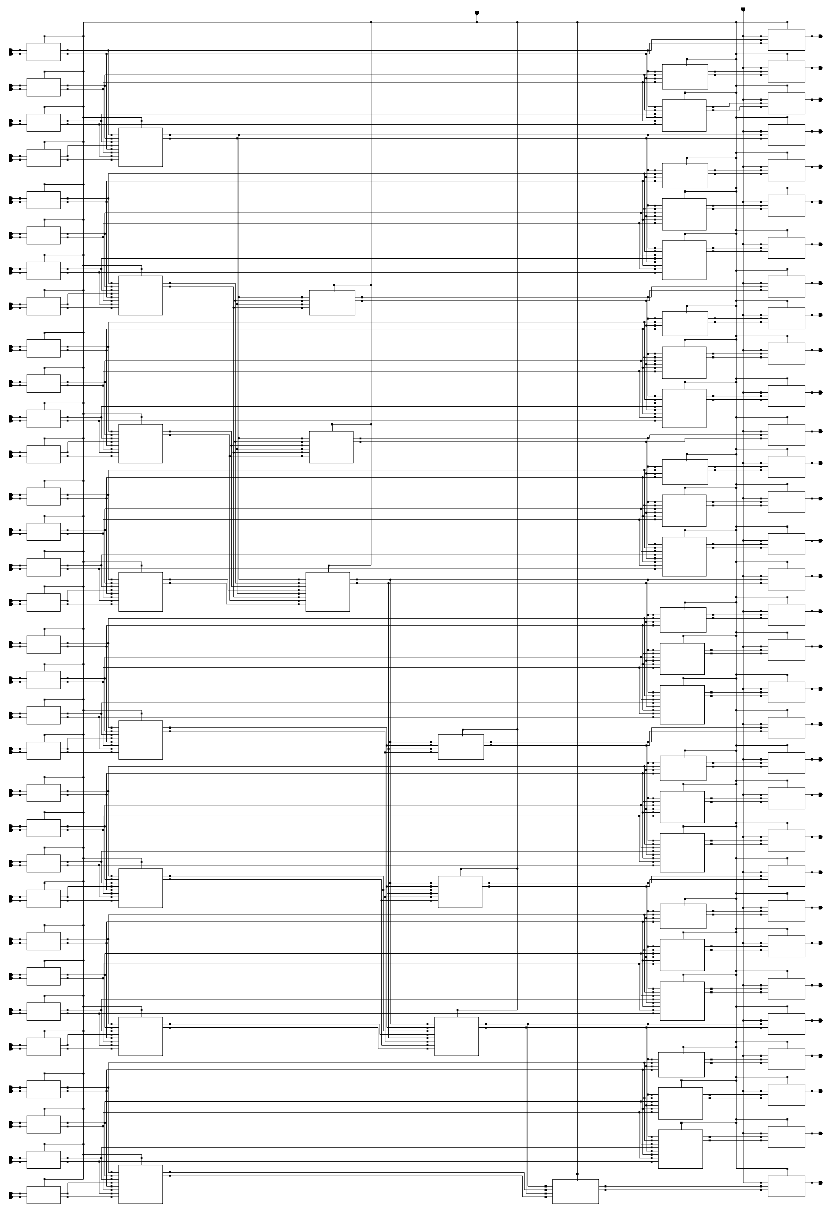


Figure 2.19: Schematic view of a 32-bit Brent-Kung radix-4 adder.

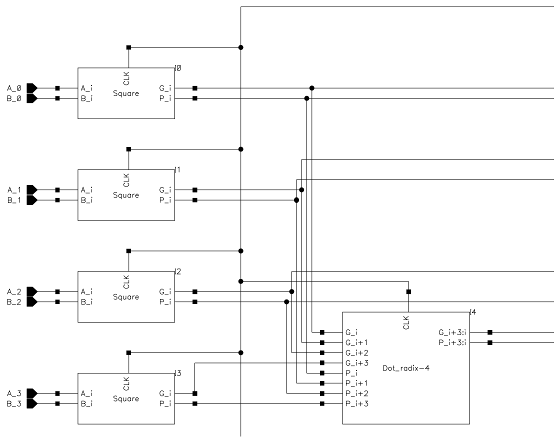


Figure 2.20: Detail showing the first four imputs of of Figure 2.15.

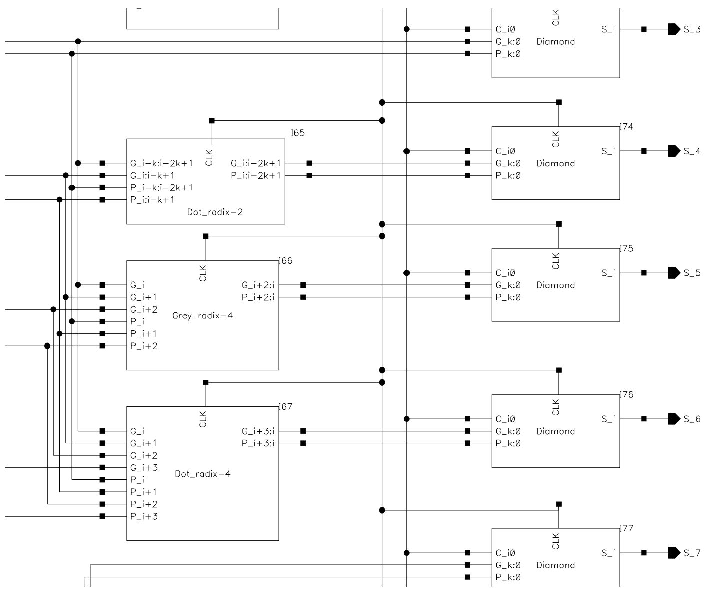


Figure 2.21: Detail showing the first four outputs of Figure 2.15.

2.2 Transistor sizing

In a digital circuit transistors sizing should meet both area occupation and performance specifications. Since the purpose of this work is only to verify the behavior of the examined structures, only preliminary sizing was carried out.

Further changes to the form factors of transistors in order to achieve higher computing performance go beyond the purpose of this work.
Preliminary transistor sizing was carried out in order to ensure to each logic gate the same delay of the minimum size inverter. First of all, $0.35\mu m$ CMOS C35 process allows a minimum transistor gate width of $0.40\mu m$. Since the optimum ratio between the p-MOS and n-MOS gate width in order to ensure symmetrical delays for both low-high and high-low commutations is 3[1], the p-MOS gate width of the minimum size inverter was set to $1.20\mu m$, while the one of the n-MOS was set to the minimum value of $0.40\mu m$.
In a Dynamic logic gate the worst delay is experimented when a series of transistors is active. In fact, by using an equivalent RC model, the delay is approximately given by the following equation:

$$t_p = 0.69 \cdot R \cdot C \tag{2.2.1}$$

where R is the equivalent electrical resistance of the transistor and C the capacitance associated to the concerned node. For a series of n transistors, the high-low delay is given by the following equation:

$$t_{p,HL} = 0.69 \cdot n \cdot R \cdot C \tag{2.2.2}$$

Therefore, being the electrical resistance of the transistor approximately inversely proportional to its gate width, in order to reduce the delay by a factor of n it is necessary to increase the width of each transistor by the same factor. This criterion was used to size all the logic gates of the adder. The following table summarizes the size of the transistors in each logic gate.

Transistor size			
Block	Logic gate	p-MOS	n-MOS (each)
	inverter	$1.20\mu m$	$0.40\mu m$
square	*propagate*	$1.20\mu m$	$1.20\mu m$
	generate	$1.20\mu m$	$0.60\mu m$
dot radix-2	*group-propagate*	$1.20\mu m$	$1.20\mu m$
	group-generate	$1.20\mu m$	$1.20\mu m$
dot radix-4	*group-propagate*	$1.20\mu m$	$2.00\mu m$
	group-generate	$1.20\mu m$	$2.00\mu m$
grey radix-4	*group-propagate*	$1.20\mu m$	$1.60\mu m$
	group-generate	$1.20\mu m$	$1.60\mu m$
diamond	*Carry*	$1.20\mu m$	$1.20\mu m$
	XOR	$1.20\mu m$	$1.20\mu m$

[1]The actual optimum ratio is 2.4, but CMOS technology only allows integer scaling of minimum length and width.

2.3 Simulation

The purpose of the simulation is to verify the performance of the adder by evaluating the delay with which the sum is computed. In order to do that it is necessary to evaluate the critical path of each adder and measure the time elapsed between the rising edges of the clock and the output signal at the end of the critical path at the first evaluation phase. For a recurrence solver adder the critical path is the one that goes from the least significant bit of the operands to the most significant bit of the sum by propagating the carry throughout the whole structure. The propagation of the carry from the least significant bit of the operands to the most significant bit of the sum occurs when the configuration of the operands is one of the following:

$$A = 011 \cdots 111$$
$$B = 000 \cdots 001$$

or, equivalently:

$$A = 111 \cdots 111$$
$$B = 000 \cdots 001$$

where the leftmost bit is the most significant one. The first one will henceforth be refferred to as *configuration 1*, while the second *configuration 2*. In the evaluation of the delay both configurations were used. In the first one the delay was evaluated for the most significant bit of the sum S_{31} and the previous carry output $C_{o,30}$, while for the second the delay was evaluated only for the last carry output $C_{o,31}$. The following graphs and tables summarize the results obtained by simulating the circuits in *Cadence* ® *Virtuoso* ® analog environment by using a 100 MHz clock frequency. The measurements of the delay were performed at 10% (330 mV), 50% (1.65 V) and 90% (2.97 V) of the high output voltage. The following tables summarize the results obtained in terms of delay (measured at 50% of the commutation) and raise time.

Kogge-Stone radix-2 adder configuration 1		
Signal	propagation delay	raise time
CLK	0	0.080ns
S_{31}	1.024ns	0.071ns
$C_{o,30}$	1.093ns	0.113ns

Kogge-Stone radix-2 adder configuration 2		
Signal	propagation delay	raise time
CLK	0	0.080ns
$C_{o,31}$	1.153ns	0.113ns

Brent-Kung radix-2 adder configuration 1		
Signal	propagation delay	raise time
CLK	0	0.080ns
S_{31}	1.482ns	0.071ns
$C_{o,30}$	1.571ns	0.113ns

Brent-Kung radix-2 adder configuration 2		
Signal	propagation delay	raise time
CLK	0	0.080ns
$C_{o,31}$	1.137ns	0.113ns

Kogge-Stone radix-4 adder configuration 1		
Signal	propagation delay	raise time
CLK	0	0.080ns
S_{31}	1.411ns	0.070ns
$C_{o,30}$	1.624ns	0.113ns

Kogge-Stone radix-4 adder configuration 2		
Signal	propagation delay	raise time
CLK	0	0.080ns
$C_{o,31}$	1.594ns	0.113ns

Brent-Kung radix-4 adder configuration 1		
Signal	propagation delay	raise time
CLK	0	0.080ns
S_{31}	2.060ns	0.071ns
$C_{o,30}$	2.032ns	0.114ns

Brent-Kung radix-4 adder configuration 2		
Signal	propagation delay	raise time
CLK	0	0.080ns
$C_{o,31}$	2.119ns	0.113ns

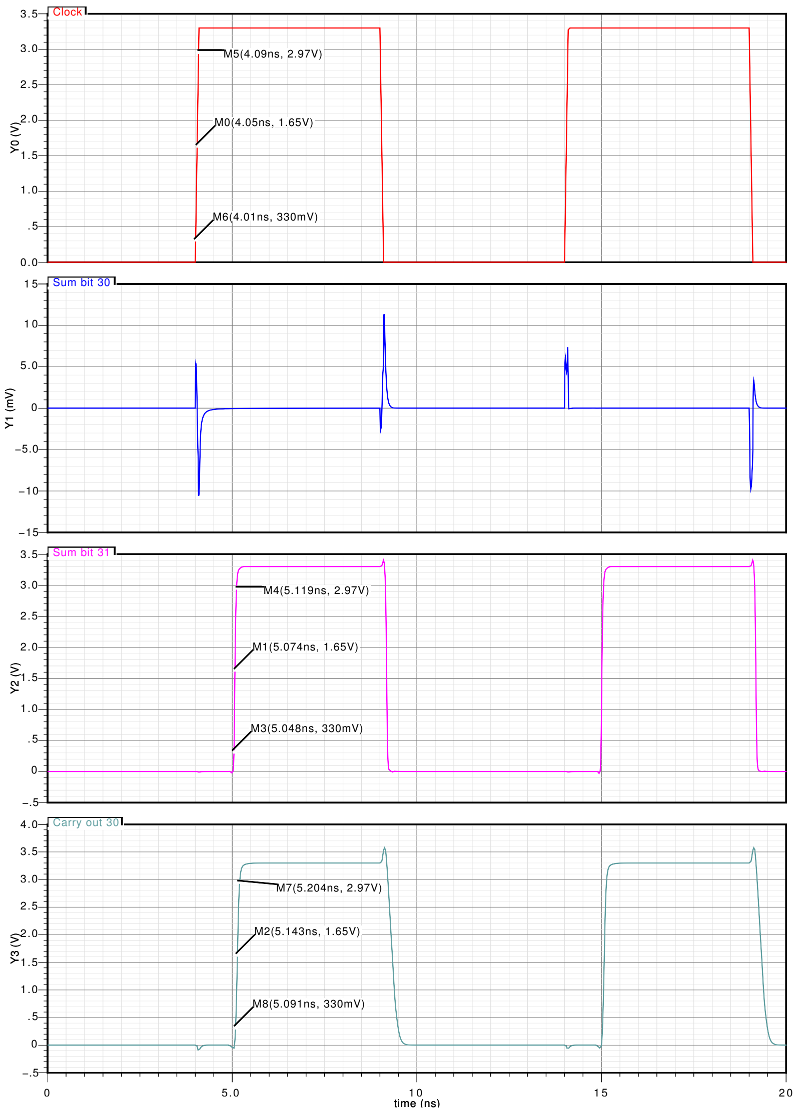


Figure 2.22: Kogge-Stone radix-2 adder configuration 1.

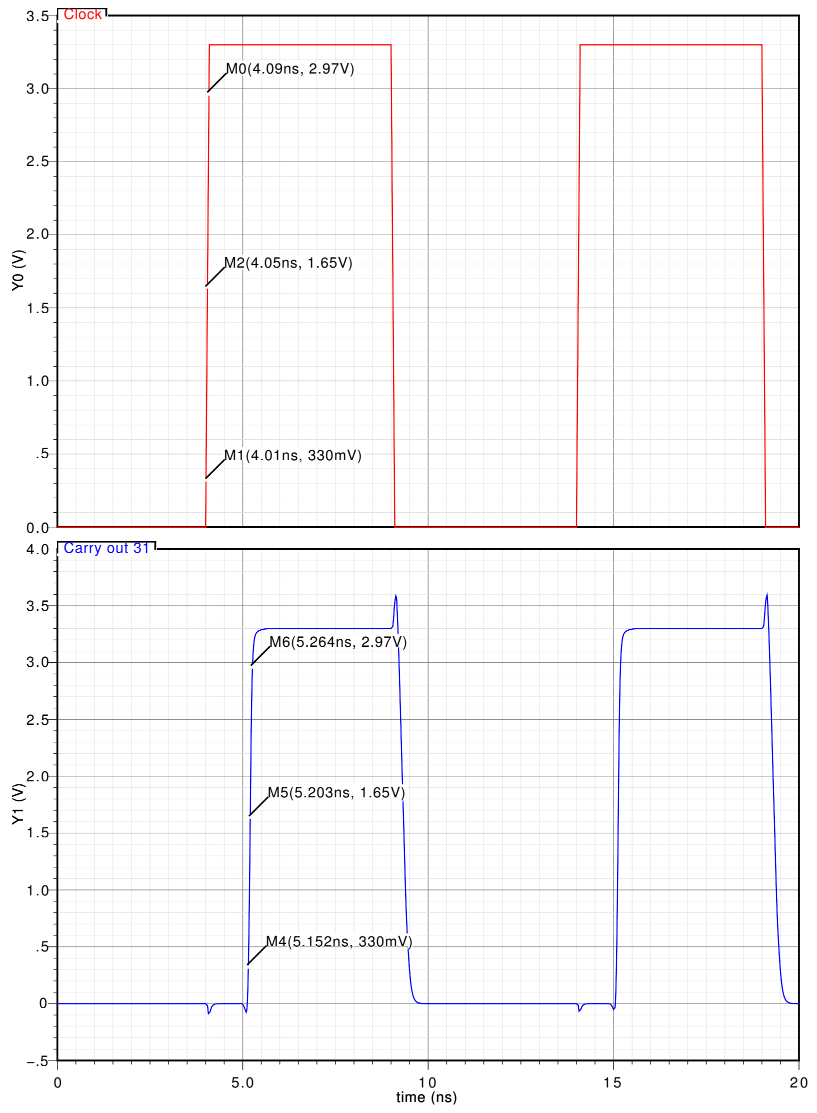


Figure 2.23: Kogge-Stone radix-2 adder configuration 2.

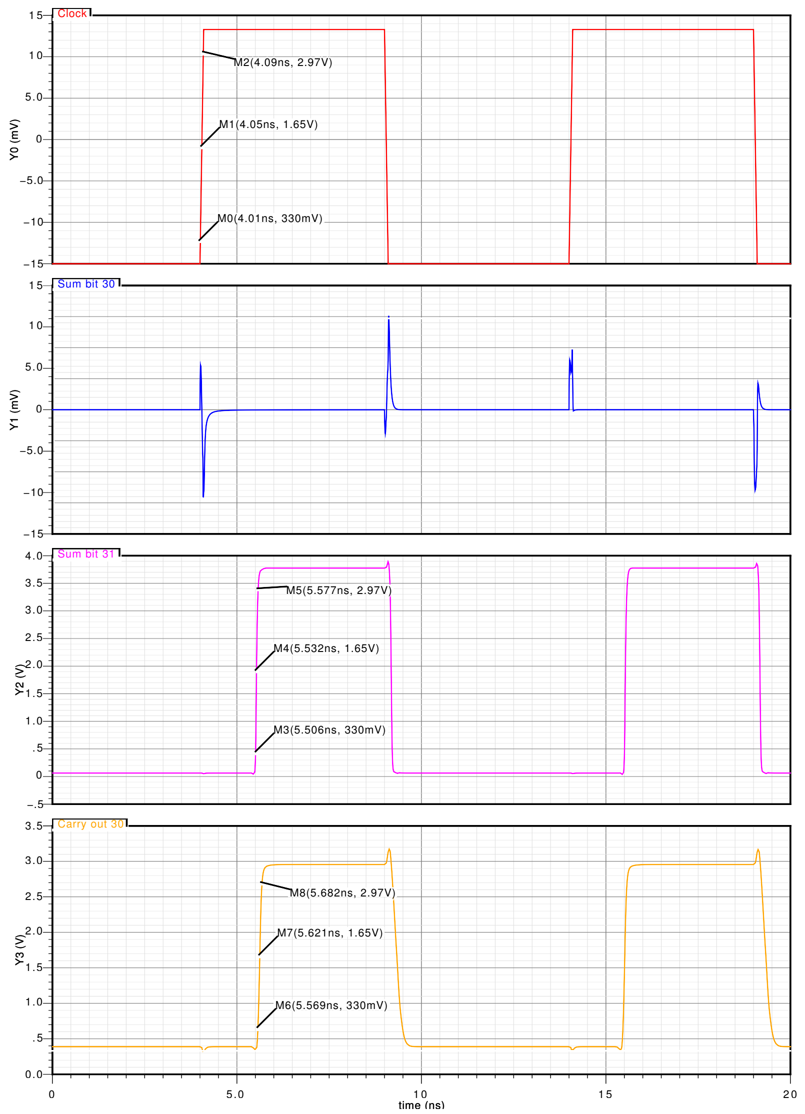


Figure 2.24: Brent-Kung radix-2 adder configuration 1.

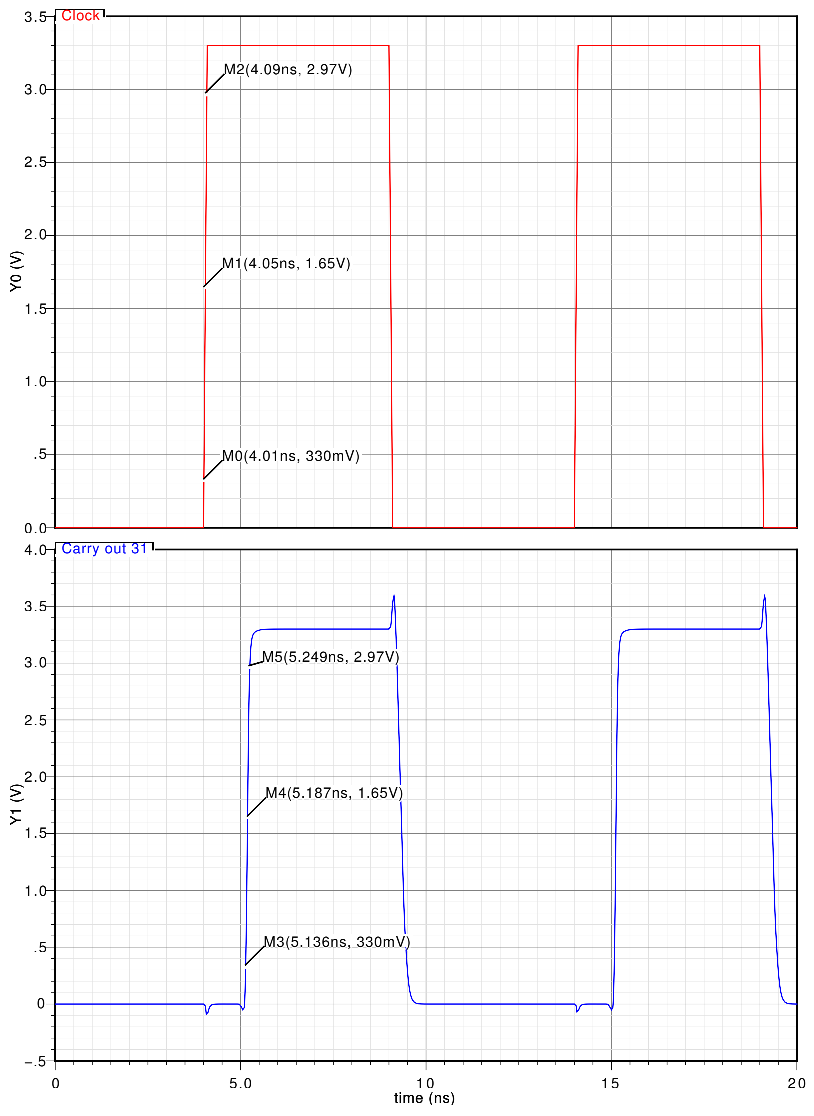


Figure 2.25: Brent-Kung radix-2 adder configuration 2.

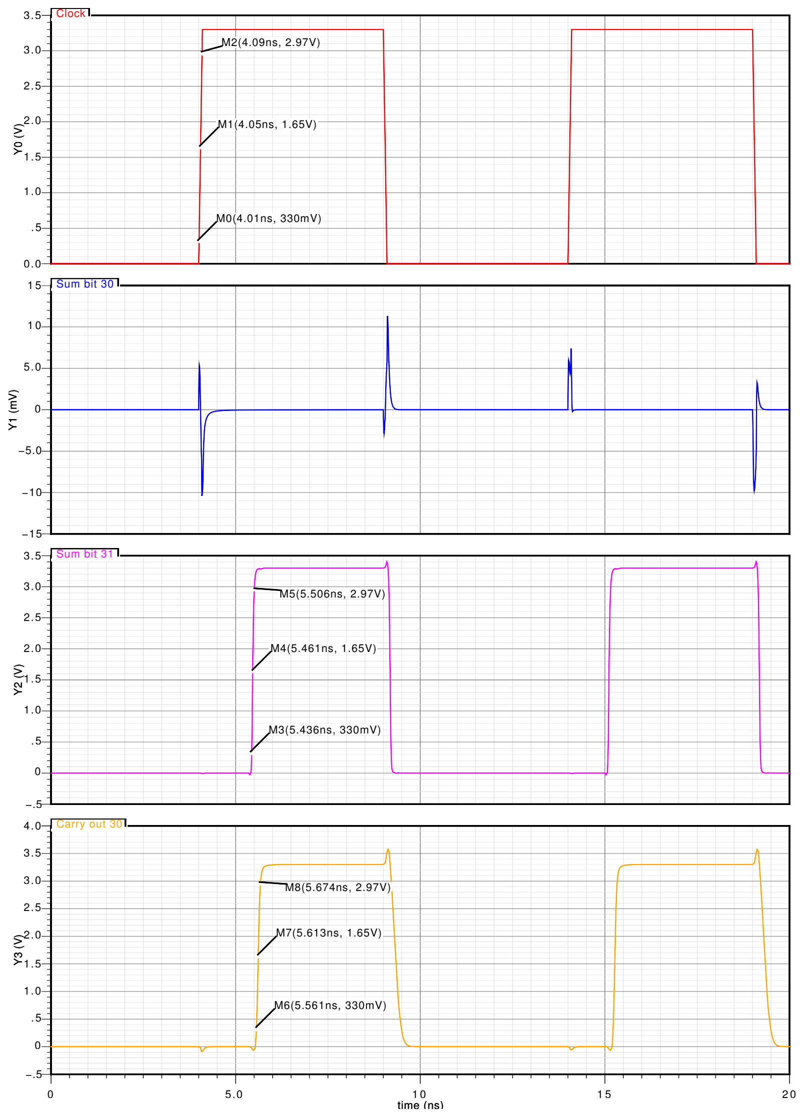


Figure 2.26: Kogge-Stone radix-4 adder configuration 1.

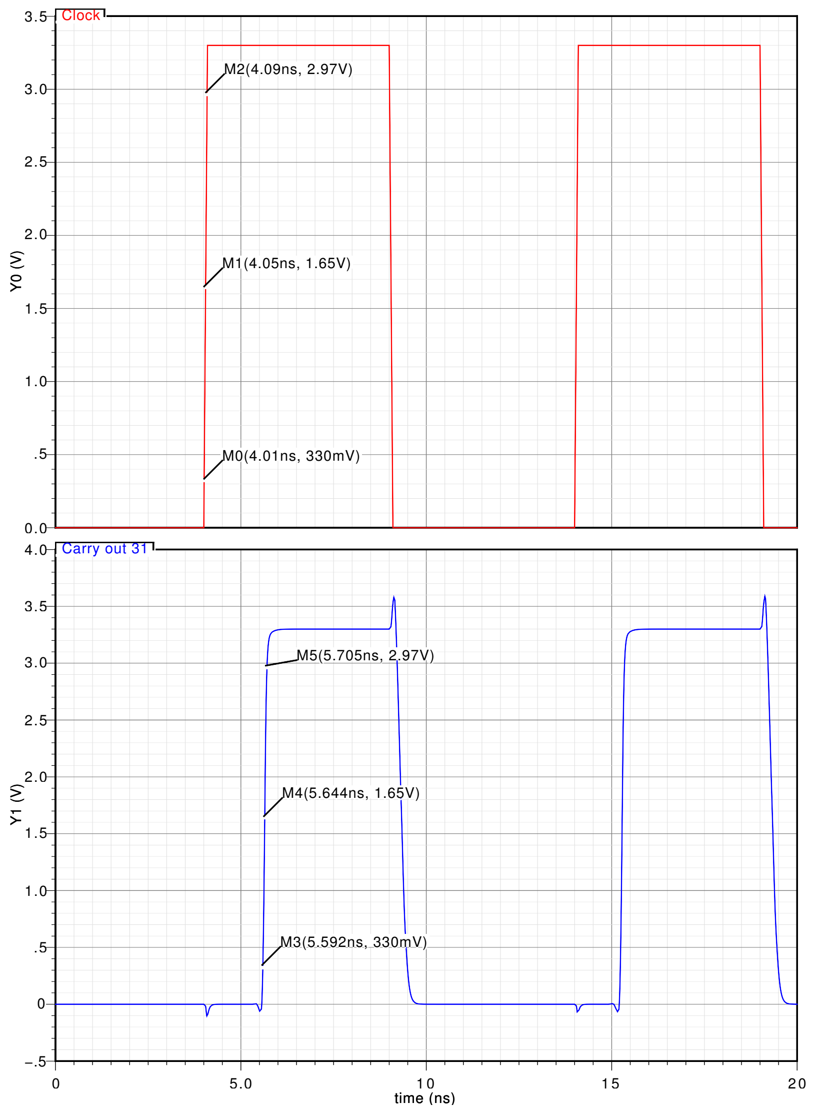


Figure 2.27: Kogge-Stone radix-4 adder configuration 2.

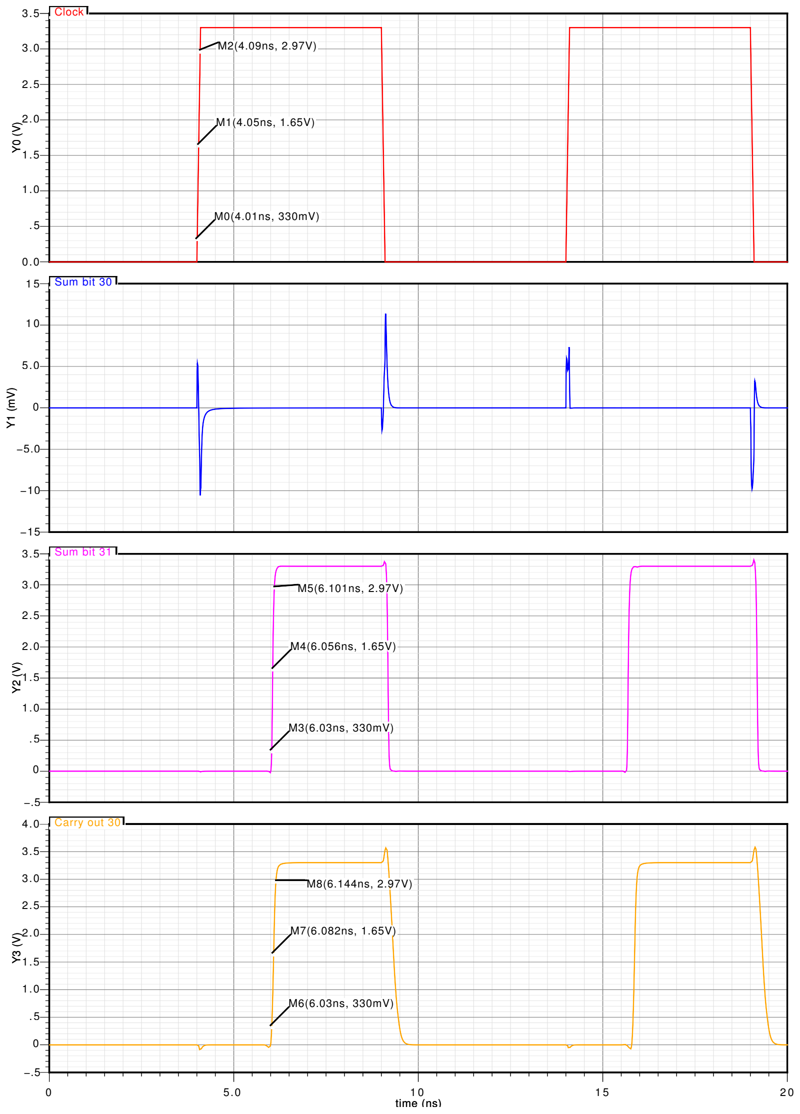


Figure 2.28: Brent-Kung radix-4 adder configuration 1.

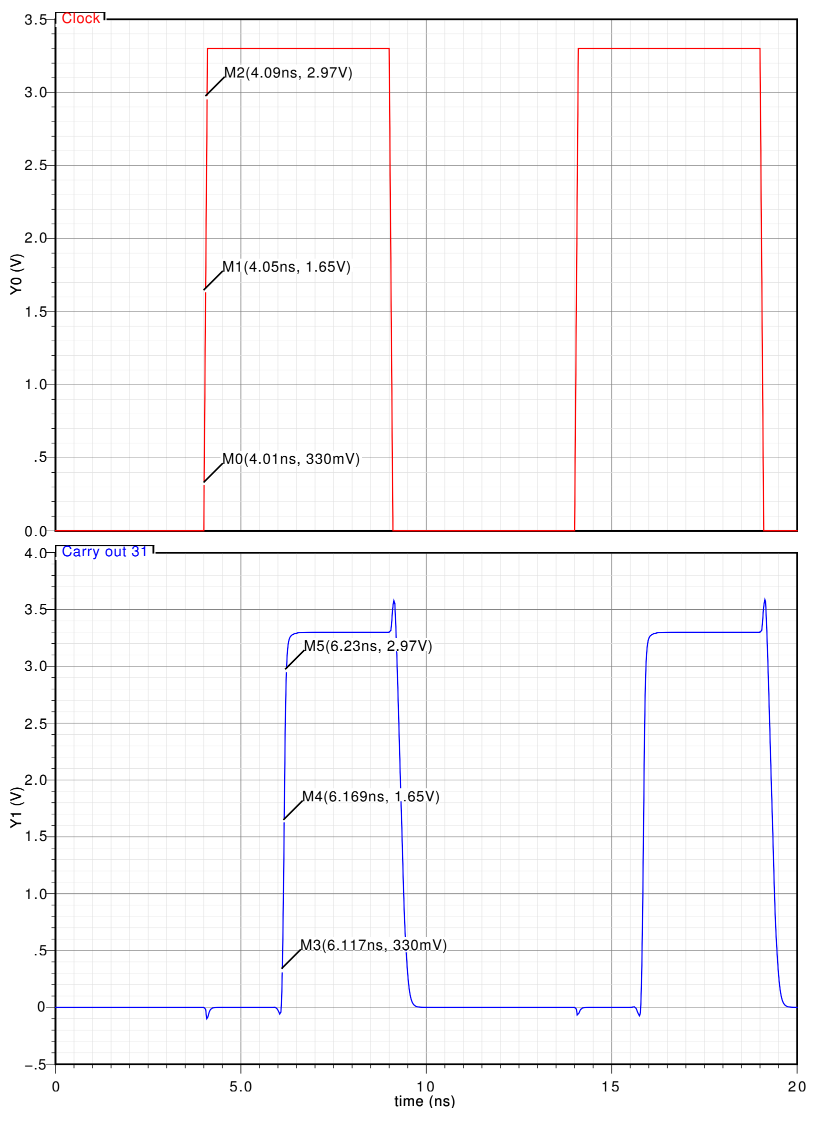


Figure 2.29: Brent-Kung radix-4 adder configuration 2.

Simulations show that the recurrence solver adders tested introduce a delay which has an order of magnitude of nanonseconds. As predicted, in both radix-2 and radix-4 cases Kogge-Stone adder topology has the highest computational speed compared to Brent-Kung architecture. On the other hand, area occupation of Kogge-Stone adder is much greater than the one used to implement Brent-Kung adder: 2669 to 1733 transistors for the radix-2 case and 1896 to 1618 for the radix-4 case. Consequently, the power consumption of the Kogge-Stone topology is much greater than the one of Brent-Kung topology.

Conclusion

As a conclusion to the work, the importance of both theory and simulation in the design of a complex logic circuit must be underlined. On the one hand, theoretical research lays the fundation to the creation of increasingly performant structures by perfectioning the logical and topological aspects of the circuit. On the other hand, simulation and verification of the theoretical results represents an essential part of the design of all digital and analog circuits. In fact, manual calculations alone are insufficient in order to implement a circuit and can only be regarded as a first step towards the realization of the system. Moreover, further optimization of the circuit can be achieved only by ulterior calculations and simulations through which the designer can discern various structures in order to meet specifications in terms of computational performance, power consumption and area occupation.

Bibliography

[1] Jan M. Rabaey, Anantha Chandrakasan, Borivoje Nikolić, *Digital Integrated Circuits, A Design Perspective*, Prentice Hall, 2^{nd} edition, 2003. ISBN: 0-13-120764-4.

[2] R. Brent and H.T. Kung, "A regular Layout for Parallel Adders", *IEEE Trans. on Computers*, vol. C-31, no. 3, pp. 260-264, March 1982.

[3] P.M. Kogge and H.S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", *IEEE Trans. on Computers*, vol. C-22, pp. 786-793, August 1973.

[4] Hoang Q. Dao and Vojin Oklobdzija, "Performance Comparison of VLSI Adders Using Logical Effort", PATMOS 2002, LNCS 2451, pp. 25-34, 2002, Springer-Verlag Berlin Heidelberg.

[5] Vojin Oklobdzija, "High-Speed VLSI Arithmetic Units: Adders and Multipliers in Design of High-Performance Microprocessor Circuits", Book Chapter 4, Book edited by A. Chandrakasan, IEEE Press, 2000.

[6] J. A. Abraham, "Design of Adders", Lecture slides, EE-382M.7, Department of Electrical and Computer Engineering, The University of Texas at Austin, September 21, 2011.