POLITECNICO DI TORINO
Electronic Engineering

Integrated Systems Technology

Project Report
Multiplier area, performance and power estimator

Group 4
Alessandro Di Paola 255174
Gianluca Tortora 238794
Riccardo Vitetta 243740

Contents

1 Introduction

The goal of this project is to create a flexible MATLAB script for the estimation of area, critical path (frequency) and power (static, dynamic) of a user-defined multiplier. Simplified models were used just for the sake of comparison between different multiplier types, hence not for gathering accurate data. The multipliers considered in this project are based on a parallel approach, in which all partial products to be summed together are computed at once. Such multipliers can be generally divided in three parts:

- Partial products generation circuit;
- Reduction tree, through which multiple operands are reduced to the sum of just two operands. In case of a radix-2 NxN multiplier the number of operands to be added will be N, in case of radix-4 the number of operands will reduce to N/2 and so on;
- Final (conventional) adder that performs the two-operand addition. Usually a fast adder is exploited: it allows a logarithmic critical path length increase w.r.t. multiplier width N.

The defined MATLAB script will allow the user to build heterogeneous multiplier architectures exploiting this concept. The script will generally ask to the user the insertion of four parameters, namely:

- ”N”, multiplier operands width, with 256 as the maximum allowed value;
- ”a”, that defines the multiplier type (Baugh-Wooley, Dadda, Wallace, MBE, Even-Odd), hence in general the partial product generation circuit;
- ”b”, that defines the reduction tree type (Dadda, Wallace);
- ”c”, that defines the final two-operand adder type (Ripple-Carry adder, parallel prefix approaches such as Ladner-Fischer, Brent-Kung, Kogge-Stone and finally Carry Look-Ahead adder).

In case of Baugh-Wooley multiplier ”b” and ”c” parameters won’t be asked, while in case of Even-Odd multiplier ”b” parameter won’t be asked, since these are less generalisable architectures.
Of course in case of Dadda and Wallace multipliers the ”b” parameter won’t be asked since already specified with ”a”.

2 Technological parameters, basic blocks and multiplier types

In this chapter, the theoretical analysis for the estimation of parameters such as area, power consumption (static and dynamic) and delay has been performed. The reference technological parameters are taken from the International Technology Roadmap for Semiconductors (ITRS), 2009 edition. Starting from it, each block is built using standard CMOS gates, by adopting different mathematical solutions and considerations. In particular, the basic gates that have been considered are the inverter and the two inputs NAND. These are the bricks with which more complex structures are made up of: full adders and half adders, mainly, and for higher level multipliers as a whole. The technological parameters are reported in table 1.

Parameter	Value	Unit	Description
L_g	27	[nm]	Gate length
V_{dd}	0.97	[V]	Supply voltage
I_{on}	1200	[uA/um]	Saturation current
MP_half	45	[nm]	Half metal pitch
β	1.29	-	Electron/hole mobility ratio
C_{g_ideal}	0.73	[fF/um]	Gate capacitance
$C_{g\_fringing}$	0.25	[fF/um]	Gate fringing capacitance
I_{leak}	100	[nA/um]	Leakage current
J_{g_max}	0.83	[kA/cm^2]	Gate current density
C_{ovl}	$0.2 * C_{g\_ideal}$	[fF/um]	Overlap capacitance
C_{j0}	1	[fF/cm^2]	Junction capacitance, no bias
L_{SD}	L_g	[nm]	S-D regions length
M	1.5	-	Capacitance overhead factor

Table 1

By considering the INV and NAND2 gates as the basic gates to build the multiplier, all the output parameters involved in this analysis are computed by taking into account the number of these logic bricks inside the structure. For instance the total area (power consumption) of the architecture should be equal to the number of NAND2 multiplied by the area (power consumption) of a single NAND2 gate, and added to the number of inverters multiplied by the area (or power) of this basic gate.

2.1 Basic gates

The idea used to build CMOS gates is to size them in order to obtain the same driving strength of the minimum inverter (with minimum width W_n), that it used as a reference. W_n is assumed to be $10 * L_g$, where L_g is the effective channel length and it's hypothesized to be equal to the length of the S/D extensions. Also, the two nMOS, in the NAND2 gate, must have a width twice w.r.t. the reference nMOS in order to obtain the same current in the pull-down network.
The CMOS technology involves the presence of a nMOS network to pull-down the output voltage, and a dual pMOS network to pull it up. For the pMOS sizing one shall consider that the mobility of the electrons will surely be higher w.r.t. the hole mobility according to a β factor:

$$\mu_n = \beta\mu_p$$

The threshold voltage V_{th} of the two devices (nMOS, pMOS) has been assumed to be the same. In this way the current during the charging and discharging phases will be the same.
The capacitance seen from the input of a standard INV is:

$$C_{FO1} = C_{g\_total}(1+\beta)$$

Where the C_{g_total} is the total gate capacitance, made up of three contributions: the ideal gate capacitance C_{g_ideal}, the fringing capacitance $C_{g\_fringing}$ and the overlapping one C_{ovl}.
The total load capacitance seen from the INV gate, instead, is computed by taking into account the C_{FO1} seen from the out, and the junction capacitances C_j:

$$C_{L,INV} = C_{FO1} + C_j$$

For NAND2 gate it has been considered a load capacitance C_{FO4}, but also interconnects and junction capacitances are considered by taking into account a factor M:

$$C_{FO4} = 4C_{FO1}$$

$$C_{L,NAND2} = C_{FO4}(1+M)$$

In this way, the areas of the INV and NAND2 gates are expressed in table 2:

Port	Area
INV	$(1+\beta)W_nL_{mos}$
NAND2	$2(2+\beta)W_nL_{mos}$

Table 2

It's important to specify that L_{mos} is the sum of L_g with the source and the drain lengths. The

expression used to compute the delay is:

$$\tau = \frac{C_{g\_total}}{I_{on}}V_{dd}$$

For the logic gates involved, it's possible to compute the delays as shown in table 3.

Port	Delay
INV	$\frac{C_{L,INV}}{C_{g\_total}}\tau$
NAND2	$\frac{C_{L,NAND2}}{C_{L,INV}}\tau_{INV}$

Table 3

The two contibutions for the power consumption derive from a dynamic component and a static one. The rule which describes the first contribution is the following:

$$P_{dyn} = \frac{1}{2}\alpha f C_L (V_{dd})^2$$

For the static power all the values, instead, are computed taking in account different possible input combinations, and summing the contributions coming from the gate (gate current) and from the S/D leakages.
In tables 4, 5 the total static current per unit width for the INV, NAND2 gates, respectively, are shown. To have an idea of the total static current, for each gate, it's required to sum all the contributions for all the possible combinations of the input, and then divide them by two (INV) or by four (NAND2).

Input	Output	I_S
0	1	$I_{sd,n} + I_{g,p}$
1	0	$I_{sd,p} + I_{g,n}$

Table 4

Input1	Input2	Output	I_S
0	0	1	$I_{sd,n} + 2I_{g,p}$
0	1	1	$I_{sd,n} + I_{g,p}$
1	0	1	$I_{sd,n} + I_{g,n} + I_{g,p}$
1	1	0	$2I_{sd,p} + 2I_{g,n}$

Table 5

2.2 Basic blocks

In order to implement the multiplier, basic blocks have to be defined. The most important ones are the half-adder (HA) and the full-adder (FA). The basic idea is to implement both structures by using only NAND2 and INV gates. Of course this is an extremely simplified model, in more realistic scenarios much more efficient structures are used. The HA structure is shown in figure 1, while the FA structure is shown in figure 2.

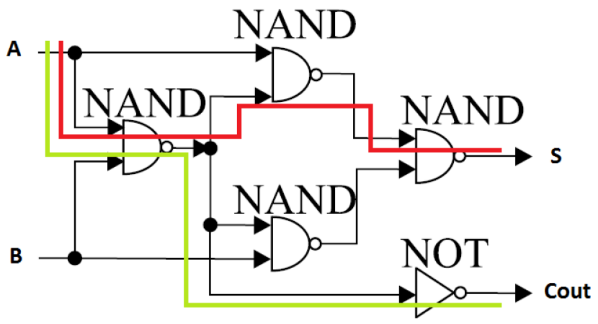

Figure 1 - Half adder, NAND2 and INV gates only

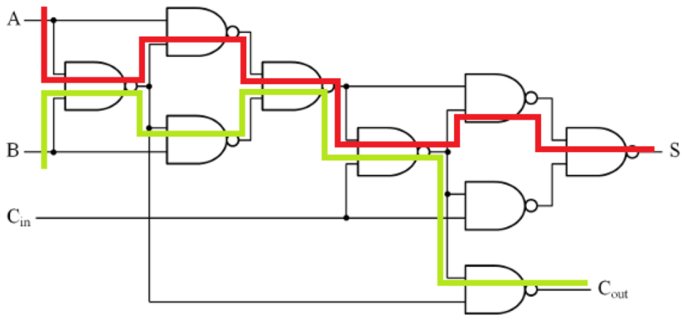

Figure 2 - Full adder, NAND2 gates only

The green and the red lines in figures 1,2 represent the critical paths from the input to, respectively, the carry and the sum outputs. By doing these assumption on HA and FA structures, the delay and area of both blocks can be determined as shown in table 6.

Gate	Area	Delay IN to S	Delay IN to C_{out}
HA	$Area_{INV} + 4Area_{NAND2}$	$3\tau_{NAND2}$	$\tau_{INV} + \tau_{NAND2}$
FA	$9Area_{NAND2}$	$6\tau_{NAND2}$	$5\tau_{NAND2}$

Table 6

2.3 Baugh-Wooley multiplier

The simplest way to implement a signed-inputs (2's complement number representation) array multiplier is to use the Baugh-Wooley algorithm. Starting from the operands A,B shown in the following

$$\begin{aligned} A &= -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \\ B &= -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \end{aligned}$$

One can write their product as

$$P = AB = +a_{n-1}b_{n-1}2^{2n-2} + \sum_{i}^{n-2}\sum_{j}^{n-2} a_i b_j 2^{i+j} - 2^{n-1}\sum_{j=0}^{n-2} a_{n-1}b_j 2^j - 2^{n-1}\sum_{i=0}^{n-2} b_{n-1}a_i 2^i$$

By calling

$$-2^{n-1}\sum_{j=0}^{n-2} a_{n-1}b_j 2^j \rightarrow X \qquad x_j = a_{n-1}b_j$$

$$-2^{n-1}\sum_{i=0}^{n-2} b_{n-1}a_i 2^i \rightarrow Y \qquad y_i = b_{n-1}a_i$$

It is possible to see that the unrolled envelope for each power of two is as shown in figure 3.

	2n-1	2n-2	2n-3	...	n	n-1	n-2	...	1	0
$-X$	1	1	$\overline{x_{n-2}}$	...	$\overline{x_1}$	$\overline{x_0}+1$	0	...	0	0
$-Y$	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0
$-X-Y$	1	0	$\overline{x_{n-2}}+\overline{y_{n-2}}$	...	$\overline{x_1}+\overline{y_1}+1$	$\overline{x_0}+\overline{y_0}$	0	...	0	0

Figure 3 - Baugh-Wooley algorithm

The final architecture for a 4x4 Baugh-Wooley multiplier is shown in figure 4.

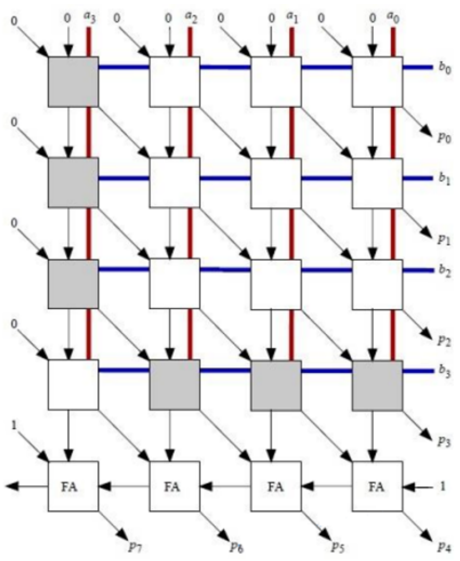

Figure 4 - 4x4 Baugh-Wooley multiplier

The white and grey blocks (considering also the FAs in the RCA's chain) shown in figure 4, that compose this architecture are shown in greater detail in figure 5.

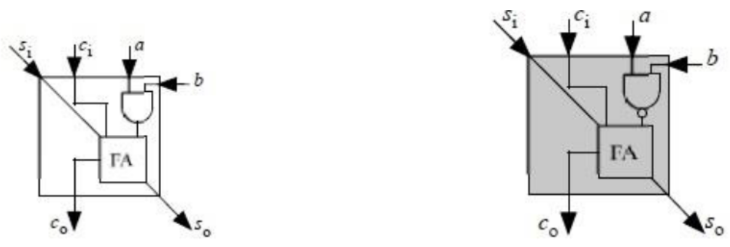

Figure 5 - Detail of white and grey blocks

It is possible to predict (table 7) how many blocks can be allocated depending on the number of bits N of the two inputs, inputs with the same width are assumed.

Block	Value
#Grey Blocks (n_{GB})	$2(N-1)$
#White Blocks (n_{WB})	$N^2 - n_{GB}$
#FA$_{RCA}$(n$_{FA}$)	N

Table 7

The area occupied by these blocks is computed by considering a FA and a AND2 gate (obtained by the sum of the areas of a NAND2 with a INV in cascade) for the white block, and FA with a NAND2 gate for the grey one (table 8).

Block	Value
$area_{GB}$	$Area_{NAND2} + Area_{FA}$
$area_{WB}$	$Area_{AND2} + Area_{FA}$

Table 8

The total area of the Baugh-Wooley is computed by summing the number of white blocks, of grey blocks and full adders, each of them multiplied by its respective area. The same considerations have been done to compute the static and the dynamic power of the structure.
By looking at the delay, the law that identifies the critical path, and relates it to the number of elements in the arithmetic device, can be expressed as follows:

$$T_{CP} = \tau_{INV} + \tau_{AND2} + N\tau_{FA,s} + N\tau_{FA,c}$$

In this way, the longest path inside this structure takes into account the delay necessary to cross just one inverter, one AND2 gate and all the FAs belonging to the white and grey blocks along the vertical direction (it is possible to identify N contributions of the IN to S path of each FA). The last term in the equation is related to the needed time to cross the final RCA.

2.4 Even-Odd multiplier

The Even-Odd multiplier belongs to the so called "tree multipliers": the key idea is to use the CSA so that, starting from the partial product multiplication, a tree of reduction is created, in order to compress the information and providing two outputs. By replicating this structure until it is no more possible to obtain other partial products, a final adder can be employed to compute the product.
In general the components are:

- Tree: made up of CSA;
- Final 2-input adder.

With respect to the other possible configurations, the Even-Odd multiplier presents a different structure for the partial product matrix. In this case, in fact, the even and the odd partial products are summed in parallel on two distinct branches of CSA: this allows to obtain a tree with half of the depth with respect to the other standard architectures.

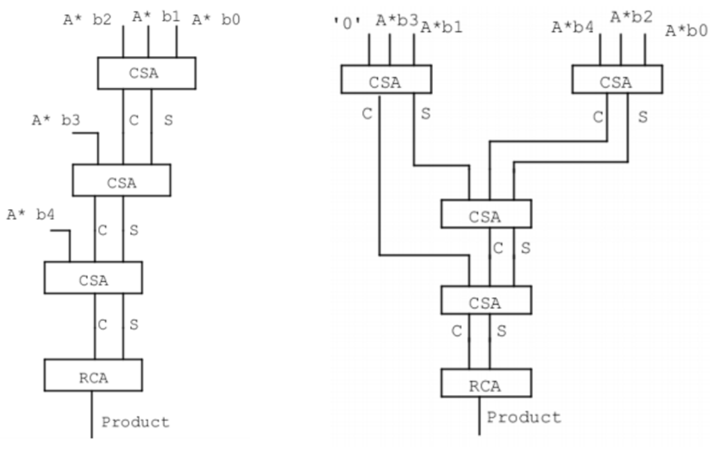

Figure 6 - Example of even-odd multiplier

The first part of the analysis is centered on the tree structure. The number of NAND2 gates inside a single CSA block, if N is the number of input bits of the two operands, is equal to

$$n_{NAND2,CSA} = 9N$$

The height of the tree, instead, can be computed as

$$h_{tree} = ceil(N)$$

In this way it is possible to calculate the number of CSA blocks:

$$n_{CSA} = 2h_{tree}–2$$

And so, starting from it, the number of NAND2 inside the tree can be computed as

$$n_{NAND2,tree} = n_{CSA}n_{NAND2,CSA}$$

The area related to this first part can be simply computed by multiplying the number of NAND2 gates inside the tree by the area of a single gate:

$$area_{tree} = n_{NAND2,tree}Area_{NAND2}$$

The same consideration can be exploited to compute the contributions related to the static and dynamic power. Regarding instead the delay, the critical path can be identified in the way that connects the inputs to the output passing through the entire tree. In the example shown in figure 7 it has been assumed to have a final RCA. The critical path can be computed as

$$\tau_{tree} = h_{tree}\tau_{FA,c}$$

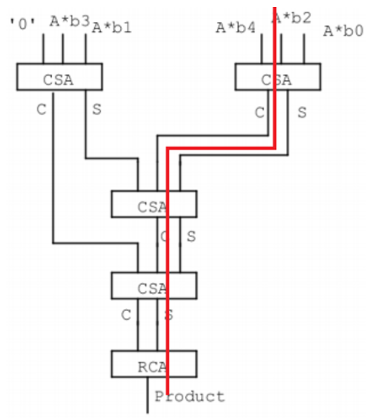

Figure 7 - Example of critical path in the even-odd multiplier

2.5 Modified Booth Multiplier

MBE is a Radix-4 approach, it halves the number of partial products with respect to the standard version of Booth Encoding, which is a Radix-2 approach. Normally it would be required to fully sign-extend the 2's complement partial product representations, but exploiting Roorda's approach it is possible to avoid such overhead, hence reducing considerably the number of compressors (full adders /half adders) in the adder plane.
The partial products p_j are selected based on three bits of operand b, namely b_{2j+1}, b_{2j} and b_{2j-1}. The encoding is shown in figure 8.

$b_{2j+1}b_{2j}b_{2j-1}$	p_j
000	0
001	a
010	a
011	2a
100	-2a
101	-a
110	-a
111	0

Figure 8 - Modified Booth Encoding.

Instead of directly generating all the possible partial products as 0, a, 2a, -a and -2a depending on multiplicand b triplet of bits, the expression describing partial products (which can be derived from direct inspection of figure 8) is more complex in MBE than in Radix-2 solutions, namely

$$p_j = (b_{2j+1} \oplus q_j) + b_{2j+1}$$

where

$$q_j = \begin{cases} 0 & \textbf{if } \overline{(b_{2j} \oplus b_{2j-1})}\,\overline{(b_{2j+1} \oplus b_{2j})} \\ a & \textbf{if } b_{2j} \oplus b_{2j-1} \\ 2a & \textbf{if } \overline{(b_{2j} \oplus b_{2j-1})}\,(b_{2j+1} \oplus b_{2j}) \end{cases}$$

The model was simplified w.r.t. theory, but the error in terms of employed resources/power is negligible, at least w.r.t. the lower-level modeling where only NAND2-INV gates were considered to be the building blocks of more complex structures.

3 Reduction tree

The main goal of a reduction tree is to reduce the sum of N operands to the sum of just two operands, that can be handled by a conventional (possibly fast) adder. Reduction trees are based on the carry-save approach, that is faster w.r.t. the carry-propagate approach. Blocks such as full adders and half adders are not seen anymore as basic addition blocks, but just as compressors.
In this work, as mentioned in section 1, it is possible to build multipliers properly choosing the partial products generation circuit, the reduction tree and the final adder. Regarding the reduction tree the user has the possibility to choose between Wallace and Dadda strategies. This choice is of course not allowed in case of special multipliers such as array multiplier (that is called ”Baugh-Wooley” if signed operands are considered) or even-odd multiplier.

3.1 Wallace

Wallace reduction tree is based on an ASAP philosophy. As opposed to Dadda reduction tree, the number of dots in each column is reduced at the earliest opportunity. Since the standard compressors are full adders and half adders, with a maximum compression ratio of 1.5 in the case of FAs, in each reduction step it's not possible to reduce the number of dots in each column with a factor higher than 1.5. A consequence of the massive resource use is the reduction of the final (two-operand) adder width, that ideally should be a positive feature in terms of critical path, but it's actually not true: in any case Dadda reduction tree is faster.

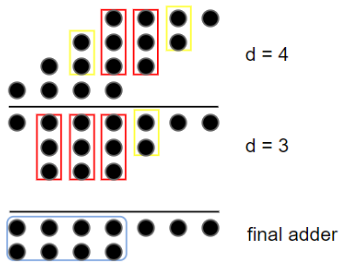

Figure 9 - 4x4 multiplication, Wallace reduction tree

In figure 9 an example of 4x4 multiplication exploiting Wallace strategy is shown. Yellow rectangles determine an HA, red rectangles determine a FA.
The approach used to determine the total number of compressors and the critical path of the reduction tree is similar to the one explained in section 4.2. The main difference is that instead of using the minimum amount of resources to reach a desired height in the following reduction step, now a parameter ”coverable_dots” is associated to each column, so that the proper amount of compressors is used. For instance if $coverable\_dots = 6$, if and only if the dots are actually available 2 FAs will be used. When the column weight is equal or greater than 2^{N+1}, the number of coverable dots will be reduced by one while going from one column to the following one (least significant to most significant).

3.2 Dadda

Dadda reduction tree is based on an ALAP philosophy. At each reduction stage it employs the minimum number of compressors so that the number of dots in each column would not exceed the one of the correspondent Wallace tree. Analytically, considering a target depth $d_0 = 2$ for the final adder, the respective depths at the previous reduction steps will be

$$d_{j+1} = floor(1.5d_j)$$

In this way the number of stages will be the same as in the Wallace tree, but with less resources employed. The final adder will be instead slightly wider w.r.t. Wallace strategy, but overall Dadda reduction tree is slightly faster (for all operand sizes) and requires fewer gates (for all but the smallest operand sizes). Dadda reduction tree is faster even though the final CPA width is larger w.r.t. Wallace reduction tree CPA, in which S bits of the final sum (S is the number of reduction stages) was already computed inside the tree. An example of a 4x4 multiplication, represented in dot notation, is shown in figure 10. Yellow rectangles represent the use of HAs, while the red ones represent the use of FAs.

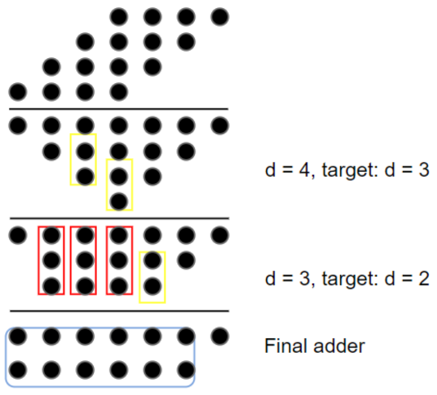

Figure 10 - 4x4 multiplication, Dadda reduction tree

In order to find the Dadda reduction tree critical path the methodology described in [4] is used. Each dot of the typical dot notation is replaced, inside a matrix, with a number that represents the delay of the bit in that particular position. An iterative algorithm reduces column by column the tree, stage after stage, where in each stage a desired height is specified into a pre-computed vector. Each column is analyzed sequentially, starting from the least significant and moving to the most significant one. The cycle (analysis of each column, in each reduction stage) is carried out until the matrix is reduced to just two rows. When a column must be reduced, a FA is used only if that specific column must compress more than one bit. Both the sum and the carry delays are propagated vertically, to the next stage: the sum in the same column, while the carry in the next (higher weight) column.

4 Final two-operand adder

In this section the possible user selectable two-operand adders are described. It is possible to choose a slow final adder such as the standard RCA (Ripple-Carry Adder), or a faster solution such as the CLA (Carry-Lookahead Adder) or a parallel-prefix approach adder (Ladner-Fischer, Brent-Kung, Kogge-Stone).

4.1 RCA

It is the simplest solution to add two N-bit numbers, since it is just a cascade of full adders. Its name derives from the fact that each carry bit "ripples" to the next full adder, hence it is a quite simple but slow solution. Note that the first (and only the first) full adder may be replaced by a half adder (under the assumption that $C_{in} = 0$).
The model of such an adder is quite simple, the only consideration that has been done is related to its width: in case of Wallace reduction tree the RCA will be characterized by a width reduced by the value S (number of reduction stages) w.r.t. Dadda reduction tree, since those bits are already computed inside the reduction tree.
The model is summarized in table 9, note that N changes whether in the previous step there was a Wallace or Dadda reduction tree.

Parameter	Value
$area_{RCA}$	$n_{NAND2,RCA} area_{NAND2}$
τ_{RCA}	$N \tau_{FA,c}$
Ps_{RCA}	$n_{NAND2,RCA} Ps_{NAND2}$
$Pdyn_{RCA}$	$n_{NAND2,RCA} Pdyn_{NAND2}$

Table 9

4.2 CLA

The CLA tree structure (figure 11) is made of "A" blocks and "B" blocks, whose operations are specified in the same figure.

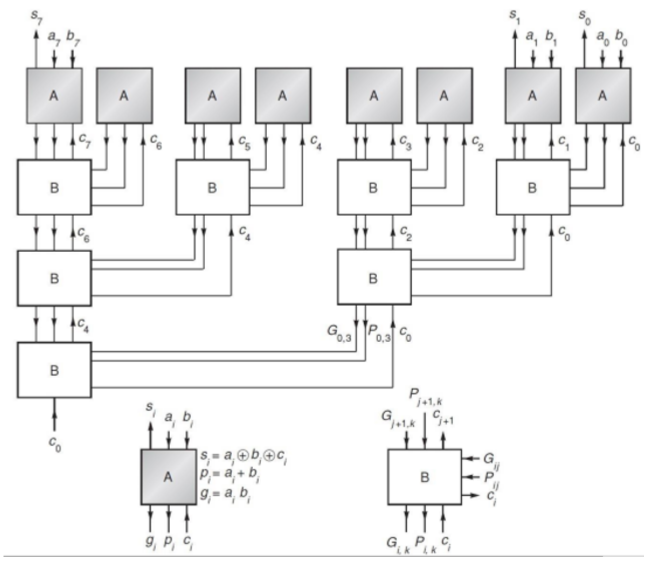

Figure 11 - CLA tree example with details on "A","B" blocks

It's possible to determine all the parameters by firstly considering the number of A blocks (N_A) equal to N, and of B blocks (N_B) equal to N-1. The parameters are summarized in table 10.

Parameter	Value
$n_{NAND2,A}$	$n_{NAND2,GEN_A} + n_{NAND2,PROP_A} + n_{NAND2,S_A}$
$n_{INV,A}$	$n_{INV,GEN_A} + n_{INV,PROP_A}$
$n_{NAND2,B}$	$n_{NAND2,GEN_B} + n_{NAND2,PROP_B} + n_{NAND2,C_B}$
$n_{INV,B}$	$n_{INV,GEN_B} + n_{INV,PROP_B} + n_{INV,C_A}$
$area_{CLA}$	$(n_{NAND2,A} + n_{NAND2,B})area_{NAND2} + (n_{INV,A} + n_{INV,B})area_{INV}$
Ps_{CLA}	$(n_{NAND2,A} + n_{NAND2,B})Ps_{NAND2} + (n_{INV,A} + n_{INV,B})Ps_{INV}$
$Pdyn_{CLA}$	$(n_{NAND2,A} + n_{NAND2,B})Pdyn_{NAND2} + (n_{INV,A} + n_{INV,B})Pdyn_{INV}$

Table 10

Regarding the delay parameters, the discussion is slightly more complex. For that reason it's necessary to analyze the paths associated to the A and B blocks. In fact each block is composed by different gates that have to be crossed. So there are various delays related to:

- Generate the result in the A block: $\tau_{generate,A}$, it is equal to the time needed to cross the A block from top to bottom ($\tau_{A,down}$);
- Propagate the result in the A block: $\tau_{propagate,A}$;
- Sum computation in A block: $\tau_{sum,A}$, it is equal to the time needed to cross the A block in the up direction ($\tau_{A,up}$);
- Generate the result in B block: $\tau_{generate,A}$, it is equal to the time needed to cross the B block from top to bottom direction ($\tau_{B,down}$);
- Propagate the result in B block: $\tau_{propagate,B}$;
- Carry computation: $\tau_{carry,B}$, it is equal to the time needed to cross the B block in the up direction ($\tau_{B,up}$);

By knowing these contributions it is possible to obtain the total amount of the delay necessary to cross the critical path as

$$\tau_{CLA} = \tau_{A,down} + (log2(2 * N - 2) - 1)\tau_{B,down} + (log2(2 * N - 2))\tau_{B,up} + \tau_{A,up}$$

4.3 Parallel-Prefix approach

Three types of parallel-prefix approach adders have been implemented: Ladner-Fischer, Brent-Kung and Kogge-Stone. The following considerations are valid for all the three adders, in particular:

- Generate/propagate signal computation blocks are composed by 2 AND2 and 1 OR2 gates, that correspond to 7 NAND2 gates. For simplicity these blocks will be called "black" blocks;
- Sum computation blocks are composed by 1 AND2 and 1 OR2 gates, that correspond to 5 NAND2 gates. The number of these blocks is $2N - 2$ (actually $2N - i$ in case of Wallace reduction tree). For simplicity these blacks will be called "grey" blocks;
- The critical path along the latter two blocks is equal to $4\tau_{NAND2}$;

- The number of NAND2 gates in the prefix (generate/propagate signals computation) stage is $num_{precomputation} = (2N-2)(6)$, while the number of NAND2 gates in the sum/carry-out computation network will be equal to $num_{result} = (2N-2)4+5$

What's in the middle of the parallel prefix graph of course depends on the specific implementation.

4.3.1 Ladner-Fischer adder

The number of NAND2 gates of the Ladner-Fischer network can be computed by multiplying by 7 (2 AND2 and 1 OR2 gates correspond to 7 NAND2 gates) the total number of group generate/group propagate signal computation blocks ("black" blocks) , and by 5 the number of "grey" blocks as follows

$$N_{NAND2,LF} = 7(\frac{2N-2}{2} log_2(2N-2)) + 5(2N-2)$$

The total number of NAND2 gates used for the final two-operand adder will include this quantity and the two contributions that are true for whatever parallel-prefix network:

$$N_{NAND2,total} = num_{precomputation} + N_{NAND2,LF} + num_{result}$$

With this value it's possible to trivially compute the area, static and dynamic power of the Ladner-Fischer adder.
Considering instead the critical path, it is possible to compute it simply by applying the following equation:

$$T_{cp} = log_2(2N-2)\tau_{"black"block} + \tau_{"grey"block} + 2\tau_{XOR2}$$

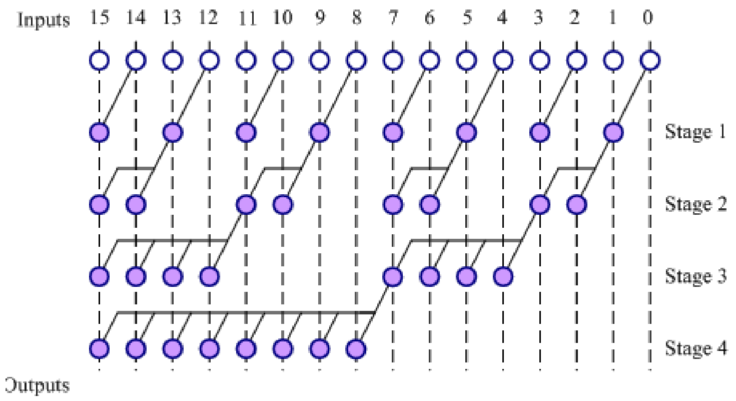

Figure 12 - 16-bit Ladner-Fischer adder

The Ladner-Fischer adder, shown in figure 12, is characterized by a very low depth (ideally fast), but also by high-fanout nodes that will translate to slower speed, especially for large width additions. The high-fanout problem is not visible in the results since it was not included into the simplified model.

4.3.2 Brent-Kung adder

The only difference w.r.t. Ladner-Fischer adder, or other adders based on the parallel-prefix approach, is related to the number of ”black” blocks and of course on the height of the adder. The number of ”black” blocks $num_{\text{"}black\text{"}blocks,BK}$ can be computed as follows

$$num_{\text{"}black\text{"}blocks,BK} = 2(2N-2) - 2 - log_2(2N-2)$$

hence one can compute the number of NAND2 gates into the Brent-Kung-specific portion of the adder as

$$N_{NAND2,BK} = 7num_{\text{"}black\text{"}blocks,BK} + 5(2N-2)$$

and finally the total number of NAND2 gates into the Brent-Kung adder will be

$$N_{NAND2,total} = num_{precomputation} + N_{NAND2,BK} + num_{result}$$

As in section 4.3.1 one can easily compute the area, static and dynamic power of the adder starting from the $N_{NAND2,total}$ quantity. The critical path can be computed as follows

$$T_{cp} = (2log_2(2N-2) - 2)\tau_{\text{"}black\text{"}block} + \tau_{\text{"}grey\text{"}block} + 2\tau_{XOR2}$$

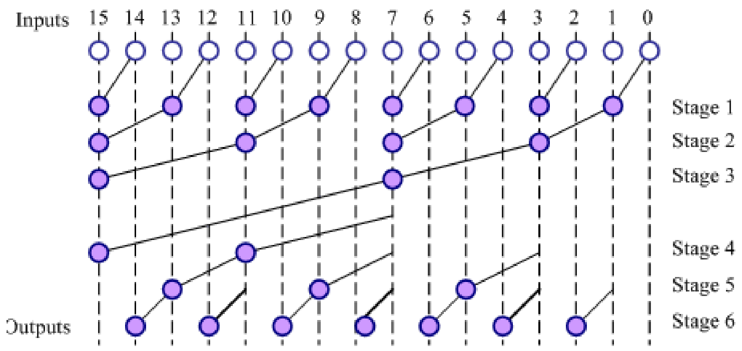

Figure 13 - 16-bit Brent-Kung adder

The Brent-Kung adder, shown in figure 13, solves the problem of high-fanout seen in the Ladner-Fischer adder, at the expense of a larger depth, hence it is (ideally) slower. It is also characterized by a lower complexity.

4.3.3 Kogge-Stone adder

One can repeat the same steps as in sections 4.3.1 and 4.3.2, the only parameters that change are the following:

$$num_{\text{"}black\text{"}blocks,KS} = (2N-2)log_2(2N-2) - (2N-2) + 1$$

$$T_{cp} = log_2(2N-2)\tau_{\text{"}black\text{"}block} + \tau_{\text{"}grey\text{"}block} + 2\tau_{XOR2}$$

With the first formula one can easily derive the total number of NAND2 gates of the Kogge-Stone adder, to then be able to compute the area, static and dynamic power. The second formula can be directly employed to compute the critical path and of course the maximum operating frequency (referring just to the adder).

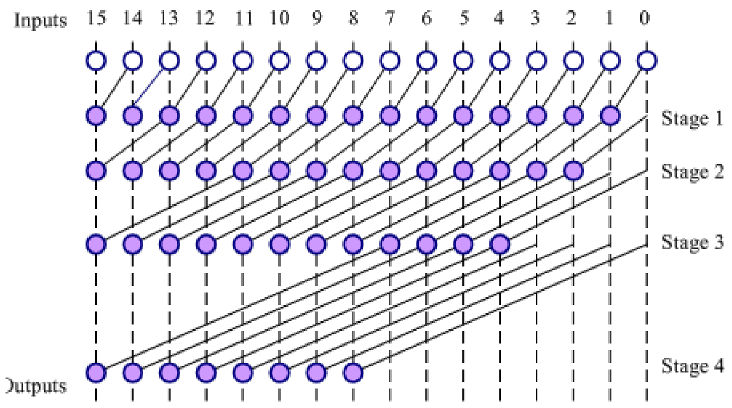

Figure 14 - 16-bit Kogge-Stone adder

The Kogge-Stone adder, shown in figure 14, solves the problem of high-fanout seen in the Ladner-Fischer adder and also guarantees the same height, hence ideally this architecture is the fastest, but also the most expensive in terms of area.

5 Results and possible improvements

In this section several use cases (possible user-defined heterogeneous multipliers) are investigated and compared. Finally, some possible improvements are discussed.

5.1 Example A: Baugh-Wooley multiplier

As an example a width of $N = 155$ was chosen, then Baugh-Wooley multiplier was selected by imposing $a = 1$. This multiplier is simply an array multiplier with the capability of working with signed 2's complement numbers. It is characterized by a special architecture, hence the user is not asked to specify the reduction tree and final two-operand adder types. The results are shown in figure xxxx.

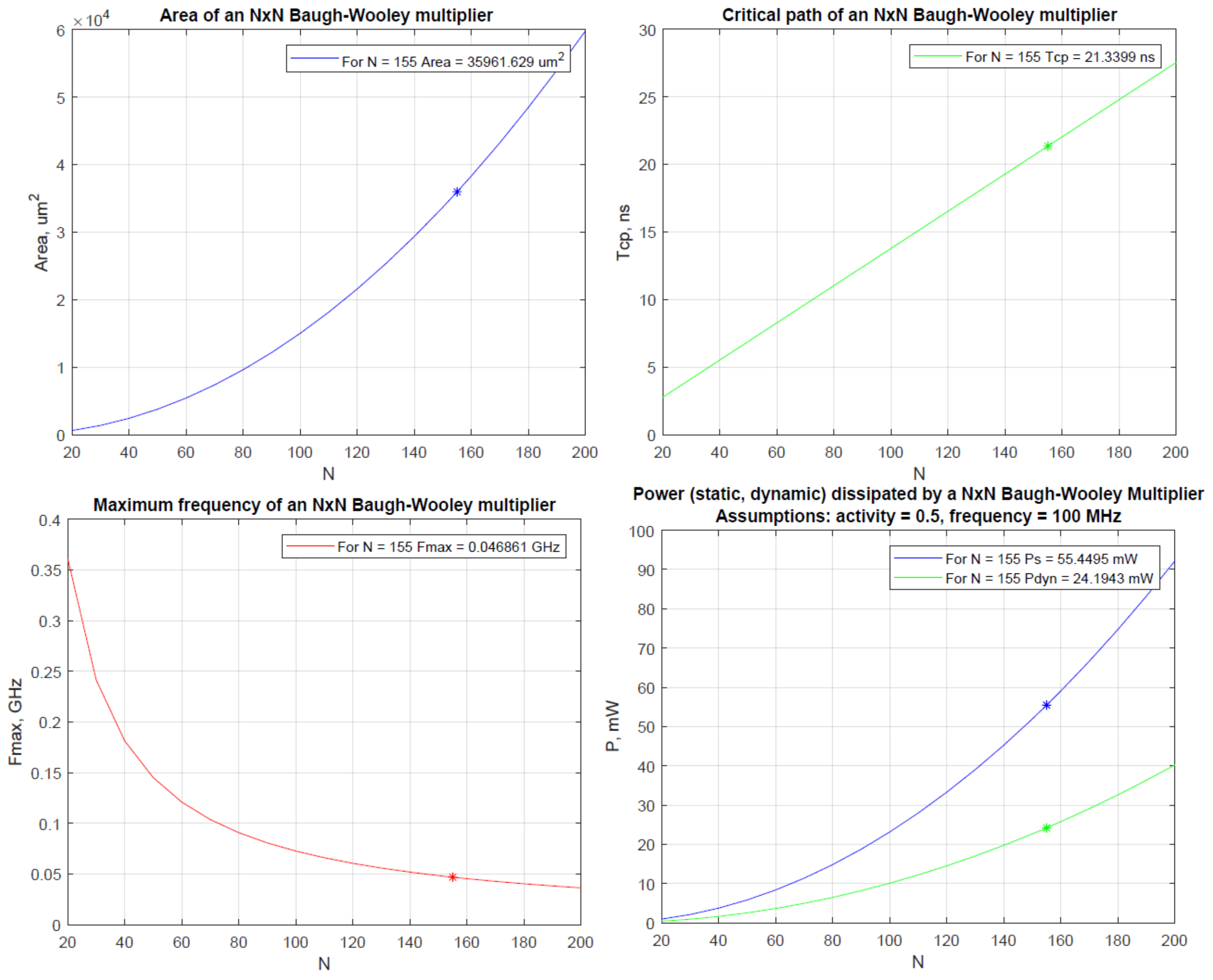

Figure xxxx - Baugh-Wooley multiplier. Values for multiplier width $N = 155$ are explicitly shown

The Baugh-Wooley multiplier is characterized, as expected, by a quite long critical path due to the carry-propagate philisophy. For a multiplier width of $N = 155$ the maximum operating frequency is lower than 47 MHz.

The dynamic power was estimated considering an activity of 0.5 and a reference operating frequency of 100 MHz, that is actually higher w.r.t. what the multiplier can actually achieve. A possible improvement to avoid this looseness is briefly discussed in section 6.5.

5.2 Example B: Dadda multiplier with RCA

Again $N = 155$ was chosen. The Dadda multiplier was selected by imposing the a parameter equal to 2, then the Ripple-Carry adder was selected by imposing the c parameter equal to 1. Of course in this case the user is not asked to insert the b parameter, since it is related to the reduction tree typology that is already implied by the a parameter. The results are shown in figure xxxx.

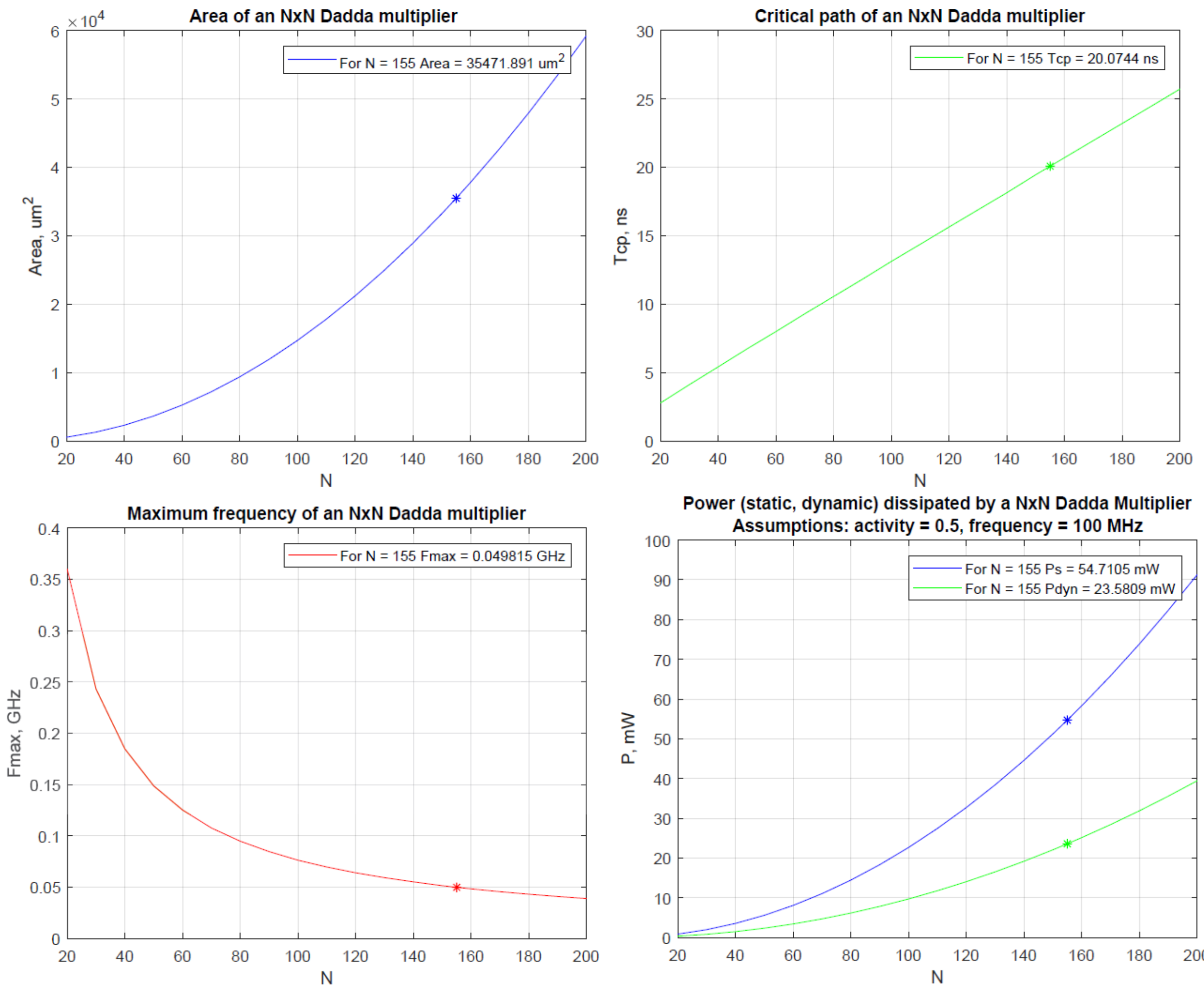

Figure xxxx - Dadda multiplier, RCA as final adder. Values for multiplier width $N = 155$ are explicitly shown

W.r.t. Baugh-Wooley multiplier (section 6.1) the Dadda multiplier with a Ripple-Carry adder as final two-operands adder shows very similar performance. The improvement in terms of area and static power is around 1.3%, maximum frequency improved of about 6.3%. The critical path still increases linearly with multiplier width N.

5.3 Example C: Dadda multiplier with Ladner-Fischer (parallel prefix) adder

Again $N = 155$ was chosen. Again the Dadda multiplier was selected by imposing the a parameter equal to 2, then the Ladner-Fischer adder (parallel prefix approach) was selected by imposing the c parameter equal to 2. The results are shown in figure xxxx.

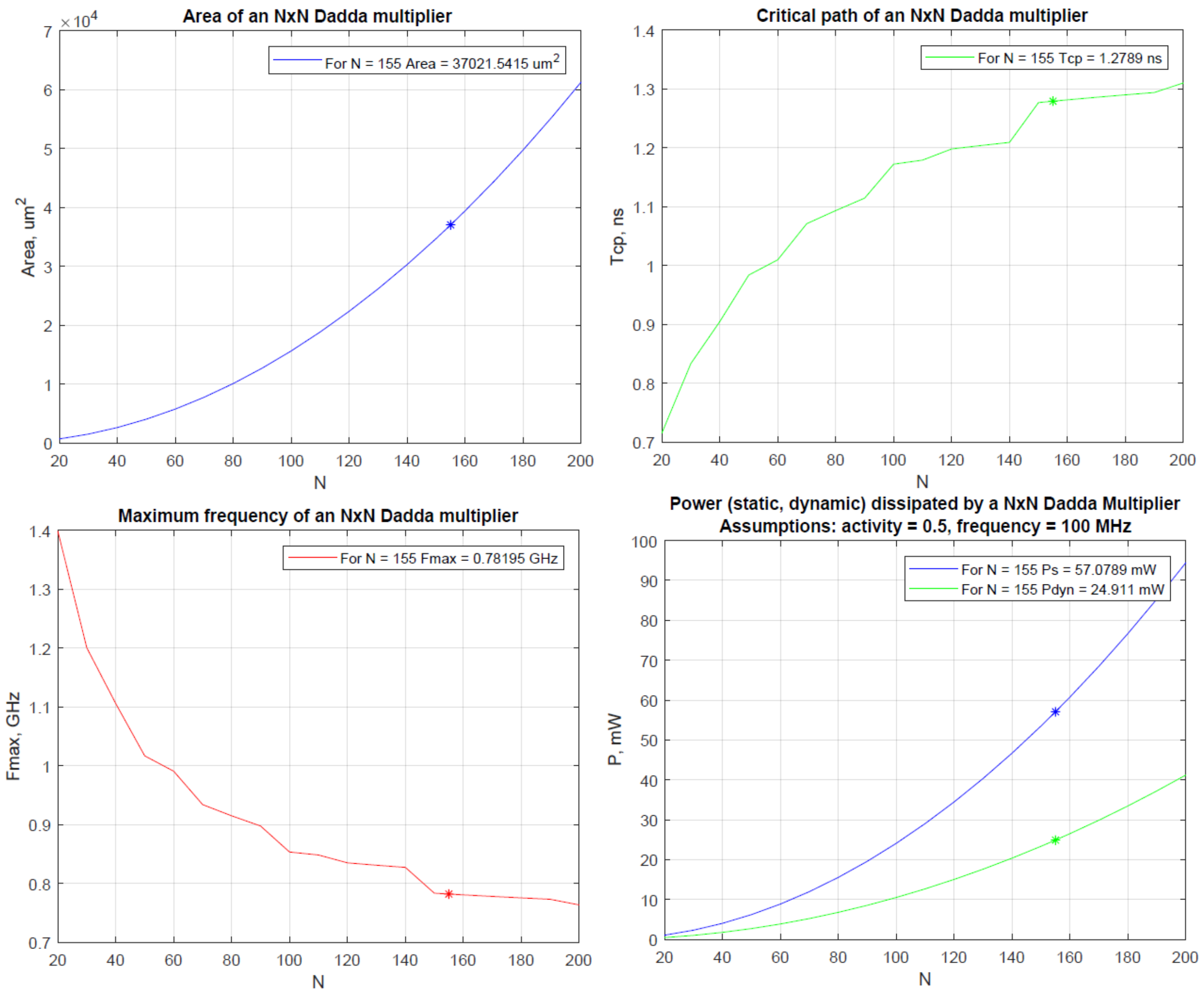

Figure xxxx - Dadda multiplier, Ladner-Fischer as final adder. Values for multiplier width $N = 155$ are explicitly shown

With a fast adder, in this case Ladner-Fischer network, since it is characterized by a logarithmic critical path dependency on multiplier width N, the performance improves dramatically w.r.t. example B (section 6.2), especially for high values of N.
W.r.t. the RCA case the maximum frequency improves massively, at the cost of a slightly higher complexity and of course higher static power. It is possible to increase the operating frequency (by default set to 100 MHz), hence the dynamic power would also increase dramatically.

5.4 Example D: MBE multiplier with Dadda reduction tree and CLA final adder

The last example better shows the potential of the script: it is possible to estimate the performance of unusual multipliers, such as MBE (Modified Booth Encoding) multiplier, with Dadda reduction tree and CLA (Carry Look-Ahead) final adder. Again, for the sake of comparison, $N = 155$ was chosen as multiplier parallelism. Then the specified parameters to "build" such a multiplier were $a = 4$, $b = 1$, $c = 5$. The results are shown in figure xxxx.

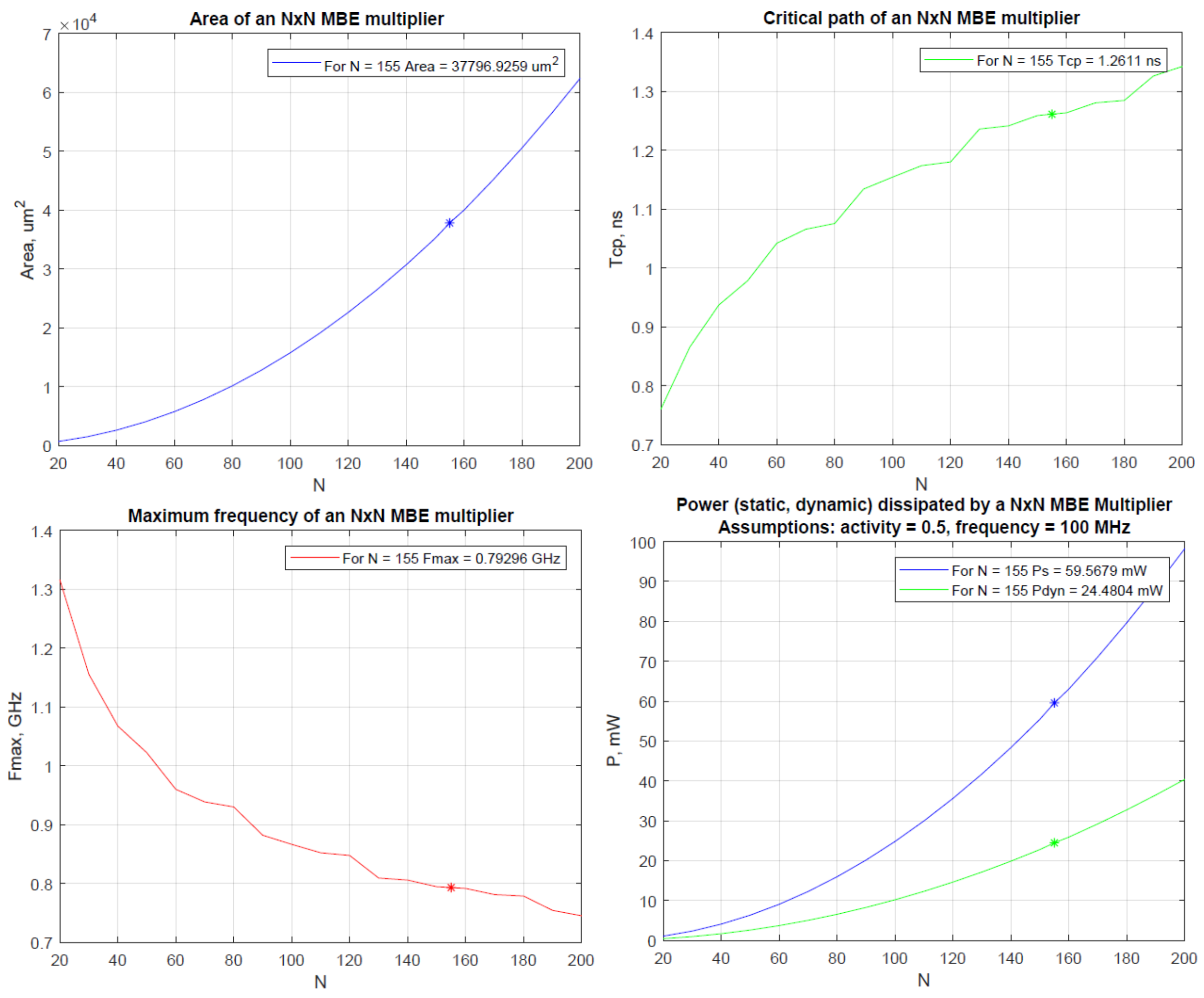

Figure xxxx - MBE multiplier, Dadda reduction tree and CLA as final adder. Values for multiplier width $N = 155$ are explicitly shown

With this multiplier one can see how it is possible to stretch even more the maximum frequency at the cost of a slightly higher complexity. With a Ladner-Fischer final adder it would be possible to improve even further the maximum frequency, of course paying in terms of area.

5.5 Possible improvements

One can possibly improve the script by refining the employed models, especially the Wallace one, or by adding new models to further enhance flexibility.
Another improvement would be choosing as operating frequency the maximum frequency instead of the reference one (100 MHz) for a better dynamic power estimation. This would be a bit tricky since the dynamic power is computed (like the other output parameters) three times, once for each module (partial product generation network, reduction tree and final adder), and of course each module has a different maximum operating frequency.

A Appendix

```
close all
clear all
clc

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%__________________ Multiplier performance estimator ______________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%             *Reference technology node: HP 2010
%     ,   ,   *Hypothesis: - only NAND2, INV gates
%    /////|                - Wn = 10*Lg
%   ///// |                - activity_factor = 0.5
%  |~~~|  |                - no logical effort theory
%  |===|  |    *The user specifies:
%  | m |  |          - "N", multiplier width (NxN)
%  | p |  |          - "a", multiplier type (related to pp generation)
%  | y | /             +--+-------------------------+
%  |===|/              |1 | Baugh-Wooley (array)    |
%  '---'               |2 | Dadda                   |
%                      |3 | Wallace                 |
%                      |4 | Modified Booth (MBE)    |
%                      |5 | Even-Odd                |
%                      +--+-------------------------+
%                    - "b", reduction tree type
%                      +--+-------------------------+
%                      |0 | None                    |
%                      |1 | Dadda                   |
%                      |2 | Wallace                 |
%                      +--+-------------------------+
%                    - "c", two-operator adder type
%                      +--+-------------------------+
%                      |0 | None                    |
%                      |1 | Ripple carry adder      |
%                      |2 | PPA: Ladner-Fischer     |
%                      |3 | PPA: Brent-Kung         |
%                      |4 | PPA: Kogge-Stone        |
%                      |5 | Carry Look-Ahead        |
%                      +--+-------------------------+
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%__________________ Input parameters _____________________________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% - The standard multiplier is Baugh-Wooley, so if a = 0 => b = 0, c = 0;
% - If Dadda or Wallace multiplier is chosen: - "b" must be equal to "a";
%                                             - "c" must belong to [1:4];
% - If MBE multiplier is chosen: - "b" must belong to [1:2];
%                                - "c" must belong to [1:4];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Multiplier inputs size
Mpy_width = input('Input data width:\n N = ');

if Mpy_width > 200
    disp("Too wide multiplier! Please reduce N");
    return
```

```
59  end
60  N = Mpy_width;
61
62
63  %% Multiplier type
64  a_mult = [ "Baugh-Wooley" ,"Dadda","Wallace","MBE", "Even-Odd"];
65  a = input(['Multiplier type:\n 1 => Baugh-Wooley \n 2 => Dadda'...
66              '\n 3 => Wallace \n 4 => MBE \n 5 => Even-Odd \n']);
67
68  if a < 2 || a > 5
69      if a ~= 1
70          disp("Wrong input, default multiplier selected (Baugh-Wooley)");
71      else
72          disp("You have selected Baugh-Wooley multiplier!");
73      end
74      a = 1;  % default multiplier : Baugh-Wooley
75      b = 0;  % to avoid an error in line 93
76      c = 0;  % to avoid an error in line 112
77  else
78      disp("You have selected "+ a_mult(a) +" multiplier!");
79  end
80
81
82  %% Reduction tree type
83  if a == 2 || a == 3
84      b = a-1;
85  elseif a == 5  %% even-odd multiplier
86      b = 0;
87  elseif a ~= 1
88      b = input('Reduction tree type:\n 1 => Dadda \n 2 => Wallace \n');
89  end
90
91
92  %% Two-operator adder
93  if (b == 0 || b > 2) && a ~= 1
94      if a == 5
95      c = input(['Final adder type:\n 1 => RCA \n 2 => PPA, '...
96       'Ladner-Fischer \n 3 => PPA, Brent-Kung\n 4 => PPA, '...
97       'Kogge-Stone \n 5 => Carry Look-Ahead \n']);
98      else
99      b = 1;  % default reduction tree : Dadda
100     disp("Wrong input, default reduction tree selected (Dadda)");
101      c = input(['Final adder type:\n 1 => RCA \n 2 => PPA, '...
102      'Ladner-Fischer \n 3 => PPA, Brent-Kung\n 4 => PPA,'...
103      'Kogge-Stone \n 5 => Carry Look-Ahead \n']);
104     end
105
106 elseif b == 1 || b == 2
107      c = input(['Final adder type:\n 1 => RCA \n 2 => PPA, '...
108      'Ladner-Fischer \n 3 => PPA, Brent-Kung\n 4 => PPA, '...
109      'Kogge-Stone \n 5 => Carry Look-Ahead \n']);
110 end
111
112 if (c == 0 || c > 5) && a ~= 1
113     c = 1;
114     disp("Wrong input, default two-input adder selected (RCA)");
115 end
116
117
118
```

```

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%_________ Technological parameters - ITRS 2009 edition ___________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Orders of magnitude
giga = 1e9;
mega = 1e6;
kilo = 1e3;
milli = 1e-3;
micro = 1e-6;
nano = 1e-9;
pico = 1e-12;
femto = 1e-15;

% Parameters
Lg = 27*nano;                       % effective channel length, [m]
Vdd = 0.97;                         % power supply voltage, [V]
Ion = 1200*micro/micro;             % saturation current with Rs!=0, [A/m]
MP1_half = 45*nano;                 % half metal pitch @ metal1, [m]
beta = 1.29;                        % electron/hole mobility ratio
Cg_ideal = 0.73*femto/micro;        % gate capacitance, [F/m]
Cg_fringing = 0.25*femto/micro;     % fringing capacitance, [F/m]
I_leak = 100*nano/micro;            % leakage current (S/D), [A/m]
Jg_max = 0.83*kilo/((10*milli)^2);    % w.c. gate current density, [A/m^2]
Covl = 0.2*Cg_ideal;                % overlap capacitance (S/G, D/G), [F/m]
Cj0 = 1*femto/(micro^2);            % junction cap. (no bias), [F/m^2]
L_SD = Lg;                          % source/drain region length, [m]
M = 1.5;                            % interc./avg gate output cap. overhead

% Assumptions
freq = 100*mega;             % reference clock frequency
activity = 0.5;              % gates switch every 2 clock cycles
Wn = 10*Lg;




%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%_______________ reference nMOS parameters computation ____________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Cg_total = Cg_ideal + Cg_fringing + Covl;   % total gate capacitance, [F/m]
tau = Cg_total*Vdd/Ion;                     % intrinsic delay, [s]
Lnmos = Lg + 2*L_SD;                        % total length, [m]
area_NMOS_min = Wn*Lnmos;                   % total area, [m^2]
Ig_max = Jg_max*Lg;                         % gate current, [A/m]




%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%________________ basic GATES parameters computation ______________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% reference (minimum) INVERTER
C_FO1 = Cg_total*(1 + beta);                % cap. seen from input, [F/m]
Ld = 4*MP1_half;                            % diffusion length, [m]
Cj_n = Cj0*Ld;                              % nMOS junction cap., [F/m]
Cj_INV = Cj_n*(1 + beta);                   % inverter junction cap., [F/m]
```

```
Cl_INV = C_FO1 + Cj_INV;                      % total cap., [F/m]
tau_INV = tau*Cl_INV/Cg_total;                % ref. inverter delay, [s]
area_INV = area_NMOS_min*(1 + beta);          % ref. inverter area, [m^2]
Is_INV = (1+beta)/2*(I_leak + Ig_max)*Wn;     % ref. inv static current, [A],
                                              % equiprobable inputs
Ps_INV = Is_INV*Vdd;                          % ref. inv static power, [W]
Pdyn_INV = Cl_INV*Wn*(Vdd^2)/2;               % ref. inv dynamic power, [W]

% NAND2 (average gate)
C_FO4 = 4*C_FO1;                              % avg load, no overhead, [F/m]
Cl_NAND2 = M*C_FO4;                           % with interc. overhead, [F/m]
tau_NAND2 = tau_INV*Cl_NAND2/Cl_INV;          % avg gate delay, [s]
area_NAND2 = 2*area_NMOS_min*(2 + beta);      % avg gate area, [m^2]
% avg gate static current, [A], assuming equiprobable inputs
Is_NAND2 = ((6 + 2*beta)*I_leak + (6 + 4*beta)*Ig_max)*Wn/4;
Ps_NAND2 = Is_NAND2*Vdd;                      % avg gate static power, [W]
Pdyn_NAND2 = Cl_NAND2*Wn*Vdd^2/2;             % avg gate dynamic power, [W]




%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%_________________ basic BLOCKS parameters computation _____________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% AND2 is obtained with a NAND2 gate in series with an INVERTER
tau_AND2 = tau_NAND2 + tau_INV;               % AND2 delay, [s]
area_AND2 = area_NAND2 + area_INV;            % AND2 area, [m^2]
Ps_AND2 = Ps_NAND2 + Ps_INV;                  % AND2 static power, [W]
Pdyn_AND2 = Pdyn_NAND2 + Pdyn_INV;            % AND2 dynamic power, [W]

% XOR2 can be obtained with 4 NAND2 gates
tau_XOR2 = 3*tau_NAND2;                       % XOR2 delay, [s]
area_XOR2 = 2*area_NAND2;                     % XOR2 area, [m^2]
Ps_XOR2 = 4*Ps_NAND2;                         % XOR2 static power, [W]
Pdyn_XOR2 = 4*Pdyn_NAND2;                     % XOR2 dynamic power, [W]

% HA is obtained with 4 NAND2 gates and one INVERTER
tau_HA_s = 3*tau_NAND2;                       % HA sum bit delay, [s]
tau_HA_c = tau_INV + tau_NAND2;               % HA carry bit delay, [s]
area_HA = area_INV + 4*area_NAND2;            % HA area, [m^2]
Ps_HA = Ps_INV + 4*Ps_NAND2;                  % HA static power, [W]
Pdyn_HA = Pdyn_INV + 4*Pdyn_NAND2;            % HA dynamic power, [W]

% FA is obtained with 9 NAND2 GATES
tau_FA_s = 6*tau_NAND2;                       % FA sum bit delay, [s]
tau_FA_c = 5*tau_NAND2;                       % FA carry bit delay, [s]
area_FA = 9*area_NAND2;                       % FA area, [m^2]
Ps_FA = 9*Ps_NAND2;                           % FA static power, [W]
Pdyn_FA = 9*Pdyn_NAND2;                       % FA dynamic power, [W]




%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%_________________ PARTIAL PRODUCT BLOCKS ESTIMATIONS ______________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The quantities Tcp, Area, Pstatic and Pdyn are updated at each of the
% following steps:  - Partial products network
%                   - Reduction tree
%                   - Final adder
% In this section partial product circuits are analyzed.
% In the case of a = 1 (reference multiplier, Baugh-Wooley) the OVERALL
% area / critical path / static and dynamic power estimations are performed
% altogether,since the regular structure allows rather simple computations.

flag = 0;
for j=1:20  % every parameter (critical path, area, static and dynamic
            % power) is estimated for 20 different values of N

    if Mpy_width > (j+1)*10
        N = (j+1)*10;
    elseif Mpy_width <= (j+1)*10 && flag == 0
        N = Mpy_width;
         flag = 1;
    else
        N = j*10;
    end

N_vector(j) = N;     % values of N used to compute all parameters


switch a

    case 1 %Baugh-Wooley multiplier
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%________________ BAUGH-WOOLEY multiplier estimations _____________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Basic Array multiplier can operate on UNSIGNED input operands, so it was
% chosen as reference multiplier the Baugh-Wooley architecture since it can
% operate on SIGNED operands. Partial products (PPs) are obtained doing the
% logical AND of a_i, b_j (a_i is a bit of operand a, b_j is a bit of
% operand b). PPs involving a_(N-1) and b_(N-1) are complemented according
% to the Baugh-Wooley theory, in order to avoid full sign-extensions.
% This architecture consists of 3 types of basic blocks:
%          - "grey blocks", that handle complemented PPs;
%          - "white blocks", that handle not complemented PPs;
%          - full adders for computing the N MSBs of the product.
% To compute parameters such as critical path and number of basic blocks
% general formulas were exploited.

        %"Grey blocks": FA + NAND2
        area_GB = area_NAND2 + area_FA;          % G.b. area, [m^2]
        n_GB = 2*(N-1);                          % G.b. number
        Ps_GB = Ps_NAND2 + Ps_FA;                % G.b. static power, [W]
        Pdyn_GB = Pdyn_NAND2 + Pdyn_FA;          % G.b. dynamic power, [W]

        %"White blocks": FA + AND2 (AND2 = NAND2 + INV)
        area_WB = area_AND2 + area_FA;           % W.b. area, [m^2]
        n_WB = N^2 - n_GB;                       % W.b. number
        Ps_WB = Ps_AND2 + Ps_FA;                 % W.b. static power, [W]
        Pdyn_WB = Pdyn_AND2 + Pdyn_FA;           % W.b. dynamic power, [W]

        n_FA = N;                                % number of FAs

        % critical path estimation, [s]
        Tcp(j) = tau_INV + tau_AND2 + N*tau_FA_s + N*tau_FA_c;
        % maximum frequency, [Hz]
        Fmax(j) = 1/Tcp(j);
        % area estimation, [m^2]
        Area(j) = n_GB*area_GB + n_WB*area_WB + n_FA*area_FA;
        % static power estimation, [W]
        Pstatic(j) = n_GB*Ps_GB + n_WB*Ps_WB + n_FA*Ps_FA;
        % dynamic power estimation, [W]
        Pdyn(j)=(n_GB*Pdyn_GB+n_WB*Pdyn_WB+ n_FA*Pdyn_FA)*freq*activity;

    case {2,3} % PPs network estimations for Dadda, Wallace multipliers
        % The following parameters will be updated in the reduction tree/
        % final adder parameters estimation sections.

        Tcp(j) = tau_AND2;                          %[s]
        Fmax(j) = 1/Tcp(j);                         %[Hz]
        Area(j) = (N^2)*area_AND2;                  %[m^2]
        Pstatic(j) = (N^2)*Ps_AND2;                 %[W]
        Pdyn(j) = (N^2)*Pdyn_AND2*freq*activity;    %[W]

    case 4 % PPs network estimation for Modified Booth multiplier.
        % A radix-4 multiplier reduces the number of rows (operands) having
        % to be reduced by the reduction tree. Booth's algorithm exploits
        % the mathematical property of being able to express a sequence of
        % additions as a subtraction: the latter can be executed faster.

        % Regarding partial products generation encoders and decoders are
        % used. - Enc: 2 XNOR2, 1 XOR2, 1 INV = 12 NAND2 + 3 INV
        %       - Dec: 2 XNOR2, 2 OR2, 1 INV =  10 NAND2 + 7 INV

        % Encoder
        area_ENC = 12*area_NAND2 + 3*area_INV;
        n_ENC = ceil(N/2);
        Ps_E = 12*Ps_NAND2 + 3*Ps_INV;
        Pd_E = 12*Pdyn_NAND2 + 3*Pdyn_INV;

        % Decoder
        area_DEC = 10*area_NAND2 + 7*area_INV;
        tau_DEC = 5*tau_NAND2 + 2*tau_INV;  % series of XNOR, OR, NAND
        n_DEC = n_ENC*(N+1);
        Ps_D = 10*Ps_NAND2 + 7*Ps_INV;
        Pd_D = 10*Pdyn_NAND2 + 7*Pdyn_INV;


        % critical path estimation, [s]
        Tcp(j) = tau_DEC + tau_INV;
        % maximum frequency, [Hz]
        Fmax(j) = 1/Tcp(j);
        % area estimation, [m^2]
        Area(j) = n_ENC*area_ENC + n_DEC*area_DEC + (n_ENC - 1)*area_NAND2;
        % static power estimation, [W]
        Pstatic(j) = n_ENC*Ps_E + n_DEC*Ps_D + (n_ENC - 1)*Ps_NAND2;
        % dynamic power estimation, [W]
        Pdyn(j)=(n_ENC*Pd_E+n_DEC*Pd_D+(n_ENC-1)*Pdyn_NAND2)*freq*activity;

    case 5 % PPs network estimation for Even-Odd multiplier and CSA tree

        n_nand_CSA = 9*N;               % Number of nand for each CSA block
        h_tree = ceil(N);               % Height of the tree
        n_CSA_blocks = 2*h_tree - 2;    % Total number of CSA blocks
        n_nand_tree = n_nand_CSA*n_CSA_blocks;      % Total number of NAND2


        Tcp(j) = h_tree*tau_FA_c;             % critical path, [s]
        Fmax(j) = 1/Tcp(j);                   % maximum frequency, [Hz]
        Area(j) = n_nand_tree*area_NAND2;     % total area , [m^2]
        Pstatic(j) = n_nand_tree*Ps_NAND2;    % static power, [W]
        Pdyn(j) = n_nand_tree*Pdyn_NAND2*freq*activity; % dyn. power, [W]

end %switch a




%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%--------------------- REDUCTION TREE ESTIMATIONS ----------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    switch b
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    case 1 %------------Dadda reduction tree--------------------------------
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        % stage height computed as Xi = floor(1.5*X(i-1))
        stage_h_vector = ([2 3 4 6 9 13 19 28 42 63 94 141 211 316 ...
                          474 711 1066 1599 2398 3597 5395 8093]);

        if a == 4                               % delay matrix, MBE case
            tau_mat = zeros(ceil(N/2), 2*N);
            tree_height = ceil(N/2);
            for i = 1:ceil(N/2)                 % initialization, MBE case
                for x = i:i+N-1
                    tau_mat(i,x) = 10e-20;      % low value, negligible
                                                % w.r.t. multiplier delay
                end
            end
        else
            tau_mat = zeros(N, 2*N);            % delay matrix, generic case
            tree_height = N;
            for i = 1:N
                for x = i:i+N-1
                    tau_mat(i,x) = 10e-20;
                end
            end
        end

        %tree_height = N;
        column_fa = zeros(1, 2*N);               % init FA counter
        column_ha = zeros(1, 2*N);               % init HA counter

        % find current stage
        i = 1;
        while(stage_h_vector(i) < tree_height)
             i = i + 1;
        end

        % init column heights
        cheight = zeros(1, 2*N);
        for k = 1:2*N
            cheight(k) = nnz( tau_mat(:,k) );
        end

        % sort delay_matrix in descending order for the first time
        tau_mat = sort(tau_mat, 'descend');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% cycle over all stages %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        for current_stage_idx = i:-1:2
            % define desired height
            desired_h = stage_h_vector(current_stage_idx-1);
            % init row pointer for each column of updated_delay_matrix
            urow_ptr = ones(1, 2*N);
            % init updated_delay_matrix for next stage
            utau_mat = zeros(desired_h, 2*N);
            % cycle over all columns

            for cidx = 1:2*N-1
                % delay_matrix is sorted in descending order thus row_ptr
                % points to the lower non-zero value in this column
                row_ptr = nnz( tau_mat(:,cidx) );
                % compute distance from current column h. to desired one
                cheight_diff = cheight(cidx) - desired_h;

                % iterate process until desired height is reached
                while (cheight_diff > 0)
                   % check if FA is needed and 3 inputs are available

                   if ( cheight_diff > 1 && row_ptr > 2)
                      % inc. FA count and dec. column height by 2
                      column_fa(cidx) = column_fa(cidx) + 1;
                      cheight(cidx) = cheight(cidx) - 2;

                      % extract "slowest" input among 3 "fastest" bits
                      row_ptr = row_ptr - 2;
                      in_delay = tau_mat(row_ptr, cidx);

                      % compute sum delay and store in this column
                      utau_mat(urow_ptr(cidx), cidx) = in_delay + tau_FA_s;
                      % compute carry delay and store it in next column
                      utau_mat(urow_ptr(cidx+1), cidx+1) = in_delay + ...
                                                           tau_FA_c;

                   else % HA needed
                      % inc. HA count and decrease this column height by 1
                      column_ha(cidx) = column_ha(cidx) + 1;
                      cheight(cidx) = cheight(cidx) - 1;

                      % extract "slowest" input delay among 2 fastest bits
                      row_ptr = row_ptr - 1;
                      in_delay = tau_mat(row_ptr, cidx);

                      % compute sum delay and store in this column
                      utau_mat(urow_ptr(cidx), cidx) = in_delay + tau_HA_s;
                      % compute carry delay and store it in next column
                      utau_mat(urow_ptr(cidx+1), cidx+1) = in_delay + ...
                                                           tau_HA_c;
                   end
                % row_ptr is moved to the next available input in the col.
                row_ptr = row_ptr - 1;
                % a value has been added to both this column and the next
                % one so updated_row_ptr must be updated
                urow_ptr(cidx) = urow_ptr(cidx) + 1;
                urow_ptr(cidx+1) = urow_ptr(cidx+1) + 1;

                % add carry bit in next column height count
                cheight(cidx+1) = cheight(cidx+1) +  1;
                cheight_diff = cheight(cidx) - desired_h;
                end

                % transfer unused delay values in this column from delay
                % matrix to updated delay matrix
                while ( row_ptr > 0 )
                    utau_mat(urow_ptr(cidx), cidx) =tau_mat(row_ptr, cidx);
                    urow_ptr(cidx) = urow_ptr(cidx) + 1;
                    row_ptr = row_ptr - 1;
                end

            end

            % update delay matrix (sorted in descending order)
            tau_mat = sort(utau_mat, 'descend');

        end

        % count total # FAs and # HAs in Dadda tree
        count_fa = sum(column_fa);
        count_ha = sum(column_ha);
        % identify slowest bit in the 2 final operands
        redtree_delay = max( max(tau_mat) );

        % critical path estimation, [s]
        Tcp(j) = Tcp(j) + redtree_delay;
        % maximum frequency, [Hz]
        Fmax(j) = 1/Tcp(j);
        % area estimation, [m^2]
        Area(j) = Area(j) + count_fa*area_FA + count_ha*area_HA;
        % static power estimation, [W]
        Pstatic(j) = Pstatic(j) + count_fa*Ps_FA + count_ha*Ps_HA;
        % dynamic power estimation, [W]
        Pdyn(j)= Pdyn(j) + (count_fa*Pdyn_FA + ...
                 count_ha*Pdyn_HA)*freq*activity;



    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    case 2 %____________Wallace reduction tree______________________________
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % stage height computed as Xi = floor(1.5*X(i-1))
        stage_h_vector = ([2 3 4 6 9 13 19 28 42 63 94 141 211 316 ...
                          474 711 1066 1599 2398 3597 5395 8093]);

        if a == 4                                  % delay matrix, MBE case
            tau_mat = zeros(ceil(N/2), 2*N);
            tree_height = ceil(N/2);
            for i = 1:ceil(N/2)                    % initialization, MBE case
                for x = i:i+N-1
                    tau_mat(i,x) = 10e-20;         % low value, negligible
                                                   % w.r.t. multiplier delay
                end
            end
        else
            tau_mat = zeros(N, 2*N);            % delay matrix, generic case
            tree_height = N;
            for i = 1:N
                for x = i:i+N-1
                    tau_mat(i,x) = 10e-20;
                end
            end
        end

        %tree_height = N;
        column_fa = zeros(1, 2*N);              % init FA counter
        column_ha = zeros(1, 2*N);              % init HA counter

        % find current stage
        i = 1;
        while(stage_h_vector(i) < tree_height)
             i = i + 1;
        end

        % init column heights
        cheight = zeros(1, 2*N);
        for k = 1:2*N
            cheight(k) = nnz( tau_mat(:,k) );
        end

        % sort delay_matrix in descending order for the first time
        tau_mat = sort(tau_mat, 'descend');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% cycle over all stages %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        for current_stage_idx = i:-1:2
            desired_h = stage_h_vector(current_stage_idx-1);
            % init row pointer for each column of updated_delay_matrix
            urow_ptr = ones(1, 2*N);
            % init updated_delay_matrix for next stage
            utau_mat = zeros(desired_h , 2*N);

            % to get maximum n. of employable FAs
            coverable_dots = zeros(1,2*N);
            carry = zeros (1, 2*N);
            % cycle over all columns
            for cidx = 1:2*N-1
                % delay_matrix is sorted in descending order thus row_ptr
                % points to the lower non-zero value in this column
                row_ptr = nnz( tau_mat(:,cidx) )+1;

                if(cidx < N+1)
                    coverable_dots(cidx) = cheight(cidx);
                else
                    coverable_dots(cidx) = cheight(cidx) +N+1 - cidx + ...
                                           carry(cidx);
                end


                while(coverable_dots(cidx)>1 && row_ptr > 1)
                   % check if FA is needed and 3 inputs are available
                    if (coverable_dots(cidx) >2 && row_ptr >2)
```

```
                        % inc. FA count and dec. column height by 2
                        column_fa(cidx) = column_fa(cidx) + 1;
                        coverable_dots(cidx) = coverable_dots(cidx) - 3;

                        % extract "slowest" input among 3 "fastest" bits
                        row_ptr = row_ptr - 2;
                        in_delay = tau_mat(row_ptr, cidx);

                        % compute sum delay and store in this column
                        utau_mat(urow_ptr(cidx), cidx) = in_delay + tau_FA_s;
                        % compute carry delay and store it in next column
                        utau_mat(urow_ptr(cidx+1), cidx+1) = in_delay + ...
                                                             tau_FA_c;
                        % add carry bit in next column height count
                        carry(cidx + 1) = carry(cidx +1) + 1;

                          % HA needed
                    elseif(coverable_dots(cidx)>1 && row_ptr >1)
                        % inc. HA count and decrease this column height by 1
                        column_ha(cidx) = column_ha(cidx) + 1;
                        coverable_dots(cidx) = coverable_dots(cidx) - 2;

                        % extract "slowest" input delay among 2 fastest bits
                        %row_ptr = row_ptr - 1;
                        in_delay = tau_mat(row_ptr, cidx);

                        % compute sum delay and store in this column
                        utau_mat(urow_ptr(cidx), cidx) = in_delay + tau_HA_s;
                        % compute carry delay and store it in next column
                        utau_mat(urow_ptr(cidx+1), cidx+1) = in_delay + ...
                                                             tau_HA_c;
                        % add carry bit in next column height count
                        carry(cidx + 1) = carry(cidx +1) + 1;
                    else
                         break;
                    end
                % row_ptr is moved to the next available input in the col.
                row_ptr = row_ptr - 1;
                % a value has been added to both this column and the next
                % one so updated_row_ptr must be updated
                urow_ptr(cidx) = urow_ptr(cidx) + 1;
                urow_ptr(cidx+1) = urow_ptr(cidx+1) + 1;

                end

                % transfer unused delay values in this column from delay
                % matrix to updated delay matrix
                while ( row_ptr > 0 )
                    utau_mat(urow_ptr(cidx), cidx) =tau_mat(row_ptr, cidx);
                    urow_ptr(cidx) = urow_ptr(cidx) + 1;
                    row_ptr = row_ptr - 1;
                end

            end
            % update delay matrix (sorted in descending order)
            tau_mat = sort(utau_mat, 'descend');

        end

        % count total # FAs and # HAs in Wallace tree
        count_fa = sum(column_fa);
        count_ha = sum(column_ha);
        count_fa_vector(1,j) = count_fa;        %% debug purposes
        count_fa_vector(2,j) = count_ha;
        count_fa_vector(3,j) = 2^(j+1);
        % identify slowest bit in the 2 final operands
        redtree_delay = max( max(tau_mat) );

        % critical path estimation, [s]
        Tcp(j) = Tcp(j) + redtree_delay;
        % maximum frequency, [Hz]
        Fmax(j) = 1/Tcp(j);
        % area estimation, [m^2]
        Area(j) = Area(j) + count_fa*area_FA + count_ha*area_HA;
        % static power estimation, [W]
        Pstatic(j) = Pstatic(j) + count_fa*Ps_FA + count_ha*Ps_HA;
        % dynamic power estimation, [W]
        Pdyn(j)= Pdyn(j) + (count_fa*Pdyn_FA + ...
                 count_ha*Pdyn_HA)*freq*activity;
    end




%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%____________________ FINAL ADDER ESTIMATIONS _____________________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Parallel prefix approach precomputations
if a == 5 || b == 1
    num_g_block = 2*N - 2;
    num_precomputation = (2*N-2)*(4+2); % comput. of blocks in prefix stage
    num_result = (2*N-2)*4+5;           % generation of sum bits, carry out
elseif b == 2
    num_g_block = 2*N - i;
    num_precomputation = (2*N-i)*(4+2);
    num_result = (2*N-i)*4+5;
end
b_block = 7;            % each b block has two AND and one OR gate (7 NAND)
g_block = 5;            % each g block has one AND and one OR gate (5 NAND)
tau_b_block = 4*tau_NAND2;              % critical path of black block
tau_g_block = 4*tau_NAND2;              % critical path of grey block

switch c


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    case 1 %____________RCA (Ripple-Carry Adder)____________________________
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        if b == 1 || a == 5                 % full-length RCA, Dadda/Even-Odd
            n_nand_RCA = 9*(2*N-3) + 5;     % 2N-3 FAs, 1 HA

            Tcp(j) = Tcp(j) + (2*N-4)*tau_FA_c + tau_FA_s; % crit.path, [s]
            Fmax(j) = 1/Tcp(j);                            % max.freq, [Hz]
            Area(j) = Area(j) + n_nand_RCA*area_NAND2;     % t.area, [m^2]
            Pstatic(j) = Pstatic(j) + n_nand_RCA*Ps_NAND2; % s.power, [W]
            Pdyn(j) = Pdyn(j) + n_nand_RCA*Pdyn_NAND2;     % dyn.power, [W]

        elseif b == 2
            n_nand_RCA = 9*(2*N-3) + 5 - 9*(i-2);       % reduced length RCA
            %(Wallace reduction tree), i-2 is the number of reduction steps

            Tcp(j) = Tcp(j) +(2*N-2-i)*tau_FA_c +tau_FA_s; % crit.path, [s]
            Fmax(j) = 1/Tcp(j);                            % max.freq, [Hz]
            Area(j) = Area(j) + n_nand_RCA*area_NAND2;     % t.area, [m^2]
            Pstatic(j) = Pstatic(j) + n_nand_RCA*Ps_NAND2; % s.power, [W]
            Pdyn(j) = Pdyn(j) + n_nand_RCA*Pdyn_NAND2*freq*activity; % [W]
        end



    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    case 2  %__________Parallel Prefix: Ladner-Fischer____________________
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        if b == 1 || a == 5
            % total number of NAND2 gates
            num_b_block = (2*N-2)/2 * log2(2*N-2);
            N_NAND_Ladner = b_block*num_b_block + g_block*num_g_block;
            % n. of NAND2: precomputation + prefix stage + final step
            N_NAND_total = num_precomputation + N_NAND_Ladner + num_result;


            Tcp(j) = Tcp(j) + (log2(2*N-2))*tau_b_block + ...
                     tau_g_block*tau_XOR2*2;               % crit.path,[s]
            Fmax(j) = 1/Tcp(j);                            % max.freq.,[Hz]
            Area(j) = Area(j)+ N_NAND_total*area_NAND2;    % tot.area,[m^2]
            Pstatic(j) = Pstatic(j)+N_NAND_total*Ps_NAND2; % p.static, [W]
            Pdyn(j) = Pdyn(j) + N_NAND_total*Pdyn_NAND2*freq*activity; %[W]

        elseif b == 2
            % total number of NAND2 gates
            num_b_block = (2*N-i)/2 * log2(2*N-i);
            N_NAND_Ladner = b_block*num_b_block + g_block*num_g_block;
            % n. of NAND2: precomputation + prefix stage + final step
            N_NAND_total = num_precomputation + N_NAND_Ladner + num_result;


            Tcp(j) = Tcp(j) + (log2(2*N-i))*tau_b_block + ...
                     tau_g_block*tau_XOR2*2;               % crit.path,[s]
            Fmax(j) = 1/Tcp(j);                            % max.freq.,[Hz]
            Area(j) = Area(j)+ N_NAND_total*area_NAND2;    % tot.area,[m^2]
            Pstatic(j) = Pstatic(j)+N_NAND_total*Ps_NAND2; % p.static, [W]
            Pdyn(j) = Pdyn(j) + N_NAND_total*Pdyn_NAND2*freq*activity; %[W]

        end



    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    case 3  %__________Parallel Prefix: Brent-Kung_______________________
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        if b == 1 || a == 5
            % total number of NAND2 gates
            num_b_block = 2*(2*N-2)-2  -log2(2*N-2);
            N_NAND_BK = b_block*num_b_block + g_block*num_g_block;
            % n. of NAND2: precomputation + prefix stage + final step
            N_NAND_total = num_precomputation + N_NAND_BK + num_result;

```

```

          Tcp(j) = Tcp(j) + (2*log2(2*N-2)-2)*tau_b_block + ...
                   tau_g_block*tau_XOR2*2;                  % crit.path,[s]
          Fmax(j) = 1/Tcp(j);                               % max.freq.,[Hz]
          Area(j) = Area(j)+ N_NAND_total*area_NAND2;       % tot.area,[m^2]
          Pstatic(j) = Pstatic(j)+N_NAND_total*Ps_NAND2; % p.static,  [W]
          Pdyn(j) = Pdyn(j) + N_NAND_total*Pdyn_NAND2*freq*activity; %[W]

      elseif b == 2
          % total number of NAND2 gates
          num_b_block = 2*(2*N-i)-2 * -log2(2*N-i);
          N_NAND_BK = b_block*num_b_block + g_block*num_g_block;
          % n. of NAND2: precomputation + prefix stage + final step
          N_NAND_total = num_precomputation + N_NAND_BK + num_result;


          Tcp(j) = Tcp(j) + (2*log2(2*N-i)-2)*tau_b_block + ...
                   tau_g_block*tau_XOR2*2;                  % crit.path,[s]
          Fmax(j) = 1/Tcp(j);                               % max.freq.,[Hz]
          Area(j) = Area(j)+ N_NAND_total*area_NAND2;       % tot.area,[m^2]
          Pstatic(j) = Pstatic(j)+N_NAND_total*Ps_NAND2; % p.static,  [W]
          Pdyn(j) = Pdyn(j) + N_NAND_total*Pdyn_NAND2*freq*activity; %[W]
      end



  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  case 4  %__________Parallel Prefix: Kogge-Stone______________________
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      if b == 1 || a == 5
          % total number of NAND2 gates
          num_b_block = (2*N-2)*log2(2*N-2) - (2*N-2) + 1;
          N_NAND_KS = b_block*num_b_block + g_block*num_g_block;
          % n. of NAND2: precomputation + prefix stage + final step
          N_NAND_total = num_precomputation + N_NAND_KS + num_result;


          Tcp(j) = Tcp(j) + (log2(2*N-2))*tau_b_block + ...
                   tau_g_block*tau_XOR2*2;                  % crit.path,[s]
          Fmax(j) = 1/Tcp(j);                               % max.freq.,[Hz]
          Area(j) = Area(j)+ N_NAND_total*area_NAND2;       % tot.area,[m^2]
          Pstatic(j) = Pstatic(j)+N_NAND_total*Ps_NAND2; % p.static,  [W]
          Pdyn(j) = Pdyn(j) + N_NAND_total*Pdyn_NAND2*freq*activity; %[W]

      elseif b == 2
          % total number of NAND2 gates
          num_b_block = (2*N-i)*log2(2*N-i) - (2*N-i) + 1;
          N_NAND_KS = b_block*num_b_block + g_block*num_g_block;
          % n. of NAND2: precomputation + prefix stage + final step
          N_NAND_total = num_precomputation + N_NAND_KS + num_result;


          Tcp(j) = Tcp(j) + (log2(2*N-i))*tau_b_block + ...
                   tau_g_block*tau_XOR2*2;                  % crit.path,[s]
          Fmax(j) = 1/Tcp(j);                               % max.freq.,[Hz]
          Area(j) = Area(j)+ N_NAND_total*area_NAND2;       % tot.area,[m^2]
          Pstatic(j) = Pstatic(j)+N_NAND_total*Ps_NAND2; % p.static,  [W]
          Pdyn(j) = Pdyn(j) + N_NAND_total*Pdyn_NAND2*freq*activity; %[W]
      end

```

```


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
case 5 %____________CLA (Carry Look-Ahead)_____________________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % A-type blocks
    n_nand_gen_A = 1;
    n_inv_gen_A = 1;
    n_nand_prop_A = 1;
    n_inv_prop_A = 2;
    n_nand_sum_A = 8;
    n_nand_A = n_nand_gen_A + n_nand_prop_A + n_nand_sum_A;
    n_inv_A = n_inv_gen_A + n_inv_prop_A;

    %Delays related to the Block A
    tau_generate_A = n_nand_gen_A*tau_NAND2 + n_inv_gen_A*tau_INV;
    tau_propagate_A = n_nand_prop_A*tau_NAND2 + tau_INV;
    tau_sum = (n_nand_sum_A-2)*tau_NAND2;
    tau_A_up = tau_sum;
    tau_A_down = tau_generate_A;


    % B-type blocks
    n_nand_gen_B = 2;
    n_inv_gen_B = 1;
    n_nand_prop_B = 1;
    n_inv_prop_B = 1;
    n_nand_carry_B = 2;
    n_inv_carry_B = 1;
    n_nand_B = n_nand_gen_B + n_nand_prop_B + n_nand_carry_B;
    n_inv_B = n_inv_gen_B + n_inv_prop_B + n_inv_carry_B;

    %Delays related to the Block B
    tau_generate_B = n_nand_gen_B*tau_NAND2;
    tau_propagate_B = n_nand_prop_A*tau_NAND2 + n_inv_prop_A*tau_INV;
    tau_carry = n_nand_carry_B*tau_NAND2;
    tau_B_up = tau_carry;
    tau_B_down = tau_generate_B;

    if b == 1 || a == 5
        n_A = 2*N-2;                    % number of A-type blocks
        n_B = 2*N-3;                    % number of B-type blocks

        % critical path, [s]
        Tcp(j) = Tcp(j)+ tau_A_down + (log2(2*N-2) - 1)*tau_B_down + ...
                 (log2(2*N-2))*tau_B_up + tau_A_up;

        % maximum frequency, [Hz]
        Fmax(j) = 1/Tcp(j);

        % total area, [s]
        Area(j) = Area(j) + (n_A*n_nand_A + n_B*n_nand_B)*area_NAND2 ...
                 + (n_A*n_inv_A+n_B*n_inv_B)*area_INV;

        % static power, [W]
        Pstatic(j)= Pstatic(j)+ (n_A*n_nand_A+n_B*n_nand_B)*Ps_NAND2 ...
                 + (n_A*n_inv_A+n_B*n_inv_B)*Ps_INV;

        % dynamic power, [W]
```

```
            Pdyn(j) = Pdyn(j) + ((n_A*n_nand_A + n_B*n_nand_B)*Pdyn_NAND2...
                    + (n_A*n_inv_A + n_B*n_inv_B)*Pdyn_INV)*freq*activity;

        elseif b == 2       % reduced length final adder operators (Wallace)
            n_A = 2*N - i;                          % number of A-type blocks
            n_B = 2*N -1 - i;                       % number of B-type blocks

            % critical path, [s]
            Tcp(j) = Tcp(j)+ tau_A_down + (log2(2*N-2) - 1)*tau_B_down + ...
                     (log2(2*N-2))*tau_B_up + tau_A_up;

            % maximum frequency, [Hz]
            Fmax(j) = 1/Tcp(j);

            % total area, [s]
            Area(j) = Area(j) + (n_A*n_nand_A + n_B*n_nand_B)*area_NAND2 ...
                     + (n_A*n_inv_A+n_B*n_inv_B)*area_INV;

            % static power, [W]
            Pstatic(j)= Pstatic(j)+ (n_A*n_nand_A+n_B*n_nand_B)*Ps_NAND2 ...
                     + (n_A*n_inv_A+n_B*n_inv_B)*Ps_INV;

            % dynamic power, [W]
            Pdyn(j) = Pdyn(j) + ((n_A*n_nand_A + n_B*n_nand_B)*Pdyn_NAND2...
                    + (n_A*n_inv_A + n_B*n_inv_B)*Pdyn_INV)*freq*activity;

        end

end

end % for cycle end



%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%_________________________PLOT SECTION_________________________________
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% scale for plot
Area_micro2 = Area./micro^2;
Tcp_nano = Tcp./nano;
Fmax_giga = Fmax./giga;
Pstatic_milli = Pstatic./milli;
Pdyn_milli = Pdyn./milli;

% get index of user-defined N
index = 0;
for i=1:20
    index = index + 1;
    if N_vector(i) == Mpy_width
        break;
    end
end

user_value_area = Area_micro2(index);
user_value_tcp = Tcp_nano(index);
user_value_fmax = Fmax_giga(index);
user_value_pstatic = Pstatic_milli(index);
user_value_pdyn = Pdyn_milli(index);
```

```matlab
figure
plot(N_vector, Area_micro2,'b-', Mpy_width, user_value_area ,'b*'); hold on
grid on
legend("For N = "+ Mpy_width +" Area = "+ Area_micro2(index) + " um^2")
title("Area of an NxN "+  a_mult(a) + " multiplier ")
xlabel('N')
ylabel('Area, um^2')


figure
% subplot(1,2,1)
plot(N_vector, Tcp_nano, 'g-',Mpy_width, user_value_tcp , 'g*'); hold on
grid on
legend("For N = "+ Mpy_width +" Tcp = "+ Tcp_nano(index) + " ns")
title("Critical path of an NxN "+  a_mult(a) + " multiplier")
xlabel('N')
ylabel('Tcp, ns')


figure
% subplot(1,2,2)
plot(N_vector, Fmax_giga, 'r-',Mpy_width, user_value_fmax , 'r*'); hold on
grid on
legend("For N = "+ Mpy_width +" Fmax = "+ Fmax_giga(index) + " GHz")
title("Maximum frequency of an NxN "+  a_mult(a) + " multiplier")
xlabel('N')
ylabel('Fmax, GHz')


figure
plot(N_vector,Pstatic_milli,'b-', N_vector, Pdyn_milli,'g-', ...
 Mpy_width,user_value_pstatic,'b*',Mpy_width,user_value_pdyn,'g*'); hold on
grid on
legend("For N = "+ Mpy_width +" Ps = "+ Pstatic_milli(index) + " mW", ...
        "For N = "+ Mpy_width +" Pdyn = "+ Pdyn_milli(index) + " mW")
title({"Power (static, dynamic) dissipated by a NxN "+ a_mult(a)...
      + " Multiplier"; "Assumptions: activity = 0.5, frequency = 100 MHz"})
xlabel('N')
ylabel('P, mW')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```