
# POLITECNICO DI TORINO

**Corso Di Laurea Magistrale in Ingegneria Elettronica**

---

**INTEGRATED SYSTEM TECHNOLOGY**

# Project Report
# Group 12


| Students | Id. number |
|---|---|
| Andrea Massa | 253271 |
| Giorgio Perrone | 252969 |
| Alberto Venzo | 249840 |
| Daniele Volpintesta | 243764 |


---

**ACADEMIC YEAR 2018 - 2019**

# Contents


**1 Pentium IV adder** **2**
- 1.1 Introduction . . . 2
- 1.2 Library logic gates . . . 2
- 1.3 Important assumptions in the design . . . 3
- 1.4 Pentium IV adder design . . . 3
  - 1.4.1 Carry-select sum generator design . . . 4
  - 1.4.2 Carry-merge sparse tree . . . 13
  - 1.4.3 Pentium IV adder . . . 20
- 1.5 Matlab implementation . . . 21
- 1.6 Results . . . 25

**2 Pipelined Wallace tree** **28**
- 2.1 Introduction . . . 28
- 2.2 Theoretical analysis . . . 28
  - 2.2.1 Wallace tree design . . . 28
  - 2.2.2 Occupied Area . . . 29
  - 2.2.3 Power consumption . . . 30
  - 2.2.4 Timing analysis . . . 30
- 2.3 MATLAB implementation . . . 31
  - 2.3.1 Function . . . 31
  - 2.3.2 Script . . . 33
  - 2.3.3 Automatic Generation of FidoCadJ drawing . . . 36
  - 2.3.4 Automatic Generation of VHDL code . . . 37
- 2.4 Results . . . 40
  - 2.4.1 Area . . . 42
  - 2.4.2 Delay . . . 43
  - 2.4.3 Static power . . . 44
  - 2.4.4 Dynamic power . . . 45
  - 2.4.5 Different pipeline percentages comparison . . . 46
  - 2.4.6 VHDL . . . 48
- 2.5 Conclusions . . . 49

**3 A Dynamic CMOS Survey** **50**
- 3.1 General aspects . . . 50
- 3.2 Domino design techniques . . . 52
- 3.3 Dual Rail Domino Logic . . . 52
- 3.4 Technology scaling comparison . . . 53
- 3.5 Final Considerations . . . 53

# Chapter 1

# Pentium IV adder

## 1.1 Introduction

The aim of this project is to create a model of Pentium IV adder in order to estimate its area, delay and power comsumption. For this purpose, a Matlab script has been implemented to derive and plot results. Simulations for 8, 12, 16, 20, 24, 28 and 32 bit cases are reported in this work.
In this project it has been supposed that any elementary logic gate necessary are available as external components. Each logic gate is assumed to be already characterized in terms of area, delay, input capacitances and leakage currents.
The approach for the design is structural: each block in the pentium IV is characterized like a black-box with its area, delay and power consumption. In this way the internal structure of any block can be changed without modifying anything else and then the Matlab script can be run with the previous settings. This could be very useful to make future improvements for more complex simulations and structural designs.

## 1.2 Library logic gates

The logic gates used in the Pentium IV adder are NAND2, NOR2, INV, XOR2 and the transmission gate (called for brevity "tgate"). In the table 1.1 are listed the gates with their parameters.

| GATE TYPE | AREA | INPUT CAPACINTANCES | DELAY | AVERAGE LEAKAGE CURRENT |
|---|---|---|---|---|
| NAND2 | $A_{nand2}$ | $C_{in_{nand2}}$ | $\tau_{int_{nand2}} + \alpha_{nand2} \cdot C_L$ | $I_{leak_{nand2}}$ |
| NOR2 | $A_{nor2}$ | $C_{in_{nor2}}$ | $\tau_{int_{nor2}} + \alpha_{nor2} \cdot C_L$ | $I_{leak_{nor2}}$ |
| INV | $A_{inv}$ | $C_{in_{inv}}$ | $\tau_{int_{inv}} + \alpha_{inv} \cdot C_L$ | $I_{leak_{inv}}$ |
| XOR2 | $A_{xor2}$ | $C_{in_{xor2}}$ | $\tau_{int_{xor2}} + \alpha_{xor2} \cdot C_L$ | $I_{leak_{xor2}}$ |
| TGATE | $A_{tgate}$ | $C_{in1_{tgate}}, C_{in2_{tgate}}, C_{in3_{tgate}}$ | $\tau_{int_{tgate}} + \alpha_{tgate} \cdot C_L$ | $I_{leak_{tgate}}$ |

Table 1.1: library logic gates

A brief explanation of the parameters is given before the adder analysis. For instance, the NAND2 gate can be considered.
$A_{nand2}$ is the area of the NAND2.
$I_{leak_{nand2}}$ is the average leakage current of the NAND2. The leakage current of a logic gate depends on the values of its inputs, therefore the average leakage current is a function of the input probabilities. For sake of semplicity, the average leakage current is computed by considering the probabilities of the inputs equal.
For the input capacitances and the delay of the NAND2 figure 1.1 can be considered.

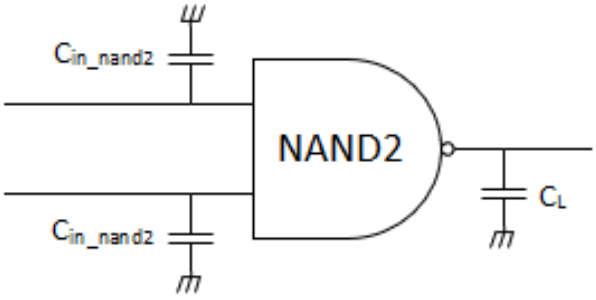



Figure 1.1: NAND2 gate

$C_{in_{nand2}}$ is the input capacitance in each input of the NAND2 gate.
The total delay of the NAND2 is $\tau_{int_{nand2}} + \alpha_{nand2} \cdot C_L$ where $\tau_{int_{nand2}}$ is the intrinsic delay of the NAND2 (the delay without load) while $\alpha_{nand2} \cdot C_L$ is the contribution that depends on the load $C_L$. The coefficient $\alpha_{nand2}$ depends on the sizes of the transistors inside the gate. The larger the transistors are, the smaller the delay is ($\alpha_{nand2}$ is smaller).
In the table, TGATE is the transmission gate. We will use the TGATE to make multiplexers. The TGATE is the only one in the table that has input capacitances depending on which input is considered (differently respect to a normal CMOS gate).
It is assumed that all the parameters in table 1.1 are given externally. In other words, the design is a function of the parameters reported in the table.
Also the supply voltage will be a input parameter because it depends on the technology adopted.

## 1.3 Important assumptions in the design

To evaluate the dynamic power in the design the switching activity $E_{sw}$ for each node of the circuit has been assumed equal to one. This is a very pessimistic case that it will not happen in reality. Evaluate the switching activity in each node is a very complex work. To do it the statistic of the inputs in our pentium IV should be known. Assuming $E_{sw} = 1$ the dynamic power is therefore overestimated. For instance, the real value could be also one order of magnitude smaller. This means that the value of the dynamic power can be considered as an upper bound.
The working frequency that it has been used for the dynamic power is the inverse of the critical path of the Pentium IV adder (obviously the clock to output delay of a flip-flop, the setup time and the skew should be considered but this is also a simple estimation for power).
The leakage current in table 1.1 is in the case that the input probabilities for each gate are the same as already said. Obviously this is not true in reality, therefore the value obtained for the static power will be not exactly. However the order of magnitude is expected reasonably correct. This simplification is done because it is too complex to obtain the real statistic of the inputs of each gate: several simulations should be performed in order to obtain the switching activity of each node.
To compute the delay of a path, all the delays of all gates in that particular path are summed. This means that, in table 1.1, the 50% delay for each gate has been considered.

## 1.4 Pentium IV adder design

The 32-bit Pentium IV adder is constituted by two blocks as reported in figure 1.2. The carry-merge sparse tree computes the carry bit in position 4, 8, 12, 16, 20, 24, 28 and 32. The carry-select sum generator computes the sum bit.

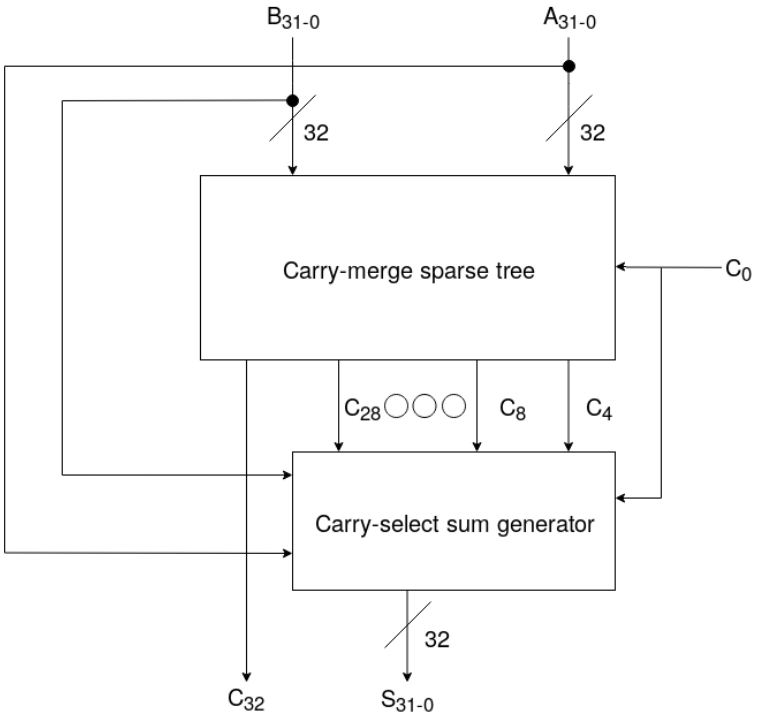



Figure 1.2: Pentium IV adder

In the following sections these two blocks will be detailed. A bottom-up approach has been used to complete this design.

### 1.4.1 Carry-select sum generator design

In this subsection the carry-select sum generator is designed.

#### Full adder

In Figure 1.3 it is shown the symbol of the full adder while in Figure 1.4 its internal circuit is shown.

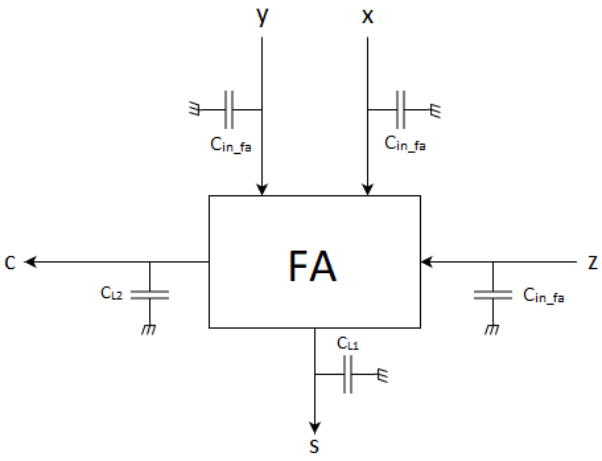



Figure 1.3: Symbol of full adder

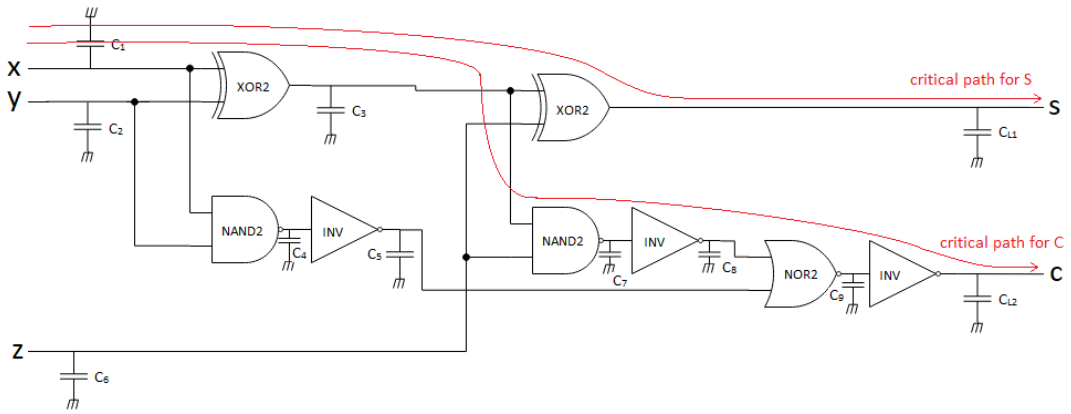



Figure 1.4: Internal full adder circuit

The full adder area is given by:

$$A_{FA} = 2A_{xor2} + 2A_{nand2} + A_{nor2} + 3A_{inv} \tag{1.1}$$

The leakage current is:

$$I_{leak_{FA}} = 2I_{leak_{xor2}} + 2I_{leak_{nand2}} + I_{leak_{nor2}} + 3I_{leak_{inv}} \tag{1.2}$$

The static power is:

$$P_{stat_{FA}} = V_{DD} \cdot I_{leak_{FA}} \tag{1.3}$$

For the dynamic power, as said, it has been assumed that the switching activity $E_{sw}$ of all the nodes in the circuit is equal to one. Therefore it is possible to write:

$$P_{dyn_{FA}} = V_{DD}^2 f_{CLK} E_{sw}(C_3 + C_4 + C_5 + C_7 + C_8 + C_9) \tag{1.4}$$

NOTE: for the dynamic power only the internal nodes are considered, which is that the input and output ones are not taken into account. The contribution of the input and output nodes will be considered when the FA-block will be used to create others blocks.
Looking at figure 1.4 it is possible to see that:

$$C_1 = C_2 = C_3 = C_6 = C_{in_{xor2}} + C_{in_{nand2}} \tag{1.5}$$

$$C_4 = C_7 = C_9 = C_{in_{inv}} \tag{1.6}$$

$$C_5 = C_8 = C_{in_{nor2}} \tag{1.7}$$

Thus:

$$P_{dyn_{FA}} = V_{DD}^2 f_{CLK} E_{sw}(C_{in_{xor2}} + C_{in_{nand2}} + 3C_{in_{inv}} + 2C_{in_{nor2}}) \tag{1.8}$$

The intrinsic delay (the delay without load) of the output $S$ is:

$$\tau_{int_{FA_S}} = \tau_{int_{xor2}} + \alpha_{xor2}C_3 + \tau_{int_{xor2}} = 2\tau_{int_{xor2}} + \alpha_{xor2}(C_{in_{xor2}} + C_{in_{nand2}}) \tag{1.9}$$

The $\alpha$ parameter is:

$$\alpha_{FA_S} = \alpha_{xor2} \tag{1.10}$$

Thus the total delay of $S$ (considering the load) is $\tau_{int_{FA_S}} + \alpha_{FA_S}C_{L1}$.

Analogously for the output $C$ it is obtained:

$$\tau_{int_{FA_C}} = \tau_{int_{xor2}} + \alpha_{xor2}C_3 + \tau_{int_{nand2}} + \alpha_{nand2}C_7 + \tau_{int_{inv}} + \alpha_{inv}C_8 + \tau_{int_{nor2}} + \alpha_{nor2}C_9 + \tau_{int_{inv}} \tag{1.11}$$

$$\alpha_{FA_C} = \alpha_{inv} \tag{1.12}$$

The delay of $C$ considering the load will be $\tau_{int_{FA_C}} + \alpha_{FA_C}C_{L2}$.

The input capacitances of the full adder are:

$$C_{in_x} = C_{in_y} = C_{in_z} = C_{in_{FA}} = C_{in_{xor2}} + C_{in_{nand2}} \tag{1.13}$$

### 4-bit ripple carry adder

In Figure 1.5 it is shown the symbol of the 4-bit ripple carry adder while in Figure 1.6 its internal circuit is shown.

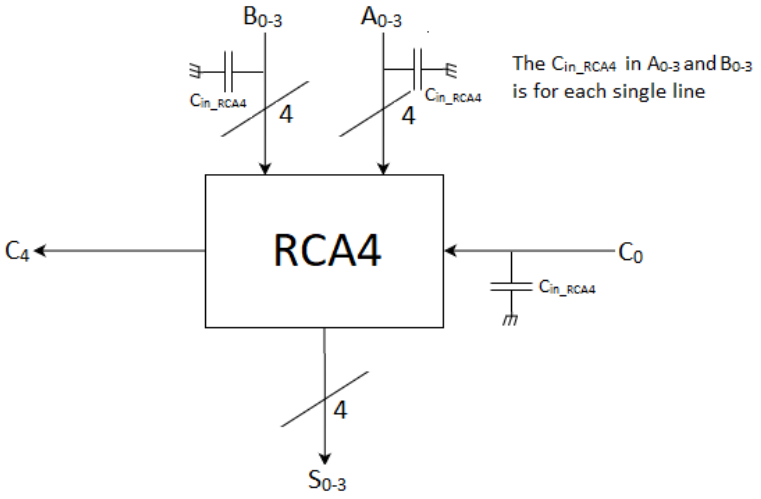



Figure 1.5: symbol of 4-bit ripple carry adder

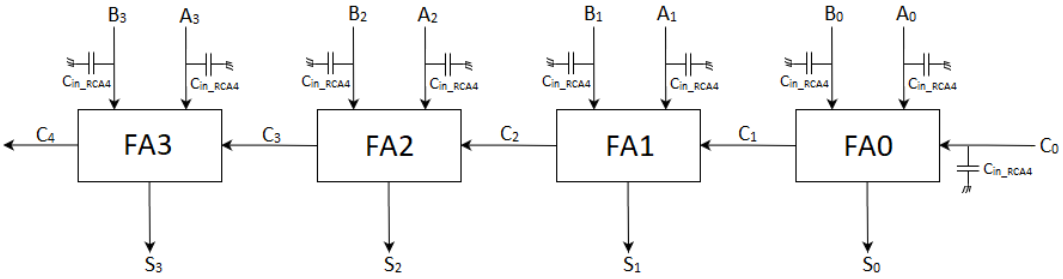



Figure 1.6: internal circuit of 4-bit ripple carry adder

The area of 4-bit ripple carry adder is given by:

$$A_{RCA4} = 4A_{FA} \tag{1.14}$$

The leakage current is:

$$I_{leak_{RCA4}} = 4I_{leak_{FA}} \tag{1.15}$$

The static power is:

$$P_{stat_{RCA4}} = V_{DD} \cdot I_{leak_{RCA4}} \tag{1.16}$$

Just as in the case of the full adder, in order to compute the dynamic power only the internal nodes have been considered:

$$P_{dyn_{RCA4}} = V_{DD}^2 f_{CLK} E_{sw}(3C_{in_{FA}}) + 4P_{dyn_{FA}} \tag{1.17}$$

NOTE: the term $4P_{dyn_{FA}}$ is for the internal nodes of the FAs.

To compute the delay, it can be observed that the delay to generate the 4 sum bits ($S_0, S_1, S_2$ and $S_3$) is the main concern. This is why the output carry ($C_4$) will be generated by the tree. Thus the delay of RCA4 to be considered is the delay to generate $S_3$. The intrinsic delay (that without load) can be written as:

$$\tau_{int_{RCA4}} = 3\tau_{int_{FA_C}} + \alpha_{FA_C}(3C_{in_{FA}}) + \tau_{int_{FA_S}} \tag{1.18}$$

For the alpha coefficient it is derived that:

$$\alpha_{RCA4} = \alpha_{FA_S} \tag{1.19}$$

The input capacitances for the carry input ($C_0$) and for each line of $A_{0-3}$ and $B_{0-3}$ is the same and equal to:

$$C_{in_{RCA4}} = C_{in_{FA}} \tag{1.20}$$

**Transmission gate**

The transmission gate has been used to make multiplexers. Using transmission gates to implement MUXs, instead of normal CMOS gates, is in fact a good solution in terms of area, delay and power consuption.
In Figure 1.7 it is shown the symbol of the transmission gate while in figure 1.8 its internal circuit.

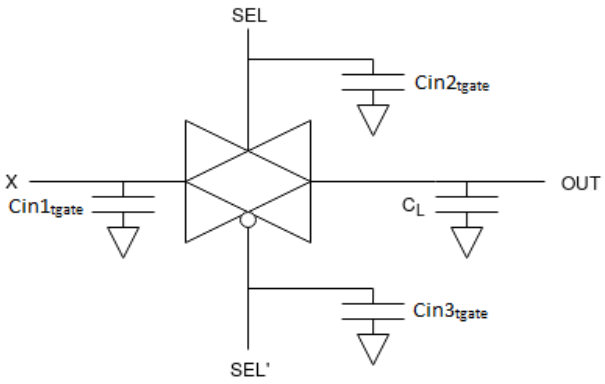



Figure 1.7: Symbol of the transmission gate

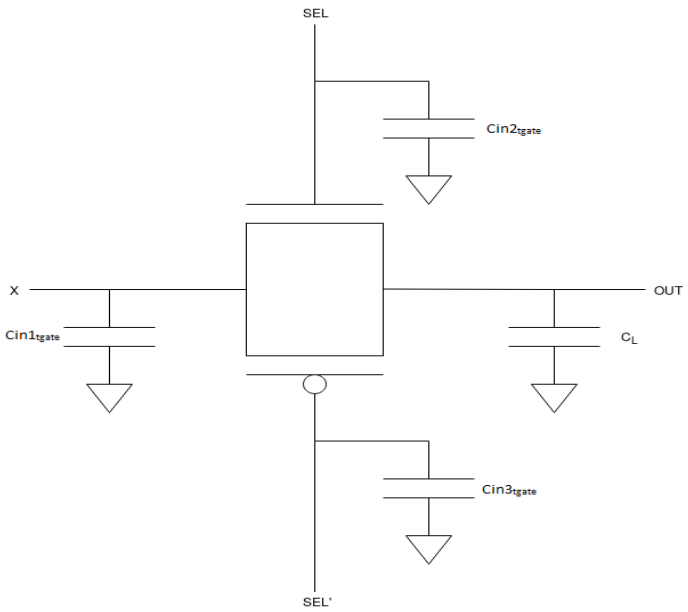



Figure 1.8: Internal circuit of the transmission gate

From Figure 1.8 it can be seen that the input capacitances of the transmission gate $C_{in1_{tgate}}$, $C_{in2_{tgate}}$ and $C_{in3_{tgate}}$ can be different. As said previously their values are given externally.
Also the area $A_{tgate}$ and the leakage current $I_{leak_{tgate}}$ are given.
The delay of the transmission gate is the time required for the signal at the input $X$ to reach the

output node ($OUT$) when the two transistors become ON. This delay is equal to $\tau_{int_{tgate}} + \alpha_{tgate} \cdot C_L$ where $\tau_{int_{tgate}}$ and $\alpha_{tgate}$ are both given.

### Multiplexer 2:1

From the transmission gate described before, a 2:1 way multiplexer (mux) can be designed and characterized. Figure 1.9 reports the designed mux.

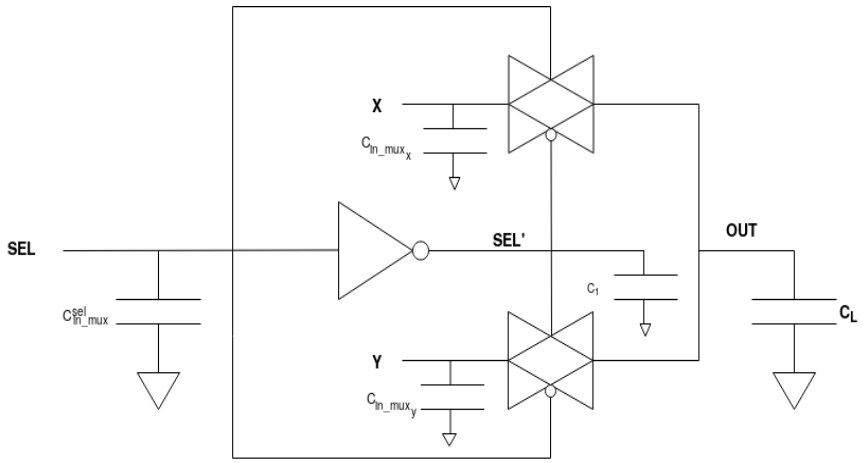



Figure 1.9: 2:1 way multiplexer implementation with transmission gates

The black-box model used for the multiplexer is the one in figure 1.10.

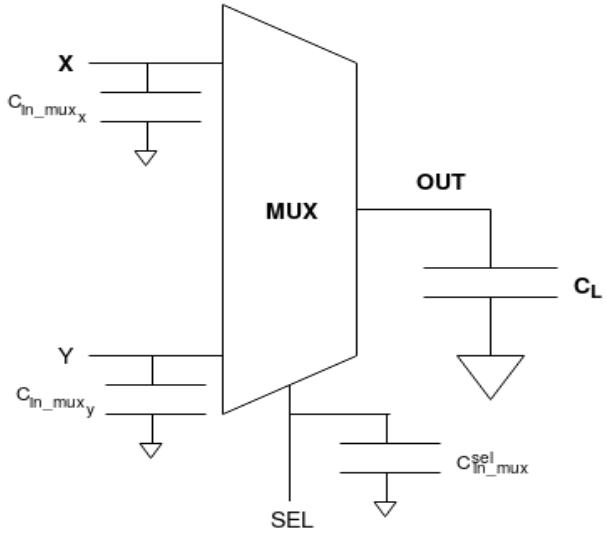



Figure 1.10: Multiplexer 2:1 symbol

The multiplexer area is given by:

$$A_{mux} = A_{inv} + 2A_{tgate} \tag{1.21}$$

The leakage current is given by:

$$I_{leak_{mux}} = I_{leak_{inv}} + 2I_{leak_{tgate}} \tag{1.22}$$

The static power is

$$P_{stat_{mux}} = V_{DD} I_{leak_{mux}} \tag{1.23}$$

Looking at the figure 1.9, the dynamic power can be derived:

$$P_{dyn_{mux}} = V_{DD}^2 f_{CLK} E_{sw} C_1 \tag{1.24}$$

Since $C_1 = C_{in2_{tgate}} + C_{in3_{tgate}}$ it is possible to obtain:

$$P_{dyn_{mux}} = V_{DD}^2 f_{CLK} E_{sw} (C_{in2_{tgate}} + C_{in3_{tgate}}) \tag{1.25}$$

The intrinsic delay (the delay without load) for the mux is:

$$\tau_{int_{mux}} = \tau_{int_{inv}} + \alpha_{inv} C_1 + \tau_{int_{tgate}} \tag{1.26}$$

Replacing $C_1$

$$\tau_{int_{mux}} = \tau_{int_{inv}} + \alpha_{inv} (C_{in2_{tgate}} + C_{in3_{tgate}}) + \tau_{int_{tgate}} \tag{1.27}$$

The $\alpha$ coefficient is:

$$\alpha_{mux} = \alpha_{tgate} \tag{1.28}$$

Thus the overall delay is $\tau_{int_{mux}} + \alpha_{mux} C_L$. The input capacitance seen from the inputs X or Y is given by:

$$C_{in_{mux}}^{input} = C_{in_{mux_x}} = C_{in_{mux_y}} = C_{in1_{tgate}} \tag{1.29}$$

while the input capacitance seen from the select input is:

$$C_{in_{mux}}^{sel} = C_{in_{inv}} + C_{in2_{tgate}} + C_{in3_{tgate}} \tag{1.30}$$

**Multiplexer 2:1 for M-length signal vectors**

In the Pentium IV adder design is useful a compact rappresentation of a selector which takes two vectors with a length equal to M and takes one of them to the output. Figure 1.11 shows the implementation of such mux (called for brevity M-mux), while figure 1.12 reports the black-box symbol.

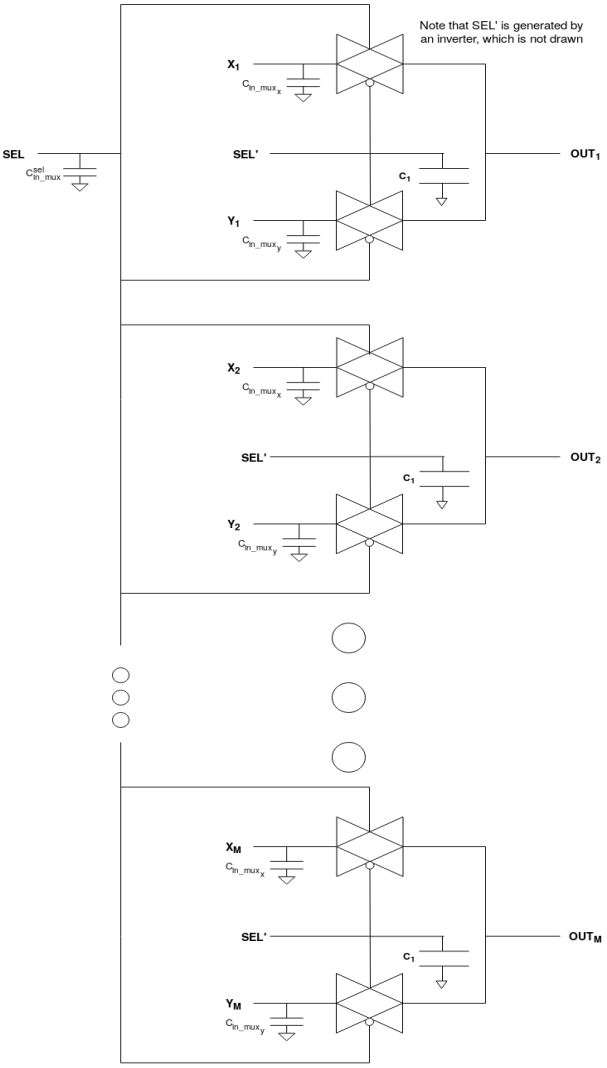



Figure 1.11: Mux 2:1 with M-bit inputs

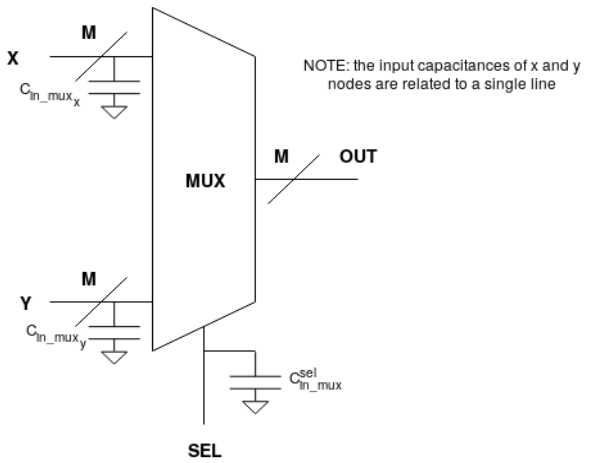



Figure 1.12: Symbol of mux 2:1 with M-bit inputs

Since in figure 1.11 the SEL' signal is generated by an inverter which is not drawn, the area of

the M-mux is:

$$A_{M-mux} = A_{inv} + 2MA_{tgate} \tag{1.31}$$

The leakage current is given by:

$$I_{leak_{M-mux}} = I_{leak_{inv}} + 2MI_{leak_{tgate}} \tag{1.32}$$

The static power is given by:

$$P_{stat_{M-mux}} = V_{DD} I_{leak_{M-mux}} \tag{1.33}$$

The dynamic power is given by:

$$P_{dyn_{M-mux}} = V_{DD}^2 f_{CLK} E_{sw} \cdot M(C_{in2_{tgate}} + C_{in3_{tgate}}) \tag{1.34}$$

The intrinsic delay is:

$$\tau_{int_{M-mux}} = \tau_{int_{inv}} + \alpha_{inv} M(C_{in2_{tgate}} + C_{in3_{tgate}}) + \tau_{int_{tgate}} \tag{1.35}$$

The $\alpha$ coefficient for the M-mux is:

$$\alpha_{M-mux} = \alpha_{tgate} \tag{1.36}$$

The input capacitances for each line of the x and y inputs are:

$$C_{in_{M-mux}}^{input} = C_{in_{mux_x}} = C_{in_{mux_y}} = C_{in1_{tgate}} \tag{1.37}$$

while the input capacitance seen from the select input is:

$$C_{in_{M-mux}}^{sel} = C_{in_{inv}} + M(C_{in2_{tgate}} + C_{in3_{tgate}}) \tag{1.38}$$

### Carry-select block

The schematic of the carry-select block and its corresponding black box are reported in Figure 1.13.

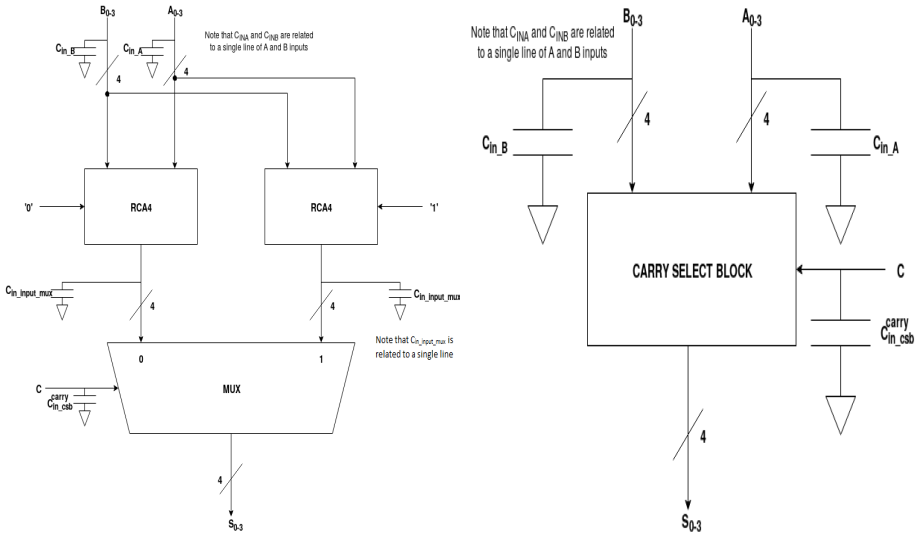



Figure 1.13: Carry-select block schematic view and block symbol

The area of the carry-select block is given by:

$$A_{csb} = 2A_{RCA4} + A_{4-mux} \tag{1.39}$$

The leakage current is:

$$I_{leak_{csb}} = 2I_{leak_{RCA4}} + I_{leak_{4-mux}} \tag{1.40}$$

The static power is given by:

$$P_{stat_{csb}} = V_{DD} I_{leak_{csb}} \tag{1.41}$$

The dynamic power is given by:

$$P_{dyn_{csb}} = V_{DD}^2 f_{CLK} E_{sw} (8C_{in_{4-mux}}^{input}) + 2P_{dyn_{RCA4}} + P_{dyn_{4-mux}} \tag{1.42}$$

Assuming that all the inputs $A_{0-3}$, $B_{0-3}$ and $C$ are stable, the delay is that of the path from $A_{0-3}$ and $B_{0-3}$ to $S_{0-3}$. Therefore the intrinsic delay (the delay without load) is:

$$\tau_{int_{csb}} = \tau_{int_{RCA4}} + \alpha_{RCA4} C_{in_{4-mux}}^{input} + \tau_{int_{4-mux}} \tag{1.43}$$

The $\alpha$ coefficient of the carry-select block is:

$$\alpha_{csb} = \alpha_{4-mux} \tag{1.44}$$

The input capacitance of a single input line ($A_{0-3}$ or $B_{0-3}$) is:

$$C_{in_{csb}}^{input} = C_{in_B} = C_{in_A} = 2C_{in_{RCA4}} \tag{1.45}$$

The carry input capacitance of the carry-select block is

$$C_{in_{csb}}^{carry} = C_{in_{4-mux}}^{sel} \tag{1.46}$$

### Carry-select sum generator

Carry-select sum generator (cssg) schematic is reported in figure 1.14 and its black box symbol in figure 1.15. In these figures the number of bits is set to 32.

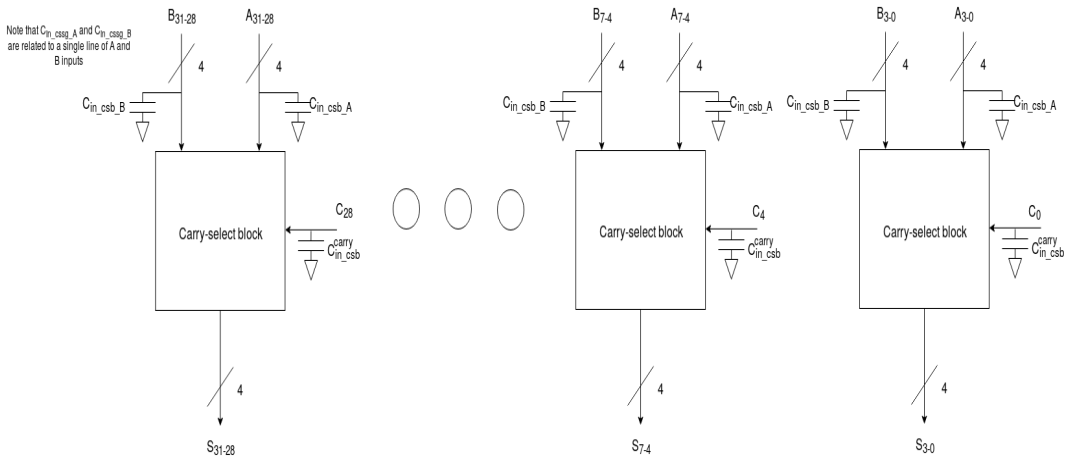



Figure 1.14: Carry-select sum generator schematic

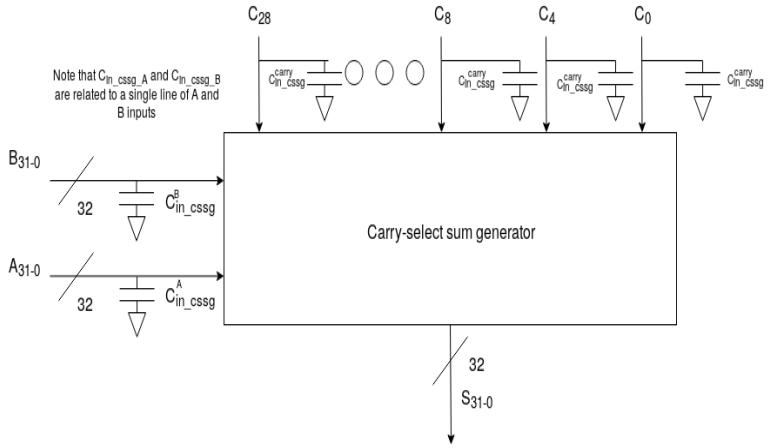



Figure 1.15: Carry-select sum generator black-box

Introducing $N$, which is the number of bits of the adder, the area can be derived:

$$A_{cssg} = \frac{N}{4} A_{csb} \tag{1.47}$$

Note that the values of $N$ considered are 8, 12, 16, 20, 24, 28 and 32.
The leakage current is:

$$I_{leak_{cssg}} = \frac{N}{4} I_{leak_{csb}} \tag{1.48}$$

The static power is:

$$P_{stat_{cssg}} = V_{DD} I_{leak_{cssg}} \tag{1.49}$$

The dynamic power is:

$$P_{dyn_{cssg}} = \frac{N}{4} P_{dyn_{csb}} \tag{1.50}$$

The intrinsic delay (the delay without load) is:

$$\tau_{int_{cssg}} = \tau_{int_{csb}} \tag{1.51}$$

The $\alpha$ parameter is:

$$\alpha_{cssg} = \alpha_{csb} \tag{1.52}$$

The input capacitances for each line of the A and B inputs are:

$$C_{in_{cssg}}^{input} = C_{in_{cssg}}^{A} = C_{in_{cssg}}^{B} = C_{in_{csb}}^{input} \tag{1.53}$$

while the input capacitance seen from each carry input is:

$$C_{in_{cssg}}^{carry} = C_{in_{csb}}^{carry} \tag{1.54}$$

### 1.4.2 Carry-merge sparse tree

In this subsection the three blocks that constitute the carry-merge sparse tree are presented. They are the pre-computation block (pcb), the propagate-generate block (pg) and the G-block (gb). Using these blocks the carry-merge sparse tree can be designed.

### Pre-computation block

In figure 1.16 it is shown the symbol and the schematic view of the pre-computation block.

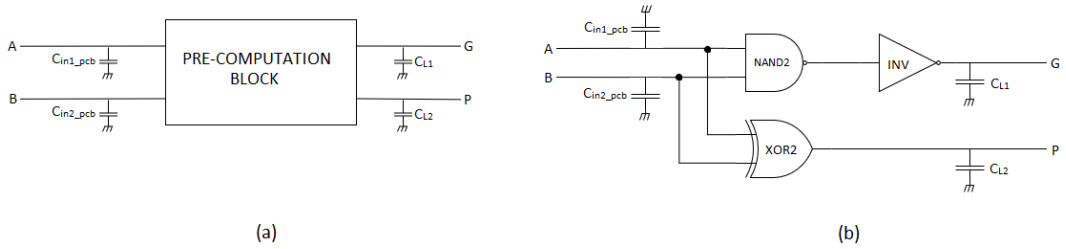



Figure 1.16: Pre-computation block. (a) Symbol. (b) Schematic view.

The area of the pre-computation block is given by:

$$A_{pcb} = A_{nand2} + A_{inv} + A_{xor2} \tag{1.55}$$

The leakage current is:

$$I_{leak_{pcb}} = I_{leak_{nand2}} + I_{leak_{inv}} + I_{leak_{xor2}} \tag{1.56}$$

The static power is:

$$P_{stat_{pcb}} = V_{DD} \cdot I_{leak_{pcb}} \tag{1.57}$$

The dynamic power is:

$$P_{dyn_{pcb}} = V_{DD}^2 f_{CLK} E_{sw} C_{in_{inv}} \tag{1.58}$$

The intrinsic delay (the delay without load) of the output $G$ is:

$$\tau_{int_{pcb_G}} = \tau_{int_{nand2}} + \alpha_{nand2} C_{in_{inv}} + \tau_{int_{inv}} \tag{1.59}$$

Then the $\alpha$ parameter is:

$$\alpha_{pcb_G} = \alpha_{inv} \tag{1.60}$$

Thus the total delay of $G$ (considering the load) is $\tau_{int_{pcb_G}} + \alpha_{pcb_G} C_{L1}$.

Analogously for the output $P$ we have:

$$\tau_{int_{pcb_P}} = \tau_{int_{xor2}} \tag{1.61}$$

$$\alpha_{pcb_P} = \alpha_{xor2} \tag{1.62}$$

The delay of $P$ considering the load will be $\tau_{int_{pcb_P}} + \alpha_{pcb_P} C_{L2}$.

The input capacitances of the pre-computation block are:

$$C_{in_{pcb}} = C_{in1_{pcb}} = C_{in2_{pcb}} = C_{in_{nand2}} + C_{in_{xor2}} \tag{1.63}$$

### Propagate-generate block

Figure 1.17 shows the propagate-generate block (pg-block) schematic and figure 1.18 its black-box model.

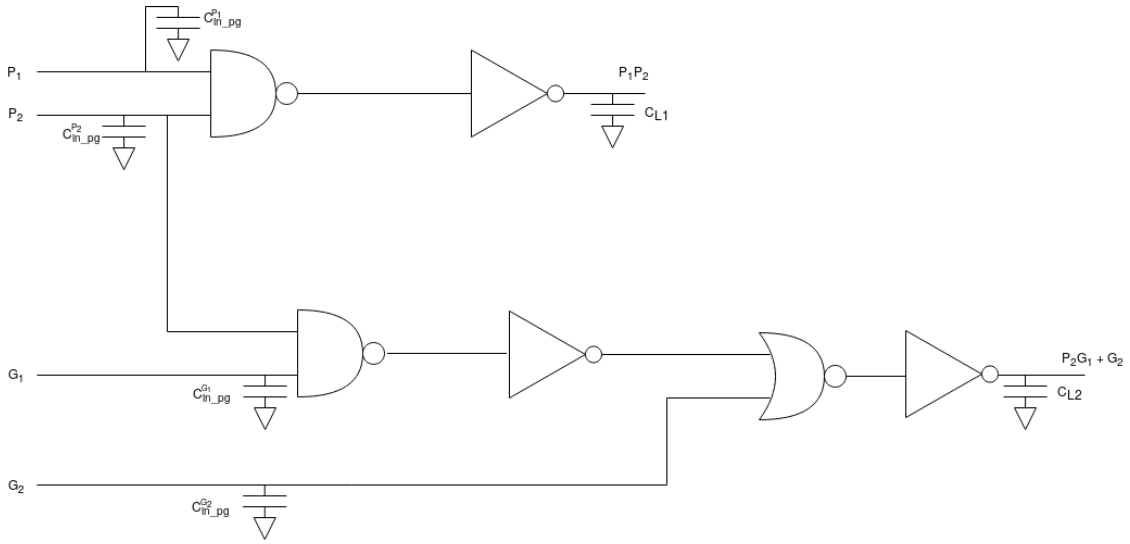



Figure 1.17: Schematic of propagate and generate carry block

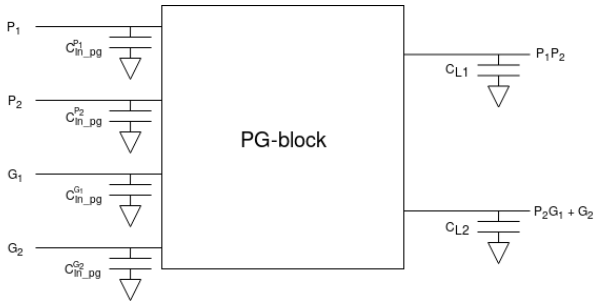



Figure 1.18: Symbol of propagate and generate carry block

From the schematic figure, the area can be evaluated as:

$$A_{pg} = 2A_{nand2} + A_{nor2} + 3A_{inv} \tag{1.64}$$

The leakage current is:

$$I_{leak_{pg}} = 2I_{leak_{nand2}} + I_{leak_{nor2}} + 3I_{leak_{inv}} \tag{1.65}$$

The static power is:

$$P_{stat_{pg}} = V_{DD} I_{leak_{pg}} \tag{1.66}$$

The dynamic power is:

$$P_{dyn_{pg}} = V_{DD}^2 f_{CLK} E_{sw} (3C_{in_{inv}} + C_{in_{nor2}}) \tag{1.67}$$

The intrinsic delay (delay without load) for the $P_1P_2$ output is:

$$\tau_{int_{pg_p}} = \tau_{int_{nand2}} + \alpha_{nand2} C_{in_{inv}} + \tau_{int_{inv}} \tag{1.68}$$

The $\alpha$ parameter is:

$$\alpha_{pg_p} = \alpha_{inv} \tag{1.69}$$

Thus the overall delay for the $P_1P_2$ output is $\tau_{int_{pg_p}} + \alpha_{pg_p} C_{L1}$.

The intrinsic delay for the $P_2G_1 + G_2$ output is:

$$\tau_{int_{pg_g}} = \tau_{int_{nand2}} + \alpha_{nand2} C_{in_{inv}} + \tau_{int_{inv}} + \alpha_{inv} C_{in_{nor2}} + \tau_{int_{nor2}} + \alpha_{nor2} C_{in_{inv}} + \tau_{int_{inv}} \tag{1.70}$$

The $\alpha$ parameter is:

$$\alpha_{pg_g} = \alpha_{inv} \tag{1.71}$$

The overall delay of this output is $\tau_{int_{pg_g}} + \alpha_{pg_g} C_{L2}$.

In order from $P_1$ to $G_2$, the input capacitances are:

$$C_{in_{pg}}^{P_1} = C_{in_{nand2}} \tag{1.72}$$

$$C_{in_{pg}}^{P_2} = 2C_{in_{nand2}} \tag{1.73}$$

$$C_{in_{pg}}^{G_1} = C_{in_{nand2}} \tag{1.74}$$

$$C_{in_{pg}}^{G_2} = C_{in_{nor2}} \tag{1.75}$$

Then the total input capacitance for this block is defined as:

$$C_{in_{pg}}^{tot} = C_{in_{pg}}^{P_1} + C_{in_{pg}}^{P_2} + C_{in_{pg}}^{G_1} + C_{in_{pg}}^{G_2} \tag{1.76}$$

### G-block

This block is the same of the previous one but there is only the second output (G-output). In figure 1.19 it is shown the symbol and the schematic view of the G-block.

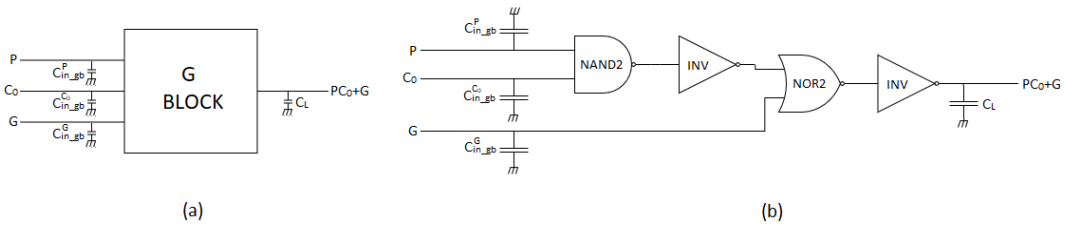



Figure 1.19: G-block. (a) Symbol. (b) Schematic view.

The area of the G-block is given by:

$$A_{gb} = A_{nand2} + A_{nor2} + 2A_{inv} \tag{1.77}$$

The leakage current is:

$$I_{leak_{gb}} = I_{leak_{nand2}} + I_{leak_{nor2}} + 2I_{leak_{inv}} \tag{1.78}$$

The static power is:

$$P_{stat_{gb}} = V_{DD} \cdot I_{leak_{gb}} \tag{1.79}$$

The dynamic power is:

$$P_{dyn_{gb}} = V_{DD}^2 f_{CLK} E_{sw} (2C_{in_{inv}} + C_{in_{nor2}}) \tag{1.80}$$

The intrinsic delay (the delay without load) is:

$$\tau_{int_{gb}} = \tau_{int_{nand2}} + \alpha_{nand2} C_{in_{inv}} + \tau_{int_{inv}} + \alpha_{inv} C_{in_{nor2}} + \tau_{int_{nor2}} + \alpha_{nor2} C_{in_{inv}} + \tau_{int_{inv}} \tag{1.81}$$

Then the $\alpha$ parameter is:

$$\alpha_{gb} = \alpha_{inv} \tag{1.82}$$

Thus the total delay (considering the load) is $\tau_{int_{gb}} + \alpha_{gb} C_L$.

The input capacitances are:

$$C_{in_{gb}}^{P} = C_{in_{nand2}} \tag{1.83}$$

$$C_{in_{gb}}^{C_0} = C_{in_{nand2}} \tag{1.84}$$

$$C_{in_{gb}}^{G} = C_{in_{nor2}} \tag{1.85}$$

### Carry-merge sparse tree

In Figure 1.20 and in Figure 1.21 are respectively shown the black-box and the schematic view of the carry-merge sparse tree for a number of bit $N = 32$.

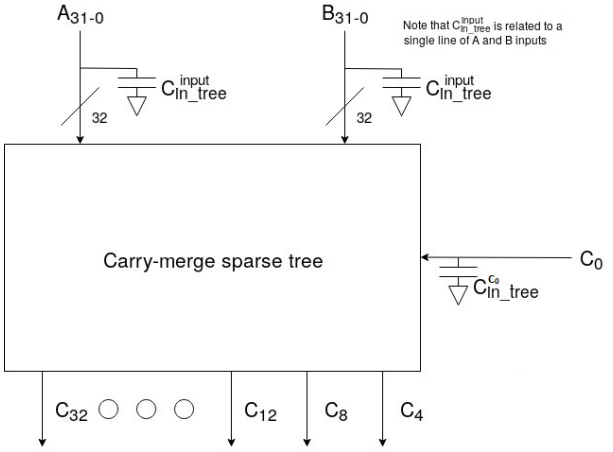



Figure 1.20: Symbol of carry-merge sparse tree with 32 bit

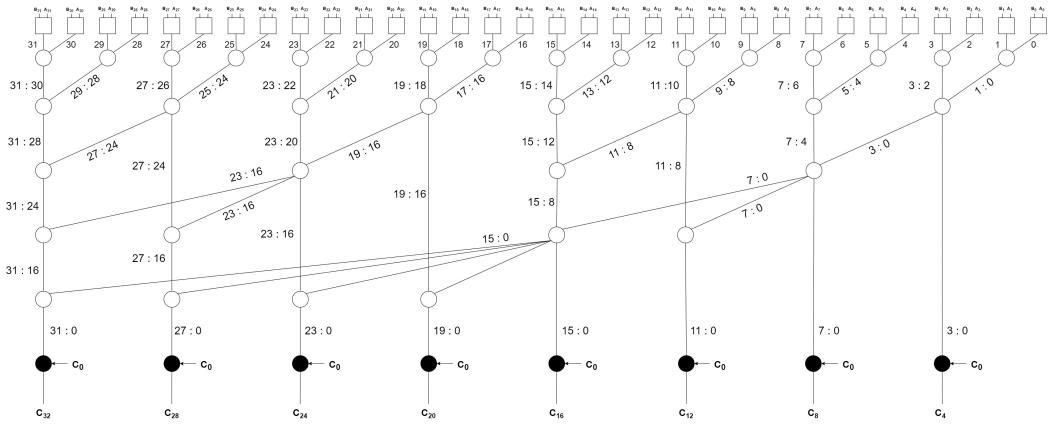



Figure 1.21: Schematic view of the carry-merge sparse tree with 32 bit

In the schematic view the white squares are pre-computation blocks (pcb), the white circles are propagate-generate blocks (pg) and the black circles are G-blocks (gb). Note that the pcb and pg-blocks have two output but in the schematic view there is only one line for both of them. This is only to have a less complicated picture of the schematic view.
The outputs of this tree are the carry from $C_4$ to $C_{32}$ with a step of 4.
Area, power consumption and delay of the tree are evaluated as a function of the number of bit $N$, in the case where $N$ can be equal to 8, 12, 16, 20, 24, 28 or 32.
The area of the tree is equal to:

$$A_{tree} = N_{pcb}A_{pcb} + N_{pg}A_{pg} + N_{gb}A_{gb} \tag{1.86}$$

Where:

$$N_{pcb} = N \tag{1.87}$$

and

$$N_{gb} = \frac{N}{4} \tag{1.88}$$

$N_{pg}$ cannot be written as a function of $N$ in a simple way. Therefore the number of pg-blocks for each case looking at the figure 1.21 can be counted.

$$N_{pg} = \begin{cases} 7 & N = 8 \\ 11 & N = 12 \\ 16 & N = 16 \\ 20 & N = 20 \\ 25 & N = 24 \\ 30 & N = 28 \\ 36 & N = 32 \end{cases} \tag{1.89}$$

The leakage current is given by:

$$I_{leak_{tree}} = N_{pcb} I_{leak_{pcb}} + N_{pg} I_{leak_{pg}} + N_{gb} I_{leak_{gb}} \tag{1.90}$$

The static power is:

$$P_{stat_{tree}} = V_{DD} \cdot I_{leak_{tree}} \tag{1.91}$$

The dynamic power is:

$$P_{dyn_{tree}} = V_{DD}^2 f_{CLK} E_{sw} \left[ N_{pg} C_{in_{pg}}^{tot} + N_{gb} \left( C_{in_{gb}}^{P} + C_{in_{gb}}^{G} \right) \right] + N_{pcb} P_{dyn_{pcb}} + N_{pg} P_{pyn_{pg}} + N_{gb} P_{dyn_{gb}} \tag{1.92}$$

Without doing simplifications (for instance considering an upper bound for the delay instead of the real delay), there is not a simple way to evaluate the delay of the tree as a function of $N$. This is why when $N$ changes, the capacitances in the nodes of the tree can change.
To evaluate the delay of the tree no simplifications are used. For this reason, each case of $N$ is considered separately.

First of all, we note that the pcb-blocks have always the same load for each value of $N$. Thus we can compute the total delay of a pcb-block as:

$$\tau_{pcb_{tree}} = max\left( \tau_{pcb_{tree}}^{G}, \tau_{pcb_{tree}}^{P} \right) \tag{1.93}$$

where $\tau_{pcb_{tree}}^{G}$ is the delay related to the $G$ output while $\tau_{pcb_{tree}}^{P}$ is related to the $P$ one. Since the load of each the pcb-block is always one pg-block, it can be written:

$$\tau_{pcb_{tree}}^{G} = \tau_{int_{pcb_G}} + \alpha_{pcb_G} \cdot max\left( C_{in_{pg}}^{G_1}, C_{in_{pg}}^{G_2} \right) \tag{1.94}$$

$$\tau_{pcb_{tree}}^{P} = \tau_{int_{pcb_P}} + \alpha_{pcb_P} \cdot max\left( C_{in_{pg}}^{P_1}, C_{in_{pg}}^{P_2} \right) \tag{1.95}$$

In order to compute the delay of the tree for each value of $N$, it is useful computing the delay of the pg-block for different loads.
$\tau_{pg_{xy}}^{G}$ is the delay related to the output $G$ of a pg-block when the load is a number $x$ of pg-blocks and a number $y$ of G-blocks. Similarly $\tau_{pg_{xy}}^{P}$ is the delay related to the output $P$ of a pg-block when the load is a number $x$ of pg-blocks and a number $y$ of G-blocks.

Thus the delay of a pg-block with a load of 1 pg-block and 0 G-block is:

$$\tau_{pg_{10}} = max\left( \tau_{pg_{10}}^{G}, \tau_{pg_{10}}^{P} \right) \tag{1.96}$$

where:

$$\tau_{pg_{10}}^{G} = \tau_{int_{pg_g}} + \alpha_{pg_g} \cdot max\left( C_{in_{pg}}^{G_1}, C_{in_{pg}}^{G_2} \right) \tag{1.97}$$

$$\tau_{pg_{10}}^{P} = \tau_{int_{pg_p}} + \alpha_{pg_p} \cdot max\left( C_{in_{pg}}^{P_1}, C_{in_{pg}}^{P_2} \right) \tag{1.98}$$

The delay of a pg-block with a load of 1 pg-block and 1 G-block is:

$$\tau_{pg_{11}} = max\left(\tau^G_{pg_{11}}, \tau^P_{pg_{11}}\right) \tag{1.99}$$

where:

$$\tau^G_{pg_{11}} = \tau_{int_{pg_g}} + \alpha_{pg_g} \cdot \left[max\left(C^{G_1}_{in_{pg}}, C^{G_2}_{in_{pg}}\right) + C^G_{in_{gb}}\right] \tag{1.100}$$

$$\tau^P_{pg_{11}} = \tau_{int_{pg_p}} + \alpha_{pg_p} \cdot \left[max\left(C^{P_1}_{in_{pg}}, C^{P_2}_{in_{pg}}\right) + C^P_{in_{gb}}\right] \tag{1.101}$$

The delay of a pg-block with a load of 2 pg-block and 1 G-block is:

$$\tau_{pg_{21}} = max\left(\tau^G_{pg_{21}}, \tau^P_{pg_{21}}\right) \tag{1.102}$$

where:

$$\tau^G_{pg_{21}} = \tau_{int_{pg_g}} + \alpha_{pg_g} \cdot \left[2max\left(C^{G_1}_{in_{pg}}, C^{G_2}_{in_{pg}}\right) + C^G_{in_{gb}}\right] \tag{1.103}$$

$$\tau^P_{pg_{21}} = \tau_{int_{pg_p}} + \alpha_{pg_p} \cdot \left[2max\left(C^{P_1}_{in_{pg}}, C^{P_2}_{in_{pg}}\right) + C^P_{in_{gb}}\right] \tag{1.104}$$

The delay of a pg-block with a load of 4 pg-block and 1 G-block is:

$$\tau_{pg_{41}} = max\left(\tau^G_{pg_{41}}, \tau^P_{pg_{41}}\right) \tag{1.105}$$

where:

$$\tau^G_{pg_{41}} = \tau_{int_{pg_g}} + \alpha_{pg_g} \cdot \left[4max\left(C^{G_1}_{in_{pg}}, C^{G_2}_{in_{pg}}\right) + C^G_{in_{gb}}\right] \tag{1.106}$$

$$\tau^P_{pg_{41}} = \tau_{int_{pg_p}} + \alpha_{pg_p} \cdot \left[4max\left(C^{P_1}_{in_{pg}}, C^{P_2}_{in_{pg}}\right) + C^P_{in_{gb}}\right] \tag{1.107}$$

The delay of a pg-block with a load of 0 pg-block and 1 G-block is:

$$\tau_{pg_{01}} = max\left(\tau^G_{pg_{01}}, \tau^P_{pg_{01}}\right) \tag{1.108}$$

where:

$$\tau^G_{pg_{01}} = \tau_{int_{pg_g}} + \alpha_{pg_g} \cdot C^G_{in_{gb}} \tag{1.109}$$

$$\tau^P_{pg_{01}} = \tau_{int_{pg_p}} + \alpha_{pg_p} \cdot C^P_{in_{gb}} \tag{1.110}$$

Now the intrinsic delay (the delay without load) of the tree, for each value of $N$, can be evaluated. We will denote the intrinsic delay as $\tau^N_{int_{tree}}$ where $N$ is the number of bit. Looking at the figure 1.21 we can write:

$$\tau^8_{int_{tree}} = \tau_{pcb_{tree}} + \tau_{pg_{10}} + \tau_{pg_{11}} + \tau_{pg_{01}} + \tau_{int_{gb}} \tag{1.111}$$

$$\tau^{12}_{int_{tree}} = \tau_{pcb_{tree}} + \tau_{pg_{10}} + \tau_{pg_{11}} + \tau_{pg_{21}} + \tau_{pg_{01}} + \tau_{int_{gb}} \tag{1.112}$$

$$\tau^{16}_{int_{tree}} = \tau^{12}_{int_{tree}} \tag{1.113}$$

$$\tau^{20}_{int_{tree}} = \tau_{pcb_{tree}} + \tau_{pg_{10}} + \tau_{pg_{11}} + \tau_{pg_{21}} + \tau_{pg_{41}} + \tau_{int_{gb}} \tag{1.114}$$

$$\tau^{24}_{int_{tree}} = \tau^{28}_{int_{tree}} = \tau^{32}_{int_{tree}} = \tau^{20}_{int_{tree}} \tag{1.115}$$

The $\alpha$ coefficient of the tree does not depend on $N$ and it is equal to:

$$\alpha_{tree} = \alpha_{gb} \tag{1.116}$$

The input capacitance for each line of the $A_{31-0}$ and $B_{31-0}$ inputs is:

$$C^{input}_{in_{tree}} = C_{in_{pcb}} \tag{1.117}$$

while the input capacitance seen from the $C_0$:

$$C^{C_0}_{in_{tree}} = N_{gb} C^{C_0}_{in_{gb}} \tag{1.118}$$

### 1.4.3 Pentium IV adder

The overall architecture can be summarized into the schematic in figure 1.22. The black-box symbol is reported in figure 1.23.

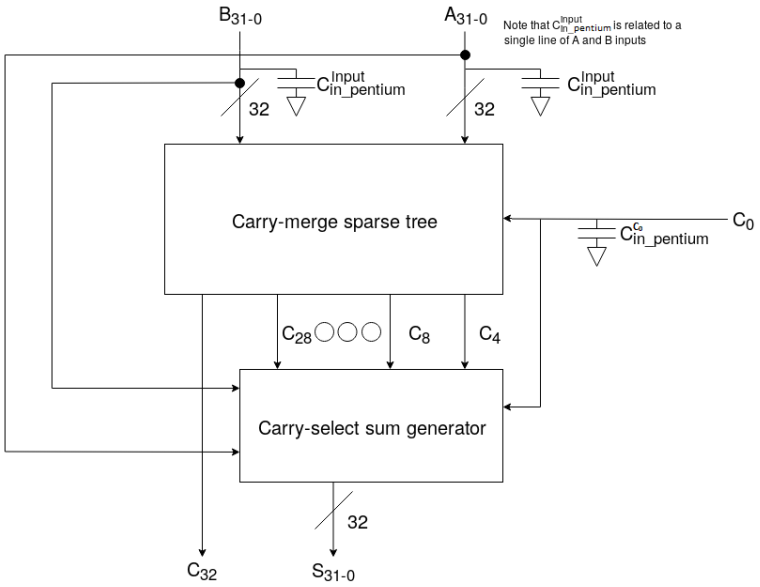



Figure 1.22: Schematic of the 32-bit Pentium IV adder

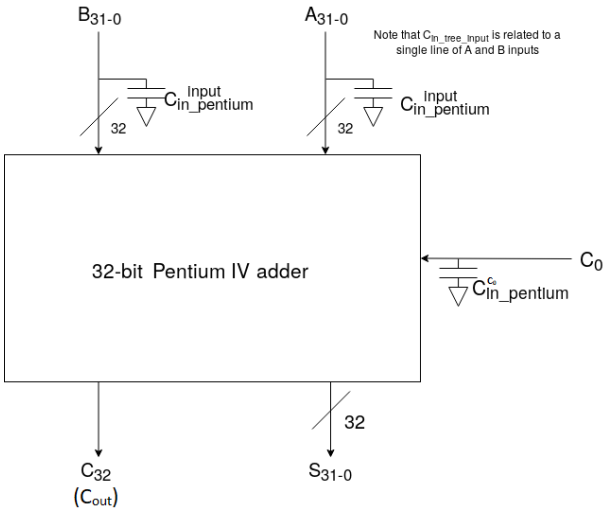



Figure 1.23: Pentium IV black-box view

The area can be evaluated as:

$$A_{pentium} = A_{tree} + A_{cssg} \tag{1.119}$$

The leakage current is:

$$I_{leak_{pentium}} = I_{leak_{tree}} + I_{leak_{cssg}} \tag{1.120}$$

The static power is:

$$P_{stat_{pentium}} = V_{DD} I_{leak_{pentium}} \tag{1.121}$$

The dynamic power is:

$$P_{dyn_{pentium}} = V_{DD}^2 f_{CLK} E_{sw} \left( \frac{N}{4} - 1 \right) C_{in_{cssg}}^{carry} + P_{dyn_{tree}} + P_{dyn_{cssg}} \tag{1.122}$$

The input signals $A_{31-0}$ and $B_{31-0}$ enter both in the tree and in the carry-select sum generator. Thus the tree and all RCA4s inside the carry-select sum generator work in parallel. This means that if the delay of the tree ($\tau^N_{int_{tree}} + \alpha_{tree} C^{carry}_{in_{cssg}}$) is greater than the delay of the RCA4 ($\tau_{int_{RCA4}} + \alpha_{RCA4} C^{input}_{in_{4-mux}}$), the delay of the Pentium without load will be $\tau^N_{int_{tree}} + \alpha_{tree} C^{carry}_{in_{cssg}} + \tau_{int_{4-mux}}$ where $C^{carry}_{in_{cssg}} = C^{carry}_{in_{csb}}$. On the contrary, if the delay of the tree is smaller than the delay of the RCA4 (usually this case does not happen if the number of bit is big), the delay of the the Pentium without load will be $\tau_{int_{cssg}} = \tau_{int_{csb}} = \tau_{int_{RCA4}} + \alpha_{RCA4} C^{input}_{in_{4-mux}} + \tau_{int_{4-mux}}$.

Therefore the delay of the Pentium without is:

$$\tau_{int_{pentium}} = max\left(\tau^N_{int_{tree}} + \alpha_{tree} C^{carry}_{in_{cssg}} + \tau_{int_{4-mux}}, \tau_{int_{cssg}}\right) \tag{1.123}$$

The $\alpha$ parameter is:

$$\alpha_{pentium} = \alpha_{cssg} \tag{1.124}$$

The input capacitance for each line of the $A_{31-0}$ and $B_{31-0}$ inputs is:

$$C^{input}_{in_{pentium}} = C^{input}_{in_{tree}} + C^{input}_{in_{cssg}} \tag{1.125}$$

while the input capacitance seen from the $C_0$:

$$C^{C_0}_{in_{pentium}} = C^{C_0}_{in_{tree}} + C^{carry}_{in_{cssg}} \tag{1.126}$$

Now the working frequency can be estimated. As already said in section 1.3, the inverse of the delay of the Pentium IV without load is used as working frequency.

$$f_{CLK} = \frac{1}{\tau_{int_{pentium}}} \tag{1.127}$$

## 1.5 Matlab implementation

All the formulas in this report have been implemented in Matlab in order to evaluate area, power consumption and delay of the Pentium IV adder. Obviously the values of the input parameters described in Table 1.1 are set to see the results, but the characterization these logic gates is assumed already done by other works. For this reason reasonable values for the input parameters, considering the internal structure of the logic CMOS gates, have been used.
In the following is reported the Matlab code.

```
clear all
close all

Vdd = 1; %V
Esw = 1;
N = [8,12,16,20,24,28,32];

% INPUT PARAMETERS

% NAND2
A_nand2 = 0.5; % (um)^2
Cin_nand2 = 4; %fF
tau_int_nand2 = 10.3; %ps
alpha_nand2 = 3; %KOhm
I_leak_nand2 = 200; %nA

% NOR2
A_nor2 = 0.625; % (um)^2
Cin_nor2 = 5; %fF
tau_int_nor2 = 10.7; %ps
alpha_nor2 = 3; %KOhm
I_leak_nor2 = 200; %nA
```

```
23
24 % INVERTER
25 A_inv = 0.1875; % (um)^2
26 Cin_inv = 3; %fF
27 tau_int_inv = 9; %ps
28 alpha_inv = 3; %KOhm
29 I_leak_inv = 100; %nA
30
31 % XOR2
32 A_xor2 = 1.875; % (um)^2
33 Cin_xor2 = 6; %fF
34 tau_int_xor2 = 12.6; %ps
35 alpha_xor2 = 3; %KOhm
36 I_leak_xor2 = 400; %nA
37
38 % TRANSMISSION GATE
39 A_tgate = 0.1875; % (um)^2
40 Cin1_tgate = 0.5; %fF
41 Cin2_tgate = 1; %fF
42 Cin3_tgate = 2; %fF
43 tau_int_tgate = 5.8; %ps
44 alpha_tgate = 3; %KOhm
45 I_leak_tgate = 50; %nA
46
47 % END INPUT PARAMETERS
48
49 % FULL ADDER
50 A_fa = 2*A_xor2 + 2*A_nand2 + A_nor2 + 3*A_inv;
51 Cin_fa = Cin_xor2 + Cin_nand2;
52 I_leak_fa = 2*I_leak_xor2 + 2*I_leak_nand2 + I_leak_nor2 + 3*I_leak_inv;
53 P_stat_fa = Vdd*I_leak_fa;
54 E_dyn_fa = (Vdd^2)*Esw*(Cin_xor2 + Cin_nand2 + 3*Cin_inv + 2*Cin_nor2);
55 tau_int_fa_sum = 2*tau_int_xor2 + alpha_xor2*(Cin_xor2+Cin_nand2);
56 alpha_fa_sum = alpha_xor2;
57 tau_int_fa_carry = tau_int_xor2 + alpha_xor2*(Cin_xor2+Cin_nand2) + tau_int_nand2 ...
       + alpha_nand2*Cin_inv + tau_int_inv + alpha_inv*Cin_nor2 + tau_int_nor2 + ...
       alpha_nor2*Cin_inv + tau_int_inv;
58 alpha_fa_carry = alpha_inv;
59
60 % MULTIPLEXER PARALLELISM 4
61 M = 4; %parallelism of mux inputs
62 Cin_inputs_mux_par4 = Cin1_tgate; %for each line of the inputs
63 Cin_sel_mux_par4 = Cin_inv + M*(Cin2_tgate+Cin3_tgate);
64 A_mux_par4 = A_inv + 2*M*A_tgate;
65 P_stat_mux_par4 = Vdd*(I_leak_inv + 2*M*I_leak_tgate);
66 E_dyn_mux_par4 = (Vdd^2)*Esw*M*(Cin2_tgate+Cin3_tgate);
67 tau_int_mux_par4 = tau_int_inv + alpha_inv*M*(Cin2_tgate+Cin3_tgate) + ...
       tau_int_tgate;
68 alpha_mux_par4 = alpha_tgate;
69
70 % MULTIPLEXER PARALLELISM 33
71 M = 33; %parallelism of mux inputs
72 Cin_inputs_mux_par33 = Cin1_tgate; %for each line of the inputs
73 Cin_sel_mux_par33 = Cin_inv + M*(Cin2_tgate+Cin3_tgate);
74 A_mux_par33 = A_inv + 2*M*A_tgate;
75 P_stat_mux_par33 = Vdd*(I_leak_inv + 2*M*I_leak_tgate);
76 E_dyn_mux_par33 = (Vdd^2)*Esw*M*(Cin2_tgate+Cin3_tgate);
77 tau_int_mux_par33 = tau_int_inv + alpha_inv*M*(Cin2_tgate+Cin3_tgate) + ...
       tau_int_tgate;
78 alpha_mux_par33 = alpha_tgate;
79
80 % RIPPLE CARRY ADDER 4 BITS
81 Cin_rca4 = Cin_fa; % for the carry input and for each line of the two data inputs
82 A_rca4 = 4*A_fa;
83 P_stat_rca4 = 4*P_stat_fa;
84 E_dyn_rca4 = (Vdd^2)*Esw*(3*Cin_fa) + 4*E_dyn_fa;
85 tau_int_rca4 = 3*tau_int_fa_carry + 3*alpha_fa_carry*Cin_fa + tau_int_fa_sum;
86 alpha_rca4 = alpha_fa_sum;
87
88 % CARRY-SELECT BLOCK
```

```
89  Cin_inputs_csb = 2*Cin_rca4; % for each line of the inputs
90  Cin_carry_csb = Cin_sel_mux_par4;
91  A_csb = 2*A_rca4 + A_mux_par4;
92  P_stat_csb = 2*P_stat_rca4 + P_stat_mux_par4;
93  E_dyn_csb = (Vdd^2)*Esw*8*Cin_inputs_mux_par4 + 2*E_dyn_rca4 + E_dyn_mux_par4;
94  tau_int_csb = tau_int_rca4 + alpha_rca4*Cin_inputs_mux_par4 + tau_int_mux_par4;
95  alpha_csb = alpha_mux_par4;
96
97  % CARRY-SELECT SUM GENERATOR
98  Cin_data_inputs_cssg = Cin_inputs_csb; % for each line of the inputs
99  Cin_carry_cssg = Cin_carry_csb; % for each carry line
100 A_cssg = (N/4)*A_csb;
101 P_stat_cssg = (N/4)*P_stat_csb;
102 E_dyn_cssg = (N/4)*E_dyn_csb;
103 tau_int_cssg = tau_int_csb;
104 alpha_cssg = alpha_csb;
105
106 % PRE COMPUTATION BLOCK
107 Cin_pcb = Cin_nand2 + Cin_xor2;
108 A_pcb = A_nand2 + A_inv + A_xor2;
109 I_leak_pcb = I_leak_nand2 + I_leak_inv + I_leak_xor2;
110 P_stat_pcb = Vdd*I_leak_pcb;
111 E_dyn_pcb = (Vdd^2)*Esw*Cin_inv;
112 tau_int_pcb_g = tau_int_nand2 + alpha_nand2*Cin_inv + tau_int_inv; % related to g ...
        output
113 alpha_pcb_g = alpha_inv; % related to g output
114 tau_int_pcb_p = tau_int_xor2; % related to p output
115 alpha_pcb_p = alpha_xor2; % related to p output
116
117 % PROPAGATED AND GENERATED CARRY BLOCK
118 Cin_pg_p1 = Cin_nand2;
119 Cin_pg_p2 = 2*Cin_nand2;
120 Cin_pg_g1 = Cin_nand2;
121 Cin_pg_g2 = Cin_nor2;
122 Cin_pg_tot = Cin_pg_p1 + Cin_pg_p2 + Cin_pg_g1 + Cin_pg_g2;
123 A_pg = 2*A_nand2 + A_nor2 + 3*A_inv;
124 I_leak_pg = 2*I_leak_nand2 + I_leak_nor2 + 3*I_leak_inv;
125 P_stat_pg = Vdd*I_leak_pg;
126 E_dyn_pg = (Vdd^2)*Esw*(3*Cin_inv+Cin_nor2);
127 tau_int_pg_g = tau_int_nand2 + alpha_nand2*Cin_inv + tau_int_inv + alpha_inv*...
        Cin_nor2 + tau_int_nor2 + alpha_nor2*Cin_inv + tau_int_inv; % related to g ...
        output
128 alpha_pg_g = alpha_inv; % related to g output
129 tau_int_pg_p = tau_int_nand2 + alpha_nand2*Cin_inv + tau_int_inv; % related to p ...
        output
130 alpha_pg_p = alpha_inv; % related to p output
131
132 % ONLY G BLOCK
133 Cin_gb_p = Cin_nand2;
134 Cin_gb_c0 = Cin_nand2;
135 Cin_gb_g = Cin_nor2;
136 A_gb = A_nand2 + A_nor2 + 2*A_inv;
137 I_leak_gb = I_leak_nand2 + I_leak_nor2 + 2*I_leak_inv;
138 P_stat_gb = Vdd*I_leak_gb;
139 E_dyn_gb = (Vdd^2)*Esw*(2*Cin_inv+Cin_nor2);
140 tau_int_gb = tau_int_nand2 + alpha_nand2*Cin_inv + tau_int_inv + alpha_inv*...
        Cin_nor2 + tau_int_nor2 + alpha_nor2*Cin_inv + tau_int_inv;
141 alpha_gb = alpha_inv;
142
143 % TREE
144 N_of_pcb_block = N;
145 N_of_pg_block = [7,11,16,20,25,30,36];
146 N_of_gb_block = N/4;
147
148 Cin_data_inputs_tree = Cin_pcb; % for each line of the data inputs A and B
149 Cin_C0_tree = N_of_gb_block*Cin_gb_c0;
150
151 A_tree = N_of_pcb_block*A_pcb + N_of_pg_block*A_pg + N_of_gb_block*A_gb;
152 P_stat_tree = N_of_pcb_block*P_stat_pcb + N_of_pg_block*P_stat_pg + N_of_gb_block*...
        P_stat_gb;
```

```
153 E_dyn_tree = (Vdd^2)*Esw*(N_of_pg_block*Cin_pg_tot + N_of_gb_block*(Cin_gb_p+...
        Cin_gb_g)) + N_of_pcb_block*E_dyn_pcb + N_of_pg_block*E_dyn_pg + N_of_gb_block...
        *E_dyn_gb;
154
155 % delay of a pcb block with one pg block as load
156 tau_pcb_tree_g = tau_int_pcb_g + alpha_pcb_g*max(Cin_pg_g1,Cin_pg_g2);
157 tau_pcb_tree_p = tau_int_pcb_p + alpha_pcb_p*max(Cin_pg_p1,Cin_pg_p2);
158 tau_pcb_tree = max(tau_pcb_tree_g,tau_pcb_tree_p);
159
160 % delay of a pg block with 1 pg block and 0 gb block as load
161 tau_pg_tree_1_0_g = tau_int_pg_g + alpha_pg_g*(1*max(Cin_pg_g1,Cin_pg_g2));
162 tau_pg_tree_1_0_p = tau_int_pg_p + alpha_pg_p*(1*max(Cin_pg_p1,Cin_pg_p2));
163 tau_pg_tree_1_0 = max(tau_pg_tree_1_0_g,tau_pg_tree_1_0_p);
164
165 % delay of a pg block with 1 pg block and 1 gb block as load
166 tau_pg_tree_1_1_g = tau_int_pg_g + alpha_pg_g*(1*max(Cin_pg_g1,Cin_pg_g2)+Cin_gb_g...
        );
167 tau_pg_tree_1_1_p = tau_int_pg_p + alpha_pg_p*(1*max(Cin_pg_p1,Cin_pg_p2)+Cin_gb_p...
        );
168 tau_pg_tree_1_1 = max(tau_pg_tree_1_1_g,tau_pg_tree_1_1_p);
169
170 % delay of a pg block with 2 pg block and 1 gb block as load
171 tau_pg_tree_2_1_g = tau_int_pg_g + alpha_pg_g*(2*max(Cin_pg_g1,Cin_pg_g2)+Cin_gb_g...
        );
172 tau_pg_tree_2_1_p = tau_int_pg_p + alpha_pg_p*(2*max(Cin_pg_p1,Cin_pg_p2)+Cin_gb_p...
        );
173 tau_pg_tree_2_1 = max(tau_pg_tree_2_1_g,tau_pg_tree_2_1_p);
174
175 % delay of a pg block with 4 pg block and 1 gb block as load
176 tau_pg_tree_4_1_g = tau_int_pg_g + alpha_pg_g*(4*max(Cin_pg_g1,Cin_pg_g2)+Cin_gb_g...
        );
177 tau_pg_tree_4_1_p = tau_int_pg_p + alpha_pg_p*(4*max(Cin_pg_p1,Cin_pg_p2)+Cin_gb_p...
        );
178 tau_pg_tree_4_1 = max(tau_pg_tree_4_1_g,tau_pg_tree_4_1_p);
179
180 % delay of a pg block with 0 pg block and 1 gb block as load
181 tau_pg_tree_0_1_g = tau_int_pg_g + alpha_pg_g*Cin_gb_g;
182 tau_pg_tree_0_1_p = tau_int_pg_p + alpha_pg_p*Cin_gb_p;
183 tau_pg_tree_0_1 = max(tau_pg_tree_0_1_g,tau_pg_tree_0_1_p);
184
185 % intrinsic delay of the tree
186 tau_int_tree(1) = tau_pcb_tree + tau_pg_tree_1_0 + tau_pg_tree_1_1 + ...
        tau_pg_tree_0_1 + tau_int_gb; % N=8;
187 tau_int_tree(2) = tau_pcb_tree + tau_pg_tree_1_0 + tau_pg_tree_1_1 + ...
        tau_pg_tree_2_1 + tau_pg_tree_0_1 + tau_int_gb; % N=12;
188 tau_int_tree(3) = tau_int_tree(2); % N=16
189 tau_int_tree(4) = tau_pcb_tree + tau_pg_tree_1_0 + tau_pg_tree_1_1 + ...
        tau_pg_tree_2_1 + tau_pg_tree_4_1 + tau_int_gb; % N=20;
190 tau_int_tree(5) = tau_int_tree(4); % N=24
191 tau_int_tree(6) = tau_int_tree(4); % N=28
192 tau_int_tree(7) = tau_int_tree(4); % N=32
193
194 % the alpha tree in independent of N
195 alpha_tree = alpha_gb;
196
197 % PENTIUM IV
198 Cin_inputs_pentium = Cin_data_inputs_tree + Cin_data_inputs_cssg; % for each line ...
        of the inputs
199 Cin_C0_pentium = Cin_C0_tree + Cin_carry_cssg;
200 A_pentium = A_tree + A_cssg;
201 P_stat_pentium = P_stat_tree + P_stat_cssg; % nW
202 E_dyn_pentium = (Vdd^2)*Esw*((N/4)-1)*Cin_carry_cssg + E_dyn_tree + E_dyn_cssg;
203 tau_int_pentium = max(tau_int_tree + alpha_tree*Cin_carry_cssg + tau_int_mux_par4,...
         tau_int_cssg); % ps
204 alpha_pentium = alpha_cssg;
205
206 % MAXIMUM OPERATING FREQUENCY
207 fmax = 1./tau_int_pentium; % THz
208
209 % DYNAMIC POWER PENTIUM IV
```

```
210 P_dyn_pentium = E_dyn_pentium.*fmax; % mW
211
212 % PLOTS
213
214 figure(1)
215 hold on
216 grid on
217 plot(N,A_pentium,'.','MarkerSize' ,25);
218 title('Area Pentium IV as a function of N');
219 xlabel('N');
220 ylabel('A  (\mum^2)');
221
222 figure(2)
223 hold on
224 grid on
225 plot(N,P_stat_pentium./1000000,'.','MarkerSize' ,25);
226 title('Static power Pentium IV as a function of N');
227 xlabel('N');
228 ylabel('P_{stat}  (mW)');
229
230 figure(3)
231 hold on
232 grid on
233 plot(N,P_dyn_pentium,'.','MarkerSize' ,25);
234 title('Dynamic power Pentium IV as a function of N');
235 xlabel('N');
236 ylabel('P_{dyn}  (mW)');
237
238 figure(4)
239 hold on
240 grid on
241 plot(N,tau_int_pentium,'.','MarkerSize' ,25);
242 title('Delay without load Pentium IV as a function of N');
243 xlabel('N');
244 ylabel('Delay  (ps)');
```

## 1.6 Results

In this section are reported in graphs the results obtained for area, power and delay of the Pentium IV considering 8, 12, 16, 20, 24, 28 and 32 bits parallelism.
As expected the area increases by increasing the number of bits (figure 1.24). The static power increases with the same trend of the area (figure 1.25). This is why each block in the Pentium IV has a leakage current, thus if the area increases also the static power increases.
If the area increases ($N$ increases), it is expected also that the dynamic power increases as it can be seen in figure 1.26.
Figure 1.27 shows the delay without load of the Pentium IV. It is possible to notice how the delay follows instead a different trend: this can be explained remembering that for $N = 12$ and $N = 16$ the tree has the exact same delay (eq.1.113). The same happens also in the cases in which $N = 20$, $N = 24$, $N = 28$ and $N = 32$ (eq.1.115).

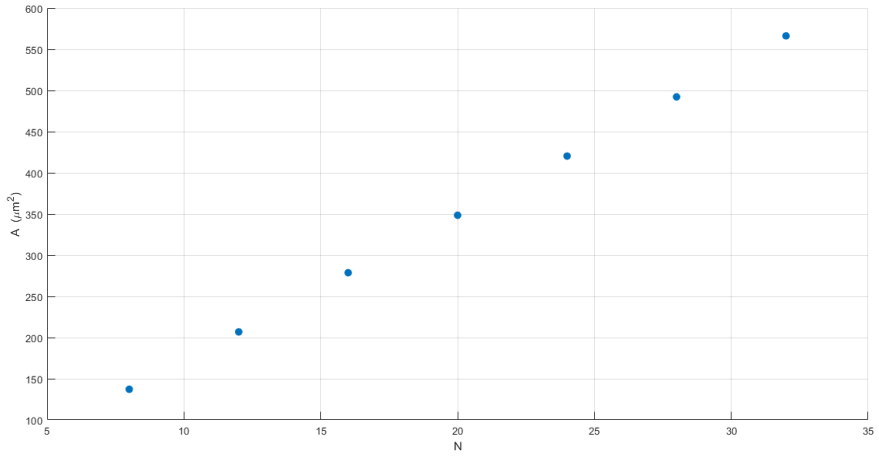



Figure 1.24: Area Pentium IV as a function of $N$

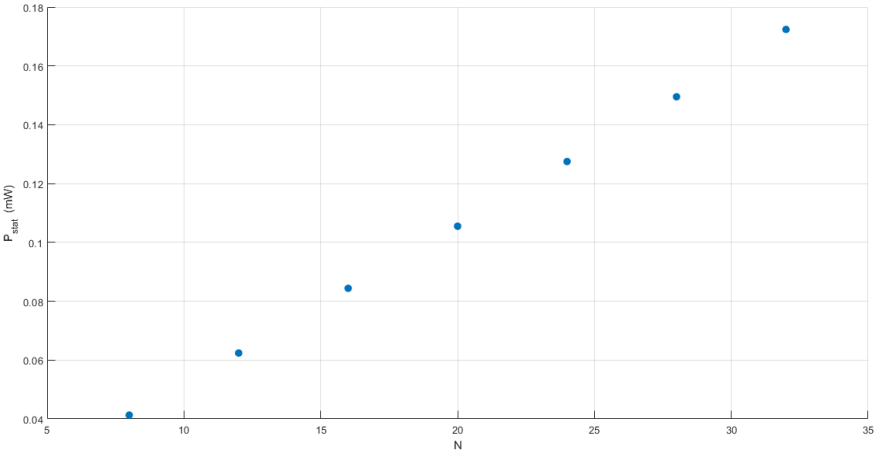



Figure 1.25: Static power Pentium IV as a function of $N$

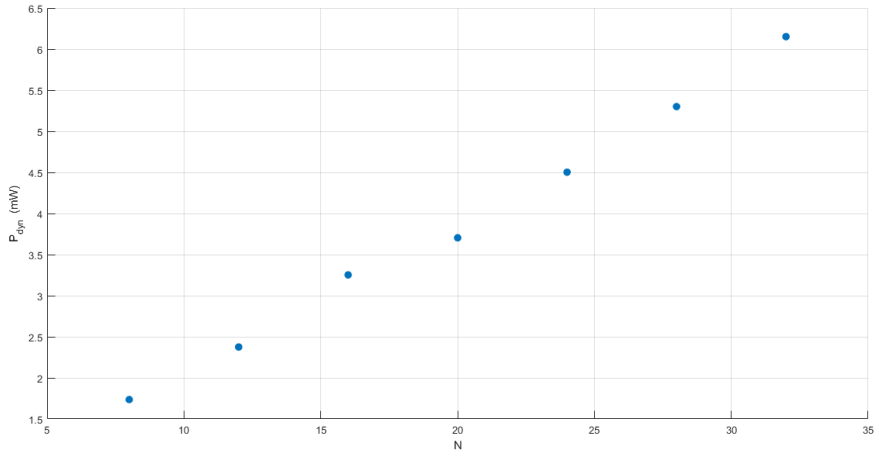



Figure 1.26: Dynamic power Pentium IV as a function of $N$

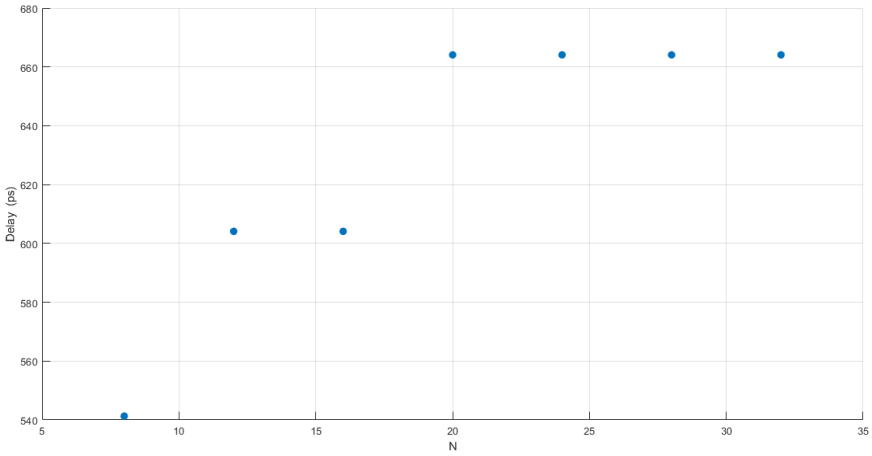



Figure 1.27: Delay without load Pentium IV as a function of $N$

# Chapter 2

# Pipelined Wallace tree

## 2.1 Introduction

The aim of this project is to implement an advanced **pipelined model** of a Wallace Tree Adder.

In particular, in the following sections, the generic structure of Wallace Tree will be described in detail and the focus will be on power consumption, occupied area and delay estimation. This particular model allows to introduce a variable number of pipeline stages along the Tree structure: the estimations will therefore take into account the pipeline structure contributions for area, power consumption and total delay. Calculations have been performed considering, for half adders and full adders, NAND2-based structures while for D flip flops both NAND2 and INV gates. Matlab implementation has been also reported.

The used technological parameters refer to *Bulk Technology HP 2009*.

## 2.2 Theoretical analysis

### 2.2.1 Wallace tree design

Wallace tree is an efficient structure in which half and full adders can be arranged in order to add multiple operands. This structrure is often used in multipliers in order to sum up together the previously-computed partial products. The main idea is to exploit a partial addition by columns.

The first step needed to implement a Wallace Tree Adder is to correctly align data so that partial terms with the same weight are in the same column.

At this point, the summands are grouped by three. The bits of each column are added together using a full adder or an half adder. An HA is able to compress two bits with weight $k$ to one bit of weight $k$ and one bit of weight $k+1$. FAs does the same thing with three bits instead allowing a compression ratio equal to 1.5. For this reason, starting from a structure with $n$ operands several levels of HAs and FAs are needed to compress these $n$ summands to two operands. Finally, when the tree has reduced the operands from $n$ to two, they are added together by a fast adder. [1]

The number of required levels $L$ and the maximum number of elements at each level can be obtained as follows. The maximum number of elements at the last level (before the final adder) is two ($l_0 = 2$).

At the generic level $j$ the maximum number of elements is instead given by the following expression:

$$l_j = \left\lfloor \frac{3}{2} l_{j-1} \right\rfloor$$

The number of required levels can be obtained by repeating this operation until $j \geq n$.

In order to better illustrate the behaviour of the system is often used the dot notation. An example can be seen in Figure 2.4 (this particular Figure has been generated through the Matlab code).

Wallace allocation of the adders is ASAP, this means that at each level dots are divided in three-dots-wide stripes and each stripe is filled with the maximum possible amount of FAs and HAs. Stripes that only have one or two operands are forwarded to the next level of the tree structure.

This execution flow can also be pipelined: in particular, in this work, different Wallace tree solutions have been explored and compared in terms of pipeline depth. Positive-edge triggered D flip flops have been inserted to store the outputs of the adders (sum and carry bits). In this way the computation performed by the tree is splitted onto multiple clock cycles depending on the depth of the tree.

### 2.2.2 Occupied Area

In order to evaluate the occupied area of the Wallace Tree, the number of full adders, half adders and D flip flops have been considered. The adders have been implemented by using exclusively ND2 gate while flip flops also include inverter gates.

In Figures 2.1, 2.2 and 2.3 are shown the designs of an half adder, a full adder and D flip flops implemented as previously described.

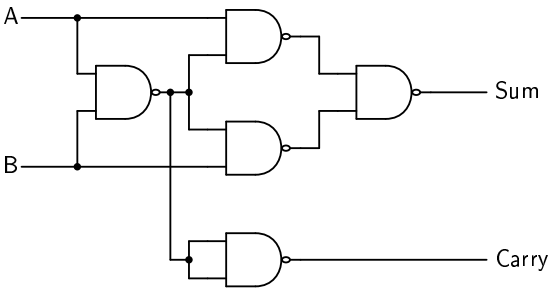



Figure 2.1: Half Adder ND2 implementation

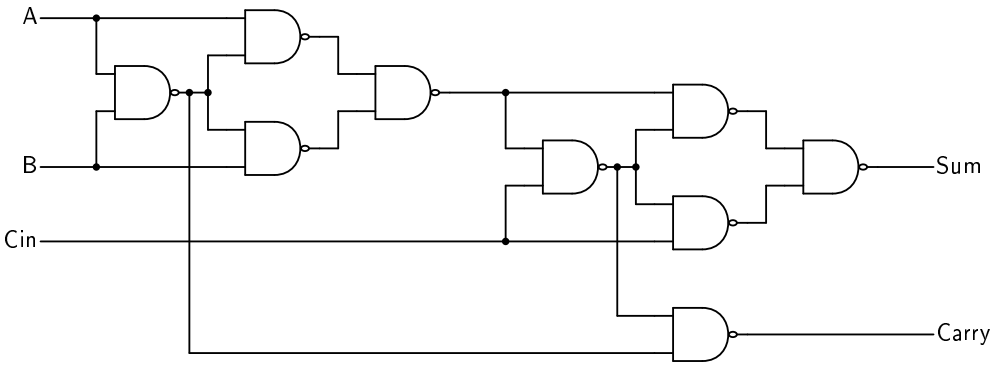



Figure 2.2: Full Adder ND2 implementation

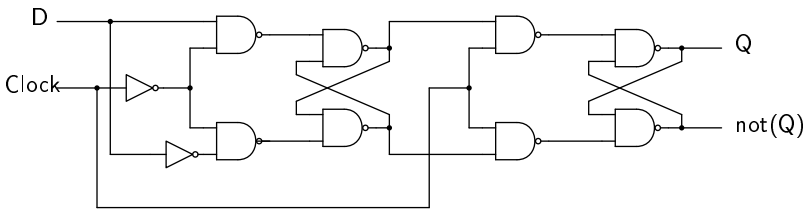



Figure 2.3: D flip flop implementation

The code requires as inputs the ND2 area and the INV one: the total area has been calculated as follows:

$$area_{FA} = 9 \cdot area_{ND2} \tag{2.1}$$

$$area_{HA} = 5 \cdot area_{ND2} \tag{2.2}$$

$$area_{FF} = 8 \cdot area_{ND2} + 2 \cdot area_{INV} \tag{2.3}$$

$$area_{total} = N_{FA} \cdot area_{FA} + N_{HA} \cdot area_{HA} + N_{FF} \cdot area_{FF} \tag{2.4}$$

$N_{FA}$, $N_{HA}$ and $N_{FF}$ are respectively referred to the number of full adders, half adders and flip flops instantiated in the Wallace structure.

### 2.2.3 Power consumption

In this section, in order to estimate the power consumption, both static power and dynamic power contributions have been taken into account.

The dynamic power of a ND2 gate has been computed as follows:

$$P_{dyn}^{ND2} = 0.5 \cdot C_{ND2} \cdot f_{ck} \cdot V_{DD}^2 \tag{2.5}$$

while for INV gate:

$$P_{dyn}^{INV} = 0.5 \cdot C_{INV} \cdot f_{ck} \cdot V_{DD}^2 \tag{2.6}$$

where $f_{ck}$ is given by the technology of reference.

It's possible to compute the dynamic power contribution also considering the $f_{max}$ computed as the reverse of the critical path.

The total dynamic power contribution is obtained as:

$$P_{dyn}^{total} = P_{dyn}^{ND2} \cdot N_{ND2} + P_{dyn}^{INV} \cdot N_{INV} \tag{2.7}$$

For static power evaluation, $I_{static}^{ND2}$ and $I_{static}^{INV}$ are required as input and have the same value. This contribution is usually obtained starting from $I_{gate}^{ND2}$ and $I_{off}^{ND2}$ currents, provided by the Roadmap. For a ND2 gate, the static power consumption is:

$$P_{static}^{ND2} = I_{static}^{ND2} \cdot V_{DD} \tag{2.8}$$

For a INV gate:

$$P_{static}^{INV} = I_{static}^{INV} \cdot V_{DD} \tag{2.9}$$

The total static power contribution is obtained from the following equation:

$$P_{static}^{total} = P_{static}^{ND2} \cdot N_{ND2} + P_{static}^{INV} \cdot N_{INV} \tag{2.10}$$

### 2.2.4 Timing analysis

In order to estimate the maximum operating frequency the critical path has been calculated. In a pipelined version of the Wallace Tree, the critical path to be considered is the longest combinatorial path between 2 layers of flip flops, in particular it is represented by the path which goes through a certain number of full adders (that depends on the number of levels of the tree and the number of pipeline stages inserted).

Since the system is timed through flip flops, the Clock-to-Output delay and set-up time have been also considered. In particular the minimum allowed clock period (in the fully pipelined case) will therefore be:

$$t_{min} = t_{ClkToOut} + delay_{FA} + t_{setup} \tag{2.11}$$

Otherwise, the number of FA between two layers of Flip-Flops has to be considered. The full adder delay is given by the following expression:

$$delay_{FA} = 6 \cdot \tau_{ND2} \tag{2.12}$$

$t_{ClkToOut}$ and $t_{setup}$ are passed as inputs to the function implemented for the estimations. The delay of a single ND2 gate, $\tau_{ND2}$, is calculated by considering the model of delay in the Roadmap, that is the following:

$$\tau_{ND2} = \frac{C_{ND2} \cdot V_{DD}}{I_{ND2}} \tag{2.13}$$

In the Matlab function that has been implemented $\tau_{ND2}$ is passed as an input as well.

Finally the Wallace Tree delay has been obtained as:

$$delay_{total} = N_{pipestage} \cdot t_{min} \tag{2.14}$$

Once obtained the critical path, it is possible to evaluate the maximum operating frequency, $f_{max}$, calculated as the reverse of $t_{min}$.

In the case of a non-pipelined implemetation the delay is given by $N$ times the FA delay, where $N$ is the number of levels of the tree.

## 2.3 MATLAB implementation

The developed Matlab code was written both as a simple script and as a function. The final function-version is the one that has been used to do multiple estimations (by varying n and k). The script reported in Paragraph 1.3.2 is instead the code that has been used to perform these estimations by passing the input values to the cited function.

### 2.3.1 Function

```
function[wallace_area,Pdyn,Pstatic,wallace_delay,Pdyn_max,f_max]=wallace(n, k, ...
    radix, ND2delay, ND2area,Wn, CLND2, fck, Vdd, IstND2,t_setup,t_clkToOut,...
    INVarea,CLINV,ppipe)
% Wallace tree design
% Note: decomment lines 45, 46, 47, 90 to generate FidoCadJ code, decomment
% lines 50, 53, 63, 81, 84, 96 to generate VHDL code.

% Inputs parameters
%n=8;    % number of input bits
%k=8;    % number of operands
%ppipe=2 % every how many levels of the tree a pipe stage is introduced
%radix=1; % radix of the multiplier usign the wallace tree to sum partial products
%ND2area=0.5e-12; % m^2
%Wn=0.27e-6; % m
%ClND2=15.47e-9*Wn; % F 15.47 fF/um
%fck=6.2e9;  % Hz
%Vdd=0.97;   % V
%IstND2=226.67e-9; % A static current
%Ion=1200;   % A/m
ClND2= CLND2*Wn; % 15.47e-9*Wn; % F 15.47 fF/um
%t_setup= 5e-9; % s
%t_clkToOut= 15e-9; %s
ClINV=CLINV*Wn;
IstINV=IstND2;
% INVarea=0.19e-12; % l_inv=l_nand
% ClINV=3e-9*Wn; %F

error='Error radix should be power of 2';
if floor(log2(radix))~= log2(radix)
    disp(error);
    radix=1;
end

% step 1 multioperand adder matrix
mat=zeros(k,n+log2(radix)*k+ceil(log2(k)));
for i=1:k
    mat(i,(size(mat,2)-(n-1+(floor(log2(radix)))*(i-1))):(size(mat,2)-(floor(log2(...
        radix)))*(i-1)))=ones(1,n);   % step 1 multioperand adder matrix
```

```
36 end
37
38 fa=0;   % number of full adder
39 ha=0;   % number of half adder
40 ff=0;   % number of flip-flops
41 n_stage=0;  % number of stage
42
43 fp=fopen('code_fidoCadJ.fcd','w');
44 new_origin=[245 75];    % origin of the FidoCadJ code
45 old_origin=new_origin;
46
47  %%%%%%height=0;%% (VHDL) height of the tree
48  %%%%%%init_wallace_VHDL(n,k,size(mat,2),radix);%%(VHDL) VHDL file initialization
49
50 s_matrix=zeros(1,size(mat,2));  % matrix of s for ha and fa allocation
51  %%%%%%[new_origin,old_origin]=dotNotation(mat,new_origin,s_matrix,old_origin);
52
53 while k>2 % At each level of the tree we divide dots in three-dots-wide stripes ...
       and each stripe is filled with the maximum possible amount of FAs and HAs. ...
       Stripes that are composed of only one or two operands are forwarded to the ...
       next level.
54     n_subm=floor(k/3);  % number of 3 operands groups
55     s_matrix=zeros(n_subm,size(mat,2));
56
57     for t=1:n_subm
58         if t==1 ff_(n_stage+1)=0; end
59         s=sum(mat(3*t-2:3*t,:));    % 3 -> full adder, 2 -> half adder, 1 -> dot ...
               to be moved to the next stage
60
61         %%%%%% gen_FAHA_VHDL(mat,t,fa,ha,height,ppipe,n_stage); %% (VHDL) allocate...
                FAs and HAs
62         fa_level=sum(s>2);
63         ha_level=sum(s>1)-sum(s>2);
64         fa=fa+sum(s>2); % accumulate number of FA
65         ha=ha+sum(s>1)-sum(s>2); % accumulate number of HA
66
67         %ff=ff+2*fa_level+2*ha_level;
68         ff_(n_stage+1)=ff_(n_stage+1)+2*fa_level+2*ha_level; % 2 flip flops for ...
               each adder output (sum and carry)
69         %%%%
70         mat(2*t-1:2*t,:)=0;
71         new_line=s>1;   % matrix line to update
72         mat(2*t-1,:)=new_line;
73         mat(2*t,:)=[new_line(2:end) 0];
74         dotTomv=(s>0)-(s>1);    % dot to move to the next step
75         mat(2*t-1,:)=or(mat(2*t-1,:),dotTomv);
76         %%%%
77         s_matrix(t,:)=s;    % update row of s_matrix with s
78     end
79      %%%%%%height=height+k;%% (VHDL) update the height of the tree
80     n_rowTomv=k-n_subm*3;
81     if n_rowTomv>0
82          %%%%%%mv_row_VHDL(n_rowTomv,height,size(mat,2),k); %% (VHDL) move out of ...
                3 rows groups dots to the next step
83         mat(ceil(k*2/3)-(n_rowTomv-1):ceil(k*2/3),:)=mat(k-(n_rowTomv-1):k,:);
84     end
85     k=ceil(k*2/3);       % number of operands of the next step
86     mat=mat(1:k,:);      % next step dot matrix
87
88     %%%%%%[new_origin,old_origin]=dotNotation(mat,new_origin,s_matrix,old_origin);
89
90     n_stage=n_stage+1;
91
92 end
93
94  %%%%%%end_wallace_VHDL(mat,height); %% (VHDL) end the VHDL file and set the ...
       outputs
95
96   ff=sum(ff_(ppipe:ppipe:n_stage));
97
```

```
98  if ppipe>n_stage
99      ppipe=0;
100 end
101 %%%%% Delay evaluation
102 %ND2delay=Vdd*ClND2/(Ion*Wn);     % s delay of a nand2 gate
103 t_clkToOut=3*ND2delay
104 fa_delay=6*ND2delay;     % fa critical path
105 if ppipe~=0
106     t_min=fa_delay*ppipe+t_setup+t_clkToOut % critical path
107     f_max=1/t_min
108     wallace_delay=floor(n_stage/ppipe)*(t_min)+fa_delay*mod(n_stage,ppipe)
109 else
110     wallace_delay=n_stage*fa_delay % critical path
111     f_max=1/wallace_delay    % maximum frequency
112 end
113
114
115 if ppipe==0
116     ff=0;
117 end
118 %%%%% Area
119 fa_area=9*ND2area;
120 ha_area=5*ND2area;
121 ff_area=8*ND2area+2*INVarea;
122 wallace_area=fa*fa_area+ha*ha_area+ff*ff_area % m^2
123
124 %%%%% Power
125 n_ND2=9*fa+5*ha+8*ff;     % number of ND2
126 PdynND2=0.5*ClND2*fck*Vdd^2;     % W dynamic power of a ND2
127 PdynND2_max=0.5*ClND2*f_max*Vdd^2;% W dynamic power of a ND2 with f_max
128 PstaticND2=IstND2*Vdd;  % W static power of a ND2
129
130 n_INV=2*ff; %number of INV
131 PdynINV=0.5*ClINV*fck*Vdd^2;% W worst case dynamic power of a INV  (switching ...
        activity=1)
132 PdynINV_max=0.5*ClINV*f_max*Vdd^2;% W worst case dynamic power of a INV with f_max...
          (switching activity=1)
133 PstaticINV=IstINV*Vdd;  % static power of a INV
134
135 Pdyn=PdynND2*n_ND2+PdynINV*n_INV % W worst case dynamic power (switching activity...
        =1)
136 Pdyn_max=PdynND2_max*n_ND2+PdynINV_max*n_INV % W worst case dynamic power with ...
        f_max (switching activity=1)
137 Pstatic=PstaticND2*n_ND2+PstaticINV*n_INV  % static power
```

### 2.3.2 Script

```
1  %  Plot wallace_area
2  close all
3  clc
4  clear all
5
6  radix=2;
7  ND2delay=12.5e-12; % s
8  ND2area=0.5e-12; % m^2
9  Wn=0.27e-6; % m
10 CLND2=15.47e-9; % 15.47e-9*Wn; % F 15.47 fF/um
11 fck=5e9;  % Hz
12 Vdd=0.97;   % V
13 IstND2=226.67e-9; % A
14 t_setup= 10e-12;
15 t_clkToOut= 15e-9; %s
16
17
18 INVarea=0.19e-12; % l_inv=l_nand
19 CLINV=3e-9*Wn; %F
20
```

```
% AREA
n=[4:128];
k=[4, 8, 16, 32, 64, 128]; %[4, 8, 16, 32, 64, 128]; %[4:1:64];  %[3:1:8];
ppipe=2;%[0:11]
wallace_area=zeros(length(k),length(n));
Pdyn=zeros(length(k),length(n));
Pstatic=zeros(length(k),length(n));
wallace_delay=zeros(length(k),length(n));
parameters=zeros(length(k),length(n),6);
for i=1:length(k)
    for j=1:length(n)

%        [wallace_area(i,j),Pdyn(i,j),Pstatic(i,j),wallace_delay(i,j), pdyn_max]=...
    wallace(n(j), k(i), radix, ND2delay, ND2area,Wn, CLND2, fck, Vdd, IstND2);
           [parameters(i,j,1),parameters(i,j,2),parameters(i,j,3),parameters(i,j...
               ,4), parameters(i,j,5), parameters(i,j,6)]=wallace(n(j), k(i), ...
               radix, ND2delay, ND2area,Wn, CLND2, fck, Vdd, IstND2,t_setup,...
               t_clkToOut,INVarea,CLINV,ppipe);

           % For different ppipe values comparison
           % for w=1:length(ppipe)
           % [parameters(i,w,1),parameters(i,w,2),parameters(i,w,3),parameters(i,w...
               ,4), parameters(i,w,5), parameters(i,w,6)]=wallace(n, k, radix, ...
               ND2delay, ND2area,Wn, CLND2, fck, Vdd, IstND2,t_setup,t_clkToOut,...
               INVarea,CLINV,ppipe(w));
           % end
    end
end
%pause


%
fclose all;
%%
txt={'Area', 'Dynamic Power', 'Static Power', 'Delay', 'Dynamic Power fmax','...
    F_m_a_x'};
txtpng={"area.png", "p_dyn.png", "p_static.png", "delay.png", "p_dyn_fmax.png","...
    throughput.png"};
txt_z={'Area [m]', 'P_d_y_n [W]', 'P_s_t [W]', 'Delay [s]','P_d_y_n_M_A_X [W]','...
    F_m_a_x [Hz]'};

lg={'k=4', 'k=8', 'k=16', 'k=32', 'k=64', 'k=128'}
for d=1:6
    figure(d)
     hold on
    title(txt(d));
    for i=1:length(k)
     plot(n,parameters(i,:,d));
     hold on
     legend(lg,'Location','NorthWest')
     hold on
    xlabel('n');
    ylabel(txt_z(d));
     grid on
    end
    if (d==1)
           saveas(gcf,'area');
           saveas(gcf,'area.png');
    elseif(d==2)
            saveas(gcf,'p_dyn');
           saveas(gcf,'p_dyn.png');
               elseif(d==3)
            saveas(gcf,'p_static');
           saveas(gcf,'p_static.png');
                           elseif(d==4)
            saveas(gcf,'delay');
           saveas(gcf,'delay.png');
                                    elseif(d==5)
            saveas(gcf,'p_dyn_fmax');
           saveas(gcf,'p_dyn_fmax.png');
```

```
                                                    elseif(d==6)
             saveas(gcf,'fmax');
            saveas(gcf,'fmax.png');
    end
    hold off
end
%pause
% %************************************************************************
% PRINT 3D Graphs
% %************************************************************************

txt_z={'Area [m]', 'P_d_y_n [W]', 'P_s_t [W]', 'Delay [s]','P_d_y_n_M_A_X [W]','...
    F_m_a_x [Hz]'};

% figure (6)
% subplot(2,2,1)
for i=1:6
    figure
    mesh(n, k, parameters(1:length(k),:,i));

    xlabel('n');
    ylabel('k');
    zlabel(txt_z(i));
    title(txt(i));
    if (i==1)
           saveas(gcf,'area3d');
           saveas(gcf,'area3d.png');
    elseif(i==2)
            saveas(gcf,'p_dyn3d');
           saveas(gcf,'p_dyn3d.png');
    elseif(i==3)
            saveas(gcf,'p_static3d');
           saveas(gcf,'p_static3d.png');
    elseif(i==4)
            saveas(gcf,'delay3d');
           saveas(gcf,'delay3d.png');
    elseif(i==5)
            saveas(gcf,'p_dyn_fmax3d');
           saveas(gcf,'p_dyn_fmax3d.png');
    elseif(i==5)
            saveas(gcf,'fmax3d');
           saveas(gcf,'fmax3d.png');
    end

end


hold off

%% For different ppipe values comparison
fclose all;
close all
txt={'Area', 'Dynamic Power', 'Static Power', 'Delay', 'Dynamic Power fmax','...
    F_m_a_x'};
txt_y={'Area [m]', 'P_d_y_n [W]', 'P_s_t [W]', 'Delay [s]','P_d_y_n_M_A_X [W]','...
    F_m_a_x [Hz]'};

for d=1:6
    figure(d)
     hold on
    title(txt(d));
    for i=1:length(ppipe)
     plot(ppipe,parameters(1,:,d))
     hold on
     xlabel('ppipe')
     ylabel(txt_y(d))
     grid on
    end
    hold off
        if (d==1)
```

```
148             saveas(gcf,'area_pipe');
149             saveas(gcf,'area_pipe.png');
150         elseif(d==2)
151             saveas(gcf,'p_dyn_pipe');
152             saveas(gcf,'p_dyn_pipe.png');
153         elseif(d==3)
154             saveas(gcf,'p_static_pipe');
155             saveas(gcf,'p_static_pipe.png');
156         elseif(d==4)
157             saveas(gcf,'delay_pipe');
158             saveas(gcf,'delay_pipe.png');
159         elseif(d==5)
160             saveas(gcf,'p_dyn_fmax_pipe');
161             saveas(gcf,'p_dyn_fmax_pipe.png');
162         elseif(d==6)
163             saveas(gcf,'fmax_pipe');
164             saveas(gcf,'fmax_pipe.png');
165     end
166 end
```

### 2.3.3 Automatic Generation of FidoCadJ drawing

In addition, it was used a *MATLAB* function to print the Wallace tree drawing. FidoCadJ was used as vector graphic editor to draw the dot notation version of the tree.

```
1  function [new_origin,old_origin] = dotNotation(mat,new_origin,s_matrix,old_origin)
2  % DOTNOTATION returns a FidoCadJ code to show the tree using dot notation.
3  % FAs are represented as solid 3 dots blocks and HAs are represented as dashed 2 ...
       dots blocks.
4  %   DOTNOTATION(mat,new_origin,s_matrix,old_origin) inputs:
5  %           mat         => current dots matrix
6  %           new_origin  => next step starting point for the FidoCadJ drawing
7  %           s_matrix    => matrix of s vectors (3 -> full adder, 2 -> half adder, ...
       1 -> dot to be moved to the next stage)
8  %           old_origin  => current step starting point for the FidoCadJ drawing
9
10 [n_row, n_column]=size(mat);
11 fp=fopen('code_fidoCadJ.fcd','a');
12
13 for i=1:size(s_matrix,1)
14     for j=1:size(s_matrix,2)
15         if s_matrix(i,size(s_matrix,2)-j+1)>1   %to detect full adder and half ...
               adder and to put the blocks
16             fprintf(fp,'RV ');  %FidoCadJ code
17             fprintf(fp,'%d ',old_origin(1)-10*(j-1)-3);
18             fprintf(fp,'%d ',(old_origin(2)+30*(i-1)-5));
19             fprintf(fp,'%d ',old_origin(1)-10*(j-1)+3);
20             fprintf(fp,'%d ',(old_origin(2)+30*(i-1)-5+30));
21             fprintf(fp,'%d\n', ones);
22             if s_matrix(i,size(s_matrix,2)-j+1)==2  %if it's a half adder->dot ...
                   line for block
23                 fprintf(fp,'FCJ 2 0\n');
24             end
25         end
26     end
27 end
28
29 old_origin=new_origin;
30
31 for i=1:n_row
32     for j=1:n_column
33         if mat(i,n_column-j+1)>0
34             fprintf(fp,'SA ');  %generation of the new dot matrix
35             fprintf(fp,'%d ', new_origin(1)-10*(j-1));
36             fprintf(fp,'%d ', (new_origin(2)+10*(i-1)));
37             fprintf(fp,'%d\n', zeros);
38         end
39     end
```

```
end
fprintf(fp,'LI ');  %line between two steps
fprintf(fp,'%d ', new_origin(1)-10*(n_column-1));
fprintf(fp,'%d ', (new_origin(2)+10*n_row));
fprintf(fp,'%d ', new_origin(1)+5);
fprintf(fp,'%d ', (new_origin(2)+10*n_row));
fprintf(fp,'%d\n', zeros);


fclose(fp);
new_origin(2)=new_origin(2)+10*(n_row+1);

end
```

### 2.3.4 Automatic Generation of VHDL code

An automatic VHDL implementation was carried out to test the algorithm. In this section the MATLAB functions called by the main script are reported.

```
function init_wallace_VHDL(n,k,length,radix)
%INIT_WALLACE_VHDL Initialize VHDL description of the wallace tree.
%    INIT_WALLACE_VHDL(n,k,lenght,radix) inputs:
%           n      => number of bits per operand
%           k      => number of operands
%           length => number of column of the dot matrix
%           radix  => radix of the multiplier using the wallace tree to add
%                     partial products

k_init=k;   % number of operands
fp=fopen('wallace.vhd','w');    % open VHDL file

% Library and entity initilization
fprintf(fp,['library IEEE;\nuse IEEE.std_logic_1164.ALL;\n\n',...
    'entity wallace is\n\tport(\n\t\t']);
for i=1:k-1
    fprintf(fp,'in%d,',i-1);    % inputs printing
end

% Entity body and architecture definition
fprintf(fp,['in%d:IN std_logic_vector (%d downto 0);\nclock,reset: IN std_logic;\n...
    \n\n\t);\n',...
    'end wallace;\n\n',...
    'architecture behavioural of wallace is\n'...
    'component HAff is\n\t port(\n\t\tx,y,clock,reset:IN std_logic;\n\t\ts,c:OUT ...
        std_logic\n\t);\nend component;\n',...   % Half Adder component definition
    'component FAff is\n\tport(\n\t\tcin,x,y,clock,reset:IN std_logic;\n\t\ts,cout...
        :OUT std_logic\n\t);\nend component;\n'...
    'component HA is\n\t port(\n\t\tx,y:IN std_logic;\n\t\ts,c:OUT std_logic\n\t)...
        ;\nend component;\n',...   % Half Adder component definition
    'component FA is\n\tport(\n\t\tcin,x,y:IN std_logic;\n\t\ts,cout:OUT std_logic...
        \n\t);\nend component;\n\n'],k-1,n-1);    % Full Adder component ...
        definition

height=0; % height of the final tree. This is needed to define the internal ...
    signals used to connect inputs, outputs, HAs and FAs
while k>2
    height=height+k;
    k=ceil(k*2/3);
end

% Array type definition: n rows = height of the tree, n columns= lenght of
% the 'mat' matrix
fprintf(fp,['type bidimensional is array (0 to %d) of std_logic_vector(%d downto ...
    0);\n',...
     'signal p:bidimensional;\nbegin\n'],height+1,length-1);

% Input assignment to internal signals
for i=1:k_init
```

```
42      fprintf(fp,'\tp(%d)(%d downto %d)<=in%d;\n',i-1,n-1+floor(log2(radix))*(i-1),...
            floor(log2(radix))*(i-1),i-1);
43  end
44
45  end
```

```
1   function gen_FAHA_VHDL(mat,t,fa,ha,height,ppipe,n_stage)
2   %GEN_FAHA_VHDL implements Half Adders and Full Adders at each step of the
3   %algorithm.
4   %   GEN_FAHA_VHDL(mat,t,VHDL_file,fa,ha,height) inputs:
5   %           mat         => current dots matrix
6   %           t           => index of current 3 rows group to be processed
7   %           fa          => current FAs count
8   %           ha          => current HAs count
9   %           ppipe       => every how many levels of the tree a pipe stage is ...
        introduced
10  %           n_stace     => current stage
11
12  VHDL_file=fopen('wallace.vhd','a'); % append text to 'wallace.vhd'
13  if mod(n_stage+1,ppipe)==0
14  for j=1:size(mat,2)
15          l=find(mat((3*t-2):(3*t),(size(mat,2)-(j-1))));    % search for '1' ...
                position in s columns starting from the north east of s
16          if length(l)==3  % allocate FA
17              fprintf(VHDL_file,['FA_ff_%d:FAff port map',...
18                  '(x=>p(%d)(%d),y=>p(%d)(%d),Cin=>p(%d)(%d),clock=>clock,reset=>...
                        reset,s=>p(%d)(%d),cout=>p(%d)(%d));\n'],fa,3*t-3+height,j-1,3*...
                        t-2+height,j-1,3*t-1+height,j-1,2*t-2+size(mat,1)+height,j-1,2*...
                        t-1+size(mat,1)+height,j);
19              fa=fa+1; % increment FA count
20          elseif length(l)==2 % allocate HA
21                  fprintf(VHDL_file,['HA_ff_%d:HAff port map',...
22                  '(x=>p(%d)(%d),y=>p(%d)(%d),clock=>clock,reset=>reset,s=>p(%d)(%d),...
                        c=>p(%d)(%d));\n'],ha,3*t-4+l(1)+height,j-1,3*t-4+l(2)+height,j...
                        -1,2*t-2+size(mat,1)+height,j-1,2*t-1+size(mat,1)+height,j);
23                  ha=ha+1; % increment HA count
24          elseif length(l)==1 % move dot to the next stage
25                      fprintf(VHDL_file,'p(%d)(%d)<=p(%d)(%d);\n',2*t-2+size(mat,1)+...
                            height,j-1,3*t-4+l(1)+height,j-1);
26          end
27  end
28  else
29  for j=1:size(mat,2)
30          l=find(mat((3*t-2):(3*t),(size(mat,2)-(j-1))));    % search for '1' ...
                position in s columns starting from the north east of s
31          if length(l)==3  % allocate FA
32              fprintf(VHDL_file,['FA_%d:FA port map',...
33                  '(x=>p(%d)(%d),y=>p(%d)(%d),Cin=>p(%d)(%d),s=>p(%d)(%d),cout=>p(%d)...
                        (%d));\n'],fa,3*t-3+height,j-1,3*t-2+height,j-1,3*t-1+height,j...
                        -1,2*t-2+size(mat,1)+height,j-1,2*t-1+size(mat,1)+height,j);
34              fa=fa+1; % increment FA count
35          elseif length(l)==2 % allocate HA
36                  fprintf(VHDL_file,['HA_%d:HA port map',...
37                  '(x=>p(%d)(%d),y=>p(%d)(%d),s=>p(%d)(%d),c=>p(%d)(%d));\n'],ha,3*t...
                        -4+l(1)+height,j-1,3*t-4+l(2)+height,j-1,2*t-2+size(mat,1)+...
                        height,j-1,2*t-1+size(mat,1)+height,j);
38                  ha=ha+1; % increment HA count
39          elseif length(l)==1 % move dot to the next stage
40                      fprintf(VHDL_file,'p(%d)(%d)<=p(%d)(%d);\n',2*t-2+size(mat,1)+...
                            height,j-1,3*t-4+l(1)+height,j-1);
41          end
42  end
43
44  end
45
46  end
```

```
function mv_row_VHDL(n_rowTomv,height,length,k)
%MV_ROW_VHDL move rows of dots to the next step of the wallace algorithm
%   MV_ROW_VHDL(n_rowTomv,VHDL_file,height,length,k) inputs:
%           n_rowTomv   => numbers of dot rows to be moved to the next
%                          step
%           height      => current height of the tree
%           length      => number of column of the dot matrix
%           k           => current number of operands already processed

VHDL_file=fopen('wallace.vhd','a'); % append text to 'wallace.vhd'

% Move dots to the next step of the algorithm
for i=1:n_rowTomv
    for j=1:length
        fprintf(VHDL_file,'p(%d)(%d)<=p(%d)(%d);\n',ceil(k*2/3)+height-n_rowTomv+(...
            i-1),j-1,height-n_rowTomv+(i-1),j-1);
    end
end

end
```

```
function end_wallace_VHDL(mat,height)
%END_WALLACE_VHDL set outputs of the VHDL wallace tree implementation
%   END_WALLACE_VHDL(mat,height,VHDL_file) inputs:
%           mat         => final 2 rows dots matrix
%           height      => final height of the tree (without the last 2 rows)

VHDL_file=fopen('wallace.vhd','a'); % append text to 'wallace.vhd'

index_add1=0;   % bit index of the first output to be added
index_add2=0;   % bit index of the second output to be added
index_res=0;    % bit index of the available part of the addition result
flag=0; % if '1' indicates the starting point for the final 2 rows adder

for j=1:size(mat,2)
    l=find(mat(:,(size(mat,2)-(j-1))));
    if and(length(l)==1,flag==0)    % set bits of the available result
            fprintf(VHDL_file,'res(%d)<=p(%d)(%d);\n',index_res,height,j-1);
            index_res=index_res+1;
    elseif length(l)==2    % set bits to be added by the external 2 row adder
            flag=1; % data to the external 2 row adder
            fprintf(VHDL_file,'add1(%d)<=p(%d)(%d);\n',index_add1,height,j-1);
            fprintf(VHDL_file,'add2(%d)<=p(%d)(%d);\n',index_add2,height+1,j-1);
            index_add1=index_add1+1;    % increment bit index of the first addend
            index_add2=index_add2+1;    % increment bit index of the second addend
    elseif and(length(l)==1,flag==1)   % set bit to be added by the external 2 row...
         adder
                if l==1 % set bit to addend 1
                    fprintf(VHDL_file,'add%d(%d)<=p(%d)(%d);\n',1,index_add1,...
                        height,j-1);
                    index_add1=index_add1+1;
                else    % set bit to addend 2
                    fprintf(VHDL_file,'add%d(%d)<=p(%d)(%d);\n',2,index_add2,...
                        height+1,j-1);
                    index_add2=index_add2+1;
                end
    end
end

% Write end of the VHDL file
fprintf(VHDL_file,'end architecture behavioural;\n');
fclose(VHDL_file);

% Define the output of the VHDL implementation
% Read VHDL file in A
fp=fopen('wallace.vhd','r');
i=1;
tline=fgetl(fp);
```

```
45 A{i}=tline;
46 while ischar(tline)
47     i=i+1;
48     tline=fgetl(fp);
49     A{i}=tline;
50 end
51 fclose(fp);
52
53 % Update A with outputs
54 A{8}=sprintf('\t\tadd1:OUT std_logic_vector(%d downto 0);',index_add1-1);
55 A{9}=sprintf('\t\tadd2:OUT std_logic_vector(%d downto 0);',index_add2-1);
56 A{10}=sprintf('\t\tres:OUT std_logic_vector(%d downto 0));',index_res-1);
57
58 % Write A to the VHDL file
59 fp=fopen('wallace.vhd','w');
60 for i = 1:numel(A)
61         if A{i+1} == -1
62              fprintf(fp,'%s', A{i});
63              break
64         else
65              fprintf(fp,'%s\n', A{i});
66         end
67 end
68 end
```

## 2.4 Results

In this section the area, delay and power consumption for both a 50%-pipelined and a non-pipelined version of the Wallace tree have been reported. The comparison has been performed for different numbers of bits and for different numbers of input operands and refers to a radix 2 multiplier implementation, where the partial product terms to be added are placed as in Figure 2.4.

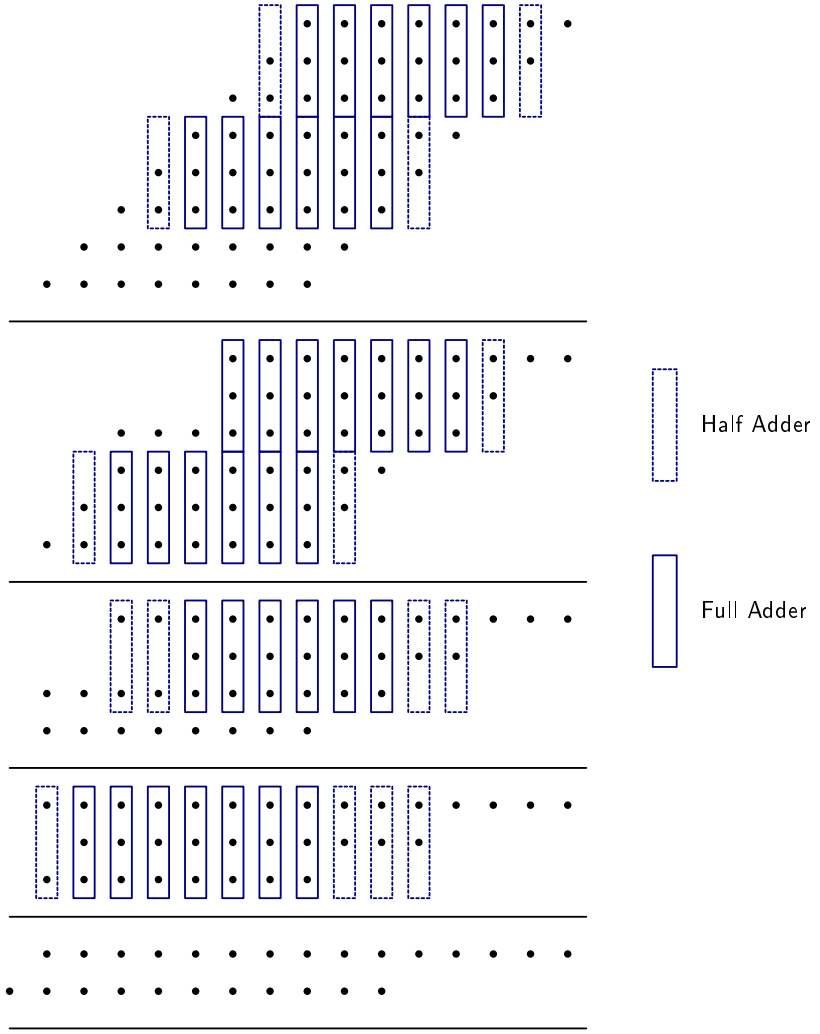
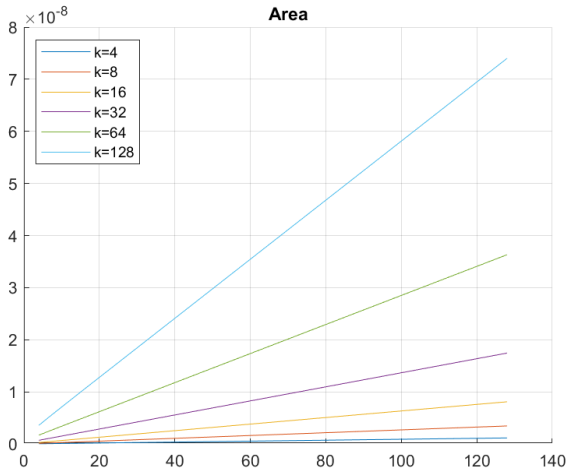
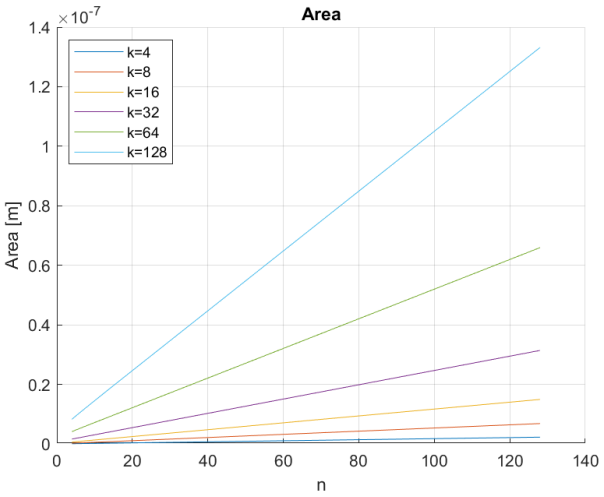



Figure 2.4: Wallace Tree with $n = 8$, $k = 8$

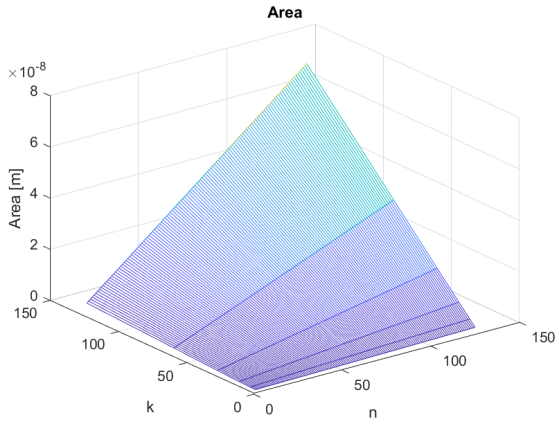
### 2.4.1 Area



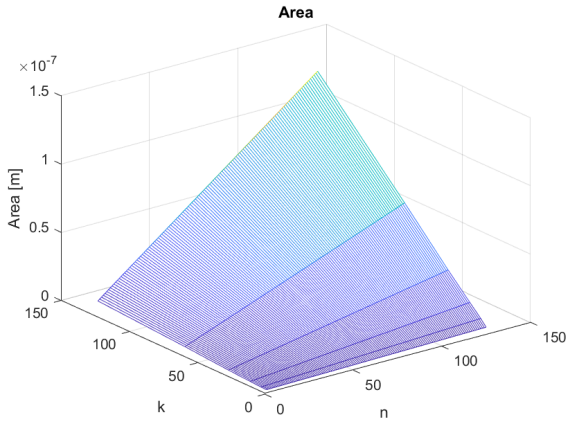

(a) Non-pipelined version




(b) 50%-Pipelined version




(c) Non-pipelined version




(d) 50%-Pipelined version

Figure 2.5: Area

As expected, the total area of the tree linearly increases with the number of bits (n) per input operand (Figures 2.5). The higher the number of operands (k) is, the larger is the total amount of the occupied area because of the higher number of stages required for the calculation.

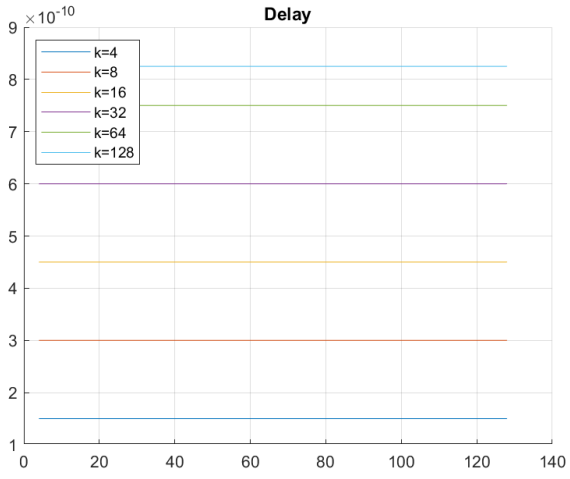
In particular in order to appreciate the differences between the two versions of the tree, some meaningful points have been reported in the following table.

| | | Occupied area [m^2] | |
|---|---|---|---|
| **n** | **k** | **Non-pipelined** | **50% pipelined** |
| **4** | **4** | 3.4e-11 | 6.504e-11 |
| **4** | **128** | 2.499e-9 | 8.278e-9 |
| **128** | **4** | 1.15e-9 | 2.267e-9 |
| **128** | **128** | 7.28e-8 | 1.332e-7 |

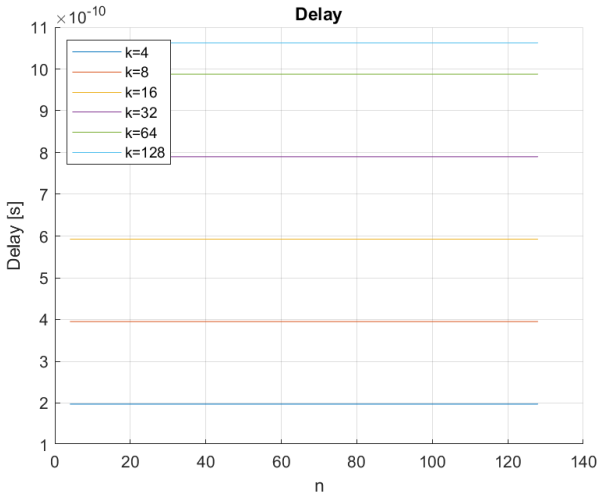
As expected, the occupied area increases in the pipelind case since different levels of Flip flops are inserted; this increase becomes more and more significant as the number of bits and operands increase.

### 2.4.2 Delay

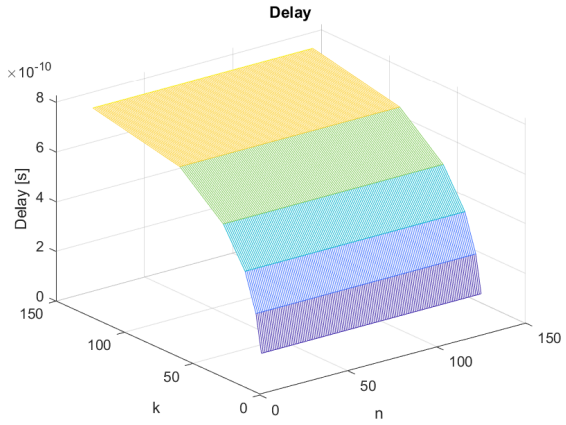
The delay is the time interval necessary to complete the computation through the whole tree. In the non-pipelined case the path is entirely combinatorial while in the pipelined version the critical path is splitted into shorter homogeneous paths. In this last case the delay is therefore computed as the product between the clock period (the critical path delay) and the number of the pipeline stages inserted.



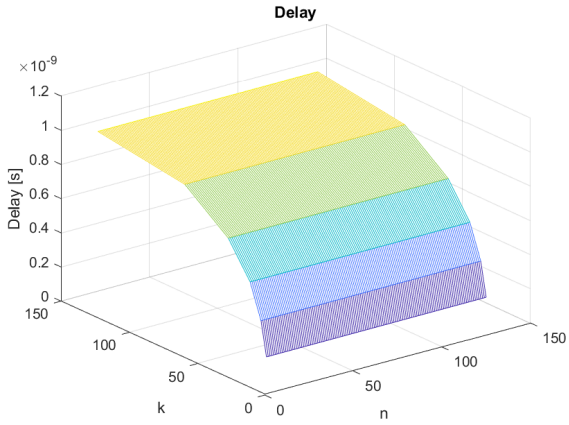

(a) Non-pipelined version




(b) 50%-Pipelined version




(c) Non-pipelined version




(d) 50%-Pipelined version

Figure 2.6: Delay

As shown in Figure 2.6, the amount of delay increases with the number of input operands, instead it is constant with respect to the number of bits. Since in each stage of the tree, full adders and half adders work in parallel, an higher input parallelism does not affect the total delay.

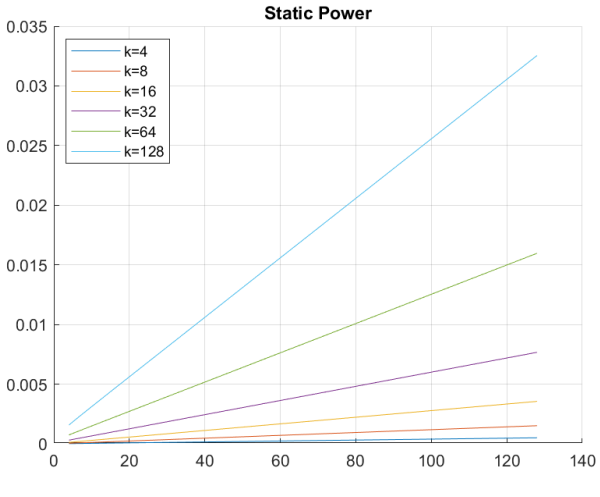
Once again, in order to appreciate the differences between the two versions of the tree, the delay has been computed for a growing number of operands.

| | Delay [ns] | |
|---|---|---|
| **k** | **Non-pipelined** | **50% pipelined** |
| **4** | 0.15 | 0.198 |
| **8** | 0.3 | 0.395 |
| **16** | 0.45 | 0.593 |
| **32** | 0.6 | 0.79 |
| **64** | 0.75 | 0.988 |
| **128** | 0.825 | 1.063 |

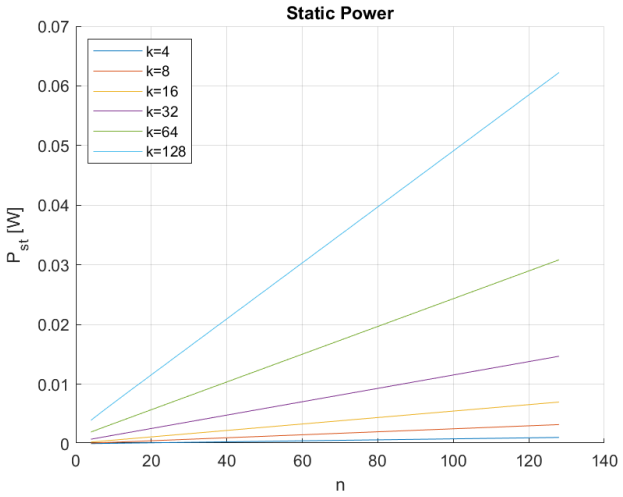
As expected, it is possible to notice how the delay increases in the pipelined case because of the overhead introduced by the registers stages. In particular, as k increases, also the number of levels of the tree increases and so more stages of flip flops are introduced.

### 2.4.3 Static power

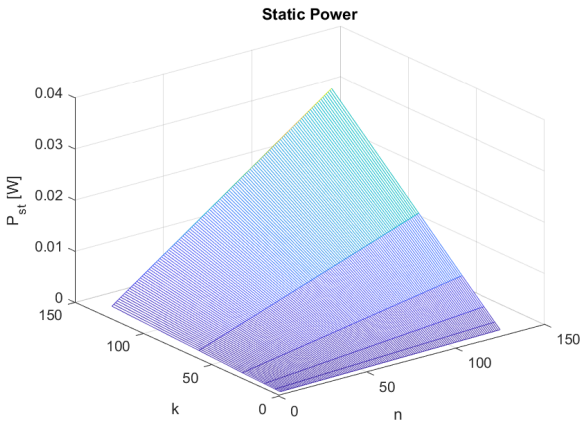
Static power increases with both the number of bits and the number of operands. This is trivial to understand why both of them depend on the total equivalent capacitance of the tree.



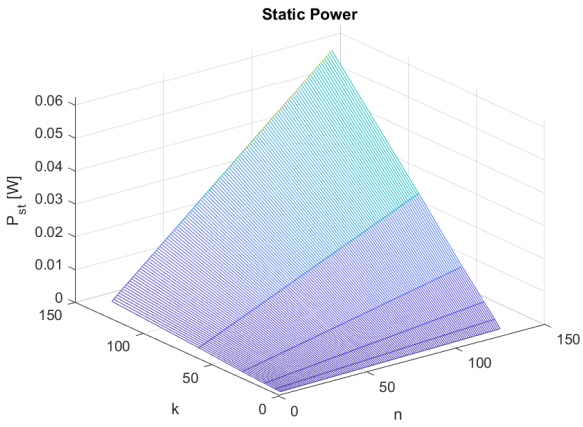

(a) Non-pipelined version




(b) 50%-Pipelined version




(c) Non-pipelined version




(d) 50%-Pipelined version

Figure 2.7: Static power

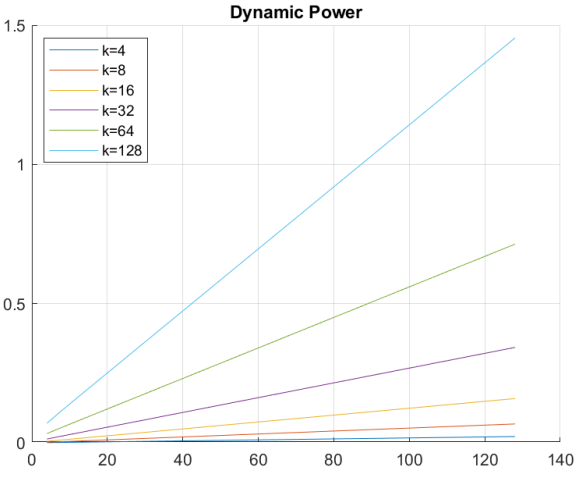
Some quantitative evaluations are reported in the table below.

| | | Static power [W] | |
|---|---|---|---|
| **n** | **k** | **Non-pipelined** | **50% pipelined** |
| **4** | **4** | 1.495e-5 | 3.078e-5 |
| **4** | **128** | 1.1e-3 | 3.933e-3 |
| **128** | **4** | 5.057e-4 | 1.067e-3 |
| **128** | **128** | 3.2e-2 | 6.224e-2 |

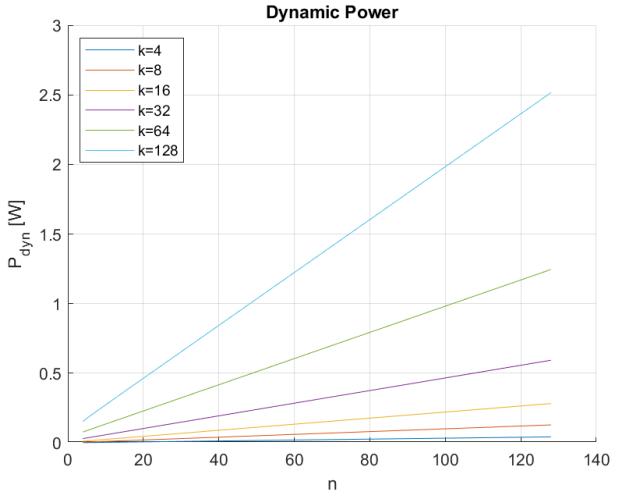
The pipelined version, again, is characterized by a larger static power consumption with respect to the non pipelined version since also the power consumption of the flip flops has been taken into account.

### 2.4.4 Dynamic power

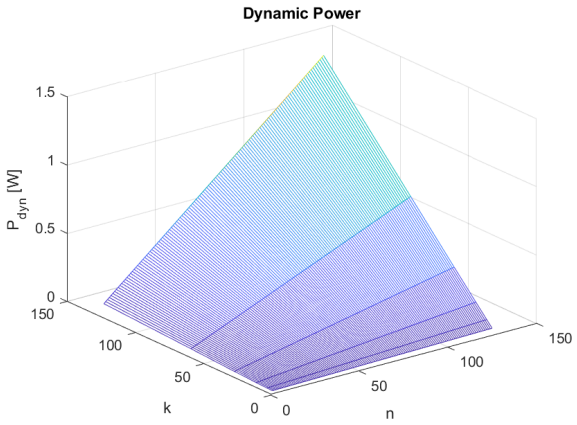
In order to make a proper comparison, dynamic power estimation has been performed for both versions of the tree considering an operating frequency of 5GHz.



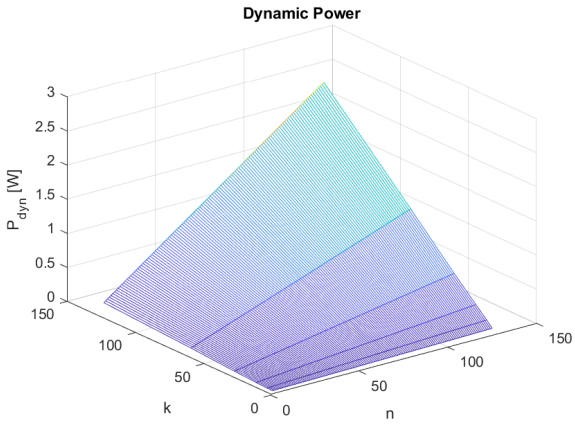

(a) Non-pipelined version




(b) 50%-Pipelined version




(c) Non-pipelined version




(d) 50%-Pipelined version

Figure 2.8: Dynamic power ($f_{ck} = 5GHz$)

In order to appreciate the differences between the two versions of the tree, some meaningful points have been reported in the following table.

| | | Dynamic power [W] | |
|---|---|---|---|
| **n** | **k** | **Non-pipelined** | **50% pipelined** |
| **4** | **4** | 6.681e-4 | 1.218e-3 |
| **4** | **128** | 0.049 | 0.155 |
| **128** | **4** | 0.023 | 0.043 |
| **128** | **128** | 1.431 | 2.516 |

Also in this case the power consumption grows with the number of bits-per-operand (n) and with the number of operands (k). In particular if n increases, the number of flip flops per level also increases; if k increases, the number of the stages of the tree and therefore the number of pipeline stages to be inserted increase. For this reason the pipelined version consumes more than the non-pipelined one.

Besides performing power estimations considering an operating frequency selected by the user, the developed script also is able to compute the maximum operating frequency depending on the critical path. It is possible to use then this value to obtain the dissipated power amount in the case of working at maximum performance. In particular, considering the technological parameters involved, the system is able to work up to a $f_{ckMax}$ of about 5.06 GHz.

### 2.4.5 Different pipeline percentages comparison

The developed Matlab code also uses a parameter called *ppipe*: it is linked to the percentage of pipelining to be applied at the tree, in particular it indicates every how many levels of the tree a pipe stage is introduced. For example:

- if $ppipe = 0$ no pipeline stages are introduced (non-pipelined version);
- if $ppipe = 1$ a pipeline stage is inserted at every level of the tree (fully pipelined version, 100%);
- if $ppipe = 2$ a pipeline stage is inserted every 2 levels of the tree (50%).

In Figure 2.9 occupied area, delay, static and dynamic power estimations are shown in the case of n=128, k=128 and a varying value of ppipe. It is possible to observe that the plots are similar. When ppipe=0 no flip flops are inserted in the design, this corresponds to the minimum area and power. When ppipe=1 the design is fully pipelined and therefore here lies the maximum for both area and power. As the pipeline percentage decreases (ppipe increases) the plots approximately follow the trend of a decreasing exponential.

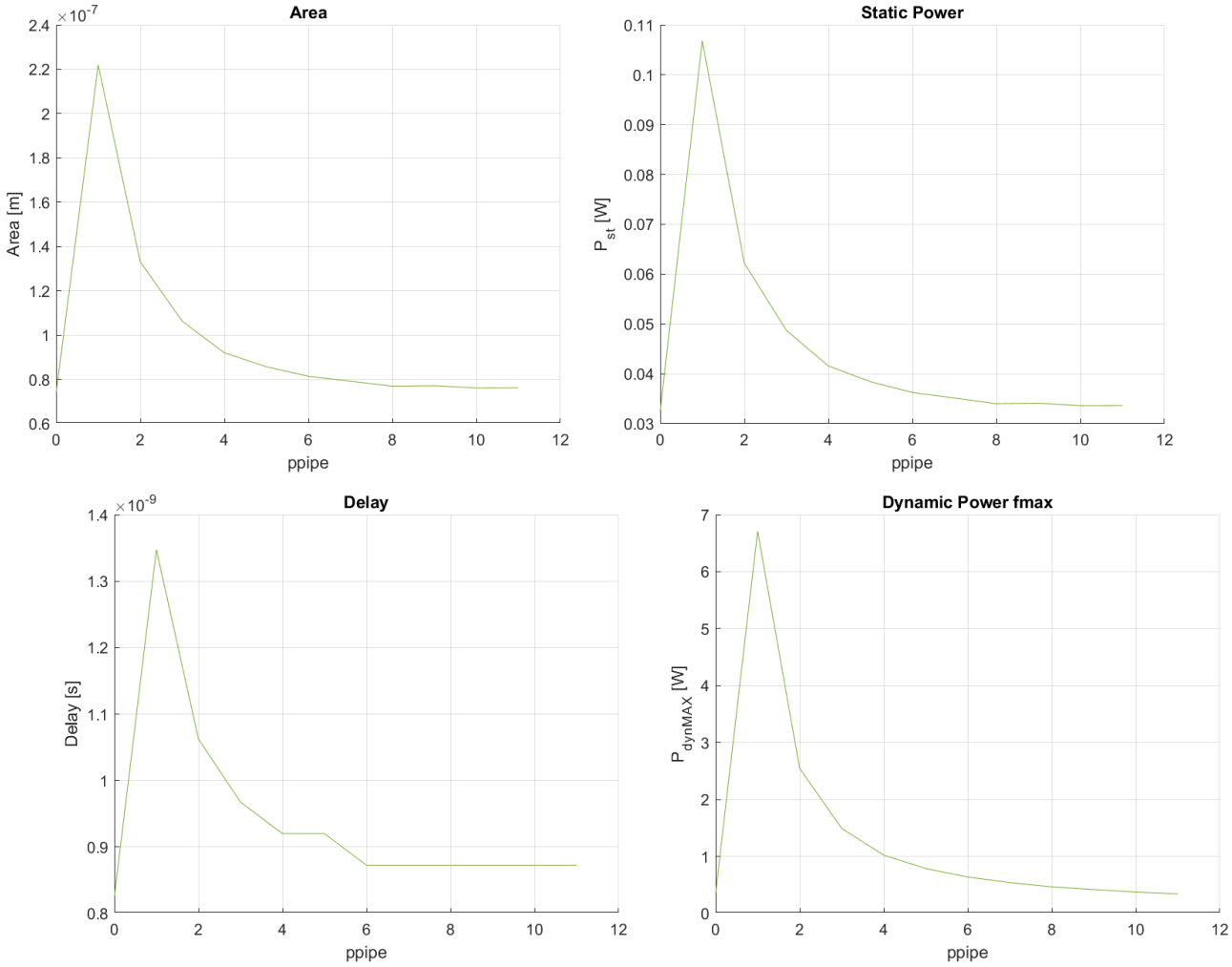



Figure 2.9: Comparisons

Considering the delay, in particular it is possible to notice how the behaviour does not strictly follow the one of the other 3 plots presented. The trend in this case is linked to the number of operands which determine the number of the levels of the tree. In this case since for 128 operands the number of levels is 11, starting from ppipe=6 the trend is perfectly flat: only one pipe level is inserted for ppipe>6 therefore the total delay of the tree is the same. Something similar happens for ppipe=4 and ppipe=5 but 2 register levels are added in this case. It is therefore possible to notice how the optimal pipe percentage to be inserted in the tree depends on the number of operands.

### Throughput

In Figure 2.10 is reported the throughput taking into account different pipeline depths cases.

The trend is similar to the previous cases: the fully pipelined case (ppipe=1) is characterized by the highest possible throughput since the critical path is the one of a single FA. This allows to have multiple elaborations in execution at the same time (at different stages of the pipelined structure). With an increasing number of pipeline stages, latency (in terms of clock cycles needed to complete an elaboration) increases as well.

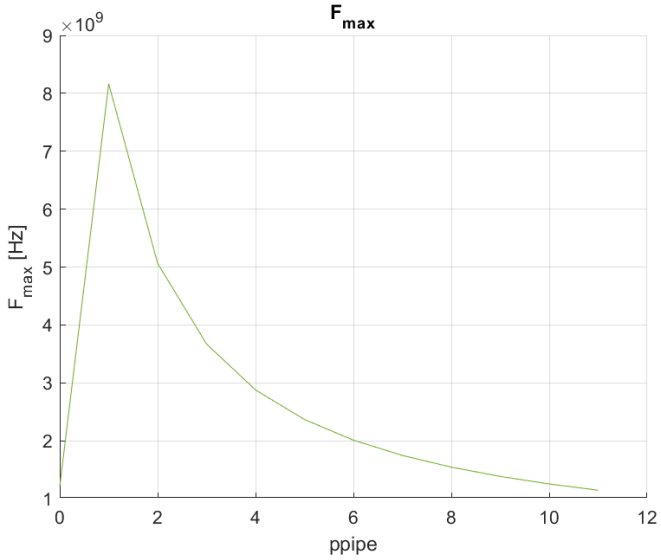



Figure 2.10: Maximum throughput for different pipeline depths

### 2.4.6 VHDL

The developed script also generates the VHDL description of the tree. The output design has been then verified through simulation. In Fig 2.11 is shown the block diagram schematic for a simple tree with n=4, k=4 and ppipe=2.

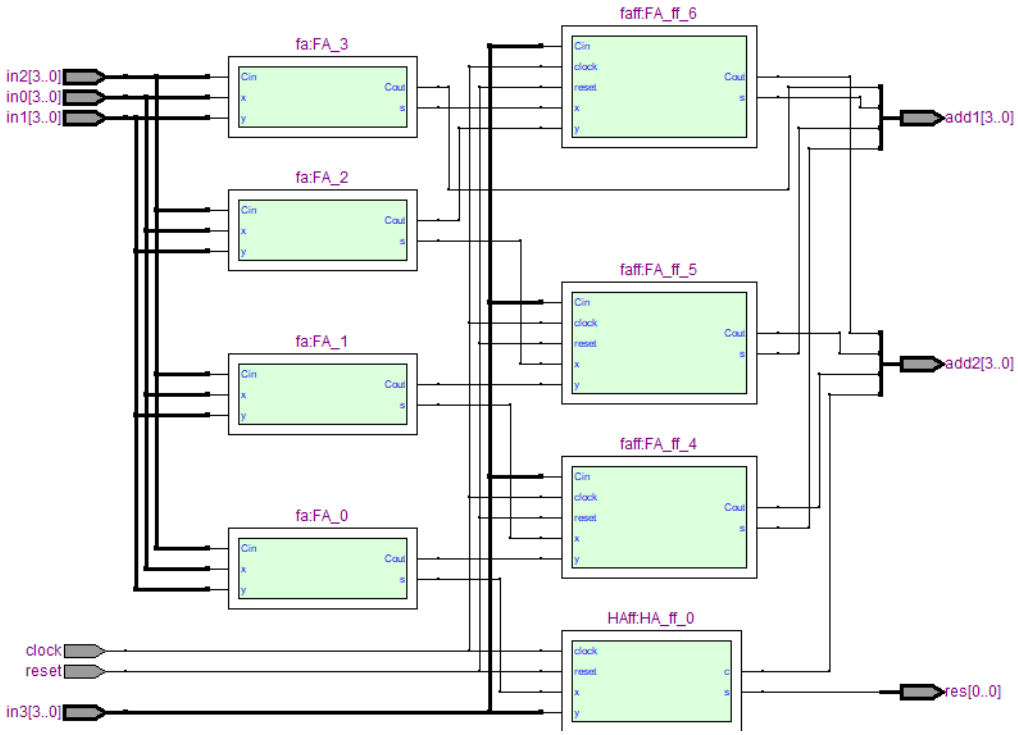



Figure 2.11: 4x4 Wallace tree design (ppipe=2)

The flip flop layer is placed after the second level of the tree, it has been included in the components $HA\_ff$ and $FA\_ff$.

## 2.5 Conclusions

It was developed a model of a Wallace tree with a percentage of pipelining which can be defined by the user. The present model also allows to not insert any pipeline stage. The main advantage of a pipelined implemantation lies in the fact that throughput increases while the time required to complete a single computation increases. Since flip flop insertion causes an increase of both area and power, it is therefore important to carefully choose how many pipeline stages should be inserted in the tree. The choice depends on the application, the optimal choice for the Wallace tree does not corresponds to a fully pipelined approach since it maximizes occupied area, consumptions and also latency.

# Chapter 3

# A Dynamic CMOS Survey

Along with the work of the two adder designs, a brief survey about dynamic logic has been made. The main issue is to find whether there are new dynamic logic technologies that can be used in adders, instead of the standard static MOS structures. The main characteristics of dynamic logic are introduced and discussed, in order to understand the positive and negative aspects of this design technique. For further general information [2] can be seen.

## 3.1 General aspects

Dynamic logic is an alternative design approach of Pseudo NMOS and static CMOS and presents several advantages similar to these ones. Figure 3.1 shows the general schematic of a dynamic logic topology. In this case a pull down network is used, just like the static CMOS logic, but the dual pull up network can be used instead.

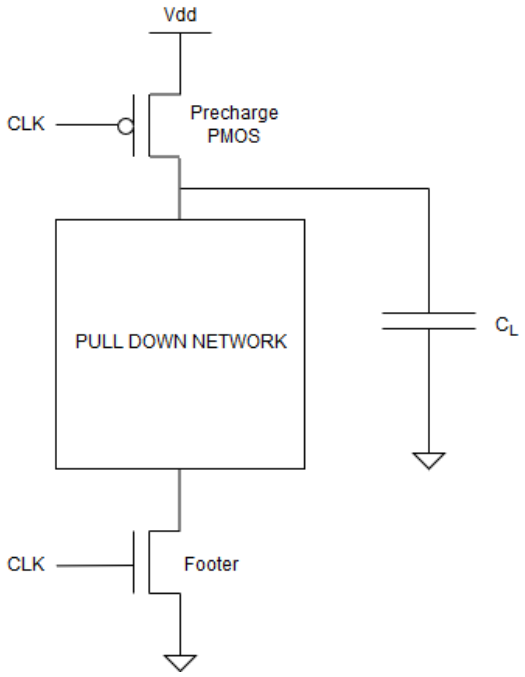



Figure 3.1: Dynamic logic general schematic.

The first positive feature is the lower area occupation: the number of transistors increases linearly with the number of inputs. A generic dynamic N port gate has N+2 transistors, which is close to the pseudo NMOS design, and less than the static CMOS logic, which is 2N. The first N transistors

are needed to implement the Pull Down (or Up) Network, while a PMOS and NMOS are used for the clock signal. The clock signal is needed to define the two working phases of the circuit, the pre-charge and the evaluation phases.

Another important aspect is the absence of static power consumption, unlike pseudo NMOS but similar to the static CMOS logic. The static power is the generated power due to short path between supply voltage and ground.

Furthermore, the circuit presents a lower fan-in with respect to the static CMOS implementation, and this aspect leads dynamic logic to better performance. These features together makes dynamic logic suitable for high performance and low cost implementation.

The dynamic logic presents several drawbacks, though. The need of a clock signal affects negatively power consumption, since at least two transistors are charged and discharged in each clock cycle. Power consumption is also affected by an activity factor $\alpha_{1\to 0}$ higher than the static CMOS. For example, if all the inputs of a NOR gate have a uniform statistic distribution, the activity factor of the dynamic logic and static logic implementations are, respectively the (3.1) and (3.2):

$$\alpha_{dynamic} = 75\% \tag{3.1}$$

$$\alpha_{static} = 18,75\% \tag{3.2}$$

The dynamic power consumption is given by the (3.3):

$$P_{dyn} = \alpha \cdot C_L V_{dd}^2 f_{clk} \tag{3.3}$$

which is proportional to the activity factor. Leakage currents are to be considered also, since they can discharge the output node, and determine the minimum period between two pre-charge phases, which is the minimum clock frequency accepted. This value is about some KHz. Leakage currents tend to discharge the output node, since this one is an high impedance node. This problem can be resolved by using a *bleeder* transistor [2].

Another aspect to be considered is the charge sharing with other output and with the clock signal (*clock feedthrough*) which affects dramatically reliability. The latter can forward bias the drain-bulk junction of the pre-charge MOS which can switch the high impedance node voltage from '1' to '0' or, worse, cause *latch-up*.

## 3.2 Domino design techniques

An evolution of dynamic logic is the Domino one, which was first described by [3]. Figure 3.2 shows the schematic of a single stage Domino circuit.

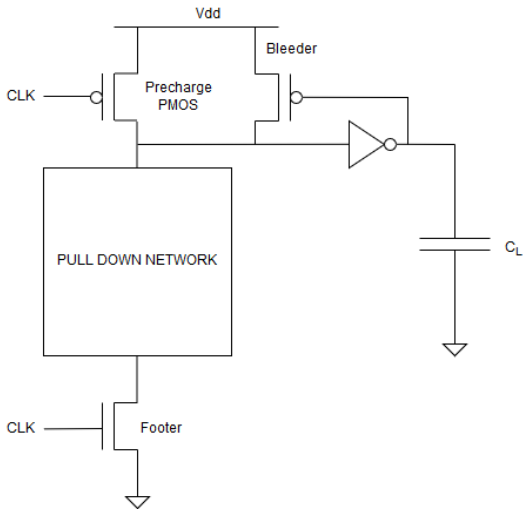



Figure 3.2: General schematic of domino logic circuit. The bleeder transistor is optional but highly recommended, in order to avoid information loss due to leakage currents.

The main advantage is that Domino circuits can be placed in multiple stages with the same topology and can be pipelined with multiphase clocking system [4] instead of using latches. At architectural level, the adoption of multiphase clocking prevent the use High performance are reached at the expense of higher power consumption due to the more clock lines needed.

## 3.3 Dual Rail Domino Logic

A particularly and interisting case of Domino logic is the Dual Rail Domino Logic (DRDL) implementation in order to make low power adders [5]. An example of Dual rail Domino circuit is reported in figure 3.3.

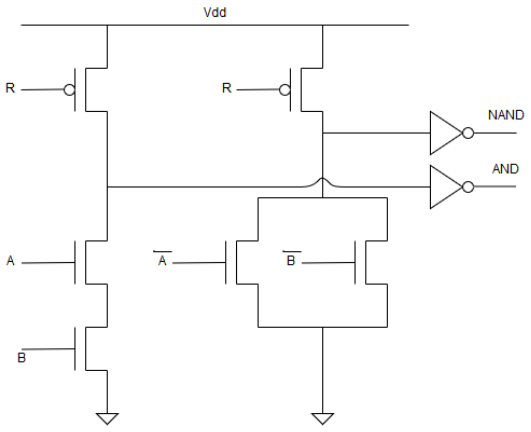



Figure 3.3: Dual Rail Domino logic implementation of a NAND circuit. The main advantage is to have inverting and non inverting function, at the expense of more area occupation.

In this implementation, circuit works in near threshold in order to reduce power consumption and uses an asynchronous pipeline. This approach reduces the energy-delay product, defined as:

$$EDP = E_{op} \times \tau_{delay} \tag{3.4}$$

where $E_{op}$ is the energy drained by the device during the SPICE simulation and $\tau_{delay}$. Although there is an improvement in energy consumption, performance and area become worse than a synchronous pipelined static CMOS implementation. Energy consumption, number of transistors and delay of [5] are reported in table 3.1. The worst case is used for synchronous adder and average case for the dual rail Domino logic one.

| | Transistors | Delay $\mu s$ | Energy $fJ$ |
|---|---|---|---|
| DRDL | 1460 | 62.6 | 499 |
| CMOS | 764 | 59.3 | 871 |

Table 3.1: Comparison between DRDL and static CMOS 8-bit adder.

As it can be seen, energy consumption is improved with respect to CMOS at the cost of almost double area and a little worse delay.

## 3.4 Technology scaling comparison

An important issue is to find whether technology scaling improve significantly the Domino circuits power consumption with respect to static CMOS logic. Two interesting works are found are compared in table .

| | Power | Pwr diff. with the previous gate |
|---|---|---|
| 32 nm FinFET static NAND2 in [6] | 440 nW | 0 % |
| 25 nm FinFET Domino AND2 in [7] | 1.6 $\mu W$ | +264 % |
| 32 nm FinFET static NOR2 [6] | 440 nW | 0 % |
| 25 nm FinFET Domino OR2 [7] | 4 $\mu W$ | +810 % |

Despite of the use of a better technology, the Domino logic suffers for higher power consumption, compared with the static CMOS logic.

## 3.5 Final Considerations

Dynamic logic based circuits are a good alternative if area and performance are critical aspects of the design. Multistage Domino circuits can be also pipelined with a multiphase clock approach, so no more latches are needed and there is more area saving. These aspects are balanced by a higher power consumption, which can be far higher than the static logic. Scaling does not improve the power consumption issue, so dynamic logic is generally not suitable for low power implementations, which is the current tendency. The Dual Rail Domino logic can be an interesting choice in case of asynchronous pipeline designs, since energy consumption is better than synchronous static implementations, although they occupy more area and have a performance similar to the static logic.

# Bibliography


[1] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE*, 1964.

[2] J. Rabaey, A. Chandrakasan, and B. Nikolic, "Circuiti integrati digitali, l'ottica del progettista," *Pearson*, 2005.

[3] R. Krambeck, C. Lee, and H. Law, "High-speed compact circuits with cmos," *IEEE*, 1982.

[4] S. Verma, A. Angeline, and K. Bhaaskaran, "Multiphase pipelining in domino logic alu," *IEEE*, 2017.

[5] T. Maruyama, M. Hamada, and T. Kuroda, "Comparative performance analysis of dual-rail domino logic and cmos logic under near threshold operation," *IEEE*, 2018.

[6] S. Nalamwar and S. Bhosale, "Design of low power logic gates by using 32nm and 16nm finfet technology," *IEEE*, 2015.

[7] S. Rasouli, H. Koike, and K. Banerjee, "High-speed low-power finfet based domino logic," *IEEE*, 2009.